ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Efficient Convolutional Neural Networks for Pixelwise Classification on Heterogeneous Hardware Systems

Technical Report

Fabian Tschopp

September 11, 2015

Supervisors: Prof. Dr. Angelika Steger, Dr. Jan Funke
Department of Computer Science, ETH Zürich

Abstract

This work presents and analyzes three convolutional neural network (CNN) models for efficient pixelwise classification of images. When using convolutional neural networks to classify single pixels in patches of a whole image, a lot of redundant computations are carried out when using sliding window networks. This set of new architectures solve this issue by either removing redundant computations or using fully convolutional architectures that inherently predict many pixels at once.

The implementations of the three models are accessible through a new utility on top of the Caffe library. The utility provides support for a wide range of image input and output formats, pre-processing parameters and methods to equalize the label histogram during training. The Caffe library has been extended by new layers and a new backend for availability on a wider range of hardware such as CPUs and GPUs through OpenCL.

On AMD GPUs, speedups of 54$\times$ (SK-Net), 437$\times$ (U-Net) and 320$\times$ (USK-Net) have been observed, taking the SK equivalent SW (sliding window) network as the baseline. The label throughput is up to one megapixel per second.

The analyzed neural networks have distinctive characteristics that apply during training or processing, and not every data set is suitable to every architecture. The quality of the predictions is assessed on two neural tissue data sets, of which one is the ISBI 2012 challenge data set. Two different loss functions, Malis loss and Softmax loss, were used during training.

The whole pipeline, consisting of models, interface and modified Caffe library, is available as Open Source software under the working title *Project Greentea*.

Acknowledgements

University of Zurich, Institute of Neuroinformatics

Firstly I would like to express my gratitude to my supervisor Dr. Jan Funke for his guidance, motivation and the opportunity to visit HHMI Janelia in Ashburn, Virginia, USA.
I also thank my supervisor Prof. Dr. Angelika Steger for collaborating with the Institute of Neuroinformatics, which made this research project possible.
I thank Stephan Gerhard and Julien Martel for interesting discussions about neural networks and this technical report.

Howard Hughes Medical Institute, Janelia

Besides my advisors I would like to thank Dr. Srinivas Turaga and Dr. Stephan Saalfeld for their collaboration at HHMI Janelia, which inspired me to extend the scope of my research and gave me insight into the applications of neural networks for image segmentation in connectomics. Besides this, Janelia has the nicest campus of all research institutes that I have seen so far.

AMD (Advanced Micro Devices)

I would like to thank AMD and especially Roy Taylor, Greg Stoner and Bruno Stefanizzi for the generous hardware sponsoring, which empowered a lot of the development on the Caffe library and enabled me to use neural network models beyond what is possible on regular hardware. I also thank Timmy Liu for his assistance and development of clBLAS and Dr. Ing. Hervé Chevanne for providing drivers and support for the AMD GPUs.
Being a fan of AMD hardware and using their devices for over ten years, it was a pleasure for me to work together with AMD engineers and using their newest hardware and software technology for research.

Family

Last but not the least, I would like to thank my family, my parents and my sister, for supporting me during my Bachelor studies at ETH Zurich.

Contents

Chapter 1

Introduction

1.1 Convolutional Neural Networks

Convolutional neural networks are forward-backward neural networks that are mostly based on convolutions with machine learnable kernels, pooling operations and element-wise non-linear activation functions. The networks can be employed for various image classification and object recognition tasks. A prominent example is the ImageNet / AlexNet [1] for object recognition. Recent networks [2] can have very many, in this case over 20, layers and millions of learnable parameters.

This work is focused on classifying biomedical data, in particular neural tissue electron microscopy images (see Chapter 2). The challenge with this kind of data sets is that training data is more scarce than with data sets that can be generated from everyday pictures such as handwritten letters or online collections of images. Annotating ground truth for neural tissue images is a lot of manual work, as every single pixel has to be labeled.

Consequently, improving training speeds is not a primary objective to optimize for. The data that has to be processed with a trained model afterwards however can easily reach terabyte-scale. It is therefore crucial to develop networks that are as efficient as possible in the forwarding step. This work presents three such efficient pixel classification networks.

When training, the classical mislabeling objectives such as Softmax or Cross-entropy loss might not be the most useful for pixelwise classifications of biological images, and using spatial context information to generate an error signal to train the neural networks can perform better. Therefore, this research also considers different training methods including the Malis [3] criterion.

To relate the objectives of this technical report with the title, it needs to be dissected into its components:

- *Efficient Convolutional Neural Networks* means the network models analyzed and designed are as efficient as possible in getting the task done - in this case, pixelwise classification of electron microscopy neural tissue images.
- *Pixelwise Classification*, as opposed to image classification, aims to propose a label to each pixel in a given image. It can also be seen as many separate

image classifications of small patches in a bigger image. This gives rise to new optimization possibilities as the contexts for the predictions overlap spatially.

- *Heterogeneous Hardware Systems* means the network models used should also run as efficient as possible on a variety of compute devices. This objective makes efficient neural networks more accessible to users and allows to use existing hardware and clusters to get segmentation tasks with neural networks done.

1.2 Caffe Library

Caffe stands for Convolutional Architecture for Fast Feature Embedding [4]. It is a state-of-the-art neural network library that has been heavily optimized for the use with nVidias CUDA technology. In many cases, the library is therefore already very efficient using certain GPUs. What was missing until now [5] is fast CPU support (the current CPU backend is mostly single threaded) and support for GPUs and accelerator devices from AMD and Intel. The library is still under development and has a large community [6].

Network models and trained weights (usually called model zoo) can be shared in Google's prototxt (network and learning configurations) and protocol buffer (trained weights and solver states) format.

The library is typically used on the command line with the Caffe binary or through a python (Pycaffe) interface (see Figure 1.3). For more advanced and intrusive interfaces, C++ interfaces can be programmed on top of the library.

All models, utilities and backend additions programmed for this project are based on and around Caffe. The changes to the library are documented in Chapter 5.

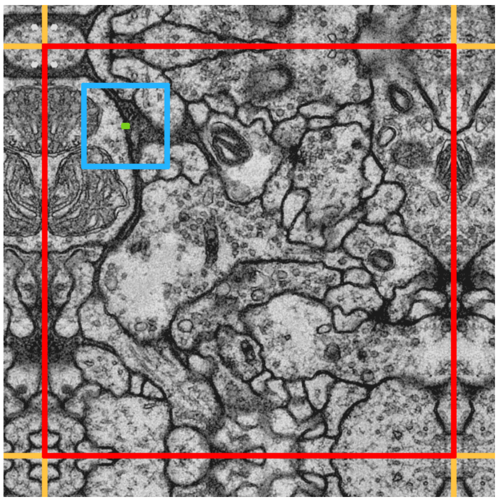
1.3 Pixelwise Classification

Pixelwise classification means labeling each pixel in an image based on a local context around the pixel. Figure 1.1 shows how this works with sliding window networks that outputs a single pixel per input tile of size $v + w = 101 + 1 = 102$.

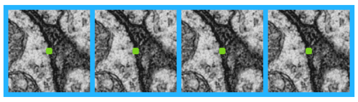
While minibatch processing can output many pixels at once, this is still inefficient (see Sections 3.2 and 6.5). The work by Hongsheng Li *et al*. [7] allows to make existing SW networks more efficient while giving identical prediction results (see Section 3.3.1).
Alternatively, fully convolutional models (U and partially also USK) directly output a bigger patch, as depicted in Figure 1.2. This method of training and processing is called patch-based ($n = 1$, $w > 1$), as opposed to minibatch-based ($n > 1$, $w = 1$). A combination of both ($n > 1$, $w > 1$) is possible but only useful when the images in the data set can not be tiled with large $w \gg 1$ (see Section 6.4).

Minibatches can still have advantages during training (see Section 4.3) because every element in the minibatch can be picked independently. During processing, networks that output large patches are always performing better.

In all cases, the images have to be extended (padded) by mirroring on the borders by $\frac{v}{2}$ pixels on each side if every pixel of the image is to be labeled.

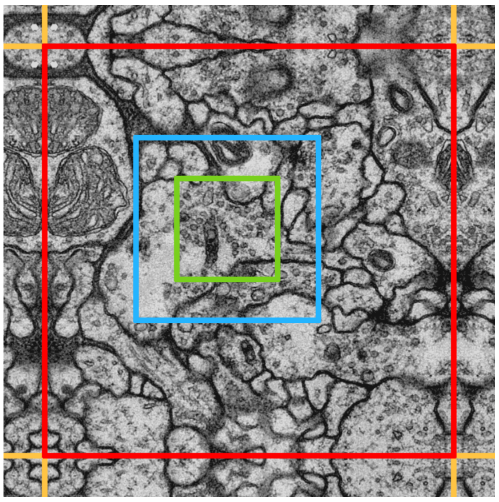


(a) Input image to classify with border mirroring to extend the image. The green rectangle is a 4 by 1 pixel area to be labeled.

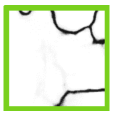


(b) Generated minibatch input of size $n = 4$ and with a context of 102 by 102 pixels. The output classification for this input will be 4 by 1 pixels. The individual images in the minibatch are only shifted by one pixel each. The data overlaps and is copied into the network redundantly.

Figure 1.1: Pixelwise image classification based on sliding window architectures. (Raw image source: ssTEM [8], [9]). The surrounding context (blue rectangles) is what determines the labeling decision of the neural network.



(a) The green area to be labeled is 128 by 128 pixels ($w = 128$). The green patch with the blue context padding ($v = 101$) is directly what the networks take as input.



(b) The data is passed through the network as large a tile of size $w + v$ by $w + v$ instead of a minibatch. There is no overlapping data being passed through the network redundantly and no duplicated convolution and pooling operations are carried out. The output prediction is a large patch ($w = 128$) instead of a stride of pixels from a minibatch (as in Figure 1.1).

Figure 1.2: Pixelwise image classification based on strided kernel and fully convolutional architectures. (Raw image source: ssTEM [8], [9]).

1.4 Existing Work

This technical report is based on the following existing work:

- SW (sliding window) network designed by Julien Martel [10], not published. The architecture is trimmed for segmenting the data set DS1.
- Strided kernel convolution and pooling kernels by Hongsheng Li *et al.* [7]. This is the fundamental approach in speeding up existing SW networks.
- Malis criterion, first introduced by Srinivas Turaga *et al.* [3]. The criterion supports an alternative way of training neural networks through affinity graphs, which is very specific and useful on biomedical data, where areas are separated by background borders.
- The Open Source Caffe library maintained by the Berkeley Vision and Learning Center [6], [4].
- U network designed by Ronneberger *et al.* [2]. This model is also optimized for biomedical images and especially the data set DS2 (ISBI 2012 [11]).
- N-dimensional convolution kernels by Jeff Donahue [12].
- Segmentation evaluation scripts of the ISBI 2012 challenge [11], [13].

1.5 New Contributions

An overview of new contributions to the Caffe landscape in terms of models, utilities and library changes is given in Figure 1.3. This work tackles the given problem on all levels - from using efficient BLAS libraries over backend development and frontends for easy use to new network models.

On the side of neural network models, this project introduces two new neural network architectures, the SK-Net (Section 3.3) and USK-Net (Section 3.5).

A meta-analysis of the three efficient networks (SK, U, USK) is given based on:

- Differences and characteristics of the network designs (Chapter 3).
- Computational cost and efficiency (Chapter 6).
- Image segmentation quality, assessed both numerically and visually (Chapter 7) for the typical foreground-background two label classification.

In order to be able to train the network models easily on various data sets, the Caffe Neural Tool (Chapter 4) has been developed.

The Caffe library (see Chapter 5) has been extended with new layers and adaptions for compability with the new backend. The new layers also affect the functionality of the CUDA backend. OpenCL backend development (see Section 5.4) was mostly focused on versatility and completeness, so that CPUs and all kinds of compute devices can be used on all network models. This includes compability to three different BLAS libraries: clBLAS, ViennaCL-BLAS and cBLAS.

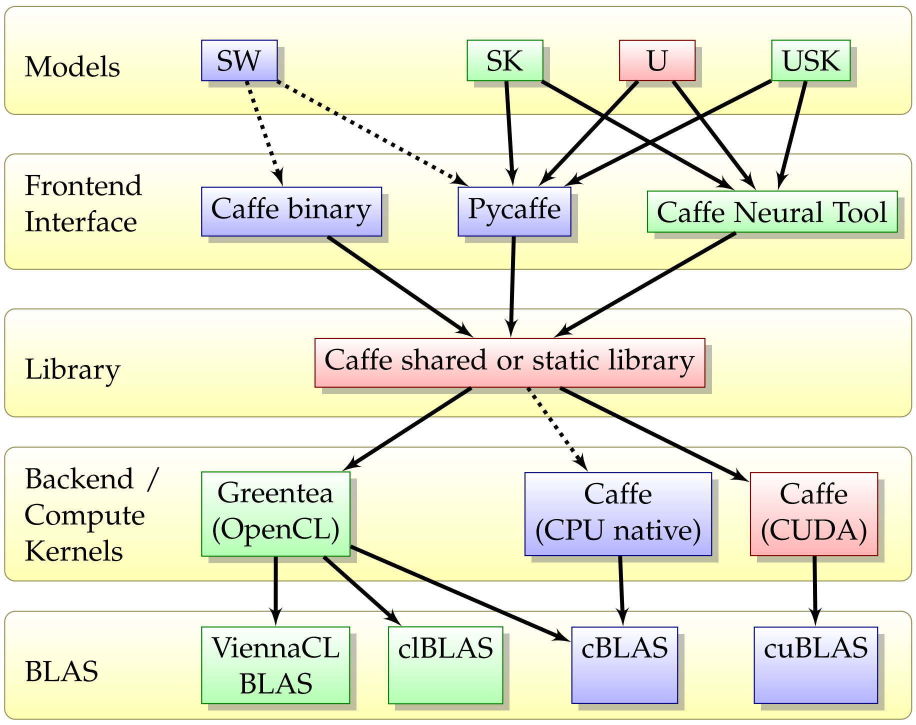


Figure 1.3: *Project Greentea* overview. Green boxes denote completely new additions to the Caffe landscape. Red boxes are parts that have been re-implemented or adapted from existing work to fit the needs of this project. Blue parts are mostly unchanged from existing work. Dashed arrows denote deprecated and only partially supported combinations.

This report is also providing an introduction into segmentation and pixelwise classification with neural networks. It contains all the details necessary to understand existing models and readers should be able to easily design their own neural networks based on the findings of this research project.

The combination of the new contributions and the Caffe library infrastructure is summarized under the working title *Project Greentea*. However, *Greentea* also stands for the new OpenCL backend architecture which has been optimized for high flexibility (see Section 5.4).

Greentea was a name of my choice because I simply prefer greentea over coffee (Caffe). Greentea can not be used as an abbreviation like it is the case with Caffe, but as greentea is supposed to be good for the brain, using it for a neural network machine learning toolset seems to be appropriate.

1.6 Terminology

The most used symbols and abbreviations in the report:

- *Forwarding, processing*: Computing data through a neural network from input to output.
- *Backwarding, training, backpropagation*: Computing neural network gradients (diff maps) in the backward direction and updating the network weights.
- *Data blob*: Memory blob containing feature maps of forward processing in the neural network.
- *Diff blob*: Memory blob containing the differential / error signal map during backpropagation.
- *BLAS*: Basic Linear Algebra Subprograms. Includes functions such as efficient matrix multiplications.
- *DS1*: Data set 1, see Section 2.1.
- *DS2*: Data set 2, see Section 2.2.
- *SW*: Sliding window networks for pixelwise classification, see Section 3.2.
- *SK*: Strided kernel networks for pixelwise classification, see Section 3.3.
- *U*: Ronneberger *et al.* [2] network architecture, see Section 3.4.
- *USK*: Network architecture combining SK and U aspects, see Section 3.5.
- f: Number of feature maps after (f_{out}) and before (f_{in}) a network layer.
- w: Size (in each dimension) of a feature map in a network layer. When not indexed or otherwise noted, it refers to the output size of a network layer.
- v: Size of the total network input padding (context).
- p: Network layer padding.
- s: Network layer stride.
- k: Network layer kernel size.
- d: Network layer kernel stride.
- L: Set of layers with layers $l \in L$.
- B: Set of memory blobs with blobs $b \in B$.
- W: Set of network weights, $|W|$ denotes the number of weights.
- M: Device or host memory usage.
- q: Number of queues in the Caffe OpenCL backend.
- n: Network minibatch size.
- A: Affinity graph data, ΔA denotes the affinity graph diff.
- I: Pixel image data, ΔI denotes the image diff.

Some symbols are used differently in some sections of the report and are explained in-place.

Chapter 2

Datasets

2.1 DS1 - Segmented anisotropic ssTEM dataset of neural tissue

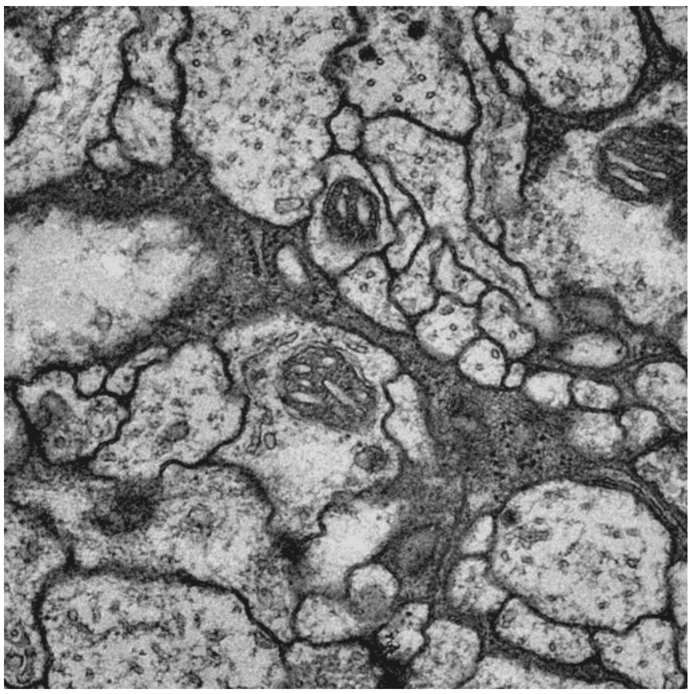
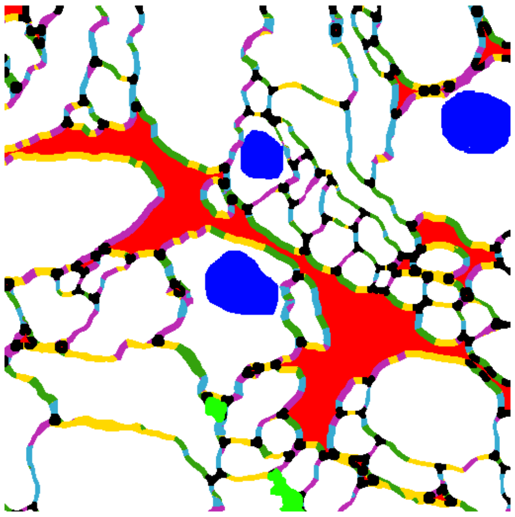


Figure 2.1: DS1 ssTEM raw image, 512 by 512 pixels of image 2 (right upper corner) (Source: ssTEM [8], [9]).

This data set shows neural tissue from a Drosophila larva ventral nerve cord and was acquired using serial section Transmission Electron Microscopy at HHMI Janelia Research Campus [8]. The training data consists of 20 images of 1024 by 1024 pixels raw ssTEM and the corresponding segmentation. It is segmented into nine different labels, which are consolidated into foreground and background for two label training and evaluation (see Section 7.2):

- #0: Horizontal cell membranes - background
- #1: +45° to vertical cell membranes - background
- #2: Vertical cell membranes - background
- #3: -45° to vertical cell membranes - background
- #4: Cell membrane junctions - background
- #5: Glia cells - background
- #6: Mitochondria - foreground
- #7: Synapses - background
- #8: Cell interior - foreground

The idea behind label consolidation in this way is that the network, during training, can learn the separation borders between neural cells. This is especially important with the Malis criterion loss (see Section 5.3.2), which can only segment into foreground and background. With the Softmax loss, it is also possible to let the network learn the labels separately and combine them accordingly afterwards. The network has to learn the features separately either way, as the membrane for example depends on different orientations in the convolution filters. This means a network can be trained on two labels only and afterwards, all nine labels can be extracted by applying a short fine-tuning training phase to the network.


(a) All 9 labels, the blue mitochondria and white cell interiors are the foreground.

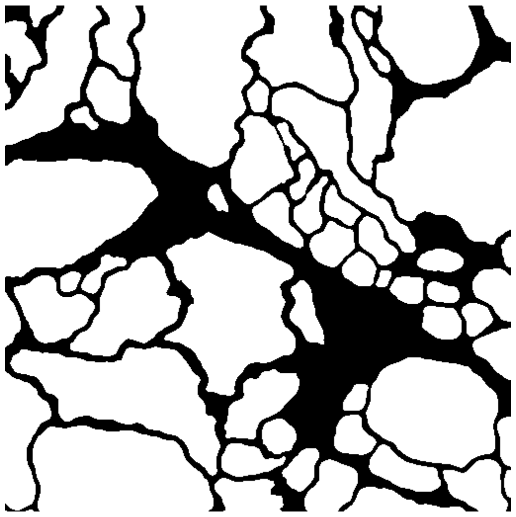

(b) Consolidated labels, background-foreground segmentation.

Figure 2.2: DS1 ssTEM label images, 512 by 512 pixels of image 2, corresponding to the raw image in Figure 2.1 (Source: ssTEM [8], [9]).

For evaluation, the training data set has been split into training and testing data because of the lack of a segmented test data stack:

- Train images: 0, 1, 3, 4, 5, 6, 8, 9, 10, 11, 13, 14, 15, 16, 18 and 19

- Test images: 2, 7, 12 and 17

As all slices of the data set are very similar, cross validation by splitting the data set in different ways was not applied. The total amount of pixels for training is therefore $16 \cdot 1024^2 \approx 16\,\mathrm{Mpixel}$.

The data set with the corresponding test and train scripts is available in the Caffe Neural Models repository [9] as *dataset_01*.

2.2 DS2 - ISBI 2012 dataset of neural tissue

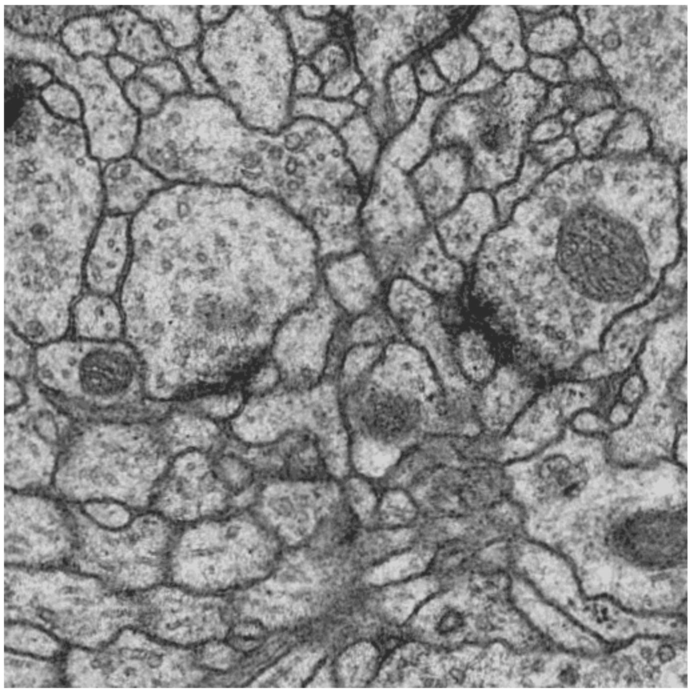


Figure 2.3: DS2 raw image, 512 by 512 pixels of training image 1 (Source: ISBI challenge [11], [9]).

This data set is from the ISBI 2012 challenge [11], [14], [15]. The raw images and corresponding segmentation images for training are 512 by 512 pixels. The neural tissue features have a similar scale to the data set DS1. The raw images have a bit less contrast and are more fuzzy.

The training set has 30 images, which gives a total of $30 \cdot 512^2 \approx 7.8\,\mathrm{Mpixel}$, which is about half as much as on DS1.

The test data used on DS2 is a separate stack of 30 images of size 512 by 512 pixels. For those images, segmentation ground truth is not available for public download, thus the evaluation reported in Section 7.3 is solely based on the reports of the official ISBI 2012 evaluation [11], which is still open for new results.

For both stacks, the data spans 2 x 2 x 1.5 microns with a resolution of 4 by 4 by 50 nm/pixel [11].

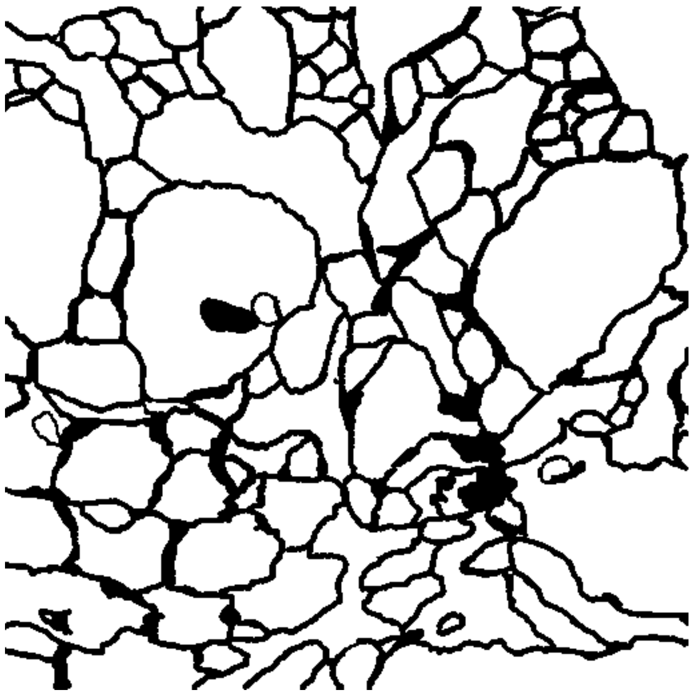


Figure 2.4: DS2 label image, 512 by 512 pixels of training image 1, corresponding to the raw image in Figure 2.3 (Source: ISBI challenge [11], [9]).

The data set with the corresponding test and train scripts is available in the Caffe Neural Models repository [9] as *dataset_02*.

Chapter 3

Models

3.1 Introduction

This chapter describes how the network architectures were set up for training and processing the data sets DS1 and DS2. They were configured for two label classification, however also nine labels were tested and even more could be learned through the Softmax loss. With Malis loss (Section 5.3.2), only foreground and background separation is implemented.

For all networks architectures, no special striding or padding was used. Therefore the striding parameter is set to be $s = 1$, except for downsampling and sliding window pooling operations, which use $s = k$ (stride matches kernel size). The padding is always $p = 0$. All networks use square size filters and feature maps, which simplifies the descriptions to just one value per parameter. The models can be generalized to arbitrary dimensions with different sizes in each dimension. At the time of the project, SK networks up to 6 dimensional can be configured with the modified Caffe library provided [5].

The mini batch size used for processing in the SW network is $n = 256$, but the choice is arbitrary and only limited by GPU memory. With SK, U and USK networks, a minibatch size of one ($n = 1$) is sufficient to reach 100% GPU utilization, and often the GPU memory is not sufficient for bigger mini batch sizes (see Section 6.4). It is rather useful to increase the network output size than using minibatches, if the data set allows it by having big enough input pictures.

All efficient networks presented here (excluding the sliding window) can be run with almost any size of output prediction maps. Only constraints on even divisibility with pooling operations, as well as the size constraints given by convolutions and strided kernels (see Section 3.3) have to be met. The networks can therefore be run on different input image sizes, depending on memory requirements (see Section 6.4) and data set image size, without re-design and re-training of the networks. The results are numerically identical in this case.

The total padding (v) of the networks is a characteristic of the network itself and can not be changed without re-design and re-training. It describes the amount of context considered for each pixel prediction.

To fit the networks to data sets with features of different scales than DS1 and DS2 (see Chapter 2), it may be necessary to adapt kernel sizes and layers to get good predictions after training. Here, the networks are configured so that a big mitochondrion (about 100 by 100 pixels) would fit into the context of a pixel prediction centered on the mitochondrion.

3.2 Sliding Window (SW-Net)

Sliding window networks classify an image by taking a pixel and a border padding v of some size around it as input and classify the center pixel by running the patch through a neural network. Then the next pixel is labeled by shifting the window patch by one pixel, classifying the neighboring pixel of the first one. The pixels can also be processed in a minibatch to increase GPU utilization and amortize direct memory access transfer times (from host to device memory). This is still very inefficient as most of the context of two neighboring pixels overlaps and the same filters are applied over the whole context. The redundant computations can be reduced for a patch of input pixels by using SK networks.

The sliding window network described here was developed by Julien Martel [10]. It is the baseline for calculating the speedups obtained with the SK, U and USK networks. The structure of the SW network was also used when designing the SK network and the core of the USK network. The reason why this was used as a basis is that it has already been trained on the data set DS1 and had good results. Learning rate, weight decay and training parameters were available. However, no numerical evaluation or publication about the network architecture exists.

Layer	**Type**	w	f_{in}	f_{out}	k	s
data	MemoryData	100	3	3	1	1
conv1 + relu1	Convolution + ReLU	94	3	48	7	1
pool1	Max Pooling	47	48	48	2	2
conv2 + relu2	Convolution + ReLU	43	48	128	5	1
pool2	Max Pooling	22	128	128	2	2
conv3 + relu3	Convolution + ReLU	20	128	192	3	1
pool3	Max Pooling	10	192	192	2	2
ip1 + relu4	InnerProduct + ReLU	1	192	1024	10	1
ip2 + relu5	InnerProduct + ReLU	1	1024	512	1	1
ip3	InnerProduct	1	512	2	1	1
prob	Softmax	1	2	2	1	1

Table 3.1: Network setup for the SW model.

The sliding window network has not been evaluated in-depth in terms of benchmarking and quality assessment. This is because the Caffe Neural Tool (see Chapter 4) used for detailed benchmarking and processing does not work with minibatches in its current form. Its segmentation performance should however be numerically equal to the SK network, which is derived from the SW network.

