QQ 浏览器 2021AI 算法大赛

队伍：Roger_ChallengeHub

Update: 2021 年 11 月 5 日

1 团队介绍

- Roger，浙江大学博士研究生，曾获得 4 次 CCF 大赛 TOP10、微信大数据挑战赛二等奖。
- Starry，vivo 算法工程师，曾获得微信大数据挑战赛 Rank3 等。

2 初赛历程

2.1 赛题回顾

本竞赛任务是超参数优化问题，参赛者需要提交超参数生成的算法代码，算法每一轮需要提供一组超参数，裁判程序会返回超参数对应的 reward。参赛者的算法要求在限定的迭代轮次和时间内，找到 reward 尽可能大的一组超参数，最终按照找到的最大 reward 来计算排名。

在每个独立的独立任务中，程序会调用 20 次评估算法 (我们称之为 20 次迭代，ITER)，每次调用要求返回 5 组参数组 (Suggestion)。根据返回的 Suggestion 评估程序会进行评估，并返回每一个 Suggestion 的得分，用于下一组寻参，直到完成 20 次迭代 (ITER)。在这 20 次迭代 (ITER) 调用中，我们把获得最高分的参数作为本次独立任务的得分。一个测试集的得分是多个独立任务的平均分，而参数程序的最终得分是 20 组测试集 (TestX) 的平均得分。

2.2 技术方案

对于这样一个黑盒优化问题，我们的思考是如何搜得又快又好，最后的方案将优化问题拆解为三个阶段：

- 初始化：希望尽可能扩宽视野
- 粗搜：借用机器学习算法进行解的快速搜索
- 精搜：在已有最好解的基础上进行寻优

2.2.1 初始化

为了使得拿到尽可能优的初始解服务后续的贝叶斯优化，这里采用**拉丁超立方抽样**算法进行初始解的生成。

拉丁超立方抽样（英语：Latin hypercube sampling，缩写 LHS）是一种从多元参数分布中近似随机抽样的方法，属于分层抽样技术，常用于计算机实验或蒙特卡洛积分等。

在统计抽样中，拉丁方阵是指每行、每列仅包含一个样本的方阵。拉丁超立方则是拉丁方阵在多维中的推广，每个与轴垂直的超平面最多含有一个样本。

假设有 N 个变量（维度），可以将每个变量分为 M 个概率相同的区间。此时，可以选取 M 个满足拉丁超立方条件的样本点。需要注意的是，拉丁超立方抽样要求每个变量的分区数量 M 相同。不过，该方法并不要求当变量增加时样本数 M 同样增加。

2.2.2 粗搜

粗搜阶段的技术选型为贝叶斯优化，主要改进借鉴了 HEBO[1] 算法的思想：

- 输入变换，即采用 input warping GP，为了克服不平稳函数的问题

$$\Phi_\theta(x;a,b)_i = 1-(1-x_i^{a_i})^{(b_i-1)} \tag{1}$$

- 输出变换，即希望变换后的目标函数值尽可能服从高斯分布

$$y_i^{(\lambda)} = \begin{cases} \frac{(y_i+\lambda_2)^{\lambda_1}-1}{\lambda_1} & \text{if } \lambda_1 \neq 0 \\ \ln(y_i+\lambda_2) & \text{if } \lambda_1 = 0 \end{cases} \tag{2}$$

- 采集函数的设计，多目标优化，考虑不同采集函数可能适用于不同场景，希望生成的解更具有泛化能力

$$\min_{x \in \mathcal{X}} \left(-\alpha_{\text{q-EI}}(\boldsymbol{x} \mid \mathcal{D}_i), -\alpha_{\text{q-PI}}(\boldsymbol{x} \mid \mathcal{D}_i), \alpha_{\text{q-UCB}}(\boldsymbol{x} \mid \mathcal{D}_i)\right) \tag{3}$$

