Image Classification in Greenplum Database Using Deep Learning

Oliver Albertini[1], Divya Bhargov[1], Alexander Denissov[1], Francisco Guerrero[1], Nandish Jayaram[3], Nikhil Kak[1], Ekta Khanna[1], Orhan Kislal[1], Arun Kumar[2], Frank McQuillan[1], Lisa Owen[1], Venkatesh Raghavan[1], Domino Valdano[1], Yuhao Zhang[2*]
[1]VMware, [2] University of California San Diego, [3]Intuit

ABSTRACT

Artificial neural networks can be used to create highly accurate models in domains such as language processing and image recognition. For example, convolutional neural networks (CNN) can be used to compare satellite images taken over time to monitor changes in rainforest cover in the Amazon basin, or to assess the damage caused by wildfires in order to help direct relief and conservation efforts. To address these important problems, we need tools that allow subject matter experts to build models and query image data stored in heterogeneous sources, and the ability to expand or contract computational resources as data volumes change. In this demonstration, we showcase the use of Greenplum as an end-to-end platform for deep learning. We cover the Pivotal Extension Framework (PXF) to fetch and transform images from cloud object stores, and exploit graphics processing unit (GPU) cycles to run Apache MADlib deep learning methods.

KEYWORDS

Deep Learning, In-Database Analytics, Image Classification, Massively Parallel Processing Databases

1 INTRODUCTION

Advanced analytics, in its various forms, is rapidly growing in importance in many organizations. Apache MADlib [3] is an open source project that provides a rich library of machine learning methods from logistic regression, decision trees, and k-means, to deep learning algorithms such as CNN and Recurrent Neural Networks (RNN). To be scalable, these algorithms must execute within a database that (1) is capable of processing large amounts of data, (2) exploits parallelism through a distributed query execution engine and (3) utilizes specialized hardware such as GPUs to train models. Greenplum [2] is a massively parallel processing database that provides such capabilities and integrates seamlessly with Apache MADlib so that algorithms can run on data inside the database. In this work, we demonstrate a scalable solution for image classification in Greenplum Database using Apache MADlib.

[*]This is a joint work done at Pivotal Inc. Authors are ordered alphabetically.

Portland '20, June 2020, Portland, Oregon, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Images are increasingly being stored in disparate sources, namely, cloud object stores, transactional databases, Hadoop data lakes, and analytical data warehouses. Greenplum Platform Extension Framework facilitates a parallel, high throughput access to images from heterogeneous data sources. We convert these images into PostgreSQL arrays rather than loading them as their raw data type (JPG, TIFF, etc.). These arrays can then be passed to Apache MADlib, which can in turn use them with deep learning libraries like Keras and Tensorflow.

Next, we provide a high level architecture of Greenplum Database, Apache MADLib, and PXF.

1.1 Greenplum Data Warehouse

Greenplum Database [2] is a massively parallel processing (MPP) analytics database that adopts a shared-nothing architecture with multiple cooperating processors. Figure 1 shows the high level architecture of Greenplum. Greenplum handles storage and processing of large amounts of data by distributing the load across several servers to create an array of individual databases, all working together to present a single database image. The *master* host is the entry point to Greenplum; it is the entity to which clients connect and submit SQL statements. The master host coordinates work with other database instances, called *segments*, to handle data processing and storage.

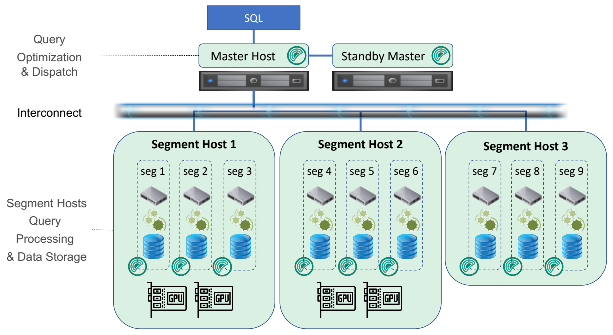


Figure 1: Example Greenplum Architecture with GPUs.

