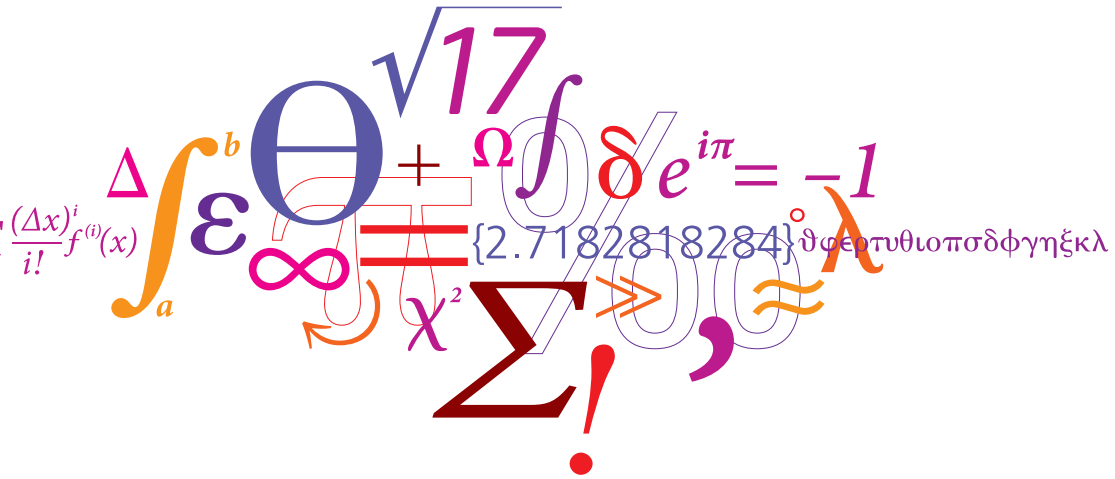
DTU Technical University of Denmark

Fast convolutional deep learning for image segmentation



Author
Lasse Seligmann Reedtz

Supervisor
Ole Winther
PhD, Associate Professor

Supervisor
Anders Boesen Lindbo Larsen
PhD student

b

Technical University of Denmark
DTU Compute
Department of Applied Mathematics and Computer Science
Building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk
IMM-M.Sc.-2015

Abstract

Recent work has shown potential for reducing total computational time when training a convolutional deep neural network for image segmentation, by processing each training image in a single pass rather than patch by patch, as demonstrated by Masci et al. [38].

This thesis will analyze and review the existing set of computing resources for and literature on deep learning to come up with the best possible solution for implementing deep convolutional neural networks for image segmentation on CPUs and GPUs. It will review the theory behind convolutional deep learning architecture, with special emphasis on its potential for speed. This thesis will also cover implementation of said convolutional deep learning for image segmentation on CPUs and GPUs, at such a quality that it can be distributed through github.

The proposed implementation will be accomplished using the frameworks CUDArray and Deeppy, which together encompass a Python library, which has been developed in collaboration with Anders Boesen Lindbo Larsen, during this thesis period. The proposed implementation will be tested with respect to speed and predictive performance on the ISBI 2012 Electron Microscopy Segmentation Challenge [8]. Final testing demonstrated relatively short processing time without significant loss of performance on said challenge.

Preface

This thesis was prepared for DTU Compute, at the Technical University of Denmark in the period from October 2014 to February 2015. It corresponds to 32.5 ECTS credits, in fulfillment of the requirements for acquiring a M.Sc. degree in engineering.

I would like to thank my supervisor Ole Winther for guidance while choosing this topic and for insightful DL theory discussions. As well as Anders Boesen Lindbo Larsen, PhD student, for coming up with CUDArray and Deeppy and allowing me to join in the fun, as well as for great technical discussions, and encouragement via skype chats from Canada. I would also like to extend deep appreciation to Jessica Svet for many hours of editing, which saved Ole and Anders from my numerous spelling errors.

Contents

Chapter 1

Introduction

As machine vision approaches, and eventually exceeds, the accuracy and speed of human performance, we may begin to see more clearly the benefits that artificial intelligence can offer us in our day-to-day lives. Particularly with regards to medical imaging, machine vision can assist with tasks ranging from mitosis detection in breast cancer histopathology images [13] to neural structure segmentation in electron microscopy images[10], pictured below in Fig. 1.1.

The applications of machine vision are wide ranging, including handwritten digit and character identification, image classification, object detection and even galaxy photo classification. At a time when we are collecting more data to sort through, than we have man hours to sort through it, machine vision is one way in which we can use artificial intelligence in a wide variety of situations to act as a decision making aid and to save time in our day to day lives.

Machine Vision and Image Segmentation

From birth we begin to train our visual system to understand the world around us, recognizing our parents faces from differing angles, identifying and classifying every new 'image' which is introduced to us. Influenced originally by the structure of a cat's visual cortex [26], and later a monkey's [27], deep hierarchical neural models were developed in the late 70's in Tokyo[21].

The first model, named the Neocognitron, was a self organizing neural network(NN) model which acquired the ability to recognize stimulus patterns based on the Gestalt principles of geometrical similarity, without being affected by the shape's position nor by small shape distortions. NN's, also known as multi-layer perceptons, got a break through in 1989, when LeCun incorporated backpropagation [35].

Over the years NN's became deeper and wider and were therefore called Deep Neural Networks(DNNs). While neural networks fell out of popularity slightly in the 90's, today within the machine learning community, DNN are again a hot topic. This is largely due to the fact that in recent years DNNs, and variations of DNNs such as Deep Convolutional Neural networks (CNNs), have become more feasible by utilizing graphic cards (GPU), and have been shown to outperform other methods in a wide array of applications and on varying data sets. These range from classifying handwritten characters in the Mnist data set [22], to localising, detecting and classifying different objects in the ImageNet challenge [33].

One task within machine vision is image classification and segmentation. Image segmentation often requires classification for each pixel in an image, for instance, in a case where one wants to identify cell walls in a image in the ISBI 2012 Electron Microscopy Segmentation challenge. These are often built with supervised machine learning techniques, like CNNs or MaxPooling CNNs (MPCNNs). Throughout this thesis, the focus will be on the use of CNNs for image segmentation and classification.

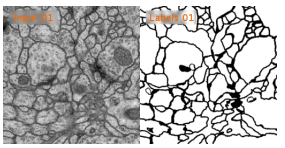


Figure 1.1: ISBI 2012 Electron Microscopy Segmentation Challenge samples [10]

1.1 Thesis Outline

This thesis has two major emphases, the first of which is a review of the theory and architecture of CNNs, including a hypothesis for a fast MPCNN using Fragmented MaxPooling for Image Segmentation tasks, an approach originally introduced by Masci et al.[38]. The second is the Implementation and Testing of such a network within a newly developed DNN python library, CUDArray and

Deepy, applied to the ISBI 2012 Electron Microscopy Segmentation Challenge [4].

Following the introduction, which includes a concise review of the existing set of computing resources for and literature on deep learning, this thesis briefly discusses in Chapter 2 the traditional and state-of-the-art architectures of CNNs used for pixel level classification tasks, including the latest Fragmented Max-Pooling approach.

Next, in Chapter 3, 'Convolutional Deep Learning Architecture,' you will find an overview of Network Layers and of the Training of a CNN using backpropagation. The 'Network Layers' and 'Training' sections include, among other topics, discussion of the most popular methods as well as the newest, potentially not yet widely adopted methods, within the Convolution layers, Max Pooling Layers and Fully Connected and Dropout layers.

Chapter 4, 'Implementation', will cover just that. The CUDArray and Deeppy libraries will be described and the MPFCNN module implementation for GPUs in Deeppy will be discussed in greater detail. Chapter 5 'Testing and Results' uses CUDArray and Deeppy frameworks on the ISBI 2012 Electron Microscopy Segmentation Challenge, reviewing speed and predictive performance. Based on final testing, this thesis will finish with a Conclusion in Chapter 6 showcasing a relatively short processing time without significant loss of performance, considering the lack of post-processing, on said challenge using a MPFCNN with a GPU implementation.

1.2 What is a Convolutional Neural Network?

A Back Propagated DCNN, first proposed by LeCunn in 1989 [35] is a hierarchical neural network, comprised mainly of three intermediate 'hidden' layer types; convolution, pooling/subsampling, and fully connected. The network starts with an input layer, is followed by the various intermediate 'hidden' layers, and finishes with an output layer. Having more hidden layers, rather than fewer, has been shown to improve the precision of the network [14]. The output layer offers a classified, transformed version of the input data. A basic CNN can be seen in Fig. 1.2.

The 'hidden' layers consist first of alternating convolution and pooling/subsampling layers, which reflect the simple and complex cells in the mammalian visual cortex. The convolution layer extracts localized features from the input image, utilizing filters, or kernels. The results are subsampled by a pooling layer,

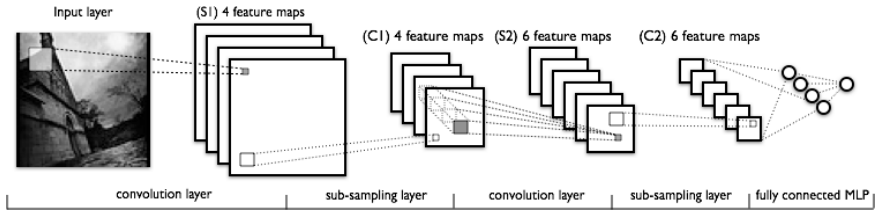


Figure 1.2: Graphical depiction of a LeNet model, from `deeplearning.net`

and these subsequent results are then passed to another convolution layer to be re-filtered.

This process continues, passing results deeper into the network, to multiple pooling and convolution layers, the total number of which is set by the network architect, resulting in a deep feed-forward convolutional network architecture. While many layer decisions are made by the network architect, feature extraction within the network is learned from data and not enforced by designers.

The feature vector results from the final pooling layer are passed to a flattening layer and then through a series of fully connected layers, finally ending with a Multinomial Logistic Regression(MLR) Output layer. It should be stated that the order, type and number of layers in a network architecture are defined by the network architect.

There are a wide variety of layer types that have not been mentioned in this thesis, but that may prove useful when applied to novel cases. This thesis will mention use of the following layer types: input, Convolution, MaxPooling, MaxPoolingFragment [38], Flattening, Indexing and Flattening, Fully Connected, Dropout [47], and Multinomial Logistic Regression.

Pierre Sermanet et al.[45] stated that 'the main advantage of ConvNets for many (recognition, localization and detection) tasks is that the entire system is trained end to end, from raw pixels to ultimate categories, thereby alleviating the requirement to manually design a suitable feature extractor. The main disadvantage is their ravenous appetite for labeled training samples.'

