Deep Learning Applications

“That drink will get you to 2800 calories for today”

“I last saw your keys in the store room”

“Remind Tom of the party”

“You’re on page 263 of this book”

Intelligent assistant  Surveillance / Remote assistance  Input keyboard
Model Serving Constraints

- **Latency constraint**
  - Batch size cannot be as large as possible when executing in the cloud
  - Can only run lightweight model in the device

- **Resource constraint**
  - Battery limit for the device
  - Memory limit for the device
  - Cost limit for using cloud

- **Accuracy constraint**
  - Some loss is acceptable by using approximate models
  - Multi-level QoS
Runtime Environment

Sensor
Processor
Radio

Camera
Touch screen
Microphone

CPU
GPU
FPGA
ASIC

Wifi
4G / LTE
Bluetooth

Cloud

streaming

CPU server
GPU server
TPU / FPGA server
Resource usage for a continuous vision app

**Imager**
- Omnivision OV2740
- 90mW

**Processor**
- Tegra K1 GPU
- 290GOPS@10W
- = 34pJ/OP

**Radio**
- Qualcomm SD810 LTE
- >800mW
- Atheros 802.11 a/g
- 15Mbps@700mW = 47nJ/b

**Cloud**
- Amazon EC2
  - CPU c4.large
    - 2x400GFLOPS
    - $0.1/h
  - GPU g2.2xlarge
    - 2.3TFLOPS
    - $0.65/h

---

**Workload**
- Deep learning 300GFLOPS @ 30GFLOPs/frame, 10fps

**Budget**
- Device power
  - 30% of 10Wh for 10h = 300mW
- Cloud cost
  - $10 person/year

**Compute power**
- 9GFLOPS
- 3.5GFLOPS (GPU) / 8GFLOPS (CPU)

---

Huge gap between workload and budget
Outline

● Model compression

● Serving System
Model Compression

- Tensor decomposition
- Network pruning
- Quantization
- Smaller model
Matrix decomposition

Fully-connected layer
- Memory reduction: \( \frac{MN}{(M+N)R} \)
- Computation reduction: \( \frac{MN}{(M+N)R} \)

Merge into one matrix
Tensor decomposition

Convolutional layer

- Memory reduction: \( \frac{D^2ST}{SR_3 + D^2R_3R_4 + TR_4} \) bounded by \( ST/R_3R_4 \)
- Computation reduction: \( \frac{D^2STH'W'}{SR_3HW + D^2R_3R_4H'W' + TR_4H'W'} \)

Decompose the entire model
Fine-tuning after decomposition
## Accuracy & Latency after Decomposition

<table>
<thead>
<tr>
<th>Model</th>
<th>Top-5</th>
<th>Weights</th>
<th>FLOPs</th>
<th>S6</th>
<th>Titan X</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>80.03</td>
<td>61M</td>
<td>725M</td>
<td>117ms</td>
<td>245mJ</td>
</tr>
<tr>
<td>AlexNet* (imp.)</td>
<td>(-1.70)</td>
<td>(×5.46)</td>
<td>(×2.67)</td>
<td>(×2.72)</td>
<td>(×3.41)</td>
</tr>
<tr>
<td>VGG-S</td>
<td>84.60</td>
<td>103M</td>
<td>2640M</td>
<td>357ms</td>
<td>825mJ</td>
</tr>
<tr>
<td>VGG-S* (imp.)</td>
<td>(-0.55)</td>
<td>(×7.40)</td>
<td>(×4.80)</td>
<td>(×3.68)</td>
<td>(×4.26)</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>88.90</td>
<td>6.9M</td>
<td>1566M</td>
<td>273ms</td>
<td>473mJ</td>
</tr>
<tr>
<td>GoogLeNet* (imp.)</td>
<td>(-0.24)</td>
<td>(×1.28)</td>
<td>(×2.06)</td>
<td>(×1.42)</td>
<td>(×1.60)</td>
</tr>
<tr>
<td>VGG-16</td>
<td>89.90</td>
<td>138M</td>
<td>15484M</td>
<td>1926ms</td>
<td>4757mJ</td>
</tr>
<tr>
<td>VGG-16* (imp.)</td>
<td>(-0.50)</td>
<td>(×1.09)</td>
<td>(×4.93)</td>
<td>(×3.34)</td>
<td>(×3.53)</td>
</tr>
</tbody>
</table>
Network pruning: Deep compression

