EXAMENSARBETE INOM ELEKTROTEKNIK,
AVANCERAD NIVÅ, 30 HP
STOCKHOLM, SVERIGE 2017

StemNet

A Temporally Trained Fully Convolutional Network for Segmentation of Muscular Stem Cells

MARTIN ISAKSSON

KTH
SKOLAN FÖR ELEKTRO- OCH SYSTEMTEKNIK

Abstract

In biomedical research, time-lapse microscopy is an important tool to be able to study processes which are too slow for humans to observe. This technique is powerful since it gives information about how parameters of single cells change over time.

The problem to be solved in this project is to segment MuSCs (Muscular Stem Cells) in images and to classify them. This is done by using a deep neural network trained using supervised learning. The network is inspired by the architecture of the U-net, but extended by using temporal data to see if it can increase its performance. The network is trained on images from the time-lapse sequence, where the temporal aspect is used to create a short-term memory for the network. The results are compared to a network of the same architecture but without the temporal aspect in the training.

The temporal approach shows that the network learns faster what is roughly a MuSC and what is not, but in the end it gives a slightly higher and more accurate classification of MuSCs by training the network without giving it a short-term memory, for this task.

Sammanfattning

Tidsförloppsmikroskopi är ett viktigt verktyg inom biomedicinsk forskning, framförallt om man vill studera processer som är för långsamma för en människa att observera. Det är en kraftfull teknik eftersom den ger information om hur parametrar för enskilda celler ändras över tiden.

Det problem som ska lösas i det här projektet är att segmentera muskelstamceller i bilder och klassificera dem. Tillvägagångssättet är att använda djupa neurala nätverk, tränade genom kontrollerad inlärning. Arkitekturen på nätverket är inspirerat utav U-net, men med tillägget att det använder temporal data för att se ifall den kan prestera bättre. Nätverket är tränat på bilder tagna från sekvenser av tidsförloppsmikroskopi, där den temporala aspekten i sekvenserna används för att skapa ett korttidsminne till nätverket. Resultatet utav nätverket jämförs med resultatet av ett nästan likadant nätverk, skillnaden är att det inte använder temporal data när det tränas.

Nätverket som tränas med temporal data visar sig snabbare lära sig vad som grovt är en muskelstamcell i en bild och vad som inte är det. Men i slutändan visar det sig att nätverket som tränas utan temporal data presterar lite bättre och har lite högre precision än nätverket med ett korttidsminne.

Table of Contents

1 Introduction

In biomedical research, time-lapse (recorded with a lower frame rate than in a normal video) microscopy (transmission microscopy) is an important tool to be able to study processes which are too slow for humans to observe. This technique is powerful since it gives information about how parameters of single cells changes over time. Different types of stem cells are often analyzed by this method, e.g. MuSCs (Muscle Stem Cells), HSC (Hematopoietic Stem Cells), HeLa (Henrietta Lacks, a patient who died in cancer in 1951 [1]) cells.

To find an object in an image is known as *segmentation*. By segmenting stem cells, i.e. retrieving the area of the cells, and also tracking each cell and finding the patterns of its path and splits (also known as cell tracking), one can draw medical conclusions. This can be done by using algorithms on these segmented areas and tracking paths to extract biologically interesting information such as lineage trees, cell sizes and migration speeds [2]. But up to this date there are not any bulletproof algorithms that can analyze time-lapse sequences without errors or that they are very time consuming (e.g. demands manual work).

To get a good tracking result, a good segmentation result is important. Today the state-of-the-art in image segmentation is deep learning, implemented by fully convolutional networks [3]. Deep learning can be used for many different purposes. It can e.g. be used to classify objects in an image, to classify a speaker, generate words and music, generate images, and much more [4], [5].

The first thing to do after deciding to use deep learning to solve a problem, is to decide if a discriminative or a generative model should be used, and if supervised or unsupervised learning should be used. The problem to be solved in this project is to segment MuSCs and to classify them. This is done by using a discriminative model trained using supervised learning. The network/model is inspired by the architecture of the U-net [3], but extended by using temporal data to potentially increase its performance. The network will consist of convolutional and deconvolutional layers, also known as a fully convolutional network. The input image, containing stem cells, will be downsampled to compressed features and then these features will be upsampled where the output has the same size as the input image, but with the stem cells segmented and classified. This is possible because the downsampling, and later upsampling steps, allow the segmentation to gain contextual knowledge from a wider spatial area when classifying single pixels as belonging to cells or background.

A cell tracking-system consists of several components, not just segmentation. The components are:

- Image preprocessing
- Segmentation
- Track-linking
- Post-processing of tracks

This project will focus on the two first components, *image preprocessing* and *segmentation* by building and train a fully convolutional network to segment MuSCs in images.

2 Related Work

2.1 U-Net Biomedical Image Segmentation

In a similar task of segmenting different cells in biomedical images, the first fully convolutional network of its kind was built by [3] and it is called the U-net. It was the EM segmentation challenge at ISBI 2012 [6] that motivated Ronneberger et al. to build this state-of-the-art network. The U-net performed better than any other competitor in terms of segmentation.

The EM segmentation challenge at ISBI 2012 only provide 30 training images, which is way too few to train a deep convolutional network. What was done to handle this issue was to use data augmentation by rotating and deforming the 30 images so that a larger training dataset could be used. Even though the use of data augmentation, it was still kind of few images to train a deep network for this task. Therefore, Ronneberger et al. chose a network architecture to make sure that this wasn´t a problem.

To segment images with help of deep learning, the best way is to use fully convolutional networks, which contain an “ordinary convolutional network”, i.e. a downsampling part, but also a deconvolutional part, i.e. an upsampling part. By using this symmetrical structure, and also sending feature information from the downsampling part to the respective (in the manner of size) upsampling part, few data/images was enough to train the network. By passing over information from the downsampling to the upsampling helps the network to learn the spatial and more detailed information better. The symmetrical network can be seen as an “U”, hence the name U-net.

In biomedical images of cells, it can sometimes be tricky to see the difference between if it is one cell or if there are several cells with touching edges. To make sure that the network learns to see this difference, Ronneberger et al. added a weight to their cost function to force the network to learn these pixels between close cells. Also a weight was added to handle the class imbalance.

The U-net showed great results compared to their competitors on several different cell images, not just one kind of cell. The network was implemented in the API Caffe [7] and only took 10 hours to train on a NVidia Titan GPU (6 GB).

The training images were of size 512x512 and the patch size in the convolutions was a receptive field of 3x3. The U-net doesn´t use padding in the convolutions, therefore the information sent from the downsampling part to the upsampling part has to be cropped to be able to be concatenated. This leads to the final output image to be of smaller size than the input image. The network is of 23 convolutional and deconvolutional layers, where there are two convolutional layer in a row before a pooling or upsampling (deconvolution) is performed. The activation function used

through the network is ReLU and the output of the network is a 1x1 convolution to map each feature vector to the desired number of classes. The cost function used is cross entropy (with a weight as mentioned) after performing a softmax on the output. Also, when a network is very deep it is important to initialize the weights in the network well, which has been done in the U-net. The weights are initialized by drawing them from a Gaussian distribution with a standard deviation of $\sqrt{\frac{2}{N}}$, where N is the number of incoming nodes of one neuron (i.e. N is the patch size times the number of feature maps in the previous layer).

2.2 U-Net RFI Image Segmentation

In the paper [8] they have also used a U-net for the classification issue of finding radio frequency interference (RFI) signals in radio data. Their goal is to mitigate the RFI signals affecting the radio data.

