Neural Network Approximation

Low rank, Sparsity, and Quantization

zsc@megvii.com

Oct. 2017
Motivation

- **Faster Inference**
  - Latency critical scenarios
    - VR/AR, UGV/UAV
  - Saves time and energy

- **Faster Training**
  - Higher iteration speed
  - Saves life

- **Smaller**
  - Storage size
  - Memory footprint

Lee Sedol v.s. AlphaGo

83 Watt 77 kWatt
Neural Network
Machine Learning as Optimization

- **Supervised learning**
  - $\theta$ is the parameter
  - $\hat{y}$ is output, X is input
  - y is ground truth
  - d is the objective function
  - $\min_{\theta} d(y, \hat{y})$
    where $\hat{y} = f(X, \theta)$

- **Unsupervised learning**
  - some oracle function $r$: low rank, sparse, K
  - $\min_{\theta} r(\hat{y})$
    where $\hat{y} = f(X, \theta)$
Machine Learning as Optimization

● Regularized supervised learning
  ○ \( \min_\theta d(y, \hat{y}) + r(\hat{y}) \)
    where \( \hat{y} = f(X, \theta) \)

● Probabilistic interpretation
  ○ \( d \) measures conditional probability
  ○ \( r \) measures prior probability
  ○ Probability approach is more constrained than the optimization-approach due to normalization problem
    ■ Not easy to represent uniform distribution over \([0, \infty)\)

\[
\begin{align*}
  d(y, \hat{y}) &= -\log p(y|\hat{y}) \\
r(\hat{y}) &= -\log p(\hat{y})
\end{align*}
\]

\[
\Rightarrow d(y, \hat{y}) + r(\hat{y}) = -\log p(y)
\]
Gradient descent

\[ \min_\theta d(y, \hat{y}) + r(\hat{y}) \]
where \( \hat{y} = f(X, \theta) \)

- Can be solved by an ODE:
  - Discretizing with step length \( \lambda \) we get gradient descent with learning rate \( \lambda \)

  \[ \dot{\theta} \approx \frac{\theta_{t+\lambda} - \theta_t}{\lambda} \]

  Derive

  \[ \theta_{t+\lambda} = \theta_t - \lambda \frac{\partial C(\theta)}{\partial \theta} \]

- Convergence proof

\[ \frac{dC(\theta)}{dt} = \frac{\partial C(\theta)}{\partial \theta} \frac{d\theta}{dt} = -\left( \frac{\partial C(\theta)}{\partial \theta} \right)^2 \leq 0 \]
Linear Regression

\[ \min_W \| y - Wx \|^2 \]

- \( \hat{y} = f(X, \theta) = WX \)
- \( d(y, \hat{y}) = \| y - \hat{y} \|^2 \)
- \( x \) is input, \( \hat{y} \) is prediction, \( y \) is ground truth.
- \( W \) with dimension \((m,n)\)
- \#param = \( m \ n \), \#OPs = \( m \ n \)

\[ y \approx \hat{y} = Wx \]
Fully-connected

- \( y \approx W_2 f(W_1(x)) \)
  - In general, will use nonlinearity to increase “model capacity”.
  - Make sense if \( f \) is identity? I.e. \( f(x) = x \)?
    - Sometimes, if \( W_2 \) is \( m \) by \( r \) and \( W_1 \) is \( r \) by \( n \), then \( W_2 \) \( W_1 \) is a matrix of rank \( r \), which is different from a \( m \) by \( n \) matrix.

- \( y \approx W_3(f(W_2 f(W_1(x)))) \)
  - Deep learning!
Neural Network

\[ y \xrightarrow{\text{Cost}} \hat{y} \]

Activations/
Feature maps/
Neurons

Input Layer

Hidden Layers

Output Layer
Gradient descent

\[ \min_{\theta} d(y, \hat{y}) + r(\hat{y}) \]
where \( \hat{y} = f(X, \theta) \)

- Can be solved by an ODE:
  - Discretizing with step length \( \lambda \) we get gradient descent with learning rate \( \lambda \)
    \[
    \dot{\theta} \approx \frac{\theta_{t+\lambda} - \theta_t}{\lambda}
    \]
    Derive \( \theta_{t+\lambda} = \theta_t - \lambda \frac{\partial C(\theta)}{\partial \theta} \)

- Convergence proof
  \[
  \frac{dC(\theta)}{dt} = \frac{\partial C(\theta)}{\partial \theta} \frac{d\theta}{dt} = -\left( \frac{\partial C(\theta)}{\partial \theta} \right)^2 \leq 0
  \]
Backpropagation

\[
\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1} \frac{\partial w_1}{\partial x} = \left( \frac{\partial y}{\partial w_2} \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \left( \left( \frac{\partial y}{\partial w_3} \frac{\partial w_3}{\partial w_2} \right) \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \ldots
\]
Neural Network Training

\[ \hat{y} \quad \text{Cost} \quad d + r \quad y \]

Activations/
Feature maps/
Neurons

Output Layer

Hidden Layers

Input Layer

Gradients
CNN: Alexnet-like
Method 2: Convolution as matrix product

- Convolution
  - feature map \(<N, C, H', W'>\)
  - weights \(<K, C, H, W'>\)

