Training ImageNet in 4 minutes with Tencent Jizhi

Yangzihao Wang, Haidong Rong
Cloud Architecture Platform Dept. TEG at Tencent
Agenda

- Motivations
- System implementation and optimizations
- Introduction of Jizhi Platform
- Case Studies
- Problems and Countermeasures
Motivations
Motivations

Problems we try to solve

- **Academic:**
  - Difficult to train with large-batch and on clusters

- **Industrial:**
  - Complex/arbitrary training pipelines for models from different fields
  - Separation between experimental models and distributed trained industrial-level models
Motivations

Problems for academic research:

Two Challenges in large-batch distributed training:

How to maintain the same accuracy with large mini-batch training with SGD?

How to achieve near-linear scalability on large clusters?
Motivations

Solution: Tencent Jizhi

A High Performance Distributed Deep Learning Training Platform
Motivations

Goals for industrial applications:

High-Performance: Integration of general optimization strategies

Efficiency: Modular feature combinations and flexible resource management

Usability: Automate/Standardize Stages in ML pipeline
System Implementation and Optimizations
Can we train ImageNet using 1024 GPUs with a batch size of 64K?

How fast can we do it?
Optimization Techniques for Large-Batch Training

Mixed-precision training with LARS

Improvements on model architecture

Improvements on communication strategies
System Implementation and Optimizations

**Mixed-precision Training with LARS**

**Layer-wise Adaptive Rate Scaling:**

Set local learning rate per layer to stabilize the rate scaling:

\[ \Delta w^l_t = \gamma \cdot \eta \cdot \frac{\|w^l_t\|}{\|\nabla L(w^l_t)\|} \cdot \nabla L(w^l_t) \]

Can be used with momentum and weight decay in SGD:

---

**Algorithm 1** SGD with LARS. Example with weight decay, momentum and polynomial LR decay.

- **Parameters:** base LR \( \gamma_0 \), momentum \( m \), weight decay \( \beta \), LARS coefficient \( \eta \), number of steps \( T \)
- **Init:** \( t = 0, v = 0 \)
  - Init weight \( w^l_0 \) for each layer \( l \)
  - while \( t < T \) for each layer \( l \)
    - \( g^l_t = \nabla L(w^l_t) \) (obtain a stochastic gradient for the current mini-batch)
    - \( \gamma_t = \gamma_0 \cdot (1 - \frac{t}{T})^2 \) (compute the global learning rate)
    - \( \lambda^l = \frac{\|\nabla^l \|}{\|g^l_t + \beta w^l_t\|} \) (compute the local LR \( \lambda^l \))
    - \( v^l_{t+1} = m v^l_t + \gamma_t \cdot \lambda^l \cdot (g^l_t + \beta w^l_t) \) (update the momentum)
    - \( w^l_{t+1} = w^l_t - v^l_{t+1} \) (update the weights)
  - end while
Mixed-precision Training with LARS

A different story for FP16:
Mixed-precision Training with LARS

- FP16 (Weight, Grad)
- FP16ToFP32
- FP32 (Weight, Grad)
- LARS
- FWD BWD
- FP16 (Weight, Grad)
- FP32ToFP16
- Weight Update
Mixed-precision Training with LARS

Table 1: Effectiveness of using LARS on ResNet-50

<table>
<thead>
<tr>
<th>Mini-Batch Size</th>
<th>Number of Epochs</th>
<th>LARS</th>
<th>Top-1 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K</td>
<td>90</td>
<td>NO</td>
<td>73.2%</td>
</tr>
<tr>
<td>64K</td>
<td>90</td>
<td>YES</td>
<td>76.2%</td>
</tr>
</tbody>
</table>
System Implementation and Optimizations

**Improvements on Model Architecture**

*Do not do weight decay on bias and \( \beta, \gamma \) in BN:*

A typical practice to penalize only the weights of the affine transformation at each layer and leaves the biases unregularized:

\[
\begin{align*}
\mathbf{w}_i^{t+1} &= \mathbf{w}_i^t - \eta \frac{\partial E}{\partial \mathbf{w}_i^t} - \lambda \mathbf{w}_i^t
\end{align*}
\]

*Do not penalize the trainable params \( \beta, \gamma \) in BN:*

\[
\begin{align*}
\hat{x}_i &\leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma^2 + \epsilon}} \\
y_i &\leftarrow \gamma \hat{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i)
\end{align*}
\]
Improvements on Model Architecture

Do not do weight decay on bias and $\beta$, $\gamma$ in BN:

<table>
<thead>
<tr>
<th>Batch</th>
<th>Epochs</th>
<th>Regularize $b$, $\beta$ and $\gamma$</th>
<th>Top1</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K</td>
<td>95</td>
<td>Yes</td>
<td>55.8%</td>
</tr>
<tr>
<td>64K</td>
<td>95</td>
<td>No</td>
<td>57.1%</td>
</tr>
</tbody>
</table>

Effect of improvements to ResNet-50 Training

<table>
<thead>
<tr>
<th>Batch</th>
<th>No Decay BN</th>
<th>Top1</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K</td>
<td>×</td>
<td>71.9%</td>
</tr>
<tr>
<td>64K</td>
<td>✓</td>
<td>76.2%</td>
</tr>
</tbody>
</table>
System Implementation and Optimizations

**Improvements on Model Architecture**

*Insert BN layer after Pool5 in AlexNet*
Insert BN layer after Pool5 in AlexNet

Figure 4: Feature Map Distribution of Pool5(a) and Pool5-BN5(b) of AlexNet as shown in Figure 3. (the horizontal axis is the training steps, the vertical axis is the feature map distributions.)
System Implementation and Optimizations

**Improvements on Model Architecture**

*Insert BN layer after Pool5 in AlexNet*

<table>
<thead>
<tr>
<th>Batch</th>
<th>No Decay Bias</th>
<th>No Decay BN</th>
<th>pool5 BN</th>
<th>Top-1 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>64K</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>55.8%</td>
</tr>
<tr>
<td>64K</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>56.3%</td>
</tr>
<tr>
<td>64K</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>56.4%</td>
</tr>
<tr>
<td>64K</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>57.1%</td>
</tr>
<tr>
<td>64K</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>58.8%</td>
</tr>
</tbody>
</table>
System Implementation and Optimizations

Improvements on Communication

For large batch training with distributed synchronized SGD, efficient gradients aggregation across all GPUs after each iteration is crucial to the training performance.
NCCL 2.0 alone cannot solve the problem:

In a cluster with $k$ GPUs, Ring all-reduce will split the data on each GPU into $k$ chunks and do the reduce in $k-1$ iterations.

When $k$ gets larger, the messages passing between nodes will become smaller and fail to utilize the full bandwidth of the network.
System Implementation and Optimizations

Improvements on Communication

Tensor Fusion:
Hierarchical All-Reduce:

\[ p \text{ GPUs}, p/k \text{ groups:} \]

Ring All-Reduce #steps: \(2(p-1)\)
Hierarchical All-Reduce #steps: \(4(k-1)+2(p/k-1)\)
In our case: \(p=1024, k=16\) achieves the best performance
Improvements on Communication

Hybrid All-Reduce:
System Implementation and Optimizations

**Results**

Both Models Converge to \( \geq \) baseline accuracy:
Results

Both Models Converge to $\geq$ baseline accuracy, and fast!

Table 4: Compare AlexNet training with different teams

<table>
<thead>
<tr>
<th>Team</th>
<th>Batch</th>
<th>Hardware</th>
<th>Software</th>
<th>Top-1 Accuracy</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>You et al. [27]</td>
<td>512</td>
<td>DGX-1 station</td>
<td>NVCaffe</td>
<td>58.8%</td>
<td>6h 10m</td>
</tr>
<tr>
<td>You et al. [27]</td>
<td>32K</td>
<td>CPU × 1024</td>
<td>Intel Caffe</td>
<td>58.6%</td>
<td>11min</td>
</tr>
<tr>
<td>This work</td>
<td>64K</td>
<td>Tesla P40 × 512</td>
<td>TensorFlow</td>
<td>58.8%</td>
<td>5m</td>
</tr>
<tr>
<td>This work</td>
<td>64K</td>
<td>Tesla P40 × 1024</td>
<td>TensorFlow</td>
<td>58.7%</td>
<td>4m</td>
</tr>
</tbody>
</table>

