Leaves Machine Learning and Optimization Library

October 17, 2017

This document describes how to use **LE**aves **M**achine Learning and **Optimization** Library (LEMO) for modeling. LEMO is an open source library for scientific computing. LEMO puts Python first, and also puts algorithm design first, with many open source computational architectures as its backend. The goal is to provide a flexible and efficient tool for applying machine learning models and developing mathematical optimization algorithms. Currently, LEMO is on its preliminary version.

Table 1 summarizes the three-layer structure of LEMO library. Firstly, machine learning models layer develops all the learning algorithms. Currently, the models are abstracted into two groups: generalized linear models and low rank models, which present many important families of learning models. Secondly, optimization algorithm layer implements a growing collection of state-of-the-art optimization algorithms, many of which are tuned for the specific machine learning models. Finally the numerical computing layer on the bottom defines algebra operations, providing a unified interface to access many scientific computing libraries, including MXNet, PyTorch. It offers a programming framework that is friendly to Numpy users and is capable of using parallel framework like CUDA as well.

Layers	Modules
Machine learning models	Generalized linear and low rank models.
Optimization algorithms	First order methods, convex and nonconvex methods etc.
Numerical interface	Basic linear algebra, vector, matrix and tensor operations

Table 1: LEMO's architecture

Open-source scientific computing libraries for machine learning and optimization have become popular in the recent years. Scikit-learn[1] is an open source machine learning toolbox in Python. It mainly focuses on machine learning and computer vision, however, the strength in deep learning and high performance computing is limited. H2O[2] is an open source library in Java, which also supports parallel and distributed computing. TensorFlow[3] is Google's open source numeric library, by far the most popular framework (based on surveys in Github). Although TensorFlow mainly focuses on deep learning, it has been turned into a general machine learning toolbox by third party contributions. Other DL architectures include MXNet[4], PyTorch[5], Chainer[6], etc. It should be mentioned that most of the DL frameworks are based on high performance architectures, mainly driven by the intensive demand of big data in deep learning tasks.

Compared to existing work, LEMO is a more flexible and convenient framework for scientists and engineers. Both the machine learning and optimization layers provide

[1] http://scikit-learn.org/stable/index.html
[2] https://www.h2o.ai
[3] https://www.tensorflow.org
[4] http://mxnet.io
[5] http://pytorch.org
[6] http://chainer.org

friendly interfaces, such that users can not only apply machine learning models, but apply and design numerical algorithms for machine learning, and more extensive engineering tasks as well. In addition, LEMO's framework is built to support open-source numerical libraries, to take full advantage of parallel and distributed computing.

1 Basics

LEMO can be running on top of PyTorch or MXNet. To install LEMO, in the source folder, run the following command

```
python setup.py install
```

To set up the backend, simply add the configuration file in the path:

```
$HOME/.lemo/config.json
```

An example file has the following form:

```
{
 "support": "mxnet",
 "floatx": "float32"
}
```

Up until now (Oct 13, 2017) LEMO supports PyTorch 0.2+ and MXNet 0.11+

2 Linear algebra

In LEMO, vectors, matrices and tensors are expressed as the `Variable` objects, and their behaviors are similar with `numpy.ndarray`. Internally, `Variable` wraps up the tensor object which is defined by the backend library. Hence the tensor data can be MXNet's NDArray or PyTorch's Tensor, depending on the backend configuration. The design purpose is to take full advantage of numpy style programming with the power of CUDA.

