Deep Compression阅读理解及Caffe源码修改

作者：may0324 (http://my.csdn.net/may0324)

最近又转战CNN模型压缩了。。。（我真是一年换N个坑的节奏），阅读了HanSong的15年16年几篇比较有名的论文，启发很大，这篇主要讲一下Deep Compression那篇论文，因为需要修改caffe源码，但网上没有人po过，这里做个第一个吃螃蟹的人，记录一下对这篇论文的理解和源码修改过程，方便日后追本溯源，同时如果有什么纰漏也欢迎指正，互相交流学习。

这里就从Why-How-What三方面来讲讲这篇文章。

Why

首先讲讲为什么CNN模型压缩刻不容缓，我们可以看看这些有名的caffe模型大小:

1. LeNet-5 1.7MB
2. AlexNet 240MB
3. VGG-16 552MB

LeNet-5是一个简单的手写数字识别网络，AlexNet和VGG-16则用于图像分类，刷新了ImageNet竞赛的成绩，但是就其模型尺寸来说，根本无法移植到手机端App或嵌入式芯片当中，就算是想通过网络传输，较高的带宽占用率也让很多用户望尘莫及。另一方面，大尺寸的模型也对设备功耗和运行速度带来了巨大的挑战。随着深度学习的不断普及和caffe,tensorflow,torch等框架的成熟，促使越来越多的学者不用过多地去花费时间在代码开发上，而是可以毫无顾及地不断设计加深网络，不断扩充数据，不断刷新模型精度和尺寸，但这样的模型距离实用却仍是望其项背。

在这样的情形下，模型压缩则成为了亟待解决的问题，其实早期也有学者提出了一些压缩方法，比如weight prune(权值修剪)，权值矩阵SVD分解等，但压缩率也只是冰山一角，远不能令人满意。今年standford的HanSong的ICLR的一篇论文Deep Compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding一经提出，就引起了巨大轰动，在这篇论文工作中，他们采用了3步，在不损失（甚至有提升）原始模型精度的基础上，将VGG和Alexnet等模型压缩到了原来的35~49倍，使得原本上百兆的模型压缩到不到10M，令深度学习模型在移动端等的实用成为可能。

How

Deep Compression 的实现主要有三步，如下图所示：

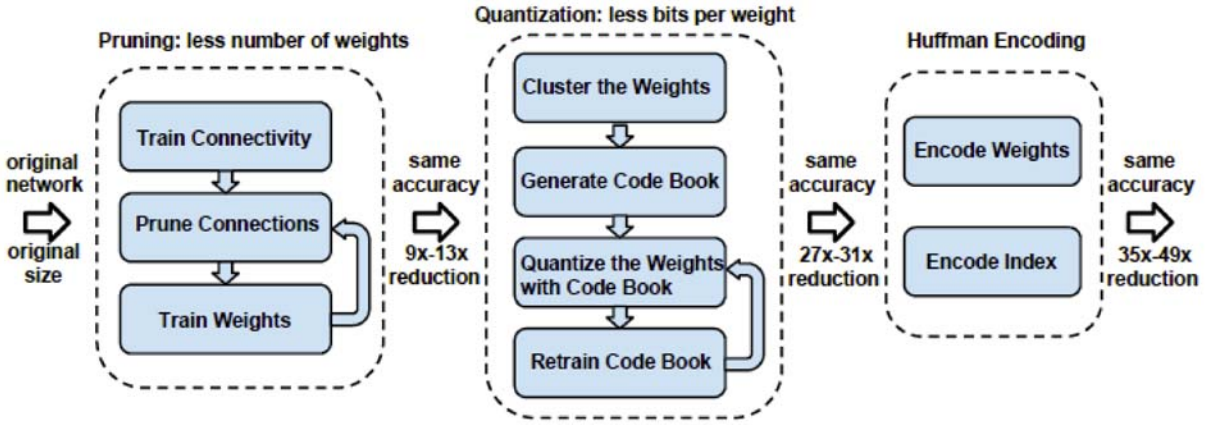


Figure 1: The three stage compression pipeline: pruning, quantization and Huffman coding. Pruning reduces the number of weights by 10×, while quantization further improves the compression rate: between 27× and 31×. Huffman coding gives more compression: between 35× and 49×. The compression rate already included the meta-data for sparse representation. The compression scheme doesn't incur any accuracy loss.

