Deep Learning: Programing and Deployment

Girish Varma
IIIT Hyderabad

Deep Learning Programming

Deep Learning Programming

1. Specify a Model Architecture
    a. Choose the complexity/structure of the curve to fit the data.
2. Specify Loss function and gradient update method
    a. Choose loss function based on problem (classification, regression, retrieval, detection).
    b. Choose how to update the learnable parameters of the model.
3. Train the Model
    a. Load minibatch of data, normalize it.
    b. Compute predictions and loss function.
    c. Compute and update gradients.
    d. Plotting loss.
4. Deploy the Model
    a. Compress the Model to be memory and runtime efficient.

Deep Learning Libraries

- Gives higher level objects for specifying model, gradient update, training and deployment.
- Automatic Differentiation of loss function with respect to model parameters.
- Fast parallel platform specific implementation of training and testing.
- Asynchronous loading of data.

Deep Learning Libraries

	Language	Created By	
Torch	Lua	NYU & IDIAP	2002
Theano	Python	Toronto & Montreal	2009
Caffe	C++	UC Berkeley	2012
Tensorflow	Python	Google	2015
Pytorch	Python	Facebook	2017

- Not comprehensive. Almost every company have their own implementation.
- Deployment happens in C/CPP, Python only used during training.

MNIST Classification

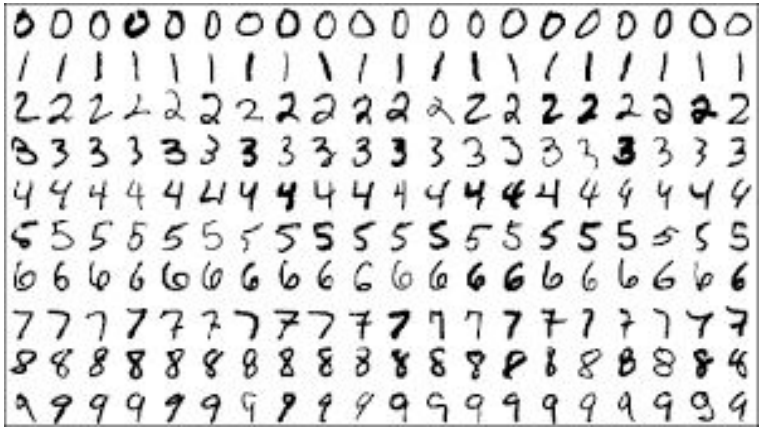
Input : x is a [28,28] shaped matrix, giving pixel values of the image

Output : y is a [10] shaped vector, giving the probabilities of being 0 to 9.

Dataset : Consist of (x,y) pairs, x is the input and y is called the label.

Divided into train, test and validation.

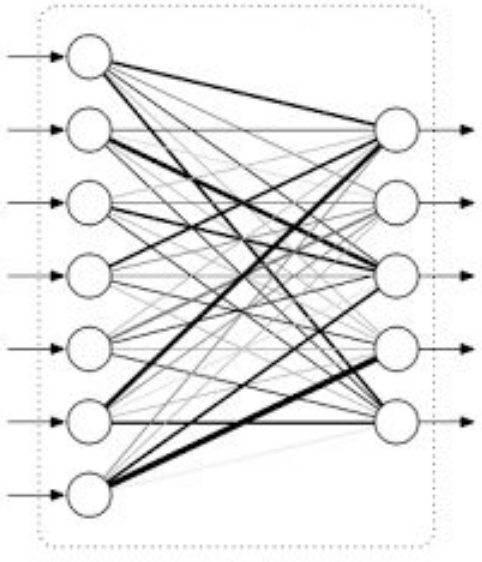
If the dataset gives y as a digit, convert it to probability vector by one hot encoding.



Recall the Neural Network Model

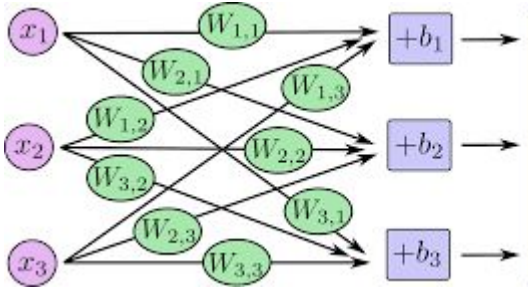
- Neuron or Perceptron
  - Input X is n dimensional, Y is 1 dimensional.
  - Has learnable parameters W = (W_1,W_2,..., W_n) (weights) and b (bias).
  - $Y = \sigma(\sum W_i X_i + b)$
  - **σ** is a non linear activation function.

- Fully Connected or Linear
  - Y is also multidimensional (dimension m).
  - Has learnable parameters W = (W_{ij}) and b = (b_j) where i <= n, j <= m
  - $Y = \sigma(WX + b)$



Specify a Model Architecture : Linear Model

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc = nn.Linear(28*28, 10)

    def forward(self, x):
        x = x.view(-1,28*28)
        x = self.fc(x)
        return x
```



Obtain prediction probabilities for 10 classes, by applying softmax function.

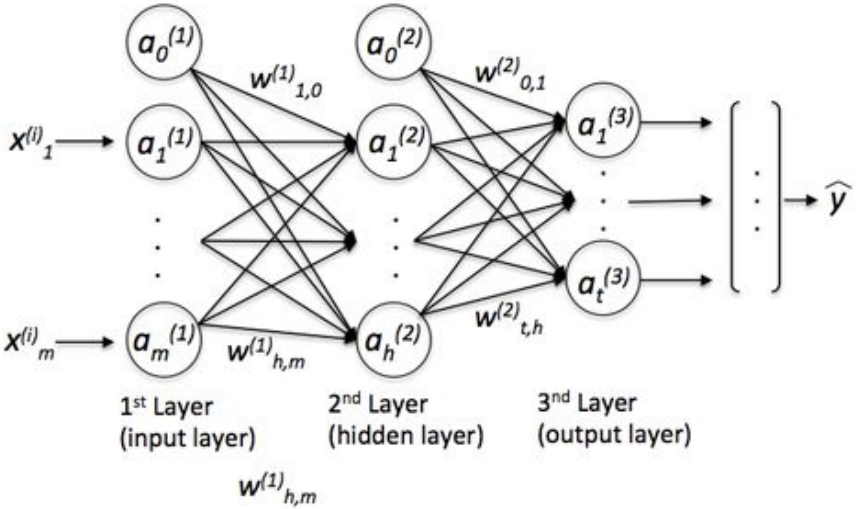
Multilayered Network

Complex data fits only more complex models.

Obtain complex models by layering multiple linear layers.

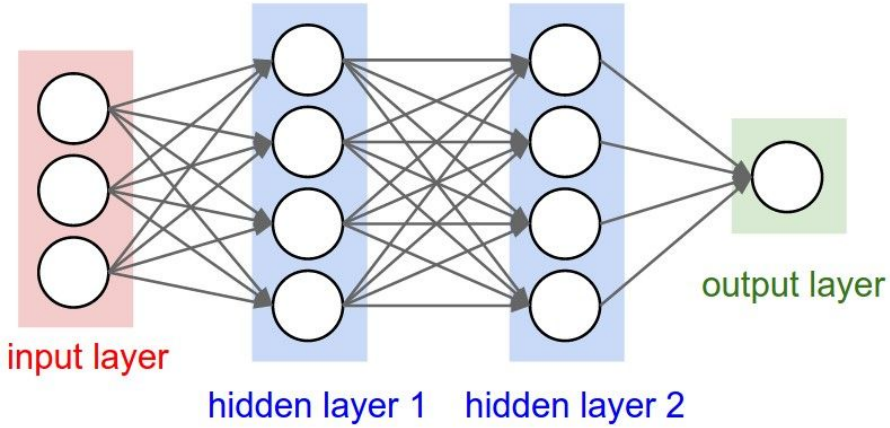
Multilayered Perceptron (MLP)

- Multiple Linear layers
  one following the other.
- Y = **σ**(V **σ**(WX+ b) + c)
- Intermediate outputs are called hidden units.



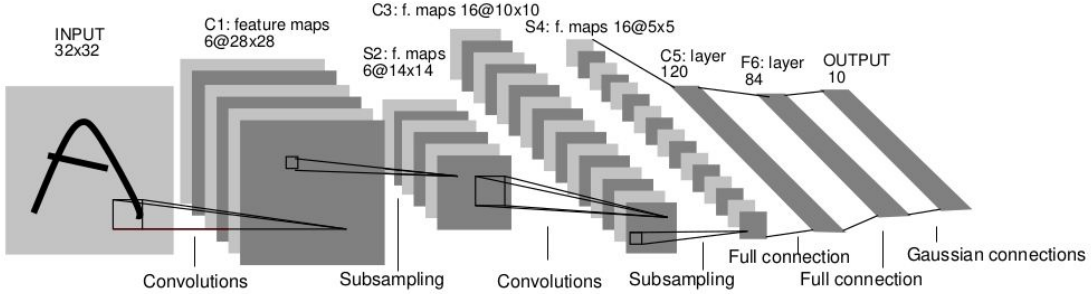
Specify a Model Architecture : MLP Model

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 120)
        self.fc2 = nn.Linear(120, 80)
        self.fc3 = nn.Linear(80, 10)

    def forward(self, x):
        x = x.view(-1,28*28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```



Specify a Model Architecture : CNN Model

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1   = nn.Linear(16*5*5, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        out = F.relu(self.conv1(x))
        out = F.max_pool2d(out, 2)
        out = F.relu(self.conv2(out))
        out = F.max_pool2d(out, 2)
        out = out.view(out.size(0), -1)
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out
```



