Outline

• Project Adam: Building an Efficient and Scalable Deep Learning Training System
• Using Google Cloud Machine Learning to predict clicks at scale
Project Adam: Building an Efficient and Scalable Deep Learning Training System

Trishul Chilimbi, etal.

Microsoft Research
Background

Deep networks learn complex representations

Accuracy improvement with larger models and more data.
Deep Learning Computational Requirements

- Size of model
  - Complexity of task

- Amount of (weakly labeled) data
  - Size of model

- Computation required
  - Complexity of task
Deep Learning Training Systems

- Single Machine with multiple GPU cards
- DistBelief system (NIPS’12)
- Distributed cluster of 16 GPU servers (ICML’13)
- Adam (USENIX’14)
Adam: Scalable DL Platform

- Data serving machines: provide training input to model training machines
- Multiple replicas on model training machines
- Asynchronously update a shared model via a global parameter server.
Adam system architecture

- Global Model Parameter Store
- Model Replica
- Model Parallelism
- Data Parallelism
- Data Shards
- Model Workers

Similar to DistBelief ‘12
Neural Network Training

- Free-forward evaluation

\[ a_i(l) = F\left((\sum_{j=1..k} w_{ij}(l-1,l) * a_j(l-1)) + b_i\right) \]

- Back-propagation:

\[ \delta_i(l_n) = (t_i(l_n) - a_i(l_n)) * F'(a_i(l_n)) \]

\[ \delta_i(l) = (\sum_{j=1..m} \delta_j(l+1) * w_{ji}(l,l+1)) * F'(a_i(l)) \]

- Weight updates

\[ \Delta w_{ij}(l-1,l) = \alpha \delta_i(l) * a_j(l-1) \text{ for } j = 1 .. k \]
Main Contribution

- **Optimization and Balance**: Optimizing and balancing both computation and communication for this application through whole system co-design.

- **High performance and Scalability**: Achieving high performance and scalability by exploiting the ability of machine learning training to tolerate inconsistencies well.

- **Demonstrate**: System efficiency, scaling, and asynchrony all contribute to improvements in trained model accuracy. Task accuracy improves with model size.
System Hardware of Adam

- A cluster of 120 identical machines organized as three equally sized racks connected by IBM G8264 switches.
- Each machine: HP Proliant server (CPU: dual Intel Xeon E5-2450L processors 16 cores, 1.8Ghz; Main memory: 98GB)
- Two 10 Gb NICs and one 1 Gb NIC.
- All machines have four 7200 rpm HDDs. A 1TB drive hosts the operating system (Windows 2012 server) and the other three HDDs are 3TB each and are configured as a RAID array.
- Model training machines selected from a pool of 90 machines
- Parameter servers selected from a pool of 20 machines
- Image servers from selected from a pool of 10 machines.
Fast Data Serving

- Data Transformation in advance
- Pre-cache images
- Asynchronous IO
- Background thread: request images in advance
- Main thread: always have required data from memory
Model Training Optimization

- Multi-Threaded Training
- Fast Weight Updates
- Reducing Memory Copies
- Memory System Optimizations
- Mitigating the Impact of Slow Machines
- Parameter Server Communication
Multi-thread Training

- Different images assigned to threads that share the model weights.
- Each training context: feed-forward evaluation and back propagation.
- Context: pre-allocated, avoid heap locks while training.
- Context & per-thread scratch buffer: NUMA-aware allocations.
Fast Weight Updates

- Each thread computes weight updates and updates the shared model weights without locking
- Weight updates: associative and commutative
- Neural networks: overcome small noise
- Risk: modify weights based on stale weight values
- Training result: Convergence
Reducing Memory Copies

- **Problem**
  - Training data values need to be communicated across neuron layers
  - Model partitioned: non-local communication
  - Copy data values: expensive memory copies

- **Goal**
  - Reduce the memory bandwidth and CPU requirements for model training

- **Solution**
  - Uniform optimized interface: accelerate the data communication
  - Non-local communication: build network library, accept pointer to the relevant block of neurons whose outputs need communication
  - Static model partitioning across machines: optimize communication
  - Reference counting: support asynchronous network IO.
Memory System Optimizations