- Convolution as FC
  - under proper padding, can extract patches
  - feature map \(<N H' W', C H W'>\)
  - weights \(<C H W, K>\)
Importance of Convolutions and FC

<table>
<thead>
<tr>
<th>param_name</th>
<th>shape</th>
<th>#floats</th>
<th>size</th>
<th>perc</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv0 W</td>
<td>(24, 3, 5, 5)</td>
<td>1800</td>
<td>7.0 KiB</td>
<td>0.21%</td>
</tr>
<tr>
<td>conv0 b</td>
<td>(24,)</td>
<td>24</td>
<td>96.0 KiB</td>
<td>0.00%</td>
</tr>
<tr>
<td>conv1 W</td>
<td>(32, 24, 3, 3)</td>
<td>6912</td>
<td>27.0 KiB</td>
<td>0.80%</td>
</tr>
<tr>
<td>conv1 b</td>
<td>(32,)</td>
<td>32</td>
<td>128.0 KiB</td>
<td>0.00%</td>
</tr>
<tr>
<td>conv2 W</td>
<td>(32, 32, 3, 3)</td>
<td>9216</td>
<td>36.0 KiB</td>
<td>1.07%</td>
</tr>
<tr>
<td>conv2 b</td>
<td>(32,)</td>
<td>32</td>
<td>128.0 KiB</td>
<td>0.00%</td>
</tr>
<tr>
<td>conv3 W</td>
<td>(64, 32, 3, 3)</td>
<td>18432</td>
<td>72.0 KiB</td>
<td>2.14%</td>
</tr>
<tr>
<td>conv3 b</td>
<td>(64,)</td>
<td>64</td>
<td>256.0 KiB</td>
<td>0.01%</td>
</tr>
<tr>
<td>fc0 W</td>
<td>(1600, 512)</td>
<td>819200</td>
<td>3.1 MiB</td>
<td>95.11%</td>
</tr>
<tr>
<td>fc0 b</td>
<td>(512,)</td>
<td>512</td>
<td>2.0 KiB</td>
<td>0.06%</td>
</tr>
<tr>
<td>fct W</td>
<td>(512, 10)</td>
<td>5120</td>
<td>20.0 KiB</td>
<td>0.59%</td>
</tr>
<tr>
<td>fct b</td>
<td>(10,)</td>
<td>10</td>
<td>40.0 KiB</td>
<td>0.00%</td>
</tr>
<tr>
<td>total size</td>
<td></td>
<td>861354</td>
<td>3.3 MiB</td>
<td></td>
</tr>
</tbody>
</table>

Neupack: inspect_model.py
NeuPeak: npk-model-manip XXX info

Most storage size
Feature map size
Most Computation
The Matrix View of Neural Network

- Weights of FullyConnected and Convolutions layers
  - take up most computation and storage size
  - are representable as matrices

- Approximating the matrices approximates the network
  - The approximation error accumulates.

\[ W_a \approx W \Rightarrow f(X, W_a) \approx f(X, W) \]
Low rank Approximation
Singular Value Decomposition

- Matrix decomposition view
  - $A = U S V^T$
  - Rows of $U$, $V$ are orthogonal. $S$ is diagonal.
    - $u$, $s$, $v^T = \text{np.linalg.svd}(x, \text{full_matrices}=0, \text{compute_uv}=1)$
    - The diagonals are non-negative and are in descending order.
    - $U^T U = I$, but $U U^T$ is not full rank

$A_{m \times n} = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \begin{bmatrix} \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$

Compact SVD
Truncated SVD

- Assume diagonals of S are in descending order
  - Always achievable
  - Just ignore the blue segments.

\[ A_k = U_k \Sigma_k V_k^T \]
1: Comparison between approximations by outer product and Kronecker product for an image. The column (a) is the origin image of size $480 \times 320$, selected from BSD500 dataset Arbelaez et al. (2011). The column (b) is the SVD approximations of (a) by outer product and the column (c) is the approximation based on Kronecker product Van Loan and Pitsianis (1993), with rank 1, 2, 5, 10 respectively from top to down. The shape of the right matrix in the Kronecker product is deliberately selected as $20 \times 16$ to make the number of parameters equal for each rank.
Matrix factorization => Convolution factorization

- Factorization into H x W followed by 1 x 1
  - feature map (N H' W', C H W)
  - first conv weights (C H W, R)
  - feature map (N H' W', R)
  - second conv weights (R, K)
  - feature map (N H' W', K)
Approximating Convolution Weight

- $W$ is a $(K, C, H, W)$ 4-tensor
  - can be reshaped to a $(CHW, K)$ matrix, etc.
- F-norm is invariant under reshape
  - $\|M - M_a\|_F = \|\text{reshape}(W) - \text{reshape}(W_a)\|_F = \|W - W_a\|_F$
Matrix factorization => Convolution factorization

- Factorization into 1x1 followed by HxW
  - feature map (N H’ W’ H W, C)
  - first conv weights (C, R)
  - feature map (N H’ W’ H W, R) = (N H’ W’, R H W)
  - second conv weights (R H W, K)
  - feature map (N H’ W’, K)