In reference to the above mentioned disadvantage, image transformation, invariance and rotation can be used to generate more training samples from a originally small set of training data. Fortunately, this is not a major disadvantage when performing image segmentation, which is a pixel level classification task. In this case, each pixel is a sample. For example, an image of size 200x200 provides 40,000 pixel samples, often meeting the CNN's need for large training sets.

1.3 Convolutional Neural Networks Over the Years

Between 1989 and 2009 progress with CNN was slow, but notable. Limitations were mainly due to hardware. We saw the introduction of MaxPooling layers in 1992 [55] and in 1998, LeCun improved his DNN using the MNIST handwritten digit dataset [36]. In 2003, neural networks were successfully applied to image interpretation [7] and digit recognition using only supervised pre-training [46].

Deep structure based approaches set records in natural language processing in 2008 [17], natural images (CIFAR 10) in 2009 [32], and Chinese characters (CASIA) in 2010 [37]. In 2009 a 3-dimensional CNN was combined with SVMs to detect human actions in surveillance videos, winning three 2009 TRECVID competitions [56].

In 2010, GPU usage began changing the game. We saw a NN using plain back-propagation with distortions break the MNIST error record simply by utilizing the GPU [15], establishing new state of the art results. Shortly after, in 2011, we saw the same team achieve super human vision performance results with the first implementation of GPU-based MPCNN on the Nvidia CUDA parallel computing platform [19] making the network both wide and deep [14].

Since 2011, records utilizing DNNs and DCNNs have been set and broken year after year. Researchers achieved high scores in 2011 in handwritten characters [16], and in natural language processing [18], as well as in speech recognition at Microsoft in 2013 [20].

More relevant to this thesis, have been the projects which utilize deep structures for solving real world applications using machine vision. Examples include classifying traffic signs in 2011 [48] using a MPCNN [11], image classification in 2012 [12] and 2014 [5] [42], improved accuracy on a subset of ImageNet in 2012 [33], object detection in 2013 [45], and facial recognition and verification in 2014 [50] [49].

A CNN won it's first image segmentation challenge in 2012 [10] and leading up to 2015 the best performing algorithms for many vision tasks, like human pose estimation [28] [52] [54] and steel defect detection [38] [39], have been based on deep convolutional neural networks [5] [9] [45] [30] [13]. Some of the methods which have been shown to improve DNNs and CNNs, have been drop-out [47], max-pooling [44], rectified linear units [33] and Local Contrast Normalization [29]. Most of these ideas were not new, rather they were newly applied to NNs from other fields.

Neural Networks and Libraries

Because DNNs continue to be an active research field, researchers need access to good tools and frameworks to effectively and efficiently test new methods. One such tool is a NN library that incorporates the latest state of the art methods, and which can be easily modified for testing even newer ideas. Fortunately, there are many high-performing, open source libraries that incorporate many of the state of the art methods available to academics.

Looking at the library landscape today, we can find a handful of very good libraries, like Theano[6] or Caffe[30], but they tend to focus heavily on either speed, like Torch[3] and ConvNet2[1], or usability like pybrain[43]. There are very few which strike a balance. Furthermore, many of the more mature libraries are not written in Python, which could soon be the most used language by academics, and is already the most used language by universities when introducing computer science students to programming.[23]

Theano is the exception, but it includes a complex optimization function, which re-compiles your code, making it quite difficult and time consuming to debug while developing a new library modification. As mentioned, this thesis includes contributions to a new open source library, CUDArray [34] and Deeppy[2], and will be covered in more detail in Chapter 4 'Implementation'.

Chapter 2

Hypothesis: Faster MPCNN with Fragmented MaxPooling for Image Segmentation

As described in Chapter 1, a traditional DCNN for pixel level classification often uses a patch based input layer to preprocess an image, before forward propagating the data through each layer of the network. Individual pixel classification requires the network to evaluate each pixel in question, while also using the data from the surrounding pixels. The patch approach extracts a small square 'patch' of pixels from the image, with the pixel to be evaluated located in the center of said patch.

The output of the input layer for a 512×512 pixel image is 512×512 patches, which are then fed to the network one at a time. Because a patch is extracted for every pixel, one pixel's patch will overlap the patch of it's neighboring pixel, both containing data from many of the same pixels, as shown in Fig. 2.1. Fortunately, the latest research by Masci et al.[38] provides solutions for these redundancies by utilizing 'fragments' instead of patches as a part of the new MaxPoolingFragment CNN (MPFCNN).

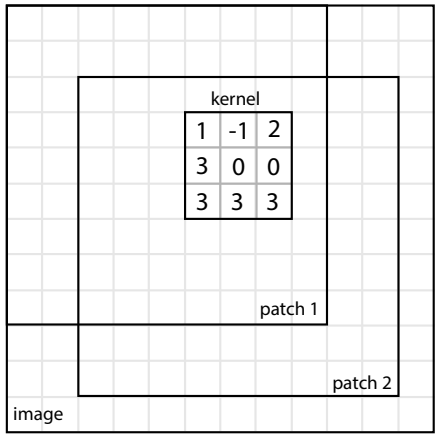


Figure 2.1: Example of duplicated convolution calculations for two patches

The most computationally expensive layer of the network is by far the convolution layer. The time complexity for segmenting an image can be found for the two networks in order to get an idea of the potential speedup. For the sake of simplicity, we assume that all calculations are done consecutively.

For the patch method, the following notation can be used to describe the time complexity for each convolution layer, assuming a square image, patch and kernel.

$$O(s^2 \times |P_{(l-1)}| \times |P_{(l)}| \times w_l^2 \times k_l^2) \tag{2.1}$$

where s is the image size in the original image, $|P_{(l)}|$ is the number of feature maps for l, w is the size of the feature map and k is the size of the kernel.

For the fragment method, the complexity can be calculated as follows:

$$O(fs^2 \times |P_{(l-1)}| \times |P_{(l)}| \times |F_l| \times k_l^2) \tag{2.2}$$

where fs is the fragment size, $|P_{(l)}|$ is the number of feature maps for l, $|F_l|$ is the number of fragments in layer l and k is the size of the kernel.

Using these complexities, we can calculate a potential speedup for segmenting an image. Using the network from Table. 5.1 an image size of 512 and a patch size of 31, it can be seen that the patch method requires 68.4 times more computations as seen in Table 2.1.

l	s	fs	$\lvert P_{l-1}\rvert$	$\lvert P_l\rvert$	w_l	$\lvert F_l\rvert$	k	$P \cdot 10^9$	$F \cdot 10^9$	speedup
1	512	512	1	48	28	1	4	157.8	0.2	784.0
3	512	256	48	48	10	4	5	1509.9	15.1	100.0
5	512	128	48	48	2	16	4	38.7	9.7	4.0
Total								1706.4	25.0	68.4

Table 2.1: Estimated time complexity P for the patch method and F for the fragment method and the relative speedup using the fragment method. Using the network described in Table. 5.1 an image size of 512 and a patch size of 31.

Today, using a MPFCNN, the entire image can be fed to the network in a single pass, without passing individual overlapping patches, or any patches at all, eliminating the redundant calculations, and greatly decreasing network training time. For the full image, all convolutions only have to be calculated once per convolution layer. An example of a MPFCNN can be seen in Fig.2.2 Each layer builds on the ideas proposed by Masci et al.[38].

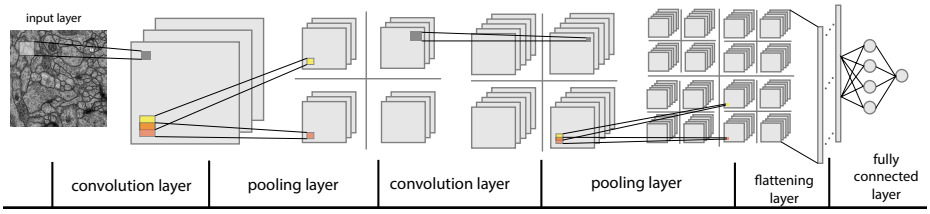


Figure 2.2: Example of a MaxPoolingFragmented Convolutional Neural Network (MPFCNN)

The MPFCNN method, introduced in 2013, has already been used to locate joints of a person in an image [53], and to reduce the loss of resolution in one instance of the ImageNet competition [45]. The team that created the MaxPoolingFragment method has already applied said method to steel defect detection [22], to neural structure segmentation in electron microscopy images[38] and to mitosis detection in breast cancer histopathology images [13], achieving state of the art performance.

Unfortunately for many academics, the new method has not, as far as I can confirm at the time of this thesis, been incorporated into any pythonic neural network libraries.

Hypothesis

Hypothesis: According to existing DL theory and CNN architecture, the best possible solution for implementing deep convolutional neural networks for fast image segmentation on CPUs and GPUs without significant loss of accuracy employs the use of Masci's Fragmented approach on a GPU-implemented Max-Pooling Convolutional Neural Network, including Supervised training through backpropagation using Stochastic Gradient Decent.

In order to confirm this hypothesis, said network will be implemented as a fully parameterizable CNN within a DNN Python library for use on CPUs and GPUs, at such a quality that it can be distributed through github. The proposed implementation will be tested with respect to speed and predictive performance using the ISBI 2012 Electron Microscopy Segmentation Challenge data.

Chapter 3

Convolutional Deep Learning Architecture

In this chapter we will describe in more depth many of the layers needed to construct and train a CNN, specifically a MPFCNN, highlighting the differences between a traditional MPCNN using patches, and the newer 'Fragmented Approach'to MPFCNN, without the use of patches.

We start with 'Network Layers' and finish with 'Training'. You can expect to find theoretical mathematical formulas throughout this chapter, and notation will be defined as needed. Note that post-processing will not be discussed in this thesis, as it does not relate as directly to the use of a CNN for Image segmentation, as to the process of finding the specific solution to a specific problem.

3.1 Network Layers

3.1.1 A Word on Forward Propagation

Forward propagation of a network is the process of input data moving in one direction through a network. This is in contrast to backpropagation, which occurs when an error is processed through the network in the exact opposite direction, subsequently training and updating each layer.

Forward propagation is used in the training pass with known test data, as well as after the network is trained, for classifying new unknown test data. In this section, 'Network Layers', we will discuss forward propagation and the general behavior of each layer. backpropagation will be discussed in section 3.2, titled 'Training'.

