MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems

Tianqi Chen, Mu Li,* Yutian Li, Min Lin, Naiyan Wang, Minjie Wang,
Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang
DMLC Team
http://dmlc.io

Abstract

MXNet is a multi-language machine learning (ML) library to easy the development of ML algorithms, especially for deep neural networks. It offers deeply embedded symbolic expression with auto differentiation, together with shallowly embedded tensor computation, which bridges the gap between the former and the host language. MXNet runs on various heterogeneous systems ranging from mobile devices to distributed GPU clusters.

In this paper we will describe both the API design and the system implementation of MXNet. We will demonstrate both computation and memory efficiency on large scale deep neural network applications on multiple GPU machines.

1 Introduction

Machine learning (ML) algorithms are becoming more complex today. All recent Imagenet challenge [13] winners used neural networks with tens to hundreds layers which require billions of floating-point operations per single example. This boost in algorithm structure and computation complexity brings challenges to ML system design and implementation.

Most ML systems embed a domain-specific language (DSL) into a host language, which is often categorized into either a *deep* or a *shallow embedding* [6]. The deep embedding constructs terms into an computation graph and then traverses it for evaluation; Examples include Theano [1], Caffe [8] and the very recent TensorFlow [12]. While with the shallow embedding, terms are executed by their semantics, bypassing the intermediate computation graph, which is adopted by Torch7 [3] and most numerical packages such as numpy. The deep embedding facilitates optimizing the evaluation of computation graphs, while the shallow embedding can easily enjoys the host language's functionalities. More More comparisons between these two paradigms are shown in Table 1.

In this paper, we present *MXNet*, often called "mix-net", which combines both shallow and deep embedding to maximize both flexibility and efficiency. It offers symbolic expressions with automatic differentiation for efficient object function and neural network evaluation. It also provides tensor computation which works seamless with both symbolic expressions and host languages including C++, Python, R, Go and Julia.

We implemented these two paradigms in a unified way. We implicitly construct computation graphs for the shallow embedding and issue them together with the graphs from the deep embedding to the same backend engine, which is able to track read and write dependencies across graphs and execute them continuously. We proposed efficient memory allocation strategies to reduce the memory footprint for better fitting with the memory constraint devices such as GPU and mobiles. Furthermore, we designed easy to use data communication module so that a MXNet program can be run on multiple machines with very little change.

*Corresponding author (muli@cs.cmu.edu)

	Shallow embedding	Deep embedding
Execute $a = b+1$	Do the computation and store the results on a as the same type with b.	Return a computation graph. We can bind data to b and do the computation later.
Advantages	It is conceptually straightforward, and often works seamless with the host language's build-in data structures, functions, debugger, and third-party libraries.	We get the whole computation graph before evaluation, which facilitates efficient memory allocation and execution optimization. It is also convenient to implement functions such as load, save, and visualization.

Table 1: Compare the shallow and deep embedding for domain specific languages.

System	Core Lang	Binding Langs	Devices (beyond CPU)	Distributed	Shallow Embedding	Deep Embedding
Caffe [8]	C++	Python/Matlab	GPU	×	√	×
Torch7 [3]	Lua	-	GPU/FPGA	×	√	×
Theano [1]	Python	-	GPU	×	×	√
TensorFlow [12]	C++	Python	GPU/Mobile	√[1]	×	√
MXNet	C++	Python/R/Julia/Go	GPU/Mobile	√	√	√

Table 2: Compare to other popular open-source ML libraries

```
>>> import mxnet as mx
>>> a = mx.nd.ones((2, 3),
... mx.gpu())
>>> print (a * 2).asnumpy()
[[ 2.  2.  2.]
 [ 2.  2.  2.]]
```

Figure 2: NDArray interface in Python

```
using MXNet
mlp = @mx.chain mx.Variable(:data) =>
  mx.FullyConnected(num_hidden=64) =>
  mx.Activation(act_type=:relu)    =>
  mx.FullyConnected(num_hidden=10) =>
  mx.Softmax()
```

Figure 3: Symbol expression construction in Julia.

MXNet is the successor of the previous systems we built, including Minerva [16], which automatically schedules tensor computations into devices, CXXNet [5], which constructs symbolic expressions from configurations, and Purine2 [11], which presents the computation graph as a bi-graph. Comparing to other open-source ML systems, MXNet provides a superset programming interface to Torch7 [3], Theano [1], Caffe [8], and the very recent released TensorFlow [12]. In addition, MXNet is lightweight, e.g. the prediction codes fit into a single 50K lines C++ source file with no other dependency, and has more languages supports. Table 2 shows more detailed comparisons.