Any differences would be due to different training methods (patches on SK versus minibatches on SW). Information about how single layers speed up from SW to SK networks (both theory and experimental) can be found in the work of Hongsheng Li *et al*. [7].

For weight initialization, the SW network uses random initialization drawn from a Gaussian distribution with $\mu = 0$ and $\sigma = 0.01$.

3.3 SK-Net

3.3.1 Converting SW Networks to SK

Hongsheng Li *et al*. [7] provide a pseudo code (page 4) on how to convert a sliding window network to a strided kernel network. However, it is incomplete on consistency checking, kernel sizes and feature map output sizes. Also, the theory of converting inner product (fully connected) layers is not described. Therefore I provide a more complete version (see Algorithm 1), although without considering padding and striding. This enforces $s = 1$ and $p = 0$ in all layers of the SK network. Each data dimension (width, height, depth) can be processed separately for the kernel size (k), kernel stride (d) and output dimension (w). The algorithm is able to convert networks and find consistency issues fully automatized.

For Caffe prototxt network configurations, only the kernel size k and kernel stride d have to be provided. Output dimensions will be computed on the fly, given the network input size $w_{\text{SK}}^{(0)}$. Padding and striding parameters can be left away, they will default to the correct values.

Algorithm 1 assumes that $|L_{\text{SW}}| = |L_{\text{SK}}| = N$, not taking into account the input data layer, which is at $i = 0$.
If a layer type is not handled in a special *if*-case, it is handled by using the exact same configuration as in the original network. This fails however if the layer does anything other than an element-wise operation (this implies $k = 1$), because of the kernel stride $d > 1$. During the conversion, the initial input size $w_{\text{SK}}^{(0)}$ is equal to what the SW network used. Afterwards, an arbitrary input size $w_{\text{SK}}^{(0)} \geq w_{\text{SW}}^{(0)}$ can be used during training and processing, and the output will be of size $w_{\text{SK}}^{(N)} = w_{\text{SK}}^{(0)} - w_{\text{SW}}^{(0)} + 1$.

Using $w_{\text{SW}}^{(0)} = w_{\text{SK}}^{(0)}$ also helps to prove that the results stay numerically the same. In this case, the strides introduced by pooling operations will be implicitly ignored. They will not be taken into account at the first inner product layer (*ip1*), which will span the whole feature map size, because the external kernel size is

$$w_{\text{SK}}^{(i-1)} = (k_{\text{SK}}^{(i)} - 1)d_{\text{SK}}^{(i)} + 1 \Longrightarrow w_{\text{SK}}^{(i)} = 1 \tag{3.1}$$

and

$$w_{\text{SW}}^{(i-1)} = k_{\text{SW}}^{(i)} = k_{\text{SK}}^{(i)} \Longrightarrow w_{\text{SW}}^{(i)} = 1 \tag{3.2}$$

at that layer. This property can be visualized as well, as in Figure 3.1, where the first inner product layer (*ip1*) has a kernel size of $k_{\text{SK}} = 3 = w_{\text{SW}}^{(9)}$.

For the originally inner product (fully connected) layers in SW-Net, which are now normal convolutions, (*ip1* to *ip3*) there are no computational savings compared to the SW network anymore. This is clear from the observation that the *ip1* layer isolates the context of each pixel and no overlappings in the feature maps exist after this layer.
The number of input (f_{in}) and output (f_{out}) feature maps remains exactly the same for SW and SK networks in all layers.

Algorithm 1 Convert SW-Net to SK-Net

```
1:  procedure CONVERT
2:      $\forall i \in [1, N].s_{\text{SK}}^{(i)} \leftarrow 1$
3:      $\forall i \in [1, N].p_{\text{SK}}^{(i)} \leftarrow 0$
4:      $w_{\text{SK}}^{(0)} \leftarrow w_{\text{SW}}^{(0)}$
5:      $d_{\text{temp}} \leftarrow 1$
6:      for $i = 1; i \leq N; i \leftarrow i + 1$ do
7:          if $l_{\text{SW}}^{(i)}$ = convolution then
8:              $l_{\text{SK}}^{(i)} \leftarrow$ convolution SK
9:              $k_{\text{SK}}^{(i)} \leftarrow k_{\text{SW}}^{(i)}$
10:             $d_{\text{SK}}^{(i)} \leftarrow d_{temp}$
11:             $w_{\text{SK}}^{(i)} \leftarrow w_{\text{SK}}^{(i-1)} - (k_{\text{SK}}^{(i)} - 1) \cdot d_{\text{SK}}^{(i)}$    ▷ $w_{\text{SW}}^{(i)} \leftarrow w_{\text{SW}}^{(i-1)} - (k_{\text{SW}}^{(i)} - 1)$
12:         else if $l_{\text{SW}}^{(i)}$ = pooling then
13:             if $w_{\text{SW}}^{(i-1)} \mod k_{\text{SW}}^{(i)} \neq 0 \vee k_{\text{SW}}^{(i)} \neq s_{\text{SW}}^{(i)}$ then return error
14:             $l_{\text{SK}}^{(i)} \leftarrow$ pooling SK
15:             $k_{\text{SK}}^{(i)} \leftarrow k_{\text{SW}}^{(i)}$
16:             $d_{\text{SK}}^{(i)} \leftarrow d_{temp}$
17:             $w_{\text{SK}}^{(i)} \leftarrow w_{\text{SK}}^{(i-1)} - (k_{\text{SK}}^{(i)} - 1) \cdot d_{\text{SK}}^{(i)}$    ▷ $w_{\text{SW}}^{(i)} \leftarrow \left\lceil \frac{w_{\text{SW}}^{(i-1)}}{k_{\text{SW}}^{(i)}} \right\rceil$
18:             $d_{temp} \leftarrow d_{temp} \cdot k_{\text{SK}}^{(i)}$
19:         else if $l_{\text{SW}}^{(i)}$ = inner product then
20:             $l_{\text{SK}}^{(i)} \leftarrow$ convolution SK
21:             $k_{\text{SK}}^{(i)} \leftarrow w_{\text{SW}}^{(i-1)}$    ▷ $k_{\text{SW}}^{(i)} = w_{\text{SW}}^{(i-1)}$ is implicit
22:             $d_{\text{SK}}^{(i)} \leftarrow d_{temp}$
23:             $w_{\text{SK}}^{(i)} \leftarrow w_{\text{SK}}^{(i-1)} - (k_{\text{SK}}^{(i)} - 1) \cdot d_{\text{SK}}^{(i)}$    ▷ $w_{\text{SW}}^{(i)} \leftarrow 1$
24:             $d_{temp} \leftarrow 1$
25:         else
26:             if $k_{\text{SW}}^{(i)} > 1$ then return error
27:             $l_{\text{SK}} \leftarrow l_{\text{SW}}$
28:             $w_{\text{SK}}^{(i)} \leftarrow w_{\text{SK}}^{(i-1)}$    ▷ $w_{\text{SW}}^{(i)} \leftarrow w_{\text{SW}}^{(i-1)}$
29:     if $d_{temp} = 1$ then return success
30:     else return error
```

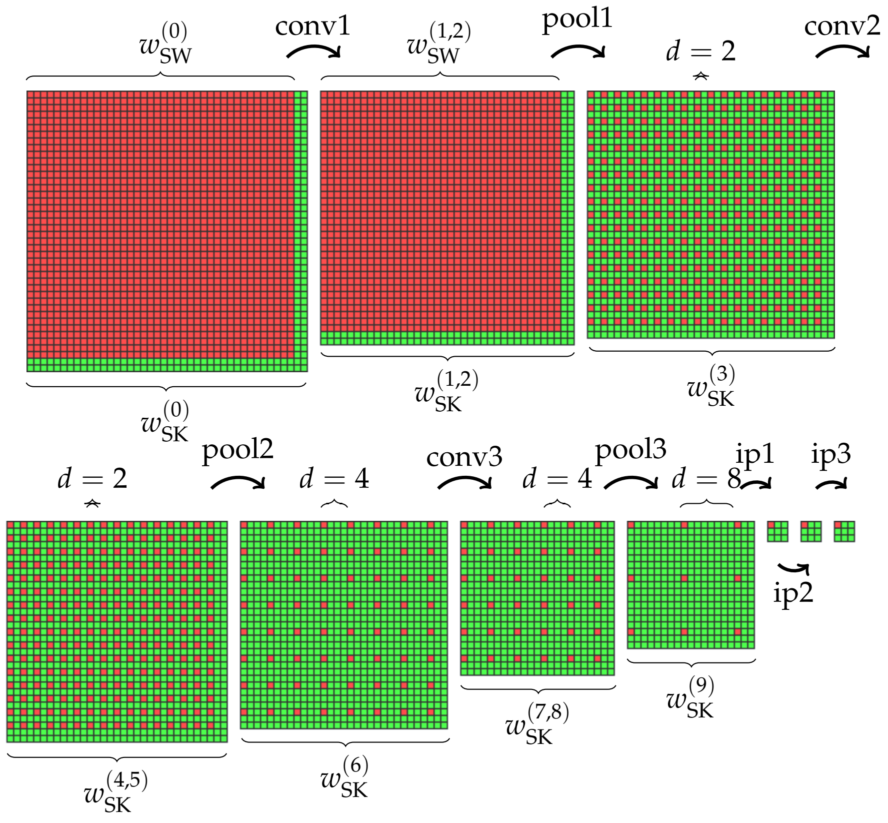


Figure 3.1: SK (strided kernel) feature maps. The kernel and feature map sizes used here are smaller than in the actual SK network in order to fit it into a reasonably sized figure. The input size is $w_{SK}^{(0)} = w_{SW}^{(0)} + 2$ and consequently the output is 3 by 3 pixels as inferred from Algorithm 1. The red squares trace the information flow in the original SW network. Convolutions and their ReLU activations are not displayed separately.

Observations on Algorithm 1:

- *Line 13*: Implies only downsamlpling pooling layers can be converted. Pooling with $s = 1$ instead of $s = k$ would need to be handled like convolutions in terms of kernel and output size, and would not change the kernel stride d. This has not been assessed further.
- *Line 17*: Caffe can handle downsampling poolings that overlap the border in SW networks, which causes implicit zero padding. This is not allowed in SK networks, therefore *Line 13* checks if $w \mod k = 0$. Continuing without this check would cause the new strided pooling layer to overlap on data that would normally be separated by a stride, and also output feature maps of wrong sizes to continue.
- *Line 18*: Using downsampling is the only operation that increases the kernel stride. All other operations either keep the context local (convolutions) or have $k = 1$ (element-wise operations).
- *Lines 11, 17, 23*: Interestingly, all converted layer types with kernel sizes now

decrease the feature map sizes by the same formula. Convolution and inner product layers implicitly inherit this behavior. Pooling does this because it separates output pixels by a stride equal to the downsampling kernel size k.

- *Line 29*: Having a kernel stride $d > 1$ in the last layer implies pixels in the output feature maps are not independent from each other. In this case, the network has not been converted correctly and possibly lacks at least one inner product layer.

3.3.2 SK Network

When processing the network in Table 3.1, the condition $w_{\text{SW}}^{(i-1)} \mod k_{\text{SW}}^{(i)} \neq 0$ given by Algorithm 1 is actually violated by the second pooling layer, having 43 mod 2 $\neq 0$. It is easy to fix this by starting at the last layer of the network and computing

$$w_{\text{SW}}^{(i-1)} = k_{\text{SW}}^{(i)} w_{\text{SW}}^{(i)} \tag{3.3}$$

for pooling layers and

$$w_{\text{SW}}^{(i-1)} = (k_{\text{SW}}^{(i)} - 1) + w_{\text{SW}}^{(i)} \tag{3.4}$$

for inner product and convolution layers. The corrected network is given in Table 3.2. Converting this to SK results in the network in Table 3.3.

Layer	**Type**	w	f_{in}	f_{out}	k	s
data	MemoryData	102	3	3	1	1
conv1 + relu1	Convolution + ReLU	96	3	48	7	1
pool1	Max Pooling	48	48	48	2	2
conv2 + relu2	Convolution + ReLU	44	48	128	5	1
pool2	Max Pooling	22	128	128	2	2
conv3 + relu3	Convolution + ReLU	20	128	192	3	1
pool3	Max Pooling	10	192	192	2	2
ip1 + relu4	InnerProduct + ReLU	1	192	1024	10	1
ip2 + relu5	InnerProduct + ReLU	1	1024	512	1	1
ip3	InnerProduct	1	512	2	1	1
prob	Softmax	1	2	2	1	1

Table 3.2: Network setup for the corrected SW model.

Layer	Type	w	f_{in}	f_{out}	k	s	d
data	MemoryData	229	3	3	1	1	1
conv1 + relu1	Conv. SK + ReLU	223	3	48	7	1	1
pool1	Max Pool. SK	222	48	48	2	1	1
conv2 + relu2	Conv. SK + ReLU	214	48	128	5	1	2
pool2	Max Pool. SK	212	128	128	2	1	2
conv3 + relu3	Conv. SK + ReLU	204	128	192	3	1	4
pool3	Max Pool. SK	200	192	192	2	1	4
ip1 + relu4	Conv. SK + ReLU	128	192	1024	10	1	8
ip2 + relu5	Conv. SK + ReLU	128	1024	512	1	1	1
ip3	Conv. SK	128	512	2	1	1	1
prob	Softmax	128	2	2	1	1	1

Table 3.3: SK network configuration.

The final three inner product layers from the SW network actually become convolutions with special properties: For *ip1*, the rules stay the same as for converted convolution layers. Afterwards, *ip2* and *ip3* can have an arbitrary kernel stride d because the kernel size k only spans one pixel in each feature map. To not cause confusion, it should be configured so that $d = 1$ for those layers.

The network still has one issue, which is not nice but acceptable and the network will still work. The issue is that there is no center pixel, because

$$(w_{\text{SW}}^{(0)} \hat{=} 102) \mod 2 = 0 \tag{3.5}$$

Each patch in the original network has a context of 102 pixels. When converting to SK and classifying a patch of 128 by 128 pixels as given in Table 3.3, the padding to add in the beginning actually becomes $v = 101$ pixels. This padding can not be split up into a border around the patch to classify. To simplify this issue, a border of 51 pixels on each side is assumed and then cropped by one pixel on the bottom and right side. This results the same behavior as running a 102 by 102 pixel sliding window network across the input, which was also padded with 51 pixels in the corrected version and 50 pixels in the original version.

To estimate the number of free parameters (all convolution weights $|W|$) that can be trained in a model, the following formula is used:

$$|W| = \sum_{l^{(i)} \in L_{\text{conv.}}} f_{\text{in}}^{(i)} \cdot f_{\text{out}}^{(i)} \cdot (k^{(i)})^2 \tag{3.6}$$

Using the values in Table 3.3, this gives $|W| \approx 20.5 \cdot 10^6$ parameters, of which most ($\approx 19.6 \cdot 10^6$) are within the *ip1* layer.

For weight initialization, the SK network uses random initialization drawn from a Gaussian distribution with $\mu = 0$ and $\sigma = 0.01$.

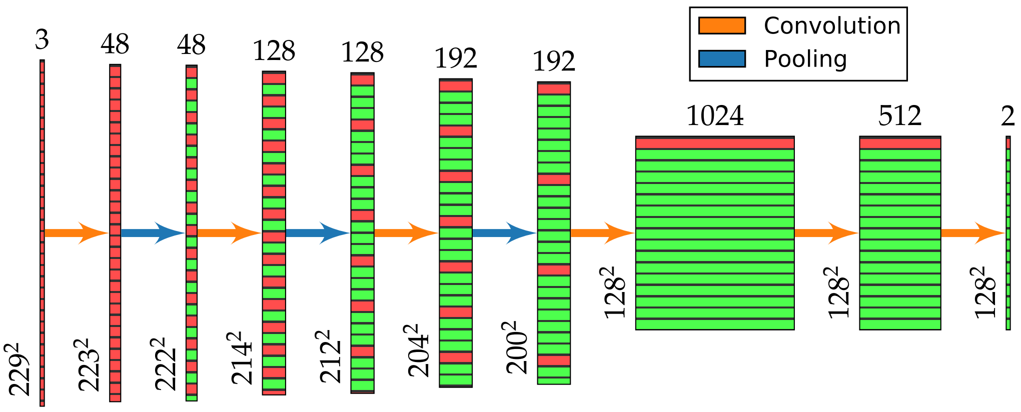


Figure 3.2: SK network configuration visualization. Green-red striped blocks represent feature maps with a kernel stride ($d > 1$). Vertical numbers represent the size of the feature maps while the horizontal numbers represent the number of feature maps.

A directed acyclic graph representation of the network can be found in the appendix A.1.

3.4 U-Net

The U-Net presented here is the network configuration as described in the Ronneberger *et al*. paper [2]. Table 3.4 describes the network in the same style as Table 3.3 for the SK network in order to compare them. The layer names are chosen in the same style as with SW and SK networks. The U-Net has contracting and expanding sections:

- Contracting: Two convolutions followed by one max pooling layer.
- Expanding: Deconvolution followed by a convolution to reduce the number of feature maps, a mergecrop and two convoluton layers.

The source code for running U-Net as well as the prototxt configuration files were not available for download at the time of this project, thus the network presented here, which is my own interpretation, might differ from the original design. The paper does not give all details, such as how the *MergeCrop* layer and *Upconvolution* work. The configurations of this U-Net included in the Caffe Neural Models [9] are therefore incompatible to the original work [2]. Segmentation results should be comparable.

Layer	**Type**	w	f_{in}	f_{out}	k	s
data	MemoryData	572	3	3	1	1
conv1 + relu1	Convolution + ReLU	570	3	64	3	1
conv2 + relu2	Convolution + ReLU	568	64	64	3	1
pool1	Max Pooling	284	64	64	2	2
conv3 + relu3	Convolution + ReLU	282	64	128	3	1

conv4 + relu4	Convolution + ReLU	280	128	128	3	1
pool2	Max Pooling	140	128	128	2	2
conv5 + relu5	Convolution + ReLU	138	128	256	3	1
conv6 + relu6	Convolution + ReLU	136	256	256	3	1
pool3	Max Pooling	68	256	256	2	2
conv7 + relu7	Convolution + ReLU	66	256	512	3	1
conv8 + relu8	Convolution + ReLU	64	512	512	3	1
pool4	Max Pooling	32	512	512	2	2
conv9 + relu9	Convolution + ReLU	30	512	1024	3	1
conv10 + relu10	Convolution + ReLU	28	1024	1024	3	1
upconv1	Deconvolution	56	1024	1024	2	2
conv11	Convolution	56	1024	512	1	1
mergecrop1	MergeCrop	56	512 + 512	1024	1	1
conv12 + relu11	Convolution + ReLU	54	1024	512	3	1
conv13 + relu12	Convolution + ReLU	52	512	512	3	1
upconv2	Deconvolution	104	512	512	2	2
conv14	Convolution	104	512	256	1	1
mergecrop2	MergeCrop	104	256 + 256	512	1	1
conv15 + relu13	Convolution + ReLU	102	512	256	3	1
conv16 + relu14	Convolution + ReLU	100	256	256	3	1
upconv3	Deconvolution	200	256	256	2	2
conv17	Convolution	200	256	128	1	1
mergecrop3	MergeCrop	200	128 + 128	256	1	1
conv18 + relu15	Convolution + ReLU	198	256	128	3	1
conv19 + relu16	Convolution + ReLU	196	128	128	3	1
upconv4	Deconvolution	392	128	128	2	2
conv20	Convolution	392	128	64	1	1
mergecrop4	MergeCrop	392	64 + 64	128	1	1
conv21 + relu17	Convolution + ReLU	390	128	64	3	1
conv22 + relu18	Convolution + ReLU	388	64	64	3	1
ip1	Convolution	388	64	2	1	1
prob	Softmax	388	2	2	1	1

Table 3.4: U network configuration.

The U-Net architecture has $|W| \approx 29 \cdot 10^6$ parameters, using Equation 3.6 and the corresponding values in Table 3.4. Thus there are about 30% more parameters than in SK-Net. The number of weights rises towards the middle of the U network, due to having the same convolution kernel size (3 by 3) throughout the network and more feature maps with every contracting step.

For weight initialization, the U network uses random initialization drawn from a Gaussian distribution with $\mu = 0$ and $\sigma = \sqrt{2/(f_{\text{in}} \cdot k^2)}$. This is the same as used in the original paper [2]. With $\sigma = 0.01$ as in SK and SW models, the network could not be trained properly.

The upconvolutions are set to nearest neighbor interpolation, which causes each pixel in the f_{in} maps to fill exactly four pixels (two by two) in the f_{out} maps.

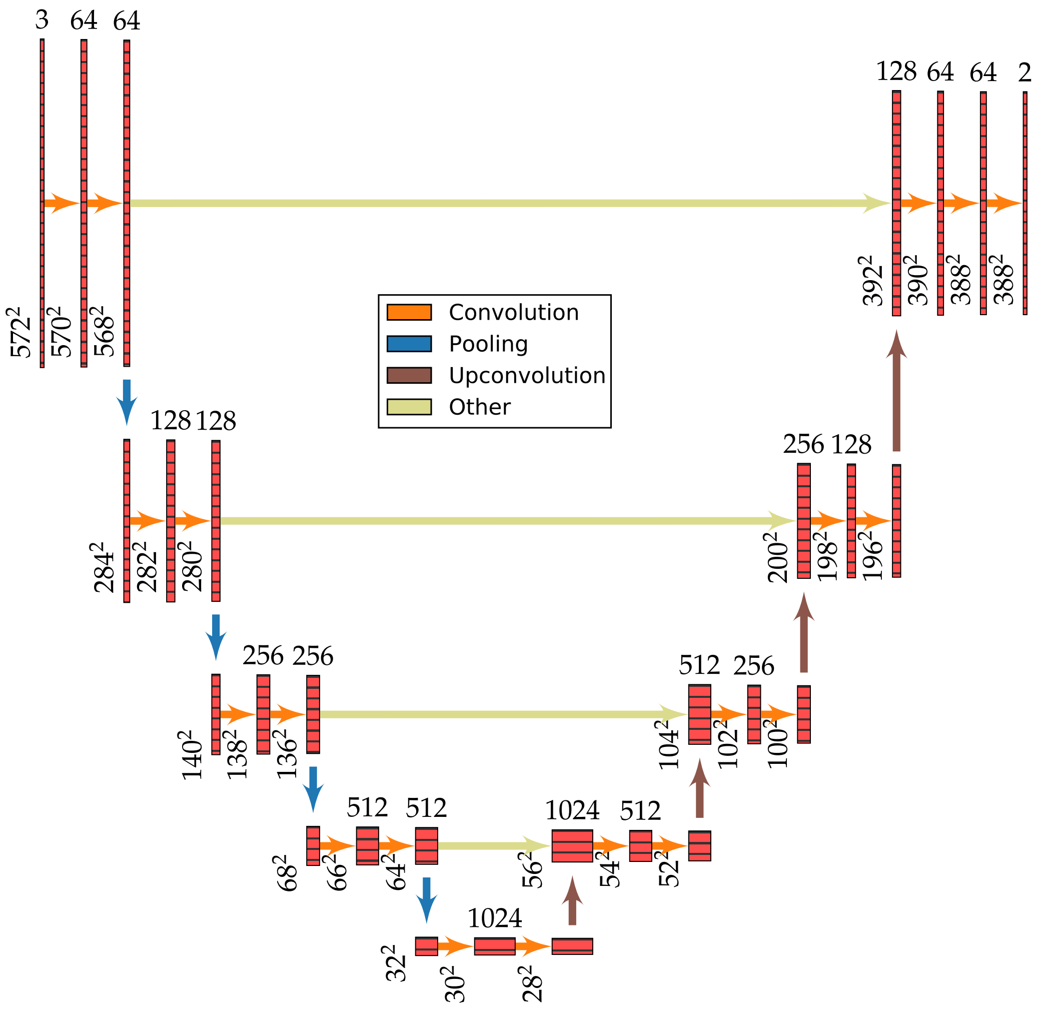


Figure 3.3: U network configuration visualization. Vertical numbers represent the size of the feature maps while the horizontal numbers represent the number of feature maps.

Figure 3.3 represents the network in a style identical to the one used by Ronneberger *et al.* [2]. A directed acyclic graph representation of the network can be found in the appendix A.2.

3.5 USK-Net

The USK network architecture combines ideas from the U and SK models. The majority of the convolutions and therefore free parameters can be trained on downsampled feature maps by using one or more contracting path steps (and their expanding counterpart) from the U-Net.
After one contracting step, the same sequence of layers as in the SK network is

applied, however with different kernel sizes to match the sizes of the inputs and outputs as required by the U subnetwork. The network has been configured so that:

- The network outputs 512 by 512 pixel labels.
- The context considered for each pixel classification is 180 by 180 pixels.
- As a result, the network input is 692 by 692 pixels.
- The SK network part (*conv3* to *ip2*) is required to accept 344 by 344 pixels as input and outputs 258 by 258 pixels. It therefore sees a context of 86 pixels. This is on once by a factor of two downsampled feature maps.

Prior experiments with two contracting and expanding steps were not very successful, as many features of both datasets (DS1 and DS2) vanished at two times downsampling and the network also became much harder to train because it gets very deep. The network mainly relied on the filter results in the two contracting and expanding step pairs (U subnet) to classify the given input. The signals coming from the SK subnetwork were largely ignored by setting their weights close to zero in the convolution following after the first mergecrop layer.
Using only one downsampling step fixed this issue and the SK subnetwork contributed properly after training. As the SK subnetwork contains the main computational costs (see Figures 6.9 and 6.10 in Section 6.6.3) and also carries most parameters, the features have to be meaningful enough at the beginning of the subnetwork, after the first few layers of the U subnetwork.

Layer	**Type**	w	f_{in}	f_{out}	k	s	d	**Subnet**
data	MemoryData	692	3	3	1	1	1	U
conv1 + relu1	Conv. + ReLU	690	3	64	3	1	1	U
conv2 + relu2	Conv. + ReLU	688	64	64	3	1	1	U
pool1	Max Pooling	344	64	64	2	2	1	U
conv3 + relu3	Conv. SK + ReLU	339	64	128	6	1	1	SK
pool2	Max Pool. SK	338	128	128	2	1	1	SK
conv4 + relu4	Conv. SK + ReLU	332	128	128	4	1	2	SK
pool3	Max Pool. SK	330	128	128	2	1	2	SK
conv5 + relu5	Conv. SK + ReLU	318	128	128	4	1	4	SK
pool4	Max Pool. SK	314	128	128	2	1	4	SK
ip1 + relu6	Conv. SK + ReLU	258	128	512	8	1	8	SK
ip2 + relu7	Conv. SK + ReLU	258	512	256	1	1	1	SK
upconv1	Deconv.	516	256	256	2	2	1	U
conv6	Conv.	516	256	128	1	1	1	U
mergecrop1	MergeCrop	516	128 + 64	192	1	1	1	U
conv7 + relu8	Conv. + ReLU	514	192	128	3	1	1	U
conv8 + relu9	Conv. + ReLU	512	128	64	3	1	1	U
ip3	Conv.	512	64	2	1	1	1	U
prob	Softmax	512	2	2	1	1	1	U

Table 3.5: USK network configuration.

The USK-Net architecture has $|W| \approx 5.5 \cdot 10^6$ parameters, using Equation 3.6 and the corresponding values in Table 3.5. This is a fraction (about 25%) of the SK- and U-Net weights. The savings mainly come from reducing the *ip1* layer, which now only has $\approx 4.2 \cdot 10^6$ weights. While the *ip1* layer is still the most expensive one, the network is more balanced than SK-Net. The inner product layers are less important, because the U subnetwork merges and convolves the feature maps from the beginning of the network together with upsampled signals from the *ip2* layer. Figure 3.4 displays the balanced feature maps on the two processing paths. Furthermore, the USK-Net inherits the advantage of having bigger kernel sizes than only 3 by 3 (U-Net), going up to 8 by 8 in the *ip1* and 6 by 6 in the *conv2* layer. This means the USK-Net can learn features with looking at a bigger context inside the feature maps, while still almost reaching the speed of U-Net (see Section 6.5) in forward processing.

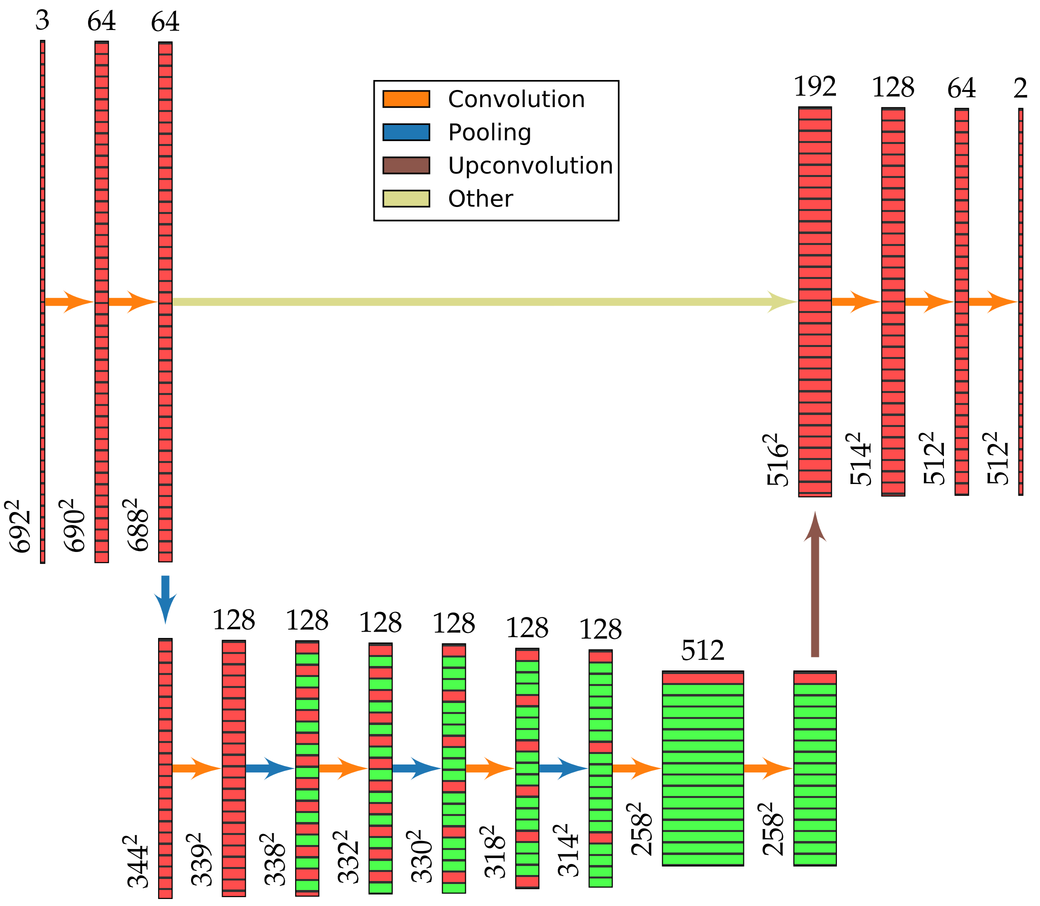


Figure 3.4: USK network configuration visualization. Green-red striped blocks represent feature maps with a kernel stride ($d > 1$). The U subnetwork blocks are just displayed red because the kernel stride does not apply there. Vertical numbers represent the size of the feature maps while the horizontal numbers represent the number of feature maps.