- Steps
  - Reshape (CHW, K) to (C, HW, K)
  - (C, HW, K) = (C, R) (R, HW, K)
  - Reshape (R, HW, K) to (RHW, K)
Horizontal-Vertical Decomposition

- Approximating with Separable Filters
- Original Convolution
  - feature map \((N \ H' \ W', \ C \ H \ W)\)
  - weights \((C \ H \ W, K)\)
- Factorization into Hx1 followed by 1xW
  - feature map \((N \ H' \ W' \ W, \ C \ H)\)
  - first conv weights \((C \ H, R)\)
  - feature map \((N \ H' \ W' \ W, R) = (N \ H' \ W', R \ W)\)
  - second conv weights \((R \ W, K)\)
  - feature map \((N \ H' \ W', K)\)
- Steps
  - Reshape \((CHW, K)\) to \((CH, WK)\)
  - \((CH, WK) = (CH, R) (R, WK)\)
  - Reshape \((R, WK)\) to \((RW, K)\)
Factorizing N-D convolution

- Original Convolution
  - let dimension be N
  - feature map \((N \times D'_1 \times D'_2 \ldots \times D'_Z, C \times D_1 \times D_2 \ldots \times D_N)\)
  - weights \((C \times D_1 \times D_2 \ldots \times D_N, K)\)

- Factorization into N number of \(D_i \times 1\)
  - \(R_0 = C, R_Z = K\)
  - feature map \((N \times D'_1 \times D'_2 \ldots \times D'_Z, C \times D_1 \times D_2 \ldots \times D_N)\)
  - weights \((R_0 \times D_1, R_1)\)
  - feature map \((N \times D'_1 \times D'_2 \ldots \times D'_Z, R_1 \times D_2 \ldots \times D_N)\)
  - weights \((R_1 \times D_2, R_2)\)
  - ...
SVD

\[ M = \sum_{i=1}^{r} \sigma_i u_i \otimes v_i \]

Kronecker Product

KPSVD:

\[ A \approx A_r = \sum_{i=1}^{r} \sigma_i u_i \otimes v_i. \]
Kronecker Conv

- \((C \cdot H \cdot W, K)\)
- Reshape as \((C_1 \cdot C_2 \cdot H \cdot W, K_1 \cdot K_2)\)
- Steps
  - Feature map is \((N \cdot C \cdot H' \cdot W')\)
  - Extract patches and reshape \((N \cdot H' \cdot W' \cdot C_2, C_1 \cdot H)\)
  - apply \((C_1 \cdot H, K_1 \cdot R)\)
  - Feature map is \((N \cdot K_1 \cdot R \cdot H' \cdot W' \cdot C_2)\)
  - Extract patches and reshape \((N \cdot K_1 \cdot H' \cdot W', R \cdot C_2 \cdot W)\)
  - apply \((R \cdot C_2 \cdot W, K_2)\)
- For rank efficiency, should have
  - \(R \cdot C_2 \approx C_1\)
Exploiting Local Structures with the Kronecker Layer in Convolutional Networks

<table>
<thead>
<tr>
<th>Methods</th>
<th>Configuration ((r, o_1, c_1, h_1, w_1))</th>
<th>Validation Error</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>/</td>
<td>7.84%</td>
<td>/</td>
</tr>
<tr>
<td>KConv-a</td>
<td>1,128,24,9,1; 1,256,64,8,1</td>
<td>8.76%</td>
<td>3.3×</td>
</tr>
<tr>
<td>KConv-b</td>
<td>1,128,48,1,9; 1,512,64,1,8</td>
<td>8.69%</td>
<td>3.0×</td>
</tr>
<tr>
<td>KConv-c</td>
<td>2,64,24,9,1; 2,256,64,8,1</td>
<td>7.87%</td>
<td>2.9×</td>
</tr>
</tbody>
</table>

We have also experimented replacing the first convolutional layer with KConv layer. In this case, KConv with \((r, o_1, c_1, h_1, w_1) = (2, 12, 1, 1, 9)\), is found to outperform Jaderberg-style rank-1 filter with \((r, o_1, c_1, h_1, w_1) = (2, 96, 1, 1, 9)\) by 0.83%.
Shared Group Convolution is a Kronecker Layer

AlexNet partitioned a conv

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
CP-decomposition and Xception

- **Xception**: Deep Learning with Depthwise Separable Convolutions 1610
- **CP-decomposition with Tensor Power Method for Convolutional Neural Networks Compression** 1701
- **MobileNets**: Efficient Convolutional Neural Networks for Mobile Vision Applications 1704
  - They submitted the paper to CVPR about the same time as Xception.