Maximal utilization of the floating point units on a machine

- Partition models: working sets for the model layers fit in L3 cache (bandwidth > main memory)

Optimize computation for cache locality

- Problem: forward evaluation and back-propagation computation have competing locality requirements (prefer a row major or column major layout for the layer weight matrix)
- Solution: 2 custom hand-tuned assembly kernels
  - pack and block the data appropriately
  - fully utilize vector units for the matrix multiply operations
Mitigating the Impact of Slow Machines

• **Impact 1:** The speed of processing an image is limited by slow machines
  • Solution:
    • Allow threads to process multiple images in parallel.
    • Dataflow framework: trigger progress on individual images based on arrival of data from remote machines.

• **Impact 2:** Need to wait for all training images to be processed at the end of an epoch (compute the model prediction error; need additional training epoch?)
  • Solution:
    • End an epoch whenever a specified fraction of the images are completely processed.
    • Randomize the image processing order for each epoch: ensure that the same set of images are not skipped each epoch
    • Waiting for 75% of the model replicas to complete processing all their images before declaring the training epoch complete
    • Result: speed up 20%, no impact on prediction accuracy.
Parameter Server Communication

• Two communication protocols

• Locally computes and accumulates the weight updates in a buffer.
  • Periodically send to the parameter server machines when hundreds images have been processed.
  • Parameter server machines directly apply accumulated updates to the stored weights.
  • Convolutional layers (weight sharing: volume of weights is low)

• Send the activation and error gradient vectors to the parameter server machines (not weight update)
  • Minimize communication traffic volume
  • Offloads computation: From model training machines (CPU is heavily utilized) to parameter server machines (CPU is underutilized)
  • Fully connected layers (more weights)
Global Parameter Server

- Throughput Optimizations
- Delayed Persistence
- Fault Tolerant Operation
- Communication Isolation
## Throughput Optimizations

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model parameters partition</strong></td>
<td>• Divided into 1 MB sized shards (contiguous partition), hashed into storage buckets distributed equally among the parameter server machines</td>
</tr>
<tr>
<td><strong>Improves the spatial locality of update processing</strong></td>
<td>• Distribution helps with load balancing</td>
</tr>
<tr>
<td><strong>Improves temporal locality &amp; relieves pressure on the L3 cache</strong></td>
<td>• Apply all updates in a batch to a block of parameters before moving to next block in the shard</td>
</tr>
<tr>
<td><strong>Use SSE/AVX instructions for applying the update</strong></td>
<td>• SSE(Streaming SIMD extensions)/AVX(Advanced Vector Extensions)</td>
</tr>
<tr>
<td><strong>NUMA aware processing</strong></td>
<td>• Shards allocated on a specific NUMA node, all update processing for the shard is localized to that NUMA node.</td>
</tr>
<tr>
<td><strong>Lock free data structures</strong></td>
<td>• Queues and hash tables in high traffic execution paths</td>
</tr>
<tr>
<td></td>
<td>• Speed up network, update, and disk IO processing.</td>
</tr>
<tr>
<td><strong>Lock free memory allocation</strong></td>
<td>• Buffers allocated from pools (4KB~32MB)</td>
</tr>
</tbody>
</table>
Delayed Persistence

Decouple durability from the update processing path

Allow for high throughput serving to training nodes.

Parameter storage is modelled as a write back cache

● Dirty chunks flushed asynchronously in the background.
  Tolerable potential data loss: DNN models allow small lost updates.
  ● Updates can be effectively recovered by retraining the model on the appropriate input data.

Allows for compressed writes to durable storage

Many updates can be folded into a single parameter update

Allows update cycles to catch up to the current state of the parameter shard
Fault Tolerant Operation

• 3 copies of each parameter shard stored on different parameter servers
  • 1 primary shard: actively served
  • 2 secondary shard: fault tolerance

• Paxos cluster: parameter servers controlled by PS controller machines
  • PS Controller: maintain configuration (mapping of shards and roles to parameter servers)
  • Handout bucket assignments (primary: lease, secondary: lease information) to parameter servers.
  • Receives heart beats from parameter server machines and relocates buckets from failed machines evenly to other active machines
  • Clients (model training machines): contact controller to determine request routing for parameter shards.