包括Pruning（权值修剪），Quantization（权值共享和量化），Huffman Coding（Huffman编码）。

1.Prunning

如果你调试过caffe模型，观察里面的权值，会发现大部分权值都集中在-1~1之间，即非常小，另一方面，神经网络的提出就是模仿人脑中的神经元突触之间的信息传导，因此这数量庞大的权值中，存在着不可忽视的冗余性，这就为权值修剪提供了根据。pruning可以分为三步：

step1. 正常训练模型得到网络权值；

step2. 将所有低于一定阈值的权值设为0；

step3. 重新训练网络中剩下的非零权值。

经过权值修剪后的稀疏网络，就可以用一种紧凑的存储方式CSC或CSR（compressed sparse column or compressed sparse row）来表示。这里举个栗子来解释下什么是CSR

假设有一个原始稀疏矩阵A

```
                4.0  1.0  0.0  0.0  2.5
                0.0  4.0  1.0  0.0  0.0
A       =       0.0  1.0  4.0  0.0  1.0
                0.0  0.0  1.0  4.0  0.0
                2.5  0.0  0.0  0.5  4.0
```

CSR可以将原始矩阵表达为三部分，即AA,JA,IC

```
AA  =  4.0  1.0  2.5  4.0  1.0  1.0  4.0  1.0  1.0  4.0  2.5  0.5  4.0
JA  =  1    4    6    9    11   14
IC  =  1    2    5    2    3    2    3    5    3    4    1    4    5
```

其中，AA是矩阵A中所有非零元素，长度为a，即非零元素个数；

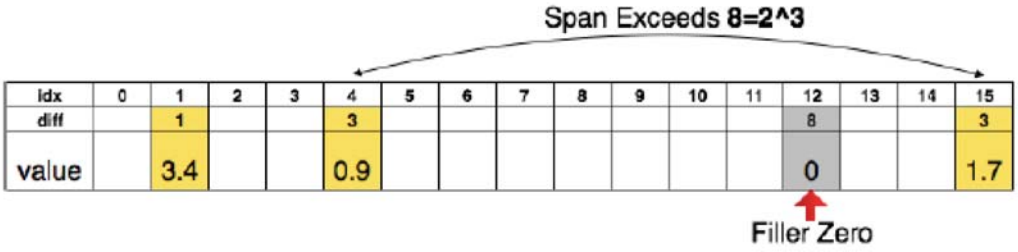
JA是矩阵A中每行第一个非零元素在AA中的位置，最后一个元素是非零元素数加1，长度为n+1, n是矩阵A的行

数；

IC是AA中每个元素对应的列号，长度为a。

所以将一个稀疏矩阵转为CSR表示，需要的空间为2*a+n+1个，同理CSC也是类似。

可以看出，为了达到压缩原始模型的目的，不仅需要在保持模型精度的同时，prune掉尽可能多的权值，也需要减少存储元素位置index所带来的额外存储开销，故论文中采用了存储index difference而非绝对index来进一步压缩模型，如下图所示：



其中，第一个非零元素的存储的是他的绝对位置，后面的元素依次存储的是与前一个非零元素的索引差值。在论文中，采用固定bit来存储这一差值，以图中表述为例，如果采用3bit，则最大能表述的差值为8，当一个非零元素距其前一个非零元素位置超过8，则将该元素值置零。（这一点其实也很好理解，如果两个非零元素位置差很多，也即中间有很多零元素，那么将这一元素置零，对最终的结果影响也不会很大）

做完权值修剪这一步后，AlexNet和VGG-16模型分别压缩了9倍和13倍，表明模型中存在着较大的冗余。

2.Weight Shared & Quantization

为了进一步压缩网络，考虑让若干个权值共享同一个权值，这一需要存储的数据量也大大减少。在论文中，采用kmeans算法来将权值进行聚类，在每一个类中，所有的权值共享该类的聚类质心，因此最终存储的结果就是一个码书和索引表。