<table>
<thead>
<tr>
<th>Layer/Modification</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>462</td>
<td>2.36</td>
</tr>
<tr>
<td>Depthwise Separable Conv</td>
<td>52.3</td>
<td>0.27</td>
</tr>
<tr>
<td>$\alpha = 0.75$</td>
<td>29.6</td>
<td>0.15</td>
</tr>
<tr>
<td>$\rho = 0.714$</td>
<td>15.1</td>
<td>0.15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5. Narrow vs Shallow MobileNet</th>
<th>Model</th>
<th>ImageNet Accuracy</th>
<th>Million Mult-Adds</th>
<th>Million Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.75 MobileNet</td>
<td>68.4%</td>
<td>325</td>
<td>2.6</td>
<td></td>
</tr>
<tr>
<td>Shallow MobileNet</td>
<td>65.3%</td>
<td>307</td>
<td>2.9</td>
<td></td>
</tr>
</tbody>
</table>
Matrix Joint Diagonalization = CP
CP-decomposition with Tensor Power Method for Convolutional Neural Networks Compression

Convolution

Xception

Filter size: \( T \times S \times D \times D \)

Filter size: \( R \times S \times 1 \times 1 \)  \( R \times 1 \times D \times D \)  \( T \times R \times 1 \times 1 \)

Channel-wise
Tensor Train Decomposition

Oseledets, 2009:

\[ A(i_1, i_2, \ldots, i_d) \approx \sum_{\alpha_1, \alpha_2, \ldots, \alpha_{d-1}} G_1(i_1, \alpha_1) G_2(\alpha_1, i_2, \alpha_2) \ldots G_d(\alpha_{d-1}, i_d) \]

\[ f = x_1 + x_2 + \ldots + x_d \]
Canonical rank: \(d\)
TT-rank: 2

\[ f = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_{d-1} & 1 \\ x_d & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{d-1} \\ x_d \end{pmatrix} \]

\[ f(x_1, \ldots, x_d) = \sin(x_1 + x_2 + \ldots + x_d) \]
FTT-decomposition has form

\[ f = \begin{pmatrix} \sin x_1 & \cos x_1 \\ \sin x_2 & \cos x_2 \\ \vdots & \vdots \\ \sin x_{d-1} & \cos x_{d-1} \end{pmatrix} \begin{pmatrix} \cos x_2 & -\sin x_2 \\ \sin x_2 & \cos x_2 \\ \vdots & \vdots \\ -\sin x_{d-1} & \cos x_{d-1} \end{pmatrix} \begin{pmatrix} \cos x_d \\ \sin x_d \end{pmatrix}. \]
Tensor Train Decomposition: just a few SVD’s

Require: \( d \)-dimensional tensor \( A \) required accuracy \( \varepsilon \)
Ensure: Cores \( G_1, \ldots, G_d \) of the TT-approximation \( B \) to \( A \) in the TT-format with smallest possibles compression ranks \( \hat{r}_k \) such that
\[
\|A - B\|_F \leq \varepsilon \|A\|_F,
\]

\{Initialization\}
Compute truncation parameter \( \delta = \frac{\varepsilon}{\sqrt{d-1}} \|A\|_F \).

Temporary tensor: \( C = A \).
\( N = \text{numel}(A), r_0 = 1. \)
for \( k = 1 \) to \( d - 1 \) do
\( C := \text{reshape}(C, [r_{k-1} n_k, \frac{N}{r_{k-1} n_k}]) \).

Compute \( \delta \)-truncated SVD: \( C = USV + E, \|E\|_F \leq \delta, \quad r_k = \text{rank}_\delta (C) \).
New core: \( G_k := \text{reshape}(U, [r_{k-1}, n_k, r_k]) \).
\( C := SV^T \).
\( N := \frac{Nr_k}{n_k r_{k-1}} \).
end for
\( G_d = C \).
Return tensor \( B \) in TT-format with cores \( G_1, \ldots, G_d \).
Tensor Train Decomposition on FC

<table>
<thead>
<tr>
<th>Type</th>
<th>1 im. time (ms)</th>
<th>100 im. time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU fully-connected layer</td>
<td>16.1</td>
<td>97.2</td>
</tr>
<tr>
<td>CPU TT-layer</td>
<td>1.2</td>
<td>94.7</td>
</tr>
<tr>
<td>GPU fully-connected layer</td>
<td>2.7</td>
<td>33</td>
</tr>
<tr>
<td>GPU TT-layer</td>
<td>1.9</td>
<td>12.9</td>
</tr>
</tbody>
</table>

Table 3: Inference time for a $25088 \times 4096$ fully-connected layer and its corresponding TT-layer with all the TT-ranks equal 4. The memory usage for feeding forward one image is 392MB for the fully-connected layer and 0.766MB for the TT-layer.
Graph Summary of SVD variants

**EXTENDED TT DECOMPOSITION**

NUMBER OF EXTENDED TT PARAMETERS $= dnr + (d-2)r^3$

**TUCKER DECOMPOSITION**

$a(i_1, ..., i_d) = \sum_{\alpha_1, ..., \alpha_{d-1}} g_1(i_1, \alpha_1) g_2(\alpha_1, i_2, \alpha_2) \ldots$  

$\ldots g_{d-1}(\alpha_{d-2}, i_{d-1}, \alpha_{d-1}) g_d(\alpha_{d-1}, i_d)$

**TENSOR-TRAIN DECOMPOSITION**

$a(i_1, ..., i_d) = \sum_{\alpha} u_1(i_1, \alpha) \ldots u_d(i_d, \alpha)$
CNN layers as Multilinear Maps

