Deep Learning with Apache SystemML

Niketan Pansare[1], Michael Dusenberry[2], Nakul Jindal[2],
Matthias Boehm[1], Berthold Reinwald[1], Prithviraj Sen[1]
[1]IBM Research - Almaden
[2]IBM Spark Technology Center

1 INTRODUCTION

Deep Learning (DL) is a subfield of Machine Learning (ML) that focuses on learning hierarchical representations of data with multiple levels of abstraction using neural networks [15]. Recent advances in deep learning are made possible due to the availability of large amounts of labeled data, use of GPGPU compute, and application of new techniques (such as ReLU, batch normalization [12], dropout [17], residual block [10], etc.) that help deal with issues in training deep networks. In spite of the need to train on large datasets, there is a disconnect between the deep learning community and the big data community. To scale to a multi-node cluster, most deep learning frameworks (such as Caffe2, TensorFlow [2] and IBM's PowerAI DDL [6]) use custom communication libraries based on either MPI (such as IBM Spectrum MPI, Facebook's Gloo) or a custom networking protocol (such as Google RPC). Unlike popular big data frameworks (such as Apache Hadoop [9] and Apache Spark [18]), these communication libraries do not provide features such as resource sharing, multi-tenancy and fault-tolerance out of the box, making them difficult to deploy on shared production clusters. This leads to ineffective use of resources in an organization, often requiring two separate infrastructures (i.e. scale-up versus scale-out). This problem is even more severe when the data generated as part of the big data pipeline (ML, data preprocessing, data cleaning) needs to be consumed by the deep learning pipeline or vice versa, as the workload characteristics of a typical machine learning algorithm (i.e. memory-bound, BLAS level-2, sparse/ultra-sparse inputs (or feature matrix), etc.) are often different than that of a typical deep learning algorithm (i.e. compute-bound, BLAS level-3, dense inputs, etc.). Apache SystemML [4] aims to bridge that gap by seamlessly integrating with underlying big data frameworks and by providing a unified framework for implementing machine learning and deep learning algorithms.

In Apache SystemML, the ML algorithms are implemented using a high-level R-like language called DML (short for Declarative Machine Learning). DML improves the productivity of data scientists by enabling them to implement their ML algorithm with precise semantics as well as abstract data types and operations, independent of the underlying data representation or cluster characteristics. For the given DML script, SystemML's cost-based compiler automatically generates hybrid runtime execution plans that are composed of single-node and distributed operations depending on data and cluster characteristics such as data size, data sparsity, cluster size and memory configurations, while exploiting the capabilities of underlying data-parallel frameworks such as MapReduce or Spark. This allows for algorithm reusability across data-parallel frameworks, and simplified deployment for varying data characteristics and runtime environments, ranging from low-latency scoring to large-scale training.

2 DEEP LEARNING APIS

NN Library. As with other ML algorithms, users can implement their deep learning models using DML. SystemML 1.0 does not support automatic differentiation, thus the user has to write the DML code for the partial derivatives (i.e., the backward pass) of each layer. SystemML ships with a `Neural Network` (NN) library that supports 20+ pre-implemented layers (for example: conv2d, affine, relu, etc.) and 6 optimizers (namely Adagrad, Adam, RMSprop, SGD, SGD with momentum, and SGD with Nesterov momentum) to assist in writing algorithms. Each layer in the NN library has an `init`, `forward`, and `backward` function. The NN library is implemented entirely in DML, allowing the user to conveniently add custom layers and modify existing layers. The DML script for training a softmax classifier in SystemML using the minibatch gradient descent algorithm and the affine, softmax, and cross entropy layers is given below:

```
source("nn/layers/affine.dml") as affine
source("nn/layers/cross_entropy_loss.dml") as cross_entropy_loss
source("nn/layers/softmax.dml") as softmax
source("nn/optim/sgd.dml") as sgd
train = function(matrix[double] X, matrix[double] Y) {
  D = ncol(X)  # num features
  K = ncol(Y)  # num classes
  lr = 0.01; batch_size = 32; num_iter = nrow(X) / batch_size
  [W, b] = affine::init(D, K)
  for(i in 1:num_iter) {
    # Get batch
    beg = (i-1)*batch_size + 1; end = beg + batch_size
    X_batch = X[beg:end,]; y_batch = Y[beg:end,]
    # Perform forward pass
    scores = affine::forward(X_batch, W, b) # or X_batch %*% W + b
    probs = softmax::forward(scores)
    # Perform backward pass
    dprobs = cross_entropy_loss::backward(probs, y_batch)
    dscores = softmax::backward(dprobs, scores)
    [dX_batch, dW, db] = affine::backward(dout, X_batch, W, b)
    # Perform update
    W = sgd::update(W, dW, lr)
    b = sgd::update(b, db, lr)
  }
}
```