3.1.2 Input Layer

The input layer is not truly considered a part of the network and does not 'learn', or rather it is not trained, it merely generates an abstraction of the input data so that the network can work with it. The input layer, a visible layer, and not a part of the hidden layers, supplies the input data to be given to the hidden layers of the network, and has no wieght associated with it.

This PreProcessing could consist of normalizing the data, or converting it to the right format. In the traditional patch method, already discussed in Chapter 2, the input layer would extract patches from the input image, such that the image could be fed to the network as a series of patches.

As previously mentioned, patches are not necessary in the Fragmented Approach, as the entire image is propagated through the MPFCNN in a single pass. When we are dealing with the new fragmented approach, the input for the Convolution layer changes from a single set of feature maps x, to a single set of Fragments $F_x = \{\bigcup_{i=1}^{N} f_i\}$. Here each fragment f_i contains a set of feature maps. The input to the network will thus have a F_y with cardinality of 1, containing one fragment f_0 corresponding to the input images feature maps.

3.1.3 Convolution Layer

Traditionally, a CNN contains multiple convolution layers, alternated between multiple Pooling layers. A convolution layer consists of a set of kernels (or filters) which are used to convolve the input image x, extracting a variety of features from the input image, generating one feature map for each kernel. After the feature maps have been created, an activation function is used to introduce non-linearity to the pre-activated feature maps y, giving an output a. When creating a convolution layer, one can modify the number and size of kernels, the size of the stride, the size of the window and the presence or absence of padding.

Discrete Convolution

The discrete convolution of an input image x with a kernel k of size $r \times s$ will yield an pre-activation output y. Each position in the output y is defined as

$$y_{ij} = (x * k)_{ij} = \sum_{pq} x_{i+p,j+q} k_{r-p,s-q} \tag{3.1}$$

Where p runs from $0 \ldots r$ and q from $0 \ldots s$. $*$ denotes the convolution operation.

To conceptualize this we can look at Fig. 3.1. From the formula, we can see that the kernel is read from the lower right corner. This is the same as taking the kernel, flipping the rows and columns $\widetilde{k}$ and placing it on top of x.

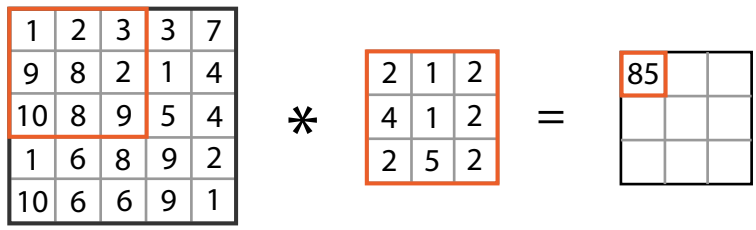


Figure 3.1: convolution on one input image with one kernel, using stride (1,1)

To calculate y, the flipped kernel $\widetilde{k}$ is placed on top of x in every possible position. The dot product of $\widetilde{k}$ and the underlying matrix is calculated and becomes the new value in y. An example of the output from the first row and first column (0,0) in 3.1 can be seen below.

$$
\begin{aligned}
y_{0,0} &= (x * k)_{0,0} \\
&= x_{0,0}k_{2,2} + x_{0,1}k_{2,1} + x_{0,2}k_{2,0} + x_{1,0}k_{1,2} + x_{1,1}k_{1,1} \\
&+ x_{1,2}k_{1,0} + x_{2,0}k_{0,2} + x_{2,1}k_{0,1} + x_{2,2}k_{0,0} \\
&= 1 \times 2 + 2 \times 5 + 3 \times 2 + 4 \times 2 + 5 \times 1 + 2 \times 4 + 10 \times 2 + 8 \times 1 + 9 \times 2 \\
&= 85
\end{aligned}
$$

One important thing to note is that the kernel k and the weight W of the layer are not the same. The relation between the kernel and the weights are defined as $k = \widetilde{W}$. Meaning, the kernel equals the weight matrix, where rows and columns have been flipped.

Convolution Layer Padding

Adding padding to an image, including the edge cases, can in some situations improve the error rate of the network, as shown in [31]. A padding of size $k_{rs}/2$ can be applied to the input image x. This allows the convolved image to have the same dimensions as the input image. Normal or typical padding is either mirror padding, where the image's pixel values are mirrored, or zero padding, as shown in Fig. 3.2.

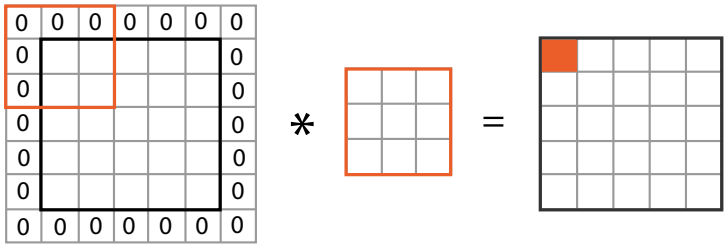


Figure 3.2: convolution of one input image, using zero padding, with one kernel, stride (1,1)

Convolution Layer Stride

Another flexible aspect of the Convolution Layer is the stride of the kernel. Modifying stride makes it possible to convolute only every `n` pixel. An example

of this can be seen in Fig. 3.3. Here a 5×5 image is convolved using a 3×3 kernel using a stride of 2, both vertical and horizontal, resulting in a 2×2 output image.

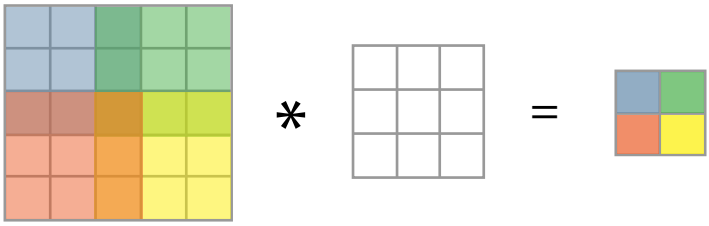


Figure 3.3: convolution of one input image with one kernel,with a stride of (2,2)

Convolution Layer Feature Maps

A grey-scale input image will initially consists of one feature map, where as a color input image will initially consist of three feature maps, one each corresponding to red, green and blue.

The feature maps which are generated by the Convolution Layer identify novel image features, potentially edges, curves or lines, as well as features which are less recognizable to the human eye. The total number of feature maps a layer produces are defined for each layer by the total number of kernels for that given layer.

When there is more than one feature map in layer $l-1$, the kernels in layer l will be 3 dimensional. For each feature map in the input x from layer $l-1$, and each feature map in layer l, there will be a weight matrix W_{ij}. In Fig. 3.4 this is illustrated graphically. When calculating the pre-activation output y_j in layer l, the output will be the sum of convolutions of the feature maps in x.

Convolution Layer Activation Function

As mentioned, the convolution layer contains and ends with an activation function. Traditionally, a tanh or sigmoid function was used. However, recent work

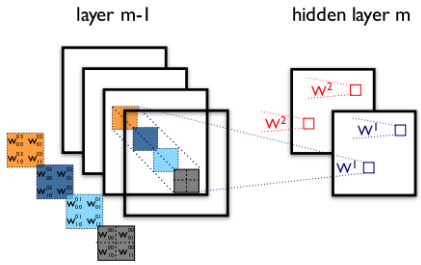


Figure 3.4: convolutions of 4 feature maps using 2 kernels. Image from `deeplearning.net/tutorial/lenet.html`

has shown that rectified linear unit (ReLU functions require less processing power and train the network faster [33].

A plot of the 3 graphs can be seen in Fig. 3.5. The activation function $g(x)$ is applied to each value in the pre-activation output y to introduce non-linearity. The output of convolution layer l is thus $a^l = g(y^l)$.

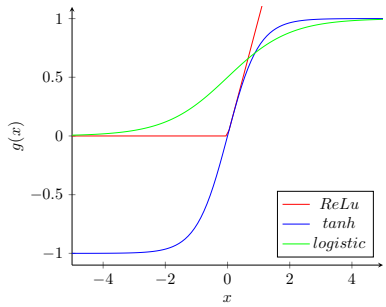


Figure 3.5: Plot of 3 commonly used activation function

Convolution Layer Fragmented Approach

In the Convolution Layer, the total number of fragments coming into the Convolution Layer is the same as the total number of fragments going out. The Convolution layer does not alter the total number of fragments, ensuring that that $|F_x| = |F_y|$. The sets of feature maps f_i in F_x are not necessarily the same size as the sets in F_y this solely depends on the number of kernels in the layer.

The images in one fragment are not affected by images in other fragments. Each fragment f_i in F_x can thus be convoluted in the same manner as the standard method described above. The pseudo code below shows an approach to calculating F_y where the method `convolv` is the convolution method described above.

```
def fragment_convolv(F_x, W):
    F_y = [ ]
    for f_i in F_x:
        y_j = convolv(x = f_i, W = W,
                      stride = 1, padding = true)
        f_j = activate(y_j)
        F_y.append(f_j)
    return F_y
```

When segmenting the image by classifying each pixel, the information of each pixel has to be retained. During convolution, the stride must therefore be set to one pixel in each direction. Furthermore, the image must be padded in order to retain the pixels from the edge cases of the input image.

3.1.4 Max-Pooling or MaxPoolingFragment Layer

The pooling, or sub-sampling, layer introduces translation invariance to the network, which improves generalization [25]. It does this by 'pooling' together, or sub-sampling, a set amount of pixels into one value. The number of pixels sub-sampled is defined by the pooling window size. The window is moved over the image in strides, where we distinguish between $stride_h$ and $stride_v$ according to which direction the window is being moved. Often the stride will equal the window size, resulting in a reduced image size.

Max-Pooling and Mean-Pooling

There are a variety of pooling techniques, but the two most common techniques are mean-pooling and max-pooling. In relation to mean-pooling, the pooled value is the mean of all values within the pooling window. In relation to max-pooling, the pooled value is the maximal value of all the values in the pooling window. Max-pooling has been shown to have a positive effect on network performance [44].

In Fig 3.6, examples of max-pooling and mean-pooling can be seen. The input given is 2 feature maps of size 4x4, using either max- or mean-pooling, with a

window size of 2x2 and a stride of 2 in both directions. The pooling is performed on each feature map giving an output of 2 feature maps of size 2x2.

