DLL: A Fast Deep Neural Network Library

Baptiste Wicht[1,2(✉)], Andreas Fischer[1,2], and Jean Hennebert[1,2]

[1] HES-SO, University of Applied Science of Western, Switzerland, Delémont, Switzerland
[2] University of Fribourg, Fribourg, Switzerland
baptiste.wicht@gmail.com

Abstract. Deep Learning Library (DLL) is a library for machine learning with deep neural networks that focuses on speed. It supports feed-forward neural networks such as fully-connected Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs). Our main motivation for this work was to propose and evaluate novel software engineering strategies with potential to accelerate runtime for training and inference'. Such strategies are mostly independent of the underlying deep learning algorithms. On three different datasets and for four different neural network models, we compared DLL to five popular deep learning libraries. Experimentally, it is shown that the proposed library is systematically and significantly faster on CPU and GPU. In terms of classification performance, similar accuracies as the other libraries are reported.

1 Introduction

In recent years, neural networks have regained a large deal of attention with deep learning approaches. Such approaches rely on the use of bigger and deeper networks, typically by using larger input dimensions to incorporate more context and by increasing the number of layers to extract information at different levels of granularity. The success of deep learning can be attributed mainly to three factors. First, there is the advent of *big data*, meaning the availability of larger quantities of training data. Second, new training strategies have been developed, such as unsupervised pre-training that allows deep networks to initialize well and also to learn efficient feature extractors on large sets of unlabelled data. Finally, better and faster hardware has helped dealing with the training of such networks. Deep systems are currently improving the state-of-the-art in many domains. Successful deep learning applications include near-human performance at recognizing objects in images [27], generating detailed image descriptions [13], adding colors to grayscale images [3] or generating highly-realistic images [7]. Moreover, the availability of free and easy-to-use libraries, as well as the availability of detailed implementation examples on public datasets, have contributed to the widespread use of deep learning technologies.

From a practical point of view, an ideal deep learning library would be easy to use, would offer fast training with good precision and would be versatile

L. Pancioni et al. (Eds.): ANNPR 2018, LNAI 11081, pp. 54–65, 2018.
https://doi.org/10.1007/978-3-319-99978-4_4

with many configuration options. Reaching all these qualities is difficult as some are contradictory. For this reason, we may observe large differences among the available libraries.

In this work, we report on the development of a deep learning library where we have clearly opted to focus on efficient computation, targeting specific network models and algorithm configurations. While we are aware of these limitations, we believe that the different optimizations we have implemented may be of interest to the scientific community. Our library, Deep Learning Library (DLL), is freely available, with source code[1]. This library can be used to train standard Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs) [18], as well as Restricted Boltzmann Machine (RBM) [26] and Convolutional RBM (CRBM) [20].

While speedups are also observed on the GPU, the proposed library has been especially optimized for speed on Central Processing Unit (CPU). Although GPUs are beginning to be the de-facto standard for training deep networks, they are not always available and some deployments are still targeting existing CPU implementations. Moreover, inference is generally performed on CPU once the network has been trained. Therefore, we believe that it remains important to be able to both train neural networks in reasonable time and achieve fast inference on CPUs. In this work, we also report successful optimizations on GPU, but we have to note that advanced parallelization capabilities of GPU where already well used [28], especially for convolutional networks [16].

Further to our speedup contributions, a special contribution of this paper is a comprehensive evaluation against several important state of the art libraries. The evaluation is carried on four models and three data sets. Comparisons are performed in terms of computation time on both CPU and GPU. This shows that state of the art libraries have still some large margin of optimization.

The rest of this paper is organized as follows. The DLL library is described in details in Sect. 2. The evaluation is presented in Sect. 3. Section 4 is presenting the results of the experiments on MNIST, Sect. 5 on CIFAR-10 and Sect. 6 on ImageNet. Finally, conclusions are drawn in Sect. 7.

2 DLL: Deep Learning Library

Deep Learning Library (DLL) is a Machine Learning library originally focused on RBM and CRBM support. It was developed and used in the context of several research work [29–32]. It also has support for various neural network layers and backpropagation techniques. It is written in C++ and its main interface is C++ (example in Sect. 2.2). The library can also be used by describing the task in a simple descriptor language, to make it easier for researchers.

