A FAST LEARNING ALGORITHM FOR IMAGE SEGMENTATION WITH MAX-POOLING CONVOLUTIONAL NETWORKS

Jonathan Masci[⋆∗] *Alessandro Giusti*[⋆] *Dan Ciresan*[⋆] *Gabriel Fricout*[†] *Jürgen Schmidhuber*[⋆]

[⋆] IDSIA – USI – SUPSI, Manno – Lugano, Switzerland
[†]ArcelorMittal, Maizières Research, Measurement and Control Dept., France

ABSTRACT

We present a fast algorithm for training MaxPooling Convolutional Networks to segment images. This type of network yields record-breaking performance in a variety of tasks, but is normally trained on a computationally expensive patch-by-patch basis. Our new method processes each training image *in a single pass*, which is vastly more efficient.

We validate the approach in different scenarios and report a 1500-fold speed–up. In an application to automated steel defect detection and segmentation, we obtain excellent performance with short training times.

***Index Terms*—** Segmentation, Convolutional Network, Detection, Industrial Application, Medical Imaging

1. INTRODUCTION

Image segmentation is a fundamental task for many applications ranging from medical imaging to industrial inspection systems. Many recent segmentation approaches build on supervised machine learning techniques, and rely on a training dataset with known ground truth segmentation.

A conventional supervised segmentation pipeline is typically based on two stages operating at the level of single pixels: a) *feature extraction*, where each pixel is projected into a richer representation by accounting for its context; and b) *classification*, where class probabilities for each pixel are computed. Once each pixel is classified, the resulting probability maps are post-processed (e.g. by enforcing smooth boundaries through filtering and thresholding, or by using techniques such as graph cuts [1] or level sets [2]). Finding the right set of features which minimizes segmentation error is a cumbersome task. The choice of features greatly affects segmentation quality.

Recent work [3, 4, 5, 6] follows a different approach, using convolutional neural networks (CNN) to segment images. Here feature extraction itself is learned from data and not enforced by designers. These approaches obtain state-of-the-art results in a very broad range of applications.

Amongst CNN variants, the MaxPooling Convolutional Network (MPCNN) has recently received a lot of attention. It obtained a long list of record-breaking results [3, 7, 8, 9]. MaxPooling layers appear fundamental for excellent performance, but their training requires to operate separately on all patches in the image. This requires a lot of computational power, a serious limitation for many industrial applications where large datasets are used and training has to be fast.

∗Jonathan Masci is supported by the *ArcelorMittal / New AIS* project. This work was partially supported by the *Supervised Deep / Recurrent Nets* SNF grant, Project Code 140399.

Contribution We propose an efficient MPCNN training algorithm operating on entire training images, avoiding redundant computations, making MPCNN easily applicable to huge training datasets. We validate it on the problem of steel defect detection, achieving excellent results in this important industrial application.

2. BACKGROUND

MaxPooling Convolutional Neural Networks (MPCNN) are hierarchical models alternating two basic operations, Convolution and MaxPooling. Their key feature is that they exploit the multi-dimensional structure of images via weight sharing, learning a set of convolutional filters. MPCNN scale well to large images and excel in many object recognition [9, 8, 10, 7] and segmentation [3, 6, 5] benchmarks. We refer to a state-of-the-art MPCNN as depicted in Figure 1. It consists of several basic building blocks briefly explained here:

Fig. 1. A schematic representation of an MPCNN. Raw input pixel values are processed by a number of interleaved Convolutional and MaxPooling layers, which are trained to extract meaningful features. Several Fully-Connected layers follow, which produce the final classification.

Convolutional Layer (C): performs a 2D filtering between input images and a bank of filters, producing another set of images denoted as maps. Fully connected input–output correspondences are adopted and maps are linearly combined. Then, a nonlinear activation function (e.g., tanh or logistic) is applied.
MaxPooling Layer (MP): down-samples the input images by a constant factor, keeping the maximum value for every non-overlapping subregion of size $p_{\text{row}} \times p_{\text{col}}$ in the images.
Fully Connected Layer (FC): this is the standard layer of a multilayer network. It performs a linear multiplication of the input vector by a weight matrix.