Keras2DML/Caffe2DML API. Currently active DL experts may be familiar with popular packages such as Keras [7] or Caffe [14] and may want to avoid learning a new language like DML. Or one may want to use publicly available pre-trained Keras/Caffe models and also leverage SystemML's distributed execution on Spark. To support such users, SystemML ships with python APIs - `Keras2DML` and `Caffe2DML` - that accept the DL models expressed in Keras or Caffe format and generate the equivalent DML script. Furthermore, these APIs allow a Python programmer to invoke SystemML's algorithms using a `scikit-learn` like API (which accepts NumPy arrays, SciPy matrices, or Pandas DataFrames) as well as Spark's MLPipeline API (which accepts `Spark DataFrames`). Since these APIs conform to MLPipeline's Estimator interface, they can be used in tandem with MLLib's feature extractors, transformers, scoring and cross-validation classes. Also, these APIs support loading pre-trained weights in the Caffe and Keras format for transfer learning and prediction. Assuming input matrices `X` and `Y` are NumPy arrays, the equivalent Python code for the above DML script is as follows:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from systemml.mllearn import Keras2DML
def train(X, Y):
  num_elems, D = X.shape; K = Y.shape[1]
  epochs = num_iter / num_elems
  model = Sequential()
  model.add(Dense(K, activation="softmax", input_dim=D))
  sgd = SGD(lr=0.01, momentum=0, nesterov=False)
  model.compile(loss="categorical_crossentropy", optimizer="sgd")
  sysml_model = Keras2DML(spark, model, input_shape=(D,1,1))
  sysml_model.set(train_algo="minibatch", test_algo="allreduce")
  sysml_model.fit(X, Y)
```

3 COMPILER AND RUNTIME SUPPORT

Tensor Representation. The primary data structure to store large amounts of data in DML is a 2D Matrix whereas in typical DL applications, a multi-dimensional matrix or a tensor is commonly used. To represent a tensor with more than 2 dimensions in DML, we linearize all but the first dimension. Therefore, a 4-dimensional tensor of shape $[N, C, H, W]$ is represented as a matrix with N rows and $C * H * W$ columns. This simplification helps us to leverage existing physical optimizations such as various sparse formats (COO, CSR and Modified CSR), blocking for handling out-of-core tensors, and broadcasting operations over scalars and vectors. Other optimizations such as sum-product optimization and code generation are also leveraged when applicable.

Builtin NN Functions. SystemML provides a set of built-in functions that are part of the language. Some common ones are `min`, `max`, `mean`, `sum`, and `solve`. Even though convolution and pooling (and their respective backward functions) can be expressed using existing DML looping constructs (as they are in the NN library), we've added them as built-in functions to enable efficient implementations. In addition to supporting ML use cases in SystemML, using the aforementioned representation of tensors and built-in functions, we support a variety of deep learning models in SystemML such as LeNet, feedforward nets, ResNets, autoencoders, simple RNNs, LSTMs, U-Net, SentenceCNN, etc.

Native BLAS Exploitation. To exploit the low-level CPU SIMD instructions, we extended the SystemML runtime to use the underlying BLAS (such as OpenBLAS and Intel MKL) for compute-intensive operations such as matrix-matrix multiplication and convolution operations. If Intel MKL is installed, the SystemML runtime uses the highly-tuned MKL-DNN primitives for the convolution operations.

GPU Backend. Most of the time in training a deep neural network is spent in matrix multiplications and convolution operations [13]. These operations can be performed, in case of dense inputs or intermediates, extremely efficiently on a GPU which often leads to a speedup of 10x as compared to CPU. Hence, it became imperative to support GPU backend in SystemML. To add a GPU backend, the SystemML optimizer was modified to compile a GPU low-level operator if the input data, intermediate data and output data for a given operation fits in the GPU device memory. The GPU backend invokes highly tuned kernels from CUDA libraries like CuBLAS, CuSPARSE, or CuDNN when available. In other instances, it invokes custom CUDA kernels packaged with SystemML. Data is lazily copied back and forth between the GPU device memory and the host memory as needed. Also, data is converted from row-major to column-major and vice-versa when needed by CUDA library operations. Data is evicted from the GPU memory using an LRU strategy. It is copied back to the host memory if it was dirty when evicted. Data on the host is spilled onto disk when appropriate.

