LightNet: A Versatile, Standalone Matlab-based Environment for Deep Learning

[Simplify Deep Learning in Hundreds of Lines of Code]

Chengxi Ye, Chen Zhao*, Yezhou Yang, Cornelia Fermüller and Yiannis Aloimonos
Computer Science Department, University of Maryland
College Park, MD, USA.
cxy@umiacs.umd.edu, chenzhao@umd.edu, yzyang@umiacs.umd.edu,
cornelia@umiacs.umd.edu, yiannis@umiacs.umd.edu

ABSTRACT

LightNet is a **lightweight**, **versatile**, **purely Matlab-based** deep learning framework. The idea underlying its design is to provide an easy-to-understand, easy-to-use and efficient computational platform for deep learning research. The implemented framework supports major deep learning architectures such as Multilayer Perceptron Networks (MLP), Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). The framework also supports both CPU and GPU computation, and the switch between them is straightforward. Different applications in computer vision, natural language processing and robotics are demonstrated as experiments.

Availability: the source code and data is available at: https://github.com/yechengxi/LightNet

Keywords

Computer vision; natural language processing; image understanding; machine learning; deep learning; convolutional neural networks; multilayer perceptrons; recurrent neural networks; reinforcement learning

1. INTRODUCTION

Deep neural networks [8] have given rise to major advancements in many problems of machine intelligence. Most current implementations of neural network models primarily emphasize efficiency. These pipelines (Table 1) can consist of a quarter to half a million lines of code and often involve multiple programming languages [5, 13, 2]. It requires extensive efforts to thoroughly understand and modify the models. A straightforward and self-explanatory deep learning framework is highly anticipated to accelerate the understanding and application of deep neural network models.

We present LightNet, a **lightweight**, **versatile**, **purely Matlab-based** implementation of modern deep neural network models. Succinct and efficient Matlab programming techniques have been used to implement all the computational modules. Many popular types of neural networks, such as multilayer perceptrons, convolutional neural networks, and recurrent neural networks are implemented in LightNet, together with several variations of stochastic gradient descent (SDG) based optimization algorithms.

Table 1: Deep Neural Network Packages

Framework	Language	Native Models	Lines of Code
Caffe	C++	CNN	74,903
Theano	Python, C	MLP/CNN/RNN	148,817
Torch	Lua, C	MLP/CNN/RNN	458,650
TensorFlow	C++	MLP/CNN/RNN	335,669
Matconvnet	Matlab, C	CNN	43,087
LightNet	Matlab	MLP/CNN/RNN	951 (1,762)*

* Lines of code in the core modules and in the whole package.

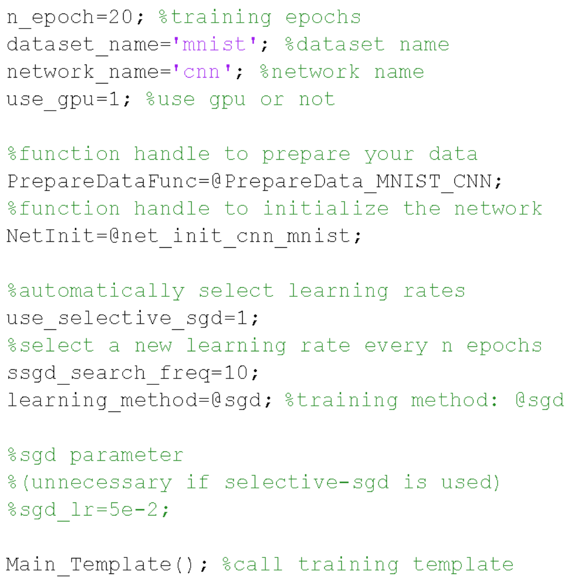
Since LightNet is implemented **solely** with Matlab, the major computations are vectorized and implemented in **hundreds** of lines of code, orders of magnitude more succinct than existing pipelines. All fundamental operations can be easily customized, only basic knowledge of Matlab programming is required. Mathematically oriented researchers can focus on the mathematical modeling part rather than the engineering part. Application oriented users can easily understand and modify any part of the framework to develop new network architectures and adapt them to new applications. Aside from its simplicity, LightNet has the following features: 1. LightNet contains the most modern network architectures. 2. Applications in computer vision, natural language processing and reinforcement learning are demonstrated. 3. LightNet provides a comprehensive collection of optimization algorithms. 4. LightNet supports straightforward switching between CPU and GPU computing. 5. Fast Fourier transforms are used to efficiently compute convolutions, and thus large convolution kernels are supported. 6. LightNet automates hyper-parameter tuning with a novel Selective-SGD algorithm.