For a more detailed description and for the MPCNN back–propagation steps we refer the reader to the relevant literature [8, 7].
Image Segmentation with MPCNN Given a trained MPCNN, a straightforward approach for segmenting an unseen image requires

to evaluate the net on every patch contained in it. This results in many redundant computations, since different patches overlap. A recent optimized approach [11] efficiently forward-propagates the whole image (instead of a single patch) through the net, resulting in a teoretical speed-up of almost three orders of magnitude during testing.

Here we extend this approach to speed up network training. We define a novel neural network layer type called MaxPoolingFragment; then, we derive the back–propagation procedure and show that the new model learns orders of magnitude faster than patch-based approaches.

3. METHOD

3.1. Notation

The following notation is adopted. The set of training images is indicated by $\mathbf{X}$; the corresponding ground-truth annotations by $\mathbf{T}$ (thus mapping a class to each pixel); x_i and t_i refer to a particular training and testing image, respectively. The net is parametrized by Θ, the union of all parameters of all layers, and consists of a list of concatenated layers. The objective function of the minimization problem is denoted $L(\Theta; \mathbf{X}, \mathbf{T})$. A layer is a function mapping input storage to output storage.

Following earlier notation [11] we define such a storage as the set $\mathbf{F} = \{\cup_{i=1}^{N} f_i\}$, where each f_i represents a stack of maps, here denoted as *Fragments*. For example, the input layer will be a storage $\mathbf{F}^{\text{input}}$, with cardinality 1. $\mathbf{F}^{\text{input}}$ contains a single fragment, corresponding to the input image x_i. This architecture strongly differs from conventional MPCNN by the choice of storage structure. In standard models every storage is a container of a stack of maps. Instead, in our case the same data is necessarily split into a number of fragments.

3.2. The MaxPoolingFragment (MPF) layer

We can now introduce our MP layer extension, the **MaxPoolingFragment** (MPF). Given an input image x_i, a conventional $k \times k$ MP layer produces a smaller image, for which only a single value in each non-overlapping $k \times k$ neighborhood is kept.

When an MPCNN is applied with a sliding window though, forwarding every patch in the image causes redundant computations. While a convolutional layer can be applied directly to the entire input image to produce all results for all possible patches, an MP layer cannot. The forward pass of our MPF layer closely follows the detailed description by Giusti et al. [11]. With a MPF layer there will be k^2 different offsets in the input map, each one producing an output fragment. Thus, if the number of fragments in input is $|\mathbf{F}^{\text{in}}|$, we will have $|\mathbf{F}^{\text{in}}|k^2$ fragments in total. All redundant computations are removed.

From a software engineering perspective, an MPF layer can also be seen as a collection of MP layers, one for each generated fragment.

Consider a training image x_i and its ground truth t_i. By means of the newly-defined MPF layer, we can quickly forward-propagate the network on the whole image to generate an output image $\hat{x_i}$, in which every pixel is replaced by the output of the corresponding MPCNN applied to every patch in x_i. We can now compute the partial derivative of the loss function L w.r.t. $\hat{x_i}$; e.g. $\hat{x_i} - t_i$ when minimizing the mean squared error loss with linear outputs. This is the first step of the back–propagation algorithm.

