XGBoost源码分析之单机多线程的实现 - 啦啦啦种太阳

原文 http://blog.csdn.net/chedan541300521/article/details/5489588 2017-02-07 09:11:59 311 ℃ 0 评论

通过阅读XGBoost源码，总结了几处自己认为比较重要的方面。如有错误，请指正：

1. 总体框架：

cli_main.cc 是程序的入口，main函数所在的文件。除了有main函数以外，还有训练参数的结构体。如模型保存路径，数据路径，迭代次数等。这个源码注释的很清楚，不再赘述。

通过调用main() -> CLIRunTask() -> CLITrain()，这里我们主要看函数CLITrain()的流程

CLITrain主要分了几步：

1.加载数据

2.初始化learner

3.调用learner->InitModel()、learner->Configure初始化Model，确定目标函数和模型，初始化目标函数结构体和模型

4.根据模型参数param.num_round迭代调用UpdateOneIter()来建树

2.learner

上一节，在learner中初始化目标函数和模型，主要赋值给了这两个变量。

/include/xgboost/learner.h

```
/*! \brief objective function */
std::unique_ptr<ObjFunction> obj_;
/*! \brief The gradient booster used by the model*/
std::unique_ptr<GradientBooster> gbm_;
```

ObjFunction是基类，定义了很多虚函数。类RegLossObj等都继承于此类，主要实现了根据不同的模型和loss，将一阶二阶导数计算出来。

以线性回归模型为例

src\objective\regression_obj.cc

定义平方损失函数：

```
// linear regression
struct LinearSquareLoss {
  static bst_float PredTransform(bst_float x) { return x; }
  static bool CheckLabel(bst_float x) { return true; }
  static bst_float FirstOrderGradient(bst_float predt, bst_float label) { return predt - label; }
  static bst_float SecondOrderGradient(bst_float predt, bst_float label) { return 1.0f; }
  static bst_float ProbToMargin(bst_float base_score) { return base_score; }
  static const char* LabelErrorMsg() { return ""; }
  static const char* DefaultEvalMetric() { return "rmse"; }
}
```

```
class RegLossObj : public ObjFunction{
...
void GetGradient(const std::vector<bst_float> &preds,
                 const MetaInfo &info,
                 int iter,
                 std::vector<bst_gpair> *out_gpair) override {
...
    out_gpair->resize(preds.size());
    // check if label in range
    bool label_correct = true;
    // start calculating gradient
    const omp_ulong ndata = http://blog.csdn.net/chedan541300521/article/details/static_cast(preds.size());
    #pragma omp parallel for schedule(static)
    for (omp_ulong i = 0; i < ndata; ++i) {
      bst_float p = Loss::PredTransform(preds[i]);
      bst_float w = info.GetWeight(i);
      if (info.labels[i] == 1.0f) w *= param_.scale_pos_weight;
      if (!Loss::CheckLabel(info.labels[i])) label_correct = false;
```

```
      out_gpair->at(i) = bst_gpair(Loss::FirstOrderGradient(p, info.labels[i]) * w,
                                   Loss::SecondOrderGradient(p, info.labels[i]) * w);

    }
...
}
```

可以发现在计算一阶导和二阶导的时候，采用了并行处理。以一条数据作为一个粒度。

GradientBooster是基类，有两个模型继承于此类。XGBoost中除了有Tree模型（GBTree），同时也实现了线性模型（GBLinear）。与梯度下降和牛顿法不同的是，在每次迭代的过程中，每个属性单独计算，采用类似于一维的牛顿法来更新一个属性。这里不是重点，有兴趣的同学可以看\src\gbm\gblinear.cc。

3.UpdateOneIter

这里可以看出每次迭代的操作，主要有：

```
  void UpdateOneIter(int iter, DMatrix* train) override {

    this->LazyInitDMatrix(train);
    this->PredictRaw(train, &preds_);                                  //获取上一轮预测值
    obj_->GetGradient(preds_, train->info(), iter, &gpair_);           //计算一阶导和二阶导
    gbm_->DoBoost(train, &gpair_, obj_.get());                         //建树
  }
```