Example code

```
import lemo.support as S

n, p = 4, 2
A = S.variable(S.randn(shape=(n, p)))
print(A)

x = S.variable(S.randn(shape=p))
print(x)

z = S.dot(A, x)

loss = S.norm(z)**2
print('||Ax-b||^2 %e' % S.scalar(loss))
```

The output of the above code is as follows:

```
class support.Variable, wrapper of MXNet ndarray

```

```
[[ 1.82998681 -0.15010811]
 [-2.53293657 -0.69604313]
 [ 1.09507072  1.28476834]
 [ 0.92212433  0.6416387 ]]
<NDArray 4x2 @cpu(0)>
class support.Variable, wrapper of MXNet ndarray

[ 0.91652578  1.57152474]
<NDArray 2 @cpu(0)>
||Ax-b||^2 2.631427e+01
```

It is known that deep learning is computationally intensive, most of the resources are allocated for forward/backward propagation. In optimization language, this is about computing the objective and the gradient. For convenience, deep learning libraries provide automatic gradient computation for modeling, so that machine learner will not be distracted by too much algebra details. In order for achieving that, the computation is done by **symbolic programming**, as opposed to **imperative programming**. In symbolic programming, variables are linked into a graph, based on the related math operations in the forward pass. In the backward pass, information will be collected for gradient computation.

LEMO provides both symbolic mode and imperative mode. Symbolic mode can be helpful if you need gradient based algorithms, imperative programming is more flexible for other types of algorithms. Below is a simple example for demonstration. Firstly, we generate some random numbers and define the least squares problems:

```
A = S.variable(S.randn((100, 100)))
x = S.variable(S.randn(100))
b = S.variable(S.randn(100))

# imperative mode: compute objective 0.5||Ax-b||^2
print("least square %.3e" % S.scalar(0.5 * S.sum_squares(S.dot
    (A, x) - b)))


def least_squares(x):
    return 0.5 * S.sum_squares(S.dot(A, x) - b)
```

In the following code, we provide two ways to compute the gradient. In LEMO, `compute_gradient` method performs in symbolic mode and builds the computational graph. Then gradient is automatically computed. In addition, we can explicitly express and then compute the gradient.

```
# symbolic mode: compute gradient
g1 = S.compute_gradient(least_squares, x)

# imperative mode: compute gradient
g2 = S.dot(S.transpose(A), S.dot(A, x) - b)

print("diff %.3e" % S.scalar(S.norm(g1 - g2)))
```

The two approaches obtain the same result:

```
[1]: run examples/demo_sym_imp.py
least square 3.843e+03
diff 0.000e+00
```

3 Machine learning models

Machine learning models are under the path `lemo/models/`. Two general machine learning models are provided.

3.1 Generalized linear models

Generalized linear models (GLM) play a central role in machine learning and statistics. For example, we can use Gaussian family for regression, and binomial models for classification. To train a GLM, we aim at finding the maximum log likelihood:

$$\max_w \frac{1}{n} \log P(y_i; x_i, w) - \lambda R(w)$$

where $P(y_i; x_i, w)$ is the likelihood of an exponential family distribution, and $R(w)$ is the regularizer. It should be noted that, equivalently, regularized loss minimization, which is often adopted by machine learning community, can be expressed as follows:

$$\min_w \frac{1}{n} \sum_i L(x_i, y_i, w) + \lambda R(w)$$

where the loss function $L()$ is the negative log-likelihood. We may use both languages interchangeably, when no ambiguity arises. In LEMO, the following exponential families are considered:

1. Gaussian family (square loss $L(y, x, w) = (y - w^T x)^2$);
2. Binomial family (logistic loss: $L(y, x, w) = \log(1 + \exp(-y w^T x))$;
3. Laplacian family (quantile or l_1 loss: $L(y, x, w) = \left|y - w^T x\right|$).

Regularization plays an important role for controlling model complexity. In LEMO, we use:

$$R(w) = (1 - \frac{\alpha}{2})||w||_2^2 + \alpha||w||_1$$

1. $\alpha = 0$: Ridge regression penalty (Tikhonov regularization) penalizes all the variables and reduces values proportionally.
2. $\alpha = 1$: Least absolute shrinkage and selection operator (Lasso). It shrinks all the parameters with same amount and performs truncation when reduced value is below the threshold.
3. $0 < \alpha < 1$: Elastic-net combines ridge regression and Lasso. It overcomes the problems in Lasso such that only one variable is picked out from a group.