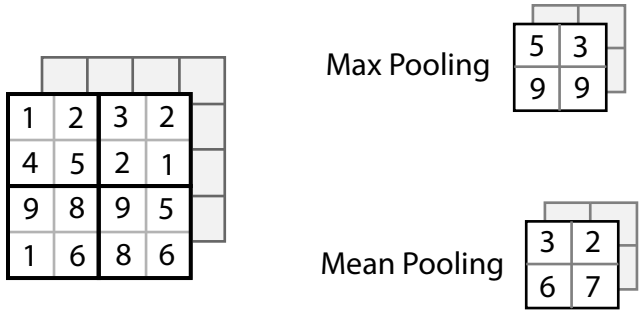


Figure 3.6: From top, Max and Mean pooling using a 2x2 window and stride (2,2)

The output of mean- and max-pooling can be described as:

$$max - pool : y_{ijk} = \max_{pq} x_{i,(j\times sv)+p,(k\times sh)+q} \tag{3.2}$$

$$mean - pool : y_{ijk} = \frac{1}{p \times q} \sum_{pq} x_{i,(j\times sv)+p,(k\times sh)+q} \tag{3.3}$$

Where i is the feature map, jk is the index of the output y_i, p and q are the pooling window size. sv and sh are vertical stride and horizontal stride.

Stride and Overlap during Pooling

By using a variable stride, it becomes possible to to overlap the pooling windows when the window size is larger than the stride. Using overlapping pooling windows has been shown to give the best results[25][33]. In other instances, setting the stride equal to the window size and without overlap, has been shown to give the best results[44]. In practice, a variety of strides should be tested on any one individual problem to find the best result.

The topic of stride in the MaxPoolingFragmentated layer is discussed further in Chapter 4 'Implementation', due to the fact that Masci's fragmentated approach has been modified and generalized in the CUDArray library implementation to allow the user to modify the stride.

MaxPoolingFragment Layer

The MaxPoolingFragment layer is where the fragments in a MPFCNN are first created. When sub-sampling an image, a MaxPooling window is moved over the image in strides, as previously described. Pooling is often used with a stride larger than one in both directions in order to reduce image size. When working with fragments this creates an information retention problem, of which Giusti et al.[22] gives an example:

> "..when we perform a 2×2 max-pooling operation on an extended map, we obtain a smaller extended map which does not contain information from all the patches contained in the input image; instead, only patches whose upper left corner lies at even coordinates of the original image are represented. Any subsequent max-pooling layer would further aggravate the problem."

This problem is dealt with by using the MPF layer, which ensures that all information from the patches within the input fragments F_x are still contained within the output fragment F_y.

As described in [38], MaxPooling with a window size of p, q will be performed on the input image with $p \times q$ different offsets. Each offset produces an output fragment, thus the total number of output fragments will be $|F_x|pq$.

An example for one input fragment can be seen in Fig. 3.7, where a 2×2 pooling window yields 2^2 fragments and offsets. The set of offsets are thus $[(0,0),(0,1),(1,0),(1,1)]$..

In Masci et al. [38], the fragments are kept the same size, as opposed to Giusti et. al. [22], where the size of the fragment depends on the offset. In this thesis we will be using the approach of Masci et al., which ensures that every pixel is retained and results in simpler lines of code.

To keep the same dimension of each fragment, padding is added to the right side and bottom of the matrix. The padding can be seen in Fig. 3.7 where a $-inf$ padding has been added to ensure that the padding is not forward propagated.

With the Fragmented approach, MaxPooling is applied to every feature map within every fragment in F_x, thus $|F_y| = |F_x| \times p \times q$. The number of feature maps in each fragment in F_y is the same as the number of feature maps in each fragment in F_x.

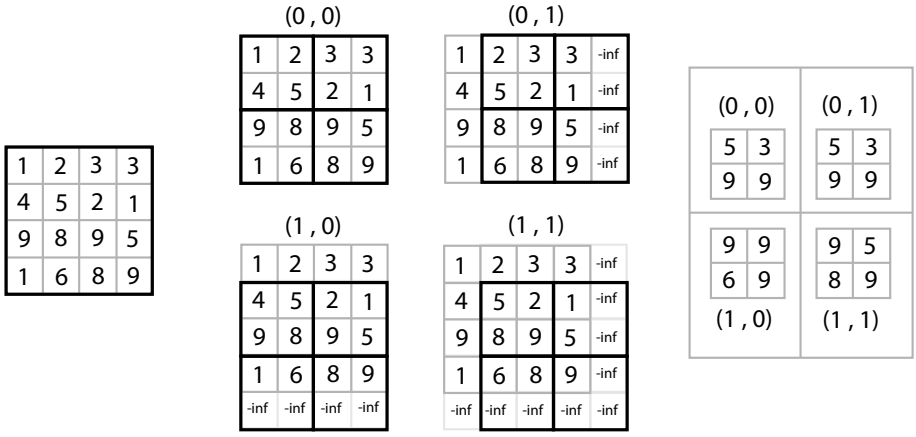


Figure 3.7: Max-pooling of a 4x4 image with a 2x2 window using fragmentation

3.1.5 Flattening Layer

After a series of alternating Convolution and MaxPooling layers, the final Pooling layer passes an output to the Flattening layer. The Flattening layer reduces that output to a 1D array in order to allow the upcoming Fully Connected layer to process the data.

Traditionally, the output of the convolution layers consists of k feature maps, each of size $i \times j$. The flattening layer flattens the 3 dimensions into an output array y as seen in Fig. 3.8. The output array y is of length $k \times i \times j$. y is then forward propagated to the Fully Connected layer.

Flattening Layer Fragmented Approach using Indexing

The Indexing and Flattening layer from the Fragmented approach, reduces the set of fragments F_x to a 2D array using an Index, which allows the upcoming Fully Connected layer to process the data in the same was as it would the output from a traditional Flattening layer. From the incoming set of fragments F_x each patch is extracted to a 2D set of arrays Y. Y will contain one array for each pixel in the image, where each array will contain the pixel information for each feature map.

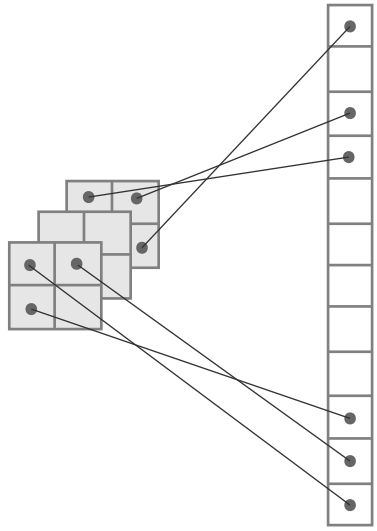


Figure 3.8: Flattening of a 2×2 image with 3 feature maps into an array

In Fig. 3.9 an example of flattening 4 fragments with 3 feature maps can be seen. In the figure, each patch that is flattened has a height and width of one pixel. In some cases it might be preferred to have a different size patch during flattening. Implementation of the Indexing and Flattening layer will be discussed in section 4.2.2.

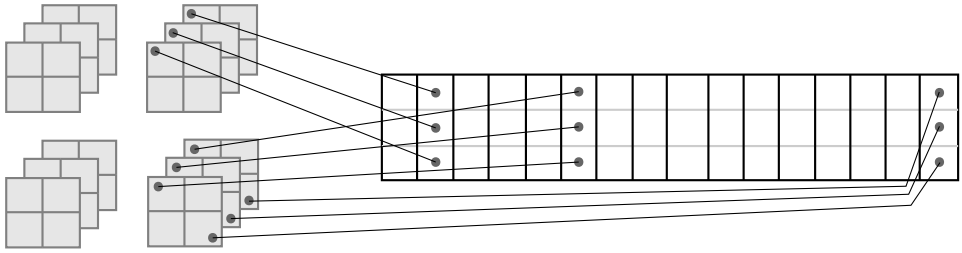


Figure 3.9: Flattening of a 2×2 image with 3 feature maps and 4 fragments into a batch of arrays

In order to put the picture back together, the index of pixels in F_x must be able to be mapped back to the original picture. Using the information from the Pooling layers, the index of each pixel in each fragment can be mapped back to

its original position in the image. This is done by tracking the index through the pooling layers and again, will be discussed in the implementation section 4.2.2.

3.1.6 Fully Connected and Dropout Layer

The Fully Connected layers behave in the same manner for the Fragmented approach as for the traditional Patch based approach. The Fully Connected layer consists of j nodes, where each node is connected to all i nodes in the previous layer giving $j \times i$ connections.

In Fig. 3.10 a standard Fully Connected layer can be seen. The input vector x is supplied from layer $l-1$ that could be the flattening layer or another Fully Connected layer. The Fully Connected layer consists of a wight matrix W containing a weight w_{ij} for each connection between x_i and y_j. y_j is called the pre-activation and is thus a linear combination of the input x. Non liniarity is applyed to y_j using an activation function $g(x)$ giving the output value a_j. The activation function is the same as described in section 3.1.3, and often takes the form of one of the functions in Fig. 3.5. The bias b has been omitted from the figure for simplicity but contributes to y.

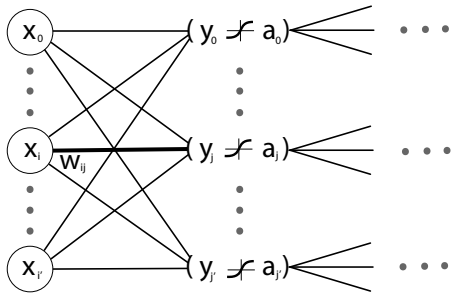


Figure 3.10: Fully connected layer with the bias omitted, x as the input, y as the pre-activation and a as the output.

The pre-activation y^l and output a^l for a Fully Connected layer can be written in vector form as:

$$y^{(l)} = W^{(l)}x + b^{(l)} \tag{3.4}$$

$$a^{(l)} = g(y^{(l)}) \tag{3.5}$$

Here, x is the output of the previous layer. Therefore, $x = a^{(l-1)}$ when layer $l-1$ is a fully connected layer.

The final Fully Connected layers will contain the same number of notes as the output of the network should contain. If the problem is a classification problem, this will equal the number of classes.

Dropout in a Fully Connected Layer

When dealing with a deep network which has a large number of parameters, over-fitting can be a problem. Dropout is a method which can reduce over-fitting by preventing co-adaptations of the training data. During each training pass, Dropout is achieved by assigning an omission probability of $1-p$ to every node in the hidden layer.

By dropping the nodes during every training pass, we can view this as training a number of entirely different networks at once. During prediction, all the nodes are used, and this can be viewed as ensembling, which has been shown to generate positive results [33].