2. USING THE PACKAGE

An example of using LightNet can be found in (Fig. 1): a simple template is provided to start the training process. The user is required to fill in some critical training parameters, such as the number of training epochs, or the training method. A Selective-SGD algorithm is provided to facilitate the selection of an optimal learning rate. The learning rate is

MM '16 October 15-19, 2016, Amsterdam, Netherlands

ACM ISBN 978-1-4503-3603-1/16/10.
DOI: http://dx.doi.org/10.1145/2964284.2973791



```
n_epoch=20; %training epochs
dataset_name='mnist'; %dataset name
network_name='cnn'; %network name
use_gpu=1; %use gpu or not

%function handle to prepare your data
PrepareDataFunc=@PrepareData_MNIST_CNN;
%function handle to initialize the network
NetInit=@net_init_cnn_mnist;

%automatically select learning rates
use_selective_sgd=1;
%select a new learning rate every n epochs
ssgd_search_freq=10;
learning_method=@sgd; %training method: @sgd

%sgd parameter
%(unnecessary if selective-sgd is used)
%sgd_lr=5e-2;

Main_Template(); %call training template
```

Figure 1: A basic example, which shows how to train a CNN on the MNIST dataset with LightNet.

selected automatically, and can optionally be adjusted during the training. The framework supports both GPU and CPU computation, through the *opts.use_gpu* option. Two additional functions are provided to prepare the training data and initialize the network structure. Every experiment in this paper can reproduced by running the related script file. More details can be found on the project webpage.

3. BUILDING BLOCKS

The primary computational module includes a feed forward process and a backward/back propagation process. The feed forward process evaluates the model, and the back propagation reports the network gradients. Stochastic gradient descent based algorithms are used to optimize the model parameters.

3.1 Core Computational Modules

LightNet allows us to focus on the mathematical modeling of the network, rather than low-level engineering details. To make this paper self-contained, we explain the main computational modules of LightNet. All networks (and related experiments) in this paper are built with these modules. The notations below are chosen for simplicity. Readers can easily extend the derivations to the mini-batch setting.

3.1.1 Linear Perceptron Layer

A linear perceptron layer can be expressed as: $y = Wx+b$. Here, x denotes the input data of size $input\_dim \times 1$, W denotes the weight matrix of size $output\_dim \times input\_dim$, b is a bias vector of size $output\_dim \times 1$, and y denotes the linear layer output of size $output\_dim \times 1$.

The mapping from the input of the linear perceptron to the final network output can be expressed as: $z = f(y) = f(Wx+b)$, where f is a non-linear function that represents the network's computation in the deeper layers, and z is the network output, which is usually a loss value.

The backward process calculates the derivative $\frac{\partial z}{\partial x}$, which is the derivative passing to the shallower layers, and $\frac{\partial z}{\partial W}$, $\frac{\partial z}{\partial b}$, which are the gradients that guide the gradient descent process.

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} = f'(y)^T \cdot W \tag{1}$$

$$\frac{\partial z}{\partial W} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial W} = f'(y) \cdot x^T \tag{2}$$

$$\frac{\partial z}{\partial b} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial b} = f'(y) \tag{3}$$

The module adopts extensively optimized Matlab matrix operations to calculate the matrix-vector products.

3.1.2 Convolutional Layer

A convolutional layer maps N_{map_in} input feature maps to N_{map_out} output feature maps with a multidimensional filter bank k_{io}. Each input feature map x_i is convolved with the corresponding filter bank k_{io}. The convolution results are summed, and a bias value b_o is added, to generate the o-th output map: $y_o = \sum_{1 \leq i \leq N_{map\_in}} k_{io} * x_i + b_o$. To allow using large convolution kernels, fast Fourier transforms (FFT) are used for computing convolutions (and correlations). According to the convolution theorem [10], convolution in the spatial domain is equivalent to point-wise multiplication in the frequency domain. Therefore, $k_i * x_i$ can be calculated using the Fourier transform as: $k_i * x_i = \mathcal{F}^{-1}\{\mathcal{F}\{k_i\} \cdot \mathcal{F}\{x_i\}\}$. Here, $\mathcal{F}$ denotes the Fourier transform and $\cdot$ denotes the point-wise multiplication operation. The convolution layer supports both padding and striding.