With less weights and less layers (less depth) than the U-Net, it is easier to train and also gave better segmentation results on the two evaluation data sets (see Chapter 7).

The USK network uses random Gaussian weight initialization with $\mu = 0$ and $\sigma = 0.01$ for the SK subnet and $\sigma = \sqrt{2/(f_{\text{in}} \cdot k^2)}$ for the U subnet. Other initializations, such as $\sigma = 0.01$ for the whole network caused the network to either disable many neurons (causing feature maps with zero activation) or stopped the loss from decreasing early during training.

A directed acyclic graph representation of the network can be found in the appendix A.3.

Chapter 4

Caffe Neural Tool

4.1 Functionality

As the standard Caffe binary [6] does not support training with patches, a new interface had to be written on top of the Caffe library. One option is to use the Pycaffe interface with custom python code to load images and pre-process them for patch training and processing as depicted in Figure 1.2.

I chose to implement a C++ interface similar to the original binary, because OpenCV and OpenMP can be used for efficient, parallelized algorithms to preprocess raw and label images.

The Caffe Neural Tool [16] can be configured for training, processing and benchmarking with a prototxt file similar to the configuration files used to set up networks and solvers (learning configurations) in Caffe. The two most important functionalities are histogram equalization during training and the image preprocessor, which can prepare the label and raw images in various ways before filling up the neural network.

Template configurations for training and processing on the data sets DS1 and DS2 as well as benchmark scripts for U-, SK- and USK-Net are available in the Caffe Neural Models repository [9].

For training, the following parameters have to be provided:

- A Caffe solver prototxt configuration which contains parameters such as learning rate, weight decay, solver method and the network to use for training.
- The padding size (v), network output (patch) size (w), network input (f_{in}) and output (f_{out}) feature map count.
- A folder with raw images and a folder with the corresponding label images, which are matched by alphabetic order.
- The preprocessor configuration block, including histogram equalization settings.

- Optionally, a solverstate file to resume a training. The already learned weights and current learning rate will be loaded instead of the initial network configuration.

For processing, the following parameters have to be provided:

- A Caffe network prototxt configuration to use for processing.
- A caffemodel file, containing the trained network weights.
- A folder with raw images and the segmentation output options (file type, pixel format and folder).
- The padding size (v), network output (patch) size (w), network input (f_{in}) and output (f_{out}) feature map count.
- The preprocessor configuration block (without histogram equalization).
- Optionally, all memory blobs in the network can be stored during processing. This feature is called *filter output* and is useful to check if the network learns the right convolution filters.

The tool also supports a variety of input and output image formats: Normal JPEG, PNG, TIF and BMP files as well as TIF image stacks (multiple images in one file) and 32 bit floating point TIF instead of integer pixel values.

When benchmarking, the tool re-uses existing training and processing configurations, but only fills the network with random data. It will report memory usage, layer wise forward and backward times and the total processing time of the network. For convolution layers, it will also estimate and store the computational complexity. This was used to generate the results in Chapter 6.

4.2 Preprocessing

The preprocessing options available are:

- Label consolidation, allowing to combine multiple labels into one. This technique was used on the data set DS1 (see Section 2.1). The consolidation is applied after histogram equalization, allowing to balance out difficult and rare labels before consolidating to only background and foreground. This also allows to mark important small, difficult features in the training data.
- Rotation of the training patches to a random multiple of 90°.
- Random mirroring of the training patches.
- Blurring training patches with a Gaussian kernel of any size. The blurring has a zero mean and the variance is picked at random from a normal distribution. The mean and variance of the distribution can be selected.
- CLAHE (contrast limited adaptive histogram equalization), with a clipping parameter. The function is integrated with OpenCV.
- Patch normalization to $[-1.0, 1.0]$ in floating point before feeding the neural network.

There are a few reasons why arbitrary rotation is not available:

- Interpolating to any angle that is not a multiple of 90° causes aliasing of training patches (labels and raw images).
- Arbitrary rotations make the training patch smaller due to corner cutting. This means there are less available total patches and the computation of histogram equalization methods gets much more complicated. The assumptions about the patch prior and label posterior distributions do not hold anymore.
- It is questionable how useful the additional training data would be. Elastic deformations would have more potential in generating unique new training data for biological images such as neural tissue EM images of DS1 and DS2.

4.3 Histogram Equalization

Histogram equalization is a technique to balance out the frequency of labels that the network sees during training. As training with patches leads to a set of dependent pixels which are in close proximity on an image (see Figure 1.2), the training results can be worse than with minibatches. With minibatches (see Figure 1.1), it is possible to create a database of single pixel labels and the minibatch can draw n independent pixels to train with in each stochastic gradient descent step.

The first equalization approach is a patch prior, which will prefer patches with rare labels. This is done by comparing the label distribution within each patch to the total label frequency in all training images.

$$r_j = \sum_{i=0}^{n-1} \frac{a_{j,i}}{c_i} \tag{4.1}$$

$$\hat{c}_i = \frac{1}{Z_i} \sum_{j=0}^{m-1} r_j \cdot a_{j,i} \tag{4.2}$$

For patches $j = 0, \ldots, m-1$ and labels $i = 0, \ldots, n-1$, Equation 4.1 calculates the weight for each patch (r_j) based on the total label distribution c_i and the frequency within each patch ($a_{j,i}$). Equation 4.2 calculates the label posterior distribution $\hat{c}_i$ based on the patch weights r_j and the label frequency within the patch. Z_i is a normalization factor to get $\sum_{i=0}^{n-1} \hat{c}_i = 1$.

The method does help if there are patches that have rare labels, because those patches will be drawn at random with a higher probability than others. An example is the synapse label (number 7) (see Figure 4.1 and Table 4.1). It cannot balance the labels which have a similar distribution in every patch - for example cell membranes versus cell interior.

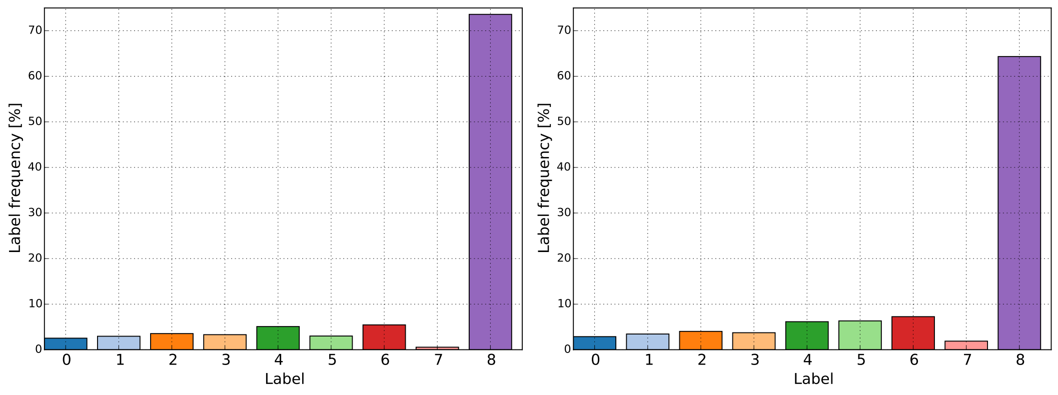
When the patch size gets bigger and approaches the size of the training images, the label distribution after the patch prior approaches the original label distribution. Thus the patch prior only works with small training patches. It can also completely equalize the histogram when using a single pixel as patch size ($w = 1$), as this is the same situation as with independent pixels.

When calculating label frequencies c_i, it is taken into account that pixel labels closer to the border of the image are covered by less patches than those in the center of the image. A patch can start and end at any offset inside the image. Corner pixels for example are only covered by the one patch that starts in that corner.

After applying the patch prior, the new label distribution $\hat{c}_i$ is depicted in Figure 4.1 with values from Table 4.1.

Method	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
None (c_i) (%)	2.5	3.0	3.5	3.3	5.1	3.0	5.5	0.6	73.6
Patch prior ($\hat{c}_i$) (%)	2.9	3.4	4.0	3.7	6.1	6.3	7.3	1.9	64.3
Masking (%)	11.1	11.1	11.1	11.1	11.1	11.1	11.1	11.1	11.1

Table 4.1: Label frequency using different histogram equalization techniques.



(a) No equalization (c_i). The majority of labels is underrepresented.

(b) Patch prior ($\hat{c}_i$) on 64 by 64 pixel patches ($w = 64$).

Figure 4.1: Label frequency using the patch prior. Label 8, the cell interior, gets slightly reduced while especially glia cells (5), mitochondria (6) and synapses (7) are represented stronger. Label numbers correspond to DS1 (see Section 2.1).

The second method masks pixels randomly in each patch, where the random function is thresholded by the inverse frequency of the label. This means a pixel of label type i gets masked (removed) from the error map if:

$$c_i^{-1} \cdot (\min_j c_j)^{-1} < p \tag{4.3}$$

In Equation 4.3, $p \in [0,1]$ is a random value picked at uniform and $c_i \in [0,1]$ the label frequency for label $i = 0,\ldots,n-1$. Less frequent labels are less likely to be masked out, while membrane and cell interior labels, which are very common, get masked with a high probability. The result is a completely balanced label histogram (see Table 4.1). The least frequent label consequently never gets masked.

The best solution to train on an exact label distribution as desired remains to do minibatch training. This is theoretically also possible with SK-, USK- and U-Net, but very inefficient in terms of training speed.

Minibatch training is the standard for most networks, also for SW-Net. The network weights are updated every time after a patch or minibatch has run through forwarding and backwarding. The gradients get accumulated over all error pixels in the Softmax loss layer and are then normalized for stable training.

Having a balanced label distribution is mostly important with Softmax and multi label classification. When doing background-foreground separation with Malis, the patch prior has little to no effect and masking can not be used at all. Malis does already focus the error on problematic zones globally over the whole patch that runs through the network (see Section 5.3.2) and therefore the label distribution during training does not matter.

Chapter 5

Caffe Library

5.1 Introduction

Adapting the Caffe library for efficient pixelwise classification on heterogeneous hardware contains the most programming work in the scope of the project, changing 20'000 lines of code compared to the BVLC master branch [5], [6].

The changes can be grouped into adaptions on different levels:

- Modified solver and network code to support the Caffe Neural Tool C++ interface.
- Modified existing layers to fit the pixelwise inputs and outputs.
- Additional layers for SK, U and USK architectures.
- Additional layers for the Malis loss.
- Redesign of N-dimensional layers to support up to 6D convolutions and max pooling with strided kernels.
- OpenCL and OpenCL hybrid code for supporting a wide range of GPUs and CPUs.
- Backend adaptions to allow dynamic backend dispatching at run- and compile-time.
- Adapted GNU Makefile and CMake build infrastructures to support the new OpenCL backend.

The changes are implemented in a way that does not break backward compability to the original library. All existing network models and trained networks can still be used. The source code remains highly maintainable and was ahead of the Caffe BVLC branch [6] during the whole scope of the project.

5.2 Modified Layers

5.2.1 SK Layers

SK (strided kernel) layers are layers with a kernel size $k > 1$ and an inner stride in the kernel ($d > 1$). The result is a kernel that looks at a feature map in a context of $(k-1) \cdot d + 1$ pixels, which is also called the external kernel size.
The motivation to have such kernels is to be able to convert single pixel prediction networks (sliding window (SW) networks), to patch prediction networks (SK). How this works is visible in the Figures 1.1 and 1.2, as well as the strided representation in Figure 3.1.

For SK convolutions, the matrix multiplication (see Section 5.5) stays the same as with normal convolutions. Only the *im2col* and *col2im* memory copy kernels have to be adapted. For this, I used the existing kernel codes provided by Hongsheng Li *et al.* [7].

Changing existing *im2col* and *col2im* kernels to support $d > 1$ is trivial:
The kernels just read the data by iterating over all input feature maps and dimensions. Within each dimension, the iteration goes over the kernel size k, copying data into the convolution buffer.
With strided kernels, when iterating over the kernel, the reading pointer has to be increased by the dimension stride multiplied by the kernel stride d instead of just adding the dimension stride. The dimension stride is $(w^{(i-1)})^j$, starting at $j = 0$ for the first dimension and $w^{(i-1)}$ denoting the input feature map size.

The same iteration scheme applies for pooling operation kernels. All other layers do not have to be adapted and work together with SK layers as long as they have a kernel size of $k = 1$ (see Algorithm 1).

Strided kernels are only implemented in CUDA and OpenCL and do not run as native CPU code.

5.2.2 N-Dimensional Layers

The Caffe library is able to specify an arbitrary amount of dimensions for the blob memory infrastructure used to pass data between layers. However, not all layers automatically work in higher dimensions. For most element-wise kernels, nothing has to be changed. Convolutions and pooling operations require a slight redesign.

For convolutions, the matrix multiplication stays again the same as with normal and strided kernel convolutions (see Section 5.5). There were existing kernels for normal N-dimensional convolutions by Jeff Donahue [12]. However, those kernels only support the default kernel stride $d = 1$.

In the scope of this research project, at HHMI Janelia, I combined the existing code of the strided kernel and N dimensional convolutions to get the most generalized form of convolutions, supporting up to 6 dimensions and kernel strides. The 6 dimension limit exists because allocating local arrays of dynamic sizes is not allowed in OpenCL and CUDA. The arrays are required to store temporary variables such as iterators for each dimension.

Derived from the convolution code, I also implemented the max pooling function as ND-SK kernel.

In the case of 1D or 2D, the SK and normal convolution layers should be considered, as looping over two dimensions is more efficient than having an outer loop over dimensions and an inner loop which processes one dimension at a time. The ND layers should be used from 3D to 6D.

This report does not analyze any networks with a dimension higher than two, but it is possible to replace *ConvolutionSK* layers by *ConvolutionND* layers in the SK-Net to get an arbitrary dimensional network. It may be required to reduce the network output size and feature map count in order to meet memory constraints.

N-dimensional kernels are only implemented in CUDA and OpenCL and do not run as native CPU code.

5.3 New Layers

5.3.1 Merge Crop

The *MergeCrop* layer is required in U- and USK-Net architectures (see Sections 3.4 and 3.5). The layer accepts two input blobs:

- Blob A of size $w_A^{(i-1)}$.
- Blob B of size $w_B^{(i-1)} \geq w_A^{(i-1)}$.

The layer outputs a blob containing all feature maps of A and B, which can have a different amount of feature maps. Input B has to be cropped to the size of A. The output feature maps are of size $w^{(i)} = w_A^{(i-1)}$.

During backpropagation, the error maps are propagated through by copying them in the inverse direction. For U- and USK-Net, backwarding is only enabled for the input A, as input B gets the differential data on a different path (from the down sampling pooling layer) in the neural network. Copying back B would overwrite the gradients and interfere with the intended training.

MergeCrop is only implemented in CUDA and OpenCL and does not run as native CPU code.

5.3.2 Malis Loss

MALIS stands for maximum affinity learning of image segmentation. The implementation in Caffe [5] that I provide is based on existing code to compute the Malis criterion for Matlab [17] and Torch [18] by Srinivas Turaga *et al.* [3].

Additional layer forward and backward functions (for interaction with Caffe), memory management and the two additional layers *ConnectedComponent* and *Affinity* are new contributions. The layers are only implemented as CPU code and do not run on OpenCL or CUDA.

Figure 5.1 describes how the Malis criterion loss is used together with the network models presented in this report (see Chapter 3). In place of the Softmax activation for two labels, a rectified linear unit or other activation could also be used. The *split* layer is required to feed the ground truth *label* blob into the two following *components* and *affinity* layers.

The layer structure is separable so that the Malis loss can be used with feeding in external connected component and label affinity maps instead of computing them indirectly. Likewise, the neural network can directly learn affinity graphs instead of pixelwise labels.

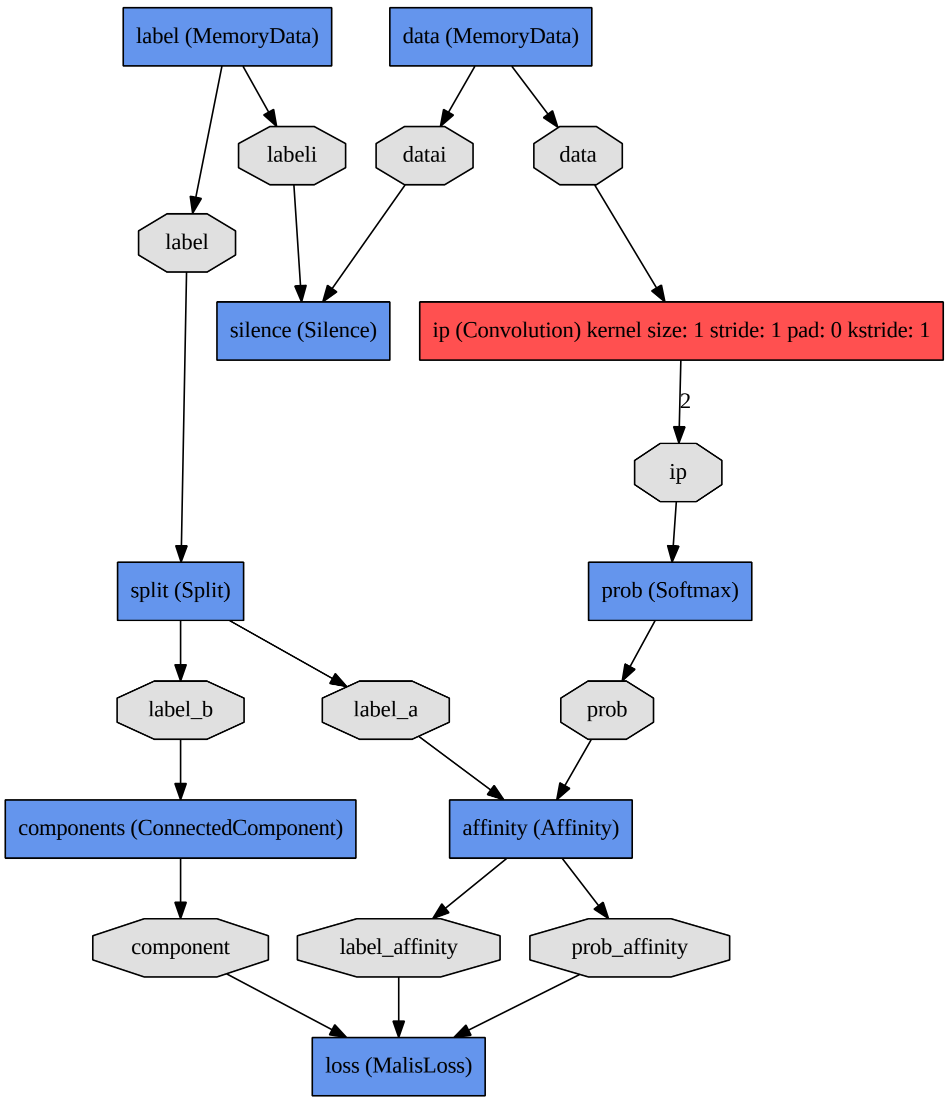


Figure 5.1: Malis loss setup DAG. In place of the single *ip* convolution layer, there would be a whole neural network (see Appendix A).

As the Malis criterion is calculating minimum spanning trees between pixel pairs on affinity graphs and is only selecting the minimum edge of the tree as error edge, if the position is indeed an error in the prediction, the actual Malis function is called twice on two affinity maps A^+ and A^-.

Having the predicted affinity graph A_{pred} (=*prob_affinity*) and the ground truth to it (A_{label} (=*label_affinity*)), two new affinity maps are generated:

$$A^{+} = \min(A_{\text{pred}}, A_{\text{label}}) \tag{5.1}$$

$$A^{-} = \max(A_{\text{pred}}, A_{\text{label}}) \tag{5.2}$$

Then, A^{+} will not contain any errors on the background prediction while A^{-} has no errors in the foreground. The result is that the Malis criterion is able to isolate errors in the background prediction (membranes with gaps) on A^{-} and find errors on the cell interior with A^{+}. The resulting error map for backpropagation is $\Delta A = \Delta A^{-} + \Delta A^{+}$.

In Equations 5.1 and 5.2 the affinity maps A denote the combination of vertical ($A^{(y)}$) and horizontal ($A^{(x)}$) affinity maps. The affinity graphs contain one value per edge between two neighboring pixels. A value close to 1 stands for high affinity while 0 means unconnected.

Details on how the Malis criterion works internally can be found in the original paper on Malis by Srinivas Turaga *et al.* [3]. The implementation is quite efficient and does not contribute massively to the training time.

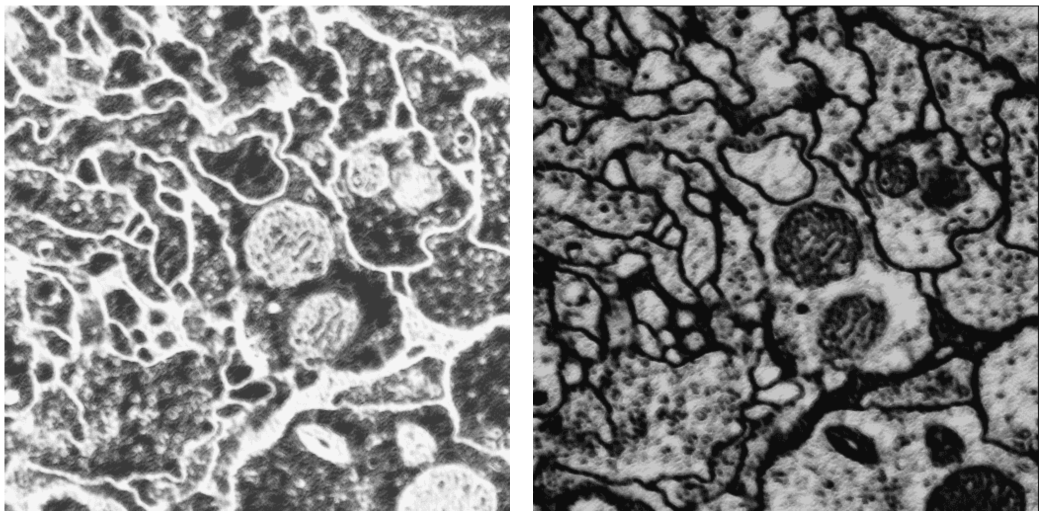
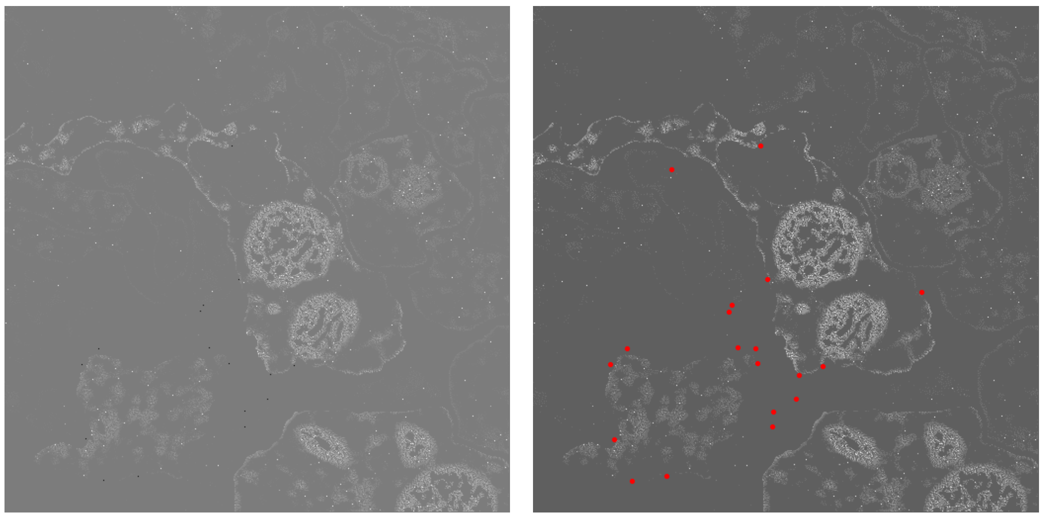

(a) Background prediction map (I^{-}). (b) Foreground prediction map (I^{+}).

Figure 5.2: Network predictions after a few training steps with Malis loss (USK-Net, Section 3.5). The network can not separate membranes and mitochondria yet. The images correspond to the data (forward) component of the memory blob *prob* in Figure 5.1.

The error map ΔI^{+}, which is derived from ΔA using the *affinity* layer during backpropagation, is depicted in Figure 5.3. On a first look, it seems like only cell interior errors, the light gray areas, exist in the picture.
When enhancing the contrast and marking errors in the membrane with red spots, it becomes more clear what happens: The Malis criterion calculates minimum spanning trees between pixels of the cell interior of two merged cells that should

be separated. The edge that will be corrected is almost always the same between two cells for all pixel pairs within those cells, due to shared edges in the spanning tree. This edge will therefore accumulate a high error value in ΔA^-, but there are only a few such spots in every training patch.

On the cell interior errors (ΔA^+), the edges to be corrected occur on the border of cell interior areas that are mislabeled as cell membrane. This leads to a denser distribution of the error, but less error intensity per edge.


(a) Original error map (ΔI^+). Membrane errors are sparse and barely visible.

(b) Contrast enhanced error map. Red marks denote membrane errors.

Figure 5.3: The error map generated by the Malis loss. The image corresponds to the diff (backward) component of the memory blob *prob* in Figure 5.1. The gray background color is zero (no) error, while black spots are negative errors from ΔA^- and light gray areas are positive errors from ΔA^+.

Malis loss and the U/USK-Net models match perfectly: As the networks can output very large patches up to 512 by 512 pixels without reaching memory limits of current GPUs (see Section 6.4), connected components and the Malis loss have a very large context to work on.
As soon as the patch size is so small that it barely covers a cell, Malis becomes useless. With sliding window networks for example, it would be necessary to compute many forwarding iterations first before being able to run Malis and generate an error map to backpropagate. It would also be necessary to retain all blobs of all forwarding iterations. The whole process would not be efficient in terms of memory consumption and training times.

5.3.3 Affinity

The affinity layer is an additional layer that has to be used in conjunction with Malis loss (see Figure 5.1).

The idea is that not only affinity graphs but also pixelwise classifications can be learned with the Malis loss. During forwarding, the affinity layer has to look at neighboring pixels in the horizontal and vertical direction of the input and compute their connectivity (affinity).

$$A^{(x)}_{x,y} = \min_{z=x,x+1} I^{+}_{z,y} \tag{5.3}$$

$$A^{(y)}_{x,y} = \min_{z=y,y+1} I^{+}_{x,z} \tag{5.4}$$

$$M^{(x)}_{x,y} = \arg\min_{z=x,x+1} I^{+}_{z,y} \tag{5.5}$$

$$M^{(y)}_{x,y} = \arg\min_{z=y,y+1} I^{+}_{x,z} \tag{5.6}$$

In Equations 5.3 to 5.6, $A^{(x)}$ is the horizontal oriented affinity map, $A^{(y)}$ the vertical one. The affinity maps have the same size as the input image I. Additionally, the minimum index maps $M^{(x)}$ and $M^{(y)}$ have to be stored so that the loss can be distribute accordingly during backpropagation.

The Softmax layer (blob *prob*) in Figure 5.1 and the ground truth labels (blob *label_a*) actually store both the foreground and background (see Figure 5.2). For the image I used to produce the affinity map, only the foreground prediction map (I^{+}) is considered because it stores 1 for foreground (connected) and 0 for background (disconnected). The resulting affinity graph will correctly have higher values for connected pixels than disconnected ones.

The affinity graph is computed twice: Once for the ground truth (blob *label_affinity* in Figure 5.1) and once for the current network prediction (blob *prob_affinity*).

During backpropagation, the affinity loss has to be attributed to single pixels again, as the Malis criterion will attribute the error map to an affinity graph. Now, both the foregound (ΔI^{+}) and background (ΔI^{-}) have to get an error map to balance out the Softmax function. The loss is attributed symmetrically, $\Delta I^{+} = -\Delta I^{-}$.

Initialization:

$$\Delta I^{+}_{(x,y)} = \Delta I^{-}_{(x,y)} \qquad = 0 \tag{5.7}$$

$$\tag{5.8}$$

Update:

$$\Delta I^{+}_{M^{(x)}_{x,y},y} \qquad += \Delta A^{(x)}_{x,y} \tag{5.9}$$

$$\Delta I^{-}_{M^{(x)}_{x,y},y} \qquad -= \Delta A^{(x)}_{x,y} \tag{5.10}$$

$$\Delta I^{+}_{x,M^{(y)}_{x,y}} \qquad += \Delta A^{(y)}_{x,y} \tag{5.11}$$

$$\Delta I^{-}_{x,M^{(y)}_{x,y}} \qquad -= \Delta A^{(y)}_{x,y} \tag{5.12}$$

Equations 5.8 to 5.12 describe how to attribute the affinity loss back to pixel loss, given the minimum index maps $M^{(x)}$ and $M^{(y)}$ computed in the forward processing step.

There are also other ways to compute an estimation to an affinity graph, such as averaging neighboring pixels. The choice for the minimum selection worked particularly well because there is no loss of resolution or aliasing when computing it this way. The Malis criterion selects the minimum edge of the affinity graph for creating the loss maps $\Delta A^{(x)}$ and $\Delta A^{(y)}$. Thus using the minimum valued pixel through $M^{(x)}$ and $M^{(y)}$ for attributing the pixel loss makes sense. Both objectives minimize the same error by either increasing or decreasing the affinity of the neighboring pixels.