* Song Han, et al. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." ICLR (2016).
Network pruning: prune the connections

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

**Hyper-parameter:** threshold, $\delta$.

**Output:** $W^{(t)}$.

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

// initialize the mask by thresholding the weights.
$Mask = 1(|W| > \text{threshold})$
$W = W \cdot Mask$

```
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
```

// Iterative Pruning
threshold $= \text{threshold} + \delta[\text{iter} + 1]$
goto Pruning Connections;
Network pruning: weight sharing

1. Use k-means clustering to identify the shared weights for each layer of a trained network. Minimize

\[
\arg \min_C \sum_{i=1}^{k} \sum_{w \in c_i} |w - c_i|^2
\]

2. Finetune the neural network using shared weights.

Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).
Network pruning: accuracy

Table 1: The compression pipeline can save $35\times$ to $49\times$ parameter storage with no loss of accuracy.

<table>
<thead>
<tr>
<th>Network</th>
<th>Top-1 Error</th>
<th>Top-5 Error</th>
<th>Parameters</th>
<th>Compress Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeNet-300-100 Ref</td>
<td>1.64%</td>
<td>-</td>
<td>1070 KB</td>
<td></td>
</tr>
<tr>
<td>LeNet-300-100 Compressed</td>
<td>1.58%</td>
<td>-</td>
<td>27 KB</td>
<td>$40\times$</td>
</tr>
<tr>
<td>LeNet-5 Ref</td>
<td>0.80%</td>
<td>-</td>
<td>1720 KB</td>
<td></td>
</tr>
<tr>
<td>LeNet-5 Compressed</td>
<td>0.74%</td>
<td>-</td>
<td>44 KB</td>
<td>$39\times$</td>
</tr>
<tr>
<td>AlexNet Ref</td>
<td>42.78%</td>
<td>19.73%</td>
<td>240 MB</td>
<td></td>
</tr>
<tr>
<td>AlexNet Compressed</td>
<td>42.78%</td>
<td>19.70%</td>
<td>6.9 MB</td>
<td>$35\times$</td>
</tr>
<tr>
<td>VGG-16 Ref</td>
<td>31.50%</td>
<td>11.32%</td>
<td>552 MB</td>
<td></td>
</tr>
<tr>
<td>VGG-16 Compressed</td>
<td>31.17%</td>
<td>10.91%</td>
<td>11.3 MB</td>
<td>$49\times$</td>
</tr>
</tbody>
</table>
Network pruning: accuracy vs compression

Figure 6: Accuracy v.s. compression rate under different compression methods. Pruning and quantization works best when combined.
Quantization: XNOR-Net


<table>
<thead>
<tr>
<th>Network Variations</th>
<th>Operations used in Convolution</th>
<th>Memory Saving (Inference)</th>
<th>Computation Saving (Inference)</th>
<th>Accuracy on ImageNet (AlexNet)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Convolution</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>0.012 -0.12 … 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>Binary Weight</td>
<td>+, −</td>
<td>~32x</td>
<td>~2x</td>
<td>%56.8</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 -1 -1 -1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−0.25 0.61 … 0.52</td>
<td>−1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Weight (XNOR-Net)</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 -1 -1</td>
<td>1 -1 -1 -1 -1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Quantization: binary weights

Algorithm 1 Training an $L$-layers CNN with binary weights:

Input: A minibatch of inputs and targets $(I, Y)$, cost function $C(Y, \hat{Y})$, current weight $W^t$ and current learning rate $\eta^t$.