Convolution: $A_{k,c} X_{n,c,h,w,h_k,w_k}$

FC: $A_{k,c} X_{n,c,h,w,1,1}$

Xception: $A_{k,c} \delta_{k,c} X_{n,c,h,w,h_k,w_k}$

Kronecker: $A_{k_2,c_2} X_{n,c_1,c_2,h,w,h_k,w_k}$
Sparse Approximation
Distribution of Weights

- Universal across convolutions and FC
- Concentration of values near 0
- Large values *cannot* be dropped
Sparsity of NN: statistics
Sparsity of NN: statistics
Weight Pruning: from *DeepCompression*

The model has been trained with excessive #epoch.
Sparse Matrix at Runtime

- Sparse Matrix = Discrete Mask + Continuous values
  - Mask cannot be learnt the normal way
  - The values have well-defined gradients
- The matrix value look up need go through a LUT
  - CSR format
    - A: NNZ values
    - IA: accumulated #NNZ of rows
    - JA: the column in the row

```
A = [ 5 8 3 6 ]
IA = [ 0 0 2 3 4 ]
JA = [ 0 1 2 1 ]
```
Burden of Sparseness

- *Lost of regularity of memory access and computation*
  - Need special hardware for efficient access
  - May need high zero ratio to match dense matrix
  - Matrices will less than 70% zero values, better to treat as dense matrices.

(a) BCSR.  (b) VBL.  (c) CSX.
Convolution layers are harder to compress than FC


<table>
<thead>
<tr>
<th>Layer</th>
<th>#Weights</th>
<th>Weights% (P)</th>
<th>Weights bits (P+Q)</th>
<th>Weight bits (P+Q+H)</th>
<th>Index bits (P+Q)</th>
<th>Index bits (P+Q+H)</th>
<th>Compress rate (P+Q)</th>
<th>Compress rate (P+Q+H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1.1</td>
<td>2K</td>
<td>58%</td>
<td>8</td>
<td>6.8</td>
<td>5</td>
<td>1.7</td>
<td>40.0%</td>
<td>29.97%</td>
</tr>
<tr>
<td>conv1.2</td>
<td>37K</td>
<td>22%</td>
<td>8</td>
<td>6.5</td>
<td>5</td>
<td>2.6</td>
<td>9.8%</td>
<td>6.99%</td>
</tr>
<tr>
<td>conv2.1</td>
<td>74K</td>
<td>34%</td>
<td>8</td>
<td>5.6</td>
<td>5</td>
<td>2.4</td>
<td>14.3%</td>
<td>8.91%</td>
</tr>
<tr>
<td>conv2.2</td>
<td>148K</td>
<td>36%</td>
<td>8</td>
<td>5.9</td>
<td>5</td>
<td>2.3</td>
<td>14.7%</td>
<td>9.31%</td>
</tr>
<tr>
<td>conv3.1</td>
<td>295K</td>
<td>53%</td>
<td>8</td>
<td>4.8</td>
<td>5</td>
<td>1.8</td>
<td>21.7%</td>
<td>11.15%</td>
</tr>
<tr>
<td>conv3.2</td>
<td>590K</td>
<td>24%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.9</td>
<td>9.7%</td>
<td>5.67%</td>
</tr>
<tr>
<td>conv3.3</td>
<td>590K</td>
<td>42%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.2</td>
<td>17.0%</td>
<td>8.96%</td>
</tr>
<tr>
<td>conv4.1</td>
<td>1M</td>
<td>32%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.6</td>
<td>13.1%</td>
<td>7.29%</td>
</tr>
<tr>
<td>conv4.2</td>
<td>2M</td>
<td>27%</td>
<td>8</td>
<td>4.2</td>
<td>5</td>
<td>2.9</td>
<td>10.9%</td>
<td>5.93%</td>
</tr>
<tr>
<td>conv4.3</td>
<td>2M</td>
<td>34%</td>
<td>8</td>
<td>4.4</td>
<td>5</td>
<td>2.5</td>
<td>14.0%</td>
<td>7.47%</td>
</tr>
<tr>
<td>conv5.1</td>
<td>2M</td>
<td>35%</td>
<td>8</td>
<td>4.7</td>
<td>5</td>
<td>2.5</td>
<td>14.3%</td>
<td>8.00%</td>
</tr>
<tr>
<td>conv5.2</td>
<td>2M</td>
<td>29%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.7</td>
<td>11.7%</td>
<td>6.52%</td>
</tr>
<tr>
<td>conv5.3</td>
<td>2M</td>
<td>36%</td>
<td>8</td>
<td>4.6</td>
<td>5</td>
<td>2.3</td>
<td>14.8%</td>
<td>7.79%</td>
</tr>
<tr>
<td>fc6</td>
<td>103M</td>
<td>4%</td>
<td>5</td>
<td>3.6</td>
<td>5</td>
<td>3.5</td>
<td>1.6%</td>
<td>1.10%</td>
</tr>
<tr>
<td>fc7</td>
<td>17M</td>
<td>4%</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4.3</td>
<td>1.5%</td>
<td>1.25%</td>
</tr>
<tr>
<td>fc8</td>
<td>4M</td>
<td>23%</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3.4</td>
<td>7.1%</td>
<td>5.24%</td>
</tr>
<tr>
<td>Total</td>
<td>138M</td>
<td>7.5% (13×)</td>
<td>6.4</td>
<td>4.1</td>
<td>5</td>
<td>3.1</td>
<td>3.2% (31×)</td>
<td>2.05% (49×)</td>
</tr>
</tbody>
</table>
Dynamic Generation of Code