5.3.4 Connected Components

The connected components layer is a small layer based on the OpenCV flood-filling algorithm and outputs separated connected components from a foreground-background labeled ground truth (Figure 5.4). Based on this map, the Malis loss knows which areas have to be separated and which are connected. Here, the cell membrane, which is considered background, is not assigned to any component.

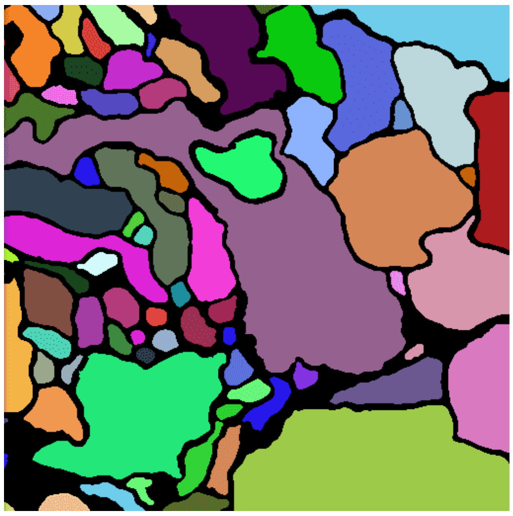


Figure 5.4: Connected components belonging to Figure 5.2. The feature map corresponds to the memory blob *component* in Figure 5.1.

5.4 OpenCL Backend

5.4.1 Implementation

In my version of the Caffe library, an additional versatile backend for various compute devices, based on OpenCL and ViennaCL [19], is available. The backend is called *Greentea* and is part of the *Project Greentea* consisting of frontend, models and modified Caffe library (see Figure 1.3). In this section, an overview of interesting aspects how the backend works and how the Caffe library had to be changed is given. Further details and a full documentation is available within the source code, which is available for download [5].

A key feature is that the OpenCL backend is feature equivalent to the CUDA backend. All GPU layers can be used on both backends. The OpenCL backend is also unit test verified and passes all test cases of the original Caffe library. The tests can be invoked by executing "make runtest" on the source code folder.

It remains possible to compile the library with support for all backends at once. The compute kernel and BLAS calls can be dispatched dynamically at runtime, depending on what kind of device is selected. Every device available is registered in a new *DeviceContext* object that stores the device and backend type.

The following aspects of the library had to be changed:

- The Caffe library enumerates all devices on all enabled backends, starting with CUDA devices. The selected GPU number determines which *DeviceContext* is set as the default.
- The *SyncedMem* class that is used to manage the device memory can now store either a CUDA GPU pointer or an OpenCL *cl_mem* memory object, depending on which device and backend the memory object belongs to.
- The *Forward_gpu* and *Backward_gpu* functions now contain both OpenCL and CUDA code to call compute kernels and BLAS functions.
- Network layers, *SyncedMem* and blob objects carry a pointer to a *DeviceContext*, which allows to express neural network object to device relationships. This is a feature for allowing future multi-device networks, where keeping track of which memory blob is on which device is essential.
- All CUDA compute kernels are translated to OpenCL code. This is trivial for the most part, as the syntax is very similar.
- The OpenCL backend can dispatch BLAS calls to ViennaCL-BLAS or clBLAS on GPUs. ViennaCL-BLAS is header-only and therefore easier to use, while clBLAS is optimized for certain AMD GPUs but has to be compiled separately.
- On CPU devices, the *Greentea* backend mixes native CPU code with OpenCL code to achieve an optimal performance (see Section 5.4.2).

5.4.2 OpenCL Hybrid

The OpenCL hybrid implementation describes how the OpenCL backend is used when selecting a CPU device instead of a GPU device. The two fundamental differences are memory allocation and BLAS library calls.

When a *SyncedMem* object is instantiated, the memory is allocated as host memory rather than device memory. Differently than with the CPU backend, the memory is allocated through OpenCL. This allows to access the underlying memory pointer of OpenCL memory objects while also being able to use the memory in OpenCL kernels.

For BLAS calls, the following steps are executed:

1. For all involved *cl_mem* memory objects, the underlying host pointer is recovered and mapped to a new CPU pointer. At this point, the OpenCL backend ensures all compute kernels accessing the memory concurrently are done executing so that it is safe to access the memory over CPU pointers.
2. The BLAS call is dispatched to a cBLAS library (Intel MKL, ATLAS or OpenBLAS). Here, the most optimized BLAS for the CPU device can be selected. Optimally, it should be a BLAS that is fully parallelized and uses all CPU cores. NUMA issues (see Section 6.7) might occur.
3. As soon as the BLAS call returns, all CPU pointers are unmapped. This signals to the OpenCL backend that it is safe again to start OpenCL kernels on the *cl_mem* objects involved.

When mapping *cl_mem* objects, it can be specified that the access is read-only. In this case, OpenCL kernels that also only use the object in read-only mode can continue to run during the BLAS call.

Using the OpenCL backend on CPUs is a design decision. An alternative would be to parallelize the existing CPU backend with OpenMP pragmas. However, as most of the computational complexity resides with the BLAS calls (see Section 5.5) and the OpenCL kernels are not using local memory extensively, they run very well also on CPUs. Only the BLAS, which is very device specific, and needs to be optimized for the memory architecture (see Section 6.3), needs to be different from the GPU version of the OpenCL backend.

5.5 Convolution Methods

Convolutions are usually computed using three different methods:

1. GEMM (matrix multiplication) convolutions, requiring a reshape of the input to fit the BLAS SGEMM scheme.
2. Direct convolutions, shifting the convolution kernel directly over the input.
3. FFT domain convolutions, requiring to compute at least two Fourier transforms and one inverse Fourier.

Caffe implements GEMM convolutions. The advantage is that highly efficient BLAS libraries are available specifically for various devices, such as clBLAS, cuBLAS and OpenBLAS. It is very hard to implement convolutions more efficiently using direct convolution. The performance of such implementations is not portable for different kernel sizes and hardware types. In my own preliminary experiments, not even 10 % efficiency could be reached in the *ip1* layer of SK-Net, compared to up to 90 % using GEMM convolution, including the time for input reshaping (see Section 6.6.1, Table 6.6 and Figure 6.6).

It is particularly difficult to get good local and global memory access patterns when programming kernels for direct convolution. BLAS libraries have already been optimized to use GPU local memory and CPU caches optimally.

GEMM convolutions also simplify the implementation of higher dimension and strided kernel convolutions, as only the code for reshaping the input has to be adapted. In Caffe, these functions are called *im2col* and *col2im*.

A huge disadvantage with GEMM convolution is the memory requirement for the convolution buffer, discussed in Section 6.4. Assuming square sized kernels and output images, Equation 5.13 gives the buffer size in float elements. The kernel and output dimension is denoted by x. The network architectures in this report all use $x = 2$ (2D).

$$M_{\text{buffer}} = f_{\text{in}} \cdot k^x \cdot w^x = K \cdot N \tag{5.13}$$

A matrix multiplication consists of three matrices, $A \in \mathbb{R}^{M \times K}$, $B \in \mathbb{R}^{K \times N}$ and $C \in \mathbb{R}^{M \times N}$. In Caffe, A is the weight matrix, B the column data after *im2col* and $C = A \cdot B$ the layer output.

The dimensions are $M = f_{\text{out}}$, $N = w^x$ and $K = k^x \cdot f_{\text{in}}$. The resulting computational complexity is $\mathcal{O}(M \cdot N \cdot (2 \cdot K - 1)) = \mathcal{O}(f_{\text{out}} \cdot w^x \cdot (2 \cdot f_{\text{in}} \cdot k^x - 1))$.

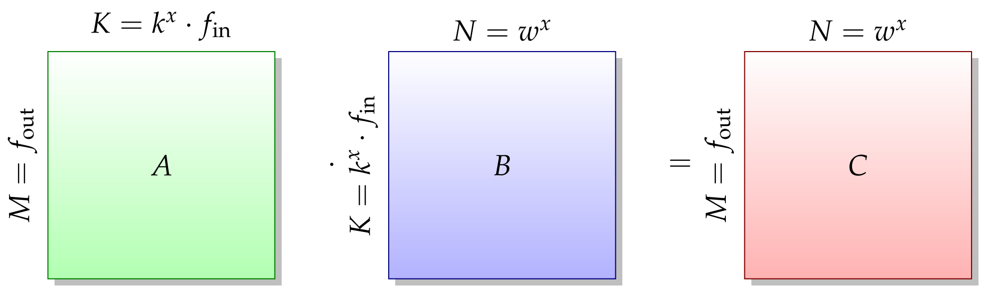


Figure 5.5: Matrix-matrix multiplication in Caffe.

In the BLAS libraries, row-major NN-SGEMM has to be used. Row-major means the leading dimension in memory is the matrix row. NN means that the matrices A and B are both not transposed. SGEMM stands for single precision general matrix matrix multiplication.

In the scope of this project, FFT convolutions have not been considered. There were no suitable FFT libraries available for all devices that needed to be supported. Normal FFT convolution also uses a lot of additional memory, as all kernels are stretched to the size of the input in Fourier space. The device memory is typically insufficient (see Section 6.2) to store all FFT kernels for reuse, and recomputing every kernel is too time intensive.

The cuDNN library also implements a modified form of GEMM convolutions, and they also evaluated FFT and direct convolution as options [20]. A recent paper about using fast FFT convolutions exists (Vasilache *et al.* [21]), but managing the device memory remains difficult.

Chapter 6

Benchmarks

6.1 Introduction

This chapter assesses the performance (or efficiency) of different models across a variety of hardware devices. This is essential for speeding up neural networks by finding and eliminating computation and memory bottlenecks.
All theoretical computations in this chapter assume two dimensional square sized compute kernels and feature maps. This is in accordance to how the network architectures are set up (see Chapter 3).

6.2 Hardware

Vendor	AMD	nVidia	Intel	Intel
Device	W9100	GTX 980	i7-4790K	2x E5-2697v3
Compute Units (Shaders) / (Threads)	44 (2816)	16 (2048)	4 (8)	28 (56)
Memory (GiB)	16	4	16	>16
Clock Frequency (MHz)	930	1126	4000	2600
Performance (GFLOP/s)	5240.0	4612.0	512.0	2329.6
Memory Speed (GiB/s)	320	224	25.6	2x 68

Table 6.1: Hardware used for benchmarking.

The device specifications in Table 6.1 are taken from the official white papers (AMD [22], nVidia [23], Intel ARK [24]). The FLOP/s performances indicated assume the fused-multiply-add (FMA) operation.

The two AMD W9100 graphics cards have kindly been sponsored by AMD [25], as noted in the acknowledgments. The card has special features such as high 64 bit precision performance and error correcting memory. Those were not used for Caffe, only the OpenCL 2.0 driver and the large amount of memory was of importance. The W9100 is a workstation card, but consumer cards from both nVidia and AMD can also be used without restrictions as long as there is enough device memory (R9 290X, R9 390X, Titan, Titan X). This is to be considered when building low-cost, high-throughput systems for neural networks.

The i7-4790K CPU and GTX 980 GPU are devices of my personal workstation, specially acquired to test with up to date hardware (as of 2015).
Extended testing with the E5-2697v3 processors and its issues (see Section 6.7) was not possible as this device was a workstation that I could only access briefly during my time at HHMI Janelia.

For all benchmarks in this chapter, the W9100 GPUs have been used, because they were the only available GPUs capable of running all models in forward- and backward-mode on a wide range of output sizes, due to memory requirements. The only exceptions to this are the hardware comparison benchmarks and where indicated explicitly.

Unless otherwise noted, *Intel* is used as an alias for the i7-4790K processor, *nVidia* for the GTX 980 GPU and *AMD* for the W9100 GPU in this chapter.

6.3 Software

The modified version of Caffe [5] in *Project Greentea* supports a variety of configurations that perform differently depending on the compute device. Table 6.2 represents the setup used for benchmarking. It is the best performing setup possible for each combination of backend and device for the models in Chapter 3.

Device	Backend	Memory Allocation	BLAS	Compute Kernels
Intel	CPU	Host (native)	OpenBLAS	Caffe (CPU native)
Intel	OpenCL	Host (OpenCL)	OpenBLAS	Greentea (OpenCL)
AMD	OpenCL	Device (OpenCL)	clBLAS	Greentea (OpenCL)
nVidia	OpenCL	Device (OpenCL)	clBLAS	Greentea (OpenCL)
nVidia	CUDA	Device (CUDA)	cuBLAS	Caffe (CUDA)

Table 6.2: Software configuration used for benchmarking.

OpenBLAS is compiled to use all CPU cores through OpenMP and supports all vector extensions available on the CPUs used. Alternatively, the cBLAS header interface also supports Intel MKL and ATLAS as replacements for OpenBLAS.
The CPU could also be used with clBLAS. This is not advisable, as clBLAS is optimized for GPUs, which have a different memory architecture than CPUs. While there is a cache hierarchy on the CPU, the GPU needs to use fast local memory to buffer blocks of the matrix (from global device memory) temporarily. Local memory does not exist on the CPU, therefore it would result in copying data needlessly on the host memory. This results in low efficiency, because CPUs already have a slow memory interface compared to GPUs (see Table 6.1).
As ViennaCL [19] was used as a part of the OpenCL backend, ViennaCL-BLAS is also available as an alternative to clBLAS. It is slower than clBLAS, but more

convenient to use as it does not need to be compiled separately and only consists of C++ header files.

6.4 Device Memory

As all networks presented can be run with almost any size of output (up to restrictions given by layers such as even divisibility of the input feature maps, see Chapter 3), the networks can be set up so that they fit the memory and computational restrictions given by each device. The networks do not have to be re-trained in this case and results are numerically identical.
A second objective may be to use processing output sizes matching the dataset: Non-square outputs such as 256 by 32 pixels are also possible. Then, the sizes can be set so that the image sizes to process are divisible by the output size of the network. Like that, no computations are wasted.

It is also important to note that there is no easy scaling rule as to how the memory requirements will change with different output sizes as this depends on the number of feature maps and their sizes on all layers as well as the maximum convolution buffer size (M_{buffer}).

An upper bound estimation for $w \gg v$ is that the memory usage increases proportional to w^2, where w is the output size and v the total padding size as in Table 6.3.
It follows that the network gets more efficient with a bigger ratio $\frac{w}{v}$, removing more overlapping computations. Using $w = 1$ on the SK network for example results in having the same efficiency as a minibatch sliding window (SW) network, which is about 50 times slower (see Table 6.5) than the corresponding SK network with 128 by 128 output.

On 3D networks, the memory allocation would scale in the order of w^3, limiting the output patch-cube size very quickly.

The smallest memory available in the test hardware was 4 GiB (see Table 6.1), thus the networks have been set up as in Table 6.3 for forward processing. The configurations are the same as for training and as described in Chapter 3, except for the USK net, where half the size was used (256 instead of 512). The resulting memory requirements are visible in Figure 6.1 and Table 6.4.

Training the USK model with 512×512 pixels, which improves training with Malis loss, requires up to 8 GiB of device memory, which is close to what scaling proportional to w^2 predicts.

Network	**SK**	**USK**	**U**
Processing output size ($w \times w$)	128×128	256×256	388×388
Training output size ($w \times w$)	128×128	512×512	388×388
Total padding size (v)	102	180	184

Table 6.3: Output and padding configurations in pixels. Output sizes are mostly flexible, but the padding size is a fixed network characteristic.

Network	**SK**	**USK**	**U**
Data Blobs (MiB)	219	560	1134
Processing (MiB)	2759	1621	1710
Training (MiB)	3056	2204	2955

Table 6.4: Peak device memory usage when using the processing output sizes denoted in Table 6.3 for both processing and training.

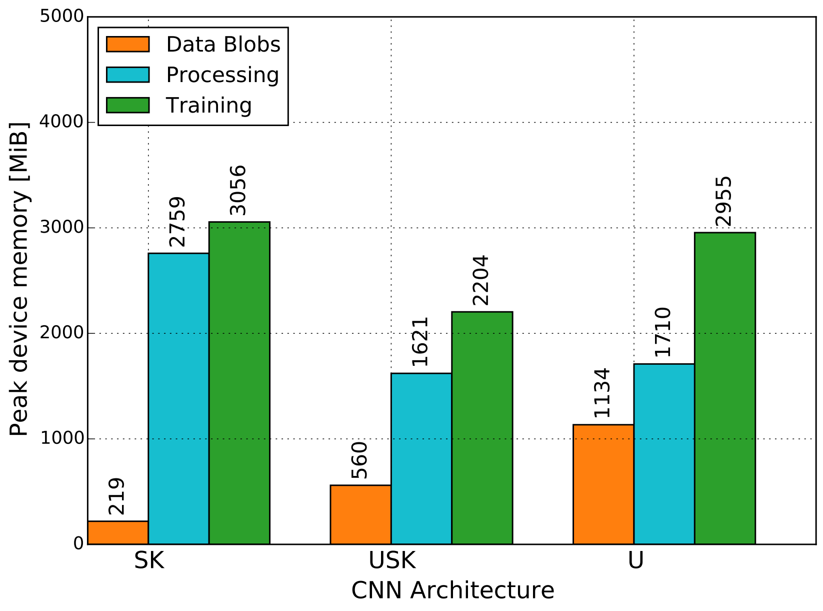


Figure 6.1: Peak device memory usage when using the processing output sizes denoted in Table 6.3 for both processing and training. The comparison reveals that SK networks would use a lot more memory than USK and U with the same output size, while USK and U are roughly comparable.

The memory usage during training is always higher than the processing requirements. This is because the error / difference is stored during back propagation. The data blobs therefore have to be stored twice (data and difference) for all layers that are back propagated - making the difference between training and processing roughly the size of the data blobs, as can be seen in Figure 6.1.

With SK and USK models, data, difference and weights are the smallest contributors to memory usage. The biggest consumption comes from the temporary buffer needed to compute convolutions with matrix multiplications. The effect is much smaller with U-Net architectures.
To assess this, the size of the buffer can be estimated:

$$M_{\text{buffer}} = \max_{l^{(i)} \in L_{\text{conv.}}} f_{\text{in}}^{(i)} (k^{(i)})^2 (w^{(i)})^2 \tag{6.1}$$

The size of M_{buffer} is in floating point elements. To get an estimate in bytes, it has to be multiplied by the GPUs floating point precision size, which is typically 4 bytes.

The maximum buffer allocation that worked with OpenCL and CUDA was 4 GiB for a single memory block. This is the upper limit for M_{buffer}. With the most expensive convolution in the *ip1* layer of SK-Net (see Section 6.6.1), it is possible to compute the limits for the network output size w, according to Equation 6.1. $f_{\text{in}} = 192$, $f_{\text{out}} = 1024$ and $k = 10$ are fixed network parameters. Thus, the maximum is $w = \sqrt{\frac{4\,GiB}{4\,B} \cdot (192 \cdot 10^2)^{-1}} \approx 236$ pixels.

For U-Net, this buffer is not the dominant factor that limits the network. In the USK-Net design, the *ip1* layer is of less importance and has less input feature maps and a smaller kernel size. This reduces the M_{buffer} limitation and allows bigger output patches, making the network more efficient.

With the OpenCL backend (Section 5.4), the memory overhead is up to $\min(n, q) \cdot M_{\text{buffer}}$, where q is the number of parallel work queues and n the minibatch size. This helps to speed up the many small convolutions that occur in SW networks by starting up to q convolutions in parallel. This feature can not be used with the CUDA and CPU backend. For CUDA, the solution in this case is to use cuDNN, which streams the convolutions in batches to be more efficient [20]. Both parallel queues and cuDNN are not of importance here, because the SK, U and USK architectures all work efficiently with having $n = 1$.

Reusing the convolution buffer is also a new feature in the improved Caffe library [5]. With the original Caffe library [6], the memory overhead would have been much higher, allocating one buffer per convolution layer:

$$M_{\text{buffer}} = \sum_{l^{(i)} \in L_{\text{conv.}}} f_{\text{in}}^{(i)} (k^{(i)})^2 (w^{(i)})^2 \tag{6.2}$$

Those buffers are in both cases persistent, because freeing them and re-allocating would cause too much run time overhead. With re-using the buffer, it would also not decrease the peak allocation any further. Alternative ways to implement convolutions are discussed in Section 5.5.

The memory consumption can not be decreased further during training, because the data blobs have to persist during forward- and backward-computation to calculate the difference and update the weights in training. However, during processing, a lower bound estimate is given by:

$$M_{\text{total}} = \min(n, q) \cdot M_{\text{buffer}} + n \cdot \max_{b^{(i-1)} \in B,\, b^{(i)} \in B} [f_{\text{in}}^{(i)} (w^{(i-1)})^2 + f_{\text{out}}^{(i)} (w^{(i)})^2] \tag{6.3}$$

This assumes that the blobs B of a network are not persistent and at most the input and output of the most memory consuming layer has to be stored. A network with its blobs can be seen as a directed acyclic graph (DAG). Therefore, the estimate is higher if there are layers / blobs that exist in parallel with each other.
Currently (as of August 2015) there is no Caffe implementation that re-uses the blobs in this way. It would be necessary to analyze the DAG when instantiating the network first. Then, blobs would have to be allocated and assigned so that no

conflicts exist. The DAG would need to be split up for analysis in the case that there are multiple devices with independent memory working on different parts of the network.

The implication is that lower end devices such as GPUs with less memory and even mobile devices would be capable of classifying with larger networks such as those presented in this report.
I chose to not implement this because there was enough GPU memory available and Figure 6.1 indicates the reduction would not be very large, as most memory is consumed by matrix-matrix multiplication buffer (M_{buffer}).

6.5 Labeling Throughput

Device	**Backend**	**SW**	**SK**	**USK**	**U**
Intel	CPU	73	0[a]	0[a]	0[a]
Intel	OpenCL	98	3 504	19 414	32 692
nVidia	OpenCL	850	28 461	267 478	390 156
AMD	OpenCL	1 108	59 513	354 865	483 766
nVidia	CUDA	1 488	85 460	658 018	1 058 125

Table 6.5: Labeling throughput
[a] not implemented

The labeling throughput (Table 6.5) is an overall performance measure for neural networks. It also shows how different devices and backends perform. Even on the CPU, using the fastest network (U) gives a speedup of 447× compared to what was achievable with Caffe [6] prior to *Project Greentea* [5]. When using a network (SK) that gives identical results as the original SW network, the CPU speedup is still a factor of 48×.

On AMD GPUs, speedups of 54× (SK-Net), 437× (U-Net) and 320× (USK-Net) compared to SW-Net are possible. The nVidia GPU scales similarly on both the OpenCL (SK: 33×, U: 459×, USK: 315×) and CUDA (SK: 57×, U: 711×, USK: 442×) backend.

SK, USK and U networks can not be executed directly on the legacy CPU backend, as layers such as strided kernel layers and merge crop have no native CPU kernels implemented. They are only available on CUDA and OpenCL. The fact that the OpenCL backend on CPUs is better parallelized (see Section 5.4.2) than native CPU execution in Caffe and that the speedups between networks are similar on CPUs and GPUs justifies not implementing the CPU kernels.

Figures 6.2 and 6.3 represent the values of Table 6.5 in both logarithmic and linear scale.

While the CUDA backend is usually the fastest, the nVidia GPU performs slower, as expected by the FLOP values in Table 6.1, relative to the AMD GPU when using OpenCL on both of them. Explaining this behavior needs insight into the individual network layer performance (Section 6.6) and the convolution operations (Section 5.5).

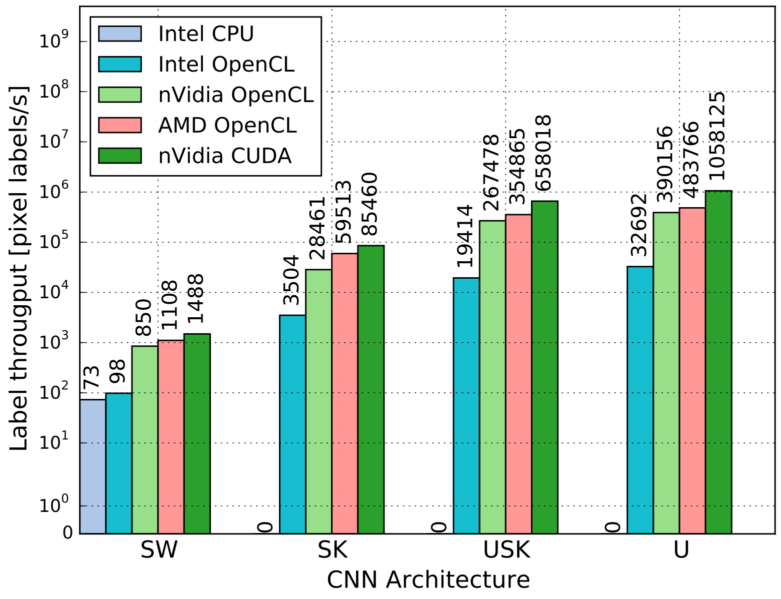


Figure 6.2: Log scaled labeling throughput

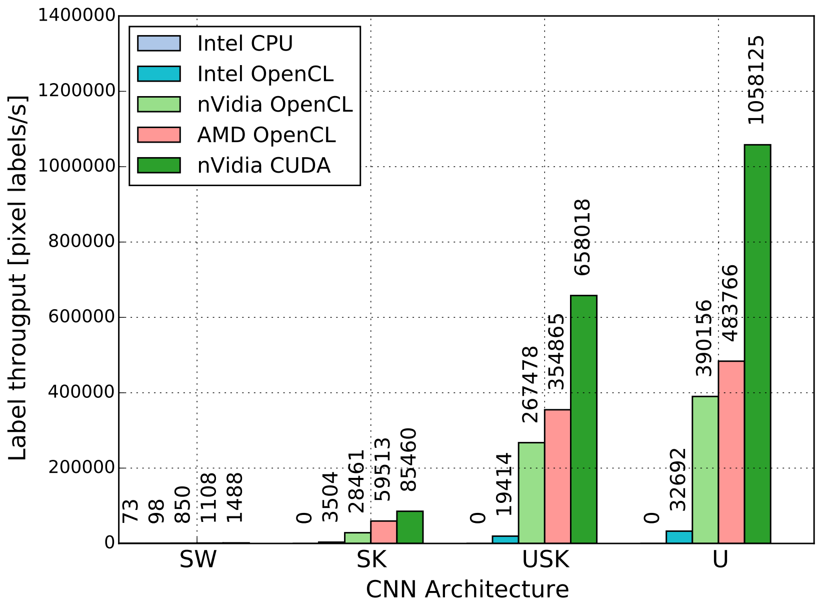


Figure 6.3: Linear scaled labeling throughput

The network performance during training was not assessed in this report, as training the models on the available data was not a limiting objective. Back propagation is usually slower than forwarding, as the differential maps, gradients and weight updates have to be computed.

6.6 Layer Performance Analysis

This section takes apart the neural networks down to individual layers to assess why certain networks are faster than others and find potential to optimize models.

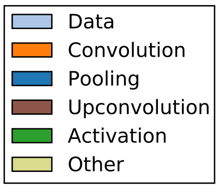


Figure 6.4: Layer-wise analysis legend.

6.6.1 SK-Net

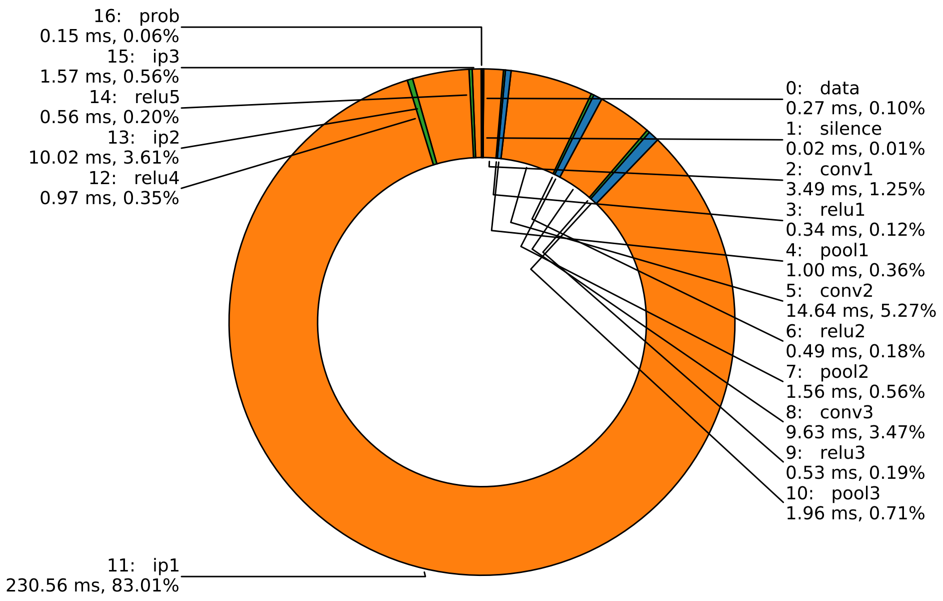


Figure 6.5: Layer-wise timings of SK-Net.

Looking at Figure 6.5, the convolution layers and especially the *ip1* layer (83 %) are responsible for the overall performance. There are many input (192) and output (1024) feature maps (see Section 3.3) and therefore a high computational complexity ($\mathcal{O}(f_{\text{out}} \cdot w^2 \cdot (2 \cdot f_{\text{in}} \cdot k^2 - 1))$) (see Section 5.5) on the *ip1* layer.

As the convolution layers only consist of a fast memory copy operation to arrange the data, so that matrix-matrix multiplications through an optimized BLAS become possible, and the SGEMM (single precision general matrix-matrix multiplication) call itself, the efficiency values in Table 6.6 and Figure 6.6 are direct proxies for how efficient the BLAS works on the devices.

Layers that operate with complexity $\mathcal{O}(f_{\text{in}} \cdot w^2)$, which includes all other layers used in the SK, USK and U networks, only contribute a small fraction to the total forwarding time (less than 1 % per layer).

