20210210

孙阳

论文学习

- 《Efficient Speech Recognition Engine with Compressed LSTM on FPGA》基于FPGA的高效压缩LSTM语音识别引擎 FPGA 2017
- 《Efficient methods and hardware for deep learning》
- 亮点：
  - 考虑到最终要多核运行并行加速的时候不同核心之间的负载均衡，提出了Load-balance-aware pruning算法
  - Deep compression，从软件端极大的压缩了网络的权重。
  - DSD，密集，稀疏，密集的训练方法，一种新的网络训练方法，能从训练层面一定的提升网络准确率
  - EIE: 在Deep compression 的基础上，EIE是基于硬件的稀疏网络加速实现，硬件上达到很好的效果。

Deep compression

- 过程：
  - 1.剪枝
  - 2.稀疏矩阵存储
  - 3.权值共享
- 压缩率：

$$r = \frac{nb}{nlog_2(k) + kb}$$

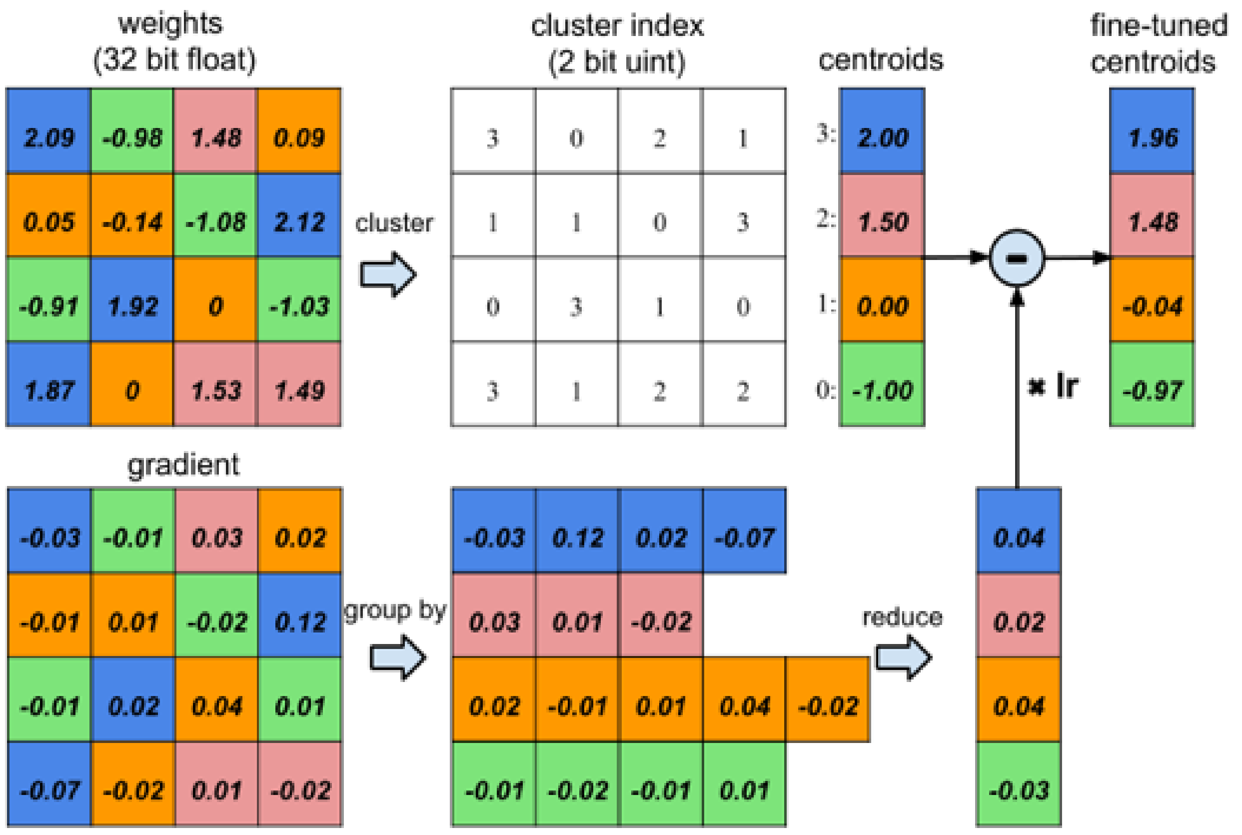

Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).