The library supports conventional neural network. As such, ANNs and CNNs can be trained. Max Pooling and Average Pooling layers are also supported for CNNs. These networks can be trained with mini-batch gradient descent. The

[1] URL https://github.com/wichtounet/dll.

basic learning options such as momentum and weight decay are supported. The library also support advanced techniques such as Dropout [10] and Batch Normalization [11]. Finally, optimizers with adaptive learning rates such as Adagrad [6], Adadelta [33] and Adam [14] are also integrated. The library also supports Auto-Encoders [2] and Convolutional Auto-Encoders [21].

Also, the library has complete support for the RBM model [26]. The model can be trained using Contrastive Divergence (CD) [9]. The implementation was designed following the model from [8]. It also supports Deep Belief Network (DBN), pretrained layer by layer and then fine-tuned using gradient descent. The RBM supports a wide range of visible and hidden unit types, such as binary, Gaussian and Rectified Linear Unit (ReLU) [23]. Support for CRBM is also integrated, following the two models from [20].

The DLL library is available online[2], free of charge, under the terms of the MIT open source license. Details of the project as well as some tutorials are available on the home page.

2.1 Performance

The focus of the library is runtime performance, for training and for inference.

The implementation uses several techniques to optimize as much as possible the runtime performance for training and inference. First, all the computations are performed using single-precision floating point numbers. This leads to a better data locality and an increased potential for vectorization. On GPU, it would even be possible to use half-precision, but modern processors do not have native capabilities for such computations. Another simple optimization is that all the computations are performed on a batch rather than on one sample at the time. This has the advantage of leveraging the necessary operations to higher level computations. Since this is also generally advantageous for the quality of the training, this is currently the most common way to train a neural network.

The forward activation of a dense layer for a mini-batch can be computed with a single matrix-matrix multiplication [31]. This is also possible for the backward pass, by transposing the weight matrix. Finally, the gradients for the dense layer can also be computed using one matrix-matrix multiplication. Thus, such a network mainly needs a good implementation of this operation to be fast.

The Basic Linear Algebra Subprograms (BLAS) interface contains a set of small and highly-optimized kernels for matrix and vector computation [17]. When using an efficient BLAS library, the matrix-matrix multiplication operation can be very efficient. Moreover, using a parallel BLAS library also leads to significantly increased performance for large layers. Moreover, although BLAS libraries are highly optimized for very large matrices, they are not as fast as possible for small matrices. Therefore, we automatically detect such cases and use custom vectorized kernels for small matrix multiplications.

Optimization is more complicated for CNNs. Indeed, the dense layers only account for a small portion of the training time. Convolutional layers use two

[2] URL https://github.com/wichtounet/dll.

forms of convolution. A valid convolution for the forward pass, which shrinks the representation and a full convolution for the backward pass to expand it. Every image batch is convolved with K kernels. It is possible to rearrange an image into columns so that a matrix-matrix multiplication can be used to compute the K valid convolutions of the image at once [24,31]. This proved to be very efficient for large images or large kernels. When images or kernels are small, it is not efficient since the rearranging of the input matrix is a slow operation. Therefore, in these cases, we observed that it is more interesting to perform a real convolution using an highly-optimized implementation. First, several floating point operations are computed during the same CPU cycle, using SSE and AVX, a technique known as Single Instruction Multiple Data (SIMD). Then, to ensure the maximum throughput, the matrices are padded so that the last dimension is a multiple of the vector size. Specialized kernels for the most used kernel sizes, such as 3×3 and 5×5, are also used. Finally, most of the convolutions can be performed in parallel since there are no dependencies between them. This proved significantly faster than the reduction to a matrix-matrix multiplication in several configurations.

There are several possible implementations for the full convolution. First, it can be expressed in terms of another operation, the Fast Fourier Transform (FFT) [22]. For this, the input image and the kernel are padded to the size of the output. Then, their transforms are computed, in parallel. The Hadamard product of the input image with the transform of the kernel is computed. The inverse transform of this product is the full convolution. Computing several convolutions of the same image with different kernels is more efficient since the image transform is only computed once. In our experiments, we observed that such implementation is very efficient for large inputs and large kernels, but it is not as interesting for small configurations. With very small kernels, it is more efficient to pad the input and the kernels and perform a valid convolution. Indeed, a full convolution is equivalent to a valid convolution with some amount of padding. When the necessary padding is small enough, it becomes significantly faster than performing the FFTs. The last option is to use an optimized implementation of the full convolution. However, due to the large number of border cases, this would only be faster than the implementation as a valid convolution for large dimensions, in which case the reduction to FFT would be faster.