在DoBoost 中确定了输出维度，调用BoostNewTrees构造森林。最后终于调用了update。在输出维度不为1时，调用BoostNewTrees是并行的。这里构造不止一棵树是为了一下一步随机森林做准备。因为每一次迭代，是分给了多个树完成，而不是一颗。DoBoost这部分代码可以在gbtree.cc中找到，这里不再赘述。

4.ColMaker

> 4.1 用于统计的结构体

ColMaker继承于TreeUpdater，是单机多线程建树的实现过程，也是本文重点记录的地方。

ColMaker提供两个函数：Init、Update，分别用于初始化和更新建树由上层调用，同时为了实现多线程，定义了三个结构体：ThreadEntry、NodeEntry、Builder。

```
class ColMaker: public TreeUpdater {
 public:
       void Init(const std::vector<std::pair<std::string, std::string> >& args) override
       void Update(const std::vector<bst_gpair> &gpair,DMatrix* dmat,const std::vector<RegTree*> &trees) override
        // training parameter
       TrainParam param;
        // data structure
       /*! \brief per thread x per node entry to store tmp data */
       struct ThreadEntr
       struct NodeEntry
        // actual builder that runs the algorithm
       struct Builder
}
```

首先看一下结构体ThreadEntry

```
struct ThreadEntry {
    /*! \brief statistics of data */
    TStats stats;
    /*! \brief extra statistics of data */
    TStats stats_extra;
    /*! \brief last feature value scanned */
    bst_float last_fvalue;
    /*! \brief first feature value scanned */
    bst_float first_fvalue;
    /*! \brief current best solution */
    SplitEntry best;
    // constructor
```

```
    explicit ThreadEntry(const TrainParam &m)
        : stats(param), stats_extra(param) {
    }
```

ThreadEntry结构体用于多线程中单个线程内的样本统计，一个线程拥有一个ThreadEntry 变量。

```
struct NodeEntry {
    /*! \brief statics for node entry */
    TStats stats;
    /*! \brief loss of this node, without split */
    bst_float root_gain;
    /*! \brief weight calculated related to current data */
    bst_float weight;
    /*! \brief current best solution */
    SplitEntry best;
    // constructor
    explicit NodeEntry(const TrainParam& param)
        : stats(param), root_gain(0.0f), weight(0.0f){
    }
```

NodeEntry 用于一个树节点，所拥有的样本的统计信息。一个节点在分列前，需要将自己的样本分给不同的线程进行处理，最后得到多个ThreadEntry 变量，汇总到每个节点自己的NodeEntry 变量中。详细过程如下。

4.2 建树总体流程

Update -> Builder.Update

```
struct Builder {
...
   public:
    // constructor
    explicit Builder(const TrainParam& param) : param(param), nthread(omp_get_max_threads()) {}
    // update one tree, growing
    virtual void Update(const std::vector<bst_gpair>& gpair,
                        DMatrix* p_fmat,
                        RegTree* p_tree) {
      this->InitData(gpair, *p_fmat, *p_tree);
      this->InitNewNode(qexpand_, gpair, *p_fmat, *p_tree);
      for (int depth = 0; depth < param.max_depth; ++depth) {
        this->FindSplit(depth, qexpand_, gpair, p_fmat, p_tree);
        this->ResetPosition(qexpand_, p_fmat, *p_tree);
        this->UpdateQueueExpand(*p_tree, &qexpand_);
        this->InitNewNode(qexpand_, gpair, *p_fmat, *p_tree);
        // if nothing left to be expand, break
        if (qexpand_.size() == 0) break;
      }
      // set all the rest expanding nodes to leaf
      for (size_t i = 0; i < qexpand_.size(); ++i) {
        const int nid = qexpand_[i];
        (*p_tree)[nid].set_leaf(snode[nid].weight * param.learning_rate);
      }
      // remember auxiliary statistics in the tree node
      for (int nid = 0; nid < p_tree->param.num_nodes; ++nid) {
        p_tree->stat(nid).loss_chg = snode[nid].best.loss_chg;
        p_tree->stat(nid).base_weight = snode[nid].weight;
        p_tree->stat(nid).sum_hess = static_cast<float>(snode[nid].stats.sum_hess);
        snode[nid].stats.SetLeafVec(param, p_tree->leafvec(nid));
      }
    }
...
}
```