- **CVPR’15: Sparse Convolutional Neural Networks**
- Relies on compiler for
  - register allocation
  - scheduling
- Good on CPU
Channel Pruning

- Learning the Number of Neurons in Deep Networks 1611

\[
\min_{\Theta} \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, f(x_i, \Theta)) + r(\Theta)
\]

- Channel Pruning for Accelerating Very Deep Neural Networks 1707
  ○ Also exploits low-rankness of features

\[
\arg \min_{\beta, w} \frac{1}{2N} \left\| Y' - \sum_{i=1}^{c} \beta_i X_i W_i^\top \right\|_F^2
\]  
subject to  \[\| \beta \|_0 \leq c'\]
Sparse Communication for Distributed Gradient Descent

Algorithm 1 Gradient dropping algorithm given gradient $\nabla$ and dropping rate $R$.

function GRADDROP($\nabla$, $R$

$\nabla^+ = $ residuals
Select threshold: R% of $|\nabla|$ is smaller

$\text{dropped} \leftarrow 0$
$\text{dropped}[i] \leftarrow \nabla[i] \forall i : |
\nabla[i]| > \text{threshold}$
$\text{residuals} \leftarrow \nabla - \text{dropped}$

return sparse(dropped)

end function

Figure 3: NMT: Training loss and validation BLEU for different dropping ratios.
Quantization
Precursor: Ising model & Boltzmann machine

- **Ising model**
  - Used to model magnetics
  - 1D has trivial analytic solution
  - 2D exhibits phase-transition
  - 2D Ising model can be used for denoising
    - When the mean signal is reliable

- **Inference also requires optimization**

\[
H(\sigma) = -\sum_{\langle ij \rangle} J_{ij} \sigma_i \sigma_j - \mu \sum_j h_j \sigma_j
\]
Neural Network Training

Quantized

Activations/
Feature maps/
Neurons

θ

Quantized

θ

X

Cost
d + r

y

Output Layer

Hidden Layers

Input Layer

Quantized

Gradients
Backpropagation

\[
\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1} \frac{\partial w_1}{\partial x} = \left( \frac{\partial y}{\partial w_2} \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \left( \left( \frac{\partial y}{\partial w_3} \frac{\partial w_3}{\partial w_2} \right) \frac{\partial w_2}{\partial w_1} \right) \frac{\partial w_1}{\partial x} = \ldots
\]

There will be no gradient flow if we quantize somewhere!
Differentiable Quantization

- Bengio ’13: Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation
  - REINFORCE algorithm
  - Decompose binary stochastic neuron into stochastic and differentiable part
  - Injection of additive/multiplicative noise
  - Straight-through estimator

Gradient vanishes after quantization.
Quantization also at Train time

- Neural Network can adapt to the constraints imposed by quantization
- Exploits “Straight-through estimator” (Hinton, Coursera lecture, 2012)

\[
x \approx \hat{x}
\]

\[
\Rightarrow \\
\frac{\partial}{\partial x} \approx \frac{\partial}{\partial \hat{x}}
\]

- Example

**Forward:** \( q \sim Bernoulli(p) \quad q \approx \mathbb{E}[q] = p \)

**Backward:** \( \frac{\partial c}{\partial p} = \frac{\partial c}{\partial q} \).
Bit Neural Network

## Binarizing AlexNet

<table>
<thead>
<tr>
<th>Network Variations</th>
<th>Operations used in Convolution</th>
<th>Memory Saving (Inference)</th>
<th>Time Saving on CPU (Inference)</th>
<th>Accuracy on ImageNet (AlexNet)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard Convolution</strong></td>
<td>+, -, ×</td>
<td>1x</td>
<td>1x</td>
<td>%56.7</td>
</tr>
<tr>
<td><strong>Binary Weight</strong></td>
<td>+, −</td>
<td>~32x</td>
<td>~2x</td>
<td>%53.8</td>
</tr>
<tr>
<td><strong>Binary Weight (XNOR-Net)</strong></td>
<td>XNOR, bitcount</td>
<td>~32x</td>
<td>~58x</td>
<td>%44.2</td>
</tr>
</tbody>
</table>
Scaled binarization

\[ \min_{\Lambda \in \text{diagonal}, B \in \{1,-1\}^{m \times n}} \| \Lambda B - W \|_F^2 \]

- Sol:

\[ \| \Lambda B - W \|_F^2 = \| (\Lambda B) \circ B - W \circ B \|_F^2 \]
\[ = \| \Lambda (B \circ B) - W \circ B \|_F^2 \]
\[ = \| \Lambda 1 - W \circ B \|_F^2 \]