Since there is no one-size-fits-all implementation for all configurations, heuristics are used to select the most suited implementations. These heuristics are based on the size of the convolution kernels and the size of the batch.

Although most of the time is contained inside the previously mentioned operations, it is still important to optimize the other operations such as activation functions and gradient computations. In our implementation, these operations are vectorized and parallelized to maximize the processor utilization.

Fortunately, when optimizing for GPU, most of the routines are already implemented in highly specialized libraries. DLL uses NVIDIA libraries in order to optimize most kernels. NVIDIA CUBLAS is used for the matrix-matrix multiplications and a few other linear algebra operations and NVIDIA CUDNN [4]

is used for the machine learning operations such as convolutions, activation functions and gradients computation. For other operations, CUDA kernels have been written to ensure that most of the time is spent on the GPU. When optimizing for GPU, it is most important to avoid copies between the CPU and GPU. Moreover, most of the kernels are launched asynchronously, without device synchronization. This significantly reduces the overhead of CUDA kernel calls.

2.2 Example

Figure 1 shows the code necessary to train a three-layer fully-connected network on the MNIST data set with the DLL library. The code starts by loading the MNIST data set in memory. Then, the network is declared layer by layer. After that, the network training parameters are set and the training is started. Finally, the accuracy on the test set is computed.

```
using namespace dll;

auto dataset = make_mnist_dataset(batch_size<100>{}, scale_pre<255>{});

using network_type = network_desc<
    network_layers<
        dense_layer<28 * 28, 500, sigmoid>,
        dense_layer<500,     250, sigmoid>,
        dense_layer<250,     10,  softmax>
    >
    , updater<updater_type::MOMENTUM>
    , batch_size<100>
>::network_t;

auto net = std::make_unique<network_type>();

net->learning_rate = 0.1;
net->momentum = 0.9;

net->display();
net->fine_tune(dataset.train(), 50);
net->evaluate(dataset.test());
```

Fig. 1. Example to train and evaluate a dense network on the MNIST data set.

3 Experimental Evaluation

We compared our library against popular libraries on four experiments. The time to train each model is compared for each library, on CPU and on GPU. Each experiment was run five times. And for each library, the best time is kept as the final measure. There is no significant different between the different runs. Their accuracy was also computed. It was shown that all the tested libraries were all exhibiting comparable accuracy when trained with the same parameters. For lack of space, these results are not shown here.

The following reference libraries have been selected:

1. Caffe [12]: A high-level Machine Learning library, focusing on speed and expression, developed in C++ and used through a text descriptor language. Caffe 1.0 was installed from the sources with GPU and MKL support.

2. TensorFlow [1]: A general low-level library, allowing expressing a data flow graph to perform numerical computation. The core of the system is written in C++, but the features are used in Python. Tensorflow 1.3.1 was installed from the sources with CUDA, CUDNN and MKL support.
3. Keras[3]: A high-level Machine Learning library, providing a frontend for Tensorflow and Theano, written in Python. It provides a large number of high-level models, easing the development of Machine Learning models. The version 2.0.8 was installed using the official package with Tensorflow 1.3.1.
4. Torch [5]: Torch is another low-level Machine Learning library, one of the earliest, started in 2002. It is used through a Lua front-end. Although it is a low-level library, it also contains high-level modules for Machine Learning. It was installed from the sources, from Git commit 3e9e141 with CUDA and MKL support.
5. DeepLearning4J[4]: DeepLearning4J is a deep learning library for Java, written in Java, C and C++. It has a very large set of features and focuses on distributed computing. The version 0.9.1 was used, from Maven.

The libraries have been selected based on their popularity and also to have a broad range of programming languages. DLL is used directly from the sources, with the latest version available at this time (Git commit 2f3c62c).