1.2 Apache MADLib Analytics Library

Apache MADLib [3] is an open source library of in-database analytic methods. It provides a suite of algorithms for machine learning, data mining, graph, and statistics that run at scale within a database engine, without the need for data import/export to other tools. MADlib has a SQL interface and currently supports Pivotal Greenplum Database and PostgreSQL. Most of the underlying algorithms are implemented internally in Python or C++ for efficiency. Currently, MADlib supports more than fifty principal algorithms, such as logistic regression, decision trees, neural networks, k-means clustering and deep learning. Since MADlib benefits from the parallel execution aspect of distributed systems like Greenplum, all the algorithms are designed in such a way that the computations can be parallelized and merged into one final result. One of the building blocks of MADlib is the use of user-defined aggregates (UDAs) and user-defined functions (UDFs). In general, aggregates are the natural way in SQL to implement mathematical functions that take as input the values of an arbitrary number of rows. UDFs are used to implement complex iterative methods which pass state across iterations intelligently. These UDFs in turn call UDAs containing the actual computation which are then executed on the individual segments. The results from these individual segments are then collected back on master which then returns a final result set. In this way, all large data movement is done within the database engine and the computation is efficiently parallelized.

1.3 Platform Extension Framework

PXF allows users to read and write data from/to heterogeneous data sources using external tables. PXF is able to scale by performing the read/write operations in parallel on all segments. PXF uses the **CREATE EXTERNAL TABLE** command to create an external table using the *pxf* protocol. Below is sample syntax for the case of image files stored on S3.

```
CREATE READABLE EXTERNAL TABLE image_table(
  fullpaths TEXT[], directories TEXT[],
  names TEXT[], image INT[]
)
LOCATION ('pxf://bucket/chip_images/*.png?
PROFILE=s3:image&SERVER=s3&BATCH_SIZE=64') FORMAT 'CSV';
```

- The *LOCATION* clause specifies the *pxf* protocol as a URI that identifies the path to the location of the external data.
- The *PROFILE* clause specifies the profile used to access the data. PXF supports profiles that access text, Avro, JSON, RCFile, Parquet, SequenceFile, ORC, and image data.
- The *SERVER* clause provides the server information such as location, access credentials, and other relevant properties.
- The *BATCH_SIZE* clause identifies the number of images PXF loads into one row.

When a Greenplum user runs a query against a PXF external table, a query plan is generated and then dispatched from the Greenplum master host to the Greenplum segments hosts. The *PXF Extension* running on each Greenplum segment process, in turn, forwards the request to the PXF Server running on the same host. A typical PXF deployment topology, as shown in Figure 2, places a *PXF Server* on each host that runs one or more Greenplum segment processes. A PXF Server receives at least as many requests as the number of segment processes running on the host, and potentially more if the query is complex and contains multiple processing slices. Each such request is assigned a PXF Server thread assigned to it.

Internally, PXF defines three interfaces to read and write data from external sources: the Fragmenter, Accessor, and Resolver. PXF supplies multiple implementations for these interfaces using different communication protocols and supports multiple data formats.

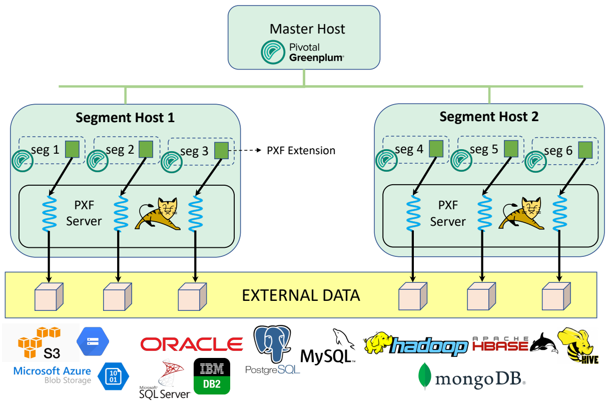


Figure 2: PXF Extension Framework Architecture.