Specify Loss, Regularizer & Gradient Update Algo

```
criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

Train the model

- Initialize model with random parameters.
- Repeat
    a. Take a small random subset of the dataset that will fit in memory (minibatch) and normailize it.
    b. Forward Pass: pass the subset through the model and obtain predictions
    c. Compute the mean loss function for the subset
    d. Backward Pass: compute the gradients of the parameters, last layer to the first,update the gradients using learning rate
    e. Plot loss

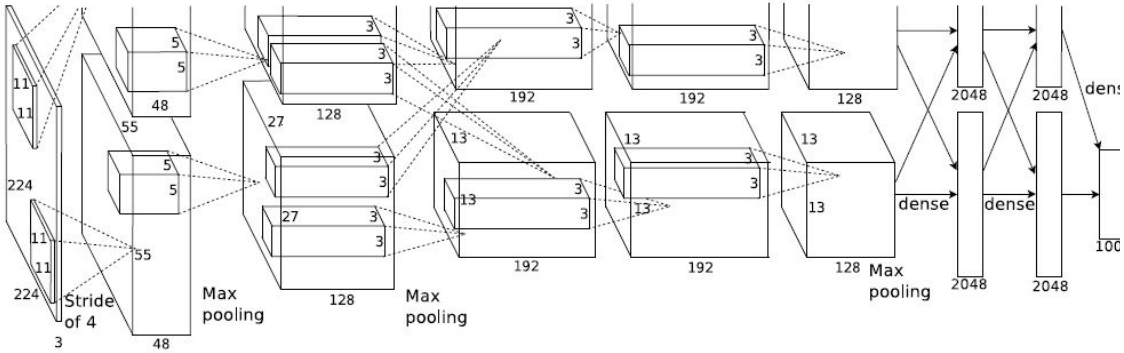
Hyperparameter optimization

- Number of layers, hidden units, filters of CNN are fixed while specifying the architecture.
- They are not learnt using backpropagation.
- How to choose?
  - Trial and error!
  - Look at datasets having the same complexity and models which work well there. Start with parameters there.

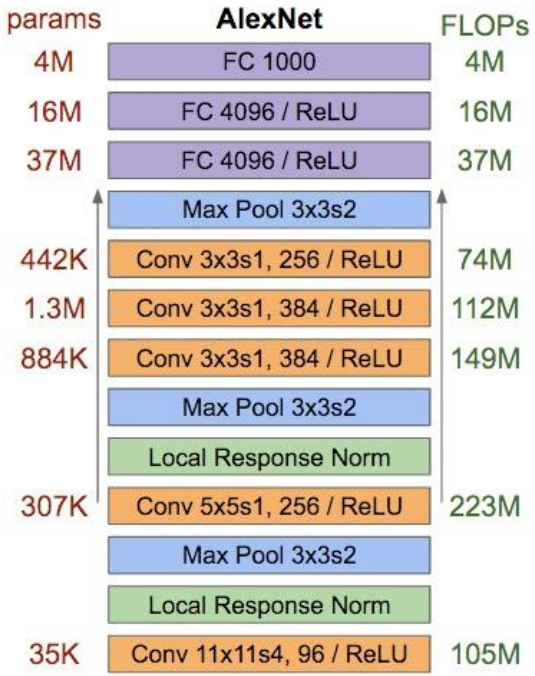
Demo

Deploying Deep Learning Models: Model Compression

Big Huge Neural Network!

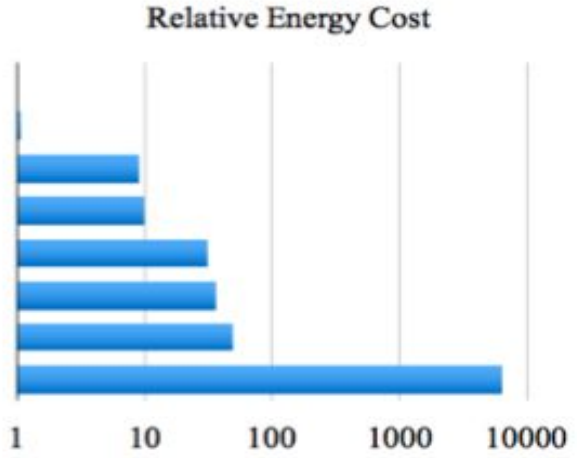


AlexNet - 60 Million Parameters = 240 MB



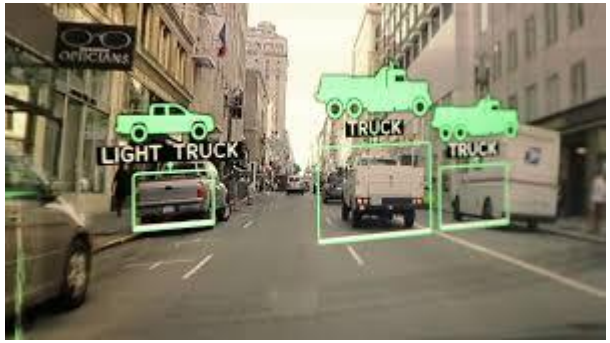
& the Humble Mobile Phone

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit Register File	1	10
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit SRAM Cache	5	50
32 bit DRAM Memory	**640**	**6400**



But wait! What about battery life?

Self Driving Cars!



Can we do 30 fps?

ORCAM! : Blind AID

Can we do 30 fps?

Running Model in the Cloud

1. Network Delay
2. Power Consumption
3. User Privacy

Issues on Mobile Devices

1. RAM Memory Usage
2. Running Time
3. Power Usage
4. Download / Storage size

Model Compression

What are Neural Networks made of?

- Fully Connected Layer : Matrices

- Convolutional Layer : Kernals (Tensors)

Reducing Memory Usage

1. Compressing Matrices
    a. Sparse Matrix => Special Storage formats
    b. Quantization
2. Architecture

Neural Network Algorithms

1. Matrix Multiplications
2. Convolutions