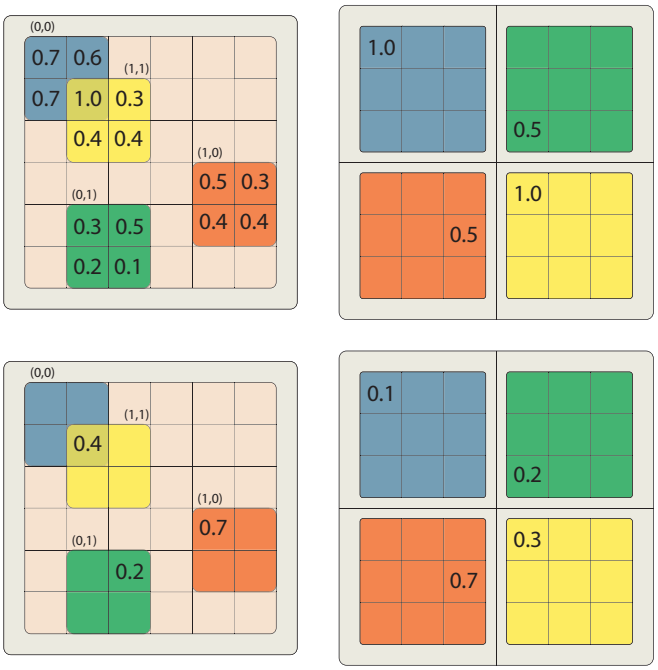

Fig. 2. MPF layer for the 2×2 pooling case. *Top*: Forward pass, *Fragments* (0,0) and (1,1) share the same maximal element; *Bottom* Back–propation pass where partial derivatives are pushed back to the previous layer in the hierarchy; the partial results of each *Fragment* are summed together.

Now we are ready to present our main contribution, namely, a procedure which is able to process these errors without considering each output pixel separately. This is made possible by an algorithm to compute the partial derivative of a MPF layer output w.r.t. its input, provided below.

3.3. Back–propagation through an MPF layer

As our framework uses fragments, we first have to redefine how a layer processes its input and computes its partial result of back–propagation. This generalizes MPCNN operations as explained in object-oriented Matlab pseudocode by Algorithms 1 and 2.

Data: Layer l, Input storage $\mathbf{F}^{\text{in}}$
Result: Output storage $\mathbf{F}^{\text{out}}$
for *i=1* **to** $\mathbf{F}^{in} \rightarrow$*nFragments()* **do**
  $\mathbf{F}^{\text{out}} = \mathbf{F}^{\text{out}} \cup l \rightarrow \text{fwd}(f_i^{\text{in}})$;
end

Algorithm 1: Pseudocode for the forward pass of a layer in our MPCNN framework operating on fragments. The backward pass is similarly derived. $\cup$ indicates the concatenation of two sets. An MPF produces a set of fragments.

This new neural network interface makes it much easier to derive and implement the backward pass for a MPF layer. Figure 2 gives an illustrative example of how a MPF layer works in the 2×2 pooling case. The output consists of 4 fragments, each containing as many maps as the input layer, indexed accordingly. We also exemplify

Data: Layer l, Input storage $\mathbf{F}^{\text{in}}$, Partial result of back–propagation $\mathbf{F}^{\delta}$
Result: g: $\frac{\partial \mathrm{L}(\Theta)}{\partial l \rightarrow params}$
for *i=1* **to** $\mathbf{F}^{\delta} \rightarrow$ *nFragments()* **do**
  g = g + $l \rightarrow$grad(f_i^{δ});
end

Algorithm 2: Gradient computation pseudocode of a generalized MPCNN layer used to operate in conjunction with a MPF layer. Gradients are accumulated as the same function is applied to every input fragment.

the case where the same element (pixel with value 1.0 in Figure 2–top) is the maximum for two different offsets of the pooling kernel (respectively $(0, 0)$ and $(1, 1)$), generating the corresponding output fragments (respectively blue and yellow). During back–propagation the partial results of differentiation, shown in Figure 2–bottom–right, are processed for every fragment and then summed together. The subsequent convolutional layer processes the 4 fragments through the interface of Algorithm 1, as shown in Figure 3, and computes the gradient by summing gradients of 4 fragments. Pseudocode for both operations is shown in Algorithms 3 and 4.

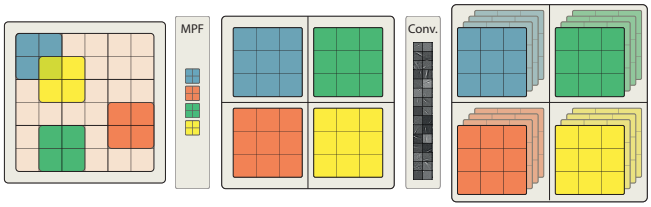

Fig. 3. Illustrative example of how subsequent convolutional layers of a MPF layer work. The same operation is applied to each output fragment; the gradient is the sum of the gradients of the fragments.