Table 5: Compare ResNet-50 training with different teams

<table>
<thead>
<tr>
<th>Team</th>
<th>Batch</th>
<th>Hardware</th>
<th>Software</th>
<th>Top-1 Accuracy</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>He et al. [13]</td>
<td>256</td>
<td>Tesla P100 × 8</td>
<td>Caffe</td>
<td>75.3%</td>
<td>29h</td>
</tr>
<tr>
<td>Goyal et al. [12]</td>
<td>8K</td>
<td>Tesla P100 × 256</td>
<td>Caffe2</td>
<td>76.3%</td>
<td>1h</td>
</tr>
<tr>
<td>Cho et al. [4]</td>
<td>8K</td>
<td>Tesla P100 × 256</td>
<td>Torch</td>
<td>75.0%</td>
<td>50min</td>
</tr>
<tr>
<td>Codreanu et al. [5]</td>
<td>32K</td>
<td>KNL × 1024</td>
<td>Intel Caffe</td>
<td>75.3%</td>
<td>42min</td>
</tr>
<tr>
<td>You et al. [27]</td>
<td>32K</td>
<td>KNL × 2048</td>
<td>Intel Caffe</td>
<td>75.4%</td>
<td>20min</td>
</tr>
<tr>
<td>Akiba et al. [2]</td>
<td>32K</td>
<td>Tesla P100 × 1024</td>
<td>Chainer</td>
<td>74.9%</td>
<td>15min</td>
</tr>
<tr>
<td>This work</td>
<td>64K</td>
<td>Tesla P40 × 1024</td>
<td>TensorFlow</td>
<td>76.2%</td>
<td>8.7m</td>
</tr>
<tr>
<td>This work</td>
<td>64K</td>
<td>Tesla P40 × 2048</td>
<td>TensorFlow</td>
<td>75.8%</td>
<td>6.6m</td>
</tr>
</tbody>
</table>
System Implementation and Optimizations

Results

Also we scale (almost) linearly:

**ResNet-50**

**AlexNet**
Future Works

AutoML and Network Optimizations

More Communication Optimizations

More Optimizers: Second-Order Optimization

More Execution Methods: Asynchronized Training
**Agenda**

- Motivations
- System implementation and optimizations
- Introduction of Jizhi Platform
- Case Studies
- Problems and Countermeasures
Introduction of Jizhi Platform
Introduction of Jizhi Platform

**What is the Jizhi Platform?**

- A General-purpose AI Platform
- A Distinctive AI Accelerating Platform
Introduction of Jizhi Platform

Architecture of Jizhi Platform

Al Applications
- Game AI
- Medical AI
- ASR

Jizhi Platform
- AutoML
- Large Batchsize Convergence Algorithm
- Optimization Strategy Integration
- Model & Computing Power Analysis
- Flexible Resources Scheduler

Al Framework
- TensorFlow
- PyTorch
- Caffe2
- CNTK

Al Infrastructure
- kubernetes
- docker
- NVIDIA
- Mellanox
- Intel

The Platform has served for more than 5 internal businesses and created the practical commercial value.
Introduction of Jizhi Platform

Features of Tencent Jizhi

· High Efficiency
  · Integrating more than 60% computing resources of Tencent into one unified pool
  · Increasing Utilization Rate through Flexible Strategy
    · High priority: Exclusive with budget
    · Low priority: Shared once idle
Introduction of Jizhi Platform

Features of Tencent Jizhi

- High Performance:
  - Model level optimization (LARS, Mix precision training, etc.)
  - Framework level optimization (State-of-Art optimizer and loss function, etc.)
  - Platform level optimization (Tensor fusion, Hierarchical allreduce, etc.)
Introduction of Jizhi Platform

Features of Tencent Jizhi

- **High Performance**

![Graphs showing performance comparisons](image)

- Allocating more than 1000 GPUs for a single task
- Running on Super large-scale cluster with near-linear speedup
- And Supporting large batch size without significant accuracy loss
Introduction of Jizhi Platform

Features of Tencent Jizhi

- High Usability
  - Automatic parallelization / Transparent to Model Engineers
  - Simple High-level API (network, dataset)
  - Support secondary development through an Open API set

```python
class Network(object):
    def __init__(self, params):
        self.params = params

    def inference(self, images):

    def cal_loss_accuracy_and_others(self, input_data):

    def cal_accuracy_and_others(self, input_data):
```
```python
def get_train_filename_list(params, path):

def get_valid_filename_list(params, path):

def get_samples(params, path, sample_file_tuple, queue):
```
Introduction of Jizhi Platform

UI of Tencent Jizhi
Introduction of Jizhi Platform

UI of Tencent Jizhi
4 Case Studies
**Appliance on Game AI**

- a Dota2-like MOBA Game
- Reinforcement Learning
## Appliance on Game AI

- **128k batch size for 5v5 on 128 GPUs**
- **Exceeding OpenAI (1M vs ~560K rounds/day) on the same number of GPUs**

<table>
<thead>
<tr>
<th>Game Type</th>
<th>GPU</th>
<th>Batch size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>5v5</td>
<td>Baseline 8 GPUs</td>
<td>8k</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Jizhi 128 GPUs</td>
<td><strong>128K</strong></td>
<td><strong>13.6</strong></td>
</tr>
</tbody>
</table>
Case Studies