Akeret et al. have trained the network on both simulated data and real data from a telescope. The problem with the real data is that there isn´t possible to get the ground truth, i.e. the segmentation mask of the input, which result in biased results since they create the “ground truth” with another algorithm doing the same thing as the network is supposed to achieve.

The network is implemented with the API TensorFlow [9]. The network only takes a few hours to train on a NVidia Kepler K20 GPU.

The network consists of 12 layers, where each convolutional layer has a receptive field and a patch size of 3x3, each pooling and upsampling is of 2x2, there is a dropout rate of 0.5 and they also use a L2 regularizer of strength 0.001. The network is trained for 100 epochs with mini batch-size of 32. The images used for training are of size 276x600. They use a momentum optimizer with a decaying learning rate, starting at 0.2. The cost function is cross entropy used on the output of a softmax.

The network shows promising results, but there is a lack of good training data, though it is said to be possible to access much more training data in the near future. One improvement to be made, that they mention, is to put some kind of penalizing to the cost function to make it easier for the network to learn which are contaminated RFI pixels and which are not.

2.3 Training Deconvolution Network for Semantic Segmentation

Fully convolutional networks have become very popular to use for the task of semantic segmentation. Noh et al. in [10] were one of the first to use these kinds of network for the task of semantic segmentation. The pre-trained VGG16 network [11] is used as their downsampling part and then they train and learn an upsampling part. To perform upsampling, first (what is here called) unpooling is performed and then deconvolution (as it is called here). This can be discussed since there exist a lot of different definitions for “unpooling” and “deconvolution” out there. The unpooling is done by passing over information from the corresponding pooling layer in the downsampling (therefore, also this network could be called U-net). The information is called switch variables, but basically it is indices of the maximum value in each 2x2 area that they use in their max-pooling. This creates a

sparse tensor (many feature maps). Each feature map is deconvolved which makes each feature map dense, by using learned filters.

The network contains in total 28 layers, 13 layers in the downsampling part, 13 layers in the upsampling part, and 2 fully-connected layers as a bridge between the downsampling and the upsampling. The pooling performed is max-pooling and the activation function used is ReLU. A softmax is performed on the output of the network to get a pixel-wise prediction.

The dataset used is PASCAL VOC 2012 dataset. It contains 12031 training and validation images in total, each image of size 224x224. Since it is a limited number of training images to learn such a deep network, a *batch normalization* is performed to reduce the internal covariate shift (to escape local minima in optimization) and *two-stage training* which means that first the network is trained with easy examples, and then fine-tune the network by training it on more challenging examples (i.e. data augmentation is used, where the easy images are centred and cropped).

The network is implemented with the API Caffe. A momentum is used as an optimizer with learning rate 0.01 and momentum as decay to 0.9. The weights in the upsampling part is initialized with zero-mean Gaussians. The patch size/receptive field in the convolutions is 3x3, except in the output where 1x1 is used for prediction of each pixel independently. The batch size is of 64 images. For convergation it takes 20000 iterations for training on the easier images and 40000 iterations for the more challenging images. Training takes 6 days on a single NVidia GTX Titan GPU (12G).

This network was state-of-the-art in the PASCAL VOC 2012 benchmark when the article was written, with an accuracy of 72.5%.

2.4 SegNet Image Segmentation

The SegNet [12] is a fully convolutional network built for the use of road scene understanding applications (e.g. autonomous driving). It is similar to the network mentioned above. The network is supposed to label each pixel with the class of the segmented object it is. To save time Badrinarayanan et al. have used the pre-trained weights and biases in the downsampling part from the VGG16 network [11]. This saves both time and memory (the fully-connected layers are skipped in the VGG16 network). In the upsampling part information is passed from the corresponding pooling layer (max-pooling) to the upsampling layer. Though compared to the U-net (even this could also be called a U-net), the information passed over is different. Here they use the indices of each and every 2x2 pooling, which only uses a small amount of memory compared to storing all the full feature maps. This will create a sparse tensor. These sparse feature maps (sparse tensor) are then convolved with a trainable decoder filter bank (where each upsample kernel is initialized using bilinear interpolation weights) to produce dense feature maps. Finally, a batch normalization is applied to each feature map.

The network consists of 26 layers, where 13 of them are pre-trained. The activation function used is ReLU and a softmax is performed on the output to get the probability of each pixel independently. The cost function used is cross entropy and the optimizer is momentum with a learning rate of 0.1 and a momentum of 0.9.

Everything is implemented with the API Caffe. For training there are 367 RGB images and for testing there are 233 RGB images, where each image is of size 360x480 (the dataset CamVid road scene). The batch size is of 12 images and train until the loss converges. Imbalanced classes are handled by weighting each class in the cost function by median frequency balancing. The weight for each class is then the median of the class frequencies of all the images in the training data divided by the class frequency. The network is trained on a NVidia Titan GPU.

The SegNet outperformed the other methods tried out for the same task, at the moment when the article was written.

The SegNet has also been trained on another dataset, containing indoor images. This was not the primary task of SegNet, rather to test/show that it performs well on this kind of data too.

Finally, there is a discussion of that Bayesian neural networks is something that should be experimented with in the future for this application.

3 Artificial Neural Networks

The human brain can be seen as a supercomputer, with its primary visual cortex (also known as V1) containing 140 million neurons and tens of billions of connections between them, and the rest of the visual cortices (V2, V3, V4 and V5) which are doing more complex image processing. It is no wonder that the human race has a habit of seeing patterns in most of things (even if there exist none) [13].

3.1 History

The development of artificial neural networks (ANNs) first started in 1943 when Warren McCulloch and Walter Pitts tried to figure out how the brain works. The first model was built as an electrical circuit of a simple neural network. In 1949 Donald Hebb discovered that the more time a neural path is used, the more it will be strengthened.

The first system ever built for commercial use was MADALINE (the non-commercial system which did the same thing was called ADALINE), a work done by Bernard Widrow and Marcian Hoff from Stanford in 1959. The system read streaming bits from a phone line, and from that it could predict the next bit. Despite the later success of neural nets, the traditional von Neumann architecture took over the computing scene.

In 1972 Kohonen and Anderson developed a neural network that used matrices. Without knowing it at the time, they were creating an array of analog ADALINE circuits. The difference was that it activated a set of outputs instead of just one. During the same decade, the first multilayer perceptron (MLP) network, which was an unsupervised network, was created. More specifically it was created in 1975. Thanks to the discovery of MLP and a paper that John Hopfield of Caltech published in 1982, the interest in the field was renewed.

In 1986, an attempt of extending the Widrow-Hoff rule to multiple layers was made. It was at that moment the, as we call it today, backprogagation method was born. However, back then it didn't exist powerful enough computers, which was needed for the computationally heavy backpropagation. There is a reason why it took all the way from then till 2011 to start using that theory at a large scale, because by then the computers were powerful enough (even though it still takes a long time to train networks with multiple layers today) [14].

3.2 Perceptrons

Artificial neural networks consist of multiple layers of neurons (also called units). There is an input layer, a hidden layer or several hidden layers (it doesn't necessarily need to exist a hidden layer), and an output layer. Normally a neural network is defined as a MLP if there are two or more hidden layers.

Perceptrons are one sort of neurons and they were developed in the 1950's and 1960's by a scientist called Frank Rosenblatt (inspired of the work McCulloch and Pitts, mentioned earlier). Very briefly, the perceptron takes several binary inputs and produces a single binary output (See *Figure 1*).