- 多目标优化的求解，为了使得 5 次 suggestion 的差异性尽可能大，我们没有直接采用多目标优化得到帕累托边界再来选解，而是直接设置差异化的多目标权重，尽可能使得解的差异性更大。
- 模型预测均值加噪，增强模型预测的泛化能力，也避免过拟合数据，陷入局部最优

$$m(\cdot) = m(\cdot) + rand() \cdot \sigma_{noise}^2 \tag{4}$$

- 探索和利用，在贝叶斯优化中需要去权衡探索和利用，从而避免陷入局部最优，这里针对探索因子 ξ 进行了若干种设计，比如随机数，周期变化。

2.2.3 精搜

借鉴 Squirrel[2] 的思想，我们希望在搜索的最后阶段更加**贪婪**，力求寻得更优，这里尝试过使用差分进化算法、以及粒子群算法。最后根据效果选择了粒子群（PSO）算法作为第三阶段的算法。

PSO 算法是基于群体的，根据对环境的适应度将群体中的个体移动到好的区域。然而它不对个体使用演化算子，而是将每个个体看作是 D 维搜索空间中的一个没有体积的微粒（点），在搜索空间中以一定的速度飞行，这个速度根据它本身的飞行经验和同伴的飞行经验来动态调整。第 i 个微粒表示为 $X_i = (x_{i1}, x_{i2}, , x_{iD})$ ，它经历过的最好位置（有最好的适应值）记为 $Pi = (p_{i1}, p_{i2}, , p_{iD})$，也称为 pbest。在群体所有微粒经历过的最好位置的索引号用符号 g 表示，即 Pg，也称为 gbest 。微粒 i 的速度用 $Vi = (v_{i1}, v_{i2}, , v_{iD})$ 表示。对每一代，它的第 $d+1$ 维（$1 \leq d+1 \leq D$）

根据如下方程进行变化：

$$v_{id+1} = w \cdot v_{id} + c1 \cdot rand() \cdot (p_{id} - x_{id}) + c2 \cdot Rand() \cdot (p_{gd} - x_{id}) \quad (5)$$

$$x_{id+1} = x_{id} + v_{id+1} \quad (6)$$

其中 w 为惯性权重（inertia weight），$c1$ 和 $c2$ 为加速常数（acceleration constants），rand() 和 Rand() 为两个在 $[0, 1]$ 范围里变化的随机值。

3 决赛历程

3.1 赛题回顾

在初赛题目中每一组超参数对应一个确定性的 reward 值，然而在实际的业务场景中由于 reward 的计算成本很高，为了更高效地搜索超参数空间，往往需要在过程中分多次来采集计算 reward 值，以便能更快地跳过一些较差的点。

因此，在决赛题目中参赛者需要实现一个早停（Early Stop）的接口：is_early_stop，在每一次迭代中会调用该接口，用于判断某组超参数是否需要停止继续迭代，这样 reward 较差的参数组可以提前结束，加快超参数搜索效率。

- 初赛题和决赛题的数据集是一样的，初赛数据集中每组超参数的 reward 值等于决赛数据集中对应超参数第 14 次迭代的 reward 均值，可以理解为实际上初赛题是决赛题的简化版本
- 决赛中每组参数在每次迭代都会有新的 reward 均值，每组参数的 reward 均值在前期（1-13 iteration）可能不稳定，随着 iteration 逐步增多而逐步趋于稳定，直到第 14 个迭代的 reward 才是最终确定可信的
- 每组参数我们采用第 14 次 iteration 的 reward 均值作为这组参数的最终 reward 均值（简称：最终 reward 均值）。我们在获取所有运行够 14 次 iteration 的参数组中，取最终 reward 均值的最高分作为本次 repeat 执行的得分，如果参数运行不够 14 次 iteration，那这组参数的 reward 是不会采纳计分

3.2 技术方案

根据我们队伍的分析，决赛想要取得高分需要攻克的问题主要在两个：

- 早停的设计，尽可能获得更多的数据点
- 早停后数据的利用，早停得到的数据点其实是有偏的，如果利用有偏数据进行建模

参考了 MFES-HB[3] 算法，我们尝试进行 multi-fidelity bayesian optimization，不过并没有取得更好的效果。

最终由于时间的原因没有再对新方法进行探索，转而进行初赛方案的优化。由于不考虑早停后，决赛只有 50 次 suggestion 的机会，因此我们将原来的三阶段算法改为两阶段算法，即只包含初始化和贝叶斯优化。