Output: updated weight $W^{t+1}$ and updated learning rate $\eta^{t+1}$.

1: Binarizing weight filters:
2: for $l = 1$ to $L$
3: for $k^{th}$ filter in $l^{th}$ layer do
4: $A_{lk} = \frac{1}{n} \| W_{lk}^t \|_1$
5: $B_{lk} = \text{sign}(W_{lk}^t)$
6: $W_{lk} = A_{lk}B_{lk}$
7: $\hat{Y} = \text{BinaryForward}(I, B, A)$ // standard forward propagation except that convolutions are computed using equation 1 or 11
8: $\frac{\partial C}{\partial W} = \text{BinaryBackward}(\frac{\partial C}{\partial \hat{Y}}, \hat{W})$ // standard backward propagation except that gradients are computed using $\hat{W}$ instead of $W^t$
9: $W^{t+1} = \text{UpdateParameters}(W^t, \frac{\partial C}{\partial W}, \eta_t)$ // Any update rules (e.g., SGD or ADAM)
10: $\eta^{t+1} = \text{UpdateLearningrate}(\eta^t, t)$ // Any learning rate scheduling function
Quantization: binary input and weights

**Binary Dot Product:** To approximate the dot product between $X, W \in \mathbb{R}^n$ such that $X^T W \approx \beta H^T \alpha B$, where $H, B \in \{+1, -1\}^n$ and $\beta, \alpha \in \mathbb{R}^+$, we solve the following optimization:

$$\alpha^*, B^*, \beta^*, H^* = \arg\min_{\alpha, B, \beta, H} \|X \odot W - \beta \alpha H \odot B\|$$  \hspace{1cm} (7)

$$C^* = \text{sign}(Y) = \text{sign}(X) \odot \text{sign}(W) = H^* \odot B^*$$

$$\gamma^* = \frac{\sum |Y_i|}{n} = \frac{\sum |X_i||W_i|}{n} \approx \left(\frac{1}{n} \|X\|_{\ell_1}\right) \left(\frac{1}{n} \|W\|_{\ell_1}\right) = \beta^* \alpha^*$$
Fig. 5: This figure compares the imagenet classification accuracy on Top-1 and Top-5 across training epochs. Our approaches BWN and XNOR-Net outperform BinaryConnect(BC) and BinaryNet(BNN) in all the epochs by large margin(∼17%).
Quantization

Accuracy of a MLP (784-128-64-10) trained on MNIST

Validation Accuracy (%) vs. Weight Precision (bits)

- Precise: 100%
- Quantize after Training: 96.78%
Quantization

Impact of Resnet Quantization on Validation Accuracy of CIFAR-10

- Param Size (MB) vs. Mode Validation. Color shows details about sum of Init Bits. Size shows sum of Width Factor. Shape shows details about sum of Depth. The data is filtered on Weight Mode, which keeps tanh. The view is filtered on sum of Width Factor, which ranges from 1 to 1.
Smaller model: Knowledge distillation

- Knowledge distillation: use a teacher model (large model) to train a student model (small model)

\[ \mathcal{L}_{HT}(W_{\text{Guided}}, W_r) = \frac{1}{2} \| u_h(x; W_{\text{Hint}}) - r(v_g(x; W_{\text{Guided}}); W_r) \|^2, \]

### Smaller model: accuracy

#### Table 1: Accuracy on CIFAR-10

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># params</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FitNet</td>
<td>~2.5M</td>
<td>91.61%</td>
</tr>
<tr>
<td>Teacher</td>
<td>~9M</td>
<td>90.18%</td>
</tr>
<tr>
<td>Mimic single</td>
<td>~54M</td>
<td>84.6%</td>
</tr>
<tr>
<td>Mimic single</td>
<td>~70M</td>
<td>84.9%</td>
</tr>
<tr>
<td>Mimic ensemble</td>
<td>~70M</td>
<td>85.8%</td>
</tr>
<tr>
<td>State-of-the-art methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maxout</td>
<td></td>
<td>90.65%</td>
</tr>
<tr>
<td>Network in Network</td>
<td></td>
<td>91.2%</td>
</tr>
<tr>
<td>Deeply-Supervised Networks</td>
<td></td>
<td>91.78%</td>
</tr>
<tr>
<td>Deeply-Supervised Networks (19)</td>
<td></td>
<td>88.2%</td>
</tr>
</tbody>
</table>