Layer	**GFLOP**	**AMD OCL**	**nV OCL**	**nV CUDA**	**Intel OCL**
conv1	0.70	3.83 %	8.79 %	13.84 %	6.16 %
conv2	14.06	18.33 %	27.13 %	56.45 %	12.89 %
conv3	18.40	36.46 %	22.05 %	21.78 %	18.14 %
ip1	644.23	53.32 %	27.15 %	90.50 %	34.83 %
ip2	17.17	32.72 %	23.83 %	62.15 %	22.35 %
ip3	0.03	0.41 %	0.75 %	0.75 %	0.12 %

Table 6.6: FLOP efficiency for the convolution layers in the SK-Net. The values are relative to Table 6.1 FLOP performances.

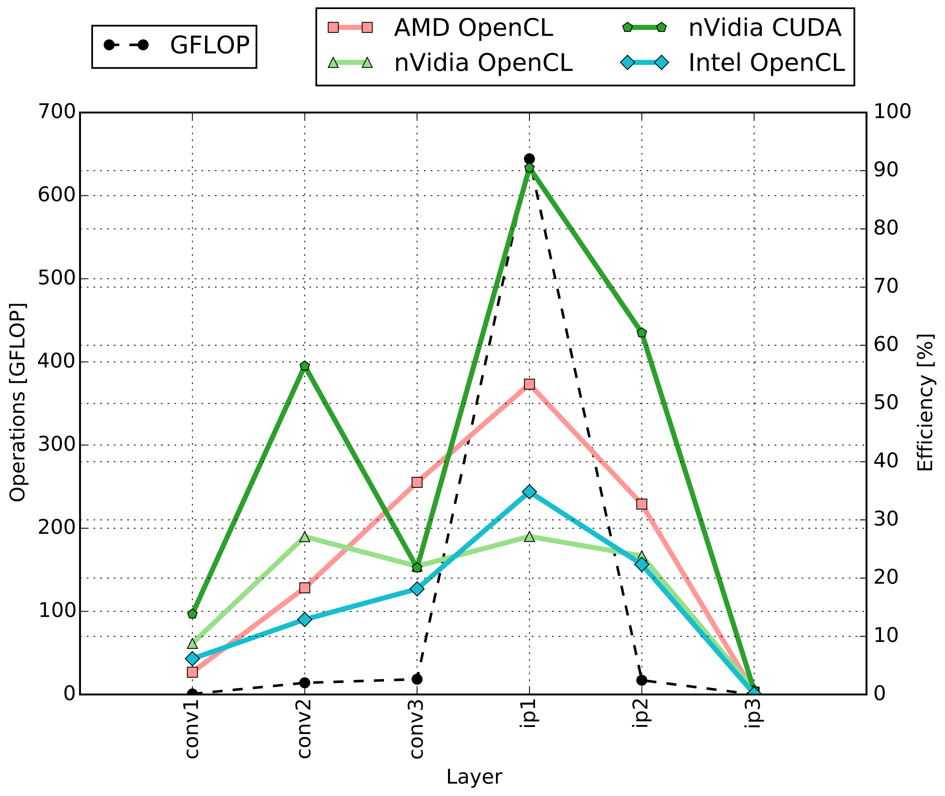


Figure 6.6: FLOP efficiency of the convolution layers in the SK-Net.

Even though *ip1* is the most expensive layer, it also is the most efficient (see Figure 6.6) on all devices, when using the BLAS which is most optimized. This means clBLAS for AMD, cuBLAS for nVidia and OpenBLAS for Intel.

It is important to note that clBLAS is not yet as optimized as cuBLAS for this type of matrix-matrix multiplications (see Section 5.5). Both GPUs currently perform worse than expected with the OpenCL backend. As both clBLAS and the *Project Greentea* are still quite recent projects, the performance is expected to increase with further optimizations in the future.

Other convolution layers are over 30 times less expensive and down to half as efficient. With small convolutions, the memory copy and kernel launch overhead lower the efficiency. Very small convolutions such as *ip3* and *conv1* may also not fully utilize the many threads available on GPUs (see Table 6.1). As the inefficient layers have short computation times here, they are not very important in optimization.

Only if the network mostly consists of such inefficient layers (which is not the case with SK-Net, but does apply to SW-Net), the number of feature maps and convolution sizes should be increased, if it improves the segmentation. Using minibatches ($n > 1$) can increase the GPU utilization on OpenCL when using multiple queues ($q > 1$) and the efficiency on CUDA when using cuDNN [20].

6.6.2 U-Net

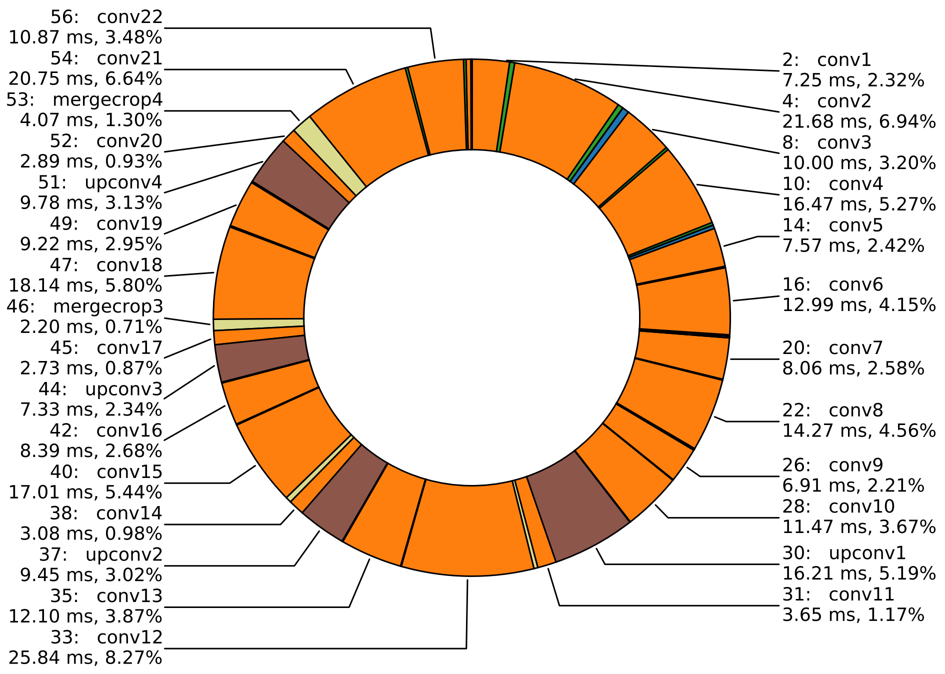


Figure 6.7: Layer-wise timings of U-Net, excluding layers that contribute less than 0.5 % of the total forwarding time.

Layer	GFLOP	AMD OCL	nV OCL	nV CUDA	Intel OCL
conv1	1.10	2.90 %	6.58 %	7.69 %	5.50 %
conv2	23.77	20.92 %	19.54 %	43.92 %	7.60 %
conv3	11.72	22.35 %	26.10 %	53.21 %	19.15 %
conv4	23.11	26.78 %	22.90 %	58.79 %	13.35 %
conv5	11.23	28.31 %	29.78 %	63.92 %	23.50 %
conv6	21.81	32.06 %	24.70 %	73.71 %	24.66 %
conv7	10.27	24.32 %	32.31 %	77.68 %	40.24 %
conv8	19.33	25.85 %	24.91 %	88.07 %	42.55 %
conv9	8.49	23.44 %	32.93 %	76.82 %	54.12 %
conv10	14.80	24.62 %	25.05 %	84.03 %	54.30 %
conv11	3.29	17.17 %	26.57 %	78.12 %	38.53 %
conv12	27.52	20.33 %	32.39 %	67.04 %	40.50 %
conv13	12.76	20.12 %	23.86 %	75.78 %	43.25 %
conv14	2.83	17.56 %	24.03 %	71.13 %	28.54 %
conv15	24.54	27.54 %	29.25 %	71.21 %	22.74 %
conv16	11.79	26.83 %	24.10 %	75.04 %	25.90 %
conv17	2.62	18.29 %	21.74 %	61.10 %	23.22 %
conv18	23.12	24.32 %	27.82 %	62.06 %	16.63 %
conv19	11.32	23.43 %	21.69 %	59.62 %	13.79 %
conv20	2.51	16.54 %	19.09 %	39.84 %	17.82 %
conv21	22.42	20.62 %	23.77 %	45.12 %	10.44 %
conv22	11.09	19.47 %	18.84 %	43.50 %	8.09 %
ip1	0.04	0.79 %	1.84 %	2.40 %	0.35 %

Table 6.7: FLOP efficiency for the convolution layers in U-Net. The values are relative to Table 6.1 FLOP performances.

In the U-Net, most convolutions have the same order of magnitude in complexity (see Table 6.7). The result is a very balanced network in terms of forward timings (see Figure 6.7). The balancing comes from trading feature map size against feature map count towards the middle of the network.

The upconvolution layers contribute each up to 5.2 % of the network forwarding time. This is less efficient than the optimum as only 4-nearest neighbor interpolation with constant weights is computed. This would not be more effort than a memory copy operation during forwarding and accumulating of the four nearest neighbor values during backward computation. Currently, it is implemented using a Caffe deconvolution, which is a reversed convolution layer. This layer already existed in Caffe, while no direct upsampling of feature maps is implemented. The convolution kernels are grouped, meaning each upsampling kernel only considers a single feature map. One advantage is that the deconvolution layer also allows adaptive weights and other interpolations such as bilinear upsampling.

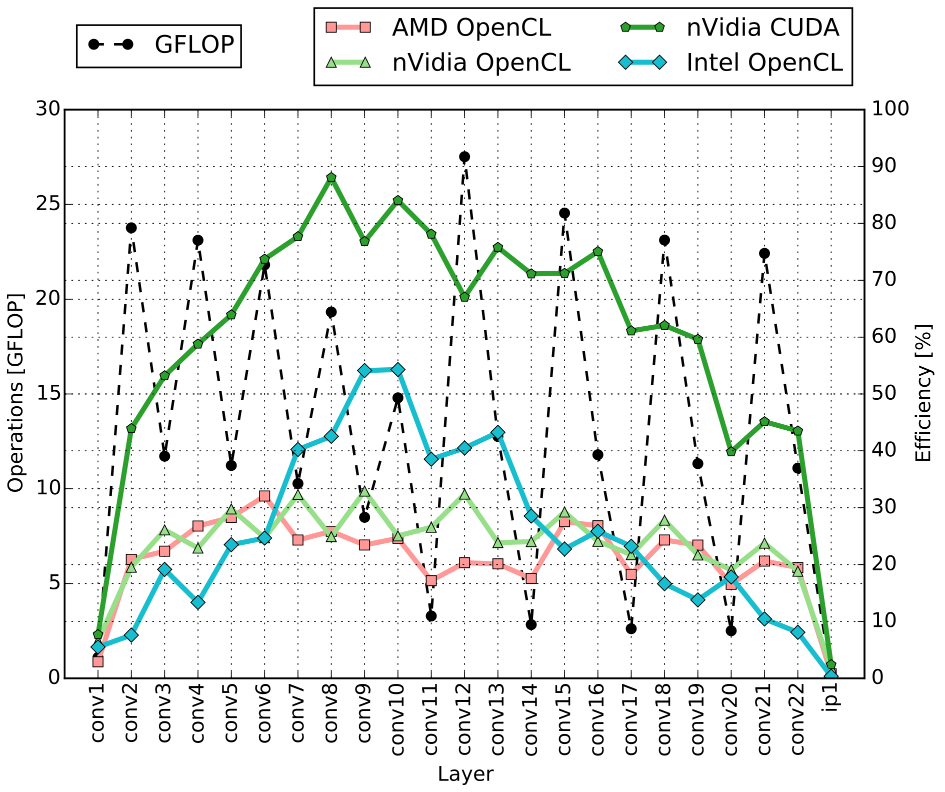


Figure 6.8: FLOP efficiency of the convolution layers in U-Net.

6.6.3 USK-Net

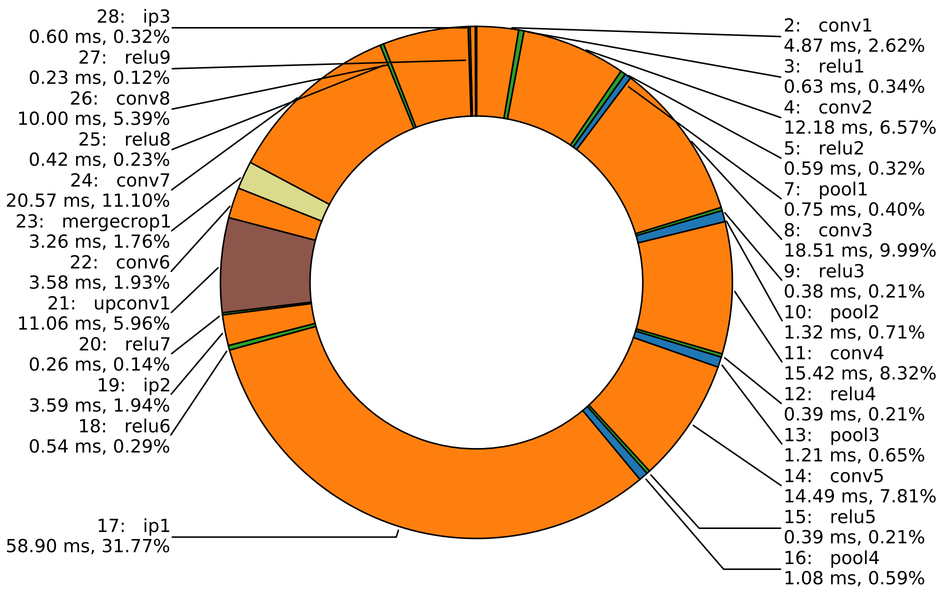


Figure 6.9: Layer-wise timings of USK-Net, excluding layers that contribute less than 0.1 % of the total forwarding time.

Layer	GFLOP	AMD OCL	nV OCL	nV CUDA	Intel OCL
conv1	0.64	2.51 %	4.97 %	7.08 %	5.43 %
conv2	13.75	21.55 %	19.25 %	42.11 %	8.04 %
conv3	26.25	27.06 %	20.14 %	62.58 %	11.73 %
conv4	21.81	26.99 %	21.78 %	62.61 %	12.57 %
conv5	18.92	24.93 %	27.45 %	63.93 %	13.06 %
ip1	141.76	45.93 %	31.25 %	86.64 %	27.82 %
ip2	4.43	23.53 %	28.14 %	72.69 %	42.59 %
conv6	4.42	23.56 %	21.29 %	63.39 %	31.44 %
conv7	29.44	27.31 %	27.88 %	61.29 %	17.26 %
conv8	9.66	18.44 %	20.45 %	46.17 %	7.34 %
ip3	0.02	0.53 %	1.73 %	1.73 %	0.36 %

Table 6.8: FLOP efficiency for the convolution layers in USK-Net. The values are relative to Table 6.1 FLOP performances.

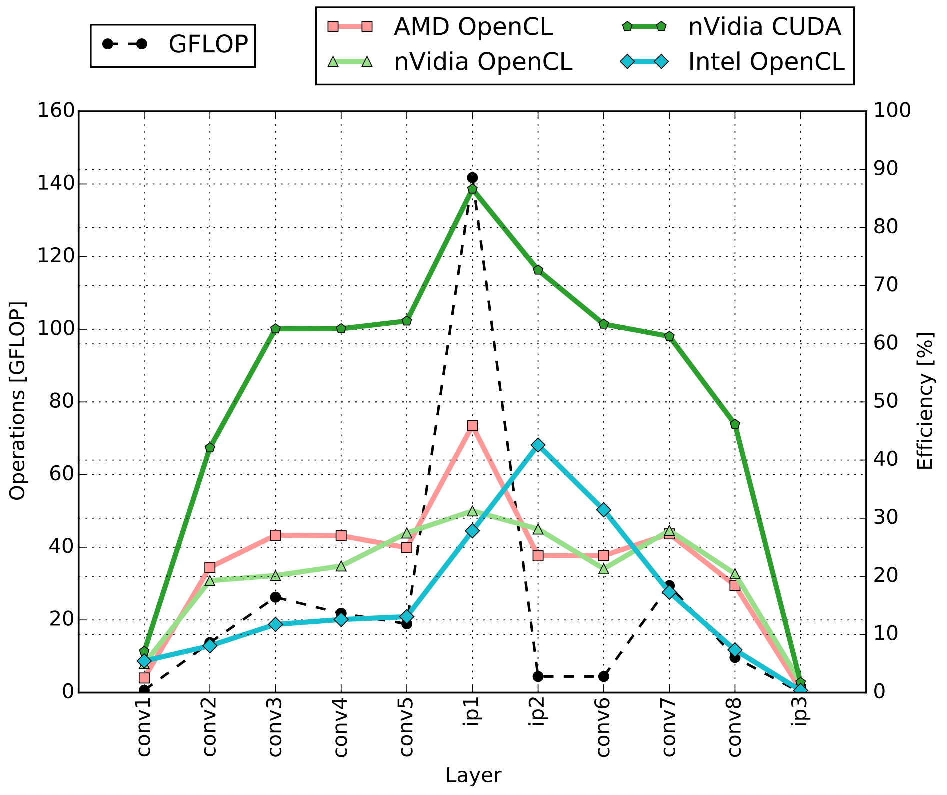


Figure 6.10: FLOP efficiency of the convolution layers in USK-Net.

In the USK network, the *ip1* layer is again the most expensive and efficient one, but the network is more balanced as *ip1* only contributes 32 % of the forwarding time, compared to 83 % in the original SK-Net. Efficiency and forwarding times are, as expected, a mixture of the SK- and U-Net.

Except for *ip3* and *conv1*, all layers have a relatively high efficiency. The inefficient layers go from $f_{\text{in}} = 64$ to $f_{\text{out}} = 2$ and $f_{\text{in}} = 3$ to $f_{\text{out}} = 64$ respectively. This will directly result in matrix multiplications with matrices that have strongly non-square shapes and are therefore less efficient (see Section 5.5).

6.7 NUMA Issues

An issue that came up testing the OpenCL hybrid backend (see Section 5.4.2) was that the performance did not scale as expected with systems that have more than one CPU. Such systems have non-unified memory access (NUMA) because the CPUs share one address space for memory, but every processor has its own cache and memory interface. Accessing data across the other CPU comes with a large performance penalty. Compute kernels, such as the matrix-matrix multiplication in the BLAS library or the custom OpenCL kernels, cause the threads to work on adjacent data. This means a write operation of one CPU is likely to invalidate cache lines across both CPUs. At this point, the synchronization overhead seems to become larger than any speedup of having additional cores working on the algorithms.

Layer	i7-4790K	2x E5-2697v3	Speedup
conv1	22.19 ms	47.02 ms	0.47
relu1	2.68 ms	5.08 ms	0.53
pool1	14.65 ms	17.17 ms	0.85
conv2	213.16 ms	176.96 ms	1.20
relu2	4.94 ms	14.04 ms	0.35
pool2	40.06 ms	44.40 ms	0.90
conv3	198.15 ms	238.91 ms	0.83
relu3	5.85 ms	17.45 ms	0.34
pool3	57.08 ms	59.05 ms	0.97
ip1	3612.29 ms	4168.21 ms	0.87
relu4	13.59 ms	47.62 ms	0.29
ip2	150.03 ms	187.05 ms	0.80
relu5	5.90 ms	24.10 ms	0.24
ip3	56.25 ms	77.51 ms	0.73
prob	0.48 ms	4.88 ms	0.10

Table 6.9: 4 core i7-4790K versus 2 CPU 28 core (NUMA) E5-2697v3, comparing SK network layerwise forward timings.

Each CPU has 14 cores, which gives 28 cores for the whole system. The number of threads for the frontend was set to 14, which gave the best performance by keeping at least the BLAS library temporarily tied to one processor by the operating system's scheduler. The OpenCL backend, which also allocates the memory, used 56 threads and allocated memory on both interfaces.
Table 6.9 shows the impact of having NUMA issues, taking the SK network as

an example. The layers are all sufficiently parallelized, which is evident when looking at Figure 6.6. Differences of the CPU architectures are also not a possible explanation as both processors are based on the same instruction sets and generation. The memory interface should also be fast enough to keep up with the computations (see Table 6.1).

Correcting for processor frequency, the speedup should be up to a factor 4.55$\times$ using all 28 cores (ReLU and pooling layers) and up to 2.275$\times$ using only one processor (inner product and convolution layers). The effective speedups measured are much slower:

- Speedups between 0.1$\times$ and 0.97$\times$ on element-wise layers with complexity up to $\mathcal{O}(f_{\text{in}} \cdot w^2)$. Cache invalidation between the two processors (28 cores) seems to be dominant. The element-wise kernels run on the OpenCL backend.
- Speedups between 0.47$\times$ and 1.20$\times$ on convolution layers with complexity up to $\mathcal{O}(f_{\text{out}} \cdot w^2 \cdot (2 \cdot f_{\text{in}} \cdot k^2 - 1))$ (matrix-matrix multiplication). The effects of OpenBLAS running on 14 cores and sub optimal memory allocations are dominant. It is a proxy for how the matrix-matrix multiplications used in all convolutions perform.

To get the expected speedup, the two processors need to be presented to the Caffe library as two separate devices. Then the library can be used in two individual instances. As the OpenCL hybrid backend uses two separate parallelization mechanisms (OpenCL kernels and a parallelized BLAS), two solutions would need to be applied:

- The Caffe frontend needs to be tied to the cores of one CPU, so that the BLAS library does not show NUMA issues.
- The OpenCL backend needs to split up the processor setup into sub-devices using device fission. The splitting rule needs to be that all cores belonging to one processor (tested by cache affinity) are tied to the same sub-device. Only one is then used per Caffe instance. Device fission is an extension to OpenCL that is already available (*cl_ext_fission* [26]).
- The cores used in the frontend and selected sub-device need to be the same.

Due to not having permanent access to a system with two processors and OpenCL installed, I did not have time to test out the solutions. Implementing the solutions remains as an open issue at the time of the project.

6.8 Alexnet

For comparison how the backends and devices perform on a widely used network for image classification that uses minibatches (with $n = 10$, $w = 227$, $f_{\text{in}} = 3$) and multiple OpenCL queues ($q = 8$), the Alexnet [1] included in the Caffe library was also evaluated.

Device	Backend	Forward	Backward	Total	Forward Speedup
Intel	CPU	452.807 ms	355.553 ms	808.420 ms	1.00
Intel	OpenCL	238.188 ms	152.811 ms	391.160 ms	1.90
AMD	OpenCL	52.348 ms	128.009 ms	180.420 ms	8.65
nVidia	OpenCL	26.055 ms	47.446 ms	73.540 ms	17.37
nVidia	CUDA	20.899 ms	16.730 ms	37.718 ms	21.66

Table 6.10: Alexnet timings, average forward-backward pass over 50 iterations.

The CUDA backend has an advantage over the OpenCL backend in terms of speed, but is less versatile. The AMD GPU seems to be less efficient with minibatches and smaller matrix-matrix multiplications than the nVidia GPU, which is why the AMD GPU performs worse than the nVidia GPU on the same backend. With the SK-, USK- and U-Net (Section 6.5), the AMD GPU performs better using the same (OpenCL) backend on both GPUs.

Especially the backward computation is much slower (by a factor of three) using OpenCL instead of CUDA. The algorithms used are the same, therefore this difference is difficult to explain. Possibly, different optimizations need to be applied in the backward step of convolution layers, which can have a sequential bottleneck by adding up the gradients over the minibatch.

Using the OpenCL hybrid backend on the Intel CPU outperforms the (legacy) CPU backend by almost a factor of two. The speedup comes from parallelization of *Greentea*'s OpenCL compute kernels, which are only single-threaded in the Caffe CPU backend. The BLAS library used for convolutions is multithreaded in both cases.

Chapter 7

Results

7.1 Introduction

In this chapter, the results of training the models from Chapter 3 are presented.

The evaluation is based on:

- Two data sets (DS1 and DS2: see Chapter 2).
- Three models (SK, U, USK: see Chapter 3).
- Two training loss functions (Softmax and Malis).
- One processing loss function (Softmax).
- Three training configurations (Softmax, Malis and Softmax + Malis).
- 10'000 training iterations per configuration, respectively 20'000 for combined training.
- Three error objectives (rand, warping and pixel error).

The amount of training iterations was chosen so that training of all 18 combinations was feasible during one week on two AMD W9100 GPUs, providing a total of 10 TFLOP/s [22].
The training data is not very large in both cases (see Chapter 2) and thus the loss always converged in under 10'000 iterations for each training method. It is possible that some trainings did overfit as no early stopping was applied. Technically, the networks did not get to see the same amount of examples during training even though the iterations are the same, as the chosen output sizes of each network were set differently. However, due to gradient accumulation, different learning rates and weight initialization it is hard to estimate the effect of the amount of labels seen. As all networks converged to a stable loss, it should negligible.

7.2 Analysis on DS1

7.2.1 Training

The training parameters used on DS1 were set to not use the error masking functionality. Masking usually gives thicker membrane (background) labels when used with Softmax by balancing out the amount of background error against foreground (cell interior) error.

Malis loss was run without using a patch prior for preferring training patches based on their label histogram. Softmax loss on the other hand was used with the patch prior enabled. This is justified by the different characters of the loss functions: Softmax computes a per-pixel error while Malis gives errors at problematic pixels only, which can be very concentrated on a few pixels (see Section 5.3.2).

In the patch pre-processing step, the images were enhanced with CLAHE (contrast limited adaptive histogram equalization) and normalized in the range of $[-1.0, 1.0]$.

To get more training data, the images were blurred with a randomly chosen 5 by 5 Gaussian kernel. The training patches were also rotated to multiples of 90° and mirrored randomly (horizontal and vertical).

Details of the pre-processing and label priors are described in Chapter 4. The exact training parameters are stored as prototxt configuration in the Caffe Neural Models repository [9].

Interestingly, it was not possible to start training of the SK network with Malis loss directly. The loss did not converge, and the output feature maps drifted to being classified as only foreground or only background. Therefore, the weights of the network in *Malis only*-training were initialized using 4000 iterations of Softmax training first. This was the lowest number of iterations where the training converged to a small loss afterwards. The other network architectures did not show such a behavior and trained well when starting with Malis directly.
This is related to weight initialization as well as the fact that Malis works better on bigger training patch sizes. Training of SK was limited to 128 by 128 pixels output, which is much less context for Malis to work on than with USK and U with 512 by 512 and 388 by 388 pixels respectively.

7.2.2 Numerical

Rank	Network	Loss Function	Rand	Warping	Pixel
1.	USK	Softmax + Malis	**0.031251499**	0.000423193	0.040650392
2.	SK	Softmax + Malis	0.043871103	**0.000324726**	0.052724789
3.	USK	Malis	0.045955948	0.000578165	0.058407081
4.	U	Softmax + Malis	0.050799468	0.000487328	0.044159558
5.	SK	Malis	0.063852001	0.000386953	0.055678171
6.	USK	Softmax	0.067068677	0.000418186	**0.030634890**
7.	U	Malis	0.078736573	0.000520945	0.059127919
8.	U	Softmax	0.091736036	0.000595331	0.033451667
9.	SK	Softmax	0.123334489	0.000549316	0.034416625

Table 7.1: DS1 error evaluation (lower is better).

Explanations for the Tables 7.1 and 7.2:

- Rank: The internal ranking, as indicated by the rand error.
- Loss Function: Softmax indicates 10'000 iterations with the Softmax loss function. Malis indicates 10'000 iterations with the Malis loss function. When both are indicated, the training was executed with 10'000 Softmax and then 10'000 Malis training iterations.
- Rand, warping and pixel: Error metrics, as proposed by Jain *et al.* [27] and used in the ISBI 2012 challenge [11]. A script for evaluation is available for Fiji [13] in the Caffe Neural Models repository [9].

The ranking on data set DS1 is as expected: Taking the average rank, USK-Net performs better than SK-Net, which is more precise than U-Net.
The same goes for training methods: Using Softmax + Malis minimizes the rand error better than using Malis only, and Softmax outperforms Malis on pixel precision. On one hand, this is because the Malis criterion improves the rand error by penalizing merge and split errors. On the other hand, training with Malis will also decrease the pixel accuracy, which can be seen when inspecting the result visually. As Softmax training initializes the networks better than Malis, the best training method is to start with Softmax and then transit to Malis for fine tuning.
Obviously, the USK network architecture performed best on both pixel accuracy and rand error. Only for the warping error, the much slower SK-Net (see Section 6.5) performs best.

7.2.3 Visual

The visual analysis is based on the image number 2 of the DS1 stack (see Section 2.1). It includes a glia cell (Figure 7.1, label A) which is considered as background. The thickness of it makes it harder to label correctly, especially with Malis loss.