1.对权值聚类

论文中采用kmeans聚类算法，通过优化所有类内元素到聚类中心的差距（within-cluster sum of squares ）来确定最终的聚类结果：

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{w \in c_i} |w - c_i|^2$$

式中，W ={w1,w2,...wn}是n个原始权值，C={c1,c2,...ck}是k个聚类。

需要注意的是聚类是在网络训练完毕后做的，因此聚类结果能够最大程度地接近原始网络权值分布。

2. 聚类中心初始化

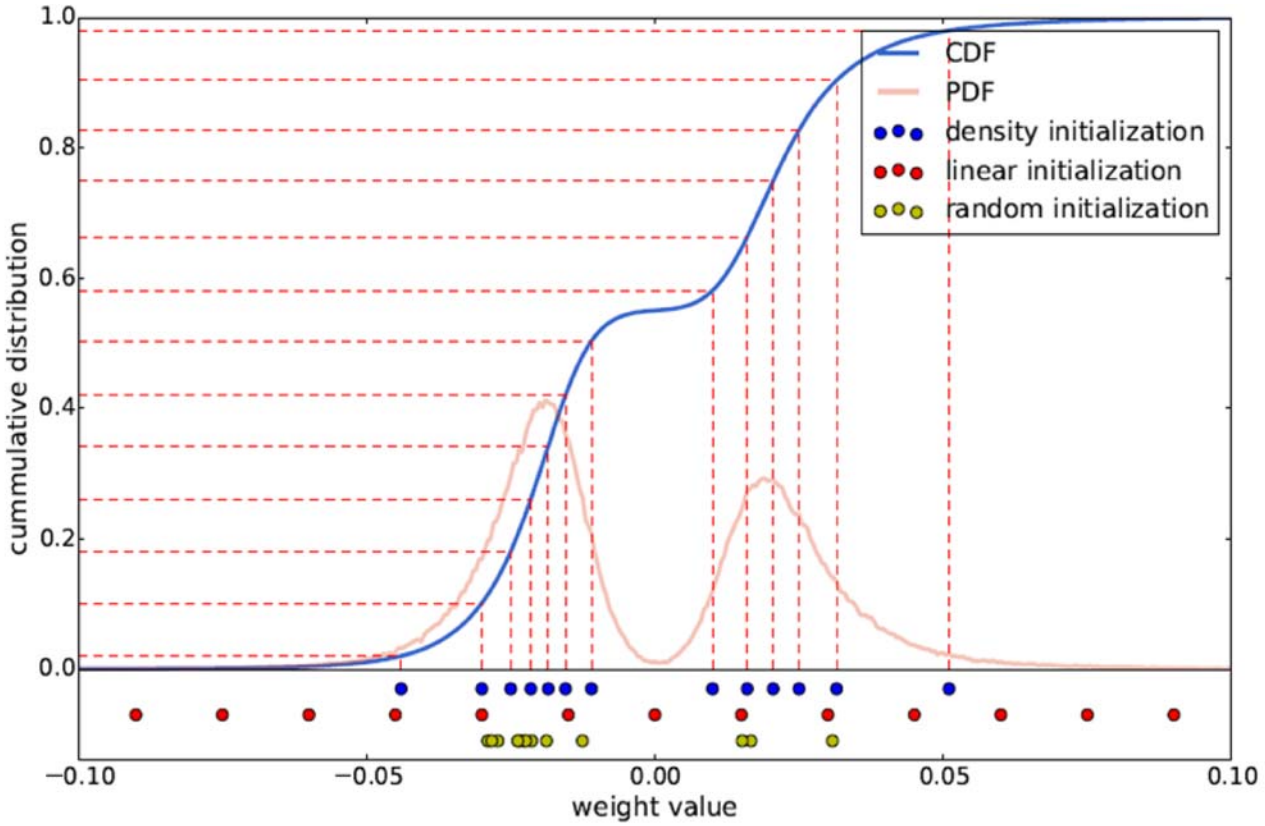
常用的初始化方式包括3种：

a) 随机初始化。即从原始数据种随机产生k个观察值作为聚类中心。

b) 密度分布初始化。现将累计概率密度CDF的y值分布线性划分，然后根据每个划分点的y值找到与CDF曲线的交点，再找到该交点对应的x轴坐标，将其作为初始聚类中心。

