A Brief Introduction to Deep Learning
Artificial Intelligence Landscape
Example Deep Learning Algorithms

- Multilayer perceptron (MLP)
- Recurrent neural network (RNN)
- Convolutional neural network (CNN)
Convolutional Neural Networks (CNN)

- Effective for computer vision
- Fewer parameters than fully connected networks
- Translational invariance
- Classic networks: LeNet-5, AlexNet, VGG
Graphics Processing Units (GPUs)

• Great at performing a lot of simple computations such as matrix operations
• Well suited to deep learning algorithms
Single Node Multi-GPU
Greenplum Database and Apache MADlib
Multi-Node Multi-GPU

SQL

Master Host

Standby Master

Interconnect

Node1
Segment Host
Keras
TensorFlow

Node2
Segment Host
Keras
TensorFlow

Node3
Segment Host
Keras
TensorFlow

NodeN
Segment Host
Keras
TensorFlow

In-Database Functions

Massively Parallel Processing

Machine learning & statistics & math & graph & utilities
## Deep Learning on a Cluster

<table>
<thead>
<tr>
<th>Num</th>
<th>Approach</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Distributed deep learning</td>
<td>Train single model architecture across the cluster. Data distributed (usually randomly) across segments.</td>
</tr>
<tr>
<td>2</td>
<td>Data parallel models</td>
<td>Train same model architecture in parallel on different data groups (e.g., build separate models per country).</td>
</tr>
<tr>
<td>3</td>
<td>Hyperparameter tuning</td>
<td>Train same model architecture in parallel with different hyperparameter settings and incorporate cross validation. Same data on each segment.</td>
</tr>
<tr>
<td>4</td>
<td>Neural architecture search</td>
<td>Train different model architectures in parallel. Same data on each segment.</td>
</tr>
</tbody>
</table>
Workflow
Data Loading and Formatting

Raw images

Numpy arrays
- LxWx3 for color
- LxWx1 for b&w

CSV files
- 1 image per line
- n images per file

Greenplum table
- id | image | class
- 1 | {...} | airplane
- 2 | {...} | car
- ...

Greenplum table
- id2 | images | class
- 1 | {...}{...} | {airplane, car, ...}
- 2 | {...}{...} | {horse, boat, ...}
- ...

Shuffled, packed and distributed

Tooling in dev

Python w/ Keras or PIL or ...

psql CDPY

MADlib pre-processor

Ready for deep learning
Iterative Model Execution

Stored Procedure for Model

```python
model = init(...) 
WHILE model not converged 
    model = SELECT 
        model.aggregation(...) 
    FROM 
        data table 
ENDWHILE
```

1. Transition Function
   Operates on tuples or mini-batches to update transition state (model)

2. Merge Function
   Combines transition states

3. Final Function
   Transforms transition state into output value

Broadcast
Distributed Deep Learning Methods

• Open area of research*
• Methods we have investigated so far:
  – Simple averaging
  – Ensembling
  – Elastic averaging stochastic gradient descent (EASGD)

* Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis
Some Results
Testing Infrastructure

- Google Cloud Platform (GCP)
- Type n1-highmem-32 (32 vCPUs, 208 GB memory)
- NVIDIA Tesla P100 GPUs
- Greenplum database config
  - Tested up to 20 segment (worker node) clusters
  - 1 GPU per segment
CIFAR-10

- 60k 32x32 color images in 10 classes, with 6k images per class
- 50k training images and 10k test images

https://www.cs.toronto.edu/~kriz/cifar.html
Places

- Images comprising ~98% of the types of places in the world
- Places365-Standard: 1.8M images from 365 scene categories
- 256x256 color images with 50 images/category in validation set and 900 images/category in test set

http://places2.csail.mit.edu/index.html
6-layer CNN - Test Set Accuracy (CIFAR-10)

Method: Model weight averaging

https://blog.plon.io/tutorials/cifar-10-classification-using-keras-tutorial/
6-layer CNN - Runtime (CIFAR-10)