We are underlying here that the goal of these experiments is not to reach state of the art performance on the tested data sets. The models are kept simple to allow comparison with a wider range of libraries. Moreover, the networks are not always trained for as many epochs as they would be, if achieving high accuracy was the goal. Finally and very importantly, we are not aware of the full details of all the libraries. We did our best to have similar network architecture and training parameters, but it could be that some implementation details lead to slightly different training, explaining time differences.

All the results presented in this chapter have been computed on a Gentoo Linux machine, on an Intel® Core™ i7-2600, running at 3.4 GHz (CPU frequency scaling has been disabled for the purpose of these tests). Both SSE and AVX vectorization extensions were enabled on the machine. BLAS operations are executed with the Intel® Math Kernel Library (MKL), in parallel mode. The GPU used is a NVIDIA Geforce® GTX 960 card. CUDA 8.0.4.4 and CUDNN 5.0.5 are used. The source code used for these experiments is available online[5].

All the experiments are trained using mini-batch gradient descent. The last layer of each network is always a softmax layer. The loss is a softmax cross entropy loss.

4 MNIST

The first experiment is performed on the MNIST data set [19]. It is a digit recognition task. The data set is made of 60'000 28×28 grayscale images for

[3] https://github.com/fchollet/keras.
[4] http://deeplearning4j.org.
[5] https://github.com/wichtounet/frameworks.

training and 10'000 images for testing. It is a very well-known data set and has been repeatedly used with most of the existing Machine Learning algorithms. Although it is considered an easy task, it remains an excellent problem for comparing libraries since most of them use it as example and have code available.

4.1 Fully-Connected Neural Network

The first tested network is a fully-connected three-layer ANN with 500 units in the first layer, 250 in the second layer and 10 final output units for classification. The first two layers are using the sigmoid function. The network is trained with mini-batches of 100 images, for 50 epochs, with a learning rate of 0.1 and a momentum of 0.9. The training accuracy is computed after each epoch and the test accuracy is computed after the end of the complete training. As an example, the code using the DLL library is presented in Fig. 1.

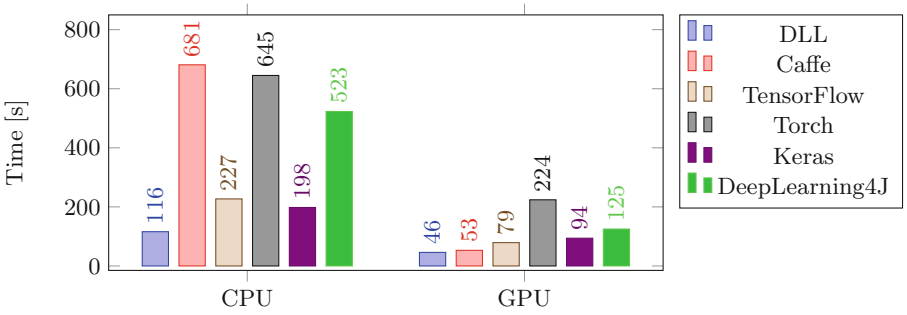

Fig. 2. Training time performance of the libraries for an ANN, on MNIST

Figure 2 presents the performance of each of the libraries. In CPU mode, DLL outperforms all the other libraries, being around 40% faster than TensorFlow and Keras, 4.5 times faster than DeepLearning4J and 5.5 times faster than Torch and Caffe. On GPU, DLL is the fastest library, closely followed by Caffe. DLL is about 40% faster than TensorFlow and twice faster than Keras. DeepLearning4J and Torch are respectively 2.5 and 5 times slower than DLL.

4.2 Convolutional Neural Network

The second network, for the same task, is a small CNN with six layers. The first layer is a convolutional layer using 8 5×5 kernels and followed by a max pooling layer with a 2×2 kernel. The third and fourth layers are using the same configuration. The last layers are fully-connected, the first with 150 units and the last with 10 units for classification. The two convolutional layers and the first fully-connected layer use a sigmoid activation function. The full network is trained in the same manner as the first network.