c) 线性初始化。将原始数据的最小值到最大值之间的线性划分作为初始聚类中心。

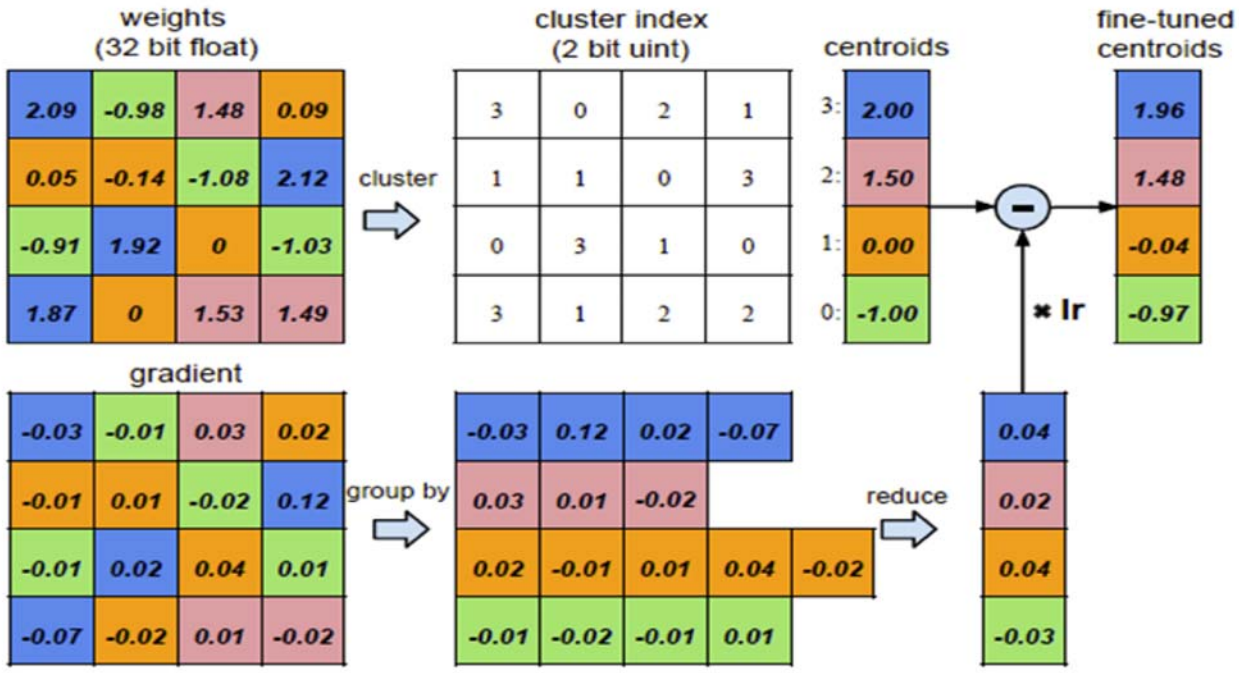
三种初始化方式的示意图如下所示：



由于大权值比小权值更重要（参加HanSong15年论文），而线性初始化方式则能更好地保留大权值中心，因此文中采用这一方式，后面的实验结果也验证了这个结论。

3. 前向反馈和后项传播

前向时需要将每个权值用其对应的聚类中心代替，后向计算每个类内的权值梯度，然后将其梯度和反传，用来更新聚类中心，如图：



共享权值后，就可以用一个码书和对应的index来表征。假设原始权值用32bit浮点型表示，量化区间为256，即

8bit，共有n个权值，量化后需要存储n个8bit索引和256个聚类中心值，则可以计算出压缩率compression ratio:

r = 32*n / (8*n + 256*32)≈4

可以看出，如果采用8bit编码，则至少能达到4倍压缩率。

3.Huffman Coding

Huffman 编码是最后一步，主要用于解决编码长短不一带来的冗余问题。因为在论文中，作者针对卷积层统一采用8bit编码，而全连接层采用5bit，所以采用这种熵编码能够更好地使编码bit均衡，减少冗余。