2 Programming Interface

The overview of MXNet is shown in Figure 1. We will discuss the frontend interface in this section, while leave the backend to Section 3.

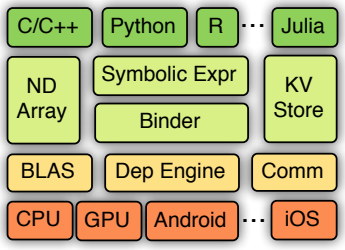

Figure 1: MXNet Overview

2.1 `Symbol`: Symbolic Expressions

MXNet uses multi-output symbolic expressions, called `Symbol`, for the deep embedding. Symbols are composited by operators, whose arguments can be other symbols' outputs. An operator may have internal states and produce more than one outputs. Each augment, internal state, and output is viewed as a variable, which is an atomic symbol returning itself. MXNet implemented Various operators, ranging from the simple "+" operation to complex neural network layers, to easy the developing of ML algorithms. Figure 3 shows the construction of a multi-layer perception symbol by chaining a variable, which presents the input data, and several layer operators.

[1] Not released so far.

To evaluate a symbol we need to declare the required outputs and bind the free variables with data, namely the variables have not be assigned with inputs. Beside evaluation, the `Symbol` supports other functions such as auto symbolic differentiation, load, save, memory estimation, and visualization.

2.2 `NDArray`: Tensor Computation

MXNet offers `NDArray` with tensor computation as the shallow embedding to fill the gap between the symbolic expression and the host language. Figure 2 shows an example which does matrix-constant multiplication on GPU and then prints the results by `numpy.ndarray`.

The `NDArray` works seamless with `Symbol`, the former is often used in both binding data and getting the outputs for the latter. We can further mix the tensor computation of `NDArray` with the `Symbol`. For example, assume there is symbolic neural network, denoted by `net`, while the weight updating function, e.g. $w = w - \eta g$, is written by `NDArray` and the learning rate η is calculated by native host language statements. Then we can implement the gradient descent method by

```
while(1) { net.foward_backward(); update(net.w, net.g) };
```

The above is as efficient as the more complex, due to the control flow, single symbolic expression implementation. The reason is that MXNet uses lazy evaluation of `NDArray` and the backend engine can resolve the data dependencies crossing `NDArray` and `Symbol` correctly. Furthermore, the above mixed implementation is often easier for debugging.

2.3 `KVStore`: Data Synchronization Over Devices

The `KVStore` is a distributed key-value store for data synchronization over multiple devices for both deep and shallow embedding. It supports to *push* a key-value pair from a device to the store, and *pull* the value for a key from the store. In addition, a user-defined updater which specifies how to write the received value into store is allowed. It furthermore offers flexible data consistent models [9], such as the commonly used sequential and eventual consistencies.

The following example implements the distributed gradient descent by using `KVStore`

```
while(1){ kv.pull(net.w); net.foward_backward(); kv.push(net.g); }
```

where the weight updating function is registered to the `KVStore`, and each worker repeatedly pulls the newest weight from the store and then pushes out the locally computed gradient. Similar to `NDArray`, the above implementation has a similar performance comparing to a single symbolic expression, because both push and pull are scheduled with others by the same backend engine.

2.4 I/O and Training Modules

MXNet ships tools to pack arbitrary size examples into a single compacted file to facilitate both sequential read and random seek. Data iterators are also provided, which do multi-thread data pre-fetch and pre-processing to reduce the overheads possibly due to remote reading from distributed filesystems or image decoding and transformation.

Commonly used optimization algorithms, such as the stochastic gradient descent, are implemented in the training module, which trains a model on a given symbolic model and data iterators. It runs in distributed environments if an additional `KVStore` is provided.

3 Implementation

3.1 Computation Graph

A binded symbolic expression is presented as a computation graph for evaluation. Figure 4 shows a part of the graph of the forward and backward of the MLP symbol in Figure 3. Before evaluation, MXNet transforms the graph to optimize the efficiency and allocates memory to internal variables.

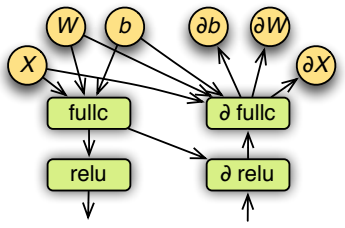

Figure 4: Computation graph for both forward and backward.