#### Table 2: Accuracy on CIFAR-100

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># params</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FitNet</td>
<td>~2.5M</td>
<td>64.96%</td>
</tr>
<tr>
<td>Teacher</td>
<td>~9M</td>
<td>63.54%</td>
</tr>
<tr>
<td>State-of-the-art methods</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maxout</td>
<td></td>
<td>61.43%</td>
</tr>
<tr>
<td>Network in Network</td>
<td></td>
<td>64.32%</td>
</tr>
<tr>
<td>Deeply-Supervised Networks</td>
<td></td>
<td>65.43%</td>
</tr>
</tbody>
</table>
Discussion

What are the implications of these model compression techniques for serving?

- **Specialized hardware for sparse models**
- **Accuracy and resource trade-off**
Serving system
Serving system

- **Goals:**
  - High flexibility for writing applications
  - High efficiency on GPUs
  - Satisfy latency SLA

- **Challenges**
  - Provide common abstraction for different frameworks
  - Achieve high efficiency
    - Sub-second latency SLA that limits the batch size
    - Model optimization and multi-tenancy causes long tail
Nexus: efficient neural network serving system

Remarks
• Frontend runtime library allows arbitrary app logic
• Packing models to achieve higher utilization
• A GPU scheduler allows new batching primitives
• A batch-aware global scheduler allocates GPU cycles for each model
Flexibility: Application runtime

class ModelHandler:
    # return output future
    def Execute(input):

class AppBase:
    # return ModelHandler
    def GetModelHandler(framework, model, version, latency_sla):
        # Load models during setup time, implemented by developer
        def Setup():
            # Process requests, implemented by developer
            def Process(request):

Async RPC, execute the model remotely

Send load model request to global scheduler
class FaceRecApp(AppBase):
    def Setup(self):
        self.m1 = self.GetModelHandler("caffe", "vgg_face", 1, 100)
        self.m2 = self.GetModelHandler("mxnet", "age_net", 1, 100)
    def Process(self, request):
        ret1 = self.m1.Execute(request.image)
        ret2 = self.m2.Execute(request.image)
        return Reply(request.user_id, ret1["name"], ret2["age"])
Application example: Traffic Analysis

class TrafficApp(AppBase):
    def Setup(self):
        self.det = self.GetModelHandler("darknet", "yolo9000", 1, 300)
        self.r1 = self.GetModelHandler("caffe", "vgg_face", 1, 150)
        self.r2 = self.GetModelHandler("caffe", "googlenet_car", 1, 150)

    def Process(self, request):
        persons, cars = [], []
        for obj in self.det.Execute(request.image):
            if obj["class"] == "person":
                persons.append(self.r1.Execute(request.image[obj["rect"]]))
            elif obj["class"] == "car":
                cars.append(self.r2.Execute(request.image[obj["rect"]]))
        return Reply(request.user_id, persons, cars)
High Efficiency

- For high request rate, high latency SLA workload, saturate GPU efficiency by using large batch size

![Graph showing VGG16 throughput vs batch size with GPU saturate region at batch size >= 32]

<table>
<thead>
<tr>
<th>Request rate</th>
<th>Latency SLA</th>
</tr>
</thead>
<tbody>
<tr>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>low</td>
<td>low</td>
</tr>
</tbody>
</table>
High Efficiency

- Suppose we can choose a different batch size for each op (layer), and allocate dedicated GPUs for each op.