Using dropout, the fully connected hidden layer operation described in the previous section, is shown below.

$$r_j^{(l)} \sim Bernoulli(p) \tag{3.6}$$

$$\widetilde{x} = r^{(l)} \odot x \tag{3.7}$$

$$y^{(l)} = W^{(l)}\widetilde{x} + b^{(l)} \tag{3.8}$$

$$a^{(l)} = g(y^{(l)}) \tag{3.9}$$

Where $\odot$ is element wise multiplication, $Bernoulli(p)$ produces a vector with $|x|$ independent probabilities, and $x = a^{(l-1)}$ when layer $l-1$ is a fully connected layer.

During prediction, the operation is as shown below, generating an ensemble of different networks, which were created during training.

$$\widetilde{x} = x \times (1-p) \tag{3.10}$$

$$y^{(l)} = W^{(l)}\widetilde{x} + b^{(l)} \tag{3.11}$$

$$a^{(l)} = g(y^{(l)}) \tag{3.12}$$

3.1.7 Multinomial Logistic Regression Output Layer

A Multinomial Logistic Regression layer is often used as the output layer for multi-class classification. The desired output of a network for a multi-class problem is the conditional probability $p(y = c \mid x)$.

To ensure that the output is a probability distribution over the classes, the softmax function is used. Given the input x from the last hidden layer in the network, the output of the network $y = g(x)$, where $g(x)$ is given by the softmax activation function, as shown below.

$$g(x) = \left[\frac{exp(x_0)}{\sum_c exp(x_c)}, \ldots, \frac{exp(x_{c'})}{\sum_c exp(x_c)} \right]^\top \tag{3.13}$$

Where the exp of each value in the input vector $x = [x_0 \ldots x_{c'}]$ is divide by the sum of the exp of all c values in x. Guaranteeing that the $g(x)$ will sum to one an d be strictly positive.

In this case it is required that the the last hidden layer in the network has the same amount of nodes as number of classes c. The output y is thus the networks prediction of the probability that the given input belongs to each class. y can then be used to train a network or can be used as an already trained network y in order to classify unknown data.

3.2 Training

When training a network, the goal is to find the best fit for weights throughout the hidden layers, such that the network produces the best results for a given problem. To train the network, we use training data in which the truth of the data is know. One example could be the MINST dataset, where the input data x is handwritten numbers as seen in Fig. 3.11. The ground truth y for x would be 6 and 8.

Figure 3.11: Handwritten characters from the MINST data set

Training the network involves running an epoch, or a training pass, containing known data. In an epoch, the data is forward propagated through the network

as described throughout section 3.1. For training, the error from the output layer is found, and then back propagated through the network to update the weights within each of the hidden layers. The training data can be propagated either one-to-one, or in batches.

3.2.1 A Word on backpropagation

The practical goal of backpropagation is to update the weights within each hidden layer in order to minimize the classification error, eventually reaching the best possible classification results.

Instead of optimizing directly on the classification error, which is either true or false and which predicts either the correct or incorrect class, we define a cost function C which allows for a smoother error function. The cost function is defined as $C(f(x^t;\theta), y)$, where θ is all the parameters of the network.

To optimize the weights and biases in the network, stochastic gradient decent is used. Stochastic Gradient Decent uses the $\partial C/\partial W$ and the $\partial C/\partial b$ of the cost function with respect to any weight W or bias b in the network, which are both calculated by the backpropagation algorithm.

We also calculate the derivative of C with respect to θ, because the derivative reveals the direction in which the biggest increase in the cost function will happen. Going in the opposite direction gives us the biggest decrease in the cost function.

A notation which is going to be used in the following section is δ^l, which represents the error of layer l. For layer l with pre-activation y^l, the error δ^l is defined as

$$\delta^l = \frac{\partial C}{\partial y^l} \tag{3.14}$$

By propagating δ^l back through the network, δ^l can be found for each layer using the chain rule. Furthermore, by using δ^l in each layer, $\partial C/\partial W$ and $\partial C/\partial b$ can be found throughout the network.

3.2.2 BP of Multinomial Logistic Regression Output Layer

As mentioned, the layer error δ^l must be extracted from the output layer and will then be backpropagated through the hidden layers in order to update weights

and biases. The error is extracted in the same way for both the fragmentation and the patch approach. For this thesis, the cross-entropy cost function is used. The cost function will take the form shown below.

$$f(x)_c = p(y = c \mid x) \tag{3.15}$$

$$l(f(x), t) = -\sum_c 1_{c=t} \, log \, f(x)_c = log \, f(x)_t \tag{3.16}$$

The partial derivative of the output layer, using softmax, can be found to be:

$$\delta_j^L = -(1_{(c=t)} - f(x)_c) \tag{3.17}$$

The Gradient for the pre-activation of the output layer is thus:

$$\delta^L = -(e(y) - f(x)) \tag{3.18}$$

Where $e(y)$ is the one-hot vector of the ground truth y and of the same size as the number of classes. Each position in $e(y)$ is zero, except in position y.

3.2.3 BP of Fully Connected and Dropout Layer

As described in section 3.1.6, the Fully Connected layer behaves in the same way when used in the new fragmentation method as when used in the traditional patch method.

The δ^l is the pre-activation error of layer l and is calculated as follows:

$$\delta^l = (W^{l+1})^T \delta^{l+1} \odot g'(y^l) \tag{3.19}$$

One way it can be understood is that at layer $l+1$, the error δ^{l+1} is known. Applying the transposed weight matrix of layer $l+1$ can be viewed as propagating the error back to layer l. Calculating the element wise multiplication between the error and $g'(y^l)$ can be viewed as propagating the error back through the activation function. This results in δ^l for the given weight input layer.

Using δ^l, the derivative of the cost function with respect to the weight can be found using the following equation:

$$\frac{\partial C}{\partial W^l} = x^l \delta^l \tag{3.20}$$

Where $x = a^{(l-1)}$ as long as layer $l-1$ is a fully connected layer, otherwise x will be the output of the flattening layer.

The input from the bias is by definition set to 1. The cost function with respect to the bias can therefore be expressed as:

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{3.21}$$

The derivative of the activation function that is used in equation 3.19 can take the shape of one of three activation function mentioned earlier can be seen below.

$$\text{sigmoid} \qquad g'(a) = g(a)(1 - g(a)) \tag{3.22}$$

$$\tanh \qquad g'(a) = 1 - g(a)^2 \tag{3.23}$$

$$\text{ReLu} \qquad g'(a) = 1_{a>0} \tag{3.24}$$

3.2.4 BP of Indexing and Flattening Layer

During forward propagating of the Flattening layer, the output from the last pooling layer is mapped to an array. When back propagating, this is done in reverse, mapping each element in $\delta^{(l+1)}$ back to the position in the respective feature map.

The same is true for the fragment approach. One thing to consider is that each pixel is processed individually by the fully connected layers. Therefore, all pixels must have been processed prior to mapping. Each $\delta_j^{(l+1)}$ from layer $l+1$ is mapped back to its respective position according to it's index.

3.2.5 BP of MaxPooling and MaxPoolingFragmented Layers

The pooling layer does not employ an activation function and therefore uses the error $\delta^{(l+1)}$ directly. Furthermore, there are no weights to be updated in the pooling layer. When backpropagating through the pooling layer, we assume that layer $l-1$ is a convolution layer and therefore must move the error through the activation function.

For every feature map j and every row and column j, k in $\delta^{(l-1)}$, upsampling,

as opposed to subsampling, is performed as follows:

$$\delta_{ijk}^{(l-1)} = 0 \text{ except for } \delta_{i,j+p',k+q'}^{(l-1)} = \delta_{ijk}^{(l)} \odot g'(y_{ijk}^{(l-1)}) \tag{3.25}$$

Where p', q' is the index of the max value in the pooling window. Thus only the positions from where the max pooling values originated are non zero.

In cases where the sub-sampling windows overlap, a given $\delta_{i,j+p',k+q'}^{(l-1)}$ can account for more positions in $\delta^{(l)}$. In these cases the $\delta_{ijk}^{(l-1)}$ will be accumulated.

For mean pooling the values are upsampled and divided by the pooling window size as follows:

$$\delta^{(l-1)} = \frac{1}{p \times q} upsample(\delta^{(l)}) \tag{3.26}$$

Where *upsample* inverts subsampling.

BP of MaxPooling Fragmented layer

In the fragment method, the pooling layer is the layer where the input fragments are themselves split into smaller fragments. When back propagating, these new fragments in δ^l are consolidated again. For each fragment j created from an input fragment i in the forward propagation pass:

$$\delta_i^{(l-1)} = \sum_i maxpool'(\delta_j^{(l)}) \tag{3.27}$$

Where $maxpool'(\dot{})$ is the equation 3.25.

This process is illustrated in Fig. 3.12 where $\delta^{(l-1)}$ contains 1 fragment and offsets $(1,1)$ and $(0,0)$ share the same max-value, thus accumulating the error values.

3.2.6 BP of Convolution Layer

When back propagating through the convolution layer, we look at each feature map i in layer $l-1$. The error for feature map i is given by:

$$\delta_i^{(l-1)} = \sum_j (\delta_j^{(l)}) \underline{*} (W_{ij}^{(l)}) \tag{3.28}$$

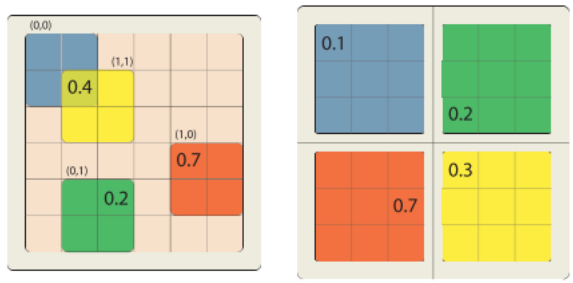


Figure 3.12: Max Pooling Backpropagation from right to left, from Masci et al.[38]

Where $\underline{*}$ is convolution, as in equation 3.1 with full padding and $\delta_j^{(l)}$ is the error of the j'th feature map in layer l.

The gradients for the the kernel weights W_{ij}^l are computed using the backpropagation algorithm. When convoluting the same weights, W_{ij}^l is used for the kernel across the whole feature map $x_i^{(l)}$. Thus the derivative of C with respect to W_{ij}^l can be expressed as the the sum of all errors in $\delta_j^{(l)}$ multiplied element-wise with the respective kernel position in $x_i^{(l)}$. This can be written more concisely using the definition of convolution.

$$\frac{\partial C}{\partial W_{ij}^l} = (\delta_j^{(l)}) * \tilde{x}_i^{(l)} \tag{3.29}$$

Where $\tilde{x}_i^{(l)}$ is the input $x_i^{(l)}$ with its rows and columns flipped and $*$ denote the convolution operation. When performing backpropagation on the convolution layer in a MPFCNN, the gradient is found for each feature map in each fragment. In the convolution layer the number of fragments do not change. The error $\delta_j^{(l-1)}$ for a fragment f_i in F_x is found in the same way as described in equation 3.28. This is due to the fact that the data structure of f_i is the same as a typical DCNN.

$$\delta_{f,i}^{l-1} = \sum_j (\delta_{f,j}^l) \underline{*} (W_{ij}) \tag{3.30}$$

Where $\underline{*}$ is convolution with full padding.