DSD网络

- Dense-sparse-dense training for deep neural networks

- 先Dense训练，获得一个网络权重；然后spars将网络剪枝，然后继续dense训练。从而获得更好的性能。

Algorithm 1: Workflow of DSD training

Initialization: $W^{(0)}$ *with* $W^{(0)} \sim N(0, \Sigma)$
Output : $W^{(t)}$.

——————————————————— *Initial Dense Phase* ——

while *not converged* **do**
  $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$;
  $t = t + 1$;
end

——————————————————— *Sparse Phase* ———

// initialize the mask by sorting and keeping the Top-k weights.
$S = sort(|W^{(t-1)}|)$; $\lambda = S_{k_i}$; $Mask = \mathbb{1}(|W^{(t-1)}| > \lambda)$;
while *not converged* **do**
  $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$;
  $W^{(t)} = W^{(t)} \cdot Mask$;
  $t = t + 1$;
end

——————————————————— *Final Dense Phase* ——

while *not converged* **do**
  $W^{(t)} = W^{(t-1)} - \eta^{(t)} \nabla f(W^{(t-1)}; x^{(t-1)})$;
  $t = t + 1$;
end
goto *Sparse Phase* for iterative DSD;

EIE网络

$$b_i = ReLU\left(\sum_{j=0}^{n-1} W_{ij} a_j\right)$$

$$b_i = ReLU\left(\sum_{j \in X_i \cap Y} S[I_{ij}] a_j\right)$$

理论学习

- 李宏毅NLP-BERT P17-P19

实验

- 一、利用cpu和HLS运行同一份RNN训练模型，进行运算速度对比
- 二、Deep-Compression-PyTorch
  - 实验使用了MNIST dataset 训练了LeNet-300-100 model
  - 重点：权值共享、哈夫曼编码

实验1代码

```
void train()
{
int epoch, i, j, k, m, p;
vector<double*> layer_1_vector;     //保存隐藏层
vector<double> layer_2_delta;       //保存误差关于Layer 2 输出值的偏导

for(epoch=0; epoch<10000; epoch++)  //训练次数
{
    double e = 0.0;  //误差
    for(i=0; i<layer_1_vector.size(); i++)
        delete layer_1_vector[i];
    layer_1_vector.clear();
    layer_2_delta.clear();

    int d[binary_dim];                  //保存每次生成的预测值
    memset(d, 0, sizeof(d));

    int a_int = (int)randval(largest_number/2.0);  //随机生成一个加数 a
    int a[binary_dim];
    int2binary(a_int, a);               //转为二进制数

    int b_int = (int)randval(largest_number/2.0);  //随机生成另一个加数 b
    int b[binary_dim];
    int2binary(b_int, b);               //转为二进制数

    int c_int = a_int + b_int;          //真实的和 c
    int c[binary_dim];
    int2binary(c_int, c);               //转为二进制数

    double *layer_1 = new double[hidenode];
    for(i=0; i<hidenode; i++)           //在0时刻是没有之前的隐含层的，所以初始化一个全为0的
        layer_1[i] = 0;
    layer_1_vector.push_back(layer_1);
```

```
//正向传播
for(p=0; p<binary_dim; p++)           //循环遍历二进制数组，从最低位开始
{
    layer_0[0] = a[p];
    layer_0[1] = b[p];
    double y = (double)c[p];          //实际值
    layer_1 = new double[hidenode];   //当前隐含层

    for(j=0; j<hidenode; j++)
    {
        //输入层传播到隐含层
        double o1 = 0.0;
        for(m=0; m<innode; m++)
            o1 += layer_0[m] * w[m][j];

        //之前的隐含层传播到现在的隐含层
        double *layer_1_pre = layer_1_vector.back();
        for(m=0; m<hidenode; m++)
            o1 += layer_1_pre[m] * wh[m][j];

        layer_1[j] = sigmoid(o1);     //隐藏层各单元输出
    }

    for(k=0; k<outnode; k++)
    {
        //隐藏层传播到输出层
        double o2 = 0.0;
        for(j=0; j<hidenode; j++)
            o2 += layer_1[j] * w1[j][k];
        layer_2[k] = sigmoid(o2);     //输出层各单元输出
    }
```