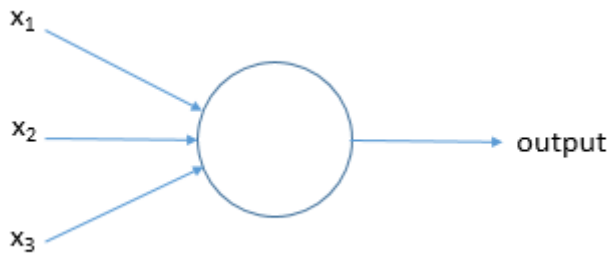

Figure 1 The perceptron.

Each binary input is multiplied with a *weight*, and if the sum of all these input-multiplications is greater than some *threshold value* the output will be 1, else it will be 0. A common approach is to move the threshold value to the same side as the weight and input, and call it the *bias*.

$$output = \begin{cases} 1 & ,if \ \sum_i w_i x_i + b > 0 \\ 0 & ,if \ \sum_i w_i x_i + b \leq 0 \end{cases} \quad (1)$$

3.3 ANNs of Today

Today it is more common with neurons that can take inputs others than binary values, and also give an output that is not binary.

Each neuron has several inputs, each multiplied by a weight. The summation of all these multiplications are added with the threshold value, also called the *bias* as mentioned above. The final value of this is used as an input to an *activation function* (the *sigmoid function* is often used as an activation function, see *Figure 2* and eq.2. Though recently *ReLU* has become more common, see *Figure 11* and eq.37), which gives an output that is sent to the next layer of neurons. This output differs from the

perceptrons since it is not binary [15], [16]. The great difference is that here a small change in the weights and bias will only cause a small change in the output, which is beneficial for the learning process.

$$f(x) = \frac{1}{1+e^{-x}} \tag{2}$$

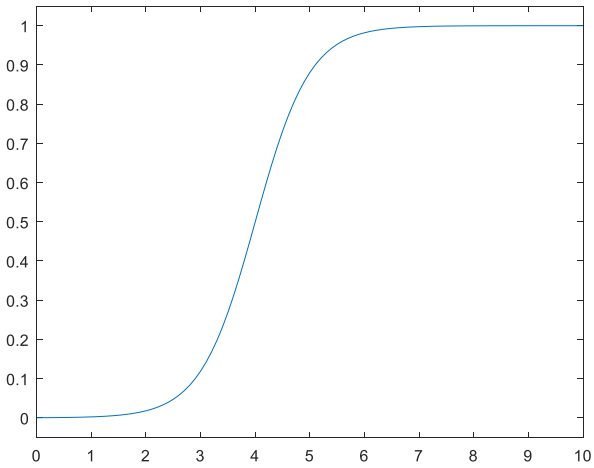

Figure 2 The sigmoid function.

When training the system, it is common to divide the data (supervised learning is used, which means that each input data has a label of its ground truth) into training data, validation data and test data (where the training data is of greater size than the validation data and test data combined). The training data is used as input to the system, where the biases are initialized (can be set to zero) and weights are randomly initialized (the initialization part can be done in several ways, but random is the most common one). The system is then evaluated by a cost function at the end of the network. This is the function that tells us if the weights and biases are well chosen, or if they have to change. The goal is to find the global minimum of the cost function (though it is easy to get stuck in a local minimum). To minimize the cost function, the earlier mentioned backpropagation method is used combined with gradient descent. As the name of the method reveals, the error is propagated backwards in the network and simultaneously update the biases and weights with help of the backpropagated cost function. Within this method, and the whole network actually, there exists several important parameters that control other parameters. These *hyperparameters* have to be selected somehow. It is here the validation data is used. After each epoch (one run over the network with all training data), when new weights and biases have been calculated, the network is tested with the validation data. To iteratively do this after each epoch, one can plot the cost function of the training data and the validation data. From this plot it is possible to extract information that tells whether a certain hyperparameter should be changed or not. Though this is an iterative process, and training neural networks consists of lots of trial-and-error to find the optimal hyperparameters, if it is even possible [17], [18].

After this whole procedure the test data is used to evaluate the network as a final product.

4 Convolutional Neural Networks

A convolution is a mathematical operation that can be done in different dimensions. Here a 2D convolution is used. This means that a filter is swept over the entire matrix (image), element by element (left to right, top to bottom). If e.g. a 3x3 filter is used, then the filter is placed over a 3x3 area of the matrix, and each and corresponding element is multiplied with each other and then everything is divided by the sum of the kernel (the division of the sum is not a part of the definition of a convolution, but it is implemented in the area of deep learning). The output is that value placed at the same location as the center of the 3x3 area in the new matrix output. If the matrix to be convolved isn't padded with zeros, the output from the convolutional procedure will be reduced in size compared to the input. In *Figure 3* a 3x3 kernel is convolved over a non-padded 8x8 matrix (only the first operation in the convolution is being showed. The division of the sum of the filter can be assumed to have already been done in the filter), which results in an output of reduced size compared to the input.

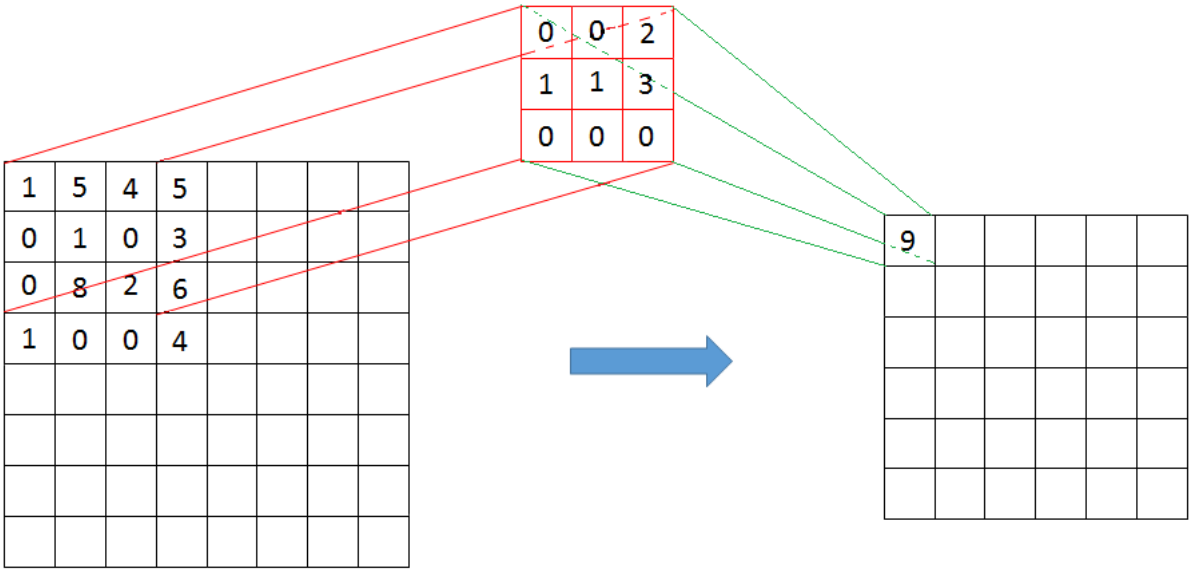

Figure 3 The first operation for the first output pixel of the 2D convolution.

In an ANN, the layers are said to be fully connected. This means that each unit, or neuron, in a layer is connected to all the units in adjacent layers.

When patterns are to be recognized in images, a convolutional neural network (CNN) is recommended to use. This is because of several reasons (e.g. that ANN doesn't take the spatial structure into account), but one important reason is the computational complexity. E.g. if the input is an image of size 50x50 pixels, this means that there is a need of 2500 units in the input layer. And then to have several hidden layers, with several hidden units, with thousands of training-images, one can easily imagine the computational effort, especially if the trial-and-error of finding the correct hyperparameters is also considered.