Method: Model weight averaging
Method: Model weight averaging

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(32,32,3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(n_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer=Adam(), metrics=['categorical_accuracy'])
1-layer CNN - Runtime (CIFAR-10)

Method: Model weight averaging
Method: Model weight averaging
VGG-11 (Config A) CNN - Runtime (Places50)

Method: Model weight averaging
Ensemble with Places365

Segment 1

Segment 2

Segment n

AlexNet

Method: Model weight averaging with simple ensemble CNN
1-layer CNN - Test Set Accuracy (Places365) (20 segments)

Method: Elastic averaging stochastic gradient descent (EASGD)

model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=(32,32,3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(n_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['categorical_accuracy'])
Lessons Learned and Next Steps
Lessons Learned

- Distributed deep learning can potentially run faster than single node, to achieve a given accuracy
- Deep learning in a distributed system is challenging (but fun!)
- Database architecture imposes some limitations compared to Linux cluster
Infrastructure Lessons Learned

• Beware the cost of GPUs on public cloud!
• Memory management can be finicky
  – GPU initialization settings and freeing TensorFlow memory
• GPU configuration
  – Not all GPUs available in all regions (e.g., Tesla P100 avail in us-east but not us-west on GCP)
  – More GPUs does not necessarily mean better performance
• Library dependencies important (e.g., cuDNN, CUDA and Tensorflow)
Future Deep Learning Work*

- 1.16 (Q1 2019)
  - Initial release of distributed deep learning models using Keras with TensorFlow backend, including GPU support
- 2.0 (Q2 2019)
  - Model versioning and model management
- 2.x (2H 2019)
  - More distributed deep learning methods
  - Massively parallel hyperparameter tuning
  - Support more deep learning frameworks
  - Data parallel models

*Subject to community interest and contribution, and subject to change at any time without notice.
Thank you!
Apache MADlib Resources

- **Web site**

- **Wiki**
  - [https://cwiki.apache.org/confluence/display/MADLIB/Apache+MADlib](https://cwiki.apache.org/confluence/display/MADLIB/Apache+MADlib)

- **User docs**

- **Jupyter notebooks**
  - [https://github.com/apache/madlib-site/tree/asf-site/community-artifacts](https://github.com/apache/madlib-site/tree/asf-site/community-artifacts)

- **Technical docs**
  - [http://madlib.apache.org/design.pdf](http://madlib.apache.org/design.pdf)

- **Pivotal commercial site**
  - [http://pivotal.io/madlib](http://pivotal.io/madlib)

- **Mailing lists and JIRAs**
  - [https://mail-archives.apache.org/mod_mbox/incubator-madlib-dev/](https://mail-archives.apache.org/mod_mbox/incubator-madlib-dev/)
  - [http://mail-archives.apache.org/mod_mbox/incubator-madlib-user/](http://mail-archives.apache.org/mod_mbox/incubator-madlib-user/)
  - [https://issues.apache.org/jira/browse/MADLIB](https://issues.apache.org/jira/browse/MADLIB)

- **PivotalR**
  - [https://cran.r-project.org/web/packages/PivotalR/index.html](https://cran.r-project.org/web/packages/PivotalR/index.html)