3.2 Generalized low rank models

Generalized low rank models (GLRM [9]) is a class of unsupervised models for feature representation learning and reconstruction. Let $A \in \mathbb{R}^{m \times n}$, GLRM solves the problem of the following form:

$$\min_{X,Y} L(A, XY) + r(X) + r(Y)$$

The loss functions includes

1. Square loss: $L(A, XY) = \|A - XY\|_F^2$
2. Absolute loss: $L(A, XY) = \|A - XY\|_1$
3. Logistic loss: $L(A, XY) = \sum_{ij} \log(1 + \exp(-Z_{ij}))$ where $Z = A \circ (XY)$. [7]

In addition, we use elastic-net regularizers for X and Y separately:

$$r(X) = \lambda_X \left(\frac{1 - \alpha_X}{2} \sum \|X_i\|_2^2 + \alpha_X \sum_i \|X_i\|_1 \right)$$

$$r(Y) = \lambda_Y \left(\frac{1 - \alpha_Y}{2} \sum \|Y_i\|_2^2 + \alpha_Y \sum_i \|Y_i\|_1 \right)$$

3.3 Support vector machines

Support vector machine (SVM) has been considered as one of the most successful classification models. However, it does not belong to the generalized linear model.

Consider the binary classification problem with N training examples $(u^{(i)}, v^{(i)})$, $i = 1, \ldots, N$. For convenience, we assume that the output $v^{(i)}$ is given by either 1 or -1, i.e., $v^{(i)} \in \{-1, 1\}$, rather than $v^{(i)} \in \{0, 1\}$ as in the previous sections. Observe that this is just a change of label for the class, but does not affect at all the fact that one particular example belongs one class or the other. The SVM problem can be formulated as

$$\begin{array}{ll} \max_{w,b} & \frac{1}{\|w\|} \\ \text{s.t.} & v^{(i)}[w^T u^{(i)} + b] \geq 1, i = 1, \ldots, N, \end{array}$$

or equivalently,

$$\begin{array}{ll} \min_{w,b} & \frac{1}{2}\|w\|^2 \\ \text{s.t.} & v^{(i)}[w^T u^{(i)} + b] \geq 1, i = 1, \ldots, N. \end{array}$$

To allow for possible violation of the constraints, we reformulate the above problem as

$$\begin{array}{ll} \min_{w,b} & \frac{1}{2}\|w\|^2 + \lambda \sum_{i=1}^N \xi_i \\ \text{s.t.} & v^{(i)}[w^T u^{(i)} + b] \geq 1 - \xi_i, i = 1, \ldots, N, \\ & \xi_i \geq 0, i = 1, \ldots, N, \end{array}$$

for some $\lambda > 0$. Observe that this problem can be written equivalently as

$$\min_{w,b} \quad \frac{1}{2}\|w\|^2 + \lambda \sum_{i=1}^N \max\{0, 1 - v^{(i)}[w^T u^{(i)} + b]\}.$$

f These formulations are called *soft-margin support vector machines*.

[7] $A \circ B = C$, then $C_{ij} = A_{ij} \cdot B_{ij}$

4 Optimization algorithms

LEMO implements many advanced algorithms for numerical optimizations. Numerical optimization ([10]) aims at finding the best element that achieves optimal values of f:

$$\min_{x \in \mathcal{X}} f(x)$$

Optimization has deep impacts on machine learning and statistics, of which many models can be described as optimization problems of certain f. LEMO provides some important optimization algorithms to support machine learning modeling. Nevertheless, these algorithms can be applied to more general engineering or operations research problems as well. Some of the algorithms are presented as follows.

4.1 Smooth optimization

A function f is smooth if it is continuously differentiable, namely, we can define the gradient $\nabla f(x)$ everywhere in the feasible domain. Gradient descent (GD) algorithms (and their stochastic version) have made great success in optimization of big data problems, in particular, deep learning models. Recently, accelerated gradient descent, originally invented by Yurii Nesterov ([7]) has attracted great attention. It has been successfully applied to machine learning, stochastic optimization, and deep learning as well. For example, see [5, 8]. LEMO implements several variants of gradient descent methods like accelerated gradient methods, and conjugate gradient methods, with adaptive stepsize and line search. In principle, these algorithms can be extended for optimizing neural networks, and we leave it for future work.