实验2结果

```
--- Before pruning ---
fc1.weight           | nonzeros =  235200 /  235200 (100.00%) | total_pruned =       0 | shape = (300, 784)
fc1.bias             | nonzeros =     300 /     300 (100.00%) | total_pruned =       0 | shape = (300,)
fc2.weight           | nonzeros =   30000 /   30000 (100.00%) | total_pruned =       0 | shape = (100, 300)
fc2.bias             | nonzeros =     100 /     100 (100.00%) | total_pruned =       0 | shape = (100,)
fc3.weight           | nonzeros =    1000 /    1000 (100.00%) | total_pruned =       0 | shape = (10, 100)
fc3.bias             | nonzeros =      10 /      10 (100.00%) | total_pruned =       0 | shape = (10,)
alive: 266610, pruned : 0, total: 266610, Compression rate :       1.00x  (  0.00% pruned)
Pruning with threshold : 0.22420135140419006 for layer fc1
Pruning with threshold : 0.1908438801765442 for layer fc2
Pruning with threshold : 0.23130165040493011 for layer fc3
Test set: Average loss: 1.0954, Accuracy: 6761/10000 (67.61%)
--- After pruning ---
fc1.weight           | nonzeros =   10285 /  235200 (  4.37%) | total_pruned =  224915 | shape = (300, 784)
fc1.bias             | nonzeros =     300 /     300 (100.00%) | total_pruned =       0 | shape = (300,)
fc2.weight           | nonzeros =    1360 /   30000 (  4.53%) | total_pruned =   28640 | shape = (100, 300)
fc2.bias             | nonzeros =     100 /     100 (100.00%) | total_pruned =       0 | shape = (100,)
fc3.weight           | nonzeros =      69 /    1000 (  6.90%) | total_pruned =     931 | shape = (10, 100)
fc3.bias             | nonzeros =      10 /      10 (100.00%) | total_pruned =       0 | shape = (10,)
alive: 12124, pruned : 254486, total: 266610, Compression rate :      21.99x  ( 95.45% pruned)
--- Retraining ---
```

实验2结果

```
--- After Retraining ---
fc1.weight          | nonzeros =   10285 /  235200 (  4.37%) | total_pruned =  224915 | shape = (300, 784)
fc1.bias            | nonzeros =     300 /     300 (100.00%) | total_pruned =       0 | shape = (300,)
fc2.weight          | nonzeros =    1360 /   30000 (  4.53%) | total_pruned =   28640 | shape = (100, 300)
fc2.bias            | nonzeros =     100 /     100 (100.00%) | total_pruned =       0 | shape = (100,)
fc3.weight          | nonzeros =      69 /    1000 (  6.90%) | total_pruned =     931 | shape = (10, 100)
fc3.bias            | nonzeros =      10 /      10 (100.00%) | total_pruned =       0 | shape = (10,)
alive: 12124, pruned : 254486, total: 266610, Compression rate :      21.99x  ( 95.45% pruned)
```

```
Layer               |   original compressed improvement percent
----------------------------------------------------------------
fc1.weight          |      83484      19452        4.29x  23.30%
fc1.bias            |       1200       1200        1.00x 100.00%
fc2.weight          |      11284       3196        3.53x  28.32%
fc2.bias            |        400        400        1.00x 100.00%
fc3.weight          |        596        398        1.50x  66.78%
fc3.bias            |         40         40        1.00x 100.00%
----------------------------------------------------------------
total               |      97004      24686        3.93x  25.45%
```