= Variance of rows of \( W \circ B \)
XNOR-Net
Binary weights network

- Filter repetition
  - 3x3 binary kernel has only 256 patterns modulo sign.
  - 3x1 binary kernel only has only 4 patterns modulo sign.
  - Not easily exploitable as we are applying CHW as filter

![Binary weight filters](image-url)

*Figure 2: Binary weight filters, sampled from of the first convolution layer. Since we have only $2^{k^2}$ unique 2D filters (where $k$ is the filter size), filter replication is very common. For instance, on our CIFAR-10 ConvNet, only 42% of the filters are unique.*
# Binarizing AlexNet

<table>
<thead>
<tr>
<th>Network Variations</th>
<th>Operations used in Convolution</th>
<th>Memory Saving (Inference)</th>
<th>Time Saving on CPU (Inference)</th>
<th>Accuracy on ImageNet (AlexNet)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Standard Convoluition</strong></td>
<td>+, −, ×</td>
<td>1x</td>
<td>1x</td>
<td>56.7</td>
</tr>
<tr>
<td>Real-Value Inputs</td>
<td>Real-Value Weights</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.11 -0.21 ... -0.34 ...</td>
<td>0.12 -1.2 ... 0.41 ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0.25 0.61 ... 0.52</td>
<td>-0.2 0.5 ... 0.68</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Binary Weight</strong></td>
<td>+, −</td>
<td>~32x</td>
<td>~2x</td>
<td>53.8</td>
</tr>
<tr>
<td>Real-Value Inputs</td>
<td>Binary Weights</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.11 -0.21 ... -0.34 ...</td>
<td>1 -1 ... 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0.25 0.61 ... 0.52</td>
<td>-1 1 ... 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Binary Weight (XNOR-Net)</strong></td>
<td>XNOR, bitcount</td>
<td>~32x</td>
<td>~58x</td>
<td>44.2</td>
</tr>
<tr>
<td>Binary Inputs</td>
<td>Binary Weights</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 -1 ... -1</td>
<td>1 -1 ... 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1 1 ... 1</td>
<td>-1 1 ... 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scaled binarization is no longer exact and not found to be useful

$$\alpha^*, B^*, \beta^*, H^* = \arg\min_{\alpha, B, \beta, H} \|X^T W - \beta \alpha H^T B\|$$

The solution below is quite bad, like when $Y = [-4, 1]$

$$C^* = \text{sign}(Y) = \text{sign}(X^T) \text{sign}(W) = H^{*^T} B^*$$
Quantization of Activations

- XNOR-net adopted STE method in their open-source our code
DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients

- Uniform stochastic quantization of gradients
  - 6 bit for ImageNet, 4 bit for SVHN
- Simplified scaled binarization: only scalar
  - Forward and backward multiplies the bit matrices from different sides.
  - Using scalar binarization allows using bit operations
- Floating-point-free inference even when with BN
- Future work
  - BN requires FP computation during training
  - Require FP weights for accumulating gradients
A has two times as many channels as B.
B has two times as many channels as C.

Table 1: Comparison of prediction accuracy for SVHN with different choices of Bit-width in a DoReFa-Net. $W, A, G$ are bitwidths of weights, activations and gradients respectively. When bitwidth is 32, we simply remove the quantization functions.
Quantization Methods

- **Deterministic Quantization**
  \[
  Q_{k}^{\text{det}}(X) = \alpha Q_{k}(\tilde{X}) + \beta \approx X
  \]

- **Stochastic Quantization**
  \[
  Q_{k}^{\text{stoc}}(X) = \alpha Q_{k}(\tilde{X} + \frac{\xi}{2^k - 1}) + \beta \approx X
  \]

  \[
  \xi \sim U(-\frac{1}{2}, \frac{1}{2})
  \]

  Injection of noise realizes the sampling.

\[
\tilde{X} = \frac{X - \beta}{\alpha}
\]

\[
\alpha = \max(X) - \min(X)
\]

\[
\beta = \min(X)
\]
Quantization of Weights, Activations and Gradients

- A half #channel 2-bit AlexNet (same bit complexity as XNOR-net)

<table>
<thead>
<tr>
<th>Quantization Method</th>
<th>Balanced Deterministic</th>
<th>Unbalanced Deterministic</th>
<th>Stochastic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weights</td>
<td>0.469</td>
<td>0.346</td>
<td>0.120</td>
</tr>
<tr>
<td>Activations</td>
<td>0.315</td>
<td>0.469</td>
<td>diverge</td>
</tr>
<tr>
<td>Gradients</td>
<td>diverge</td>
<td>diverge</td>
<td>0.469</td>
</tr>
<tr>
<td>XNOR-net</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNN</td>
<td></td>
<td>0.442*</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.279*</td>
<td></td>
</tr>
</tbody>
</table>
Quantization Error measured by Cosine Similarity

- Wb_sn is n-bit quantization of real W
- x is Gaussian R. V. clipped by tanh

```
Wb_s1 Wb_s2 Wb_s3 Wb_s4 Wb_s6 Wb_s8 W
0.684 0.684 0.699 0.795 0.882 0.888 0.888 xb
0.742 0.742 0.758 0.862 0.957 0.964 0.964 x_2
0.763 0.763 0.780 0.887 0.985 0.991 0.992 x_3
0.768 0.768 0.785 0.893 0.991 0.998 0.998 x_4
0.769 0.770 0.786 0.894 0.993 0.999 1.000 x_6
0.769 0.770 0.786 0.895 0.993 1.000 1.000 x_8
0.769 0.770 0.786 0.895 0.993 1.000 1.000 x
```