- **Github**
  - [https://github.com/apache/madlib](https://github.com/apache/madlib)
  - [https://github.com/pivotalsoftware/PivotalR](https://github.com/pivotalsoftware/PivotalR)
<table>
<thead>
<tr>
<th>Num</th>
<th>Category</th>
<th>Lessons learned</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cost</td>
<td>Beware the cost of scale testing with GPUs</td>
<td>Easy to spend $30K/month on scale testing on 2-3 clusters from 1-20 segments (worker nodes) on GCP with Tesla P100 GPUs.</td>
</tr>
<tr>
<td>2</td>
<td>GPU memory management</td>
<td>During initialization, set <code>gpu_options.allow_growth=False</code></td>
<td>There are a lot of blog posts and forum answers where people recommend setting this to True. We tried that at first, but concluded it’s not the best idea for our purposes. Setting it to True means it only requests a small amount of memory when you initialize the tensorflow session, but then every time you perform any operations (fit, evaluate, etc.) it will try to allocate more memory as needed. This is dangerous because you never know when you will run out of memory, especially if multiple GPU’s are sharing. It’s unpredictable and the errors are very difficult to sort out after it happens. Much cleaner to diagnose is setting it to False, where a fixed fraction of the GPU memory (see next option: <code>per_process_gpu_memory_fraction</code>) is allocated up front, and never grown after that. Any issues will come up right at the beginning, not suddenly after you’ve trained half the dataset!</td>
</tr>
<tr>
<td>3</td>
<td>GPU memory management</td>
<td>During initialization, set <code>gpu_options.per_process_gpu_memory_fraction</code></td>
<td>This tells each segment what fraction of the GPU memory to use. If every segment has its own GPU, then pick something close to 1.0. We have been using 0.9 just in case there are some small things that need to run (such as nvidia-smi tool, used for monitoring GPU memory usage.) If you want more than one segment to share a GPU, this has to be less than 0.5.</td>
</tr>
<tr>
<td>4</td>
<td>CPU memory management</td>
<td>CPU memory needs not always crystal clear.</td>
<td>All the hosts on our two gpus clusters are high memory machines (208 GB). We’ve never seen it use more than 50GB or so in hotop, but at some point we were getting crashes with only 120GB and raising it to 208GB.</td>
</tr>
<tr>
<td>5</td>
<td>TensorFlow memory management</td>
<td>Be very careful about freeing TensorFlow memory.</td>
<td>The only guaranteed solution that works is to make sure the process that runs TensorFlow dies before any new TF session is started (killing the process will free the memory). Our current solution includes creating a TF session for each iteration and closing it at the end of the iteration.</td>
</tr>
<tr>
<td>6</td>
<td>Multi-GPU</td>
<td>Increasing GPUs per segment did not help beyond 2.</td>
<td>We’ve tried running Keras on only 1 segment per host with 1, 2, 4, and 8 GPU’s per segment, and compared performance. 2 was a nice improvement from 1, but beyond that 4 and 8 didn’t seem to add much.</td>
</tr>
<tr>
<td>7</td>
<td>GPU selection</td>
<td>Certain GPUs n/a in certain zones</td>
<td>Availability of a certain type of GPU may depend on the region in GCP. For ex Tesla P100 is not available in us-east but is available in us-west.</td>
</tr>
<tr>
<td>8</td>
<td>GPU selection</td>
<td>We are currently using the Tesla P100 GPU which is more expensive but way faster than Tesla K80.</td>
<td>We saw segmentation faults with the GPU setup if the TensorFlow version is newer than what is supported by cuDNN library. One solution is to downgrade the TF version or alternatively upgrade the cuDNN version.</td>
</tr>
<tr>
<td>9</td>
<td>Library dependencies</td>
<td>CUDA, cuDNN and TensorFlow versions must be in sync.</td>
<td>The problem was that the cuDNN library was older than what TF 1.11 expected. We had installed tensorflow using pip install tensorflow-gpu which always gives us the latest TF. I downgraded the TF version to 1.9 so that it matches the cuDNN library pip install --upgrade --force-reinstall tensorflow-gpu==1.9.0 --user</td>
</tr>
</tbody>
</table>
SQL Interface

```sql
-- fit
SELECT madlib_keras_fit (  
  source_table,  
  model,  
  dependent_varname,  
  independent_varname,  
  model_arch_table,  
  model_arch_id,  
  compile_params,  
  fit_params,  
  num_iterations,  
  num_classes,  
  use_gpu,  
  validation_table,  
  name,  
  description,  
  initial_weights,  
  distributed  
) ;

-- optional params
-- select GPU or CPU
-- data to validate,
-- user entered name
-- user entered description
-- weights to initialize for training
-- run model centralized on one node or distributed

-- predict
SELECT madlib_keras_predict (  
  model,  
  model_id  
  data_table,  
  output_table,  
  id_col_name  
) ;

-- model to predict
-- id if model repository used
-- table to predict on
-- table to write predictions
-- row id in test table
```
Greenplum Integrated Analytics