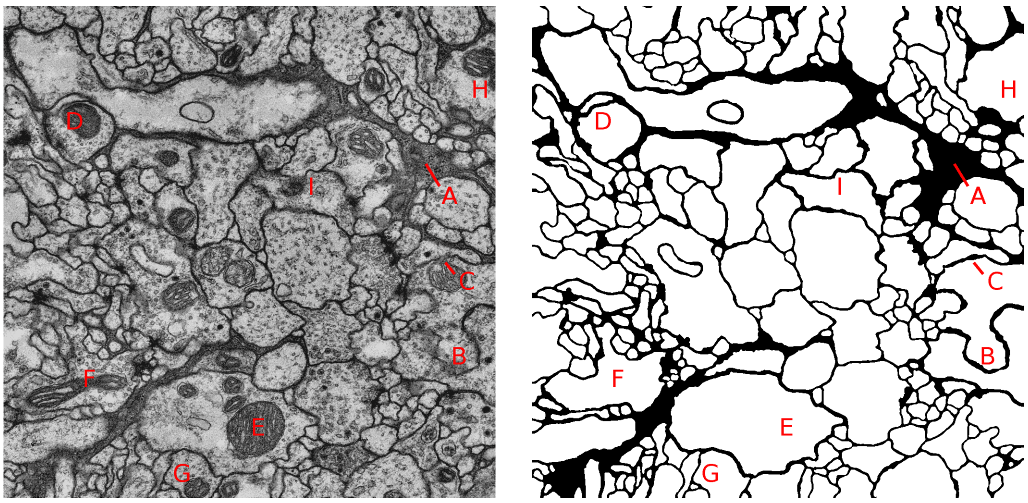
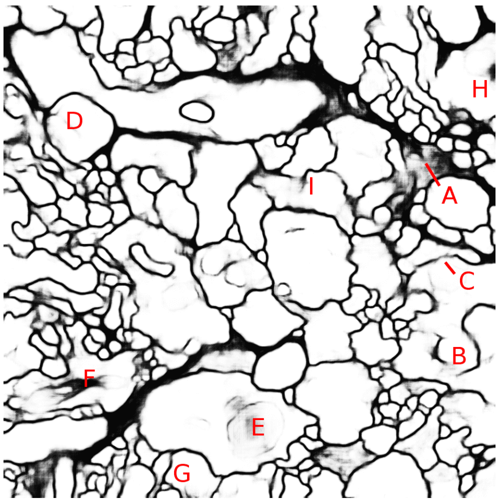
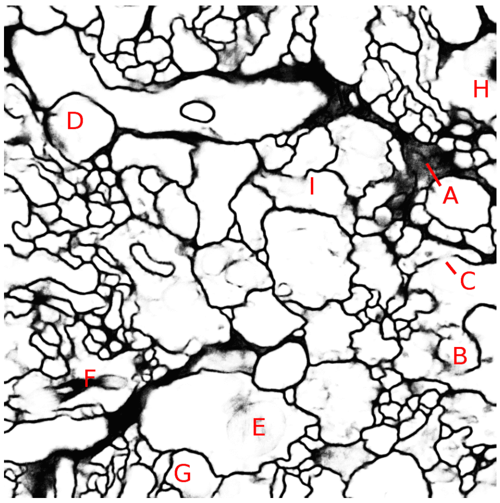


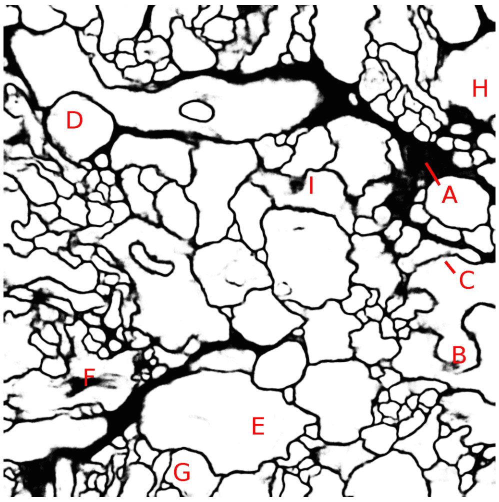
Figure 7.1: DS1 ssTEM raw and corresponding ground truth, 1024 by 1024 pixels (Source: ssTEM [8], [9]).



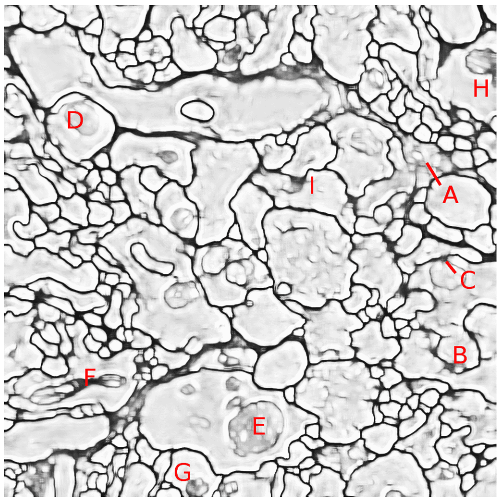
(a) SK network, Softmax.



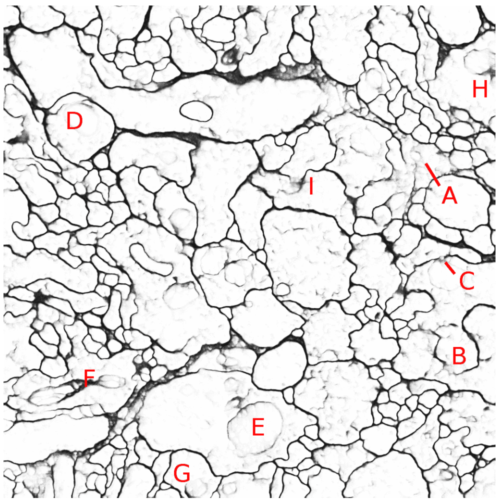
(b) U network, Softmax.



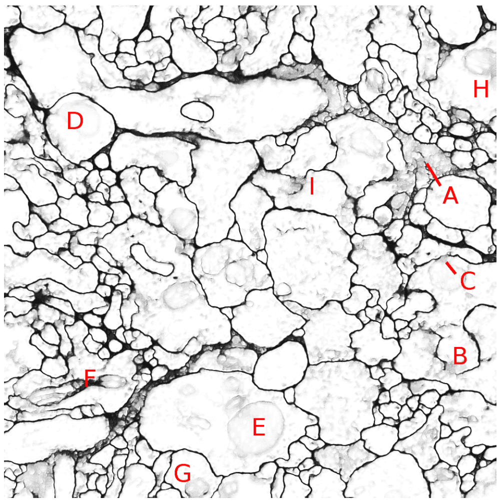
(c) USK network, Softmax.



(d) SK network, Malis.

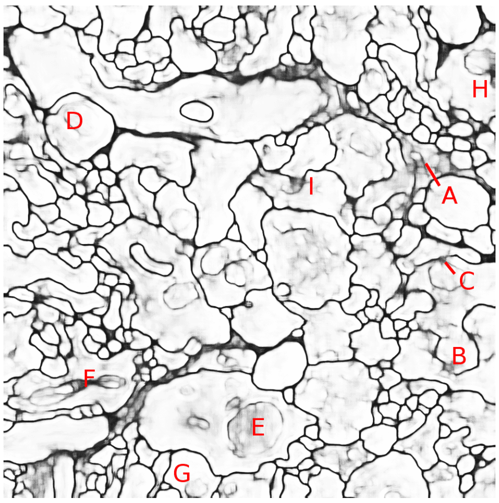


(e) U network, Malis.

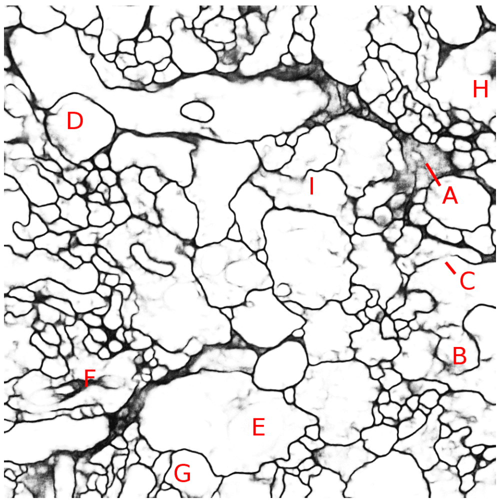


(f) USK network, Malis.

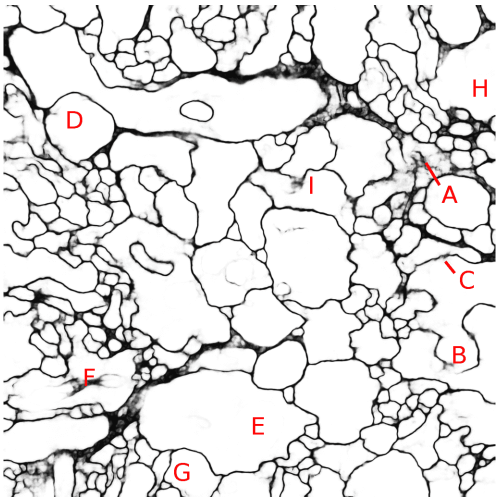
Figure 7.2: Visual comparison of all results in Table 7.1.



(a) SK network, Softmax + Malis.



(b) U network, Softmax + Malis.



(c) USK network, Softmax + Malis.

Figure 7.2: Visual comparison of all results in Table 7.1.

Explanation to the labels in Figure 7.2:

(A) Glia cell that should be considered background. Malis loss does focus on separating foreground objects, so this part gets only labeled correctly in the trainings that use Softmax only (a, b and c).

(B) Diffuse membrane with light texture, which is hard to separate from cell interior. The separation is correct only in SK + Malis (d), while all other configurations lead to connected cells, although all trainings with Malis (d to i) show uncertainty, which could be sufficient to segment correctly. Diffuse parts are always on the edge of being pushed to foreground or background, which makes the segmentation fragile.

(C) Membrane with close proximity to a mitochondrion. Except for U networks that have been trained with Softmax (b and h), this gets labeled correctly. The downsampling of U-Net is sub-optimal here when pixel error (without scaling) instead of foreground separation is the training objective.

(D) Same case as label C, but here the mitochondrion is labeled with high certainty by all networks, leading to mislabeling of the nearby membrane.

(E) Removing an isolated mitochondrion with high certainty. The USK network (c and i) performed best on this task. Malis trainings are usually worse, again because Malis does not focus on pixel accuracy.

(F) Mitochondrion with long, thin structure. It consequently leads to a wrong classification as membrane because of its shape. This is not an issue, as isolated misclassifications within a cell can be rejected on a higher level when reconstructing the connectome.

(G) Mitochondrion close to the image border. As the training and classification uses border mirroring to make up for the missing context, it can lead to more errors near the border, independent of the network architecture.

(H) Same case as label G. Softmax + Malis on USK-Net performed best on removing the mitochondria on E, F, G and H.

(I) Diffuse membrane with a dark texture. It is expected that this gets completely classified as membrane. However, this was not the case on all networks and trainings. The worst case applies on U-Net (b and h), where the cells are almost connected.

As expected, membranes get labeled thinner with Malis as this error criterion is stopping to provide loss at membranes as soon as two cells are sufficiently separated (see Section 5.3.2). Even though the separation border is thin, training with Malis separates cells very well and scores best on both DS1 (Table 7.1) and DS2 (Table 7.2).

The visual results match the numerical evaluation. USK performs best on pixel error with Softmax and best on rand error with Softmax + Malis. However, when using Softmax, there is not a huge difference in pixel error and visual results between the networks. Malis does not work very well on SK networks (d), leaving a lot of uncertainty in the prediction. It can still be a good segmentation, as thresholding of gray scale values can be applied. The numeric evaluation [13] does test different thresholds, which is why the score is rather good (see Table 7.1) despite the uncertainty.

7.3 Analysis on DS2

7.3.1 Training

Training on DS2 was similar to training on DS1. The main difference is that the patch prior was not used on both Softmax and Malis. Instead, the error masking was enabled when using Softmax, which gives thicker borders. This is motivated by having input images which are slightly more blurred and thus the cell membranes are less sharp and harder to distinguish from cell interior.

Here, the SK network already converged using 2000 training iterations of Softmax before switching over to Malis. Starting with Malis directly was also not possible.

7.3.2 Numerical

Rank	Network	Loss Function	Rand	Warping	Pixel
1.	SK	Softmax + Malis	**0.060110507**	**0.000495529**	0.106053779
2.	USK	Malis	0.085927407	0.000848007	0.110390552
3.	SK	Malis	0.086975487	0.000572968	0.107365432
4.	SK	Softmax	0.087380844	0.000585556	0.075981492
5.	U	Softmax + Malis	0.097356122	0.000940704	0.101856259
6.	USK	Softmax	0.102450251	0.000851440	**0.073163943**
7.	U	Malis	0.121984927	0.001038742	0.111951817
8.	USK	Softmax + Malis	0.128440534	0.000858688	0.101919859
9.	U	Softmax	0.148045042	0.001293564	0.083922396

Table 7.2: DS2 error evaluation (lower is better).

Interestingly, the USK-Net with Softmax + Malis loss training performs unexpectedly worse on the dataset DS2 than on DS1, where it performed best. What is common for both DS1 and DS2 is that the USK network combined with Softmax training performs best on the pixel error. Overall, this data set has a ranking much harder to explain than on DS1. Visual inspection reveals that the USK-Net was often overconfident when labeling the cell interior, which connected cells that should be separated. On the DS1 data set, this did not happen.

Network	Rand	Warping	Pixel
U	0.0382	0.000353	0.0611

Table 7.3: (Source: Ronneberger *et al.* [2]).

Finally, the results of U-Net obtained by Ronneberger *et al.* [2] (Table 7.3) could not be reached, probably due to having used less transformations to extend the training data. It is not inherently clear if the U network would perform better than SK and USK given the bigger training data set.

Ideas to improve training include:

- Weight maps to scale the loss instead of only masking it.
- Add elastic deformations, shifting and scaling instead of only rotation and mirroring to increase the amount of training data.
- Experiment with different loss functions than Malis and Softmax, or alternate between them during training.

7.3.3 Visual

The visual analysis is based on the image number 16 of the DS2 test stack (see Section 2.2).

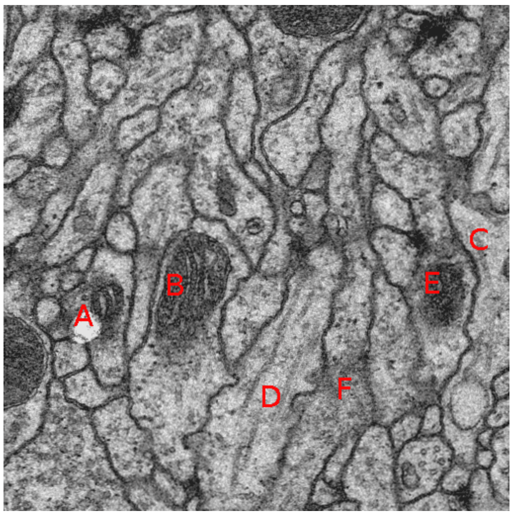


Figure 7.3: DS1 raw image, 512 by 512 pixels (Source: ISBI challenge [11], [9]).

Explanation to the labels in Figure 7.2:

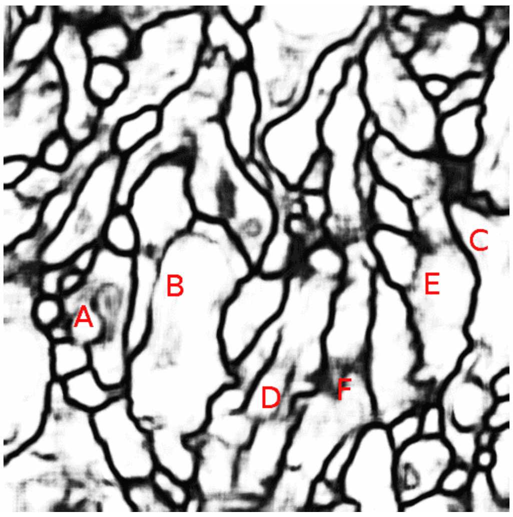
The visual results of Softmax training (a to c) have thicker membrane labels than on DS1. This is because the error masking was enabled here and, as a result, the network has seen the same amount of error pixels for both foreground and background.

(A) A bright spot, which is an error in the electron microscopy image. The membrane affected by it is misclassified by all networks and trainings. The local contrast enhancement with CLAHE did not help here.

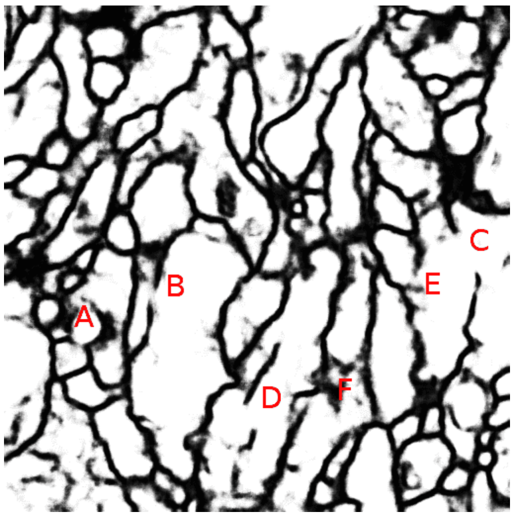
(B) Mitochondrion with close proximity parallel to a cell membrane. The USK network removes the membrane with all trainings (c, f and i). U-Net (b, e and h) performs a little better, but still merges the two cells. Only the SK network does it correctly (a, d and g), but has some uncertainty on the mitochondrion instead (d).

(C) Diffuse section of a cell membrane. This is not an issue on most training/architecture combinations, except for U-Net with Softmax (b and h).

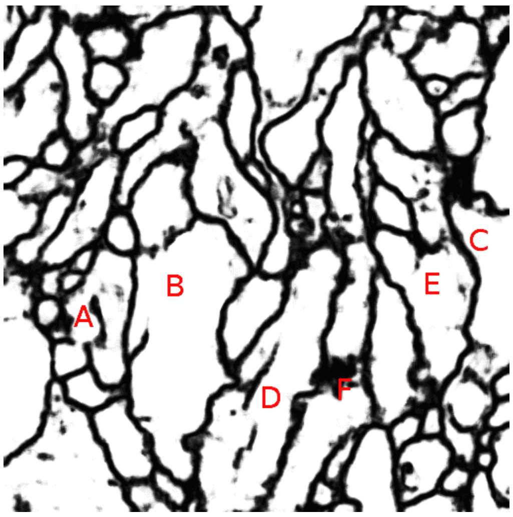
(D) Oriented structures, even when faint, are partially labeled as membrane when sharp enough and of similar thickness as the membrane. This is no issue when isolated within the cell and not cutting through a cell that should be connected. With convolutional networks, this is hard to impossible to label correctly. It would require more high level knowledge of the object, such as if the predicted membrane is enclosing a cell or not.



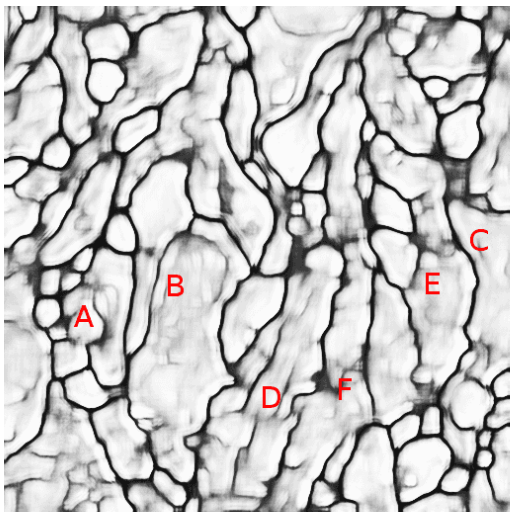
(a) SK network, Softmax.



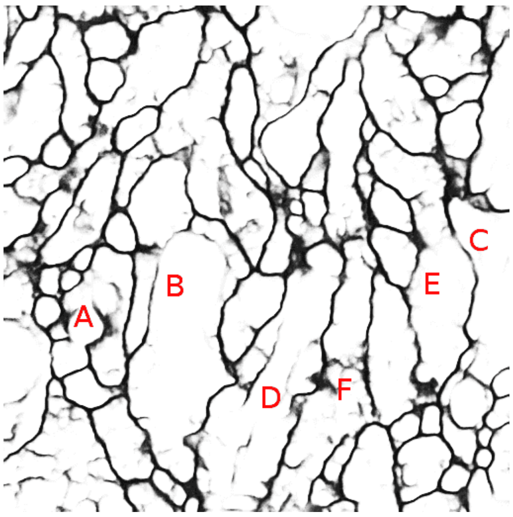
(b) U network, Softmax.



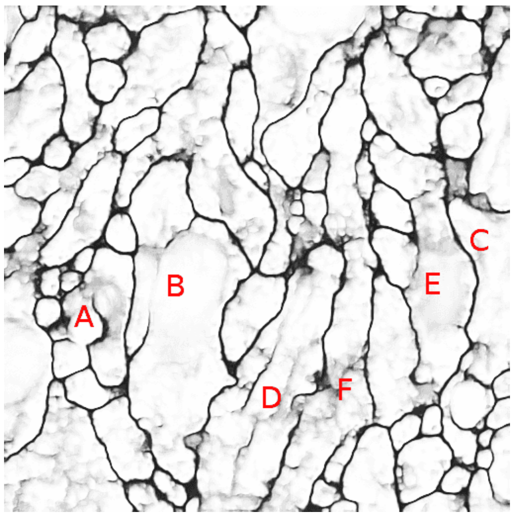
(c) USK network, Softmax.



(d) SK network, Malis.

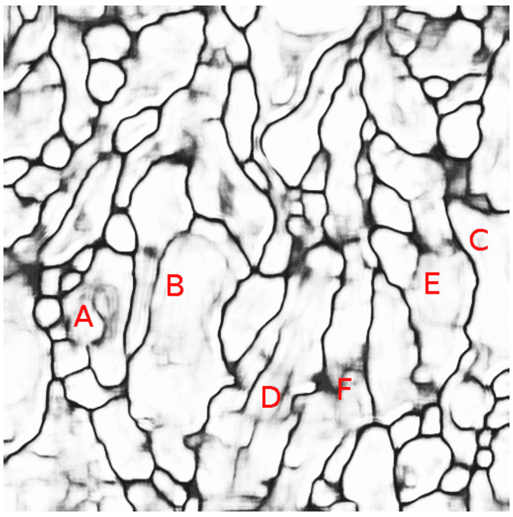


(e) U network, Malis.

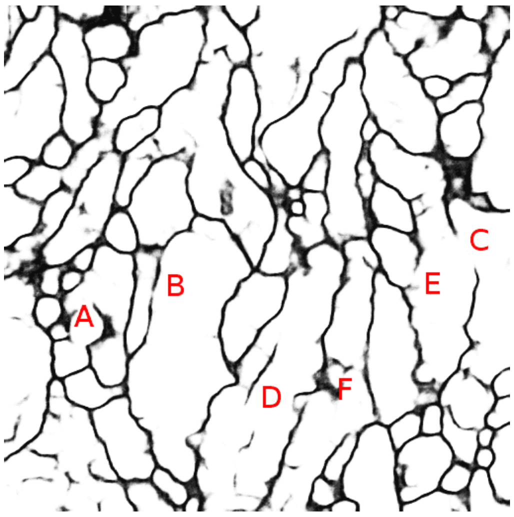


(f) USK network, Malis.

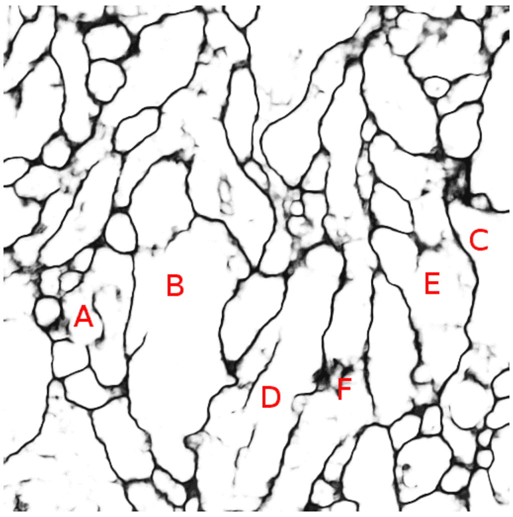
Figure 7.4: Visual comparison of all results in Table 7.2.



(a) SK network, Softmax + Malis.



(b) U network, Softmax + Malis.



(c) USK network, Softmax + Malis.

Figure 7.4: Visual comparison of all results in Table 7.2.

(E) Diffuse mitochondrion. The same situation as with B applies. Especially U- and USK-Net with Softmax training (b and c) get it wrong.

(F) Diffuse cell interior that is similar to the membrane texture. All networks see a membrane connection through this area. In the training data there are some examples of diffuse membranes, so the networks have slightly overfitted on the training data for this case.

Chapter 8

Conclusion

8.1 Research Time Line

An overview of the research time line, in order to give a context on what shaped the objectives and decisions made during the project:

From	**To**	**Activity / Event**
05.11.2014	05.11.2014	Collaboration request at UZH INI.
09.11.2014	22.01.2014	Discussing ideas with Dr. Jan Funke.
14.12.2014	14.12.2014	Hongsheng Li *et al.* paper released (SK kernels) [7].
07.02.2015	07.02.2015	Research proposal finished and accepted.
23.02.2015	23.02.2015	Research beginning.
26.02.2015	06.03.2015	Getting the sliding window network to work [10].
08.03.2015	18.04.2015	OpenCL backend development [5].
10.04.2015	21.04.2015	Discussing the project with AMD [25].
22.04.2015	22.04.2015	Arrival of AMD's W9100 GPUs (hardware sponsoring).
19.04.2015	19.04.2015	Pull request of the modified Caffe to BVLC [4].
09.05.2015	09.05.2015	Public release of the Caffe Neural Models [9].
09.05.2015	09.05.2015	Public release of the Caffe Neural Tool [16].
15.05.2015	15.05.2015	Ronneberger *et al.* paper released (U-Net) [2].
20.05.2015	25.06.2015	Testing of U-Net and design of USK-Net.
27.06.2015	12.07.2015	Collaboration at HHMI Janelia, Virginia, USA [28].
29.06.2015	14.07.2015	Implementing Malis loss and N-D SK kernels for Caffe.
12.07.2015	15.07.2015	Critical source code development finished.
13.07.2015	20.08.2015	Writing the report and final evaluation experiments.
24.08.2015	-	Post-research support of *Project Greentea* and ongoing development in collaboration with AMD, HHMI Janelia and the Caffe community.

8.2 Implications

The first idea for the research project was to implement strided kernels. However, with the release of the Hongsheng Li *et al*. [7] paper, the problem already got solved. We got their source code and I was able to translate existing sliding window networks to strided kernel networks.
These events lead to a shift of focus to implement the OpenCL backend and support a variety of hardware. This was important to see how existing CPU clusters and AMD GPUs could be used instead of only nVidia GPUs.

A nice side effect of completely re-writing the whole Caffe library to OpenCL was gaining a complete understanding of the library, the bottlenecks, how all layers work and what the most important objectives for optimizations and network design are.

It turned out that running networks across devices does not give an advantage in the case of SK, U and USK architectures, as perfect scaling is possible when running independent instances of the network on each device. This only requires from the devices to have enough memory to hold the networks. This assumption was met when AMD's W9100 GPUs became available to me.

SK networks did not scale as desired and up to 100'000 pixel classifications per second were only about 1/10th of the desired speed. The original ideas to speed up the layers of the SK network by using methods such as multi device execution, Fourier transform convolutions or direct convolutions did not work.
With the release of the Ronneberger *et al*. paper [2], the research focus was shifted to analyzing the U-Net approach, which is able to classify up to one megapixel per second.
Training of U-Net was more difficult than SK-Net and thus I tried to implement my own network architecture based on the findings of SK-Net and U-Net, which resulted in the experimental USK-Net. The USK-Net performs similarly to U-Net and produced better results with small training data sets (see Chapter 7).

Looking at ISBI 2012 results [11] and their test metrics, as well as the fact that one of the authors of the Malis criterion [3], Dr. Srinivas Turaga, was at HHMI Janelia for collaboration, lead to the development of an additional loss layer for Caffe (see Section 5.3.2).

Finally, the last feature implemented before freezing the source code was N dimensional strided kernel support for max pooling and convolution layers, as this was a feature requested by Dr. Stephan Saalfeld and Dr. Srinivas Turaga at HHMI Janelia. This can be used to run modified SK, U and USK network architectures on 3D blocks of volumetric-isotropic data sets, or even 3D over time (4D).

8.3 Difficulties Encountered

The obvious difficulty was to keep up with the general research in pixelwise classification of images, as important papers [7], [2] were released during the project research. A shift of focus from the original plans were required a few times. This includes taking into account new results and dropping planned approaches.

It was also a lot of work to keep up with the changes of the Caffe library [4], as they changed many core aspects such as network file format and shape specifications for memory blobs. This broke compability with existing code from Honghsheng Li *et al.*'s approach [7] as well as the existing sliding window network [10], [9]. Constantly pulling new changes from the BVLC master branch [6] and adapting my own branch to those changes was necessary. The benefit gained by doing this is that backwards compatibility to the official version is always guaranteed and that my own branch was ahead during the whole scope of the project.

With programming the Caffe Neural Tool, the diversity of formats for labels and input data was complex to handle. Especially loading and storing *TIF* pictures that can have a variety of pixel formats and support stacking multiple images in a single file can be tricky.

At last, it was not always obvious why a network does learn the expected features or not. Training parameter tests require up to ten hours of training on a GPU, which is very acceptable during production, but rather cumbersome during debugging. Evaluation and training of the networks for numeric results was only possible after freezing critical parts of the source code (computation kernels and layer implementations), because the results can differ after fixing bugs and other changes of the library. This resulted in having only two weeks left for this stage.

8.4 Reproducibility of Results

The results obtained in this report are guaranteed to be reproducible by the use of the following software pipeline, using CUDA or OpenCL hardware equivalent to the hardware used in this report.

Repositories belonging to *Project Greentea*:

- Caffe [5]

  URL: `https://github.com/naibaf7/caffe`

  Commit checksum: `f84c2a4fb8d633bc7d8fc9771eb06a3cf2215212`

- Caffe Neural Tool [16]

  URL: `https://github.com/naibaf7/caffe_neural_tool`

  Commit checksum: `780e7dd72e4f88c80729e2b33d6c0137d479016a`

- Caffe Neural Models [9]

  URL: `https://github.com/naibaf7/caffe_neural_models`

  Commit checksum: `dbb06d8352aa9c2ba99458cb9e9068500ebacc11`

As long as the ISBI 2012 challenge is ongoing, the data set DS2 results can be reproduced on their website [11]. The training and test data for the data set DS1 remains included in the Caffe Neural Models repository.

8.5 Outlook

In this outlook I give a brief introduction on future plans for the improved Caffe version [5], extended use cases for the models [9], missing features for the Caffe Neural Tool [16] and ideas that did not fit into the time scope of the project.

8.5.1 Device Abstracted Backend

Currently, the OpenCL, CUDA and (legacy) CPU backend are implemented side-by-side and there is quite some code duplication in the Caffe library. To support future multi-device training methods and remove redundancy, the backend should be further unified so that the only remaining code duplication resides with the actual compute kernels used inside the layers. This will minimize bugs that occur on only one backend and make software verification much easier to handle. It will also shorten the time required for newly developed layers to become available on all devices.
At the time of the project, the improved Caffe library [5] drops full support on the legacy CPU backend in favor of an OpenCL hybrid solution (see Section 5.4.2) on CPUs. Some tuning to work correctly on NUMA processors (see Section 6.7) is still required. The CPU backend remains as a fallback for layers that do not work on OpenCL and CUDA, such as the Malis loss criterion (see Section 5.3.2).