So, CNNs are great for classification of images and are fast for training deep, multi-layer networks. Though, with CNN there are more hyperparameters to decide/optimize (as far as possible, since optimizing all the hyperparameters is not a converging process), which will be mentioned further on.

Convolutional neural networks use three basic ideas:

- Local receptive fields
- Shared weights and biases
- Pooling

4.1 Local Receptive Fields

In fully connected layers, the input can be seen as a vertical line of neurons. In CNNs, the input can be seen as a square of the same size as the input image, where every pixel is replaced by a neuron (see *Figure 4*).

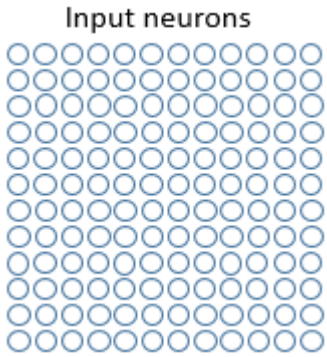

Figure 4 The input neurons.

As in the ordinary neural network, the input neurons/pixels are connected to the hidden layer of neurons. But instead of the case where every neuron in the input is connected to every neuron in the hidden layer, the connections are made in small localized regions of the input image. I.e. each neuron in the first hidden layer is connected to a region of input neurons, as in *Figure 5*.

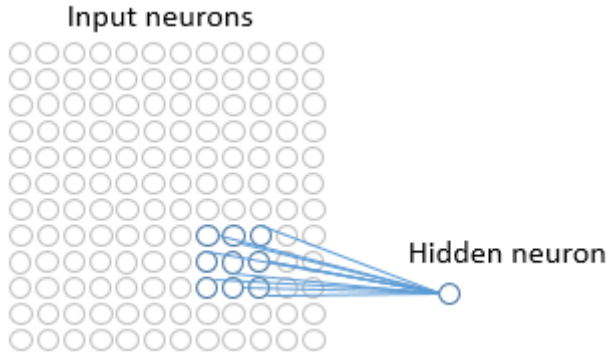

Figure 5 Local perceptive field.

The region in the input image, which are connected to a hidden neuron, is called the *local receptive field* for that hidden neuron (size m x m). It can be thought of as a

little window on the input pixels where each connection learns a weight and the hidden neuron learns an overall bias.

The local receptive field is slid over the whole image, and for each local receptive field there is a different hidden neuron in the first hidden layer (this is the convolution part). When the field is slid across the image, it can move one or several pixels at a time. How many pixels the window is being moved is called the *stride length*.

4.2 Shared Weights and Biases

For each hidden neuron, there are m x m weights and a bias. What is special about this is that all the hidden neurons have the same m x m weights and bias. This means that all the hidden neuron in the first hidden layer detect the same feature (a feature could be a vertical line, corner, etc.) in the input image, just at different locations. Therefore, CNNs are well adapted to the translational invariance of images. Because of this, the *map* from the input layer to the hidden layer is called a *feature map*. The weights and bias defining the feature maps is called the *shared weights* and the *shared bias*.

In a CNN, each hidden layer (also called *kernel* or *filter*) has several feature maps, since each feature map only recognize one feature in an image (see *Figure 6*).

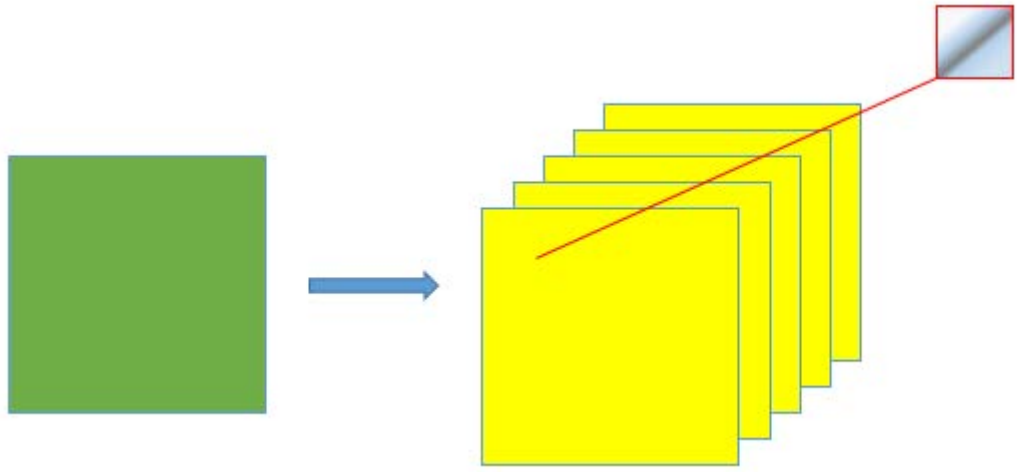

Figure 6 The first convolutional layer generates several feature maps (5 here) from the input image. Each feature map has its own filter (the trained weights) and therefore each feature map finds different features in the input image.

The advantage of using shared weights and shared bias is that it reduces the number of parameters in the CNN.

4.3 Pooling

CNNs contain layers called *pooling layers*, which are used after the convolutional layer. The pooling layer simplify the information in the output from the convolutional layer (make each feature map condensed).

A common pooling-method is the *max-pooling*. This method takes the maximum activation output in a k x k (e.g. k=2) region, which becomes the pooling unit (see *Figure 7*). The pooling is done to each feature map separately.

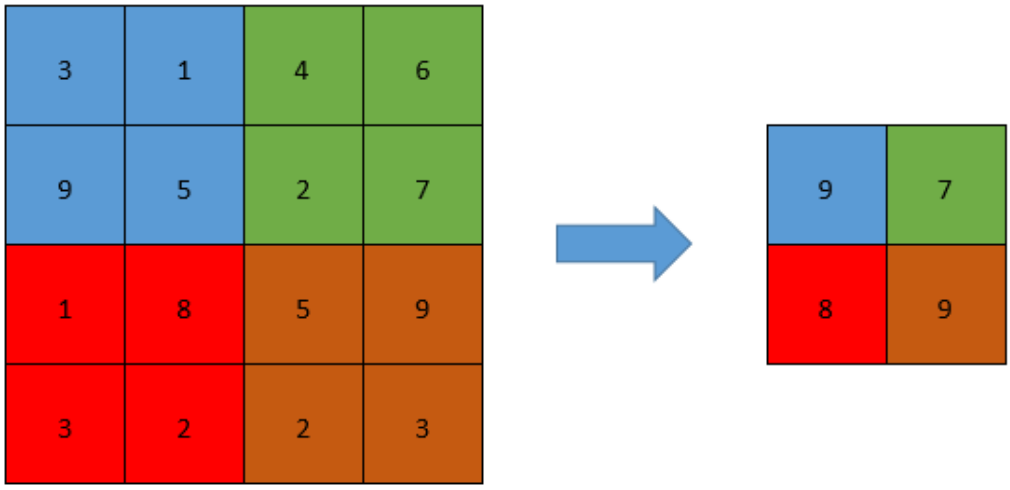

Figure 7 The procedure of max-pooling (stride length equal to two).

Max-pooling can be thought of as a way of asking the network if any feature is found in the image. If so, the exact positional information is thrown away. The only important part for the network is the position relative to other features.