上述代码的流程可以描述为：

初始化变量

初始化节点

For (depth 1：max_depth){

找分割点，更新变量p_tree，维护树的增长

重置样本所属节点，更新变量position，主要将样本从父节点分配到刚分裂出来的叶子节点中

将叶子预分裂的节点挑出来，更新变量qexpand_，这个变量是为了存储待处理的新节点

```
    初始化新节点
}
```

qexpand_中的节点设置为叶子节点
把队列里的节点信息加到树模型中

4.3 抽样

在XGBoost中，为了防止过拟合，有很多地方用到了抽样。其中一种是通过参数对样本进行抽样：

Builder -> Update - >InitData

```
  // mark subsample
        if (param.subsample < 1.0f) {
          std::bernoulli_distribution coin_flip(param.subsample);
          auto& rnd = common::GlobalRandom();
          for (size_t i = 0; i < rowset.size(); ++i) {
            const bst_uint ridx = rowset[i];
            if (gpair[ridx].hess < 0.0f) continue;
            if (!coin_flip(rnd)) position[ridx] = ~position[ridx];
          }
```

在XGBoost的设计中，position[ridx]如果小于0，则认为该样本被删除。由于loss是凸函数，二阶导必定大于0，所以如果小于0，可能在计算过程中出现溢出等异常出现，则不考虑该样本。所以如果不将样本加入到计算中，只需取反。在上述代码中，是通过伯努利分布生成随机数来进行抽样。

XGBoost还对模型的属性进行抽样：

Builder -> Update - > FindSplit

```
std::vector<bst_uint> feat_set = feat_index;
if (param.colsample_bylevel != 1.0f) {
        std::shuffle(feat_set.begin(), feat_set.end(), common::GlobalRandom());
        unsigned n = static_cast<unsigned>(param.colsample_bylevel * feat_index.size());
        feat_set.resize(n);
      }
```

利用std::shuffle随机排序取前n来实现列抽样

4.4 寻找分割点

Builder -> Update - > FindSplit

```
    // find splits at current level, do split per level
    inline void FindSplit(int depth,
                          const std::vector<int> &qexpand,
                          const std::vector<bst_gpair> &gpair,
                          DMatrix *p_fmat,
                          RegTree *p_tree) {
      ...
      dmlc::DataIter<ColBatch>* iter = p_fmat->ColIterator(feat_set);
      while (iter->Next()) {
        this->UpdateSolution(iter->Value(), gpair, *p_fmat);
      }
      // after this each thread's stemp will get the best candidates, aggregate results
      this->SyncBestSolution(qexpand);
      // get the best result, we can synchronize the solution
      for (size_t i = 0; i < qexpand.size(); ++i) {
        const int nid = qexpand[i];
        NodeEntry &e = snode[nid];
        // now we know the solution in snode[nid], set split
        if (e.best.loss_chg > rt_eps) {
          p_tree->AddChilds(nid);
          (*p_tree)[nid].set_split(e.best.split_index(), e.best.split_value, e.best.default_left());
          // mark right child as 0, to indicate fresh leaf
          (*p_tree)[(*p_tree)[nid].cleft()].set_leaf(0.0f, 0);
          (*p_tree)[(*p_tree)[nid].cright()].set_leaf(0.0f, 0);
        } else {
          (*p_tree)[nid].set_leaf(e.weight * param.learning_rate);
        }
      }
    }
```

XGBoost把样本转换成了以列为基础的迭代器。这一部分设计等我整理一下再传上来。代码将数据以列为基础通过调用UpdateSolution计算分割点。为什么要这么做呢，这里已经能想到了，每一列至少交给了一个线程去处理。而SyncBestSolution则是将线程处理后的数据进行汇总的过程。