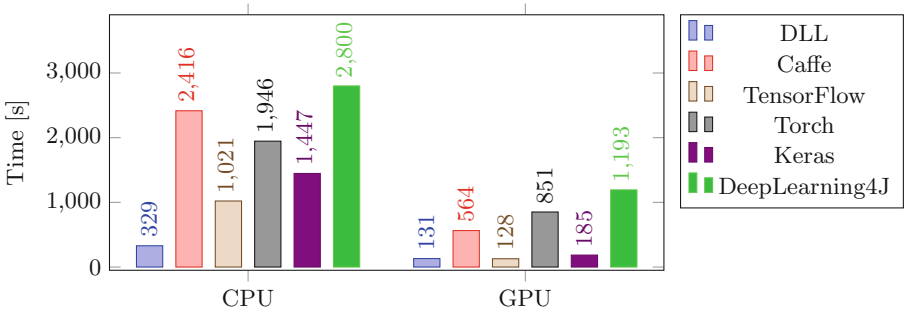

Fig. 3. Training time performance of the libraries for a CNN, on MNIST

Figure 3 presents the results obtained on this experiment. Again, DLL is the fastest library on CPU, by a significant margin, three times faster than TensorFlow and almost four times faster than Keras. DLL is more than 8 times faster than the slowest library, DeepLearning4J. This shows the effects of the in-depth CPU optimization of the convolutions. On GPU, TensorFlow and DLL are the fastest libraries, about 30% faster than Keras and significantly faster than Caffe (4 times), Torch (6.5 times) and DeepLearning4J (9 times).

5 CIFAR-10

The second data set that is tested is CIFAR-10 [15], a data set for object recognition, consisting of 50'000 images for training and 10'000 for testing, in 10 different classes. The data set is composed of colour images of 32×32 pixels.

A larger CNN is used for this task. The first layer is convolutional with 12 5×5 kernels, followed by a 2×2 max pooling layer. They are followed by another convolutional layer with 24 3×3 kernels and a 2×2 max pooling layer. A dense layer with 64 hidden units is then used, followed by a softmax layer with 10 output units. All the layers but the last one are using ReLUs. The network is trained similarly to the previous networks, with a learning rate of 0.001.

In Fig. 4, the training times for this task are presented. The speedups are less significant than for the previous CNN. Nevertheless, DLL still manages to be the fastest library on CPU. It is about twice faster than TensorFlow, Keras, DeepLearning4J and Torch and about three times faster than Caffe. On GPU, DLL is also the fastest library on this experiment, about 30% faster than TensorFlow and 40% faster than Keras. It is three times faster than Caffe and about 4.5 times faster than Torch and ten times faster than DeepLearning4J. This network is significantly larger than in the MNIST experiment. This seems to indicate that most libraries are more optimized for larger networks. This shows that GPU performance is better when a lot of data is available.

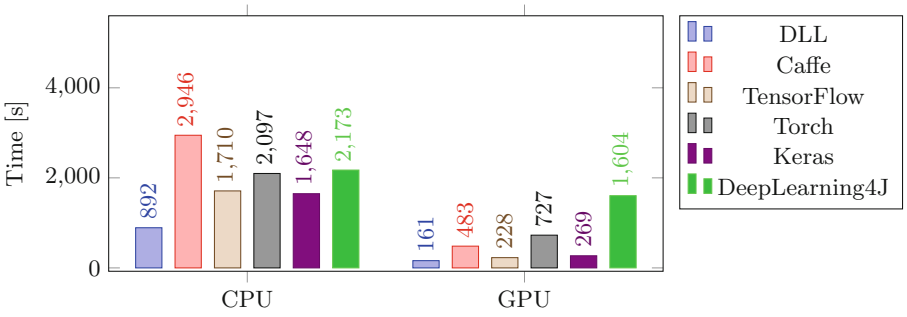

Fig. 4. Training time performance of the libraries on the CIFAR-10 task

6 ImageNet

The last experiment is performed on ImageNet, a large data set for image classification. We consider the sub part of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012 [25], there are 50'000 validation images, 100'000 test images, around 1.2 million training images and 1000 categories. All the images have been resized to 256×256 images.

The entire data set cannot be kept in memory. Therefore, the images are loaded from the disk for each epoch. For this experiment, only Caffe provides an official, up-to-date, code for this data set. The DeepLearning4J reader was based on existing official reader for structures similar to ImageNet. For Keras, TensorFlow and Torch, a simple data reader has been written with the image loading tools available in each library.

