WUST

学习总结

Mode by : 董 勇

目录
CONTENT

PART 01

[1]S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," arXiv:1510.00149 [cs], Feb. 2016, Accessed: Dec. 30, 2020. [Online]. Available: http://arxiv.org/abs/1510.00149.

01 Paper

motivation

1. 先前看的文章都是针对在fpga上压缩的，大部分都是直接给的压缩好的数据，但是如何将数据压缩成这样，对于具体的压缩策略实施时，如何选择重要的权重没有提到。

1. 引入了深度压缩，分为三个阶段：剪枝、训练量化和哈夫曼编码

01 Paper

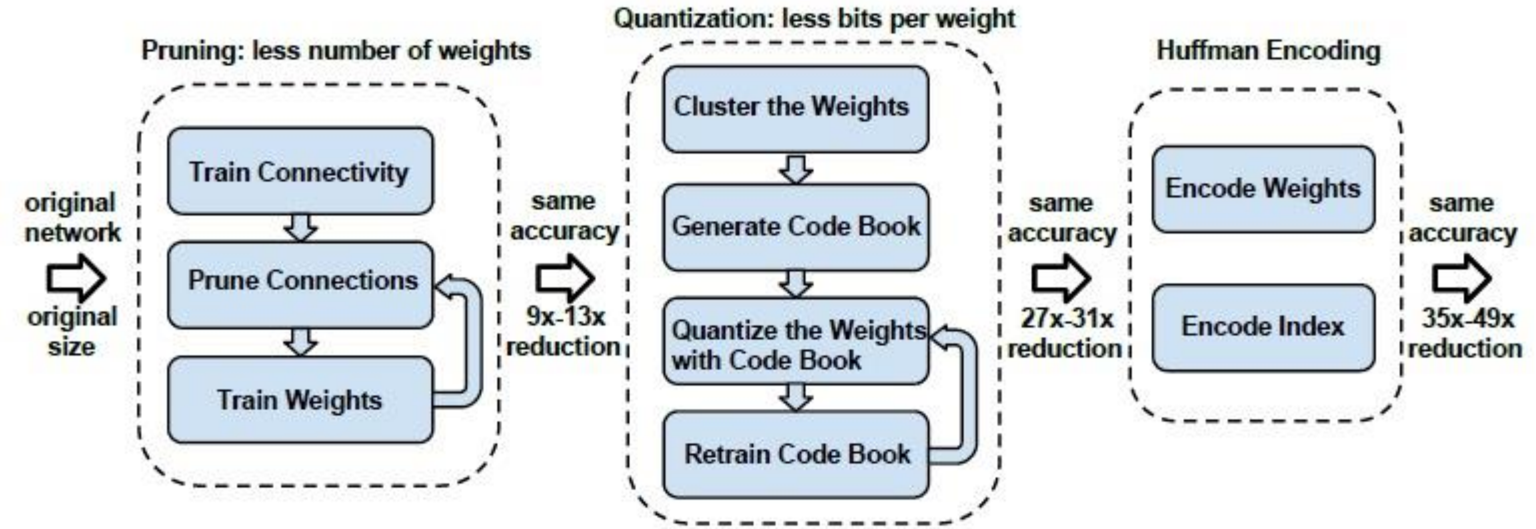

Figure 1: The three stage compression pipeline: pruning, quantization and Huffman coding. Pruning reduces the number of weights by $10\times$, while quantization further improves the compression rate: between $27\times$ and $31\times$. Huffman coding gives more compression: between $35\times$ and $49\times$.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

训练一个网络

将小于某个阈值的权重扔掉，设为0不再训练

重新训练这个剪枝完的稀疏连接的网络

01 Paper

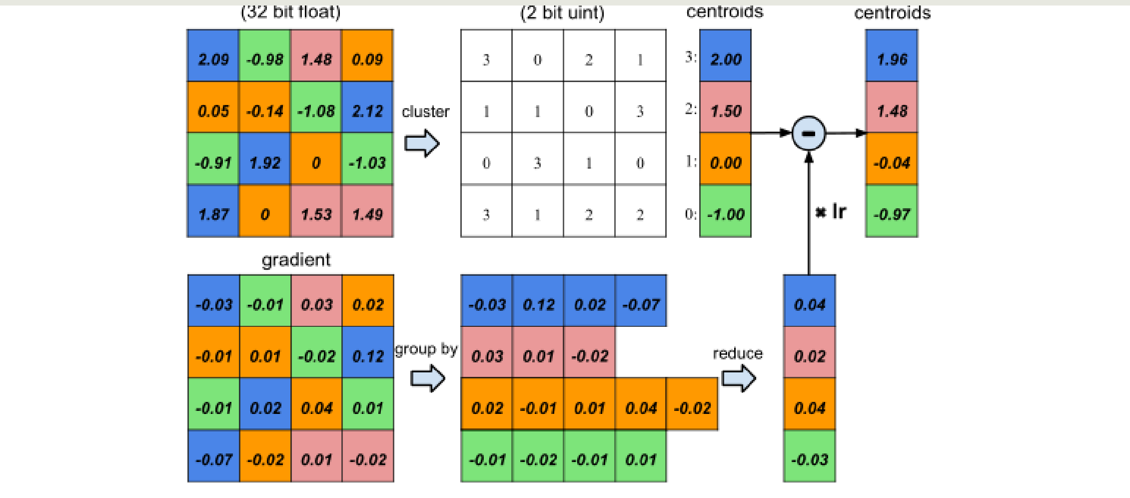

Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).