Pooling layers provide an invariance to small translations of the input. There are several different pooling operations, e.g. max-pooling, average-pooling, mean-pooling, etc. The most common one is max-pooling. The general idea of pooling is to locally aggregate the input by applying a nonlinearity to the content of some patches. The parameter of the pooling layer is to decide the pooling area. If the pooling area is e.g. 2x2, then a patch of size 2x2 is slid all over each feature map. The stride length has to be set here as a hyperparameter. That is for how many pixels the patch should move at a time. E.g. with stride equal to one there will be overlapping with patch-size 2x2, but with stride equal to two there won't be any overlap.

The output is the same amount of feature maps, reduced in size, with more important and compressed information. If a feature map (input) gets pooled by this max-pooling layer (if the patch-size is 2x2), the size is reduced by half. For each 2x2 patch, there is only one-pixel output, the maximum value of the four values of the patch. And for average-/mean-pooling, it works in the same way, but instead of the maximum value of the four pixels, the output is the average/mean of the four pixels.

4.4 The Network

The CNN is these three parts put together. Usually the network ends with a fully connected layer (or several fully connected layers), which are exactly the same as described for ANNs.

The backpropagation method is also used for CNNs for training, as it was described for ANNs.

5 Fully Convolutional Networks

A fully convolutional network (FCN) [19] is where both downsampling (an ordinary convolutional neural network) and upsampling (also called deconvolution [20]) are used together to give an output of the same size (not necessarily exactly the same size) as the input of the network. These types of networks are often used in segmentation tasks, since the output is the input with a highlight of the area of the segmented object. Instead of getting an output from the downsampling part that is a label from the classifier, the output is features that is being used as input in an unpooling step or deconvolutional step (the definition of a deconvolutional layer is different in different papers). E.g. in the API TensorFlow [9] one can use the function tf.image.resize_images which uses interpolation. A more common approach is to use the transpose of a convolution with stride larger than one, to increase the size of the image. A proper way to do this, is to use the same techniques as in the U-net [3] (which has been implemented in other networks as well), where the convolutional output from each layer in the downsampling part is sent over to the corresponding upsampling layer, and concatenated as an input to that layer. Another way is to send over the pooling information in the downsampling part to the upsampling part, and then use a trained filter for unpooling/deconvolution. This improves the spatial information in the upsampling part and therefore also the whole fully convolutional network.

5.1 Upsampling

The part of the network that generates an image from features of a smaller dimension, is what defines a fully convolutional network. This upsampling part can be solved in different ways, either by unpooling [21] (several techniques for this) or by deconvolution (also called convolutional sparse coding [22]). A technique mentioned in [22], is to first use deconvolution to scale up the image, combined with information from the corresponding pooling layer in the downsampling part of the network, which create "switches" (the indices of the activated elements in the pooling step) that contain information about the location of the features (see *Figure 8* [22]).

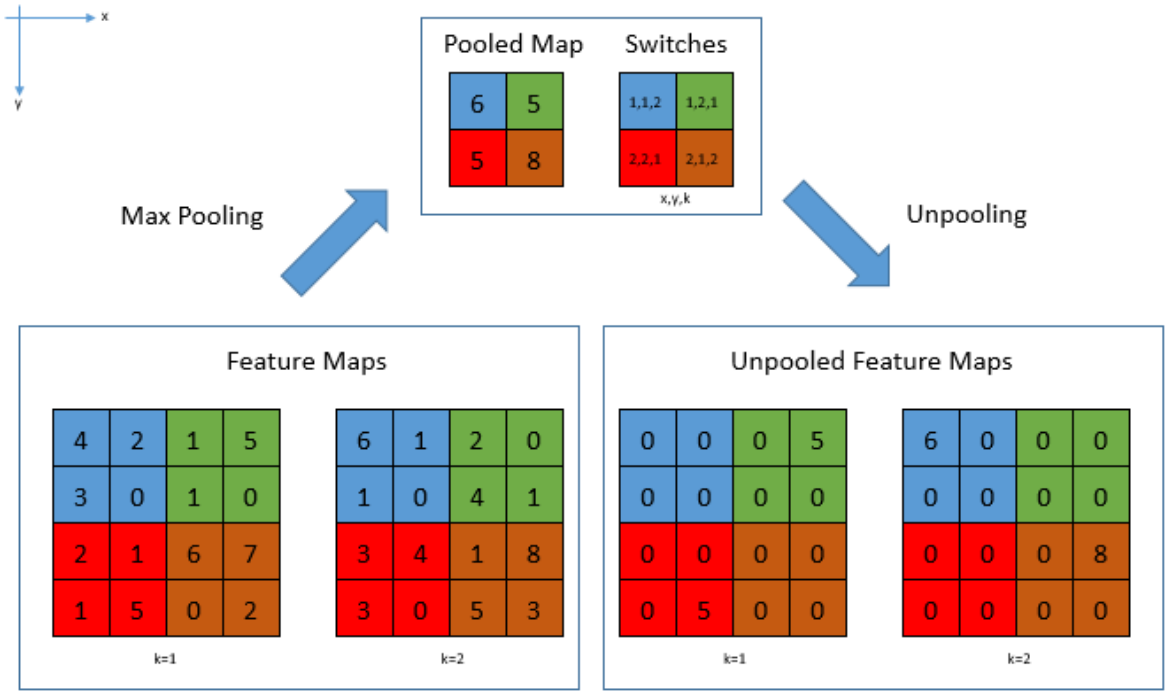

Figure 8 Unpooling/upsampling with the use of switches, where x,y is the pixel location and k is the kth feature map in the kernel.

To perform a deconvolution (here used as upsampling) can be done by using a transposed convolution layer with a stride greater than one (if padding is used).

Transposed convolutions [23] are used when going from something with the shape of some convolution output, to something that has the shape of its input while maintaining a connectivity pattern that is compatible with the mentioned convolution [24]. For upsampling they are used to project feature maps to a higher dimension. This is done by flipping the filter kernel over the first and second dimension and then just perform a convolution.

The difference between a 2D convolution and a 2D transposed convolution is that in the convolution, as mentioned earlier, the dot product is used between the image and the filter. In the transposed convolution, each input pixel is multiplied with a learned filter and the output is the new upscaled matrix. Since it is a convolution, the filter is slid over the whole input which gives a superimposed output where each filter-multiplication is added where it is superimposed. By using a stride length greater than one (if padding is used), the size of the output will be larger than the input, i.e. there will be an upsampling since the stride here is an output-stride where it is about how to move the filter, instead of an input-stride as in an ordinary convolution where it is about how many pixels of the input matrix to move at a time. See *Figure 9* to better understand the transposed convolution procedure by using an example [25].

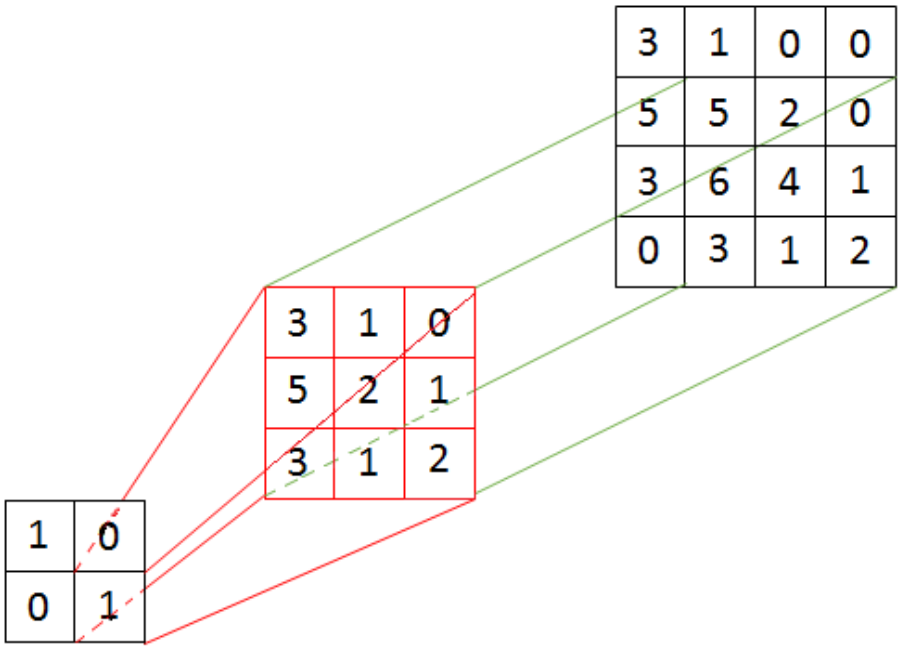

Figure 9 The procedure of transposed convolution/upsampling. In the example, the result shown is after the last input element. Here there are stride length one and no padding is used.