Each fragment uses the same set of weights W. The derivative of C with respect to W can thus be accumulated from each fragment f in layer l.

$$\frac{\partial C}{\partial W_{ij}^l} = \sum_f ((\delta_{f,j}^{(l)}) * \tilde{x}_{f,i}) \tag{3.31}$$

3.2.7 Weight update during BackPropagation

Updating the weights in the hidden layers can be done once the $\frac{\partial C}{\partial W_t}$ is found. Finding the derivative of the cost with respect to θ shows the direction giving the biggest increase in the cost function. When updating the weight to minimize the cost function we want to step in the opposite direction.

$$\Delta = -\epsilon \frac{\partial C}{\partial W_t} \tag{3.32}$$

$$W_{t+1} = W_t + \Delta \tag{3.33}$$

Where t is the iteration index, ϵ is the learning rate, and for batches, $\frac{\partial C}{\partial W_t}$ is the average derivative with respect to W_t of the t^{th} batch.

As described by [24], when moving down the error surface using gradient decent, the weights can get stuck in a local minima. One solution for this, is using momentum which acts as a low pass kernel that allows the network to ignore small features in the error surface. Without momentum there is a bigger chance that the network could get stuck in shallow local minima, but because the network not only relies on the given trajectory, but takes the previous trajectories into account, such minima can be avoided. When updating using momentum, the the following rules are used, where t is the iteration index and m is the momentum constant:

$$\Delta_t = m\Delta_{t-1} - \epsilon \frac{\partial C}{\partial W_t} \tag{3.34}$$

$$W_{t+1} = W_t + \Delta_t \tag{3.35}$$

Using weight decay can also be beneficial for training the network, regularizing the weights by penalizing large weights. The weight update for momentum and weight decay becomes:

$$\Delta_t = m\Delta_{t-1} - \epsilon \frac{\partial C}{\partial W_t} - \lambda \epsilon W_t \tag{3.36}$$

$$W_{t+1} = W_t + \Delta_t \tag{3.37}$$

Where λ is the regularization parameter. The biases are effectively updated in the same manner as the weights. Because the fragmentation approach is based on representing each patch within the whole image, it can be view as processing all the patches in one big batch. When updating the weights for the convolution layers, $\frac{\partial C}{\partial W_t}$ must be divided with the number of pixels in the image, effectively taking the the mean of $\frac{\partial C}{\partial W_t}$ for the batch. In the fully connected layer each pixel is processed individually, standard rules are applied when updating W for a single training example or for batches.

Chapter 4

Implementation

In order to confirm the hypothesis made in Chapter 2, a MPCNN using the Fragmentation approach, or a MPFCNN, will be implemented as a fully parameterizable CNN within a DNN Python library for use on CPUs and GPUs, at such a quality that it can be distributed through github.

The new library, capabilities will be based on python and will hook to CUDA code. The library will be developed with speed, modularity and extensibility in mind. In this chapter we will begin to understand the CUDArray and DeepPy frameworks and we will review implementation of specific components of said frameworks.

4.1 What are CUDArray and Deeppy

CUDArray(CA) and DeepPy(DP) together comprise a Pythonic NN library, which was established by Anders Boesen Lindbo Larsen, and has been developed through collaboration between Anders and myself. CA and DP make accessible to our academic colleagues the state of the art fragmentation approach for MPCNN. CA and DP are both open source projects under the MIT license.

During this thesis period, my contribution to the CUDArray library has consisted of developing and implementing the following library components:

- CPU implemented patch based approach for the Convolution layer
- CPU implemented patch based approach for the Pooling layer
- CPU, Multi-threaded CPU implemented fragment based approach for the Indexing and Flattening layer.
- CPU, Multi-threaded CPU and GPU implemented fragment based approach for the Convolution layer
- CPU, Multi-threaded CPU and GPU implemented fragment based approach for the MaxPoolingFragment

Also during this thesis period, my contribution to the Deeppy module of the CA library has consisted of developing and implementing the following library components:

- a patch based approach for the Convolution layer
- a patch based approach for the Pooling layer
- a patch based approach for the Dropout layer
- a fragment based approach for the Convolution layer
- a fragment based approach for the MaxPoolingFragment
- a fragment based approach for the Indexing and Flattening layer.
- updates and modifications of the network layer connections

4.1.1 CUDArray Library: a subset of the NumPy Library

As is described on the github repository, CA is a CUDA-accelerated subset of the NumPy library, and acts as the backend of the library where many of the calculations are performed. The goal of CA is to combine the ease of development from NumPy with the computational power of Nvidia GPUs in a lightweight and extensible framework. CA supports different data types, offers CPU fallback and, as described by Anders [34],

> 'CUDAaray aims to be a drop-in replacement for NumPy for all NN needs. This means that CUDAaray does not offer all the functionality of NumPy, instead it offers only the essentials needed for NN's. CUDArray also offers more specific functionality for NN, DNN and DCNN such as image convolution, softmax and one hot encoding.'

4.1.2 Deeppy Module for State of the Art Neural Nets

Again as is described on the github repository, DeepPy is a module of CUDArray which tries to combine state-of-the-art deep learning models with a Pythonic interface in an extensible framework [2], and acts as the frontend of the library. DP binds the math together to form the neural network capabilities, using CUDArray to perform calculations.

4.1.3 Motivation for CUDArray and Deeppy

Considering that the MPFCNN method has been shown to achieve state of the art performance and results for a variety of applications throughout 2014, it is important that academics have access to the new MPFCNN method within easily accessible libraries as soon as possible.

Existing NN Libraries

Fortunately though, as mentioned in Chapter 1 'Introduction', there are a number of high-performing, open source NN libraries available to academics which incorporate many of the state of the art methods.

Unfortunately again however, there are very few which strike a balance between speed and usability. The following is a brief review of major available NN libraries, with and without Python.

- Caffe: This is the closest non-Python library to what was envisioned for CUDArray and DeepPy. The only caveat is that it has a C++ backend, while using a Python interface. It was developed by the Berkeley Vision and Learning Center.[30]
- CUDAMat combined with Gnumpy:[41] [51] These combine to form a Python matrix class that uses CUDA for performing calculations on a

GPU. It is also closely related to the approach taken by CUDArray. As described by Anders [34], 'CUDAMat implements common matrix operations and exposes them through a Python module without adhering to the NumPy interface. A notable limitation of CUDAMat is its focus on 2D arrays of data type float'.

- Theano: A GPU-based numerical Python library, Theano allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. It also includes a complex optimization function, which re-compiles your code, making it quite difficult and time consuming to debug while developing a new library modification. It trains fast, but array expressions must be explicitly compiled before usage.[6]

- Torch7: Based on the LuaJIT language with an underlying C/CUDA implementation, this library offers a Matlab-like environment for state-of-the-art machine learning algorithms.[3]

- cuda-convnet2: A fast convolutional neural network in C++/CUDA. Very fast, but not very user-friendly. It is generally hard to hack and to modify.[1]

- nnForge: a C++ library for training convolutional and fully-connected neural networks. The author does very well in Kaggle competitions.[40]

- PyLearn2: Most of PyLearn2's functionality is built on top of Theano.It is incredibly easy to get started with, but is like a black box when it comes to debuging or modifying.[43]

Project Motivation

The motivation behind CUDArray was to facilitate neural network programming with a simple, extensible, lightweight framework, allowing for quick and flexible experimentation, without requiring a GPU. This would allow researchers to develop and test new networks quickly on a laptop computer, before running the network on a larger external server.

Some of the existing networks train fast, but testing a new idea requires a large time investment in learning the framework or code to build the network, before the training can even begin. Providing custom CUDA kernels for CA and DP is made relatively easy thanks to the lightweight framework, and the simplicity and modularity of CA makes it easy to extend with custom functionality, since little knowledge about the framework is needed. One needs only to understand Python and Numpy in order to begin using CA and DP.

After my own experiences with NN libraries, I was eager to contribute to the CA and DP frames works for three major reasons. First, CA and DP are written almost entirely in Python, a language that is largely and increasingly popular among academics [23].

Second, CA and DP would be very modular, loosely coupled and largely without optimization through code recompiling, meaning debugging would be much easier. Essentially, the code you write and submit largely will not be altered.

And Third, CA and DP are object oriented, allowing one to create a neural network using layer objects, rather than a network configuration file. These three aspects of CA and DP ensure that the frameworks would be highly flexible and extensible, as well as easy to approach and efficient to work with.

4.2 Implementation of MPFCNN in CA and DP

In this section, we will review only the implementation of the fragmentation approach to MPCNNs in CUDArray and DeepPy which vary from the implementations of [38] and [22]. My full contributions to CA and DP were reviewed in the beginning of section 4.1 and can be found in the framework's github repository. Most of the discussion in this section centers around the MPF layer and the Flattening layer.

4.2.1 MaxPoolingFragmented Layer Implementation and Modifications

When implementing the MPF layer as described in section 3.1.4, the focus was on implementing a MaxPooling operation where the stride and window size were parameters which could be defined by the user.

In their paper, Giusti and Masci et al. fix the window and stride size to $k \times k$. They base the number of fragments created, on k. When this assumption is ignored to allow for flexibility, the number of fragments created in the pooling layer must then be based on the stride size.

$$|F|^{(l+1)} = |F|^{(l)} \times stride_h \times stride_v \tag{4.1}$$

The number of offsets o is thus defined as:

$$|o| = stride_h \times stride_v \tag{4.2}$$

To ensure that all patch information is retained, the height and width of the pooled image, and thus the number of pooling operations performed on y for each offset, is defined as follows:

$$|y_h| = \lfloor \frac{(x_h - st_h)}{st_h} \rfloor + 1 \tag{4.3}$$

$$|y_v| = \lfloor \frac{(x_v - st_v)}{st_v} \rfloor + 1 \tag{4.4}$$

for the vertical and horizontal size of y, given the size of x and the stride st.