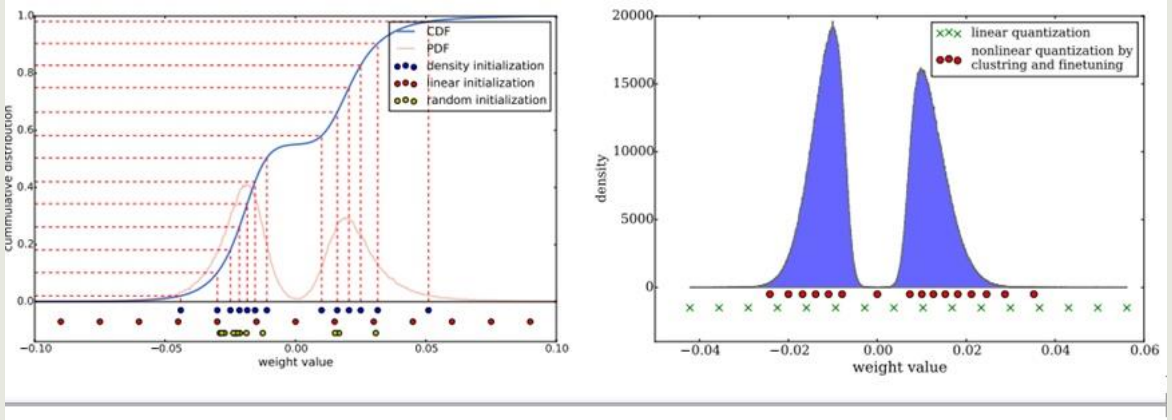

图 4 Left：三个不同方法的质心初始化。Right：微调前后的权值分布和码本分布

质心初始化影响聚类的质量，因而会影响网络的预测精确度。本文实验了三种初始化方法：Forgy（random），density-based和linear初始化。

01 Paper

质心初始化影响聚类的质量，因而会影响网络的预测精确度。本文实验了三种初始化方法：Forgy（random），density-based和linear初始化。

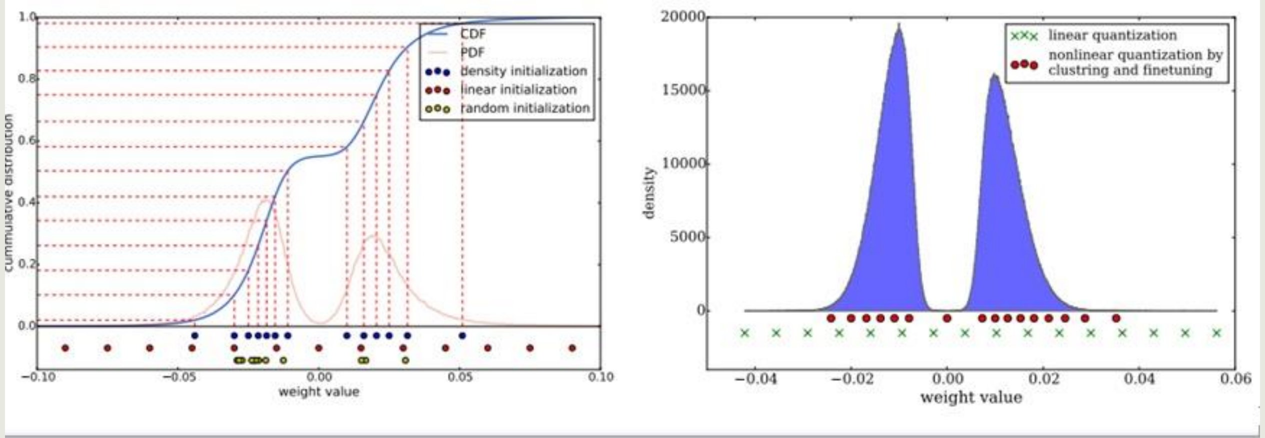

图 4·Left：三个不同方法的质心初始化。Right：微调前后的权值分布和码本分布。

较大的权值比较小的权值有着更重要的角色，但是较大的权值很少。因此，Forgy和density-based初始化很少有大的绝对值，这就导致较少的权值被微弱表达。但是Linear初始化没有遇到这个问题，实验部分比较了准确性，发现Linear初始化效果最好。