- **Fragmenter** splits the external dataset into independent fragments that can be read in parallel. The fragmenter does not retrieve the actual data, it works only with metadata.
- **Accessor** is responsible for reading or writing data from/to external data sources, as well as converting data into individual records. PXF provides different accessors to connect to remote databases using JDBC, cloud object storage vendors, and remote Hadoop clusters using HDFS Client libraries.
- **Resolver** maps each field data type and value into a format that Greenplum or the external system understands.

2 TECHNICAL DETAILS

2.1 Reading Images with PXF

PXF's extensible architecture enables fetching image data from an external data source such as S3 or Hadoop. The main advantage of using PXF is its high concurrency and throughput when loading a large set of images into Greenplum. PXF has a built-in mechanism to fragment the total set of images into subsets that can be retrieved and processed in parallel by each Greenplum segment worker.

Image Representation. PXF reads each image file as a row of data, and translates the image, pixel by pixel, into a mathematical representation suitable for efficient storage and processing in Greenplum. PXF translates each pixel of an image into a three-element array representing red, green, and blue color values, respectively. For instance, a CMYK image of size $256 \times 256 \times 4$ can thus be represented as a three-dimensional integer array with a size of $256 \times 256 \times 3$.

Image Batching. With this approach, a single image can be stored in a database tuple using an integer array type. However, squeezing more data in a tuple is advantageous for machine learning as it reduces the number of table scans during model training. We attempt to fit multiple images into a single database tuple. Since Greenplum allows up to 1GB of data in a tuple we are able to fit 675 images, of size 256×256 pixels, in a single tuple.

PXF normally accesses a row of data from a single file, which it breaks into fields to form a database tuple and then finally transfers it to Greenplum. For image loading, however, PXF reads data from as many files as can fit into a single tuple.

Streaming Images. Constructing a database tuple out of all images at once before sending the tuple to Greenplum is inefficient. To address this, we implemented an iterator that appends one image at a time to the tuple at processing time and sends it to Greenplum before the whole tuple has been retrieved and processed.
Streaming Fragments. To reduce the memory footprint further, we stream the metadata during the fragmentation call. This optimization yielded a drop in PXF JVM memory usage from 8GB to 2GB when fetching 1.8 million images from the Places365 database[7].

2.2 Model Selection Using Apache MADlib

After images have been loaded into the database and transformed using PXF, the next step is to build an accurate classification model. Training deep neural networks is resource-intensive since hundreds of trials may be needed to generate a good model architecture and associated hyper-parameters. For example, trying 5 CNN architectures with 5 values each for learning rate and regularizer adds up to 125 combinations. This is the challenge of model selection, which is time consuming and expensive, especially if you are only training one model at a time. Massively parallel processing databases like Greenplum can have hundreds of segments, so we exploit this parallel compute architecture to address the challenge of model selection by training many model configurations in parallel.
Deep learning frameworks and GPUs on Greenplum. There are different ways to leverage GPUs in a database. One approach is to try to make certain database operations faster by parallelizing some portion of the workload (e.g., aggregations, sorting, grouping) and to reduce dependency on indexing and partitioning. This approach results in what are commonly called GPU databases, which are typically "ground up" development projects designed around GPUs. A different approach is to employ GPUs to power standard deep learning libraries on an existing database, without making changes to the query processing function of the database server itself. This is the approach we took with Greenplum: combine all of the capabilities of a mature, fully-featured MPP database with GPU acceleration to train deep learning models faster, using all of the data residing in the database.

Figure 1 shows GPUs attached to 2 out of the 3 segment hosts in the database cluster. Greenplum supports heterogeneous architectures like the one depicted because having GPUs attached to every host machine can be very costly. Deep learning frameworks and associated libraries like CUDA, cuDNN are installed on each host where GPUs reside. MADlib currently supports Keras with a TensorFlow backend, though other frameworks may be added.
Model hopper parallelism. To train many models at the same time we implement a novel approach from recent research called *model hopper parallelism* (MOP) [6]. MOP combines task and data parallelism by exploiting the fact that stochastic gradient descent (SGD), a widely used optimization method, is robust to data visit order.