Data: Input storage $\mathbf{F}^{\text{in}}$, Pooling kernel P
Result: Output storage $\mathbf{F}^{\text{out}}$, MP indices $\mathbf{mpIDXS}$
$\mathbf{F}^{\text{out}} = \{\}$;
for *i=1* **to** $\mathbf{F}^{in} \rightarrow$*nFragments()* **do**
  for *j=1* **to** *size(P, 1)*size(P, 2)* **do**
    [r, c] = ind2sub(j, size(P)) ;
    [a, b] = MP_fwd(f_i^{in}(r:end, c:end), P) ;
    $\mathbf{F}^{\text{out}} = \mathbf{F}^{\text{out}} \cup a$;
    $\mathbf{mpIDXS} = \mathbf{mpIDXS} \cup b$;
  end
end

Algorithm 3: Forward pass of an MPF layer. This algorithm produces a set of fragments for each input fragment, one for each offset in the pooling kernel. The output is their union, as shown in Figure 2. MP(f^{in}(r:end, c:end), P) indicates the usual MPCNN downsampling operation applied to the fragment f^{in}. Within the input maps, **mpIDXS** stores the index of every maximum value produced by the pooling operation; it is used to back–propagate sub–gradients.

Data: Input storage $\mathbf{F}^{\text{in}}$, Result of fwd $\mathbf{F}^{\text{out}}$, P, $\mathbf{mpIDXS}$
Result: Output storage $\mathbf{F}^{\delta}$
for *i=1* **to** $\mathbf{F}^{out} \rightarrow$*nFragments()* **do**
  for *j=1* **to** *size(P, 1)*size(P, 2)* **do**
    s = findSourceFragment(i,j) ;
    $f_s^{\delta} = f_s^{\delta}$ + MP_bkp(f_s^{δ}, $mpIDXS_s$) ;
  end
end

Algorithm 4: Back–propagation pass of a MPF layer. The algorithm produces a set of fragments equal to the one of the input layer $\mathbf{F}^{\text{in}}$ used in Algorithm 3. MP_bkp places the partial result of the chain rule of derivatives in the position indicated in $mpIDXS_s$. As each element might have contributed multiple times as shown in Figure 2–bottom, they need to be accumulated in the output layer $\mathbf{F}^{\delta}$.

4. RESULTS

We validate the proposed approach on two different applications, namely membrane segmentation (Section 4.1) and steel defect detection (Section 4.2).

In both applications, networks are trained to minimize the multi-class cross-entropy loss (MCCE) – a commonly-used error function for classification tasks. This error is computed by considering each pixel independently. Full connectivity is used for the convolutional layers. With C $7 \times 7 \times 8$ we indicate a layer with 8 output maps and filters of size 7×7.

Our framework is implemented on CPU with Matlab and uses Intel Performance Primitives to perform convolutions through a Mex–function.

4.1. Membrane Segmentation

We use the public dataset of the ISBI 2102 Electron Microscopy Segmentation challenge [12]. It consists of a volume of 30 gray level images of size 512×512 pixels.

As in previous work [3] we exploit the rotational invariance of the problem and synthesize additional training images by arbitrarily rotating and flipping the given training images. Also, pixels outside of the boundary of testing images are synthesized by mirroring – which allows to preserve the size of the output. We consider the network architectures N3 and N4 of Ciresan et al. [3], which contributed to the top-scoring entry in the challenge.

We train our system using stochastic gradient descent safeguarded by the Armijo rule, updating the weights after every image has been presented to the net. Convergence is reached generally after 100 epochs, when the network has seen 3000 different images, the equivalent of roughly 390 million patches.

Figure 4 shows a segmentation example for a test slice. Table 1 compares our training times with the training times of the highly-optimized GPU patch-based approach [3]: although our implementation runs on the CPU in the Matlab environment, it yields a huge speed–up. On the other hand, the segmentation performance of the resulting network exhibits negligible differences: 6.8% vs 6.6% pixel error rates for N3, respectively, for our method and the one of Ciresan et al. [3]. Errors are evaluated directly on the competition server.