Saturates
Figure 1: Prediction accuracy of AlexNet variants on Validation Set of ImageNet indexed by epoch number. “W-A-G” gives the specification of bitwidths of weights, activations and gradients. E.g., “1-2-4” stands for the case when weights are 1-bit, activations are 2-bit and gradients are 4-bit. The figure is best viewed in color.
Figure 3: (a) is histogram of weights of layer “conv3” of “1-2-6” AlexNet model at epoch 5, 15, 35, respectively. (b) is histogram of activations at the same epochs.
Effective Quantization Methods for Recurrent Neural Networks 2016

<table>
<thead>
<tr>
<th>Model</th>
<th>weight-bits</th>
<th>activation-bits</th>
<th>PPW balanced</th>
<th>PPW unbalanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>2</td>
<td>2</td>
<td>152</td>
<td>164</td>
</tr>
<tr>
<td>LSTM</td>
<td>2</td>
<td>3</td>
<td>142</td>
<td>155</td>
</tr>
<tr>
<td>LSTM (Hubara et al., 2016a)</td>
<td>2</td>
<td>3</td>
<td>220</td>
<td></td>
</tr>
<tr>
<td>LSTM (Hubara et al., 2016a)</td>
<td>4</td>
<td>4</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

(a) histogram of floating-point weight values
(b) histogram-equalized weight values
(c) quantized weight values

(a) floating point copy of weights in QNN after 60 epochs
(b) imbalanced quantization (no equalization)
(c) balanced quantization with median (before matching value range)
(d) balanced quantization with mean (before matching value range)
Training Bit Fully Convolutional Network for Fast Semantic Segmentation 2016

<table>
<thead>
<tr>
<th>bit-width (W / A)</th>
<th>mean IoU</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 / 32</td>
<td>69.8%</td>
<td>-</td>
</tr>
<tr>
<td>8 / 8</td>
<td>69.8%</td>
<td>64</td>
</tr>
<tr>
<td>4 / 4</td>
<td>68.6%</td>
<td>16</td>
</tr>
<tr>
<td>3 / 3</td>
<td>67.4%</td>
<td>9</td>
</tr>
<tr>
<td>2 / 2</td>
<td>65.7%</td>
<td>4</td>
</tr>
<tr>
<td>1 / 4</td>
<td>64.4%</td>
<td>4</td>
</tr>
<tr>
<td>4 / 1</td>
<td>diverge</td>
<td>4</td>
</tr>
<tr>
<td>1 / 2</td>
<td>62.8%</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5: Results of different bit-width allocated to weight and activation on PASCAL VOC 2012 val set.

Figure 4: Examples on PASCAL VOC 2012.
FPGA is made up of many LUT's

a) Conceptual structure of an FPGA device.
b) Three-input LUT-based logic cell
TernGrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning

- Weights and activations not quantized.

### Table 2: Accuracy comparison for AlexNet.

<table>
<thead>
<tr>
<th>base LR</th>
<th>mini-batch size</th>
<th>workers</th>
<th>iterations</th>
<th>gradients</th>
<th>weight decay</th>
<th>DR</th>
<th>top-1</th>
<th>top-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>256</td>
<td>2</td>
<td>370K</td>
<td>floating</td>
<td>0.0005</td>
<td>0.5</td>
<td>57.33%</td>
<td>80.56%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TernGrad</td>
<td>0.0005</td>
<td>0.2</td>
<td>57.61%</td>
<td>80.47%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TernGrad-noclip</td>
<td>0.0005</td>
<td>0.2</td>
<td>54.63%</td>
<td>78.16%</td>
</tr>
<tr>
<td>0.02</td>
<td>512</td>
<td>4</td>
<td>185K</td>
<td>floating</td>
<td>0.0005</td>
<td>0.5</td>
<td>57.32%</td>
<td>80.73%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TernGrad</td>
<td>0.0005</td>
<td>0.2</td>
<td>57.28%</td>
<td>80.23%</td>
</tr>
<tr>
<td>0.04</td>
<td>1024</td>
<td>8</td>
<td>92.5K</td>
<td>floating</td>
<td>0.0005</td>
<td>0.5</td>
<td>56.62%</td>
<td>80.28%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TernGrad</td>
<td>0.0005</td>
<td>0.2</td>
<td>57.54%</td>
<td>80.25%</td>
</tr>
</tbody>
</table>

† DR: dropout ratio, the ratio of dropped neurons. ‡ TernGrad without gradient clipping.
More References

Backup after this slide

Slide also available at my home page:
https://zsc.github.io/
THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:

"MY MODEL'S TRAINING"

HEY! GET BACK TO WORK!

TRAINING!

OH. CARRY ON.
Low-rankness of Activations

- Accelerating Very Deep Convolutional Networks for Classification and Detection