The mapping from the o-th output feature map to the network output can be expressed as: $z = f(y_o)$. Here f is the non-linear mapping from the o-th output feature map y_o to the final network output. As before (in Sec. 3.1.1), $\frac{\partial z}{\partial x_i}$, $\frac{\partial z}{\partial k_i}$, and $\frac{\partial z}{\partial b_o}$ need to be calculated in the backward process, as follows:

$$\frac{\partial z}{\partial x_i} = \frac{\partial z}{\partial y_o} \cdot \frac{\partial y_o}{\partial x_i} = f'(y_o) \star k_i, \tag{4}$$

where $\star$ denotes the correlation operation. Denoting the complex conjugate as $conj$, this correlation is calculated in the frequency domain using the Fourier transform as: $x \star k = \mathcal{F}^{-1}\{\mathcal{F}\{x\} \cdot conj(\mathcal{F}\{k\})\}$.

$$\frac{\partial z}{\partial k^*_{io}} = \frac{\partial z}{\partial y_o} \cdot \frac{\partial y_o}{\partial k^*_{io}} = f'(y_o) \star x_i, \tag{5}$$

where k^* represents the flipped kernel k. Thus, the gradient $\frac{\partial z}{\partial k_{io}}$ is calculated by flipping the correlation output. Finally,

$$\frac{\partial z}{\partial b_o} = \frac{\partial z}{\partial y_o} \cdot \frac{\partial y_o}{\partial b_o} = 1^T \cdot vec(f'(y_o)) \tag{6}$$

In words, the gradient $\frac{\partial z}{\partial b_o}$ can be calculated by point-wise summation of the values in $f'(y_o)$.

3.1.3 Max-pooling Layer

The max pooling layer calculates the largest element in $P_r \times P_c$ windows, with stride size $S_r \times S_c$. A customized *im2col_ln* function is implemented to convert the stridden pooling patches into column vectors, to vectorize the pooling computation in Matlab. The built-in *max* function is called on these column vectors to return the pooling result and the indices of these maximum values. Then, the indices in the original batched data are recovered accordingly. Also, zero padding can be applied to the input data.

Without the loss of generality, the mapping from the max-pooling layer input to the final network output can be expressed as: $z = f(y) = f(Sx)$, where S is a selection matrix, and x is a column vector which denotes the input data in this layer.

In the backward process, $\frac{\partial z}{\partial x}$ is calculated and passed to the shallower layers: $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot S = f'(y)^T S$.

When the pooling range is less than or equal to the stride size, $\frac{\partial z}{\partial x}$ can be calculated with simple matrix indexing techniques in Matlab. Specifically, an empty tensor $dzdx$ of the same size with the input data is created. $dzdx(from) = dzdy$, where $from$ is the pooling indices, and $dzdy$ is a tensor recording the pooling results. When the pooling range is larger than the stride size, each entry in x can be pooled multiple times, and the back propagation gradients need to be accumulated for each of these multiple-pooled entries. In this case, the $\frac{\partial z}{\partial x}$ is calculated using the Matlab function: $accumarray()$.

3.1.4 Rectified Linear Unit

The rectified linear unit ($ReLU$) is implemented as a major non-linear mapping function, some other functions including $sigmoid$ and $tanh$ are omitted from the discussion here. The $ReLU$ function is the identity function if the input is larger than 0 and outputs 0 otherwise: $y = relu(x) = x \cdot ind(x > 0)$. In the backward process, the gradient is passed to the shallower layer if the input data is non-negative. Otherwise, the gradient is ignored.

3.2 Loss function

Usually, a loss function is connected to the outputs of the deepest core computation module. Currently, LightNet supports the softmax log-loss function for classification tasks.