Graph Transformation Firstly, remember that the output variables are specified during binding, which could be a subset of the whole outputs. For example, gradients are not required for prediction, or the last layers can be skipped when extracting features from internal layers. Then we only need to traverse the subgraph needed for the output subset. Secondly, operators can be grouped into a single one. For example, we can present $a \times b + 1$ by a single BLAS or GPU kernel operation rather than two to reduces the number of function calls. Finally, we manually implemented well-optimized "big" operations, such as a layer in neural network, which also reduce the computation graph size.

Memory Allocation Device memory is often limited, especially for GPUs and mobile devices. Naively allocating new memory to every variable is not desirable. Given a computation graph, the life time of each variable, namely the period between the creation and the last time will be used, is known. So we can reuse memory for non-intersected variables. However, an ideal allocation strategy requires $O(n^2)$ time complexity, where n is the number of variables.

We proposed two heuristics strategies with linear time complexity. The first is called *inplace*. It simulates the procedure of traversing the graph, and keeps the number of depended variables that are not used so far. Once a variable's number goes 0, then we recycle its memory. The second one, named *sharing*, allowing two variables to share memory if they cannot be run in parallel, because the sharing brings an additional dependency which destroys parallelization. In particular, each time this strategy finds the longest path in the graph that has not assigned before, and performs the allocation.

3.2 Dependency Engine

In MXNet we designed a general purpose engine which is able to schedule arbitrary workloads from any modules. The engine places a unique tag, e.g. the start memory address, for each source unit such as a NDArray object. A workload is pushed into the engine by giving the function clossure for running the workload, which would be asynchronous, and tags this workload will be read together with the tags will be written.

Different to most dataflow engines [16, 12], ours allows to specify the write dependencies, which facilitates memory sharing. It is also convenient in some other cases. For example, assume there are two random number generation operations, which mutate the same random seed. Then we can inform the engine they will write the random seed to explicitly indicate the dependency.

The engine continuously schedules the pushed workloads for execution if their dependencies are satisfied. Since there usually exists multiple computation resources such as CPU, GPUs, and the memory/PCIe buses, the engine uses multiple threads for execution to improve the resource utilization.

3.3 Data Communication

We implemented `KVStore` based on the parameter server [9, 10, 4], which is shown in Figure 5. It differs to previous works in two aspects: first, we use MXNet's engine to schedule the operations and manage the data consistency, which not only makes the data synchronization works seamless with the others such as `Symbol`, and also greatly simplifies the parameter server implementation. Second, we adopt a two-level server structure. A level 1 server manages the data synchronization between the devices within in a single machine, while a level 2 server manages inter-machine synchronization. We may aggregate data on the level 1 servers to reduce machine traffic, and use different consistency models between in-machine and inter-machine synchronization.

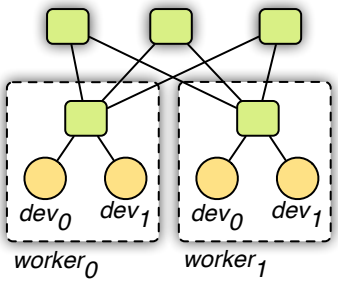

Figure 5: Data communication.

4 Evaluation

Compare to Others on Performance. We fist compare MXNet with Torch7, Caffe, and TensorFlow on the popular "convnet-benchmarks" [2]. All these systems are compiled with CUDA 7.5 and CUDNN 3 except for TensorFlow, which only supports CUDA 7.0 and CUDNN 2. We use

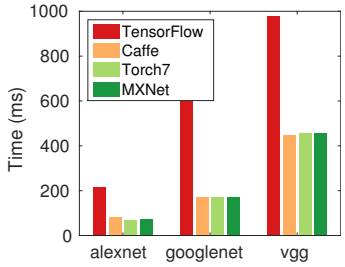

Figure 6: Compare MXNet to others on a single forward-backward performance.