8.5.2 Improving Training Data

As a first step to improve results, all network architectures should be evaluated using more training data, acquired artificially or from more ground truth.
It is advisable to try out all models on a given data set, as there is no clear winner among the networks. Results may vary strongly as the numerical analysis revealed (see Section 7.2 versus 7.3).

8.5.3 Parameter Grid Search

The network architectures presented here are only examples of a whole class of possible networks. Many parameters such as kernel sizes and how SK networks are combined with U type networks can be evaluated. Especially deep multi-path networks can be useful, merging the feature maps of different architectures before making the ouput label predictions. The new USK architecture is such an example.

8.5.4 Testing of Volumetric Architectures

Depending on the data set, SK, U and USK networks can be configured in many more ways for 3D than for 2D. For example, the depth direction is likely to have less physical resolution than the width and height dimension, due to how the data is acquired with slicing and electron microscopy. This should be considered when choosing the kernel size , kernel stride and span of the depth dimension. An example is the ISBI 2012 dataset which spans 2 x 2 x 1.5 microns with a resolution of 4 by 4 by 50 nm/pixel [14], [11], [15].

8.5.5 Improving Test Metrics

Visual inspection and the error metrics used in this report can only give information about how accurate the label predictions are compared to the ground truth. For connectomics, this may not be the most important objective. The tools which will further process the segmentations may be able to correct certain errors in the predictions by testing how likely the final result is, while other merge and split errors lead to uncorrectable errors. This is an objective that could be more useful when selecting the network architecture, loss function and training method.

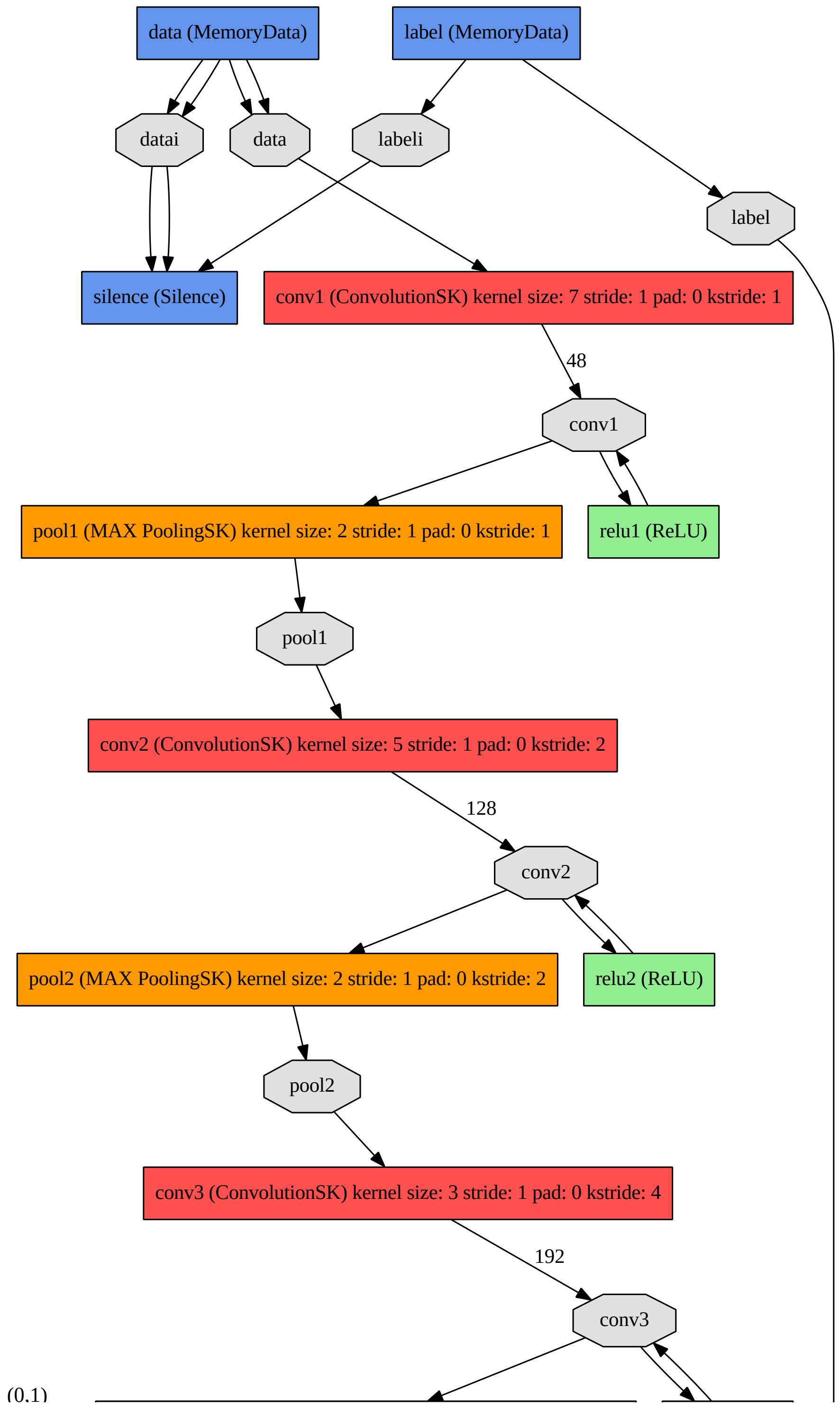
8.6 Final Words

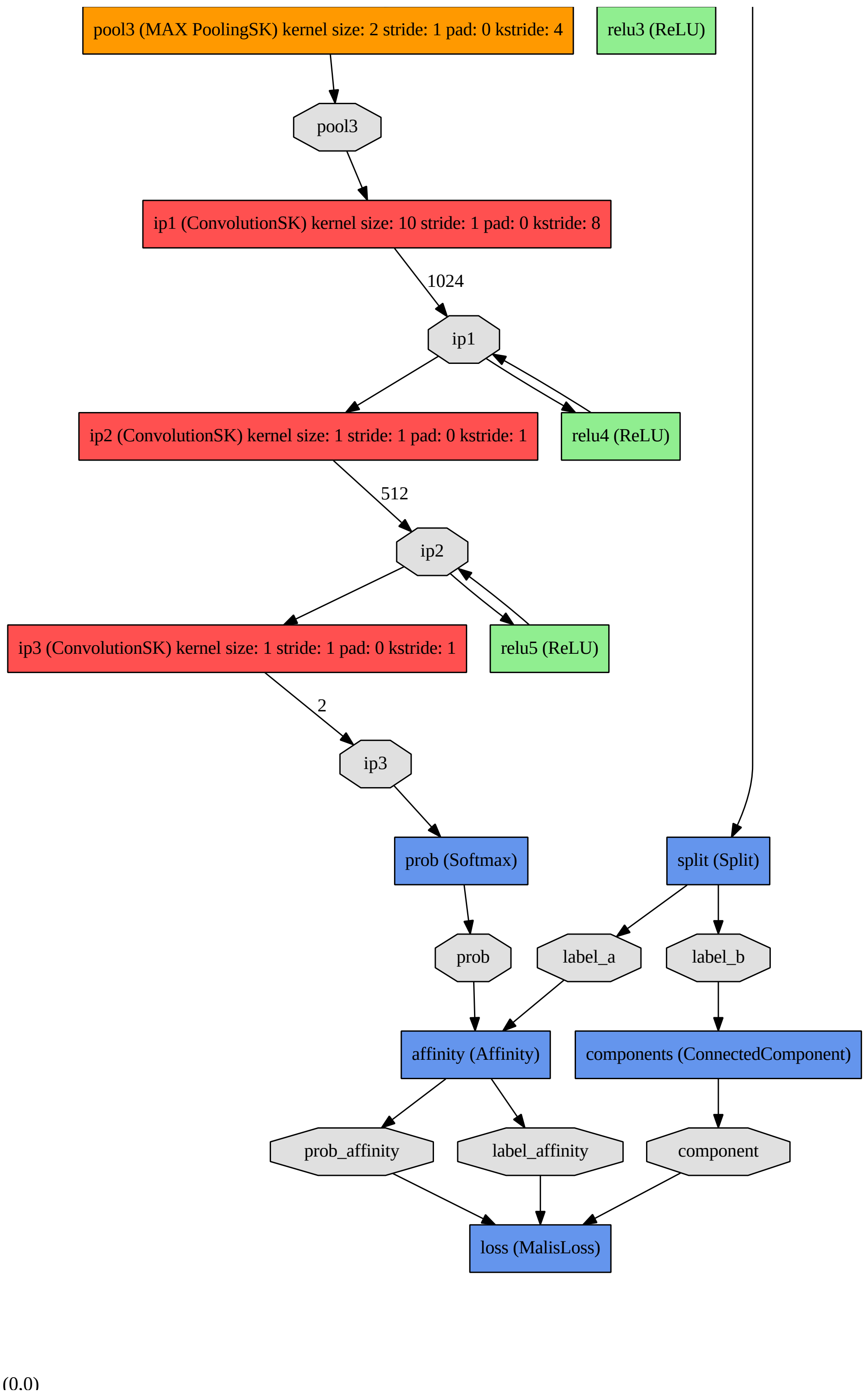
This research project combines many disciplines, such as high performance computing, machine learning, visual computing and a bit of connectomics. I was able to implement most of the originally planned features and found replacements for ideas that turned out to work badly. Finally, the results of the research include a useful, versatile stack of Open Source software (*Project Greentea*) that can be extended in the future. During working on the project, it was already possible to establish a growing user base [29]. The models and tools introduced with this project can be used efficiently on a large variety of data sets and hardware, making it very flexible. The collaboration at HHMI Janelia was also a great experience. Hardware sponsoring by AMD shows that programming of the *Project Greentea* and efficient machine learning libraries in general is of high interest also for hardware manufacturers.

Appendix A

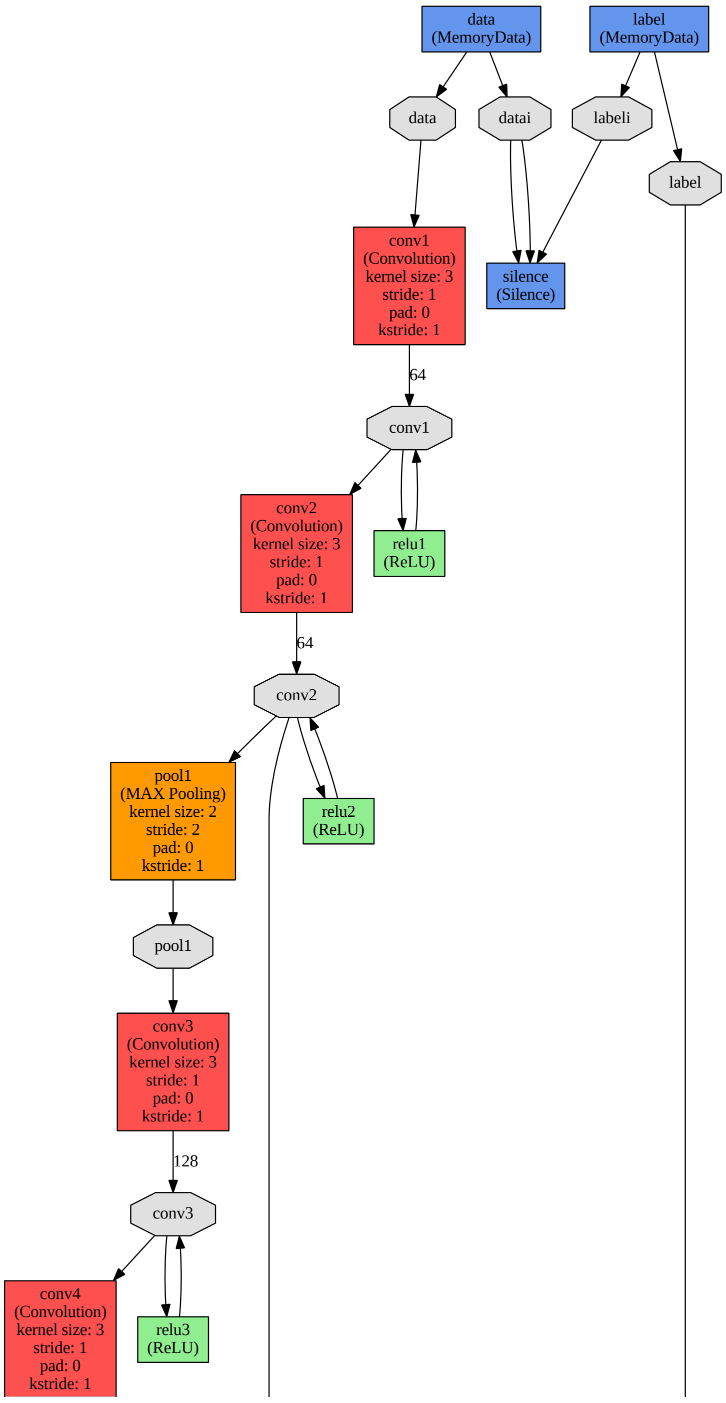
Network Architectures

A.1 SK-Net

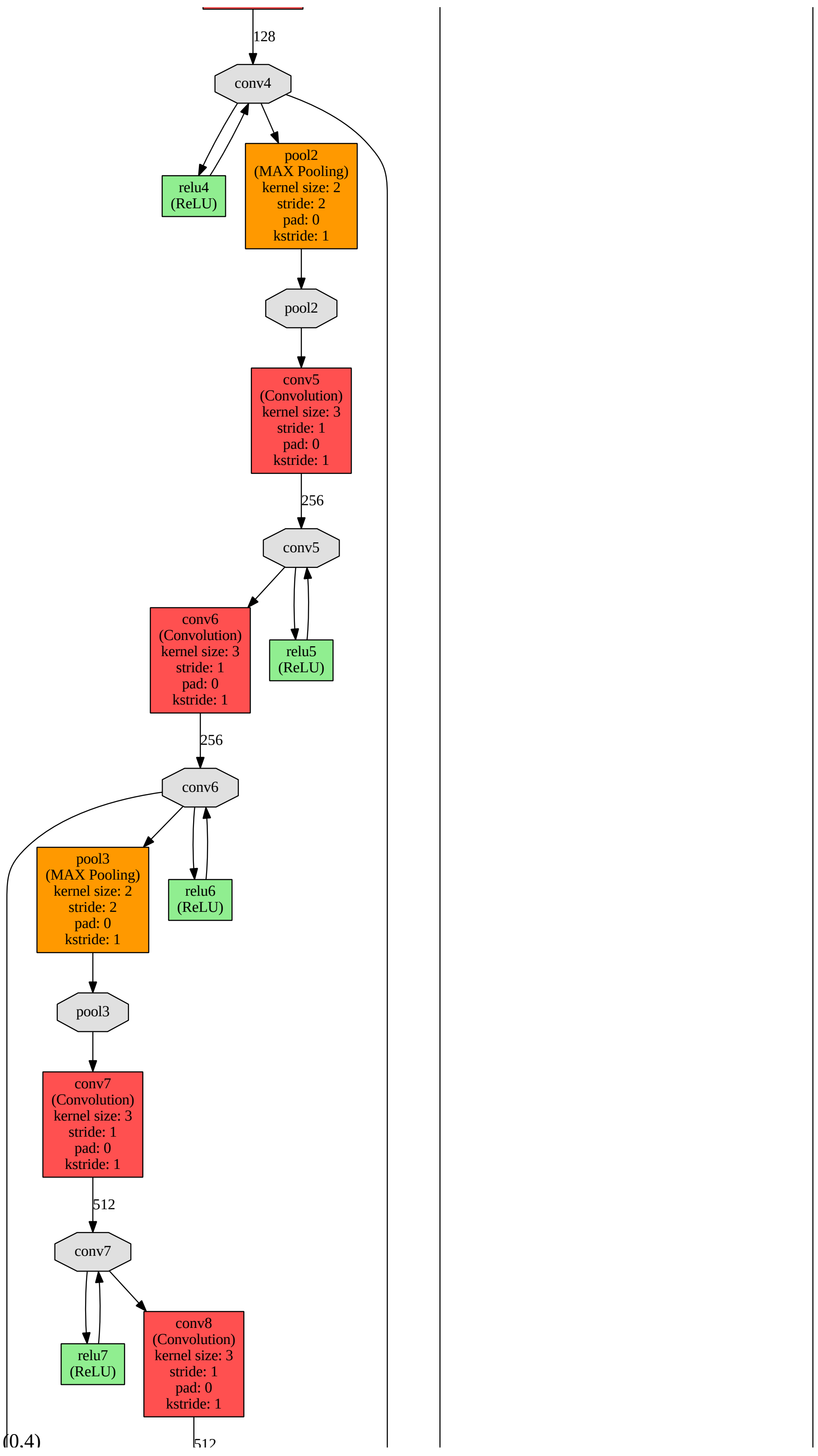

data (MemoryData)
label (MemoryData)
datai
data
labeli
label
silence (Silence)
conv1 (ConvolutionSK) kernel size: 7 stride: 1 pad: 0 kstride: 1
48
conv1
pool1 (MAX PoolingSK) kernel size: 2 stride: 1 pad: 0 kstride: 1
relu1 (ReLU)
pool1
conv2 (ConvolutionSK) kernel size: 5 stride: 1 pad: 0 kstride: 2
128
conv2
pool2 (MAX PoolingSK) kernel size: 2 stride: 1 pad: 0 kstride: 2
relu2 (ReLU)
pool2
conv3 (ConvolutionSK) kernel size: 3 stride: 1 pad: 0 kstride: 4
192
conv3
(0,1)

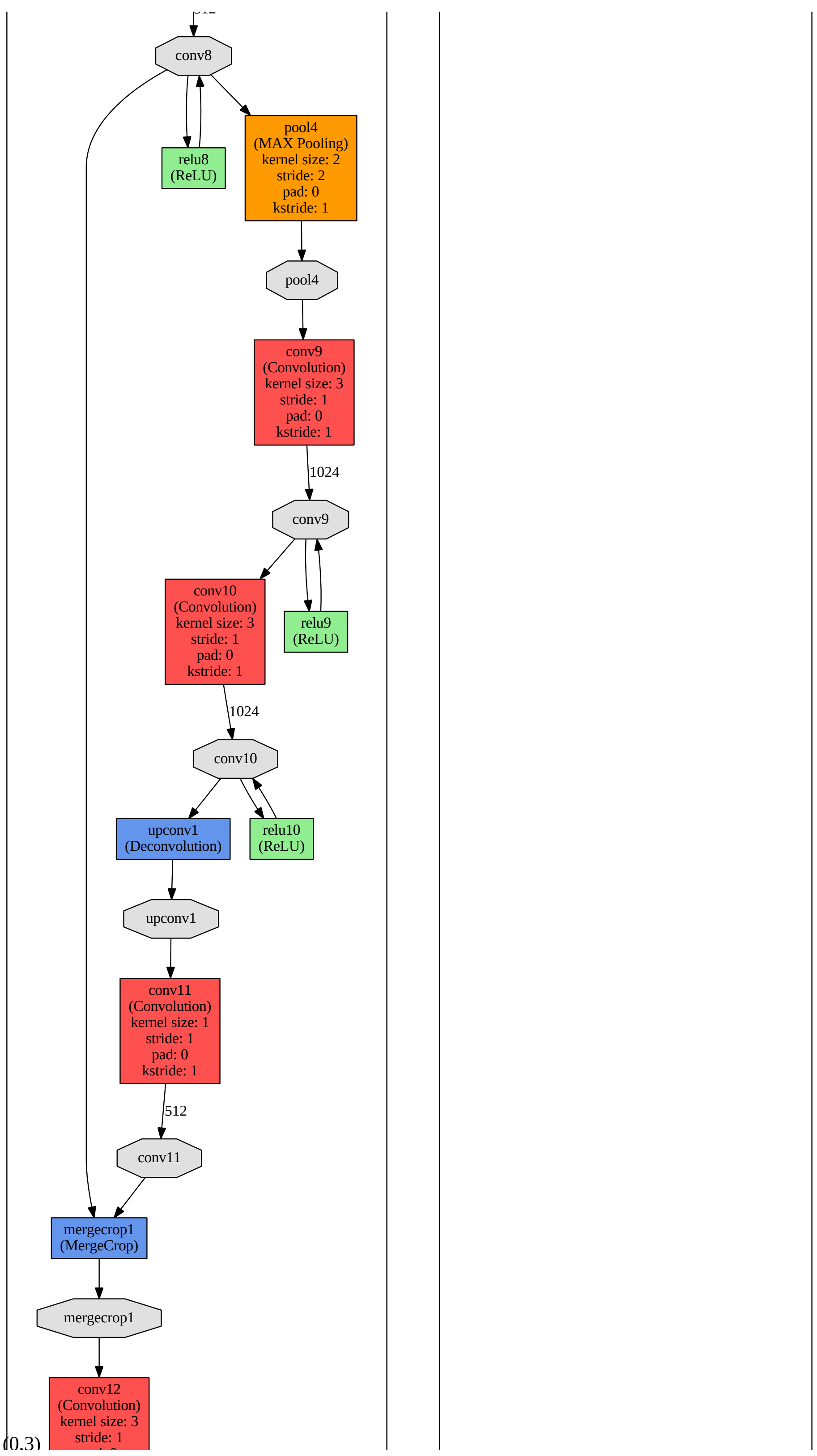

pool3 (MAX PoolingSK) kernel size: 2 stride: 1 pad: 0 kstride: 4
relu3 (ReLU)
pool3
ip1 (ConvolutionSK) kernel size: 10 stride: 1 pad: 0 kstride: 8
1024
ip1
ip2 (ConvolutionSK) kernel size: 1 stride: 1 pad: 0 kstride: 1
relu4 (ReLU)
512
ip2
ip3 (ConvolutionSK) kernel size: 1 stride: 1 pad: 0 kstride: 1
relu5 (ReLU)
2
ip3
prob (Softmax)
split (Split)
prob
label_a
label_b
affinity (Affinity)
components (ConnectedComponent)
prob_affinity
label_affinity
component
loss (MalisLoss)
(0,0)

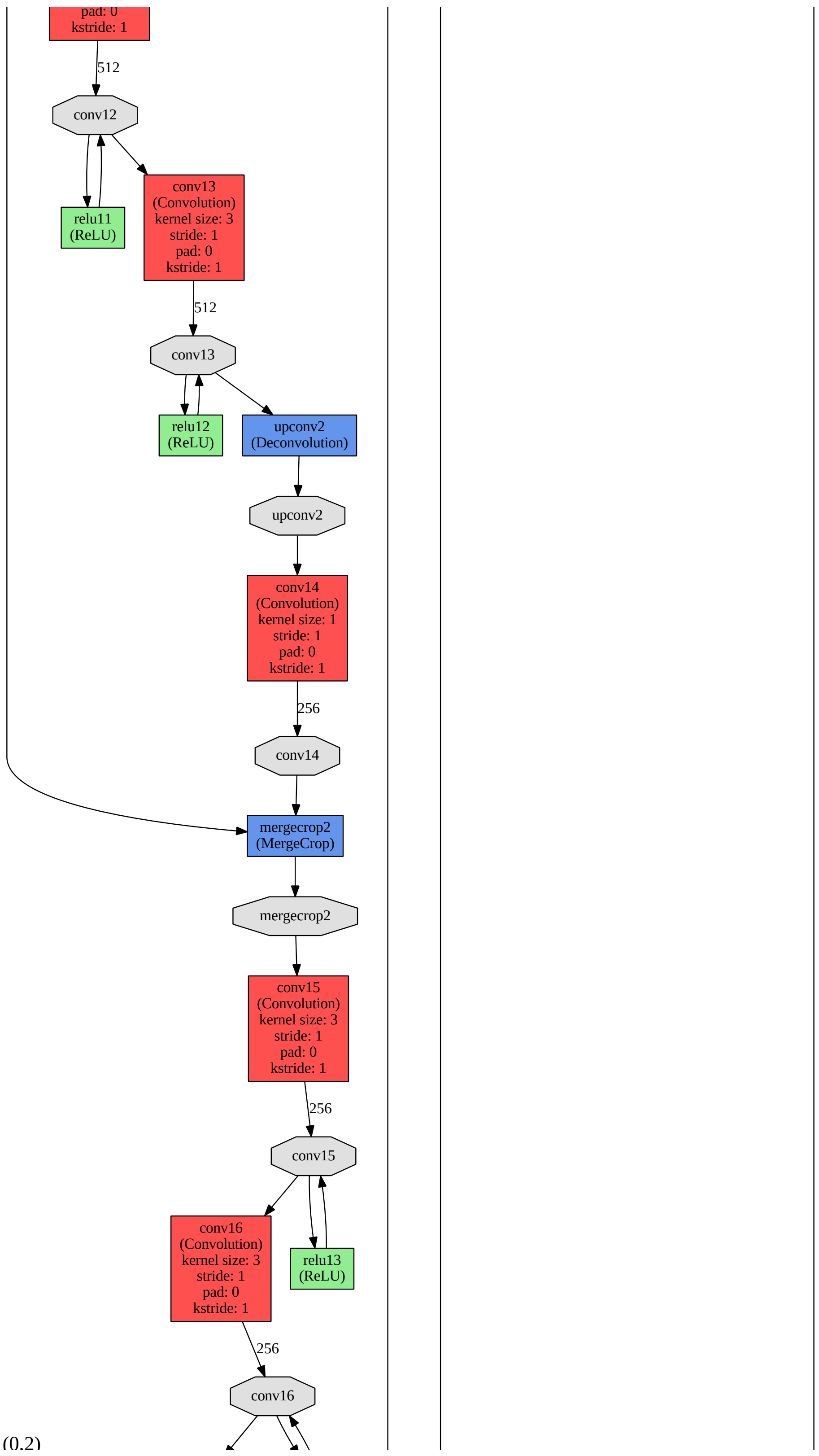
A.2 U-Net

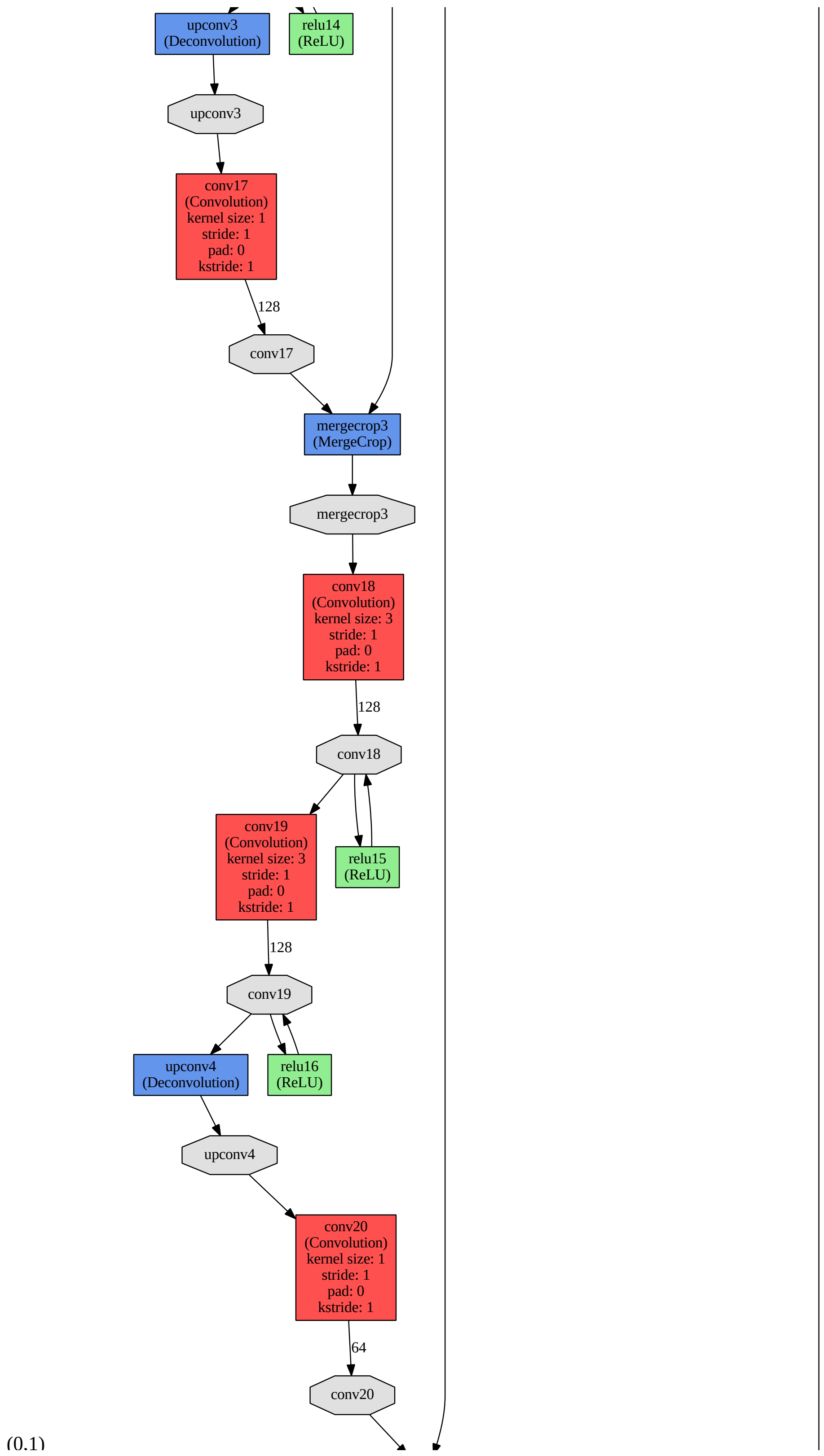

data
(MemoryData)
label
(MemoryData)
data
datai
labeli
label
conv1
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
silence
(Silence)
64
conv1
conv2
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
relu1
(ReLU)
64
conv2
pool1
(MAX Pooling)
kernel size: 2
stride: 2
pad: 0
kstride: 1
relu2
(ReLU)
pool1
conv3
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
128
conv3
conv4
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
relu3
(ReLU)