What is good with the transposed convolution compared to an interpolation, e.g. bilinear upsampling, is that the transposed convolutional layer is learning, while bilinear upsampling is fixed [19].

6 Preprocessing & Data

The simplest form of preprocessing might be a linear transformation of the input data. A more complex form is dimensionality reduction, which can improve the result even though information is reduced.

To gain some prior knowledge and use it in the preprocessing stage can improve a neural networks' result dramatically.

Regarding the dimensionality reduction, it can be as simple as discarding a subset of the original input data, or it can consider approaches involving forming linear or non-linear combinations of the original variables to generate inputs for the neural network. These combinations of input are called *features*, and the process of generating the features is called *feature extraction*.

The motivation of dimensionality reduction is that it can reduce the impact of the worst effects of the consequences of high dimensionality. If a network has fewer inputs it has fewer adaptive parameters (e.g. weights) to be determined, which often means that the network gets better generalization properties. Also, a neural network with fewer weights may be faster to train.

Just consider the example of having an image of size 250x250 as input. This means that there are 62500 input units and therefore 62501 weights (including the bias) for every hidden unit to learn. A huge computational resource would be required to

find the minimum of the cost function. To tackle this problem, preprocessing the data can be a solution. A technique of dimensionality reduction can be considered, where *pixel averaging* is used on a block of pixels of the input image. The averaged pixels are examples of features. This procedure reduces the information, and can therefore lead to poor results. However, this problem doesn't exist for convolutional neural networks, which will be described further on [26].

Regarding the simpler preprocessing task, it is important to normalize and rescale (linear transformation) the data before it is used as input to the network. Each input variable should have a zero mean and a unit standard deviation over the transformed training set. Because of this, the weights can be initialized randomly.

For ANNs there are way more preprocessing-tasks to consider, compared to CNNs. Another example (other than mentioned above) is *missing value*, which means that an input variable is missing its input data. There exist different techniques to handle this issue (some better than the other), e.g. to express these variables in terms of regression over the other variables using the available data, and then use the regression-function to fill in the missing values.

However, these issues are not discussed regarding CNNs (in the same way as for ANNs). For CNNs there are two main preprocessing tasks:

- Normalization
- Augmentation

The main reason why this is the case is that the CNN itself perform a feature extraction, which will be described further on.

6.1 Normalization

The pixel values often lie in the range [0,255], and feeding these values into the network can cause the neuron to saturate (it basically stops learning). Therefore, the learning will be really slow. Some preprocessing techniques are:

- *Mean/Median image subtraction*
- *Per-channel normalization* (the method mentioned above, with zero mean and unit standard deviation)
- *Per-channel mean subtraction*
- *Whitening* (turn the distribution into a normal distribution)
- *Dimensionality reduction* (e.g. PCA, but this is not that common in deep learning)

The rule that is important to keep in mind is, always use the simplest method (the easier, the more effective).

6.2 Augmentation

Overfitting is when the network learns the training data too well, and doesn't get generalized. To prevent the network from overfitting, a great way is to create artificial data. This is often done by simply mirroring, rotating, etc. the input image.

6.3 Segmentation

All the input images need to be of the same size, which can be achieved by cropping or padding each image if they might be of different sizes.

In fully convolutional networks the ground truth, or the label, for each input image will be a binary mask in two-class classification problems. These binary masks normally contain zeros as background and ones as foreground. They will have the same size as the output from the network.

6.4 StemNet

The different time-laps sequences contain different sizes of the images. Therefore, the images have been resized to 1024x1024. Either the images have been cropped or padded.

The images are frames from time-lapse sequences from a microscope. Unfortunately, there are usually features like stationary debris, a microwell or regions of non-uniform illumination in the background which interfere with the segmentation [2]. Fortunately, one can perform a background subtraction to remove most of these features. This is done by computing the background image and subtract it from all the images in a sequence (See *Figure 10*). Each frame in a sequence has been preprocessed by removing the time-axis median of the space-time volume spanned by the image data, for each pixel in a sequence from each frame. If a voxel with spatial coordinate (x, y) and at time t is written as $I(x, y, t)$, the background pixels (of that time-lapse sequence) are given by:

$$I_{bg}(x, y, t) = \underset{t\epsilon\,\{1,\ldots,T\}}{\text{median}}\, I(x, y, t) \tag{3}$$

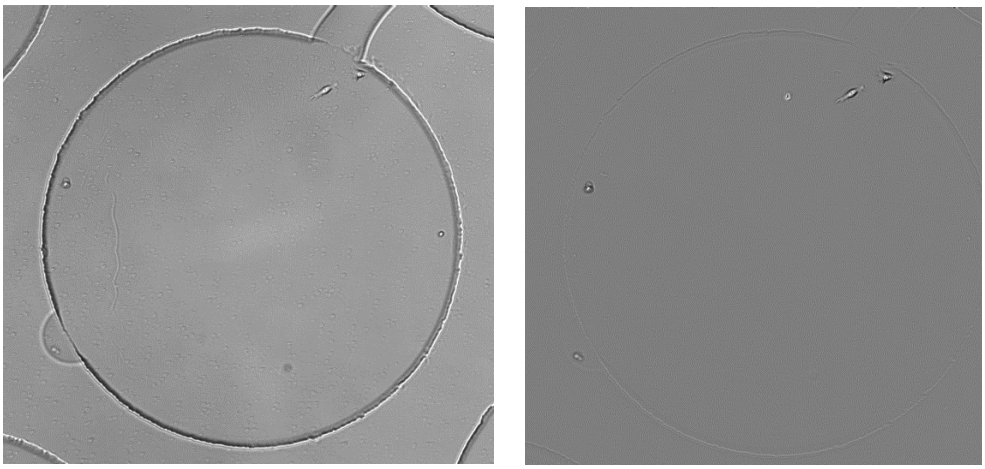

Figure 10 The left image (1114x1040) is an original frame from a time-lapse sequence of frames from the microscope, and the right image (1024x1024) is the same frame after it has been preprocessed.

Each frame has a corresponding binary segmentation mask that is used as the truth for where there are stem cells (if any) in the frame. These segmentation masks are also of size 1024x1024. All the black pixels correspond to background and all the white pixels corresponds to foreground and stem cells.

7 Learning

A neural network learns by updating/changing its weights and biases. That is an iterative process which takes a lot of time and needs computational resources. In general, the network learns by making a forward pass of the input, calculating the error of the prediction, and then passing backward the error in order to change the weights and biases after how wrong the prediction was. The method of the backward pass is called backpropagation, since the errors are propagated back through the network.