When using an overlapping pooling operation, as described above, said operation will result in the pooling window exceeding the boundaries of the image. To handle this we state that the operation will always start in the top left corner and padding is added to the right and bottom side of the image. The padding added is of value $-inf$ which ensures that the padding will not be chosen by the max pooling. An example of this can be seen in Fig. 4.1.

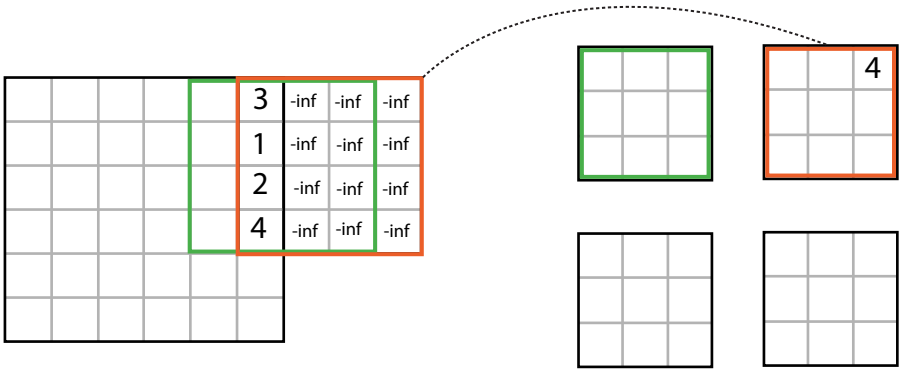


Figure 4.1: 4×4 Pooling window with a stride of $(2, 2)$ exeeding the image boundaries on its 3 stride in offset $(0, 1)$

4.2.2 Flattening Layer Implementation and Modifications

While it is assumed that Giusti and Masci et al. modify their Flattening layers, their method of doing so was not explicitly stated. The following is a description of how this layer was handled in CA and DP.

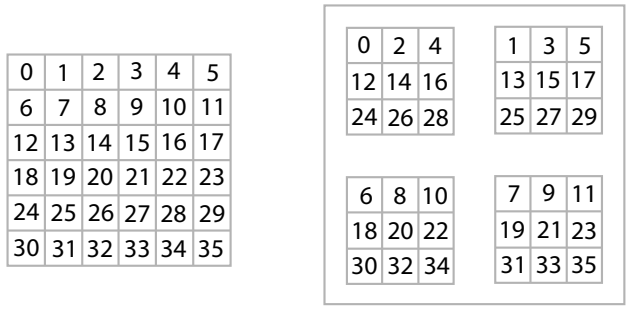


Figure 4.2: Indexing of a 6×6 image through a pooling layer with stride $(2, 2)$

When using the traditional patch method to segment an image, the patch is not required to be sub-sampled down to one pixel before it reaches the fully connected layers. To reflect this in the fragment approach, a new flatting layer was developed. When building the MPFCNN, the Flattening layer is given a window size:

```
dp.Flatten_seg(win_shape=(3,3))
```

When flattening a pixel p_{fij}, the window is place over the image with p_{fij} as the middle pixel. All pixels under the window are flattened to a 1d array, keeping p_{fij} in the middle of the array. In the edge cases where the window exceeds the image boundaries mirror padding will be used. When back propagating through the flattening layer, the error value for $\delta_{fkij}^{(l-1)}$ will be an accumulation of all the errors in $\delta^{(l)}$ where the given pixel is inside the bounds of the flattening window.

Indexing Implementation

Indexing is implemented by tracing the fragmentation of a $0...n$ arranged matrix through the pooling layers. An example of the Indexing process through a single pooling layer with stride $(2, 2)$ can be seen in Fig. 4.2. The indexing is used to sort the pixels, starting from the top left corner, continuing in a row-wise manner. This is done during flattening, such that the output of the fully connected layer is already sorted. The indexing calculation gives insignificant overhead, due to the fact that it only requires one forward pass through the pooling layers in the network. The indexing of the network remains constant and the indexing is the same for all feature maps.

Chapter 5

Tesing and Results

In this sections different test parameters will be discussed. The test machines and hardware used can be seen in Appendix A.1. The data used in this section consists of the images from the ISBI challenge.

5.1 Speedup

In practice, speed of a network can vary between different implementations. To get a fair assessment, the new method has been tested against multiple implementations. The network used for testing the speed of the Fragmented approach, is the network described in Table 5.1.

During testing, a 512×512 image was forward propagated through the network. No training, pre-processing or post processing time was recorded. This means that the time necessary for creating the patches when using the patch based approach is not included in the testing time.

Layer (l)	Type	Maps	Window size
1	convolution	48 maps	4×4
2	max pool	48 maps	2×2
3	convolution	48 maps	5×5
4	max pool	48 maps	2×2
5	convolution	48 maps	4×4
6	max pool	48 maps	2×2
7	flatten	100 neurons	1×1
8	fully connected	100 neurons	1×1
9	fully connected	2 neurons	1×1

Table 5.1: Neural network architecture used for testing the speed of different approaches

The testing time results can be seen in Table 5.2. From the table we can see that the new fragmentation approach greatly increased the speed of the network. Even the CPU implementation, running on a single thread, outperforms the patch approach on a GPU.

Method	Time per image [s]	Relative to GPU-patch
CPU Patch	2126.78	-
Theano GPU Patch	67.48	1.00
Deeppy CPU Fragmentation 1 core	47.62	1.41
Deeppy CPU Fragmentation 20 cores	4.12	15.89
Deeppy GPU Fragmentation	0.55	123.6

Table 5.2: Time for different implementations to segment an test example of size 512×512

The actual speedup of 123.6 exceeds the calculated theoretical speedup of 68.4 from Table 2.1 in Chapter 2. This is mainly due to the fact that, in the patch based method, all patches could not be processed in a single batch, most likely due to memory restriction on the GPU, though this assumption requires further investigation.

While performing the speed tests, the batch for the patch based method using Theano on a GPU was restricted to 64^2, whereas, a full image could be processed on the GPU using the fragment method, without restriction. From Table 5.3, it can be seen that the speedup increases as the image size increases, again likely due to the memory limitation on the GPU.

image size	speedup
64	10.3
128	35.0
256	82.8
384	113.3
512	123.6

Table 5.3: Speedup of segmentation from the patch method on GPU to the fragmentation method on GPU, for the given image size, from

From Fig. 5.1, we can see the time it takes to forward propagate images of different sizes accross differing methods. For small images using the Fragmented approach, the multi-threaded CPU implementation is on par with the GPU implementation. Thus, the GPU is truely utilized when processing larger images.

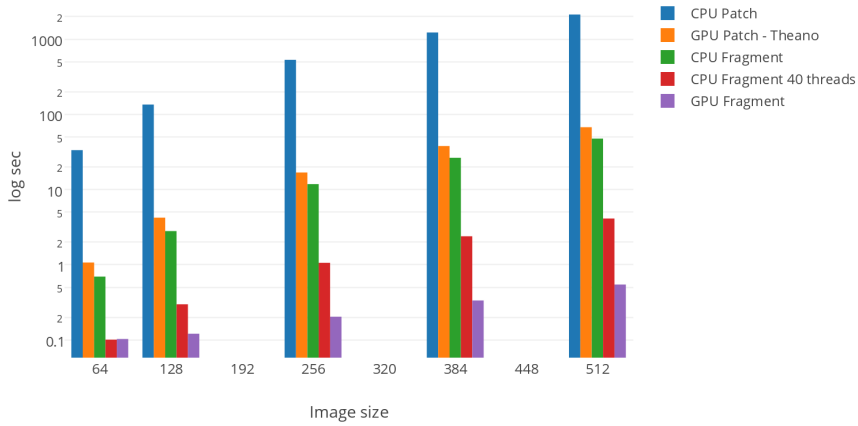


Figure 5.1: The time in sec on a log scale for the net described in tabel 5.1 to segment a square image of the given size

5.2 Layer Performance

One of the changes made to the fragmented approach network in CA and DP, was enabling window size flexibility in the Flattening layer. In order to test this,

four networks with the architecture described in Table 5.1, with fully connected layers of 200 nodes were trained.

Two of the networks used a flattening layer of 1×1 and two used a flattening layer of 3×3. The results can be seen in Fig. 5.2, where the test error for each epoch is shown. The error rate used is the mean absolute error of classifying each pixel in the images.

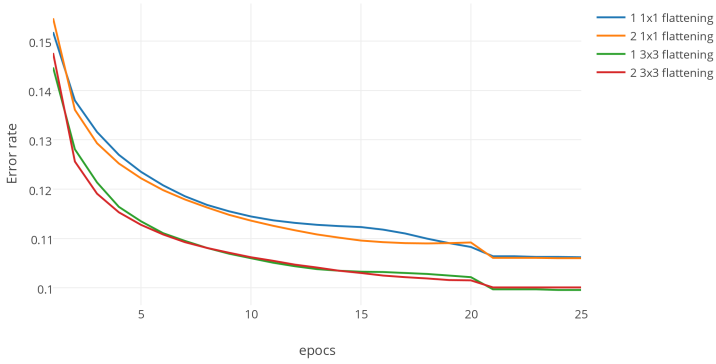


Figure 5.2: Test error for 4 networks using the architecture in Table 5.1 with a 200 fully connected layer and a 1×1 or 3×3 flattening layer. Using 100 training and 20 testing images

From figure 5.2, it can be seen that using the larger window size decreased the error by 6.2%. Which show that the window size of the flattening layer does have an effect on the error rate.

5.3 ISBI 2012 Electron Microscopy Segmentation Challenge

In order to test the the performance of the implemented fragmentation method, a network based on the architecture in [10] was used on the ISBI 2102 Electron Microscopy Segmentation challenge, where the main focus was on the pixel error, defined as 1 - the maximal F-score of pixel similarity.

In Fig. 5.3 an example of the input image can be seen as the left most image. Second from the left shows an output from one of the hidden layers. After this the output image can be seen and then the binary output. The right most and last image show the true labeled binary.