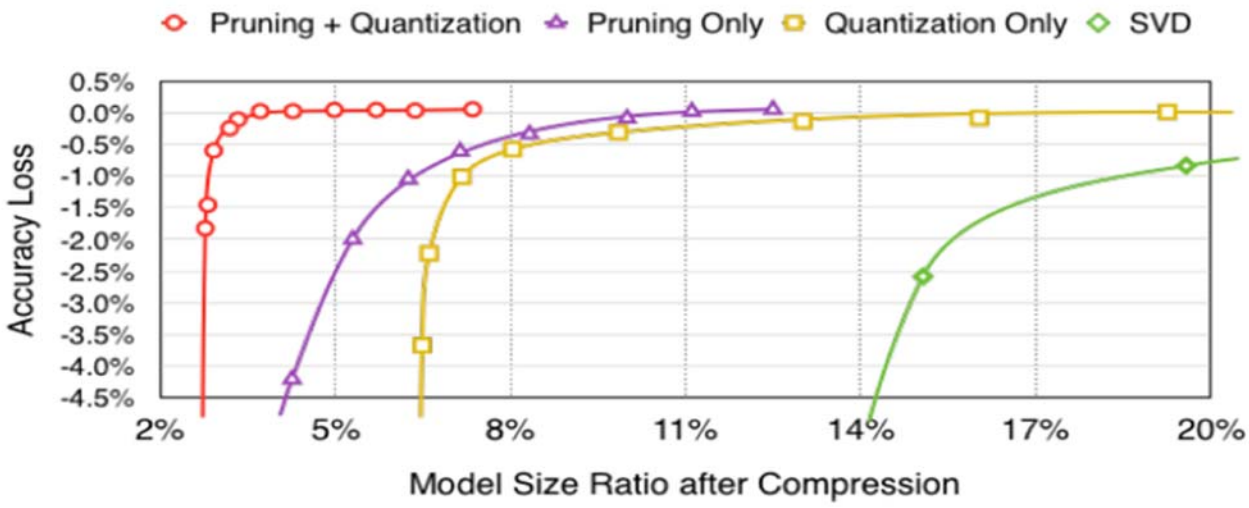
4.Evaluation

实验结果就是能在保持精度不变（甚至提高）的前提下，将模型压缩到前所未有的小。直接上图有用数据说话。

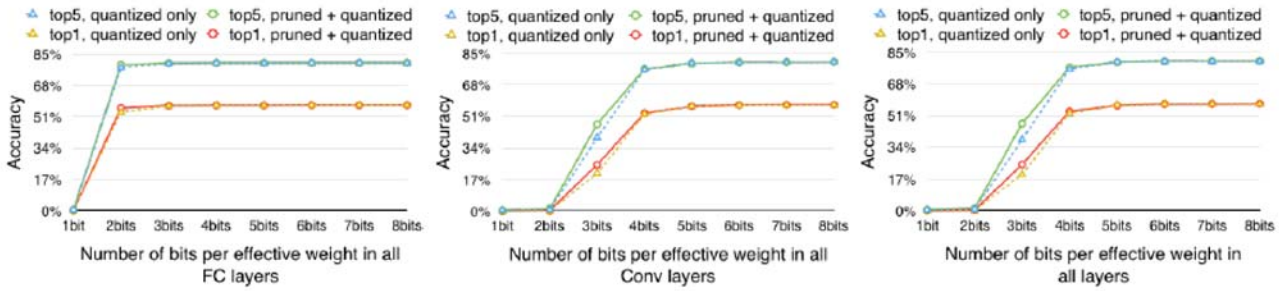
Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
LeNet-300-100 Ref	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	**27 KB**	40×
LeNet-5 Ref	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	**44 KB**	39×
AlexNet Ref	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	**6.9 MB**	35×
VGG-16 Ref	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	49×

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
Baseline Caffemodel (BVLC)	42.78%	19.73%	240MB	1×
Fastfood-32-AD (Yang et al., 2014)	41.93%	-	131MB	2×
Fastfood-16-AD (Yang et al., 2014)	42.90%	-	64MB	3.7×
Collins & Kohli (Collins & Kohli, 2014)	44.40%	-	61MB	4×
SVD (Denton et al., 2014)	44.02%	20.56%	47.6MB	5×
Pruning (Han et al., 2015)	42.77%	19.67%	27MB	9×
Pruning+Quantization	42.78%	19.70%	8.9MB	27×
Pruning+Quantization+Huffman	**42.78%**	**19.70%**	**6.9MB**	**35×**

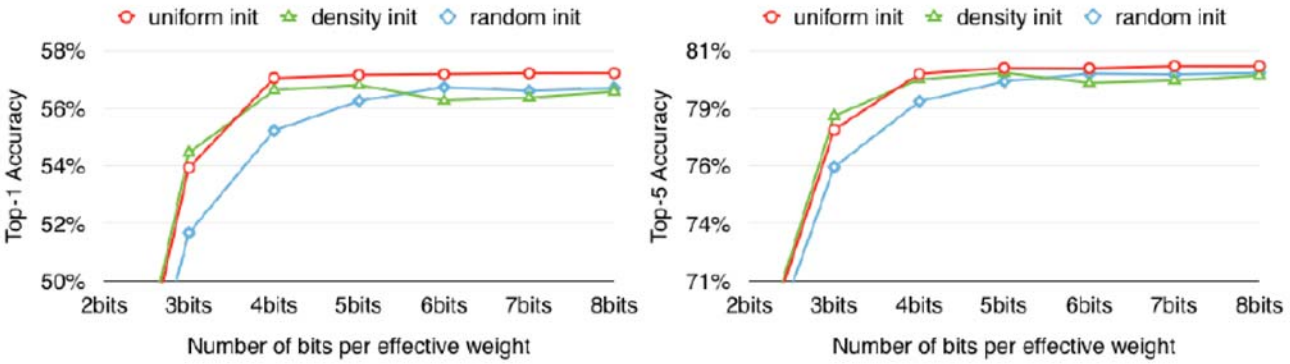
5.Discussion



不同模型压缩比和精度的对比，验证了pruning和quantization一块做效果最好。



不同压缩bit对精度的影响，同时表明conv层比fc层更敏感， 因此需要更多的bit表示。



不同初始化方式对精度的影响，线性初始化效果最好。

#CONV bits / #FC bits	Top-1 Error	Top-5 Error	Top-1 Error Increase	Top-5 Error Increase
32bits / 32bits	42.78%	19.73%	-	-
8 bits / 5 bits	42.78%	19.70%	0.00%	-0.03%
8 bits / 4 bits	42.79%	19.73%	0.01%	0.00%
4 bits / 2 bits	44.77%	22.33%	1.99%	2.60%

卷积层采用8bit，全连接层采用5bit效果最好。

What

此部分讲一讲修改caffe源码的过程。其实只要读懂了文章原理，修改起来很容易。

对pruning过程来说，可以定义一个mask来“屏蔽”修剪掉的权值，对于quantization过程来说，需定义一个indice来存储索引号，以及一个centroid结构来存放聚类中心。

在include/caffe/layer.hpp中为Layer类添加以下成员变量：

```
vector<int> masks_;
vector<int> indices_;
vector<Dtype> centroids_;
```

以及成员函数：

```
virtual void ComputeBlobMask(float ratio) {}
/**
```

由于只对卷积层和全连接层做压缩，因此，只需修改这两个层的对应函数即可。

在include/caffe/layers/base_conv_layer.hpp添加成员函数

```
virtual void ComputeBlobMask(float ratio) {}
```