The network is significantly larger than the previous networks. It is made of five convolutional layers, with 16 3×3 kernels for the first two layers and 32 3×3 kernels for the next three layers. Each of these layers is followed by a ReLU activation function and a 2×2 max pooling layer. All the convolutional layers are using zero-padding so that their output is the same size as their input The last two layers are a dense layer with 2048 hidden units, with a ReLU function and a dense layer with 1000 outputs. The training is different than for the other data sets. The full network is only trained for five epochs with each library. The networks are trained using a batch size of 128. However, Torch and DeepLearning4J models were trained with a batch size of 64, respectively 16, samples. Indeed, both of these libraries needed more than 12GB of RAM to train with a batch size of 128 images. This may lead to some small degradation of the performance for those two libraries.

For the sake of comparison, the average time to train one batch of samples is used as results. For Torch and DeepLearning4J, the results are the times for several batches, to make up for 128 samples. These results are presented in Fig. 5. DLL shows to be again the fastest library on CPU for training this large model, 35% faster than Keras, about 45% faster than TensorFlow and twice faster than Caffe. Torch is already more than 3 times slower than DLL and DeepLearning4J

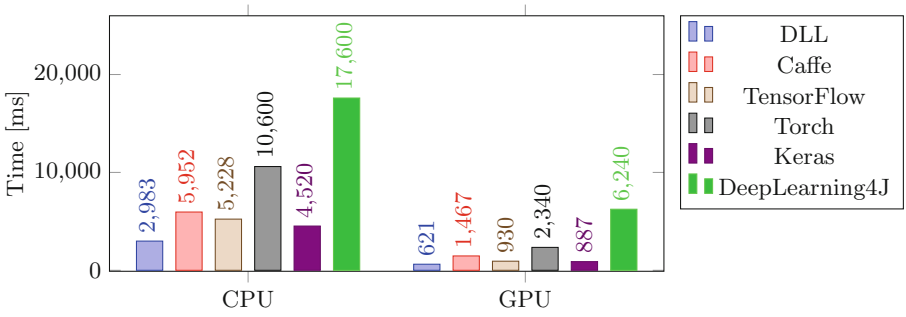

Fig. 5. Training time performance of the libraries, on ImageNet. The time is the average time necessary for the training of one batch of 128 elements.

around 6 times slower. On GPU, DLL is, also, the fastest library. Comparisons with Keras and TensorFlow show that most of the difference comes from the poor performance of reading the ImageNet data from the Python code. Once this is taken into account, the three libraries have comparable performance. DLL is more than twice faster than Caffe and almost four times faster than Torch and almost 10 times faster than DeepLearning4J.

7 Conclusion and Future Work

For all the experiments and the different neural networks models that were tested, the DLL library has shown to be the fastest gradient descent based library for training the model when using CPU and GPU. For each test, the accuracies of the models trained with DLL are similar to the models trained by the other five Machine Learning libraries.

The speedups provided by the library on CPU mode are especially important for convolutional layers for which advanced optimization was performed. The library was especially optimized for small convolutions, but is still able to bring significant speedups for large images such as the images from the ImageNet data set. Moreover, while some libraries are mostly optimized for the convolutional and fully-connected parts of the computation, every part of the training in the DLL library was tuned. However, since DLL is written in C++, programs using it need to be compiled. This may make it more complicated for researchers to use. Finally, while the language itself is very common about performance software developers, it is not very common for machine learning researchers. Therefore, there is more of a barrier for use compared to libraries using more common languages for machine learning.

A few DLL routines are not optimized enough for GPU, such as Dropout and Batch Normalization. Future work could also include better support for Recurrent Neural Networks (RNNs), which would be a great advantage for the library. Finally, the library has currently been optimized only on few machines and especially consumer grade processors and graphics cards. It would be greatly

beneficial to take advantage of more threads or advanced vectorization capabilities such as those provided by the latest Intel® Xeon processors or more recent and more powerful NVIDIA graphics cards.