Sparse Operations. SystemML maintains the number of non-zeros for each intermediate matrix, decides upon dense or sparse formats, and selects appropriate runtime operators for combinations of dense and sparse inputs. For sparse-safe operations (such as convolution and matrix multiplication), this reduces the number of floating point operations and improves memory efficiency. For example, there are four physical convolution operators (using lowering technique [5]), dense input / dense filter, sparse input / dense filter, dense input / sparse filter and sparse input / sparse filter.

Distributed Operations. The `Keras2DML` and `Caffe2DML` APIs allow the user to configure the execution strategy of the underlying optimization algorithm with the parameters `train_algo` and `test_algo`. For example, if `train_algo` is set to `"minibatch"`, the generated DML script contains a `for` loop that loops over the dataset one batch at a time. If `batch_size` is small enough such that the input, output and intermediate matrices fit in the driver JVM, then SystemML will generate a single-node plan. If on the other hand, the user sets `batch_size` to a very large value (for example, `train_algo` set to `"batch"`) or if the weights no longer fit on the driver JVM, then SystemML will generate a distributed data-parallel plan where the large input activations or weights are partitioned into fixed size blocks and represented internally as `RDD` [4]. For scoring using a compute-intensive deep network such as ResNet-50 on a large dataset, it is often better to use the task-parallel loop construct - `parfor` - with a small `batch_size` instead of the `for` loop [3]. This script is generated automatically by setting `test_algo` to `"allreduce"`. The `parfor` optimizer then automatically creates optimal parallel execution plans that exploits multi-core, multi-gpu, and cluster parallelism based on the underlying cluster and data characteristics. As an example, the `parfor` optimizer compiles a row-partitioned `remote-parfor` plan for the ResNet-50 prediction script that avoids shuffling and scales linearly with the number of cluster nodes over large data.

4 FUTURE WORK

We plan to extend the existing code generation framework in SystemML for deep learning operations. This includes supporting vertical fusion across layers, horizontal fusion for shared inputs (e.g. reuse temporary `im2col` intermediates in presence of multiple convolution operators consuming the same input), and code generation for heterogeneous hardware including GPUs. Asynchronous algorithms such as HogWild! [16], and Stale-Synchronous SGD [11] will be supported in SystemML through parameter server abstractions [1]. This will help in making SystemML a unified framework for small- and large-scale machine learning that supports data-parallel, task-parallel, and parameter-server-based execution strategies in a single framework. We also plan to investigate the automatic exploitation of the optimization tradeoff between hardware efficiency and statistical efficiency [8].

REFERENCES

[1] SYSTEMML-2083 - Language and Runtime for Parameter Servers. https://issues.apache.org/jira/browse/SYSTEMML-2083, 2018.

[2] Martín Abadi et al. TensorFlow: A System for Large-scale Machine Learning. In *OSDI*, 2016.

[3] Matthias Boehm et al. Hybrid Parallelization Strategies for Large-scale Machine Learning in SystemML. *PVLDB*, 7(7), 2014.

[4] Matthias Boehm et al. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13), 2016.

[5] Sharan Chetlur et al. cuDNN: Efficient Primitives for Deep Learning. *CoRR*, 2014.

[6] Minsik Cho et al. PowerAI DDL. *CoRR*, 2017.

[7] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[8] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. *SIGARCH*, 45, 2017.

[9] Jeff Dean et al. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[10] Kaiming He et al. Deep Residual Learning for Image Recognition. *CoRR*, 2015.

[11] Qirong Ho et al. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS*, 2013.

[12] Sergey Ioffe et al. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, 2015.

[13] Yangqing Jia. *Learning Semantic Image Representations at a Large Scale*. PhD thesis, EECS Department, University of California, Berkeley, May 2014.

[14] Yangqing Jia et al. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR*, 2014.

[15] Yann LeCun et al. Deep Learning. *Nature*, 521, 2015.

[16] Benjamin Recht et al. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*. 2011.

[17] Nitish Srivastava et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *JMLR*, 2014.

[18] Matei Zaharia et al. Spark: Cluster Computing with Working Sets. HotCloud, 2010.