这两处定义的函数都是基类的虚函数，不需要具体实现。

在include/caffe/layers/conv_layer.hpp中添加成员函数声明：

```
  virtual void ComputeBlobMask(float ratio);
};
```

类似的，在include/caffe/layers/inner_product_layer.hpp也添加该函数声明。

在src/caffe/layers/conv_layer.cpp 添加该函数的声明，用于初始化mask和对权值进行聚类。

```
template <typename Dtype>
void ConvolutionLayer<Dtype>::ComputeBlobMask(float ratio)
{
  //  LOG(INFO)<<"conv blob mask"<<endl;
  int count = this->blobs_[0]->count();
  this->masks_.resize(count);

  //this->dmasks_ = new Dtype[count] ;

  this->indices_.resize(count);
  this->centroids_.resize(CONV_QUNUM);

  //calculate min max value of weight
  const Dtype* weight = this->blobs_[0]->cpu_data();
  Dtype min_weight = weight[0] , max_weight = weight[0];
  vector<Dtype> sort_weight(count);

  for (int i = 0; i < count; ++i)
  {
    //this->masks_[i] = 1; //initialize
    sort_weight[i] = fabs(weight[i]);
  }

  sort(sort_weight.begin(), sort_weight.end());

  max_weight = sort_weight[count - 1];

  //cout << sort_weight[0] << " " << sort_weight[count - 1] << endl;
  int index = int(count*ratio) ; //int(count*(1- max_weight)) ;
  Dtype thr ;
  Dtype* muweight = this->blobs_[0]->mutable_cpu_data();
  float rat = 0;
  if(index > 0){

    //thr = ratio;
    thr= sort_weight[index-1];
    LOG(INFO) << "CONV THR: " <<thr << " " <<ratio <<endl;


    for (int i = 0; i < count; ++i)
    {
  this->masks_[i] = ((weight[i] >= thr || weight[i] < -thr) ? 1 : 0) ;
        //this->masks_[i] = (weight[i] > thr ? 1 : 0);//(weight[i]==0 ? 0 :1) ;//(weight[i] > thr ? 1 : 0);
        muweight[i] *= this->masks_[i];
        //this->dmasks_[i] = this->masks_[i] ;
        rat += (1-this->masks_[i]) ;
     }

  }
  else {
      for (int i = 0; i < count; ++i)
      {
          this->masks_[i] = (weight[i]==0 ? 0 :1); //keep unchanged
    rat += (1-this->masks_[i]) ;
      }
  }
   LOG(INFO) << "sparsity: "<< rat/count <<endl;
  min_weight = sort_weight[index];
  int nCentroid = CONV_QUNUM;
  kmeans_cluster(this->indices_, this->centroids_, muweight, count, this->masks_,/* max_weight, min_weight,*/ nCentroid, 1000);

}
```

同时，修改前向和后向函数。

在前向函数中，需要将权值用其聚类中心表示，红框部分为添加部分：

```
template <typename Dtype>
void ConvolutionLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  Dtype* muweight = this->blobs_[0]->mutable_cpu_data();
  int count = this->blobs_[0]->count();
  for (int i = 0; i < count; ++i)
  {
     if (this->masks_[i])
       muweight[i] = this->centroids_[this->indices_[i]];
  }
  const Dtype* weight = this->blobs_[0]->cpu_data();
  for (int i = 0; i < bottom.size(); ++i) {
    const Dtype* bottom_data = bottom[i]->cpu_data();
    Dtype* top_data = top[i]->mutable_cpu_data();
    for (int n = 0; n < this->num_; ++n) {
      this->forward_cpu_gemm(bottom_data + n * this->bottom_dim_, weight,
          top_data + n * this->top_dim_);
      if (this->bias_term_) {
        const Dtype* bias = this->blobs_[1]->cpu_data();
        this->forward_cpu_bias(top_data + n * this->top_dim_, bias);
      }
    }
  }
}

