Fully Convolutional Neural Networks in Julia

Victor Jakubiuk

Fall 2015

18.337: Numerical Computing With Julia
Massachusetts Institute of Technology

1 Introduction

The goal of my research in computational neuroscience is to design and implement a high-performance data processing pipeline for electromicroscopy images (*EM*). We begin with a 3-dimensional volume, consisting of a number of 2-dimensional cross-sections (slices) that we want to transform into a 3-dimensional map of a mammal's brain (called *connectome*).

One of the stages in the pipeline, and also the most important for error-free brain reconstruction, is the probability map generation. A probability map of a (2-dimensional) slice is a pixel map indicating the probability of each EM input pixel being a *membrane* or a *non-membrane* (extracellular space, or a cell's inner body). A value of 1 indicates a membrane, and 0 indicates a non-membrane. Since the input images are single-channel 8-bit (grayscale), for convenience, we discretize and normalize the values in probability maps into range 0..255 (see figure 1).

It may seem like a straightforward task to perform a number of simple operations (thresholding, contrasting etc.), or even to use the Canny edge detector to transform a raw EM image into the probability map, but this is unfortunately not the case. The amount of noise and uncertainty prevents these naive techniques from producing accurate results.

Traditionally, good results have been obtained using *random forest classifiers*. Currently, state-of-the-art results rely on the use of *convolutional neural networks*, which, for performance reasons are mostly implemented in *CUDA* or C++, making them inconvenient for quick prototyping or testing.

Not only is *Julia* a great language for rapid prototyping, with scientific toolkit capabilities on-par with MATLAB, but it also provides advanced built-in paralellization primitives, and performs close to C++ speed. Thus, in this project I extend the Julia's Mocha.jl library with support for fully-convolutional neural networks that are used in image segmentation. This enables us to generate probability maps on multi-core CPUs an order of magnitude faster than running a window-based prediction on top-performing NVIDIA GPU cards.

2 Current Libraries

As of this writing, *Caffe* is the most popular, state-of-the-art C++ neural network library developed by the Berkley Vision and Learning Center [4]. It is specifically designed for practitioners of machine learning and computer vision (provides MATLAB bindings), both for training and classifying. It supports execution both on CPUs and GPUs, and comes with a publicly available repository of networks, called Model Zoo. Additionally it has a strong support of NVIDIA, which provided its own training package with a convenient user interface, call DIGITS [1].

Mocha.jl is its Julia's counterpart, heavily inspired by and compatible with Caffe. Mocha supports multiple *backends*: the pure Julia backend (convenient to modify), the C++ backend ($2x$ faster than native Julia's) and the GPU backend

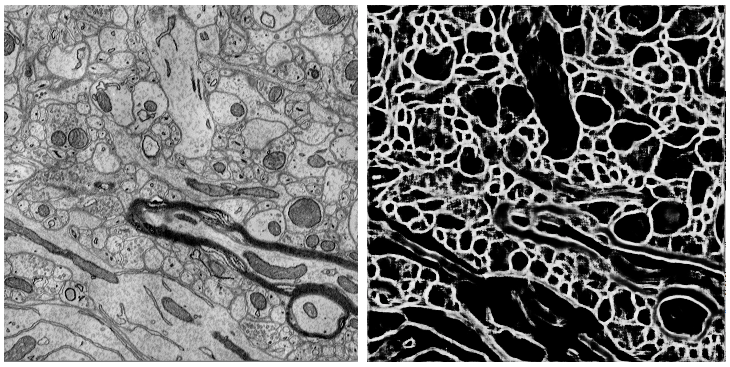


Figure 1: **Left:** An EM image (1 slice). **Right:** Its probability map.

(using cuDNN). A major advantage of this library is that it can directly load Caffe-trained models, enabling my team and I to re-use our existing pre-trained models. In this work, I have extended the pure Julia backend to support fully-convolutional networks.