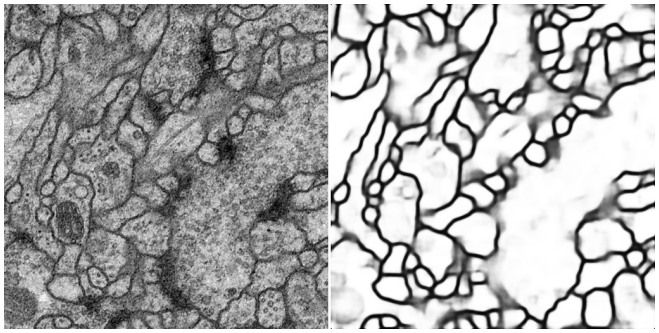

Fig. 4. A slice of the test set segmented using network N3 trained by our novel approach.

Table 1. Comparison of training times for the Membrane. The overhead for generating the transformed samples is also included in the overall computation. The relative speed–up of our method is shown in parenthesis.

	Patch (GPU) [3] patches/s	Image (CPU, Matlab), ours patches/s
N3	260	4500 (17× speed–up)
N4	130	3000 (23× speed–up)

4.2. Steel Defect Detection

We use a proprietary dataset from ArcelorMittal, consisting of 534 images, each with resolution 550×240. Images are acquired with a matrix camera directly from a production plant. Illumination is highly variable, and many images are severely under- or over-exposed, which hinders naïve processing techniques. 70 of such samples contain a defect which covers part of each image. This type of defect is very difficult to detect due to its variable and subtle appearance (see Figure 5). A ground-truth segmentation of the area (if any) containing the defect of each image is given. We use 50% randomly–sampled images for training, 25% for validation, and the rest for testing.

We consider the problem of segmenting the defect (if any) in each testing image. Note that this is a harder problem than simply detecting whether there is a defect somewhere in the image – which can be solved using a threshold on the number of pixels classified as defect.

The network operates on a 31×31 window and has the following structure: C $7 \times 7 \times 8$, MPF 2×2, C $5 \times 5 \times 8$, MPF 2×2, C $5 \times 5 \times 8$, FC 100, FC 2. We use LBFGS, which delivered the best performance. We also down–sample the images by a factor of 4 to further speed up learning. We minimize the MCCE loss function per class because of the unbalanced dataset. There are in fact only very few pixels which correspond to the defect, therefore learning is prone to naïve convergence to solutions which always favor predicting the background.

Every training epoch takes on average 44s (also accounting for the overhead due to LBFGS optimization). This amounts to 0.16s per image on a i7–2600 quad-core machine, where the whole system is trained in two hours. Because an image contains roughly 8.2K patches, our system is effectively processing 50K patches per second. Training on a patch-by-patch fashion is significantly slower even when highly optimized and implemented on GPU processors.

Segmenting a new image requires 0.07s, which makes online realtime implementation feasible for practical deployment within routine steel production activities. In order to assess the segmentation performance of our model, we sample 5K positive and 5K negative pixels from the test set. This produces an unbiased evaluation to measure the per–pixel error rate. Random guessing reaches only 50% error, while our MPCNN obtains 5.8%. Figure 5 shows a typical segmentation result for a test set image.

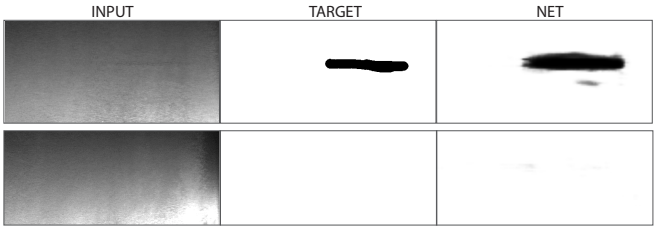

Fig. 5. A typical steel defect example. Segmentation is almost perfect, illustrating the power of the proposed approach for industrial applications.

We design our detection pipeline as follows. After learning an MPCNN on the training set, we determine on the validation set the threshold yielding the best detection performance.