```

在后向函数中，需要添加两部分，一是对mask为0，即屏蔽掉的权值不再进行更新，即将其weight_diff设为0，另一个则是统计每一类内的梯度差值均值，并将其反传回去，红框内为添加部分。

```
template <typename Dtype>
void ConvolutionLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom) {
  const Dtype* weight = this->blobs_[0]->cpu_data();
  Dtype* weight_diff = this->blobs_[0]->mutable_cpu_diff();
  int count = this->blobs_[0]->count();
  for (int i = 0; i < top.size(); ++i) {
    const Dtype* top_diff = top[i]->cpu_diff();
    const Dtype* bottom_data = bottom[i]->cpu_data();
    Dtype* bottom_diff = bottom[i]->mutable_cpu_diff();
    // Bias gradient, if necessary.
    if (this->bias_term_ && this->param_propagate_down_[1]) {
      Dtype* bias_diff = this->blobs_[1]->mutable_cpu_diff();
      for (int n = 0; n < this->num_; ++n) {
        this->backward_cpu_bias(bias_diff, top_diff + n * this->top_dim_);
      }
    }
    if (this->param_propagate_down_[0] || propagate_down[i]) {
      for (int n = 0; n < this->num_; ++n) {
        // gradient w.r.t. weight. Note that we will accumulate diffs.
        if (this->param_propagate_down_[0]) {
          this->weight_cpu_gemm(bottom_data + n * this->bottom_dim_,
              top_diff + n * this->top_dim_, weight_diff);
        }
        Dtype* weight_diff = this->blobs_[0]->mutable_cpu_diff();
        for (int j = 0; j < count; ++j)
          weight_diff[j] *= this->masks_[j];
        vector<Dtype> tmpDiff(CONV_QUNUM);
        vector<int> freq(CONV_QUNUM);
        for (int j = 0; j < count; ++j)
        {
          if (this->masks_[j])
          {
            tmpDiff[this->indices_[j]] += weight_diff[j];
            freq[this->indices_[j]]++;
          }
        }
        for (int j = 0; j < count; ++j)
        {
          if (this->masks_[j])
            weight_diff[j] = tmpDiff[this->indices_[j]] / freq[this->indices_[j]];
        }
        // gradient w.r.t. bottom data, if necessary.
        if (propagate_down[i]) {
          this->backward_cpu_gemm(top_diff + n * this->top_dim_, weight,
              bottom_diff + n * this->bottom_dim_);
        }
      }
    }
  }
}

```

kmeans的实现如下，当然也可以用Opencv自带的，速度会更快些。

```
template<typename Dtype>
void kmeans_cluster(vector<int> &cLabel, vector<Dtype> &cCentro, Dtype *cWeights, int nWeig
{
    //find min max
    Dtype maxWeight=numeric_limits<Dtype>::min(), minWeight=numeric_limits<Dtype>::max();
    for(int k = 0; k < nWeights; ++k)
    {
        if(mask[k])
        {
            if(cWeights[k] > maxWeight)
                maxWeight = cWeights[k];
            if(cWeights[k] < minWeight)
                minWeight = cWeights[k];
        }
    }
    // generate initial centroids linearly
    for (int k = 0; k < nCluster; k++)
        cCentro[k] = minWeight + (maxWeight - minWeight)*k / (nCluster - 1);

    //initialize all label to -1
    for (int k = 0; k < nWeights; ++k)
        cLabel[k] = -1;

    const Dtype float_max = numeric_limits<Dtype>::max();
    // initialize
    Dtype *cDistance = new Dtype[nWeights];
    int *cClusterSize = new int[nCluster];

    Dtype *pCentroPos = new Dtype[nCluster];
    int *pClusterSize = new int[nCluster];
    memset(pClusterSize, 0, sizeof(int)*nCluster);
    memset(pCentroPos, 0, sizeof(Dtype)*nCluster);
    Dtype *ptrC = new Dtype[nCluster];
    int *ptrS = new int[nCluster];

    int iter = 0;
    //Dtype tk1 = 0.f, tk2 = 0.f, tk3 = 0.f;
    double mCurDistance = 0.0;
    double mPreDistance = numeric_limits<double>::max();

    // clustering
    while (iter < max_iter)
    {
        // check convergence
        if (fabs(mPreDistance - mCurDistance) / mPreDistance < 0.01) break;
        mPreDistance = mCurDistance;
        mCurDistance = 0.0;

        // select nearest cluster

        for (int n = 0; n < nWeights; n++)
        {
            if (!mask[n])
                continue;
            Dtype distance;
```

```
            Dtype mindistance = float_max;
            int clostCluster = -1;
            for (int k = 0; k < nCluster; k++)
            {
                distance = fabs(cWeights[n] - cCentro[k]);
                if (distance < mindistance)
                {
                    mindistance = distance;
                    clostCluster = k;
                }
            }
            cDistance[n] = mindistance;
            cLabel[n] = clostCluster;
        }


        // calc new distance/inertia

        for (int n = 0; n < nWeights; n++)
        {
            if (mask[n])
                mCurDistance = mCurDistance + cDistance[n];
        }


    // generate new centroids
    // accumulation(private)

        for (int k = 0; k < nCluster; k++)
        {
            ptrC[k] = 0.f;
            ptrS[k] = 0;
        }

        for (int n = 0; n < nWeights; n++)
        {
            if (mask[n])
            {
                ptrC[cLabel[n]] += cWeights[n];
                ptrS[cLabel[n]] += 1;
            }
        }

        for (int k = 0; k < nCluster; k++)
        {
            pCentroPos[ k] = ptrC[k];
            pClusterSize[k] = ptrS[k];
        }

        //reduction(global)
        for (int k = 0; k < nCluster; k++)
        {

            cCentro[k] = pCentroPos[k];
            cClusterSize[k] = pClusterSize[k];
```

```
111
112             cCentro[k] /= cClusterSize[k];
113         }
114
115         iter++;
116     //  cout << "Iteration: " << iter << " Distance: " << mCurDistance << endl;
117         }
118         //gather centroids
119         //#pragma omp parallel for
120         //for(int n=0; n<nNode; n++)
121         //    cNodes[n] = cCentro[cLabel[n]];
122
123         delete[] cDistance;
124         delete[] cClusterSize;
125         delete[] pClusterSize;
126         delete[] pCentroPos;
127         delete[] ptrC;
128         delete[] ptrS;
129 }
```