01 Paper

不同压缩方法在AlexNet上的比较

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×**

PART

02

- NLP(p9-p12)

PART 03

- Network pruning

03/ Experiment

筛选方法：抓出每一个block的batchnorm的γ即可

```
def network_slimming(old_model, new_model):
    params = old_model.state_dict()
    new_params = new_model.state_dict()

    # selected_idx: 每一层选择的neuron_index
    selected_idx = []
    # 逐一抓取选择的neuron_index
    for i in range(8):
        # 根据上表，我们要抓的gamma稀疏在cnn.{i}.1.weight内
        importance = params[f'cnn.{i}.1.weight']
        # 抓取总共要筛选几个neuron
        old_dim = len(importance)
        new_dim = len(new_params[f'cnn.{i}.1.weight'])
        # 以Ranking做Index排序，较大的会在前面
        ranking = torch.argsort(importance, descending=True)
        # 把筛选结果放入selected_idx中。
        selected_idx.append(ranking[:new_dim])

    now_processed = 1
    for (name, p1), (name2, p2) in zip(params.items(), new_params.items()):
        # 如果是cnn层，则移植参数
        # 如果是FC层，或是该参数只有一个数字，那么直接复制
        if name.startswith('cnn') and p1.size() != torch.Size([]) and now_processed != ler
            # 当处理到Pointwise的weight时，让now_precessd+1,表示该层的移植已经完成
            if name.startswith(f'cnn.{now_processed}.3'):
                now_processed += 1

            # 如果是pointwise，weight会被上一层的pruning和下一层的pruning所影响，因此需要特判
            if name.endswith('3.weight'):
                # 如果是最后一层cnn，则输出的neuron不需要prune掉。
```

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

03/ Experiment

```
rate 0.9500 epoch   0: train loss: 0.4825, acc 0.8676 valid loss: 1.1327, acc 0.7971
rate 0.9500 epoch   1: train loss: 0.4858, acc 0.8656 valid loss: 1.1253, acc 0.7965
rate 0.9500 epoch   2: train loss: 0.4980, acc 0.8648 valid loss: 1.0623, acc 0.8003
rate 0.9500 epoch   3: train loss: 0.4837, acc 0.8726 valid loss: 1.1127, acc 0.8015
rate 0.9500 epoch   4: train loss: 0.5124, acc 0.8643 valid loss: 1.1098, acc 0.7974
rate 0.9025 epoch   0: train loss: 0.5723, acc 0.8444 valid loss: 1.2317, acc 0.7790
rate 0.9025 epoch   1: train loss: 0.5853, acc 0.8424 valid loss: 1.1912, acc 0.7851
rate 0.9025 epoch   2: train loss: 0.5930, acc 0.8392 valid loss: 1.1866, acc 0.7822
rate 0.9025 epoch   3: train loss: 0.5724, acc 0.8439 valid loss: 1.1844, acc 0.7799
rate 0.9025 epoch   4: train loss: 0.6047, acc 0.8408 valid loss: 1.1769, acc 0.7808
rate 0.8574 epoch   0: train loss: 0.7062, acc 0.8101 valid loss: 1.2120, acc 0.7671
rate 0.8574 epoch   1: train loss: 0.6902, acc 0.8100 valid loss: 1.2048, acc 0.7618
rate 0.8574 epoch   2: train loss: 0.6883, acc 0.8147 valid loss: 1.1950, acc 0.7685
rate 0.8574 epoch   3: train loss: 0.6973, acc 0.8113 valid loss: 1.1849, acc 0.7650
rate 0.8574 epoch   4: train loss: 0.6766, acc 0.8087 valid loss: 1.2003, acc 0.7641
rate 0.8145 epoch   0: train loss: 0.8337, acc 0.7742 valid loss: 1.2930, acc 0.7344
rate 0.8145 epoch   1: train loss: 0.8037, acc 0.7789 valid loss: 1.3014, acc 0.7327
rate 0.8145 epoch   2: train loss: 0.8287, acc 0.7763 valid loss: 1.3470, acc 0.7306
rate 0.8145 epoch   3: train loss: 0.8248, acc 0.7737 valid loss: 1.2703, acc 0.7402
rate 0.8145 epoch   4: train loss: 0.8238, acc 0.7739 valid loss: 1.2995, acc 0.7370
rate 0.7738 epoch   0: train loss: 1.0424, acc 0.7257 valid loss: 1.4421, acc 0.7061
rate 0.7738 epoch   1: train loss: 0.9900, acc 0.7370 valid loss: 1.4656, acc 0.6991
rate 0.7738 epoch   2: train loss: 0.9918, acc 0.7361 valid loss: 1.4323, acc 0.7093
rate 0.7738 epoch   3: train loss: 1.0161, acc 0.7322 valid loss: 1.4760, acc 0.7015
rate 0.7738 epoch   4: train loss: 1.0146, acc 0.7283 valid loss: 1.4445, acc 0.7020
```