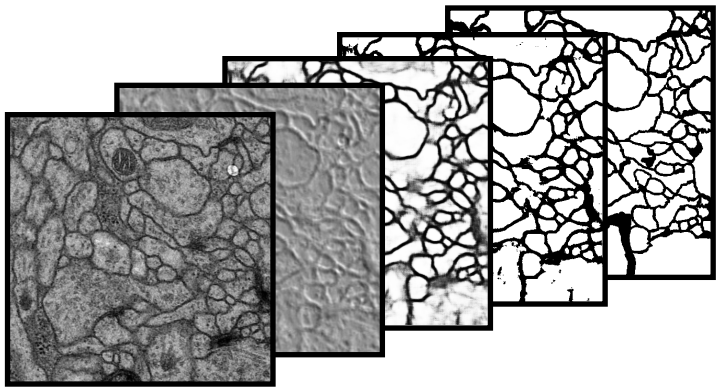


Figure 5.3: From the left: Input image, Hidden layer, output image, one hot output image, true segmentation

The network was trained using 225 training images and 15 validation images. The 240 images were synthesized from the original 30 images by flipping and rotating the images. After 35 epochs, the network reached convergence. Each epoch took on average of 1244.5 seconds. The network processed on average approximately 47,394 patches per second, in comparison to Masci et al.'s average of 4500 patches per second [38].

After training, a test set was segmented and sent to the competition server for evaluation. The result was a pixel error of 0.078, and furthermore, placed 13 out of 34 on the challenge leaders board[4]. This pixel error result was obtained using only the MPFCNN and no post-processing of the images.

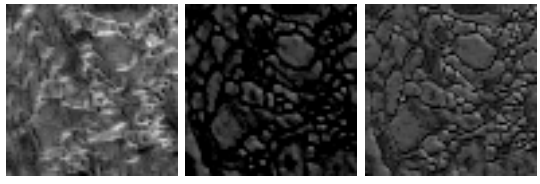


Figure 5.4: 3 feature maps from the same fragment at the lower level of the network

Chapter 6

Conclusion

After a review of the theory and architecture of MPCNNs for image segmentation, a review of the implementation of a MPFCNN in the frameworks CUDArray and DeepPy, and a review of the testing and results of the framework and network, we arrive at the conclusion.

The implementation of the fragmentation method in Cudarray and DeepPy achieves segmentation performance on par with Masci et al.'s state of the art implementation. Our pixel error rate of 0.078 with no post processing, compared to their pixel error rate of 0.068 [38], confirms that the CA DP MPFCNN was implemented correctly. These pixel error rates are in reference to the ISBI 2012 Electron Microscopy Segmentation Challenge.

In addition to the successful pixel error rate, CA and DP's MPFCNN with GPU implementation achieves state of the art performance in relation to speed, offering our academic colleagues access to a image segmentation method which is approximately 123.3 times faster than the existing patch based method within existing NN libraries.

CA and DP as they have been implemented in this thesis are currently located on github within their own branch. They will in the very near future, likely immediately following this thesis, be merged into the master thread, once naming conflicts have been resolved. Also in the future, I intend to explore using

dropout for the convolution layers, which has recently been shown to further reduce the overall classification error rate [47], as well as explore additional network layer functionality, like local response normalization.

The results detailed in Chapter 5 confirm our hypothesis that the best possible solution available today for implementing DCNNs for fast image segmentation and pixel level classification tasks on CPUs and GPUs without significant loss of accuracy employs the use of Masci's Fragmented approach on a GPU-implemented Max-Pooling Convolutional Neural Network which includes Supervised training through backpropagation using Stochastic Gradient Decent. Said network can now be found on github under the MIT Open Source license within the CUDArray and Deeppy frameworks.

Appendix A

First Appendix

A.1 Test Computers

CPU Test computer:

42 x IBM NeXtScale nx360 M4 nodes

Each node is configured with:

- 2x Intel Xeon Processor E5-2680 v2 (ten-core, 2.80GHz, 25MB L3 Cache)
- 128 GB memory
- Scientific Linux 6.4
- QDR Infiniband interconnect
- 500 GB internal SATA (7200 rpm) disk for OS and applications

In total:

- 840 Cores

- 5,25 TB RAM

GPU Test computer:

4 x HP ProLiant SL390s G7 nodes – GPGPU

Each node is configured with:

- 2x Intel Xeon Processor X5550 (six-core, 2.66GHz, 12MB L3 Cache)
- 2x Tesla S2050 GPUs
- 48 GB memory
- Scientific Linux 6.4
- QDR Infiniband interconnect
- 500 GB internal SATA (7200 rpm) disk for OS and applications

In total:

- 48 Cores
- 192 GB RAM

Bibliography

[1] cuda-convnet2,. `https://code.google.com/p/cuda-convnet2/`. [Online; accessed 01-DEC-2014].

[2] deeppy,. `https://github.com/andersbll/deeppy`. [Online; accessed 26-Feb-2015].

[3] torch,. `http://torch.ch/`. [Online; accessed 26-Feb-2015].

[4] Segmentation of neuronal structures in EM stacks challenge - ISBI 2012,. `http://brainiac2.mit.edu/isbi_challenge/`, 2012. [Online; accessed 26-Feb-2015].

[5] Hossein Azizpour, Ali Sharif Razavian, Josephine Sullivan, Atsuto Maki, and Stefan Carlsson. From generic to specific deep representations for visual recognition. *arXiv preprint arXiv:1406.5774*, 2014.

[6] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[7] Sven Behnke. *Hierarchical neural networks for image interpretation*, volume 2766. Springer Science & Business Media, 2003.

[8] 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 biology*, 8(10):e1000502, 2010.

[9] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*, 2014.

[10] Dan Ciresan, Alessandro Giusti, Luca M. Gambardella, and Jürgen Schmidhuber. Deep neural networks segment neuronal membranes in electron microscopy images. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2843–2851. Curran Associates, Inc., 2012.

[11] Dan Ciresan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. A committee of neural networks for traffic sign classification. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1918–1921. IEEE, 2011.

[12] Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[13] Dan C Cireşan, Alessandro Giusti, Luca M Gambardella, and Jürgen Schmidhuber. Mitosis detection in breast cancer histology images with deep neural networks. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2013*, pages 411–418. Springer, 2013.

[14] Dan C Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1237, 2011.

[15] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

[16] Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Convolutional neural network committees for handwritten character classification. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 1135–1139. IEEE, 2011.

[17] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[18] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537, 2011.

[19] C Cuda. Programming guide. *NVIDIA Corporation, July*, 2012.

[20] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoffrey Zweig, Xiaodong He, Jason Williams, et al. Recent advances in deep learning for speech research at microsoft. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8604–8608. IEEE, 2013.

[21] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[22] Alessandro Giusti, Dan C Cireşan, Jonathan Masci, Luca M Gambardella, and Jürgen Schmidhuber. Fast image scanning with deep max-pooling convolutional neural networks. *arXiv preprint arXiv:1302.1700*, 2013.

[23] By Philip Guo. Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities. `http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-fulltext`, July 7, 2014. [Online; accessed 01-DEC-2014].

[24] Martin T Hagan, Howard B Demuth, Mark H Beale, et al. *Neural network design*. Pws Pub. Boston, 1996.

[25] GE Hinton and N Srivastava. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv: ...*, pages 1–18, 2012.

[26] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[27] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[28] Arjun Jain, Jonathan Tompson, Mykhaylo Andriluka, Graham W Taylor, and Christoph Bregler. Learning human pose estimation features with convolutional networks. *arXiv preprint arXiv:1312.7302*, 2013.

[29] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *Proc. International Conference on Computer Vision (ICCV'09)*. IEEE, 2009.

[30] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[31] Alex Krizhevsky and G Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 2010.

[32] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[34] Anders Boesen Lindbo Larsen. Cudarray: Cuda-based numpy. Technical report, Technical University of Denmark (DTU), 2014.

[35] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[36] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[37] Cheng-Lin Liu, Fei Yin, Da-Han Wang, and Qiu-Feng Wang. Chinese handwriting recognition contest 2010. In *Pattern Recognition (CCPR), 2010 Chinese Conference on*, pages 1–5. IEEE, 2010.

[38] Jonathan Masci, Alessandro Giusti, Dan Cireşan, Gabriel Fricout, and Jürgen Schmidhuber. A fast learning algorithm for image segmentation with max-pooling convolutional networks. *arXiv preprint arXiv:1302.1690*, 2013.

[39] Jonathan Masci, Ueli Meier, Dan Ciresan, Jürgen Schmidhuber, and Gabriel Fricout. Steel defect classification with max-pooling convolutional neural networks. In *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pages 1–6. IEEE, 2012.

[40] Maxim Milakov. nnForge,. `https://milakov.github.io/nnForge/`. [Online; accessed 26-Feb-2015].

[41] Volodymyr Mnih. Cudamat: a cuda-based matrix class for python. *Department of Computer Science, University of Toronto, Tech. Rep. UTML TR*, 4, 2009.

[42] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on*, pages 512–519. IEEE, 2014.

[43] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 2010.

[44] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *Artificial Neural Networks–ICANN 2010*, pages 92–101. Springer, 2010.

[45] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[46] Patrice Y Simard, Dave Steinkraus, and John C Platt. Best practices for convolutional neural networks applied to visual document analysis. In *2013 12th International Conference on Document Analysis and Recognition*, volume 2, pages 958–958. IEEE Computer Society, 2003.

[47] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[48] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. The german traffic sign recognition benchmark: a multi-class classification competition. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1453–1460. IEEE, 2011.

[49] Yi Sun, Yuheng Chen, Xiaogang Wang, and Xiaoou Tang. Deep learning face representation by joint identification-verification. In *Advances in Neural Information Processing Systems*, pages 1988–1996, 2014.

[50] Yi Sun, Xiaogang Wang, and Xiaoou Tang. Deep learning face representation from predicting 10,000 classes. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1891–1898. IEEE, 2014.

[51] Tijmen Tieleman. Gnumpy: an easy way to use gpu boards in python. *Department of Computer Science, University of Toronto*, 2010.

[52] Jonathan Tompson, Murphy Stein, Yann Lecun, and Ken Perlin. Real-time continuous pose recovery of human hands using convolutional networks. *ACM Transactions on Graphics (TOG)*, 33(5):169, 2014.

[53] Jonathan J Tompson, Arjun Jain, Yann LeCun, and Christoph Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In *Advances in Neural Information Processing Systems*, pages 1799–1807, 2014.

[54] Alexander Toshev and Christian Szegedy. Deeppose: Human pose estimation via deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1653–1660. IEEE, 2014.

[55] Juyang Weng, Narendra Ahuja, and Thomas S Huang. Cresceptron: a self-organizing neural network which grows adaptively. In *Neural Networks, 1992. IJCNN., International Joint Conference on*, volume 1, pages 576–581. IEEE, 1992.

[56] Ming Yang, Shuiwang Ji, Wei Xu, Jinjun Wang, Fengjun Lv, Kai Yu, Yihong Gong, Mert Dikmen, Dennis J Lin, and Thomas S Huang. Detecting human actions in surveillance videos. In *TREC Video Retrieval Evaluation Workshop*, 2009.