Appliance on Automatic Speech Recognition

- LSTM model/ DNN model
- Super Large-scale Dataset
  - 100 thousand or even hundreds of thousands hours corpus
  - Larger than 10 TB
- More Than 3 months/epoch on 4 * Tesla M40
**Appliance on Automatic Speech Recognition**

- **Training time reduced from more than 3 months to 20 hours**

<table>
<thead>
<tr>
<th></th>
<th>samples/s</th>
<th>1 epoch</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline 4 GPU</td>
<td>210</td>
<td>2194h</td>
<td>--</td>
</tr>
<tr>
<td>Jizhi 4 GPU</td>
<td>956</td>
<td>482h</td>
<td>4.55</td>
</tr>
<tr>
<td>Jizhi 120 GPU</td>
<td>753</td>
<td>20h</td>
<td><strong>107</strong></td>
</tr>
</tbody>
</table>
5 Problems and Countermeasures
Problems and Countermeasures

Node affinity is not always satisfied

- The assigned nodes are not always under the same switch
- Imbalance Bandwidth between different nodes

How to maximize performance in this case?
Counter measure

- Reducing bandwidth requirements by
- Asynchronous training algorithm
  - Such as BMUF [Kai et al. 2016][1]
- Gradient compression algorithm
  - Such as Deep Gradient Compression [Yujun Lin et al. 2017][2]

---

Algorithm 1: BlockMomentumSGD with Nesterov Block Momentum:

**Input:**
- The initial model $w_0$;
- Training data with labels $X$;
- Block momentum $\eta_b$ and learning rate $\eta_0$;
- Synchronization period $n$;
- Number of workers $K$;

**Initialization:** $v_0 = 0$

for $t = 1, \ldots, T$

for $k = 1, \ldots, K$ parallel do

1. Initialize the local models: $w_{t-1}^{(k)} = w_{t-1} - \eta_0 v_{t-1}$
2. Update local models using SGD:
   for $r = 1, \ldots, n$ do
     - Draw a mini-batch from $X$;
     - Calculate gradient on the current mini-batch;
     - Optionally) Additional gradient processing, e.g., SGD momentum or adagrad
   end
3. Send block gradient $g_k = w_{t-1}^{(k)} - w_{t}^{(k)}$ to the master;

end

- Aggregate and filter block gradients:

$$v_k = \eta_b v_{t-1} + (1 - \eta_b) \sum g_k$$

$$w_t = w_{t-1} - v_t$$

---

Problems and Countermeasures

Hyper-parameter tuning can be expensive

- Too many tunable parameters
- Training with one hyper-parameter set takes too long (e.g. ASR)
- Limited budget for GPU time
**Countermeasure**

- **Second-order Optimization**
- Hessian-free algorithm [James et al. 2012][1]
- Distributed K-FAC (Jimmy et al. 2017)[2]

---

Problems and Countermeasures

**Imbalance Computing Resource**

- Computing resource should be evenly distributed across batches
- But sometimes there is not enough GPUs with the same type in pool
Problems and Countermeasures

Countermeasure

- Asynchronous Decentralized Training Algorithm
  - Such as AD-PSGD (Xiangru et al. 2018) [1]

---

Algorithm 1 AD-PSGD (logical view)

1. **Require**: Initialize local models \( \{x_i^0\}_{i=1}^{n} \) with the same initialization, learning rate \( \gamma \), batch size \( M \), and total number of iterations \( K \).
2. **for** \( k = 0, 1, \ldots, K - 1 \) **do**
3. \hspace{1em} Randomly sample a worker \( i_k \) of the graph \( G \) and randomly sample an averaging matrix \( W_k \) which can be dependent on \( i_k \).
4. \hspace{1em} Randomly sample a batch \( \xi_{k,j} := \{y^j_{k,1}, y^j_{k,2}, \ldots, y^j_{k,M}\} \) from local data of the \( i_k \)-th worker.
5. \hspace{1em} Compute the stochastic gradient locally \( g_k(x_k^j; \xi_{k,j}^j) := \sum_{m=1}^{M} \nabla F(x_k^j; z_{k,m}) \).
6. \hspace{1em} Average local modes by \( \bar{a} [x_{k+1/2}^1, x_{k+1/2}^2, \ldots, x_{k+1/2}^n] \leftarrow [x_{k}^1, x_{k}^2, \ldots, x_{k}^n]W_k \)
7. \hspace{1em} Update the local model \( x_k^j \leftarrow x_k^j - \gamma g_k(x_k^j, \xi_{k,j}) \) and \( x_{k+1/2}^j \leftarrow x_{k+1/2}^j, \forall j \neq i_k \).
8. **end for**
9. Output the average of the models on all workers.

Q&A
Thanks