A given image is flagged as containing a defect if the number of "defect" pixels it contains exceeds a given threshold. The threshold is set to 5000 (i.e. half the area of the smallest conceivable area for a defect) and is not critical.

Table 2 shows detection performance. Our method makes only 3 mistakes and correctly detects 3 additional defects mislabeled during annotation.

Table 2. Detection error results of our efficient learning framework for MPCNN. Test evaluation times for a given image are also reported along with the patch based evaluation with equal implementation (e.g. Matlab).

Test err %	Patch (CPU)	Image (CPU)
2.3	110s	0.07s (1500x speed–up)

5. CONCLUSIONS

We introduced a fast and efficient algorithm to train MPCNN for image segmentation. The network is able to process the whole image at once without having to consider separate patches. This greatly speeds up training in comparison to approaches that process each patch independently – including those optimized on GPU. No significant loss in accuracy is observed. Now we can train huge network architectures on large datasets within manageable timeframes.

We achieve state-of-the-art performance on the challenging membrane dataset used for the ISBI EM segmentation challenge. In an application to automatic steel inspection we reach realtime processing speed, achieving a speed–up factor of 1500 w.r.t. the corresponding patch-based method.

6. REFERENCES

[1] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 23, no. 11, pp. 1222–1239, 2001.

[2] A. Tsai, A. Yezzi Jr, W. Wells, C. Tempany, D. Tucker, A. Fan, W.E. Grimson, and A. Willsky, "A shape-based approach to the segmentation of medical imagery using level sets," *Medical Imaging, IEEE Transactions on*, vol. 22, no. 2, pp. 137–154, 2003.

[3] Dan C. Cireşan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber, "Deep neural networks segment neuronal membranes in electron microscopy images," in *NIPS*, 2012.

[4] Clement Farabet, Camille Couprie, Laurent Najman, and Yann LeCun, "Learning hierarchical features for scene labeling," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2013, in press.

[5] Srinivas C. Turaga, Joseph F. Murray, Viren Jain, Fabian Roth, Moritz Helmstaedter, Kevin Briggman, Winfried Denk, and H. Sebastian Seung, "Convolutional networks can learn to generate affinity graphs for image segmentation," *Neural Comput.*, vol. 22, no. 2, pp. 511–538, Feb. 2010.

[6] Srinivas Turaga, Kevin Briggman, Moritz Helmstaedter, Winfried Denk, and Sebastian Seung, "Maximin affinity learning of image segmentation," in *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, and A. Culotta, Eds., pp. 1865–1873. 2009.

[7] Jonathan Masci, Ueli Meier, Dan C. Cireşan, Fricout Gabriel, and Jürgen Schmidhuber, "Steel defect classification with max-pooling convolutional neural networks," in *International Joint Conference on Neural Networks*, 2012.

[8] Dan C. Cireşan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber, "Flexible, high performance convolutional neural networks for image classification," in *International Joint Conference on Artificial Intelligence (IJCAI2011)*, 2011.

[9] Dan C. Cireşan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber, "A committee of neural networks for traffic sign classification," in *International Joint Conference on Neural Networks (IJCNN2011)*, 2011.

[10] Dan C. Cireşan, Ueli Meier, Luca M. Gambardella, and Jürgen Schmidhuber, "Convolutional neural network committees for handwritten character classification," in *ICDAR*, 2011, pp. 1250–1254.

[11] Alessandro Giusti, Dan C. Cireşan, Jonathan Masci, Luca M. Gambardella, and Jürgen Schmidhuber, "Fast scanning with deep neural networks," Tech. Rep. IDSIA-01-13, Istituto Dalle Molle di Studi sull'Intelligenza Artificiale (IDSIA), 2013.

[12] Albert Cardona, Stephan Saalfeld, Stephan Preibisch, Benjamin Schmid, Anchi Cheng, Jim Pulokas, Pavel Tomancak, and Volker Hartenstein, "An integrated micro- and macroarchitectural analysis of the drosophila brain by computer-assisted serial section electron microscopy," *PLoS Biol*, vol. 8, no. 10, pp. e1000502, 10 2010.