• Primary parameter server
  • Send heart beats to secondary machines
  • Replicates changes to shards to secondary machines.

• Secondary parameter server
  • Check lease information of the bucket before committing
  • Prolonged absence of heart beat form primary: Initiate a role change proposal to be primary to the controller
Communication Isolation

- Parameter update & persistence are decoupled
- Isolate 2 paths into two 10Gb NICs
- Isolate administrative traffic from controller to 1Gb NIC
Benchmarks

**MNIST**

- 28x28 images of the 10 handwritten digits
- 60,000 training images and 10,000 test images

**ImageNet**

- 15 million labeled high-resolution images
- 22,000 different categories
- down-sampled, fixed 256x256 resolution
- half for training, the other half for testing
- top-1 accuracy
Baseline Performance and Accuracy

**Model Training Node Performance**
- Single model training machine, no parameter server
- Benchmark: MNIST model

**Parameter Server Node Performance**
- Multi-core scaling of a single parameter server
- Benchmark: ImageNet 22K model

**Trained Model Accuracy**
- Benchmark: MNIST model
- Standard model, ~2.5 million connections
Baseline Performance and Accuracy

• Model Training Node Performance

- Single model training machine, no parameter server
- MNIST model, 2.5 million connections
- Training speed = Model connections * Training examples * Number of Epochs / (Wall clock time)

Result: excellent scaling
Baseline Performance and Accuracy

- Parameter Server Node Performance

- Multi-core scaling of a single parameter server
- ImageNet 22K

Limiting factor: Network bandwidth

Bottleneck: Memory bandwidth

Good scaling
Baseline Performance and Accuracy

• Trained Model Accuracy
  • MNIST benchmark, without any transformation
  • Standard model:
    • 2 convolutional layers followed by two fully connected layers
    • A final 10 class softmax output layer
    • 2.5 million connections

<table>
<thead>
<tr>
<th>Systems</th>
<th>MNIST Top-1 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodfellow et al [12]</td>
<td>99.55%</td>
</tr>
<tr>
<td>Adam (Asynchronous)</td>
<td>99.63%</td>
</tr>
<tr>
<td>Adam (synchronous)</td>
<td>99.39%</td>
</tr>
</tbody>
</table>

System Scaling and Accuracy

- **Scaling Model Size with more Workers**
  - Maximum size model efficiently train on a given multi-machine configuration.

- **Scaling with Model Replicas**
  - Impact of adding more model replicas

- **Trained Model Accuracy**
  - Train a large and deep convolutional network for the ImageNet 22K
System Scaling and Accuracy

• Scaling Model Size with more workers

- Use single training epoch of ImageNet
- A single model replica with no parameter server
- Increase the model size until training speed decrease
- **Result:** model size increase super-linearly
System Scaling and Accuracy

- **Scaling with Model Replicas**

  - Each replica contains 4 machines
  - ImageNet model partitioned across these machines.
  - Parameter server: 20 machines
  - **Result:** scales well with additional replicas.

![Graph showing scaling with additional replicas](image-url)
System Scaling and Accuracy

• Trained Model Accuracy
  - ImageNet 22K category
  - Deep convolutional network
    - 5 convolutional layers followed by 3 fully connected layers
    - 1rst, 2nd and 5th convolutional layers followed by a 3x3 max-pooling layer.
    - Fully-connected layers contain 3000 hidden units.
    - A final 22,000-way softmax output layer
    - Convolutional kernels sizes: 3x3~7x7
    - Convolutional feature map sizes: 120~600
    - Resulting model > 2 billion connections

• Servers
  - 4 image servers, 48 model training machines, 10 parameter servers
  - 16 model replicas (4 machines/replica)
System Scaling and Accuracy

• Trained Model Accuracy (Cont’d)

<table>
<thead>
<tr>
<th>Systems</th>
<th>ImageNet 22K Top-1 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Le et al. [18]</td>
<td>13.6%</td>
</tr>
<tr>
<td>Le et al. (with pre-training) [18]</td>
<td>15.8%</td>
</tr>
<tr>
<td>Adam</td>
<td>29.8%</td>
</tr>
</tbody>
</table>