3 Networks theory

A *deep neural network* (DNN) consists of the input layer, the output layer, and multiple "hidden" layers of units in between [2]. Each layer takes as the input the output of the preceding layer, executes its own specific differentiable function (usually a combination of linear and non-linear function), and passes its output to the following layer. The advantage of a deep network over a *shallow* network is its ability to compose features from lower layers, and thus model more complex data with fewer units than would be required for a similarly performing shallow network. Most DNNs are feedforward networks - data passes from the input towards the output without cycles, forming a directed acyclic graph, though recurrent neural networks are also known.

The *convolutional layer* is the core of the CNN. Each layer consists of a set of filters (kernels) of small spatial dimensions (usually between 3x3 and 19x19). During the prediction phase (forward pass), each kernel convolves (slides) across the width and height of the input image, producing a 2D activation map. The convolution is, essentially, a discrete dot product of the kernel parameters and the input (in our case, the "central" pixel and its neighborhood).

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] = \sum_{m=-\infty}^{\infty} f[n-m]g[m]$$

Intuitively, once the network learns the kernel's parameters, the kernel will activate when it "sees" this specific feature at some place in the input (image). Since multiple kernels are convolved with the input layer, the output consists of

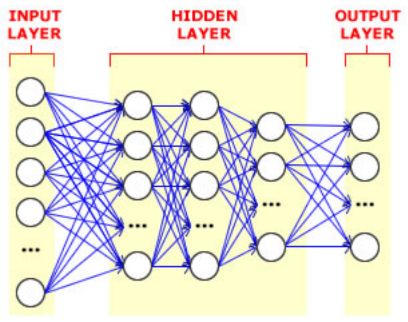


Figure 2: Conceptual representation of a deep neural network

a stack activation maps. For example, if the input layer is a volume 1024x1024x3 (such as a 3-channel RGB image), and the square *kernel size* is 7, then the kernel will need to learn $7x7x3 = 147$ weights. Additionally, sometimes we specify the *stride* of the layer, which correspond to the number of pixels skipped during the convolution. If the stride is 1, then we convolve every input pixel, but for strides $S > 1$, we conolve every S^{th} input pixel. Since the kernel is of size > 1, we sometime need to pad the input layer to preserve the same dimensions in the output layer. If the input layer is of size NxN, the kernel size is K, padding P and the stride S, then the output layer will have dimensions:

$$N_{output} = \frac{N - K + 2P}{S}$$

Thus, by stacking convolutional layers together, we can reduce the input layer size, but the number of convolutions (multiplications) is high, and its usually the most computationally expensive layer.

The *pooling layer* progressively reduces the dimensions of the input layer, which is helpful in minimizing the number of parameters and multiplications required throughout the network, as well as in decreasing overfitting. The most common pooling function is the max kernel, but averaging or l^2-norm are also used. The kernels are usually small and, similarly to the convolutional layer, the stride is also applied. The most popular variations are of $K = 2, S = 2$ and $K = 3, S = 2$. Thus, for the input layer of dimensions N_{in} x N_{in} x D_{in}, kernel size K and stride S, the output has dimensions:

$$N_{out} = \frac{N_{in} - K}{S}$$
$$D_{out} = D_{in}$$

The *fully connected* layer (also known as the *inner product*) has full connections to all activation maps in the previous layer. It simply multiplies the input by a weight matrix and introduces a bias offset. In some way, the fully