PRUNING

Compressing Matrices by making them Sparse

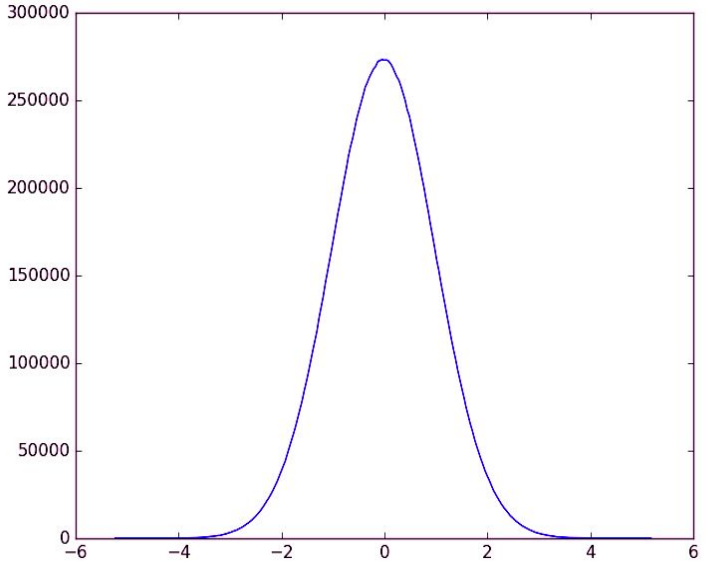
WHY PRUNING ?

Deep Neural Networks have redundant parameters.

Parameters having negligible value and can be ignored.

Removing them does not affect performance.

Figure: Distribution of weights after Training



Why do you need redundant parameters?
Redundant parameters are needed for training to converge to a good optima.

Optimal Brain Damage by Yann Le Cunn in 90's
https://papers.nips.cc/paper/250-optimal-brain-damage

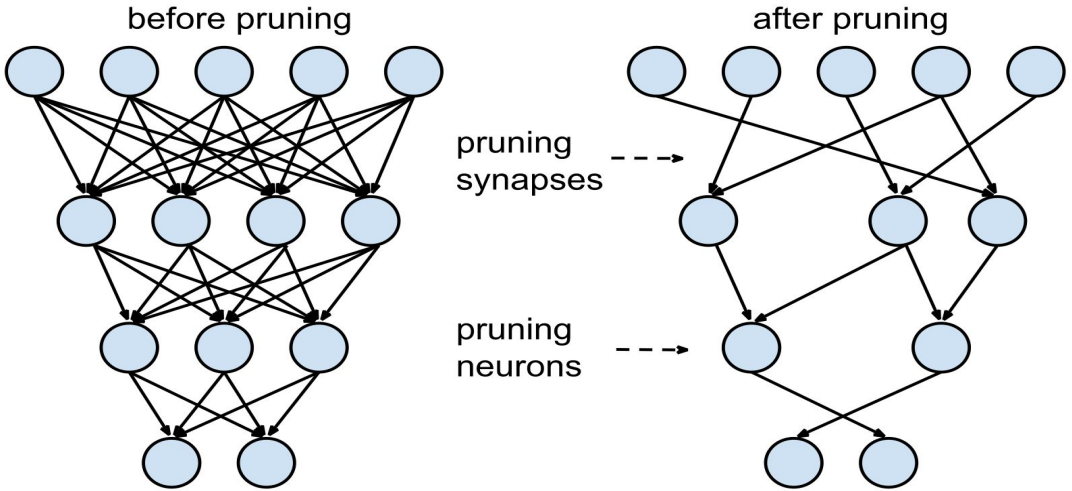
TYPES OF PRUNING

- Fine Pruning : Prune the weights
- Coarse Pruning : Prune neurons and layers

- Static Pruning : Pruning after training
- Dynamic Pruning : Pruning during training time

Weight Pruning

(After Training)

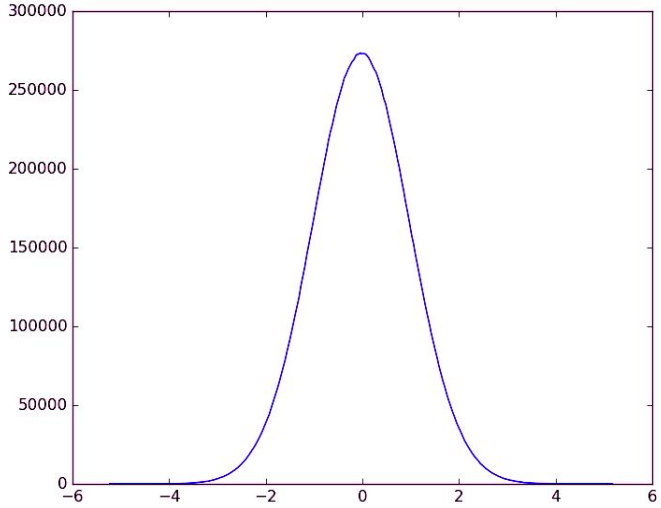


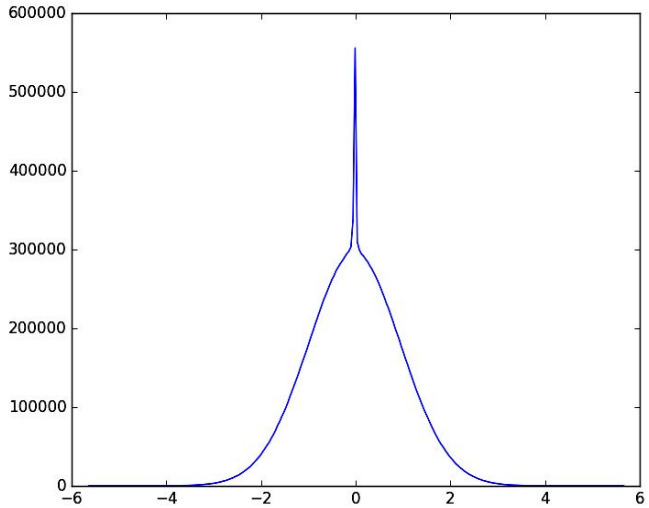
- The matrices can be made sparse. A naive method is to drop those weights which are 0 after training.
- Drop the weights below some threshold.
- Can be stored in optimized way if matrix becomes sparse.
- Sparse Matrix Multiplications are faster.

Ensuring Sparsity

Addition of L1 regulariser to ensure sparsity.

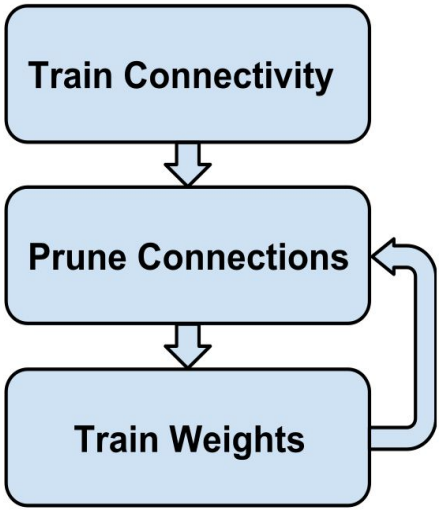
L1 better than L2 because of the slope near 0.

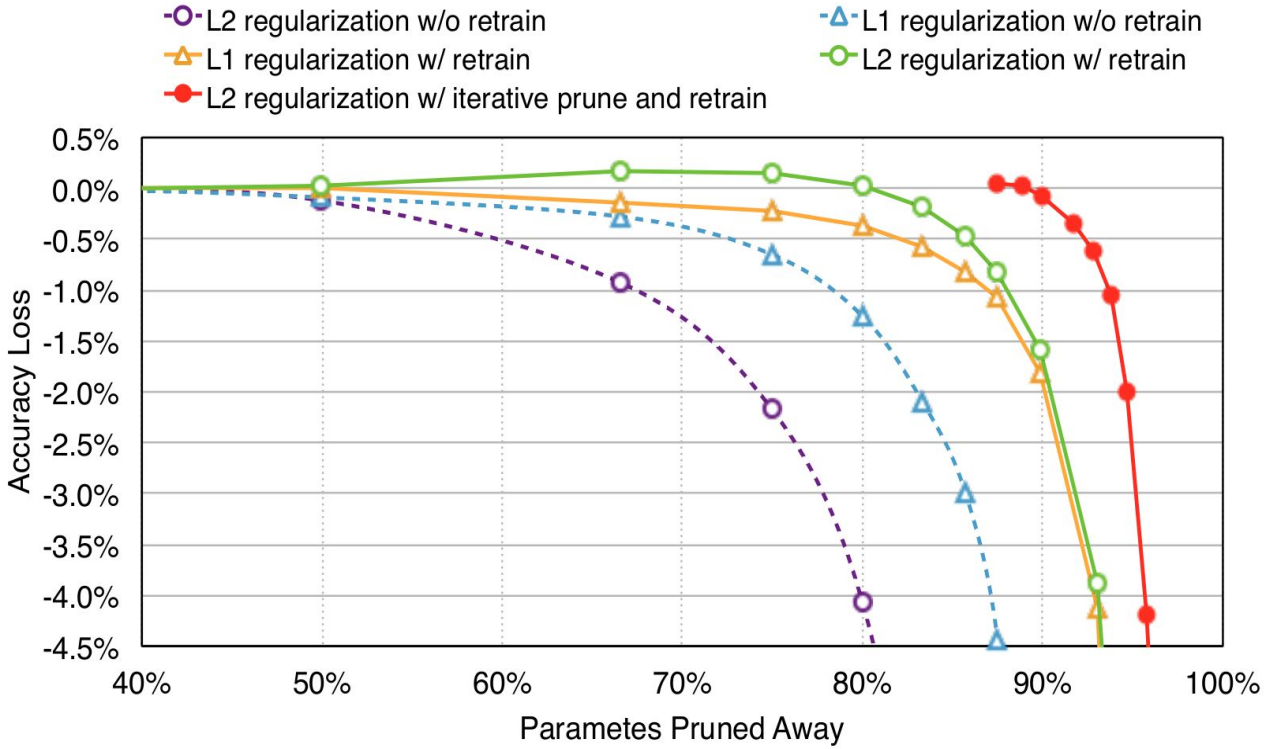




Sparsify at Training Time

Iterative pruning and retraining





Learning both Weights and Connections for Efficient Neural Networks

https://arxiv.org/pdf/1506.02626

by S Han - 2015 - Cited by 233 - Related articles

Table 4: For AlexNet, pruning reduces the number of weights by 9× and computation by 3×.

Layer	Weights	FLOP	Act%	Weights%	FLOP%
conv1	35K	211M	88%	84%	84%
conv2	307K	448M	52%	38%	33%
conv3	885K	299M	37%	35%	18%
conv4	663K	224M	40%	37%	14%
conv5	442K	150M	34%	37%	14%
fc1	38M	75M	36%	9%	3%
fc2	17M	34M	40%	9%	3%
fc3	4M	8M	100%	25%	10%
Total	61M	1.5B	54%	**11%**	**30%**

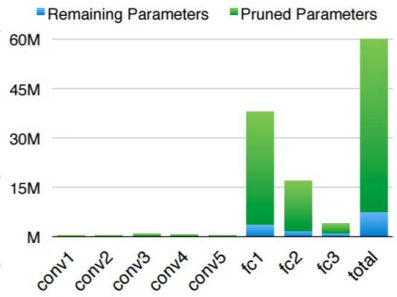
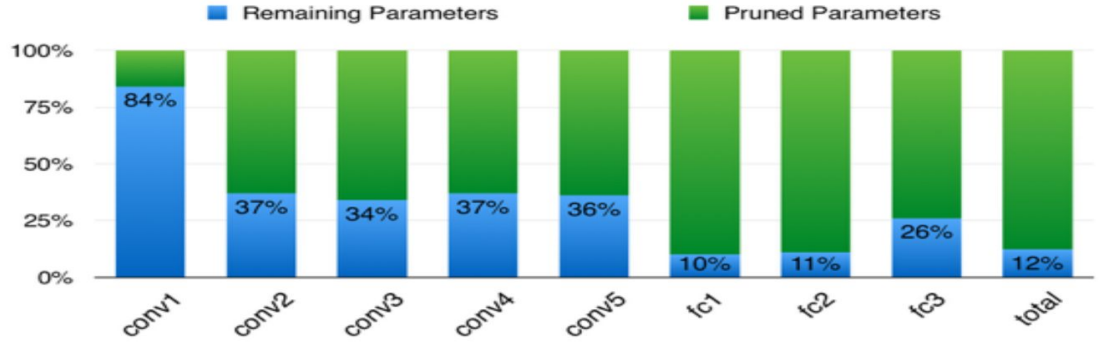


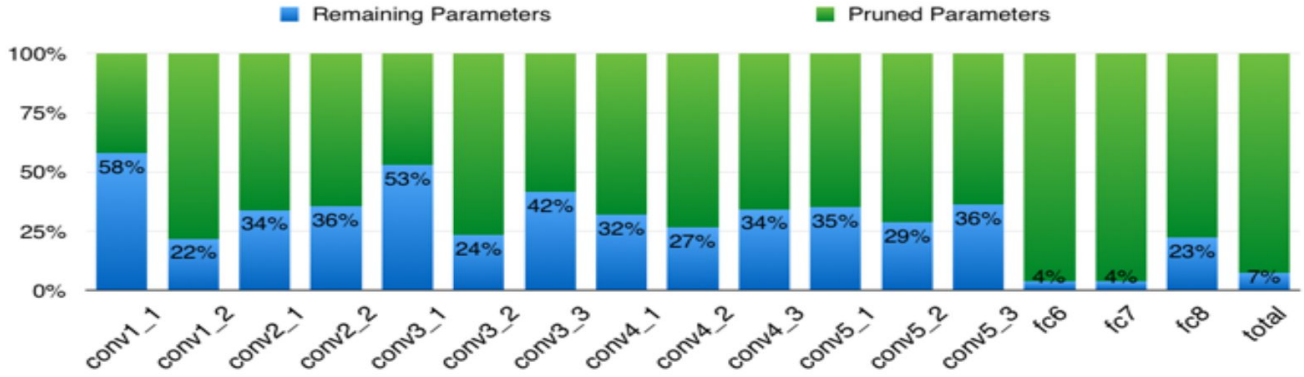