References

1. Abadi, M., et al.: TensorFlow: large-scale machine learning on heterogeneous systems (2015). http://tensorflow.org/
2. Bengio, Y.: Learning deep architectures for AI. Foundations and trends® in Machine Learning, pp. 1–127 (2009)
3. Cheng, Z., Yang, Q., Sheng, B.: Deep colorization. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 415–423 (2015)
4. Chetlur, S., et al.: cuDNN: efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)
5. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: a matlab-like environment for machine learning. In: BigLearn, NIPS Workshop. No. EPFL-CONF-192376 (2011)
6. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. **12**, 2121–2159 (2011)
7. Goodfellow, I., et al.: Generative adversarial nets. In: Advances in Neural Information Processing Systems, pp. 2672–2680 (2014)
8. Hinton, G.E.: A practical guide to training restricted Boltzmann machines. In: Montavon, G., Orr, G.B., Müller, K.-R. (eds.) Neural Networks: Tricks of the Trade. LNCS, vol. 7700, pp. 599–619. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35289-8_32
9. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Comput. **18**, 1527–1554 (2006)
10. Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.R.: Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580 (2012)
11. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167 (2015)
12. Jia, Y., et al.: Caffe: convolutional architecture for fast feature embedding. arXiv preprint arXiv:1408.5093 (2014)
13. Karpathy, A., Fei-Fei, L.: Deep visual-semantic alignments for generating image descriptions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3128–3137 (2015)
14. Kingma, D., Ba, J.: Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
15. Krizhevsky, A., Hinton, G.E.: Learning multiple layers of features from tiny images. Technical report (2009)
16. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp. 1097–1105 (2012)
17. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic linear algebra subprograms for FORTRAN usage. ACM Trans. Math. Softw. (TOMS) **5**, 308–323 (1979)
18. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. In: Proceedings of the IEEE, pp. 2278–2324 (1998)

19. LeCun, Y., Cortes, C., Burges, C.J.C.: The mnist database of handwritten digits (1998). http://yann.lecun.com/exdb/mnist/. Accessed 04 Feb 2018
20. Lee, H., Grosse, R., Ranganath, R., Ng, A.Y.: Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In: Proceedings of the 26th Annual International Conference on Machine Learning, pp. 609–616. ACM (2009)
21. Masci, J., Meier, U., Cireşan, D., Schmidhuber, J.: Stacked convolutional auto-encoders for hierarchical feature extraction. In: Honkela, T., Duch, W., Girolami, M., Kaski, S. (eds.) ICANN 2011. LNCS, vol. 6791, pp. 52–59. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21735-7_7
22. Mathieu, M., Henaff, M., LeCun, Y.: Fast training of convolutional networks through FFTs. arXiv preprint arXiv:1312.5851 (2013)
23. Nair, V., Hinton, G.E.: Rectified linear units improve restricted Boltzmann machines. In: Proceedings of the International Conference on Machine Learning, pp. 807–814 (2010)
24. Ren, J.S., Xu, L.: On vectorization of deep convolutional neural networks for vision tasks. arXiv preprint arXiv:1501.07338 (2015)
25. Russakovsky, O., et al.: ImageNet large scale visual recognition challenge. Int. J. Comput. Vis. (IJCV) **115**, 211–252 (2015)
26. Smolensky, P.: Information processing in dynamical systems: foundations of harmony theory. Technical report, Colorado University (1986)
27. Szegedy, C., et al.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9 (2015)
28. Upadhyaya, S.R.: Parallel approaches to machine learning: a comprehensive survey. J. Parallel Distrib. Comput. **73**, 284–292 (2013)
29. Wicht, B.: Deep learning features for image processing. Ph.D. thesis, University of Fribourg (2018)
30. Wicht, B., Fischer, A., Hennebert, J.: Deep learning features for handwritten keyword spotting. In: 2016 23rd International Conference on Pattern Recognition (ICPR), pp. 3434–3439. IEEE (2016)
31. Wicht, B., Fischer, A., Hennebert, J.: On CPU performance optimization of restricted Boltzmann machine and convolutional RBM. In: Schwenker, F., Abbas, H.M., El Gayar, N., Trentin, E. (eds.) ANNPR 2016. LNCS (LNAI), vol. 9896, pp. 163–174. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46182-3_14
32. Wicht, B., Hennebert, J.: Mixed handwritten and printed digit recognition in sudoku with convolutional deep belief network. In: 2015 13th International Conference on Document Analysis and Recognition (ICDAR), pp. 861–865. IEEE (2015)
33. Zeiler, M.D.: Adadelta: an adaptive learning rate method. arXiv preprint arXiv:1212.5701 (2012)