![Diagram showing workload characteristic with request rate vs latency SLA]

- Workload characteristic diagram
  - Request rate: low to high
  - Latency SLA: low to high

![Graphs showing throughput for VGG16 conv2_1 and VGG16 fc6]

- Throughput graphs
  - VGG16 conv2_1 (winograd): batch size 16
  - VGG16 fc6: throughput vs inputs/sec

- GPU allocation for ops:
  - op1, op2, ..., op_n

35
Split Batching

Optimization problem

\[
\max_{b_1 \ldots b_n} \sum_i \frac{\max_j tp_j(b_j)}{tp_i(b_i)}
\]
equivalent to

\[
\min_{b_1 \ldots b_n} \sum_i 1/tp_i(b_i)
\]
such that

\[
\sum_i lat_i(b_i) + \sum_{b_i \neq b_{i+1}} \text{overhead(out}_i \ast b_i) \leq \text{latency sla}
\]
Split Batching

VGG16 optimal batch size for each segment for latency SLA 15ms
High Efficiency

- This type of workload cannot saturate GPU in temporal domain
- Suppose the optimal batch size is $b$ under latency SLA

Execute multiple models on one GPU
Execute multiple models on a single GPU

Model A requests

GPU execution

(a) Single model execution: worst latency is $2\text{lat}_A$
Execute multiple models on a single GPU

(a) Single model execution: worst latency is $2\text{lat}_A$

(b) Execute multiple models **concurrently**: worst latency for model A and B is $2(\text{lat}_A + \text{lat}_B)$
Execute multiple models on a single GPU

Use larger batch size as latency is reduced and predictive

(c) Execute multiple models in round-robin fashion:
- worst latency for model A is $2\text{lat}_A + \text{lat}_B$
- worst latency for model B is $\text{lat}_A + 2\text{lat}_B$
High Efficiency

Solution depends

- If saturate GPU in temporal domain due to low latency: allocate dedicated GPU(s)
- If not: can use multi-batching to share GPU cycles with other models
Prefix batching for model specialization

• Specialization (long-term / short-term) re-trains last a few layers

A new model
Prefix batching for model specialization

• Specialization (long-term / short-term) re-trains last a few layers

• Prefix batching allows batch execution for common prefix

A new model

Different suffixes execute individually

Common prefix can be batched together
Meet Latency SLA: Global scheduler

• First apply split batching and prefix batching if possible
• Multi-batching: bin-packing problem to pack models into GPUs
• Bin-packing optimization goal
  • Minimize the resource usage (number of GPUs)
• Constraint
  • Requests have to be served within latency SLA
• Degrees of freedom
  • Split workloads into smaller tasks
  • Change the batch size
Best-fit decreasing algorithms

1. For each workload, $T_i$ is the max throughput that can be achieved on a GPU within latency SLA, and allocate dedicated GPUs

<table>
<thead>
<tr>
<th>Application $i$, latency SLA $L_i$</th>
<th>Model $M_{ki}$</th>
<th>$T_i$</th>
<th>$T_i$</th>
<th>$r_i$</th>
<th>$R_i$</th>
<th>$R_i$</th>
<th>$R_i$</th>
</tr>
</thead>
</table>

46
Best-fit decreasing algorithms

1. For each workload, $T_i$ is the max throughput that can be achieved on a GPU within latency SLA, and allocate dedicated GPUs.

2. For each residue workload, split latency SLA into batching cycle and exec cycle.

Application $i$, latency SLA $L_i$

Model $M_{k_i}$

Latency constraint $L_i$

Bin to fit in execution cycles

batching cycle $d_i = b/r_i$
exec cycle $p_{k_i}(b)$
Best-fit decreasing algorithms

1. For each workload, $T_i$ is the max throughput that can be achieved on a GPU within latency SLA, and allocate dedicated GPUs

2. For each residue workload, split latency SLA into batching cycle and exec cycle

3. Sort all residue workloads by occupancy $\ell_{ki}(B_i)/d_i$, and merge them by best-fit
Reference

- Song Han, et al. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding." ICLR (2016).