7.1 Forward Pass

During the forward pass, the input is sent through the network, where each neuron, in each layer, either activates (send an output) or doesn't (no output). This will lead to different outputs of the whole network, depending on the values of the weights and biases. So for a specific input and specific weights and biases, the network gives an output that is compared with the truth, i.e. the correct classification (for example) for that input.

7.2 Cost Function

The error of the prediction of the network, which is calculated by the cost function (also known as *the objective function*), is also known as the cost or the loss. This can be calculated in different ways, i.e. there are several different cost functions and therefore the way of calculating the error is case dependent. The best known cost functions are:

- Quadratic cost [13]
- Cross-entropy cost [13]
- Dice coefficient [27]
- Exponential cost [28]
- Hellinger distance [29]
- Kullback-Leibler divergence [30]
- Generalized Kullback-Leibler divergence [31]
- Itakura-Saito distance [32]

The most common/used cost functions are the quadratic cost and the cross-entropy cost. These both cost functions has the properties one wants from a cost function [13]:

- It should be positive
- Goes towards zero when the neurons get better at computing the desired output

But the cross-entropy cost function also has a property which makes it better than the quadratic cost in most cases, which is that it avoids the problem of slow learning, which can be common with the quadratic cost function.

7.2.1 Quadratic Cost

The quadratic cost is often mathematically more tractable than other cost functions because of the properties of variances and that it is symmetric. I.e. an error above the target causes the same cost, or loss, as an error of the same magnitude below the target. The quadratic cost is defined as:

$$C = \frac{1}{n}\sum_x C_x \,, \tag{3}$$

where C_x is the cost for each input., e.g. if the cost function is the quadratic cost:

$$C_x = \frac{||y_{truth} - y_{pred}||^2}{2} \,. \tag{4}$$

7.2.2 Cross-Entropy Cost

The cross-entropy considers two probability distribution, which are the true probability (the true label) and the given distribution (the prediction). The cross-entropy gives a measure of similarity between these two probability distributions. The cross-entropy is defined as:

$$C = -\frac{1}{n}\sum_x y_{truth} \ln y_{pred} + (1 - y_{truth}) \ln(1 - y_{pred}) \,. \tag{5}$$

In tasks of segmentation, dice coefficient can come in handy to use as a cost function.

7.2.3 Dice Coefficient

The dice coefficient [27] is the intersection divided by the union of y_{truth} and y_{pred} multiplied by a factor of two, i.e. calculated as:

$$DC = \frac{2|y_{pred} \cap y_{truth}|}{|y_{pred}| + |y_{truth}|} \,. \tag{6}$$

Then the final cost, or loss, is calculated as:

$$Cost = 1 - DC \,. \tag{7}$$

7.3 Softmax

Softmax is not a necessary operation. To apply a softmax on the output means that the output will be a probability distribution where each output is a positive number between 0 and 1 and all output sums up to 1. The softmax equation is the following [13]:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \,, \tag{8}$$

where j corresponds to the jth output neuron of the network (output layer L) and the denominator is the sum over all output neurons.

Each output from a softmax will have a probability of belonging to one of the classes in the classification problem.

7.4 Optimizer

The main goal, in order to optimize the performance of the network, is to minimize the cost, or loss, from the cost function. The best known optimizers are:

- Gradient descent [13]
- SGD [13]
- ADAM [33]
- Momentum [13]
- RMSprop [34]
- AdaGrad [35]
- Ftrl Proximal [36]
- AdaDelta [37]
- ProximalAdaGrad [38]

The gradient descent algorithm is the most common/used one (considering its variants like SGD, Momentum etc.) and it finds a minima by first calculate the gradient of the error function, then move in the opposite direction of the gradient, to end up at a minima. It is here where the *learning rate* comes in. The learning rate is a factor multiplied to the gradient and decides how large steps to take when locating the minima. Usually the learning rate is small. If the cost function is called C, then the steps of the gradient descent algorithm can be written as [13]:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla \mathrm{C}||^2 , \tag{9}$$

where η is the learning rate and ∇C is the gradient of the cost function. ΔC is the change which should be negative and always decrease since we want to find a minima. This can be rewritten in component format, to see how the weights (w_k) and biases (b_l) updates/changes during training:

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{10}$$

$$b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l} . \tag{11}$$

The most common cost function is called Stochastic Gradient Descent (SGD), but in

CNNs two other cost functions are becoming more common, ADAM and Momentum, which are faster variants of SGD.

7.4.1 SGD

When calculating the gradient of the cost function, one has to calculate the gradient for each training input separately, i.e. [13]:

$$\nabla C = \frac{1}{n}\sum_x \nabla C_x \,. \tag{12}$$

This can take a long time when there are a lot of training inputs. Therefore, SGD is used to speed up the learning.

By using SGD we estimate the gradient by computing the gradient of a single input for a small sample of randomly chosen inputs, instead of using all the inputs. Then by averaging over these samples, a good estimate of the true gradient can be calculated. So if the random samples are $X_1, X_2, \ldots, X_m$ and:

$$\frac{1}{n}\sum_x \nabla C_x \approx \frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j} \,, \tag{13}$$

then by using eq.12 and eq.13, the gradient can approximately be approximated as:

$$\nabla C \approx \frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j} \,. \tag{14}$$

This can also be expressed in the components that we want to train, i.e. the weights and biases:

$$w_k \rightarrow w_k' = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{15}$$

$$b_l \rightarrow b_l' = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l} \,. \tag{16}$$

This update is iterated until an enough amount of *mini-batches* (samples) has been used, which is until all the training data has been used, which is also known as an *epoch*.

7.4.2 ADAM (Adaptive Moment Estimation)

This method computes adaptive learning rates for each parameter [33].
Initialize initial 1st moment vector, 2nd moment vector and timestep as:

$$m_0 \leftarrow 0 \tag{17}$$

$$v_0 \leftarrow 0 \tag{18}$$

$$t \leftarrow 0\ . \tag{19}$$

If the cost function is written as $f(\theta)$, then the updates are:

$$t \leftarrow t + 1 \tag{20}$$

$$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \tag{21}$$

$$lr_t \leftarrow learning\_rate \cdot \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \tag{22}$$

$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \tag{23}$$

$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2 \tag{24}$$

$$\widehat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t} \tag{25}$$

$$\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t} \tag{26}$$

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{\widehat{m}_t}{\hat{v}_t + \epsilon}\ , \tag{27}$$

where m_t and v_t are estimates of the first moment and the second moment of the gradients, i.e. the mean and the non-centered variance respectively. $\widehat{m}_t$ and $\hat{v}_t$ are the bias-corrected first and second moment estimate, respectively.

The algorithm updates exponential moving averages of the gradient and the squared gradient. The hyperparameters β_1 and β_2 controls the exponential decay rates of the moving averages, and g_t is the gradient of the cost function at timestep t. The resulting parameter/output is θ_t and α is the stepsize. ϵ is a small value.

7.4.3 Momentum

This technique is based on gradient descent or SGD, it is just faster and more stable (basically) [13]. What has been done in this method is to introduce a friction coefficient, called *momentum coefficient*. This has nothing to do with the momentum from physics. We introduce velocity variables, one for each weight or bias. This makes the learning fast, i.e. we get to the minima faster. The momentum coefficient is necessary to not overshoot and ”pass” the minima. Consider w as both the weights and biases and v as the velocity variable, then [13]:

$$v \rightarrow v' = \mu v - \eta \nabla C \tag{28}$$

$$w \rightarrow w' = w + v'\ , \tag{29}$$

where μ is the momentum coefficient.