The method works as follows. Suppose we have a set S of training configurations that we want to train for k epochs. The dataset is shuffled once and distributed to p workers (segments). We pick p training configurations from S and assign one configuration per worker where each configuration is trained for a single sub-epoch. When a worker completes a sub-epoch, it is assigned a new configuration that has not yet been trained on that worker. Thus, a model "hops" from one worker to another until all workers have been visited, which completes one epoch of SGD for each model. In this way we can train all S configurations for k epochs (Figure 3).

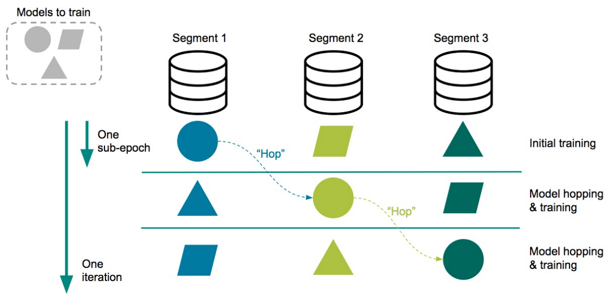


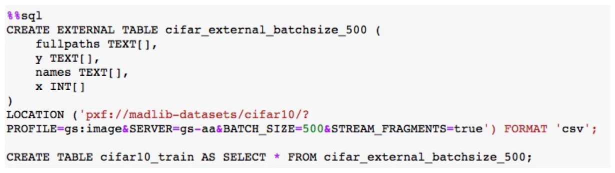
Figure 3: Model Hopper Parallelism.

API. Below are the two SQL queries needed to run MOP on Greenplum. The first loads model configurations you want to train into a model selection table. The second calls the MOP function to fit each of the models in the model selection table.

```
SELECT load_model_selection_table(
    'model_arch_table',  -- model architecture table
    'model_selection_table',  -- output table
    ARRAY[...],  -- model architecture ids
    ARRAY[...],  -- compile hyperparameters
    ARRAY[...]   -- fit hyperparameters);

SELECT madlib_keras_fit_multiple_model(
    'data',  -- data table
    'trained_models',  -- output table name
   'model_selection_table',  -- model selection table
    100,  -- number of iterations
    TRUE,-- use GPUs
    ...  -- optional parameters);
```

3 DEMONSTRATION PLAN

Demonstration Setup. We use a cluster of 4 hosts on Google Cloud Platform (GCP), each with 32 vCPUs, 150 GB of memory and 4 X NVIDIA Tesla P100 GPUs. Greenplum Database 5 and Apache MADlib 1.17 are installed on the cluster, with 4 segments (workers) configured per host; this means 16 workers in total. We use Keras 2.2.4 and TensorFlow 1.13.1.
Attendee Interactivity. To interactively demonstrate deep learning with Greenplum, we use a Jupyter Notebook, which is a common data science tool. Attendees can query and plot results for a trained model, such as loss and accuracy, similar to Figure 7. Attendees can also run inference using this model to display the top predicted classes for a new image, similar to Figure 8. For a more in-depth demonstration, attendees can modify the model architecture by adding or dropping layers, then retrain the model to see the effect on loss and accuracy. While models are training, attendees can view CPU, GPU and network activity for each host on the GCP console. More details on the end-to-end demonstration steps are shown below. (Note: to enable the demonstration to run quickly, we will provide smaller datasets to choose from.)



```
%%sql
CREATE EXTERNAL TABLE cifar_external_batchsize_500 (
    fullpaths TEXT[],
    y TEXT[],
    names TEXT[],
    x INT[]
)
LOCATION ('pxf://madlib-datasets/cifar10/?
PROFILE=gs:image&SERVER=gs-aa&BATCH_SIZE=500&STREAM_FRAGMENTS=true') FORMAT 'csv';

CREATE TABLE cifar10_train AS SELECT * FROM cifar_external_batchsize_500;
```