全连接层的修改和卷积层的一致不再赘述。同样的，可以把对应的.cu文件中的gpu前向和后向函数实现也修改了，方便gpu训练。

最后，在src/caffe/net.cpp的CopyTrainedLayersFrom(const NetParameter& param)函数中调用我们定义的函数，即在读入已经训练好的模型权值时，对每一层做需要的权值mask初始化和权值聚类。

```
796     layers_[target_layer_id]->ComputeBlobMask(ratio) ;
```

至此代码修改完毕，编译运行即可。

Reference

[1] SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size

[2] Deep Compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding

[3] Learning both Weights and Connections for Efficient Neural Networks

[4] Efficient Inference Engine on Compressed Deep Neural Network

总结：

最后再提一句，几乎所有的模型压缩文章都是从Alexnet和VGG下手，一是因为他们都采用了多层较大的全连接层，而全连接层的权值甚至占到了总参数的90%以上，所以即便只对全连接层进行“开刀”，压缩效果也是显著的。另一方面，这些论文提出的结果在现在看来并不是state of art的，存在可提升的空间，而且在NIN的文章中表明，全连接层容易引起过拟合，去掉全连接层反而有助于精度提升，所以这么看来压缩模型其实是个不吃力又讨好的活，获得的好处显然是双倍的。但运用到特定的网络中，还需要不断反复试验，因地制宜，寻找适合该网络的压缩方式。

查看原文>> (http://blog.csdn.net/may0324/article/details/52935869)

0

看过本文的人也看了：

- 深度学习知识结构图 (http://lib.csdn.net/base/deeplearning/structure)
- FPGA机器学习之stanford机器学习第五堂... (http://lib.csdn.net/article/deeplearning/54529)
- 【连载】【FPGA黑金开发板】Verilog H... (http://lib.csdn.net/article/deeplearning/52578)
- xilinx fpga学习笔记5：Xst综合属性 (http://lib.csdn.net/article/deeplearning/52390)
- 基于FPGA的多通道数据采集系统设计 (http://lib.csdn.net/article/deeplearning/54523)
- [D-VI] my_second_fpga（1位加法器 Veri... (http://lib.csdn.net/article/deeplearning/53657)

发表评论

输入评论内容

发表

22**个评论**

(http://my.csdn.net/rill_zhen)

rill_zhen (http://my.csdn.net/rill_zhen)

求助一下，修改conv_layer.cpp/conv_layer.cu后，使用CPU/GPU进行训练时（先用CPU模式，后来尝试用GPU模式，都有错误），一旦调用forward_cpu/forward_gpu就会报错。尝试了各种方法后，我把muweight[i] = this->centroids_[this->indices_[i]]注释掉就能正常运行，backward_cpu/backward_gpu()也有类似情况，请问你有遇到过这种情况么？

2017-03-08 09:26:14 回复

(http://my.csdn.net/rill_zhen)

rill_zhen (http://my.csdn.net/rill_zhen)

回复rill_zhen： 尝试过下面评论中的方法，但没有效果。

2017-03-08 09:30:24 回复

(http://my.csdn.net/u013478129)

u013478129 (http://my.csdn.net/u013478129)

哪位善长仁翁愿意分享一下这个修改好的代码在github呢?我一直很好奇,这个代码或者算法是不是有些很神秘的地方,或者什么潜规则,原作song han也不愿意分享这个代码?

2017-03-02 22:21:44 回复


(http://my.csdn.net/u013478129)

u013478129 (http://my.csdn.net/u013478129)

回复u013478129： 哪位善长仁翁愿意分享一下这个修改好的代码在github呢?我一直很好奇,这个代码或者算法是不是有些很神秘的地方,或者什么潜规则,原作song han也不愿意分享这个代码?37303 9065 at qq.com

2017-03-03 16:02:33 回复


(http://my.csdn.net/qq_35800608)

qq_35800608 (http://my.csdn.net/qq_35800608)

博主，你好，请问一下Deep Compression的具体运行步骤是什么，ReadMe里面的内容没有看懂

2017-02-21 22:00:07 回复

加载更多