在决赛中主要的提分点在于对贝叶斯优化中**探索和利用**的探究，初赛时我们没有考虑目标函数的数量级，采用随机的方式对贝叶斯优化中探索因子进行设置，虽然能够使得推荐的参数有效避免局部最优，但是缺点在于收敛太慢。因此我们决赛的主要思想是，每一次迭代都希望比之前最好的结果有一定比例的提升，基于此设计了一个周期变换的探索因子

```
if len(suggestions_history) % 5 == 0:
  gain = 4
elif len(suggestions_history) % 4 == 0:
  gain = 3
elif len(suggestions_history) % 3 == 0:
  gain = 2
elif len(suggestions_history) % 2 == 0:
  gain = 1
else:
  gain = 0
if max(y) > 0:
  x_i = abs(max(y)) * 0.05 * gain + 1e-4
else:
  x_i = abs(min(y)) * 0.05 * gain + 1e-4
```

决赛算法的其他部分保持和初赛一致。

4 核心代码

核心代码按照算法流程进行给出：

初始化，LHS 采样后根据变量的最大最小值进行还原

```
l_bounds = []
u_bounds = []
for p_name, p_conf in sorted(self.parameters_config.items(), key=lambda x:
    x[0]):
  l_bounds.append(p_conf['double_min_value'])
  u_bounds.append(p_conf['double_max_value'])
  self.l_bounds = l_bounds
  self.u_bounds = u_bounds

samp = lhs(len(self.parameters_config), 10, criterion='maximin', iterations
    =100, random_state=2021)
self.samp = samp * (np.array(u_bounds) - np.array(l_bounds)) + np.array(l_
    bounds)
```

目标函数变换，使得尽可能接近高斯分布

```
if min(y_datas) <= 0:
  y = power_transform(y_datas.reshape(-1, 1) / np.std(y_datas), method='yeo
      -johnson')
else:
  y = power_transform(y_datas.reshape(-1, 1) / np.std(y_datas), method='box
      -cox')
  if np.std(y) < 0.5:
    y = power_transform(y / np.std(y), method='yeo-johnson')
if np.std(y) < 0.5:
  raise RuntimeError('Power transformation failed')
```

代理模型的设计：考虑了输入变换，以及输出的标准化

```
class GPyGP:
  def __init__(self):
    self.xscaler = MinMaxScaler((0, 1))
    self.yscaler = StandardScaler()
    self.num_epochs = 20
    self.verbose = False

  def fit_scaler(self, X, y):
    self.xscaler.fit(X)
    self.yscaler.fit(y)
```

```
def trans(self, X, y=None):
  X = self.xscaler.transform(X)
  if y is not None:
    y = self.yscaler.transform(y)
    return X, y
  return X

def fit(self, X, y):
  self.fit_scaler(X, y)
  X, y = self.trans(X, y)

  k1 = GPy.kern.Linear(X.shape[1], ARD=False)
  k2 = GPy.kern.Matern32(X.shape[1], ARD=True)
  k2.lengthscale = np.std(X, axis=0).clip(min=0.02)
  k2.variance = 0.5
  k2.variance.set_prior(GPy.priors.Gamma(0.5, 1), warning=False)
  kern = k1 + k2
  xmin = np.zeros(X.shape[1])
  xmax = np.ones(X.shape[1])
  warp_f = GPy.util.input_warping_functions.KumarWarping(X, Xmin=xmin,
      Xmax=xmax)
  self.gp = GPy.models.InputWarpedGP(X, y, kern, warping_function=warp_f)
  self.gp.likelihood.variance.set_prior(GPy.priors.LogGaussian(-4.63,
      0.5), warning=False)
  # self.gp.optimize_restarts(max_iters=self.num_epochs, verbose=self.
      verbose, num_restarts=5, robust=True)
  self.gp.optimize(max_iters=self.num_epochs)


def predict(self, X):
  X = self.trans(X)
  py, ps2 = self.gp.predict(X)
  mu = self.yscaler.inverse_transform(py.reshape(-1, 1))
  var = np.clip(self.yscaler.var_ * ps2.reshape(-1, 1), 1e-6, np.inf)
  return mu, var

@property
def noise(self):
  var_normalized = self.gp.likelihood.variance[0]
  noise = (var_normalized * self.yscaler.var_).reshape(1, )
  return noise
```