Table 5: For VGG-16, pruning reduces the number of weights by 12× and computation by 5×.

Layer	Weights	FLOP	Act%	Weights%	FLOP%
conv1_1	2K	0.2B	53%	58%	58%
conv1_2	37K	3.7B	89%	22%	12%
conv2_1	74K	1.8B	80%	34%	30%
conv2_2	148K	3.7B	81%	36%	29%
conv3_1	295K	1.8B	68%	53%	43%
conv3_2	590K	3.7B	70%	24%	16%
conv3_3	590K	3.7B	64%	42%	29%
conv4_1	1M	1.8B	51%	32%	21%
conv4_2	2M	3.7B	45%	27%	14%
conv4_3	2M	3.7B	34%	34%	15%
conv5_1	2M	925M	32%	35%	12%
conv5_2	2M	925M	29%	29%	9%
conv5_3	2M	925M	19%	36%	11%
fc6	103M	206M	38%	4%	1%
fc7	17M	34M	42%	4%	2%
fc8	4M	8M	100%	23%	9%
total	138M	30.9B	64%	**7.5%**	**21%**

Remaining parameters in Different Layers



ALEXNET



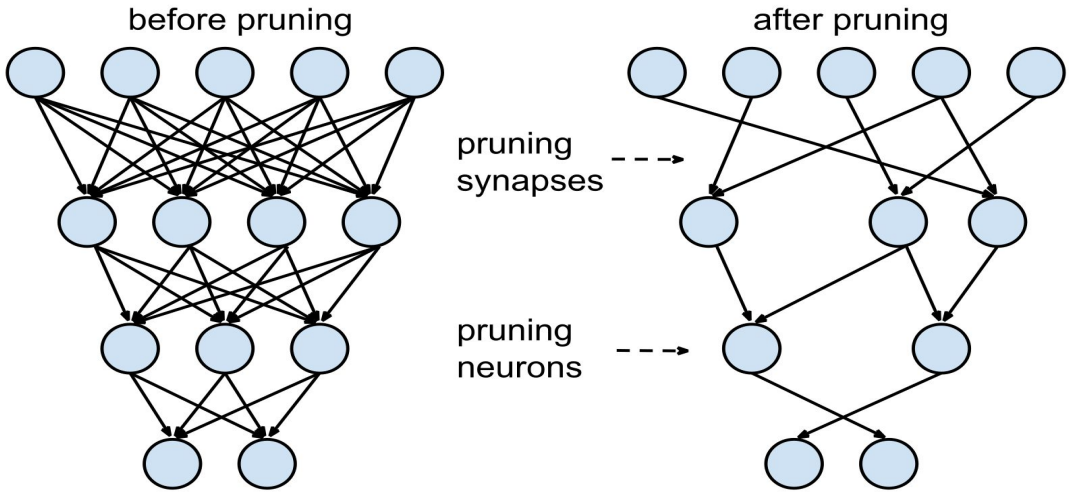
VGG16

DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING Song Han, Huizi Mao, William J. Dally

Comments on Weight Pruning

1. Matrices become sparse. Storage in HDD is efficient.
2. Same memory in RAM is occupied by the weight matrices.
3. Matrix multiplication is not faster since each 0 valued weight occupies as much space as before.
4. Optimized Sparse matrix multiplication algorithms need to be coded up separately even for a basic forward pass operation.

Neuron Pruning



➔ Removing rows and columns in a weight matrix.

➔ Matrix multiplication will be faster improving test time.

DropNeuron: Simplifying the Structure of Deep Neural Networks Wei Pan, Hao Dong, Yike Guo

Dropping Neurons by Regularization

$$\texttt{li\_regulariser} := \lambda_{\ell_i} \sum_{\ell=1}^{L} \sum_{j=1}^{n^\ell} \|\mathbf{W}^\ell_{:,j}\|_2 = \lambda_{\ell_i} \sum_{\ell=1}^{L} \sum_{j=1}^{n^\ell} \sqrt{\sum_{i=1}^{n^{\ell-1}} \left(W^\ell_{ij}\right)^2}$$

$$\texttt{lo\_regulariser} := \lambda_{\ell_o} \sum_{\ell=1}^{L} \sum_{i=1}^{n^{\ell-1}} \|\mathbf{W}^\ell_{i,:}\|_2 = \lambda_{\ell_o} \sum_{\ell=1}^{L} \sum_{i=1}^{n^{\ell-1}} \sqrt{\sum_{j=1}^{n^\ell} \left(W^\ell_{ij}\right)^2}$$

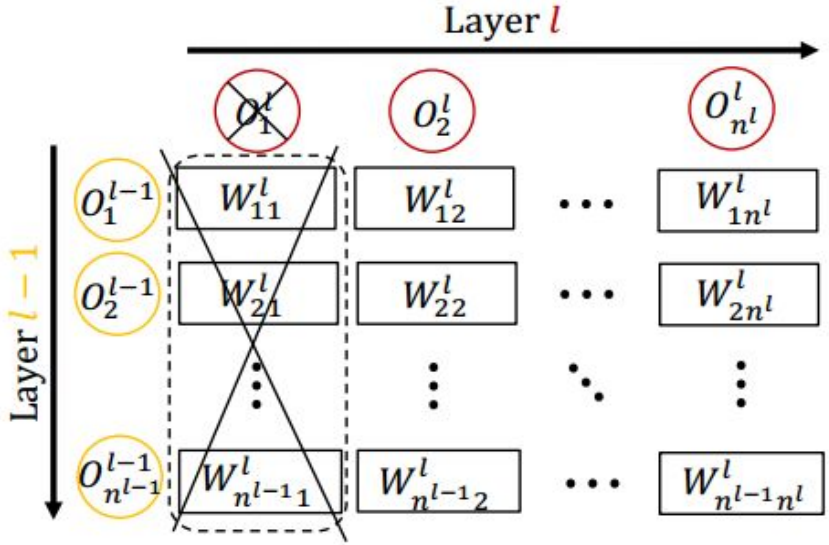
DropNeuron: Simplifying the Structure of Deep Neural Networks Wei Pan, Hao Dong, Yike Guo