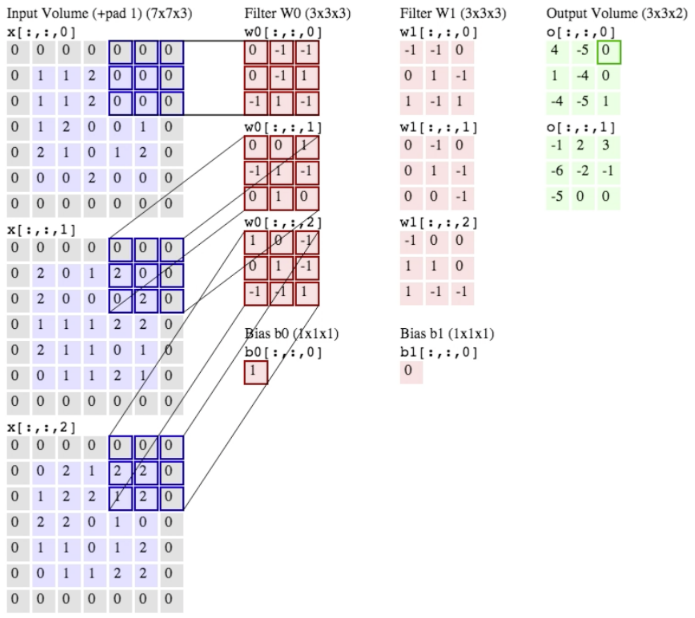


Figure 3: An example of the convolutional kernel

connected layer is equivalent to the convolutional layer. Whereas the convolutional layer is connected to a local region in the input, the fully connected layer is connected to all inputs. One can be easily converted into the other.

4 Theoretical improvement

4.1 Sliding Window Classification

All of our networks are trained on small, labeled patches. Each 2-dimensional patch classifies exactly one pixel from our training set either as a membrane or non-membrane. The patch defines a small neighborhood of odd dimensions, between 19px by 19px and 255px by 255px, to ensure that the middle pixel is unambiguously discriminated. In a single forward pass through the network, we provide such a neighborhood (most commonly 49px by 49px) and classify the central pixel as the output.

However, our EM image slices have dimensions between 1024 by 1024 pixels to 16,000 by 16,000 pixels, so a single pass of such networks cannot classify each pixel in the input. Instead, we use the sliding window technique, where a square window of our network's input size (ie. 49x49) slides across the entire image. If the network's input size is KxK pixels, and the image size is N_{input} x N_{input}, the output image would have dimensions of $N_{output} = N_{input} - K + 1$. To avoid this decrease in size, we pad the image with a border of width $\lfloor K/2 \rfloor$, filled with 0 value, or mirror of the image values.

This results in a significant increase in the number of computations: instead of being forward propagated through the network once, each pixel in the input image contributes to K^2 forward computations now.

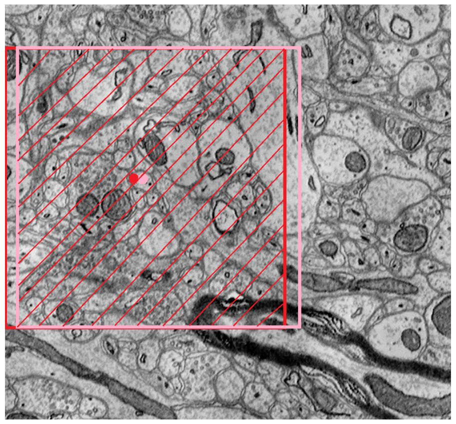


Figure 4: The sliding window approach - notice (on the high-level) that majority of computations are repeated.

4.2 Fully Convolutional Networks

This computational blow up quells even the most power GPUs. For example, on a NVDIA Quadro K6000 with 12GB memory, processing a 1024px by 1024px grayscale image (1MB) with the sliding window of size $K = 49$ and the AlexNet network takes a staggering 30 minutes (forward propagation only), and clearly is not a viable approach.

Looking closer into the computations performed across all $(1024 - K)^2$ windows positions, we observe that most of them are repeated, or follow a very similar pattern. Using the dynamic programming technique, we can significantly speed it up, preserve exactly the same output ([3], [5]) and re-use the same weights from the original (patches-based) classifiers. Specifically, let's optimize the convolutional and max-pooling layers.

Let $L + 1$ by the number of layers in a network. $l = 0$ is the input map, and the max-pooling and convolutional layers are indexed $1..L$. Let P_0 represent the input image with one or more input maps (for example, equal to the number of input image's channels) of width and height w_0 (for simplicity, we assume they are all square).