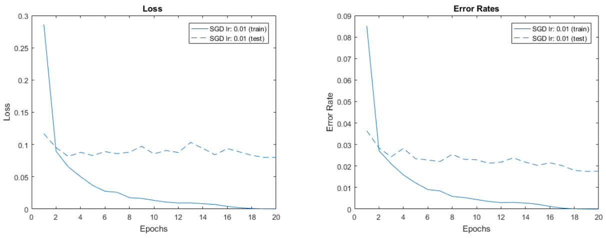
3.3 Optimization Algorithms

Stochastic gradient descent (SGD) algorithm based optimization algorithms are the primary tools to train deep neural networks. The standard SGD algorithm and several of its popular variants such as Adagrad [3], RMSProp [12] and Adam [6] are also implemented for deep learning research. It is worth mentioning that we implement a novel Selective-SGD algorithm to facilitate the selection of hyper-parameters, especially the learning rate. This algorithm selects the most efficient learning rate by running the SGD process for a few iterations using each learning rate from a discrete candidate set. During the middle of the neural net training, the Selective-SGD algorithm can also be applied to select different learning rates to accelerate the energy decay.

4. EXPERIMENTS

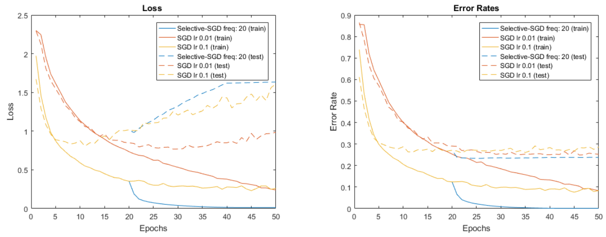
4.1 Multilayer Perceptron Network

A multilayer perceptron network is constructed to test the performance of LightNet on MNIST data [9]. The network takes 28 × 28 inputs from the MNIST image dataset and has 128 nodes respectively in the next two layers. The 128-dimensional features are then connected to 10 nodes to calculate the softmax output. See Fig. 2 for the experiment results.



(a) (b)

Figure 2: Loss and error rates during training and testing phases using LightNet on the MNIST dataset.



(a) (b)

Figure 3: Loss and error rates of training and testing with LightNet on the CIFAR-10 dataset.

4.2 Convolutional Neural Network

LightNet supports using state-of-the-art convolutional network models pretrained on the ImageNet dataset. It also supports training novel network models from scratch. A convolutional network with 4 convolution layers is constructed to test the performance of LightNet on CIFAR-10 data [7]. There are $32, 32, 64, 64$ convolution kernels of size 5×5 in the first three layers, the last layer has kernel size 4×4. $relu$ functions are applied after each convolution layer as the non-linear mapping function. LightNet automatically selects and adjusts the learning rate and can achieve state-of-the-art accuracy with this architecture. Selective-SGD leads to better accuracy compared with standard SGD with a fixed learning rate. Most importantly, using Selective-SGD avoids manual tuning of the learning rate. See Fig. 3 for the experiment results. The computations are carried out on a desktop computer with an Intel i5 6600K CPU and a Nvidia Titan X GPU with 12GB memory. The current version of LightNet can process 750 images per second with this network structure on the GPU, around 5× faster than using CPU.

4.3 LSTM Network

The Long Short Term Memory (LSTM) [4] is a popular recurrent neural network model. Because of LightNet's versatility, the LSTM network can be implemented in the LightNet package as a particular application. Notably, the core computational modules in LightNet are used to perform time domain forward process and back propagation for LSTM.

The forward process in an LSTM model can be formulated as:

$$i_t = sigmoid(W_{ih} h_{t-1} + W_{ix} x_t + b_i), \tag{7}$$

$$o_t = sigmoid(W_{oh}h_{t-1} + W_{ox}x_t + b_o), \tag{8}$$

$$f_t = sigmoid(W_{fh}h_{t-1} + W_{fx}x_t + b_f), \tag{9}$$

$$g_t = tanh(W_{gh}h_{t-1} + W_{gx}x_t + b_g), \tag{10}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t, h_t = o_t \odot tanh(c_t), \tag{11}$$

$$z_t = f(h_t), z = \sum_{t=1}^{T} z_t. \tag{12}$$

Where $i_t/o_t/f_t$ denotes the response of the input/output/forget gate at time t. g_t denotes the distorted input to the memory cell at time t. c_t denotes the content of the memory cell at time t. h_t denotes the hidden node value. f maps the hidden nodes to the network loss z_t at time t. The full network loss is calculated by summing the loss at each individual time frame in Eq. 12.