Dropping principles

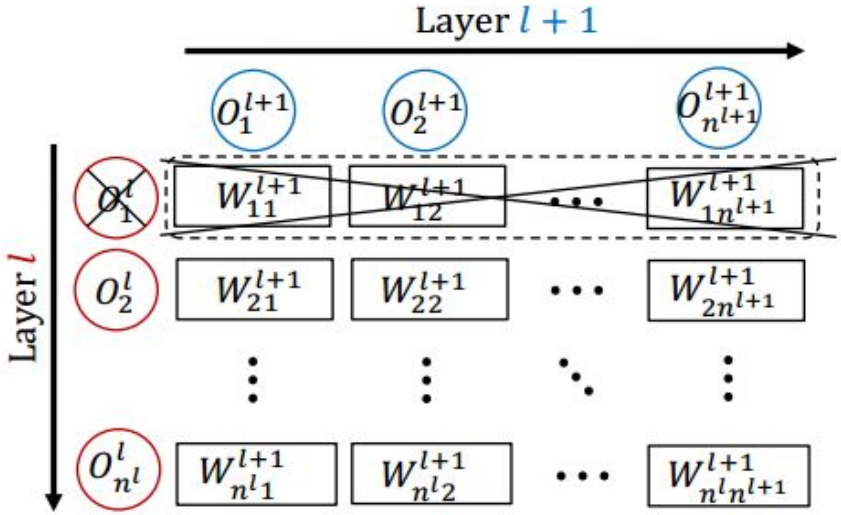
- All input connections to a neuron is forced to be 0 or as close to 0 as possible. (force li_regulariser to be small)
- All output connections of a neuron is forced to be 0 or as close to zero as possible. (force lo_regulariser to be small)
- Add regularisers to the loss function and train.
- Remove all connections less than threshold after training.
- Discard neuron with no connection.

DropNeuron: Simplifying the Structure of Deep Neural Networks Wei Pan, Hao Dong, Yike Guo

Effect of neuron pruning on weight matrices



(c) Removal of incoming connections to neuron O_1^ℓ, i.e., the group of weights in the dashed box are all zeros



(d) Removal of outgoing connections from neuron O_1^ℓ, i.e., the group of weights in the dashed box are all zeros

DropNeuron: Simplifying the Structure of Deep Neural Networks Wei Pan, Hao Dong, Yike Guo

Results on FC Layer (MNIST)

Table 3: Summary of statistics for the fully connect layer of LeNet5 (average over 10 initialisations)

Regularisation	$\mathbf{W}^{\text{FC1}}\%$	$\mathbf{W}^{\text{FC2}}\%$	$\mathbf{W}^{\text{total}}\%$	Accuracy	Accuracy (no prune)
DO+P	55.15%	62.81%	55.17%	99.07%	99.12%
ℓ_1+DO+P	5.42%	51.66%	5.57%	99.01%	98.96%
ℓ_1+DN+P	1.44%	16.82%	1.49%	99.07%	99.14%
Regularisation	$\mathbf{O}^{\text{FC1}}\%$	$\mathbf{O}^{\text{FC2}}\%$	$\mathbf{O}^{\text{output}}\%$	$\mathbf{O}^{\text{total}}\%$	**Compression Rate**
DO+P	$\frac{3136}{3136} = 100\%$	$\frac{504}{512} = 98.44\%$	$\frac{10}{10} = 100\%$	$\frac{3650}{3658} = 99.78\%$	1.81
ℓ_1+DO+P	$\frac{1039}{3136} = 33.13\%$	$\frac{320}{512} = 62.5\%$	$\frac{10}{10} = 100\%$	$\frac{1369}{3658} = 37.42\%$	17.95 [4]
ℓ_1+DN+P	$\frac{907}{3136} = 28.92\%$	$\frac{110}{512} = 21.48\%$	$\frac{10}{10} = 100\%$	$\frac{1027}{3658} = 28.08\%$	67.04

DropNeuron: Simplifying the Structure of Deep Neural Networks Wei Pan, Hao Dong, Yike Guo

QUANTIZATION

Binary Quantization

$$\hat{W}_{ij} = \begin{cases} 1 & \text{if } W_{ij} \geq 0, \\ -1 & \text{if } W_{ij} < 0. \end{cases}$$