采集函数的设计，这边不采用 HEBO 中直接对多目标优化进行求解，而

是转化为多目标加权的形式进行求解

```
def _MACE(x_x, g_p, y_max, x_i, kappa, ratio):
  with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    py, ps2 = g_p.predict(x_x)

  py = py.reshape(-1, )
  ps2 = ps2.reshape(-1, )  # var or std
  noise = np.sqrt(2.0 * g_p.noise)
  ps = np.sqrt(ps2)
  ucb = (py + noise * np.random.randn(*py.shape)) + kappa * ps

  a_a = (py + noise * np.random.randn(*py.shape) - y_max - x_i)
  z_z = a_a / ps
  EI = a_a * norm.cdf(z_z) + ps * norm.pdf(z_z)

  z_z = (py + noise * np.random.randn(*py.shape) - y_max - x_i) / ps
  PI = norm.cdf(z_z)

  DI = (py + noise * np.random.randn(*py.shape) - y_max - x_i) / a_a
  DI = a_a * norm.cdf(DI) + ps * norm.pdf(DI)

  return ratio[0]*ucb + ratio[1]*EI + ratio[2]*PI + ratio[3]*DI
```

多目标优化的权重设计，使得权重的差异性大一些，有利于解的多样性

```
Ratio = lhs(4, n_suggestions)
ratio = Ratio[index, :]
ratio /= sum(ratio)
```

探索和利用的权衡，周期性从大到小变换，即由全局优化向局部优化不断切换

```
if len(suggestions_history) % 5 == 0:
  gain = 4
elif len(suggestions_history) % 4 == 0:
  gain = 3
elif len(suggestions_history) % 3 == 0:
  gain = 2
elif len(suggestions_history) % 2 == 0:
  gain = 1
else:
  gain = 0
if max(y) > 0:
  x_i = abs(max(y)) * 0.05 * gain + 1e-4
```

```
else:
  x_i = abs(min(y)) * 0.05 * gain + 1e-4
```

解的去重，如果生成的解有重复，丢弃重复解并用 LHS 采样生成替代解

```
def check_unique(self, X, rec):
  X = pd.DataFrame(X)
  rec = pd.DataFrame(rec)
  return (~pd.concat([X, rec], axis = 0).duplicated().tail(rec.shape[0]).
      values).tolist()

suggestions = suggestions[self.check_unique(x_datas, suggestions)].reshape
    (-1, len(self.parameters_config))
if suggestions.shape[0] < n_suggestions:
  samp = self.lhs_sample(n_suggestions - suggestions.shape[0])
  suggestions = np.vstack([suggestions, samp])
```