If the l^{th} layer is a convolution with kernel size k, then its output P_l will be a set of square maps, each of size $w_l = w_{l-1} - k$. In general, $|P_l| \neq |P_{l-1}$, unless $k = 1$. If the l^{th} layer is a max-pooling layer with kernel size k, and

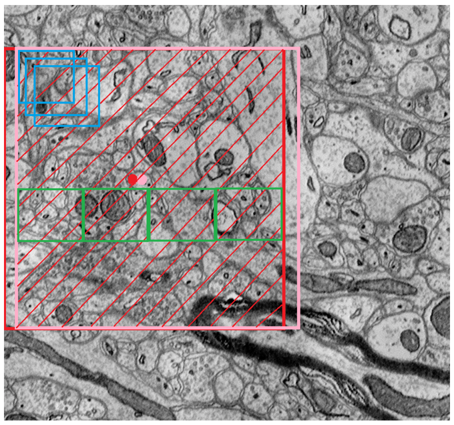


Figure 5: Individual kernels (blue - convolutional, green - max-pooling) are also repeated within the sliding window.

since max-pooling processes every input map, then we get that $|P_l| = |P_{l-1}|$, and $w_l = w_{l-1}/k$, assuming that $w_l \equiv 0 (\text{mod } k)$.

With a patch-based (window-sliding) approach, we are assuming that $w_0 = s$, the size of the input patch. Instead, let's take an input image $s > w_0$. Define F_l as a set of *fragments*, where each fragment f (indexed $1..F_l$) is associated with a set of I_l^f *extended* maps, each of the same size. Define $s_{x,l}^f$ and $s_{y,l}^f$ as width and height of the extended map in I_l^f. Thus, for the $l = 0$, we get $|F_0| = 1$ and $s_{x,0}^1 = s_{y,0}^1 = s$.

In the convolution layer l, the number of fragments is the same as in the input, $F_l = F_{l-1}$ and each extended map shrinks by the convolutional kernel size, that is $s_{x,l}^f = s_{x,l-1}^f - k + 1$ and $s_{y,l}^f = s_{y,l-1}^f - k + 1$. I_l^f is obtained by applying the convolutional kernel in same way as in the window-sliding method to preceding map I_{l-1}^f.

The max-pooling layer computes the kernel at k^2 offsets for each fragment, thus $F_l = k^2 F_{l-1}$. Specifically, let I_{l-1}^f be the input extended map, and $O = \{0, 1, ..., k-1\} \times \{0, 1, ..., k-1\}$ the set of offsets. The for each offset $(o_x, o_y) \in O$, an output extended map I_l^f is created, such that each of its pixels (x_o, y_o) corresponds to the maximum value of all pixels (x, y) in I_{l-1}^f, such that:

$$o_x + kx_o \leq x \leq o_x + kx_o + k - 1$$

$$o_y + ky_o \leq y \leq o_y + ky_o + k - 1$$

While the number of output maps is k^2 times the number of input maps, each output map's size decreases by a factor of k: $s_{x,l}^f = \lfloor \frac{s_{x,l-1}^{f'} - o_x}{k} \rfloor$ and $s_{y,l}^f = \lfloor \frac{s_{y,l-1}^{f'} - o_y}{k} \rfloor$.

Lets computer the theoretical speed up in convolutional layers. Let C_l be the number of floating point calculation required per layer l. In the sliding-window approach, let $|P_l|$ be the total number of maps, s the size of the image, w_l the size of the map, then:

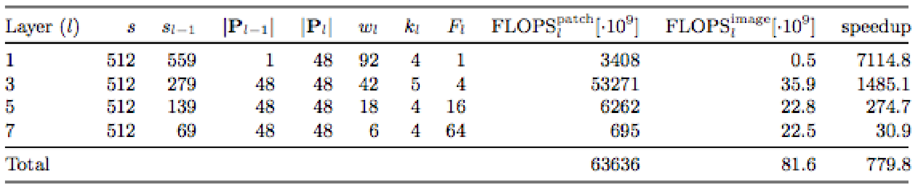
$$C_l = s^2 \cdot |P_{l-1}| \cdot |P_l| \cdot w_l^2 \cdot k_l^2 \cdot 2$$

The extra factor of 2 comes from performing one addition and one multiplication per weight.