Size Drop : 32X

Runtime : Much faster (7x) matrix multiplication for binary matrices.

Accuracy Drop : Classification error is about 20% on the top 5 accuracy on ILSVRC dataset.

COMPRESSING DEEP CONVOLUTIONAL NETWORKS USING VECTOR QUANTIZATION Yunchao Gong, Liu Liu , Ming Yang, Lubomir Bourdev

Binary Quantization while Training

- Add regularizer and round at the end of training

$$\sum_i W^2{}_i(1 - W^2{}_i)$$

Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or −1 Matthieu Courbariaux, Itay Hubara , Daniel Soudry , Ran El-Yaniv, Yoshua Bengio

8-bit uniform quantization

- Divide the max and min weight values into 256 equal divisions uniformly.
- Round weights to the nearest point
- Store weights as 8 bit ints

Size Drop : 4X

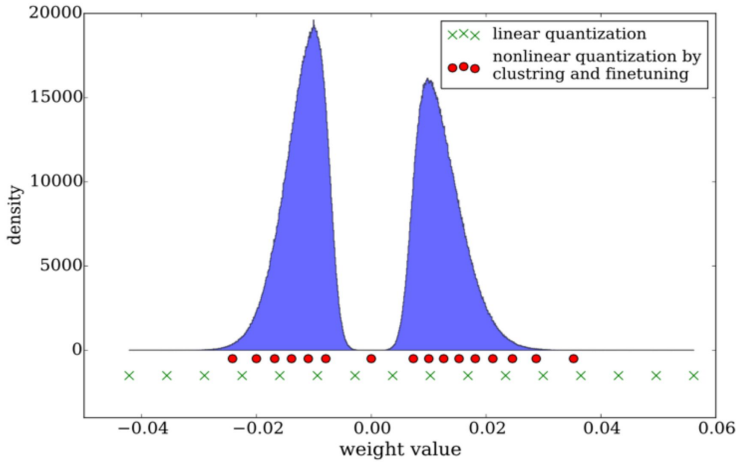
Runtime : Much faster matrix multiplication for 8 bit matrices.

Accuracy Drop : Error is acceptable for classification for non critical tasks

https://petewarden.com/2016/05/03/how-to-quantize-neural-networks-with-tensorflow/

Non Uniform Quantization/ Weight Sharing

$$\min \sum_{i}^{mn} \sum_{j}^{k} \|w_i - c_j\|_2^2,$$



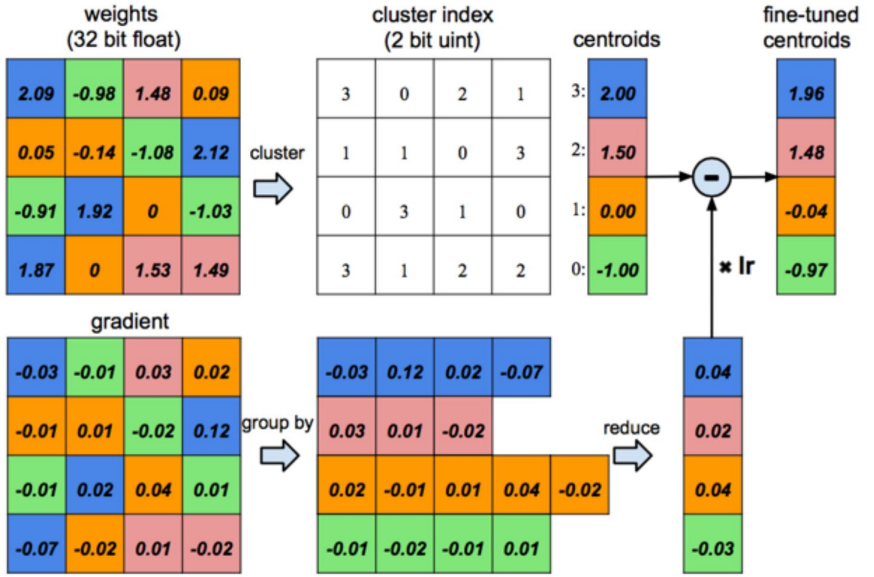
- perform k-means clustering on weights.
- Need to store mapping from integers to cluster centers. We only need log (k) bits to code the clusters which results in a compression factor rate of 32/ log (k). In this case the compression rate is 4.

DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING Song Han, Huizi Mao, William J. Dally

Weight Sharing while Training

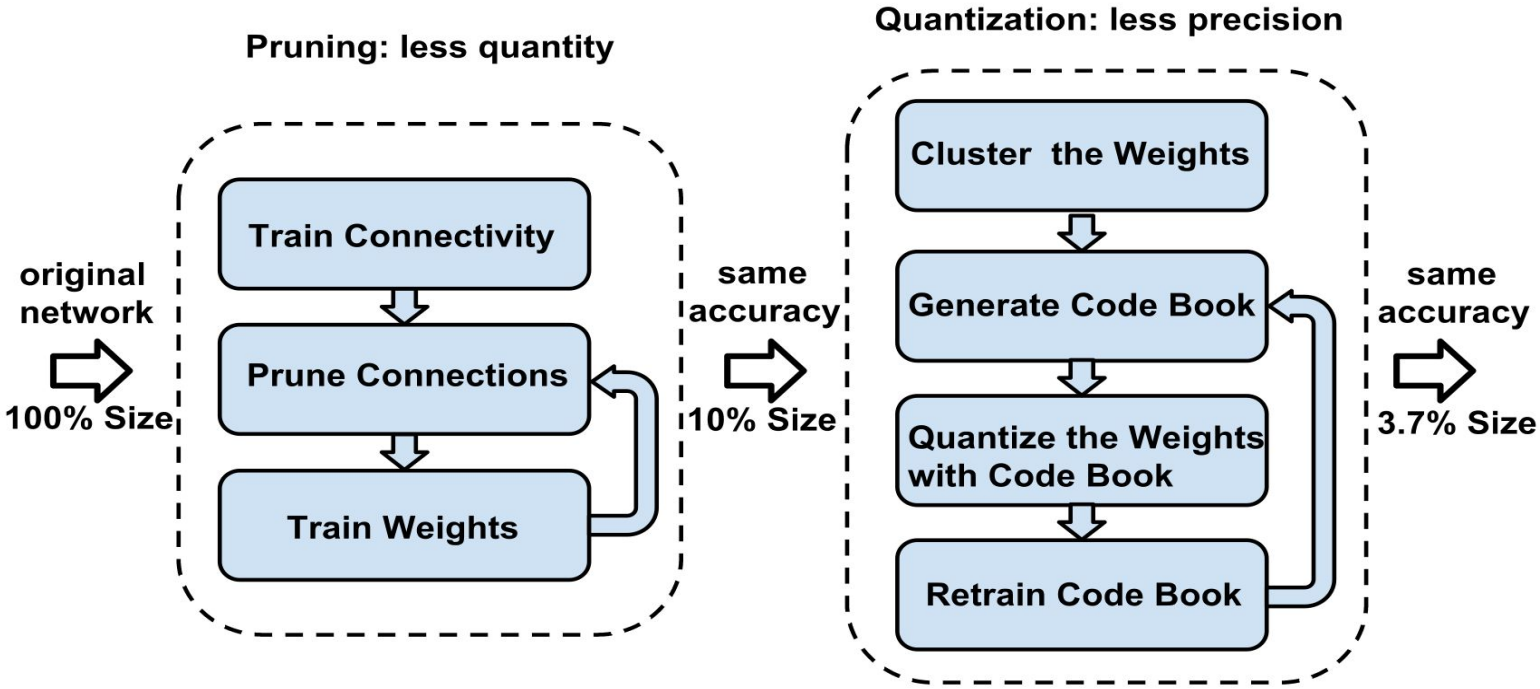
- Iterate
  - Train
  - Cluster weights
  - Make them same

- Compute gradients with respect to centroids so that weight sharing is preserved during gradient update.



DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING Song Han, Huizi Mao, William J. Dally

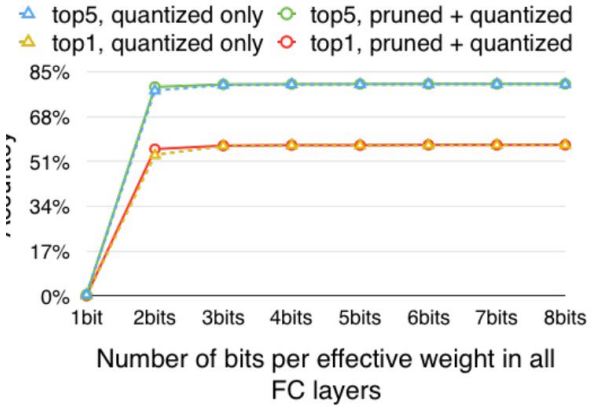
Deep Compression by Song Han

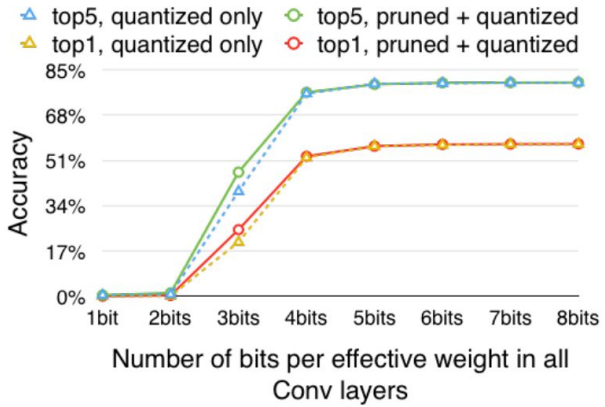


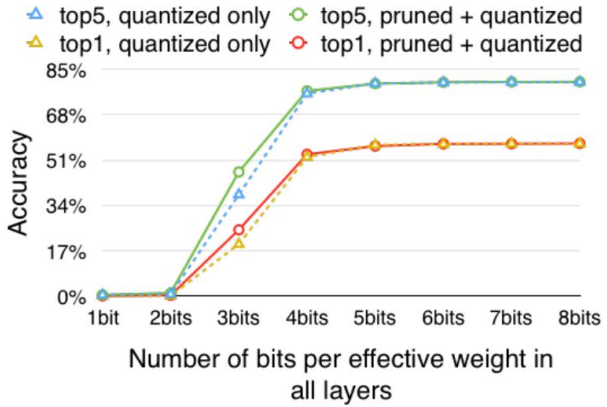
DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING Song Han, Huizi Mao, William J. Dally

Deep Compression by Song Han

Pruning and Quantization Works Well Together







DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING Song Han, Huizi Mao, William J. Dally

Product Quantization

Partition the given matrix into several submatrices and we perform k-means clustering for all of them.

$$W = [W^1, W^2, \ldots, W^s], \qquad \min \sum_{z}^{m} \sum_{j}^{k} \|\boldsymbol{w}_z^i - \boldsymbol{c}_j^i\|_2^2,$$

$$\hat{W} = [\hat{W}^1, \hat{W}^2, \ldots, \hat{W}^s], \quad \text{where}$$

$$\hat{\boldsymbol{w}}_j^i = \boldsymbol{c}_j^i, \quad \text{where} \quad \min_j \|\boldsymbol{w}_z^i - \boldsymbol{c}_j^i\|_2^2.$$

COMPRESSING DEEP CONVOLUTIONAL NETWORKS USING VECTOR QUANTIZATION Yunchao Gong, Liu Liu , Ming Yang, Lubomir Bourdev

Residual Quantization

First quantize the vectors into k-centers.

$$\min \sum_{z}^{m} \sum_{j}^{k} \|\boldsymbol{w}_z - \boldsymbol{c}_j^1\|_2^2,$$

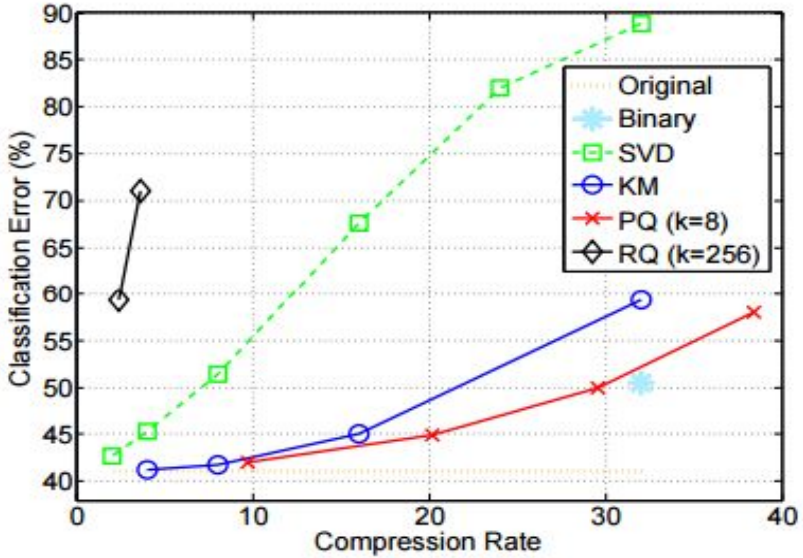
Next step is to find out the residuals for each data point(w-c) and perform k-means on the residuals

Then the resultant weight vectors are calculated as follows.

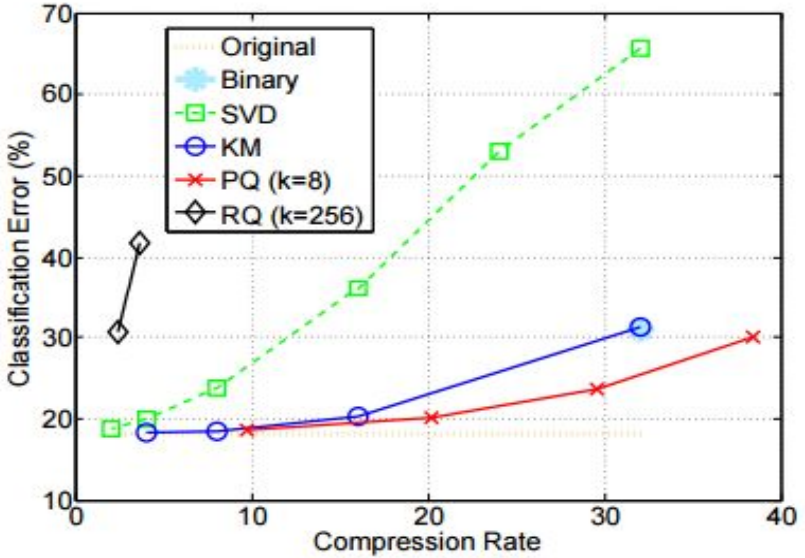
$$\hat{\boldsymbol{w}}_z = \boldsymbol{c}_j^1 + \boldsymbol{c}_j^2 + \ldots, \boldsymbol{c}_j^t,$$

COMPRESSING DEEP CONVOLUTIONAL NETWORKS USING VECTOR QUANTIZATION Yunchao Gong, Liu Liu , Ming Yang, Lubomir Bourdev

Comparison of Quantization methods on Imagenet



(a) Accuracy@1



(b) Accuracy@5

Figure 3: Comparison of different compression methods on ILSVRC dataset.

COMPRESSING DEEP CONVOLUTIONAL NETWORKS USING VECTOR QUANTIZATION Yunchao Gong, Liu Liu , Ming Yang, Lubomir Bourdev

FIXED POINT REPRESENTATION

FLOATING POINT VS FIXED POINT REPRESENTATION

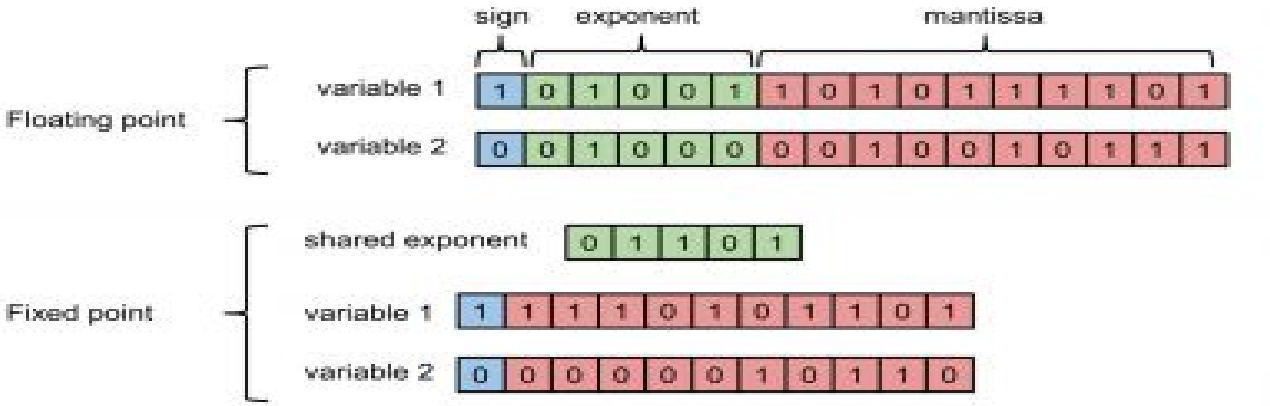


Figure 1: Comparison of the floating point and fixed point formats.

Fixed point