而进入UpdateSolution后，如果数据量不大则按属性进行多线程处理了，每个线程通过调用EnumerateSplit方法按实现。但是当数据量比较大，或者通过参数设置parallel_option，可以在此基础上进行更细的多线程拆分。

通过函数ParallelFindSplit实现：

Builder -> Update - > FindSplit -> ParallelFindSplit

```
// parallel find the best split of current fid
// this function does not support nested functions
inline void ParallelFindSplit(const ColBatch::Inst &col,
                              bst_uint fid,
                              const DMatrix &fmat,
                              const std::vector<bst_gpair> &gpair) {
  // TODO(tqchen): double check stats order.
  const MetaInfo& info = fmat.info();
  const bool ind = col.length != 0 && col.data[0].fvalue =http://blog.csdn.net/chedan541300521/article/details/= col.data[col.length - 1].fvalue;
  bool need_forward = param.need_forward_search(fmat.GetColDensity(fid), ind);
  bool need_backward = param.need_backward_search(fmat.GetColDensity(fid), ind);
  const std::vector &qexpand = qexpand_;
  #pragma omp parallel
  {
    const int tid = omp_get_thread_num();
    std::vector &temp = stemp[tid];
    // cleanup temp statistics
    for (size_t j = 0; j < qexpand.size(); ++j) {
      temp[qexpand[j]].stats.Clear();
    }
    bst_uint step = (col.length + this->nthread - 1) / this->nthread;
    bst_uint end = std::min(col.length, step * (tid + 1));
    for (bst_uint i = tid * step; i < end; ++i) {
      const bst_uint ridx = col[i].index;
      const int nid = position[ridx];
      if (nid < 0) continue;
      const bst_float fvalue = col[i].fvalue;
      if (temp[nid].stats.Empty()) {
        temp[nid].first_fvalue = fvalue;
      }
      temp[nid].stats.Add(gpair, info, ridx);
      temp[nid].last_fvalue = fvalue;
    }
  }
  // start collecting the partial sum statistics
  bst_omp_uint nnode = static_cast(qexpand.size());
  #pragma omp parallel for schedule(static)
  for (bst_omp_uint j = 0; j < nnode; ++j) {
    const int nid = qexpand[j];
    TStats sum(param), tmp(param), c(param);
    for (int tid = 0; tid < this->nthread; ++tid) {
      tmp = stemp[tid][nid].stats;
      stemp[tid][nid].stats = sum;
      sum.Add(tmp);
      if (tid != 0) {
        std::swap(stemp[tid - 1][nid].last_fvalue, stemp[tid][nid].first_fvalue);
      }
    }
    for (int tid = 0; tid < this->nthread; ++tid) {
      stemp[tid][nid].stats_extra = sum;
      ThreadEntry &e = stemp[tid][nid];
      bst_float fsplit;
      if (tid != 0) {
        if (stemp[tid - 1][nid].last_fvalue != e.first_fvalue) {
          fsplit = (stemp[tid - 1][nid].last_fvalue + e.first_fvalue) * 0.5f;
        } else {
          continue;
        }
      } else {
        fsplit = e.first_fvalue - rt_eps;
      }
      if (need_forward && tid != 0) {
        c.SetSubstract(snode[nid].stats, e.stats);
        if (c.sum_hess >= param.min_child_weight &&
            e.stats.sum_hess >= param.min_child_weight) {
```

```
          bst_float loss_chg = static_cast(
              constraints_[nid].CalcSplitGain(param, fid, e.stats, c) - snode[nid].root_gain);
          e.best.Update(loss_chg, fid, fsplit, false);
        }
      }
      if (need_backward) {
        tmp.SetSubstract(sum, e.stats);
        c.SetSubstract(snode[nid].stats, tmp);
        if (c.sum_hess >= param.min_child_weight &&