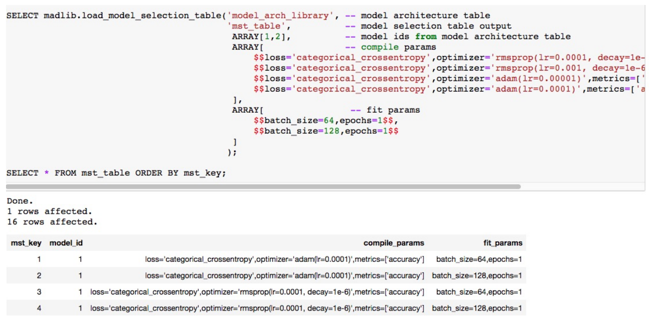
Figure 4: Screenshot of Jupyter Notebook for Loading Data

Step 1: Load image data with PXF. First we load image data from an external object store with PXF (Figure 4). We load 50K training images from the well known CIFAR-10 dataset of 32x32 color images in 10 classes [1]. This takes approximately 9 minutes on the demonstration cluster, which includes converting images to PostgreSQL arrays using the Python Imaging Library (PIL), normalizing RGB values, and packing 500 image arrays per row in the heap table.

Step 2: Load model selection tuples. We load models from [4] and [5] which have 553K and 1.25M trainable parameters, respectively. We use grid search to generate a set of 16 training configurations. Table 1 shows the details. Figure 5 shows the query to load the model selection table, along with the first few rows of the table indicating the compile and fit parameters.

Model architecture	Batch size	Optimizer	Learning rate
{ref [5], ref [4]}	{64, 128}	{Adam, RMSprop}	$\{10^{-3}, 10^{-4}\}$

Table 1: Workloads for Classifying CIFAR-10 Images



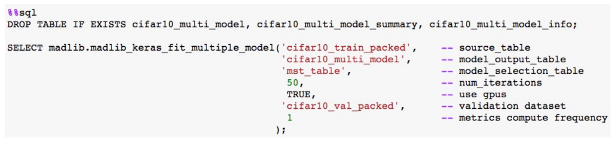
```
SELECT madlib.load_model_selection_table('model_arch_library', -- model architecture table
                                          'mst_table',         -- model selection table output
                                          ARRAY[1,2],          -- model ids from model architecture table
                                          ARRAY[               -- compile params
                                              $$loss='categorical_crossentropy',optimizer='rmsprop(lr=0.0001, decay=1e-
                                              $$loss='categorical_crossentropy',optimizer='rmsprop(lr=0.001, decay=1e-6
                                              $$loss='categorical_crossentropy',optimizer='adam(lr=0.00001)',metrics=['
                                              $$loss='categorical_crossentropy',optimizer='adam(lr=0.0001)',metrics=['a
                                          ],
                                          ARRAY[                -- fit params
                                              $$batch_size=64,epochs=1$$,
                                              $$batch_size=128,epochs=1$$
                                          ]
                                         );

SELECT * FROM mst_table ORDER BY mst_key;
```

```
Done.
1 rows affected.
16 rows affected.
```

mst_key	model_id	compile_params	fit_params
1	1	loss='categorical_crossentropy',optimizer='adam(lr=0.0001)',metrics=['accuracy']	batch_size=64,epochs=1
2	1	loss='categorical_crossentropy',optimizer='adam(lr=0.0001)',metrics=['accuracy']	batch_size=128,epochs=1
3	1	loss='categorical_crossentropy',optimizer='rmsprop(lr=0.0001, decay=1e-6)',metrics=['accuracy']	batch_size=64,epochs=1
4	1	loss='categorical_crossentropy',optimizer='rmsprop(lr=0.0001, decay=1e-6)',metrics=['accuracy']	batch_size=128,epochs=1