- ➔ Fixed point formats consist in a signed mantissa and a global scaling factor shared between all fixed point variables. It is usually ‘fixed’.
- ➔ Reducing the scaling factor reduces the range and augments the precision of the format.
- ➔ It relies on integer operations. It is hardware-wise cheaper than its floating point counterpart, as the exponent is shared and fixed.

Disadvantages of using fixed point

- When training deep neural networks :
  - Activations , gradients and parameters have very different ranges.
  - The ranges of the gradients slowly diminish during training.
  - Fixed point arithmetic is not optimised on regular hardware and specialised hardware such as FPGAs are required.

- As a result the fixed point format with its unique shared fixed exponent is ill-suited to deep learning.
- The dynamic fixed point format is a variant of the fixed point format in which there are several scaling factors instead of a single global one.

Summary

- Pruning weights and neurons
- Uniform Quantization
- Non Uniform Quantization / Weight Sharing

Global Average Pooling

Problems with fully connected (FC) layers:

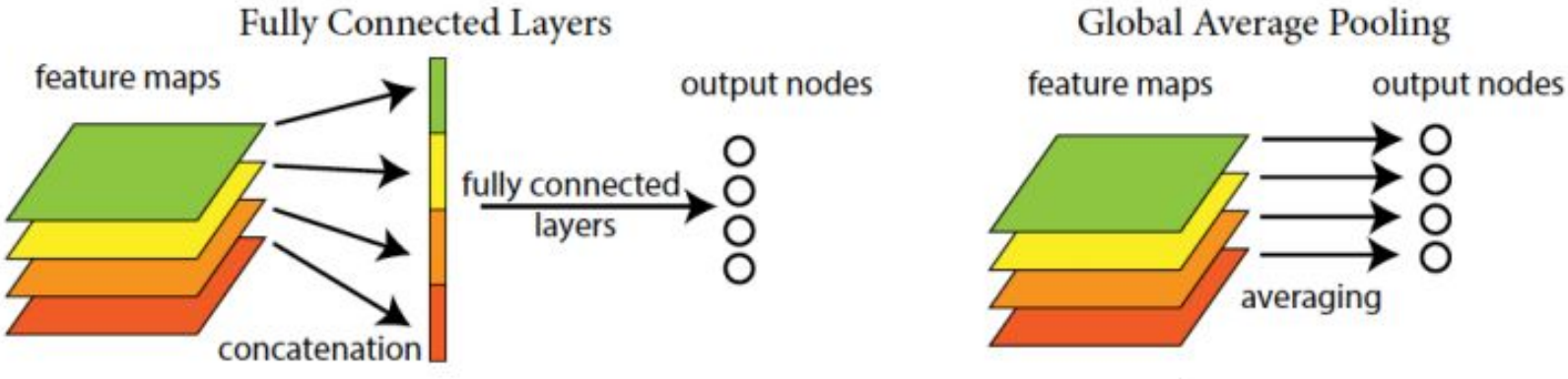
- More than 90% parameters of Alexnet and VGG are in the Fully Connected layers.
- One single particular layer in VGG contains 100 million parameters alone.
- Difficult to interpret how the category level information is passed back from the objective cost layer to the previous convolution layer.
- Prone to overfitting.
- Heavily dependent on regularization methods like dropout.

Global Average Pooling

Global Average Pooling as replacement to FC layers:

- An alternative is to use spatial average of feature maps.
- Huge reduction the number of parameters as compared to the Fully Connected layer.
- Enforces correspondence between feature maps and categories.
- Stronger local modelling using the micro network.
- It is itself a structural regularizer and hence doesn’t need dropout.
- We get a uniform sized feature vector irrespective of input image size.

Global Average Pooling



Demo : Model Compression