In the fully convolutional approach, the number of operations is given by:

$$C_l = s_{x,l} \cdot s_{y,l} \cdot |P_{l-1}| \cdot |P_l| \cdot F_l \cdot k_l^2 \cdot 2$$

For example, taking a 7-layer network (a representative network for EM segmentation) consisting of only convolutional and max-pooling layers, we could obtain the following theoretical speedup:



Layer (l)	s	s_{l-1}	$\|\mathbf{P}_{l-1}\|$	$\|\mathbf{P}_l\|$	w_l	k_l	F_l	FLOPS$_l^{\text{patch}}[\cdot 10^9]$	FLOPS$_l^{\text{image}}[\cdot 10^9]$	speedup
1	512	559	1	48	92	4	1	3408	0.5	7114.8
3	512	279	48	48	42	5	4	53271	35.9	1485.1
5	512	139	48	48	18	4	16	6262	22.8	274.7
7	512	69	48	48	6	4	64	695	22.5	30.9
Total								63636	81.6	779.8

5 Julia implementation

One of the strengths of Mocha.jl is its full compatibility with Caffe - in particular, the ability to re-use their trained models. Caffe comes with an impressive library called Model Zoo, as well as strong support of NVIDIA and the community. I believe it did not make sense to re-invent the wheel and re-implement all these packages in native Julia. However, having the ability to do the research in Julia and see the output of my networks in *minutes* (using fully-convolutional networks) instead of *hours* (with the sliding window) is extremely helpful.

Thus, I implemented FullMocha.jl as an extension to Mocha.jl, without changing most of the syntax and the API, and instead just re-implemented parts of the Julia backend to support fully-convolutional networks. Thus, the end-user, can re-use their existing Mocha code, and simply replace the call to *solve(solver, net)* with *solveFully(solver, net)*, and immediately benefit from our algorithmic speedup.

While the sliding window approach is embarrassingly parallelizable, either with the DArray or the SharedArray, this naive approach does not help with our data sizes, thus it has not been benchmark here. To take the full advantage of our multi-core machines, I used the SharredArray to share data (resulting maps) in between processes. The speedups obtained are on par with our theoretical expectations. Additionally, the C++ fully-convolutional library has been

Table 1: Fully-convolutional speedup

Name	**Classification time (s)**	**Slowdown**	**Speedup**
Python/Caffe, GPU	180s	1	1
Python/Caffe, CPU	954s	5.3	0.19x
Julia/Mocha, CPU	1526s	8.5	0.12x
Fully Conv Julia, CPU	47s	0.26	3.8x
Native C++ Fully Conv	3.8s	0.002	47x

benchmark, with the results provided in table 1 (forward-pass on a 100px by 1024px EM image, 7-layer network).

6 Summary

The results above clearly show that using the fully-convolutional approach significantly speeds up full image classification. Not only is the Julia's fully-convolutional CPU implementation faster than the window-based Julia or Python implementations, it is also $4x$ faster than the GPU implementation running on top-of-the-line NVIDIA Quadro card. If similar use cases are sufficiently common, and this performance benefit can be maintained in production-quality code, this result may effect NVIDIA's push towards CNN networks.

Additionally, my lab-mate implemented a highly-optimized C++ version that provides another magnitude in speed-up (due to efficient cache access patterns and no inter-process SharedArray communication). It should be possible to plug-in this C++ implementation as Mocha's back-end to get more speed-up. This is potential further work.

FullMocha.jl currently only supports convolutional and max-pooling layers. Once other common layers (ReLU, drop-out, fully-connected etc.) are implemented, the library will be open sourced for the benefit of the Julia's machine learning community.

References

[1] Nvidia digits library. http://devblogs.nvidia.com/parallelforall/easy-multi-gpu-deep-learning-digits-2.

[2] Yoshua Bengio. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2(9):1–127, 2009.

[3] Alessandro Giusti, Dan C. Ciresan, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Fast image scanning with deep max-pooling convolutional neural networks. *CoRR*, abs/1302.1700, 2013.

[4] 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 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[5] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CVPR (to appear)*, November 2015.