粒子群算法

```
class PSO:

  def __init__(self, l_bounds, u_bounds, dimensions):
    self.l_bounds = l_bounds
    self.u_bounds = u_bounds
    self.dimensions = dimensions
    self.pop_size = 5
    self.iteration = 0
    self.x = np.random.uniform(low=0.0, high=1.0, size=(self.pop_size, self
        .dimensions))
    self.v = np.random.randn(self.pop_size, dimensions)
    self.y = self.x.copy()
    self.pg = self.x[self.pop_size - 1, :].copy()
    self.f_pg = None
    self.p = np.zeros(self.pop_size)
    self.fx = np.zeros(self.pop_size)

    self.w = 0.5
    self.c1 = 2
    self.c2 = 2

  def parse_suggestions_history(self, suggestions_history):
    suggestions_history_use = copy.deepcopy(suggestions_history)
    x_datas = [[item[1] for item in sorted(suggestion[0].items(), key=
        lambda x: x[0])]
    for suggestion in suggestions_history_use]
    y_datas = [suggestion[1] for suggestion in suggestions_history_use]
```

```
    x_datas = np.array(x_datas)
    y_datas = np.array(y_datas)

    return x_datas, y_datas

  def set_init(self, suggestions_history):
    # 设置粒子群的初始解
    x_datas, y_datas = self.parse_suggestions_history(suggestions_history)
    top_idx = np.argsort(y_datas)[-5:]
    self.x = x_datas[top_idx].astype(np.float)
    self.fx = -y_datas[top_idx]
    self.p = -y_datas[top_idx]
    self.y = self.x.copy()
    self.pg = self.x[self.pop_size - 1, :].copy()
    self.f_pg = self.p[self.pop_size - 1].copy()

    self.iteration = 1

  def boundary_check(self, vector, fix_type='random'):
    # 确保上下界约束

    violations = np.where((vector > 1) | (vector < 0))[0]
    if len(violations) == 0:
      return vector
    if fix_type == 'random':
      vector[violations] = np.random.uniform(low=0.0, high=1.0, size=len(
          violations))
    else:
      vector[violations] = np.clip(vector[violations], a_min=0, a_max=1)
    return vector

  def suggest(self, suggestions_history):
    if self.iteration == 0:
      self.set_init(suggestions_history)
    else:
      self.observe(suggestions_history)

    trials = []
    for i in range(self.pop_size):
      self.v[i, :] = self.w * self.v[i, :] + self.c1 * np.random.rand() * (
      self.y[i, :] - self.x[i, :]) + self.c2 * np.random.rand() * (self.pg
          - self.x[i, :])
      self.x[i, :] = self.x[i, :] + self.v[i, :]
      self.x[i, :] = np.clip(self.x[i, :], self.l_bounds, self.u_bounds)
```

```
      trials.append(self.x[i, :])
    trials = np.array(trials)
    return trials

  def observe(self, suggestions_history):
    # 加入观测数据
    x_datas, y_datas = self.parse_suggestions_history(suggestions_history)
    y = y_datas[-5:]
    for i in range(self.pop_size):
      self.fx[i] = -y[i]
      if -y[i] < self.p[i]:
        self.p[i] = -y[i]
        self.y[i, :] = self.x[i, :].copy()
      if self.p[i] < self.f_pg:
        self.pg = self.y[i, :].copy()
        self.f_pg = self.p[i]
    self.iteration += 1
```

5 总结思考

QQ 浏览器的超参优化是一个十分好的题目，具有很强的现实意义，能够解决深度/机器模型中调参耗时的问题。我们在该比赛中取得了初赛第四，决赛第九的成绩。与取得的成绩相比，更重要的是在该比赛中学习到了很多东西。下面我将对我们这次比赛做一个总结。

初赛是一个十分典型的黑盒优化问题，每次迭代我们需要返回 5 个参数组合，然后线上计算 5 个参数组合的得分，我们根据已经有的参数组合和相应的得分生成新的参数组合，反复执行 20 次，总共生成 100 个参数组合。

官方给了 2 个 baseline，一个随机生成，一个是简单地根据贝叶斯优化进行调参。在仔细查看了官方的 baseline 之后，我们试着调整贝叶斯模型，主要的操作有：1：保证生成的解不重复 2：初始化的次数变多 (等到有 20 或者更多随机初始化的结果的时候再调用贝叶斯模型求解) 最后线上的得分是 0.40+

尝试过之后，我们决定采用启发式或者元启发式的方法生成解，最后采用了粒子群算法。具体的方法为 (保留上一步对贝叶斯的改进操作)：

1：初始化一次次数后，选取最大的 5 个参数组合，组成一个粒子群。2：执行简单的粒子群算法除此之后进行的优化操作有：初始化使用超拉丁采样此外我们还采取了双粒子群循环迭代的方法两种方法的最后得分在 0.5 – 0.6+