1Bn connection model
2000 machines
1 week

+10 million unlabeled images
1000 machines
3 days

> 2Bn connection model
62 machines
> 13.6% (1 day)
29.8% (10 days)
System Scaling and Accuracy

- Trained Model Accuracy (Cont’d)

<table>
<thead>
<tr>
<th>Systems</th>
<th>ImageNet 22K Top-1 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Le et al. [18]</td>
<td>13.6% (1Bn connection)</td>
</tr>
<tr>
<td>Le et al. (with pre-training) [18]</td>
<td>15.8%</td>
</tr>
<tr>
<td>Adam</td>
<td>29.8% (&gt;2Bn connection)</td>
</tr>
</tbody>
</table>

Asynchrony increases accuracy

<table>
<thead>
<tr>
<th>Systems</th>
<th>MNIST Top-1 Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goodfellow et al [12]</td>
<td>99.55%</td>
</tr>
<tr>
<td>Adam (Asynchronous)</td>
<td>99.63%</td>
</tr>
<tr>
<td>Adam (synchronous)</td>
<td>99.39%</td>
</tr>
</tbody>
</table>

Training larger models increases accuracy

29.8% > 2 Bn connections

24% 1.1 Bn connections

---

Summary

High multi-threaded scalability on a single machine
- Permit threads to update local parameter weights without locks.

Good multi-machine scalability
- Minimize communication traffic by performing the weight update computation on the parameter server machines
- Asynchronous batched updates to parameter values that take advantage of these updates being associative and commutative.

Enables training models to high accuracy
- Exploit its efficiency to train very large models
- Leverage asynchrony to further improve accuracy
Using Google Cloud Machine Learning to predict clicks at scale
Andreas Sterbenz
A managed service that enables developers and data scientists to build and bring superior machine learning models to production.

https://cloud.google.com/ml-engine/

Focus on models, not operations.

google Cloud with Machine learning

TensorFlow
Keras
XGBoost
Predict display ad clicks

• Train model to predict display ad clicks on Criteo Labs clicks logs.
  • Over 1TB in size
  • Millions of display ads.
  • Contain feature values and click feedback
  • Each line: one training example as tab-separated-values (TSV) with a mix of numerical and categorical (string valued) features + a label column indicating if the ad was clicked.

• Train several models using different machine-learning techniques to predict the clickthrough rate.

• Code: https://github.com/GoogleCloudPlatform/cloudml-samples/tree/master/criteo_tft
Set up Cloud environment

- Set up GCP project
  - Set up a GCP account
  - Create a GCP project
  - Activate the Cloud ML Engine API
- Set up environment
  - Install the Cloud SDK
  - Remote environment on Cloud Shell

https://cloud.google.com/ml-engine/docs/tensorflow/getting-started-training-prediction
Preprocessing the data

• **Start the Cloud Dataflow job**
  ```
  PROJECT=$(gcloud config list project --format "value(core.project)")
  BUCKET="gs://${PROJECT}-ml"
  GCS_PATH="${BUCKET}/${USER}/largeclicks"
  python preprocess.py --training_data "${GCS_PATH}/day_*" \
    --eval_data "${GCS_PATH}/eval_day_*" \
    --output_dir "${GCS_PATH}/preproc" \
    --project_id $PROJECT \
    --cloud
  
  gsutil ls "${GCS_PATH}/preproc"
  ```

• **Autoscaling**: automatically chooses the appropriate number of machines to use

• Output: compressed **TFRecords** files is written as to Google Cloud Storage
Training a linear model

• Linear classifier

• Simple feature engineering
  • Different feature columns
  • Integer columns: bucketized column
    column = tf.contrib.layers.bucketized_column(
      tf.contrib.layers.real_valued_column(
        'int-feature-{}'.format(index),
        default_value=-1,
        dtype=tf.int64),
      boundaries)
  • Categorical columns: sparse integerized column
    column = tf.contrib.layers.sparse_column_with_integerized_feature(
      column_name, bucket_size=vocab_size, combiner='sum')
Training a linear model (Cont’d)