To optimize the LSTM model, back propagation through time is implemented and the most critical value to calculate in LSTM is: $\frac{\partial z}{\partial c_s} = \sum_{t=s}^{T} \frac{\partial z_t}{\partial c_s}$.

A critical iterative property is adopted to calculate the above value:

$$\frac{\partial z}{\partial c_{s-1}} = \frac{\partial z}{\partial c_s}\frac{\partial c_s}{\partial c_{s-1}} + \frac{\partial z_{s-1}}{\partial c_{s-1}}. \tag{13}$$

A few other gradients can be calculated through the chain rule using the above calculation output:

$$\frac{\partial z_t}{\partial o_t} = \frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial o_t}, \frac{\partial z}{\partial \{i,f,g\}_t} = \frac{\partial z}{\partial c_t}\frac{\partial c_t}{\partial \{i,f,g\}_t}. \tag{14}$$

The LSTM network is tested on a character language modeling task. The dataset consists of $20,000$ sentences selected from works of Shakespeare. Each sentence is broken into 67 characters (and punctuation marks), and the LSTM model is deployed to predict the next character based on the characters before. 30 hidden nodes are used in the network model and RMSProp is used for the training. After 10 epochs, the prediction accuracy of the next character is improved to 70%.

4.4 Q-Network

As an application in reinforcement learning, We created a Q-Network [11] with the MLP network. The Q-Network is then applied to the classic Cart-Pole problem [1]. The dynamics of the Cart-Pole system can be learned with a two-layer network in hundreds of iterations. One iteration of the update process of the Q-Network is:

$$\begin{aligned} Q_{new}(state_{old}, act) &= reward + \gamma Q_{current}(state_{new}, act_{best}) \\ &= reward + \gamma max_a Q_{current}(state_{new}, a) \\ &= reward + \gamma V(state_{new}). \end{aligned} \tag{15}$$

The *action* is randomly selected with probability *epsilon*, otherwise the *action* leading to the highest score is selected. The desired network output Q_{new} is calculated using the observed reward and the discounted value $\gamma V(state_{new})$ of the resulting state, predicted by the current network through Eq. 15.

By using a least squared loss function:

$$\begin{aligned} z &= (y - Q_{current}(state_{old}, act))^2 \\ &= (Q_{new}(state_{old}, act) - Q_{current}(state_{old}, act))^2, \end{aligned} \tag{16}$$

the Q-Network can be optimized using the gradient:

$$\frac{\partial z}{\partial \theta} = \frac{\partial z}{\partial Q_{current}}\frac{\partial Q_{current}}{\partial \theta}. \tag{17}$$

Here θ denotes the parameters in the Q-Network.

5. CONCLUSION

LightNet provides an easy-to-expand ecosystem for the understanding and development of deep neural network models. Thanks to its user-friendly Matlab based environment, the whole computational process can be easily tracked and visualized. This set of the main features can provide unique convenience to the deep learning research community.

6. ACKNOWLEDGEMENTS

This work was funded by the support of the National Science Foundation under grant SMA 1540917 and grant CNS 1544797, and by DARPA through U.S. Army grant W911NF-14-1-0384.

7. REFERENCES

[1] Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *Systems, Man and Cybernetics, IEEE Transactions on*, 5 (1983), 834–846.

[2] Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., Bouchard, N., Warde-Farley, D., and Bengio, Y. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590* (2012).

[3] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research 12* (2011), 2121–2159.

[4] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[5] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia* (2014), ACM, pp. 675–678.

[6] Kingma, D., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[7] Krizhevsky, A., and Hinton, G. Learning multiple layers of features from tiny images, 2009.

[8] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[9] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[10] Mallat, S. *A wavelet tour of signal processing: the sparse way.* Academic press, 2008.

[11] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[12] Tieleman, T., and Hinton, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning 4* (2012), 2.

[13] Vedaldi, A., and Lenc, K. Matconvnet: Convolutional neural networks for matlab. In *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference* (2015), ACM, pp. 689–692.