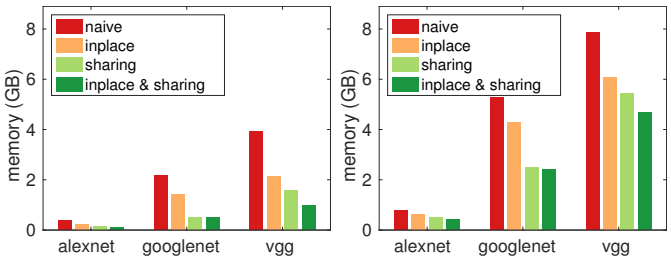

Figure 7: Internal memory usage of MXNet under various allocation strategies for only forward (left) and forward-backward (right) with batch size 64.

batch size 32 for all networks and run the experiments on a single Nvidia GTX 980 card. Results are shown in Figure 6. As expected that MXNet has similar performance comparing to Torch7 and Caffe, because most computations are spent on the CUDA/CUDNN kernels. TensorFlow is always 2x slower, which might because it is still premature and uses a lower version CUDNN.

Memory Usage. We then investigate the memory allocation strategies. The memory usages of the internal variables excepts for the outputs are shown in Figure 7. As can be seen, both "inplace" and "sharing" can effective reduce the memory footprint. Combing them leads to a 2x reduction for all networks during model training. While this reduction increases to 4x for model prediction. Therefore, even for the most expensive VGG net, we only needs less than *16MB* extra memory beside the model to predict an image.

Scalability Finally we demonstrate the scalability of MXNet on multiple machines. We run the experiment on Amazon EC2 g2.8x instances, each of which is shipped with four Nvidia GK104 GPUs and 10G Ethernet. We train googlenet [15] with batch normalization [7] on the ILSVRC12 dataset [14] which consists of 1.3 million images and 1,000 classes. We fix the learning rate to .05, momentum to .9, weight decay to 0.05, and feed each GPU with 36 images in one batch.

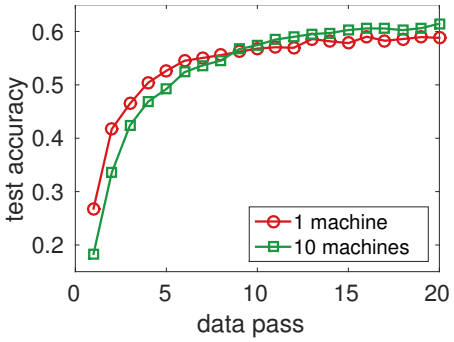

Figure 8: Progress of googlenet on ILSVRC12 dataset on 1 and 10 machines.

The convergence results are shown in Figure 8. As can be seen, the distributed training converges slower comparing to the single machine at the beginning. However, the former outperforms the latter after 10 data passes. The average cost of a data pass is 14,203 sec on a single machine, while it reduces to 1,422 sec for 10 machines. Therefore, to achieve the same test accuracy, there is even a super linear speedup, namely 10 machines is more than 10 times faster than a single machine.

5 Conclusion

We described MXNet, which is a machine learning library combining deeply embedded symbolic expression with shallowly embedded tensor computation to maximize the efficiency and flexibility. It is lightweight and runs on various heterogeneous systems ranging from mobiles to distributed GPU clusters. We shown experiments on large scale deep learning applications to demonstrate its efficiency. We believe that MXNet will benefit further researches on both machine learning algorithm and system. The codes are available at `http://dmlc.io`.

Acknowledgment: We sincerely thanks Dave Andersen, Carlos Guestrin, Tong He, Chuntao Hong, Qiang Kou, Alex Smola, Dale Schuurmans and all other DMLC contributors.

References

[1] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

[2] Soumith Chintala. Easy benchmarking of all public open-source implementations of convnets, 2015. `https://github.com/soumith/convnet-benchmarks`.

[3] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[4] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *Neural Information Processing Systems*, 2012.

[5] CXXNet Developers. Cxxnet: fast, concise, distributed deep learning framework, 2015. `https://github.com/dmlc/cxxnet`.

[6] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: Deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 339–347. ACM, 2014.

[7] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[9] M. Li, D. G. Andersen, J. Park, A. J. Smola, A. Amhed, V. Josifovski, J. Long, E. Shekita, and B. Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[10] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. Communication efficient distributed machine learning with the parameter server. In *Neural Information Processing Systems*, 2014.

[11] Min Lin, Shuo Li, Xuan Luo, and Shuicheng Yan. Purine: A bi-graph based deep learning framework. *arXiv preprint arXiv:1412.6249*, 2014.

[12] Abadi Martn, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous systems. 2015.

[13] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[14] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, pages 1–42, 2014.

[15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[16] Minjie Wang, Tianjun Xiao, Jianpeng Li, Jiaxing Zhang, Chuntao Hong, and Zheng Zhang. Minerva: A scalable and highly efficient training platform for deep learning, 2014.