• Run “gcloud” on cloud ML engine

```bash
> JOB_ID="largeclicks_linear_${USER}_${(date +%Y%m%d_%H%M%S)}"
> gcloud beta ml jobs submit training "$JOB_ID"
   --module-name trainer.task
   --package-path trainer
   --staging-bucket "$BUCKET"
   --region us-central1
   --config config-large.yaml
   --async
   --dataset large
   --model_type linear
   --ignore_crosses
   --l2_regularization 1000
   --output_path "${GCS_PATH}/output/${JOB_ID}"
   --metadata_path "${GCS_PATH}/preproc/metadata.json"
   --eval_data_paths "${GCS_PATH}/preproc/features_eval*"
   --train_data_paths "${GCS_PATH}/preproc/features_train*"
```

**Config-large.yaml**

```
trainingInput:
  scaleTier: CUSTOM
  masterType: large_model
  workerType: complex_model_m
  parameterServerType: large_model
  workerCount: 60
  parameterServerCount: 29
```

For Python training module trainer.task
Training a linear model (Cont’d)

• 60 worker machines
• 29 parameter machines
• Training time: 70 minutes
• Evaluation loss: 0.1293
Improvement: Adding crosses

• Improves the model’s predictive capability
• Sparse features can be crossed (conjuncted or combined) in order to form non-linearities, and then fed into linear models.
• Adding crosses enables algorithm to learn which non-linear combinations of features are relevant

```python
column = tf.contrib.layers.crossed_column(  
    [columns[index - 1] for index in cross],  
    hash_bucket_size=int(1e6),  
    combiner='sum')
```

• Evaluation loss: **0.1272** vs. 0.1293
• Training time: **2.5 hours** vs. 70 minutes
Training a deep neural network

- Advantages of DNNClassifier:
  - Learn complex feature combinations automatically
  - No need to specify crosses.

- Feature columns:
  
  ```python
  column =
  tf.contrib.layers.sparse_column_with_integerized_feature(
      column_name, bucket_size=vocab_size, combiner='sum')

  embedding_size = int(math.floor(6 * vocab_size**0.25))
  embedding = tf.contrib.layers.embedding_column(column,
                                                   embedding_size,
                                                   combiner='mean')
  ```

  ```python
  estimator =
  tf.contrib.learn.DNNClassifier(
      hidden_units=argv.hidden_units,
      feature_columns=columns,
      model_dir=output_dir)
  ```
Training a deep neural network (Cont’d)

• Run “gcloud” on cloud ML engine

```bash
> gcloud beta ml jobs submit training "$JOB_ID" \
   --module-name trainer.task \
   --package-path trainer \
   --staging-bucket "$BUCKET" \
   --region us-central1 \
   --config config-large.yaml\ 
   --async \ 
   --dataset large \ 
   --model_type deep \ 
   --hidden_units 1062 1062 1062 1062 1062 1062 1062 1062 1062 1062 1062 \ 
   --batch_size 512 \ 
   --num_epochs 1 \ 
   --output_path "${GCS_PATH}/output/${JOB_ID}" \ 
   --metadata_path "${GCS_PATH}/preproc/metadata.json" \ 
   --eval_data_paths "${GCS_PATH}/preproc/features_eval*" \ 
   --train_data_paths "${GCS_PATH}/preproc/features_train*
```

11 fully connected layers with 1,062 neurons each
Training time: 26 hours
Evaluation loss: 0.1257
Comparing the results

<table>
<thead>
<tr>
<th>Modeling technique</th>
<th>Training time</th>
<th>Loss</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear model</td>
<td>70 minutes</td>
<td>0.1293</td>
<td>0.7721</td>
</tr>
<tr>
<td>Linear model with crosses</td>
<td>142 minutes</td>
<td>0.1272</td>
<td>0.7841</td>
</tr>
<tr>
<td>Deep neural network, network, 1 epoch</td>
<td>26 hours</td>
<td>0.1257</td>
<td>0.7963</td>
</tr>
<tr>
<td>Deep neural network, network, 3 epochs</td>
<td>78 hours</td>
<td>0.1250</td>
<td>0.8002</td>
</tr>
</tbody>
</table>

Graph of loss for the different models vs training time in hours.
Summary

• Google Cloud ML enable users focus on models, not operation
• Cloud Machine Learning and Tensorflow make it easy to train models on very large amounts of data
• Seamlessly switch between different types of models with different tradeoffs for training time and loss
Thank you!