Figure 5: Defining and Loading Model Selection Tuples

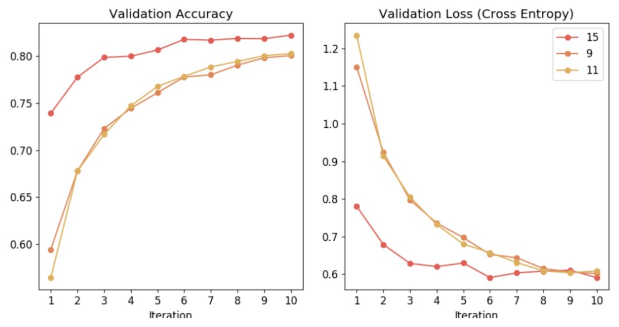
Step 2: Load model selection tuples. We load models from [4] and [5] which have 1.25M and 553K trainable parameters, respectively. We use grid search to generate a set of 16 training configurations. Table 1 shows the details. Figure 5 shows the query to load the model selection table, along with the first few rows of the table indicating the compile and fit parameters.



```
%%sql
DROP TABLE IF EXISTS cifar10_multi_model, cifar10_multi_model_summary, cifar10_multi_model_info;

SELECT madlib.madlib_keras_fit_multiple_model('cifar10_train_packed',    -- source_table
                                              'cifar10_multi_model',     -- model_output_table
                                              'mst_table',               -- model_selection_table
                                               50,                       -- num_iterations
                                               TRUE,                     -- use gpus
                                              'cifar10_val_packed',      -- validation dataset
                                               1                         -- metrics compute frequency
                                             );
```

Figure 6: Training Models

Step 3: Train models. Now we are ready to train the models, which is done using the query in Figure 6. Figure 7 shows the learning curves for the best 3 of the 16 configurations trained. Note that the chart shows 10 iterations; however we use 5 passes over the data per sub-epoch, so we are effectively doing 50 total passes over the data during training. The best model achieves 82.2% accuracy on the validation set, though this could likely be improved with tuning. Training takes approximately 27 minutes on the demonstration cluster. If we had used a single host only with 4 segments, training would take approximately 50 minutes.



Training config. 15 - model=ref [7], RMSprop, learning rate=10^{-3}, batch size=64
Training config. 9 - model=ref [7], Adam, learning rate=10^{-4}, batch size=64
Training config. 11 - model=ref [7], RMSprop, learning rate=10^{-4}, batch size=64

Figure 7: Plot Results

```
dog 0.9532127
deer 0.025846407
bird 0.009315644
```

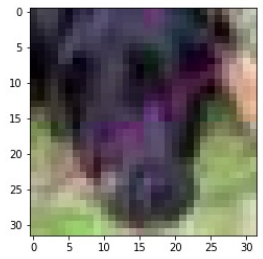


Figure 8: Example Inference with Top 3 Probabilities

Step 4: Inference. We take our best model from Step 3 to use for prediction. In this part of the demo, the attendee can run the inference on new images and see how well the model does at correctly classifying them (as shown in Figure 8). For the human, this may be an easy task. Let’s see how well the AI can do!

REFERENCES

[1] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2009. The CIFAR-10 dataset. https://www.cs.toronto.edu/ kriz/cifar.html. (2009).
[2] Greenplum. 2015. The World’s First Open-Source and Massively Parallel Data Platform. (2015). https://greenplum.org/
[3] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *PVLDB* 5, 12 (2012), 1700–1711.
[4] Jason Brownlee. 2019. How to Develop a CNN From Scratch for CIFAR-10 Photo Classification. https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10-photo-classification/. (2019).
[5] Keras Documentation. 2019. Train a simple deep CNN on the CIFAR10 small images dataset. https://keras.io/examples/cifar10_cnn/. (2019).
[6] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2019. Cerebro: Efficient and Reproducible Model Selection on Deep Learning Systems. In *Workshop on Data Management for End-to-End Machine Learning*. 6:1–6:4.
[7] Bolei Zhou, Agata Lapedriza, Aditya Khosla, Aude Oliva, and Antonio Torralba. 2017. Places: A 10 million Image Database for Scene Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017).