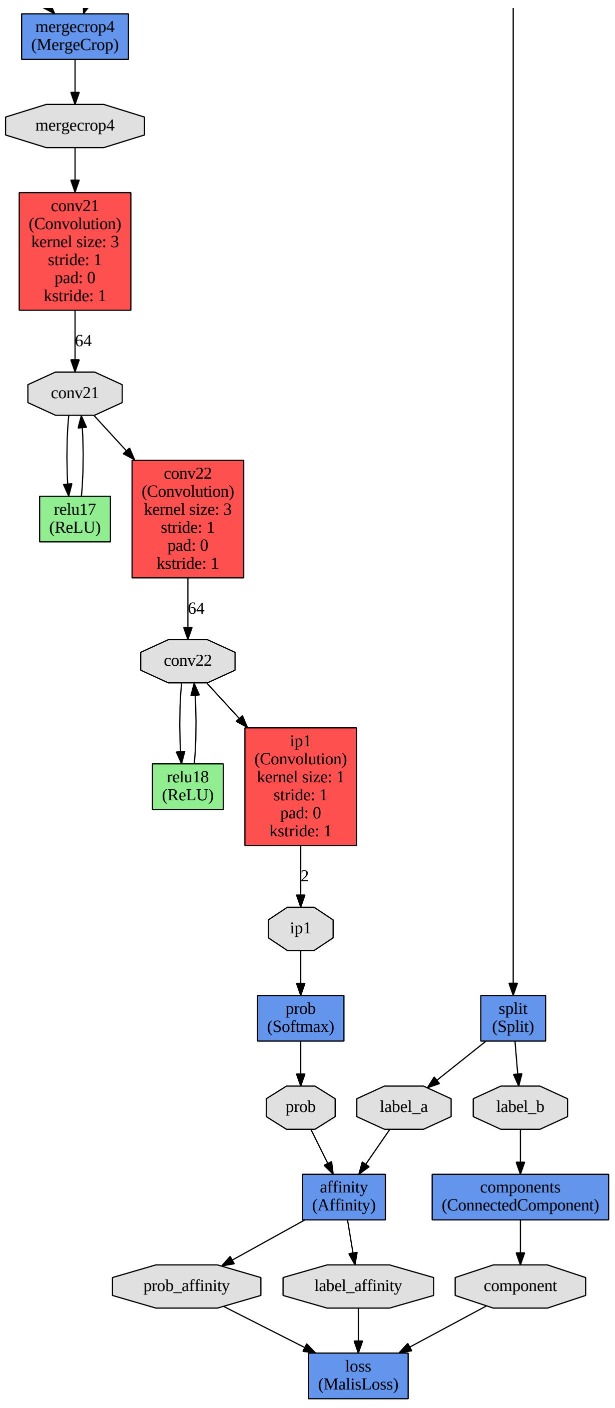
(0.5)


128
conv4
relu4
(ReLU)
pool2
(MAX Pooling)
kernel size: 2
stride: 2
pad: 0
kstride: 1
pool2
conv5
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
256
conv5
conv6
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
relu5
(ReLU)
256
conv6
pool3
(MAX Pooling)
kernel size: 2
stride: 2
pad: 0
kstride: 1
relu6
(ReLU)
pool3
conv7
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
512
conv7
relu7
(ReLU)
conv8
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
(0.4)
512


conv8
relu8
(ReLU)
pool4
(MAX Pooling)
kernel size: 2
stride: 2
pad: 0
kstride: 1
pool4
conv9
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
1024
conv9
conv10
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
relu9
(ReLU)
1024
conv10
upconv1
(Deconvolution)
relu10
(ReLU)
upconv1
conv11
(Convolution)
kernel size: 1
stride: 1
pad: 0
kstride: 1
512
conv11
mergecrop1
(MergeCrop)
mergecrop1
conv12
(Convolution)
kernel size: 3
stride: 1
(0.3)

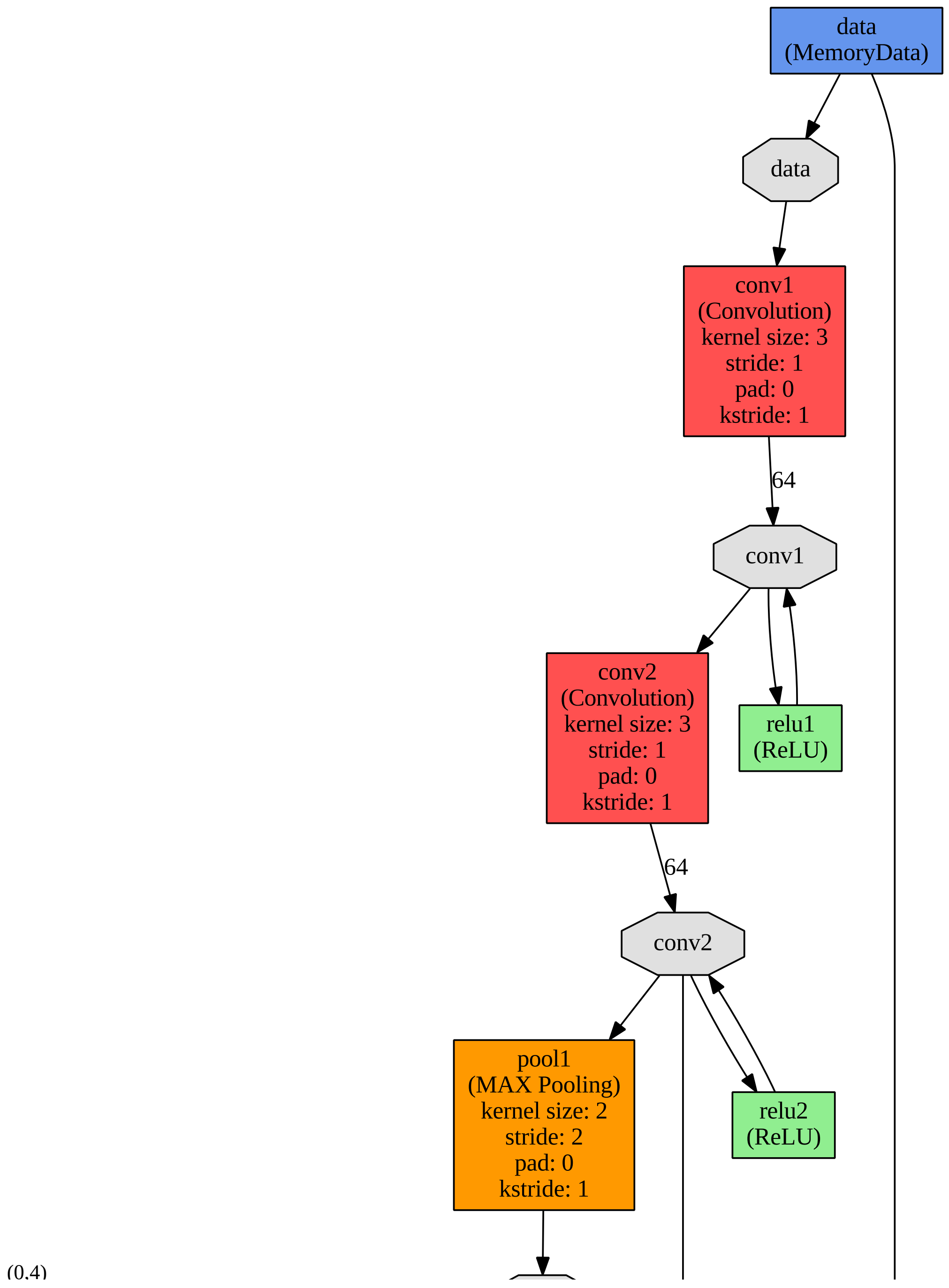

pad: 0
kstride: 1
512
conv12
relu11
(ReLU)
conv13
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
512
conv13
relu12
(ReLU)
upconv2
(Deconvolution)
upconv2
conv14
(Convolution)
kernel size: 1
stride: 1
pad: 0
kstride: 1
256
conv14
mergecrop2
(MergeCrop)
mergecrop2
conv15
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
256
conv15
conv16
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
relu13
(ReLU)
256
conv16
(0.2)

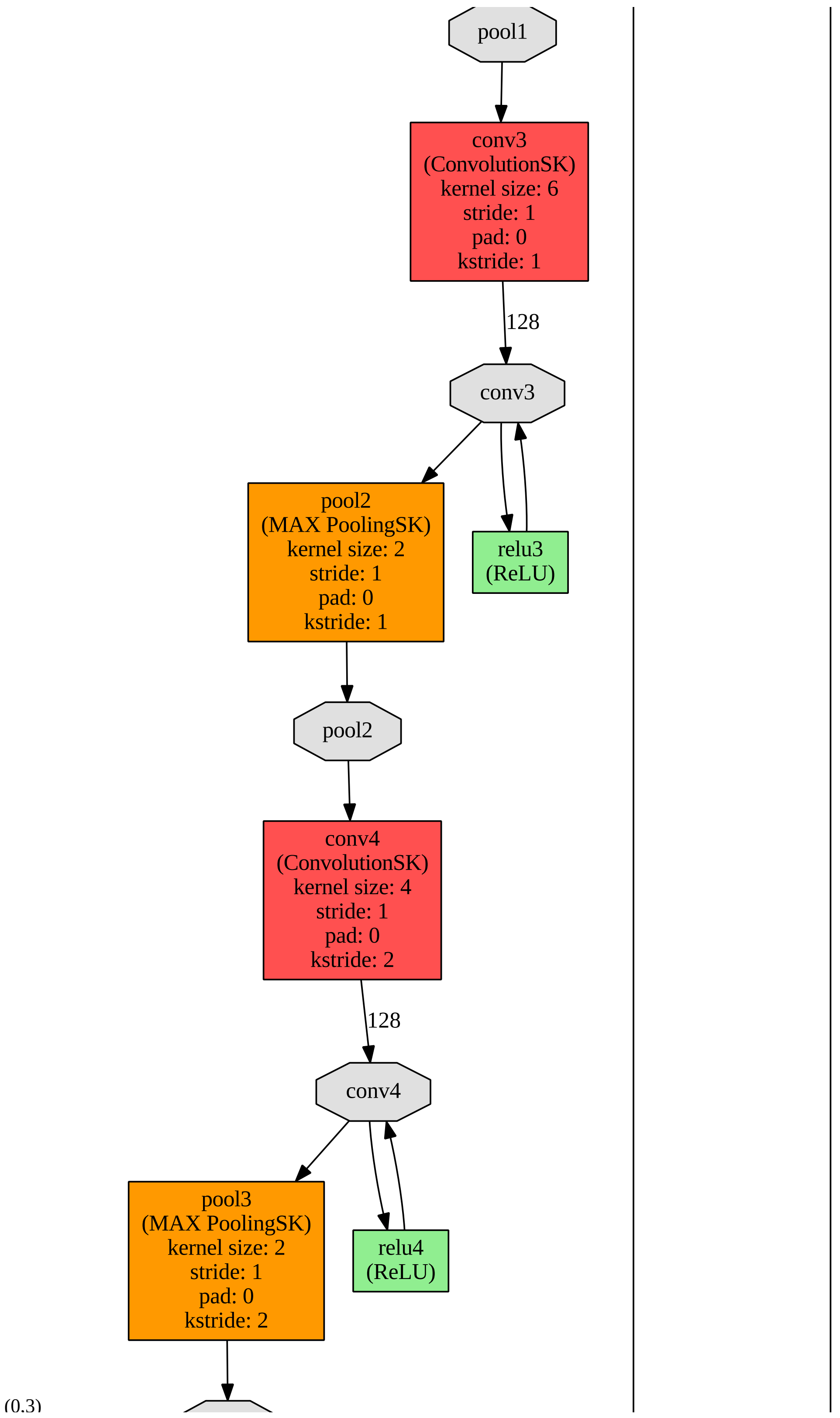

upconv3
(Deconvolution)
relu14
(ReLU)
upconv3
conv17
(Convolution)
kernel size: 1
stride: 1
pad: 0
kstride: 1
128
conv17
mergecrop3
(MergeCrop)
mergecrop3
conv18
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
128
conv18
conv19
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
relu15
(ReLU)
128
conv19
upconv4
(Deconvolution)
relu16
(ReLU)
upconv4
conv20
(Convolution)
kernel size: 1
stride: 1
pad: 0
kstride: 1
64
conv20
(0,1)

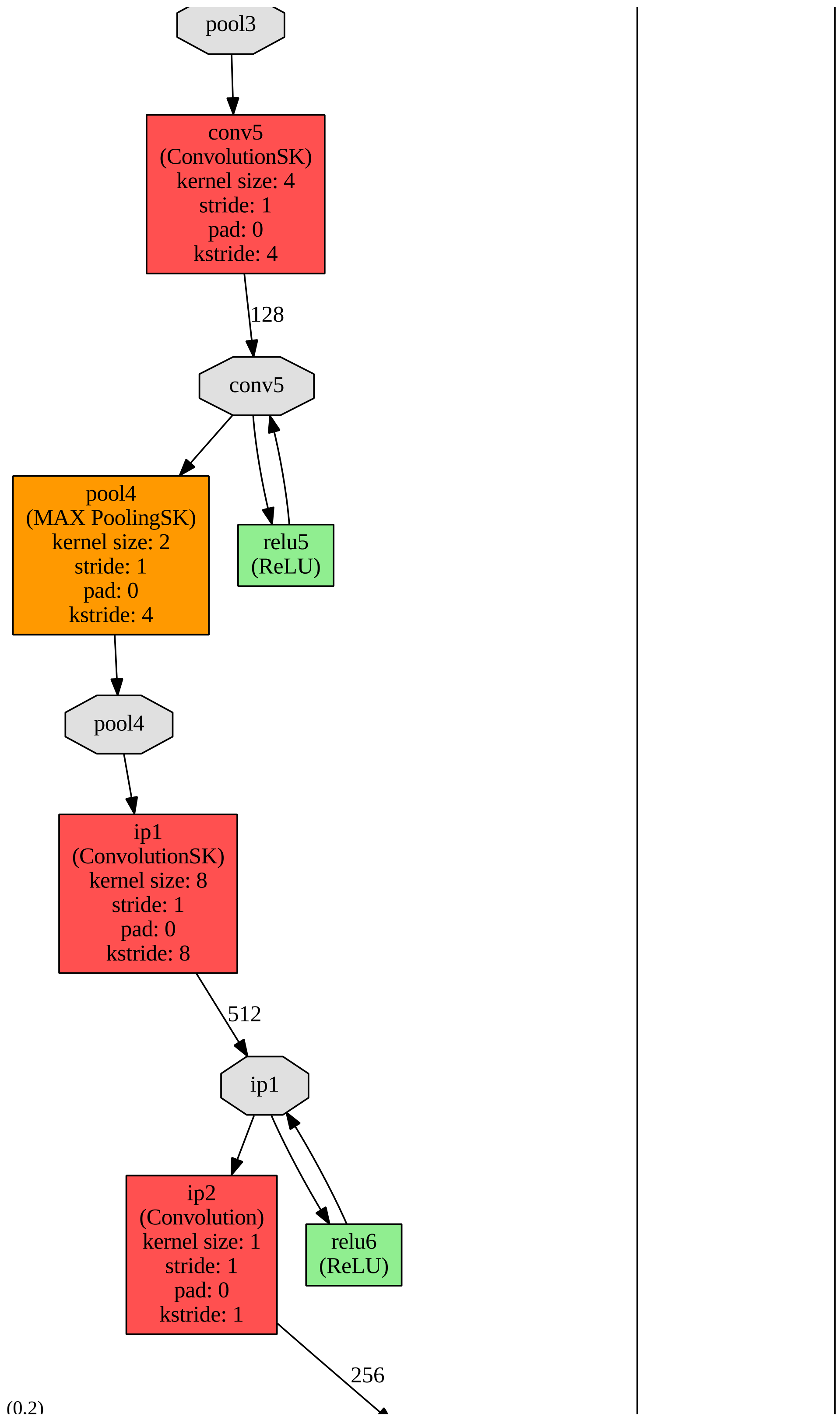

mergecrop4
(MergeCrop)
mergecrop4
conv21
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
64
conv21
relu17
(ReLU)
conv22
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
64
conv22
relu18
(ReLU)
ip1
(Convolution)
kernel size: 1
stride: 1
pad: 0
kstride: 1
2
ip1
prob
(Softmax)
split
(Split)
prob
label_a
label_b
affinity
(Affinity)
components
(ConnectedComponent)
prob_affinity
label_affinity
component
loss
(MalisLoss)

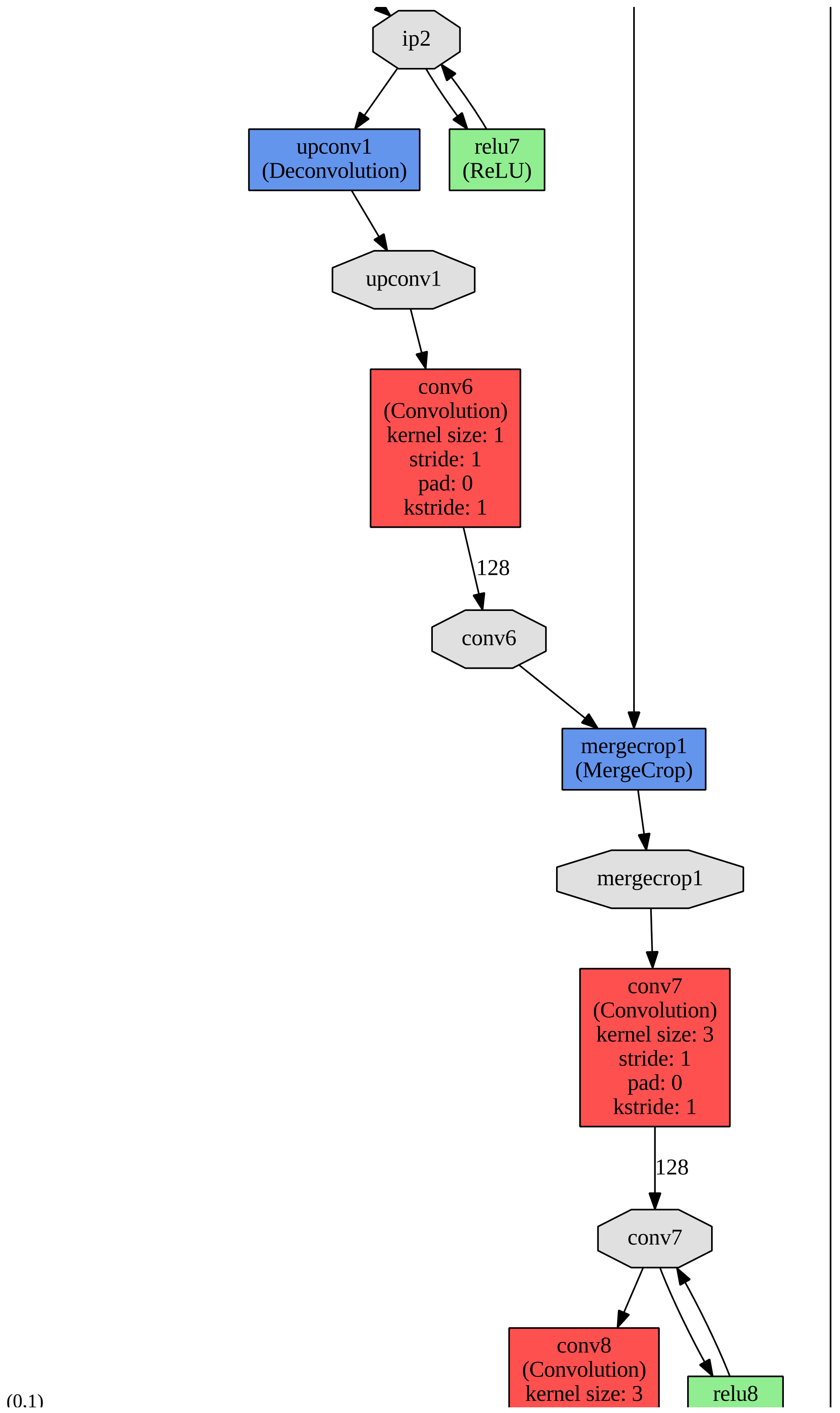
(0.0)

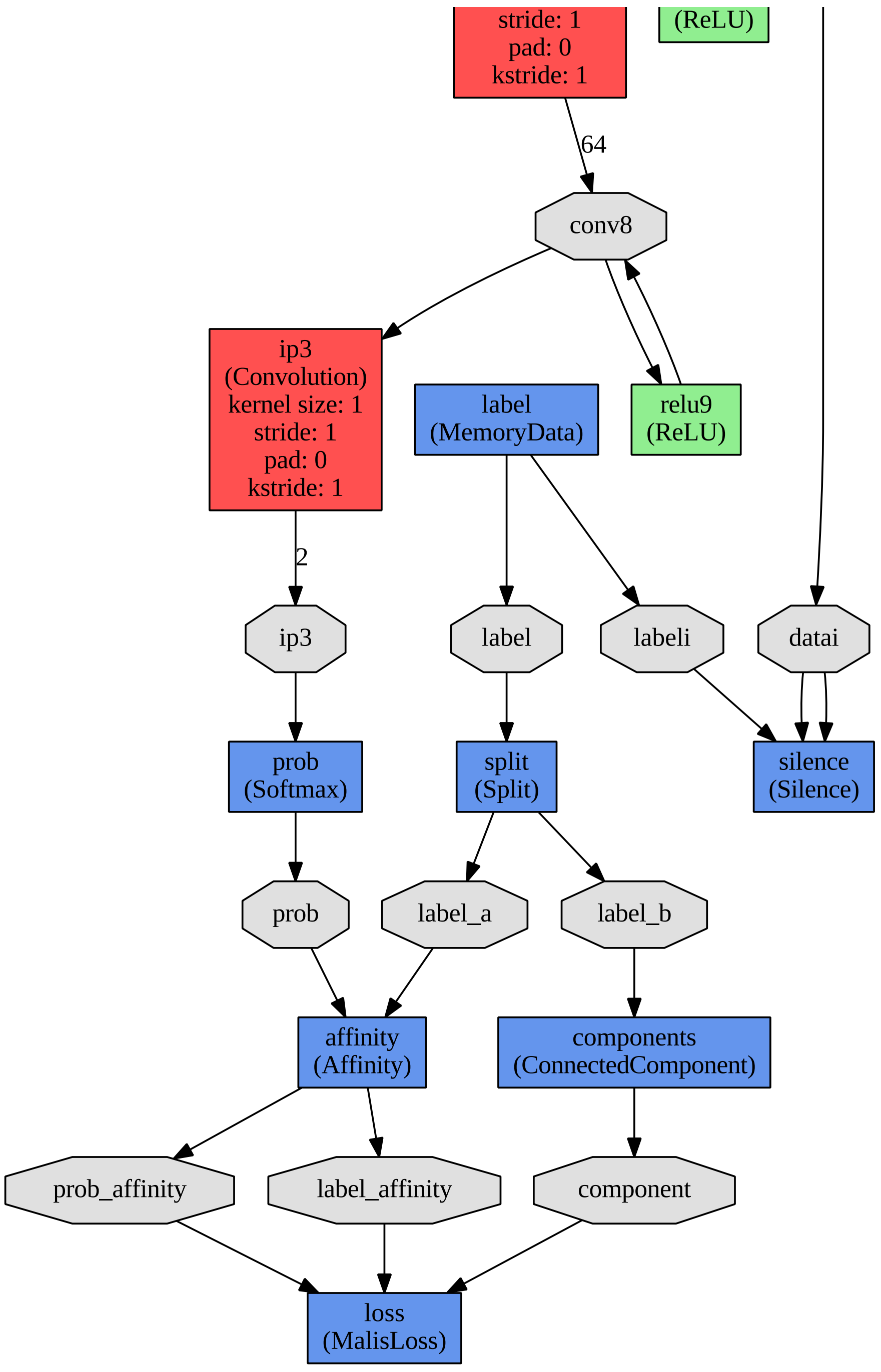
A.3 USK-Net


data
(MemoryData)
data
conv1
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
64
conv1
conv2
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
relu1
(ReLU)
64
conv2
pool1
(MAX Pooling)
kernel size: 2
stride: 2
pad: 0
kstride: 1
relu2
(ReLU)
(0,4)


pool1
conv3
(ConvolutionSK)
kernel size: 6
stride: 1
pad: 0
kstride: 1
128
conv3
pool2
(MAX PoolingSK)
kernel size: 2
stride: 1
pad: 0
kstride: 1
relu3
(ReLU)
pool2
conv4
(ConvolutionSK)
kernel size: 4
stride: 1
pad: 0
kstride: 2
128
conv4
pool3
(MAX PoolingSK)
kernel size: 2
stride: 1
pad: 0
kstride: 2
relu4
(ReLU)
(0.3)


pool3
conv5
(ConvolutionSK)
kernel size: 4
stride: 1
pad: 0
kstride: 4
128
conv5
pool4
(MAX PoolingSK)
kernel size: 2
stride: 1
pad: 0
kstride: 4
relu5
(ReLU)
pool4
ip1
(ConvolutionSK)
kernel size: 8
stride: 1
pad: 0
kstride: 8
512
ip1
ip2
(Convolution)
kernel size: 1
stride: 1
pad: 0
kstride: 1
relu6
(ReLU)
256
(0.2)


ip2
upconv1
(Deconvolution)
relu7
(ReLU)
upconv1
conv6
(Convolution)
kernel size: 1
stride: 1
pad: 0
kstride: 1
128
conv6
mergecrop1
(MergeCrop)
mergecrop1
conv7
(Convolution)
kernel size: 3
stride: 1
pad: 0
kstride: 1
128
conv7
conv8
(Convolution)
kernel size: 3
relu8
(0,1)


stride: 1
pad: 0
kstride: 1
(ReLU)
64
conv8
ip3
(Convolution)
kernel size: 1
stride: 1
pad: 0
kstride: 1
label
(MemoryData)
relu9
(ReLU)
2
ip3
label
labeli
datai
prob
(Softmax)
split
(Split)
silence
(Silence)
prob
label_a
label_b
affinity
(Affinity)
components
(ConnectedComponent)
prob_affinity
label_affinity
component
loss
(MalisLoss)

(0,0)

References

[1] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[2] O. Ronneberger, P. Fischer and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *ArXiv e-prints* (May 2015). arXiv:1505.04597 [cs.CV].

[3] S. C. Turaga et al. "Maximin affinity learning of image segmentation". In: *ArXiv e-prints* (Nov. 2009). arXiv:0911.5372 [cs.CV].

[4] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[5] Fabian Tschopp. *Caffe Improved*. URL: https://github.com/naibaf7/caffe (visited on 20th Aug. 2015).

[6] *BVLC Caffe*. URL: https://github.com/BVLC/caffe (visited on 20th Aug. 2015).

[7] H. Li, R. Zhao and X. Wang. "Highly Efficient Forward and Backward Propagation of Convolutional Neural Networks for Pixelwise Classification". In: *ArXiv e-prints* (Dec. 2014). arXiv:1412.4526 [cs.CV].

[8] Stephan Gerhard et al. *Segmented anisotropic ssTEM dataset of neural tissue*. 2013. URL: http://dx.doi.org/10.6084/m9.figshare.856713 (visited on 20th Aug. 2015).

[9] Fabian Tschopp and Julien Martel. *Caffe Neural Models*. URL: https://github.com/naibaf7/caffe_neural_models (visited on 20th Aug. 2015).

[10] Julien Martel. *Sliding Window Network*. URL: https://www.ini.uzh.ch/people/jmartel (visited on 20th Aug. 2015).

[11] *ISBI 2012 Challenge*. URL: http://brainiac2.mit.edu/isbi_challenge/ (visited on 20th Aug. 2015).

[12] Jeff Donahue. *Caffe ND convolutions*. URL: https://github.com/BVLC/caffe/pull/2049 (visited on 20th Aug. 2015).

[13] *Segmentation Evaluation Metrics*. URL: http://fiji.sc/Segmentation_evaluation_metrics_-_Script (visited on 20th Aug. 2015).

[14] *TrackEM*. 2012. URL: https://www.ini.uzh.ch/~acardona/trakem2.html (visited on 20th Aug. 2015).

[15] Albert Cardona et al. "An Integrated Micro- and Macroarchitectural Analysis of the Drosophila Brain by Computer-Assisted Serial Section Electron Microscopy". In: *PLoS Biology* 8.10 (Oct. 2010). Ed. by Kristen M. Harris, e1000502. DOI: 10.1371/journal.pbio.1000502. URL: http://dx.doi.org/10.1371/journal.pbio.1000502.

[16] Fabian Tschopp. *Caffe Neural Tool*. URL: https://github.com/naibaf7/caffe_neural_tool (visited on 20th Aug. 2015).

[17] Srinivas Turaga. *Malis criterion for Matlab*. URL: https://github.com/srinituraga/malis/ (visited on 20th Aug. 2015).

[18] Srinivas Turaga. *Malis criterion for Torch*. URL: https://github.com/srinituraga/lua---imgraph/ (visited on 20th Aug. 2015).

[19] K. Rupp, F. Rudolf and J. Weinbub. "ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs". In: *Intl. Workshop on GPUs and Scientific Applications*. 2010, pp. 51–56.

[20] S. Chetlur et al. "cuDNN: Efficient Primitives for Deep Learning". In: *ArXiv e-prints* (Oct. 2014). arXiv:1410.0759.

[21] Nicolas Vasilache et al. *Fast Convolutional Nets With fbfft: A GPU Performance Evaluation*. 2014. eprint: arXiv:1412.7580.

[22] *AMD W9100 Whitepaper*. URL: http://www.amd.com/Documents/FirePro_W9100_Data_Sheet.pdf (visited on 20th Aug. 2015).

[23] *nVidia GTX 980 Whitepaper*. URL: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF (visited on 20th Aug. 2015).

[24] *Intel ARK database*. URL: http://ark.intel.com/ (visited on 20th Aug. 2015).

[25] *AMD*. URL: http://www.amd.com/ (visited on 20th Aug. 2015).

[26] *OpenCL Device Fission*. URL: https://www.khronos.org/registry/cl/extensions/ext/cl_ext_device_fission.txt (visited on 20th Aug. 2015).

[27] Viren Jain et al. "Boundary Learning by Optimization with Topological Constraints". In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Institute of Electrical & Electronics Engineers (IEEE), June 2010. DOI: 10.1109/cvpr.2010.5539950. URL: http://dx.doi.org/10.1109/CVPR.2010.5539950.

[28] *HHMI Janelia*. URL: https://www.janelia.org/ (visited on 20th Aug. 2015).

[29] Fabian Tschopp. *Caffe Pull Request*. URL: https://github.com/BVLC/caffe/pull/2610 (visited on 20th Aug. 2015).