4.2 Nonsmooth optimization

Optimization of nonsmooth objective is pervasive in machine learning field. For example, take a look at support vector machines or Lasso. If the optimization objective has the additional property of convexity, subgradient methods can be applied with guaranteed convergence to the optimality, see [3]. Moreover, many nonsmooth models admit conjugate forms ([1]), hence can be approximated by smooth functions and further accelerated by fast gradient methods ([6], [7]). The key is to turn the optimization into a series of smoothed problems, each of which can be solved by some faster optimization algorithms. We show the structure of such algorithms as follows:

1. Set a smoothing parameter μ for approximation.
2. Solve the smoothed problem $\min_x F_\mu(x)$ with some algorithms.
3. Obtain solution x_μ and update μ.
4. Go to 1.

4.3 Nonconvex optimization

Many machine learning models are formulated as structured nonconvex problems as follows:

$$\min_{x,y} F(x,y) = f(x,y) + g(x) + h(y) \tag{1}$$

where the objective is convex on x and y individually, but nonconvex on x and y jointly. For such problems, we apply alternating minimization to turn (1) into a series of problems of one block of variables. We summarize the procedure as follows:

1. Initialize $t = 0$, x_0, y_0;
2. Fix y_t and update $x_{t+1} \approx \text{argmin}_x F(x, y_t)$;
3. Fix x_{t+1} and update $y_{t+1} \approx \text{argmin}_y F(x_{t+1}, y)$;
4. $t \leftarrow t + 1$;
5. Goto 2.

Since the exact minimization in each subproblem can be difficult, we adopt proximal gradient methods that have been applied for deterministic, stochastic and composite programming. For example see [2]. In each iteration, we perform one step of proximal gradient descent on each block of variables alternatively. In addition, we apply more advanced techniques, such as accelerated methods ([4]) or conjugate gradient methods to improve the efficiency on the subproblems.

5 Examples

In below, from a simple example, we show how to call machine learning models in LEMO. Consider the binomial family in GLM (logistic loss classification):

$$\min_w \frac{1}{n} \log(1 + \exp(-yw^T x)) + \lambda \left(\frac{1-\alpha}{2} \|w\|_2^2 + \alpha \|w\|_1 \right)$$

Logistic loss classification

```
1
2 family = 'binomial'
3 lambda_ = 1e-4
4 alpha = 0
5 md = GLM(family=family, max_iters=200, lambda_=lambda_, alpha=
      alpha,
6           verbose=False, tol=1e-4)
7 md.fit(X, y)
```

6 CUDA support

LEMO also supports CUDA for parallel programming. To enable CUDA, simply define a context `S.cuda_device(ind)` where `ind` is the device number, before running the algorithm. As an example, we consider running the accelerated methods for the problem:

$$\min_x \frac{1}{2}\|Ax - b\|^2$$

where $A \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$ are random matrices and vectors where each entry is from $\mathcal{N}(0, 1)$. The following snapshot provides a simple example for comparing LEMO's performance on CPU and GPU.

Least squares optimization

```
1   with S.cuda_device(ind=0):
2       A = S.variable(A_np)
3       b = S.variable(b_np)
4       x = S.variable(x_start)
5       tstart = time.time()
6
7       def oracle(xx):
8           return 0.5 * S.sum_squares(S.dot(A, xx) - b)
9
10      agd = AcceleratedGradientDescent(oracle, x, lip=lip,
            verbose=True)
11      agd.minimize()
12      gputime.append(time.time() - tstart)
```

In Figure 1, we plot the algorithm running time with implementation on CPU and GPU. The tested CPU is `Intel Xeon E5-2630v3` and the tested GPU is `Quadro K2200`. Both algorithms run 500 iterations, using PyTorch as its backend.