Data Transformation

Text

Geospatial

Data Science Productivity Tools

Traditional BI

Machine Learning

Deep Learning

Graph
Scalable, In-Database Machine Learning

Apache MADlib: Big Data Machine Learning in SQL

Open source, top level Apache project

For PostgreSQL and Greenplum Database

Powerful machine learning, graph, statistics and analytics for data scientists

- Open source: https://github.com/apache/madlib
- Downloads and docs: http://madlib.apache.org/
- Wiki: https://cwiki.apache.org/confluence/display/MADLIB/
MADlib project was initiated in 2011 by EMC/Greenplum architects and Professor Joe Hellerstein from University of California, Berkeley.

UrbanDictionary.com:

mad (adj.): an adjective used to enhance a noun.

1- dude, you got skills.
2- dude, you got mad skills.
### Supervised Learning
- Neural Networks
- Support Vector Machines (SVM)
- Conditional Random Field (CRF)
- Regression Models
  - Clustered Variance
  - Cox-Proportional Hazards Regression
  - Elastic Net Regularization
  - Generalized Linear Models
  - Linear Regression
  - Logistic Regression
  - Marginal Effects
  - Multinomial Regression
  - Naive Bayes
  - Ordinal Regression
  - Robust Variance
- Tree Methods
  - Decision Tree
  - Random Forest

### Graph
- All Pairs Shortest Path (APSP)
- Breadth-First Search
- Hyperlink-Induced Topic Search (HITS)
- Average Path Length
- Closeness Centrality
- Graph Diameter
- In-Out Degree
- PageRank and Personalized PageRank
- Single Source Shortest Path (SSSP)
- Weakly Connected Components

### Unsupervised Learning
- Association Rules (Apriori)
- Clustering (k-Means)
- Principal Component Analysis (PCA)
- Topic Modelling (Latent Dirichlet Allocation)

### Nearest Neighbors
- k-Nearest Neighbors

### Utility Functions
- Columns to Vector
- Conjugate Gradient
- Linear Solvers
  - Dense Linear Systems
  - Sparse Linear Systems
- Mini-Batching
- PMML Export
- Term Frequency for Text
- Vector to Columns

### Sampling
- Balanced
- Random
- Stratified

### Time Series Analysis
- ARIMA

### Data Types and Transformations
- Array and Matrix Operations
- Matrix Factorization
  - Low Rank
  - Singular Value Decomposition (SVD)
- Norms and Distance Functions
- Sparse Vectors
- Encoding Categorical Variables
- Path Functions
- Pivot
- Sessionize
- Stemming

### Statistics
- Descriptive Statistics
  - Cardinality Estimators
  - Correlation and Covariance
  - Summary
- Inferential Statistics
  - Hypothesis Tests
  - Probability Functions

### Model Selection
- Cross Validation
- Prediction Metrics
- Train-Test Split

### Aug 2018
Execution Flow

Client → Database Server

- SQL
- String

Database Server → Master

- Stored Procedure
- Result Set

Master → Segment 1
- Aggregation

Segment 1 → Segment 2

... → Segment n

Tools:
- jupyter
- Apache Zeppelin
- pgAdmin
- psql
Architecture

User Interface

- High-Level Iteration Layer (iteration controller)
- Functions for Inner Loops (implements ML logic)
- Low-level Abstraction Layer (array operations, C++ to DB type-bridge, …)
- C API (Greenplum, PostgreSQL, HAWQ)

C++

Python

SQL

RDBMS

Built-in Functions

Built-in Functions

Functions for Inner Loops

Low-level Abstraction Layer

C++

Eigen

Boost