            tmp.sum_hess >= param.min_child_weight) {
          bst_float loss_chg = static_cast(
              constraints_[nid].CalcSplitGain(param, fid, tmp, c) - snode[nid].root_gain);
          e.best.Update(loss_chg, fid, fsplit, true);
        }
      }
    }
    if (need_backward) {
      tmp = sum;
      ThreadEntry &e = stemp[this->nthread-1][nid];
      c.SetSubstract(snode[nid].stats, tmp);
      if (c.sum_hess >= param.min_child_weight &&
          tmp.sum_hess >= param.min_child_weight) {
        bst_float loss_chg = static_cast(
            constraints_[nid].CalcSplitGain(param, fid, tmp, c) - snode[nid].root_gain);
        e.best.Update(loss_chg, fid, e.last_fvalue + rt_eps, true);
      }
    }
  }
  // rescan, generate candidate split
  #pragma omp parallel
  {
    TStats c(param), cright(param);
    const int tid = omp_get_thread_num();
    std::vector &temp = stemp[tid];
    bst_uint step = (col.length + this->nthread - 1) / this->nthread;
    bst_uint end = std::min(col.length, step * (tid + 1));
    for (bst_uint i = tid * step; i < end; ++i) {
      const bst_uint ridx = col[i].index;
      const int nid = position[ridx];
      if (nid < 0) continue;
      const bst_float fvalue = col[i].fvalue;
      // get the statistics of nid
      ThreadEntry &e = temp[nid];
      if (e.stats.Empty()) {
        e.stats.Add(gpair, info, ridx);
        e.first_fvalue = fvalue;
      } else {
        // forward default right
        if (fvalue != e.first_fvalue) {
          if (need_forward) {
            c.SetSubstract(snode[nid].stats, e.stats);
            if (c.sum_hess >= param.min_child_weight &&
                e.stats.sum_hess >= param.min_child_weight) {
              bst_float loss_chg = static_cast(
                  constraints_[nid].CalcSplitGain(param, fid, e.stats, c) -
                  snode[nid].root_gain);
              e.best.Update(loss_chg, fid, (fvalue + e.first_fvalue) * 0.5f, false);
            }
          }
          if (need_backward) {
            cright.SetSubstract(e.stats_extra, e.stats);
            c.SetSubstract(snode[nid].stats, cright);
            if (c.sum_hess >= param.min_child_weight &&
                cright.sum_hess >= param.min_child_weight) {
              bst_float loss_chg = static_cast(
                  constraints_[nid].CalcSplitGain(param, fid, c, cright) -
                  snode[nid].root_gain);
              e.best.Update(loss_chg, fid, (fvalue + e.first_fvalue) * 0.5f, true);
            }
          }
        }
        e.stats.Add(gpair, info, ridx);
        e.first_fvalue = fvalue;
      }
    }
  }
}
```

上述代码将某一属性的样本空间的数据进行分块，块大小为(col.length + this->nthread - 1) / this->nthread ），这一步是并行的，每一块是一个线程，每个线程统计属于自己的样本的每个节点的一阶导和二阶导数的和，还需要统计边缘值（first_fvalue、last_fvalue ）

按照块边缘，初始化线程变量stemp[tid][nid].best，这一步是节点并行，因为需要用到两个相邻线程所用有的数据块的边缘值：stemp[tid - 1][nid].last_fvalue + e.first_fvalue) * 0.5f，所以无法按照块并行

第三步则是按照数据块并行，每个线程处理自己的数据块来更新stemp[tid][nid].best，这样每个线程都有一个根据自己样本计算出来的属于自己的最优分割点。

最后再根据上文中提到的SyncBestSolution来汇总所有线程，最后进行树的增长、qexpand更新、position更新等操作。

总结：

本文梳理了XGBoost 在训练过程的源码的流程，通过阅读，了解了XGBoost的运行机制、设计模式为以后自己编写代码提供了宝贵经验。

上一篇：【mapreduce】hadoop2.x—mapreduce实战和 下一篇：实践中 XunSearch（讯搜）更新索引方案对比