此时我们无意中发现了一个之前的黑盒优化的比赛，从中学习到了不少的东西，最后我们初赛的模型是这样的：以贝叶斯优化为基础，其余的优化操作有：1：coxbox 对目标函数 y 进行空间变换，使得目标函数比较分散变得集中 2：对参数组合进行空间缩放，将其缩放至 0 – 1 内，并且继续将 y 进行 StandardScaler。3：贝叶斯中 acquisition function 的优化，结合了 EI，UCB 等常见的函数组合成 MACE 函数，进行多目标的优化。4：贝叶斯核函数的选取，选取了线性核函数和非线性核函数的组合核函数。这是我们初赛最后的基础模型，除此之外，我们考虑到最后局部优化的问题，引入了粒子群和差分进化的方法在最后几轮进行局部优化。

我们尝试了以随机森林为优化模型的贝叶斯，发现效果并不如意 (也有可能是参数或者某种姿势不对)，该模型时间长，并且效果没用我们上面的贝叶斯要好。除此之外：初赛的时候我们线下可以稳定取得 0.99 以上的成绩，但是由于初赛的线程设置原因 (该问题在我们队伍的强烈反馈后在决赛中被解决了)，我们的模型在初赛中经常超时，尝试了很多种方法在线上都没有解决后，我们无奈只能减少贝叶斯的迭代次数，再线上实际判断运行时间，在即将超时的时候调用差分进化或者粒子群快速生成解。这也一定程度上影响了我们初赛的成绩。

下面就是决赛的思路了，决赛的题目相较于初赛的题目发生了很大的改变，具体表现为考虑到实际情况中每次验证一个参数的时间很长很长，但是我们可以得到参数组合的中间结果，可以辅助判断该参数的效果，如果判断觉得该参数不行的话，可以及时停止该参数的验证工作，将资源留给别的参数的验证。最后总结就是如果不采用及时停止的策略的话，我们可以迭代 10 次，总共可以生成 50 个解，比初赛少了一半。

首先我们验证一下我们初赛的模型在不止停的情况下的效果，无奈发现分数只有 0.5+。对于这种问题来说，难点主要有：1：采用停止的方

法，2：采用了止停之后虽然可以生成更多的解，但是由于训练数据会变少 (提前停止的数据无法直接放到贝叶斯模型中训练)。决赛我们表现的并不好，因为这两个难点我们都没有很好的解决，特别是第二点。在这个比赛期间我们做的操作有：一：止停操作：最近 5 次的最大值小于当前的最优值；当前的迭代的上界小于目前的最优值；每次迭代生成的值方差过大；根据我们线下观测的结果，通过这三种止停方法过滤的参数都是可以过滤的，但是由于过滤了过多的参数导致我们贝叶斯无法训练充分，生成的解并不太好。最终线下的结果也比较低。二：我们后来了解到了 MFES-HB 文章 [3]，采取了他们将止停的数据也进行训练的思想，但是最终的结果没有取得预期，可能是止停数据和模型个数没有调整好。三：使用 MFES-HB 将上界下界分别训练模型，将其结合在一起，效果也是降低了。很遗憾的我们采取的止停的策略和相应的方法都没有取得很好的结果，因此，我们最后只能继续优化初赛的模型，主要是针对性地调整函数，比如 acquisition function 中的 ξ，κ 等，最终在线上取得了 0.64+ 的结果。

总体来说，这是一个十分棒的比赛，通过该比赛让我们接触到一个之前只是略微了解的领域，并且学习了一些该领域最近的研究情况，对于我们自身的知识有了很大的增长，感谢举办方，接下来希望可以学习前排的模型，了解到自己在比赛中没有注意或者没有接触到的知识。

6 参考文献

1. Cowen-Rivers A I, Lyu W, Wang Z, et al. Hebo: Heteroscedastic evolutionary bayesian optimisation[J]. arXiv e-prints, 2020: arXiv: 2012.03826.
2. Awad N, Shala G, Deng D, et al. Squirrel: A Switching Hyperparameter Optimizer[J]. arXiv preprint arXiv:2012.08180, 2020.
3. Li Y, Shen Y, Jiang J, et al. MFES-HB: Efficient Hyperband with Multi-Fidelity Quality Measurements[J]. arXiv preprint arXiv:2012.03011, 2020.