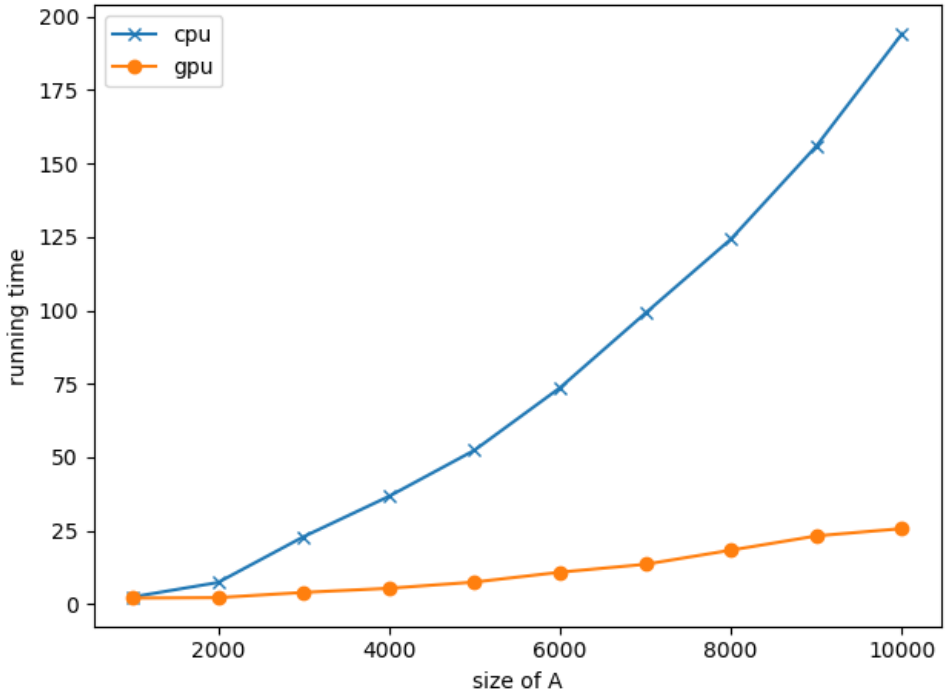

Figure 1: Running time of Nesterov's algorithm on least squares problem. $x-$axis: size of A, from $n = 1000$ to 10000. $y-$axis: running time of the algorithms (in seconds).

7 Conclusion and Future Work

In this report, we introduce LEMO, a new scientific computing library for machine learning and numerical optimization. In the future, we plan to enrich LEMO and include models and algorithms like deep learning, ADMM, second-order methods and stochastic gradient methods. We will focus on the development and implementation of efficient algorithms for distributed and decentralized machine learning and optimization.

References

[1] S. Becker, J. Bobin, and E. J. Candès. Nesta: A fast and accurate first-order method for sparse recovery. *SIAM Journal on Imaging Sciences*, 4(1):1–39, 2011.

[2] J. Bolte, S. Sabach, and M. Teboulle. Proximal alternating linearized minimization for nonconvex and nonsmooth problems. *Mathematical Programming*, 2014.

[3] S. Boyd and A. Mutapcic. Subgradient methods. *Lecture notes of EE364b, Stanford University, Winter Quarter*, 2007, 2006.

[4] S. Ghadimi, G. Lan, and H. Zhang. Generalized uniformly optimal methods for nonlinear programming. *arXiv preprint arXiv:1508.07384*, 2015.

[5] G. Lan. An optimal method for stochastic composite optimization. *Mathematical Programming*, 133(1):365–397, 2012.

[6] Y. Nesterov. Smooth minimization of non-smooth functions. *Mathematical programming*, 103(1):127–152, 2005.

[7] Y. Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

[8] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[9] M. Udell, C. Horn, R. Zadeh, S. Boyd, et al. Generalized low rank models. *Foundations and Trends® in Machine Learning*, 9(1):1–118, 2016.

[10] S. J. Wright and J. Nocedal. Numerical optimization. *Springer Science*, 35(67-68):7, 1999.