7.4.4 General

If the error is small, we don't want to change the weights and biases too much, but if the error is large, we want to change the weights and biases more drastically. Therefore, the derivative of the cost function (error function) with respect to the weights and biases, is what is used in the backpropagation.

7.5 Backpropagation

To be able to use the methods mentioned above, one has to be able to calculate the gradient of the cost. This is done by an algorithm called *backpropagation* [13].

The backpropagation gives an expression of the derivative of the cost function C with respect to the weights and biases. From that we can get how fast the cost changes depending on the changes of the weights and biases in the network.

For a cost function to be able to be used for backpropagation, it needs to fulfill two criteria:

- it can be written as an average over cost functions for individual training examples
- it can be written as a function of the outputs from the network

We calculate the partial derivatives $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$ where w is the weight from neuron k in layer $(l-1)$ to neuron j in layer l and b is the bias of the j neuron in layer l. Then we relate δ_j^l to each partial derivative, which is the error in the j neuron in layer l.

What is wanted is for the neural network to perform with an accuracy as high as possible. To increase its accuracy, one has to minimize the error. The error comes from that the output from an activation function is different from what is optimal. E.g. from activation function output a we want the output $\sigma(z_j^l)$, where:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \,. \tag{30}$$

But what we actually get is $\sigma(z_j^l + \Delta z_j^l)$. When this propagates through the network, the final cost will be $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$. Therefore, we choose Δz_j^l so that we minimize the final cost as much as possible. E.g. Δz_j^l can be set to the opposite sign of $\frac{\partial C}{\partial z_j^l}$, to make sure that $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$ is small. This tells us that $\frac{\partial C}{\partial z_j^l}$ can be seen as the error in each neuron, i.e.:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}\,. \tag{31}$$

The backpropagation algorithm is based on four fundamental equations:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)\,, \tag{32}$$

or in matrix-form:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{33}$$

$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'(z^l) \tag{34}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{35}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l\,, \tag{36}$$

where $\odot$ is the Hadamard product, also known as elementwise multiplication and σ is the activation function (usually sigmoid or ReLU, but here a general activation function).

Eq.32 is the error in the output layer L, eq.34 is the error expressed in terms of the error in the next layer $(l+1)$, eq.35 is the rate of change of the cost with respect to the bias, eq.36 is the rate of change of the cost with respect to the weight.

If a_k^{l-1} in eq.36 is small, it is said that the weight *learns slowly*. Depending on the activation function, this is a consequence of that the neuron has saturated (sigmoid as activation function) or that the neuron doesn't activate because of a too large gradient (ReLU as activation function, see *Figure 11* and eq.37). If the activation function is a sigmoid (eq.2) this means that its output is close to 0 or 1.

$$f(x) = \max(0, x) \tag{37}$$

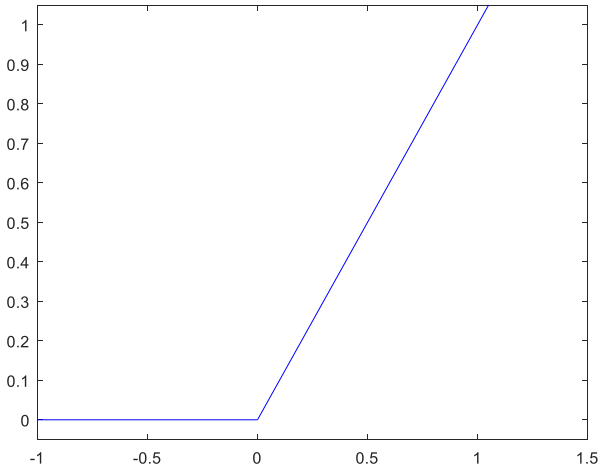

Figure 11 The ReLU function.The activation function only gives an output if the input is greater than zero. This introduces nonlinearity to the network.

All four equations eq.32-eq.36 are consequences of the chain rule:

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L}\,. \tag{38}$$

But the output from the activation function a_k^L of the kth neuron will only depend on the weighted input z_j^L when $k = j$, and can therefore be rewritten as:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\,, \tag{39}$$

which is the same as:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)\,. \tag{40}$$

This is eq.32 in component form. To prove eq.34 one can again apply the chain rule:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\delta_k^{l+1} \tag{41}$$

$$z_k^{l+1} = \sum_j w_{kj}^{l+1}\, a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1}\, \sigma(z_j^l) + b_k^{l+1} \tag{42}$$

and by differentiating:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}\sigma'(z_j^l)\,, \tag{43}$$

which implemented in eq.41 gives us eq.34 in component form:

$$\delta_j^l = \sum_k w_{kj}^{l+1}\, \delta_k^{l+1}\sigma'(z_j^l)\,. \tag{44}$$

For eq.36 we can use the chain rule as:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l}\,, \tag{45}$$

where the first factor has already been proven to be the error and the second factor:

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial}{\partial w_{jk}^l} \sum_j w_{jk}^l \, a_j^{l-1} + b_k^{l+1} = a^{l-1} \, . \tag{46}$$

By implementing eq.46 in eq.45:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} a^{l-1} = a^{l-1} \delta_j^l \, . \tag{47}$$

For eq.35 the first derivative factor is the same as eq.45 and the second derivative factor is:

$$\frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial}{\partial b_j^l} \sum_j w_{jk}^l \, a_j^{l-1} + b_k^{l+1} = 1 \, . \tag{48}$$

By implementing eq.48, the final result is:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \, . \tag{49}$$

8 Overfitting

An issue with training neural networks, that appears quite often, is overfitting. That means that the network learns the training data too well and isn't general at all. I.e. if an overfitted network should e.g. classify an image that has not been included in the training data, the network simply cannot get it correct.

A sign of overfitting is, when looking at the loss or accuracy of the training data and the validation data, if the two graphs starts to go different ways. More specific, if the validation accuracy decreases while the training accuracy increases or if the validation loss increases while the training loss decreases (see *Figure 12*).

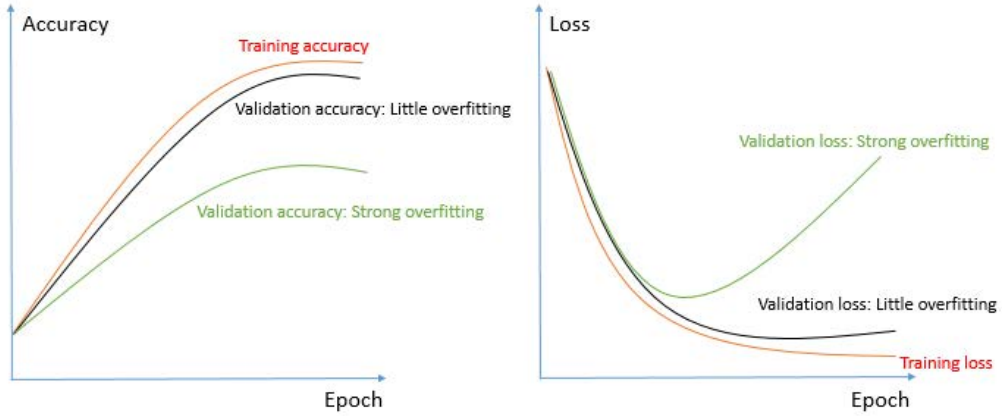

Figure 12 Signs of overfitting by looking at the accuracy and loss of the training and validation.

There are different techniques to prevent overfitting (one mentioned earlier), either used separately or combined:

- dropout
- data augmentation
- regularization term
- batch normalization

8.1 Dropout

Dropout [39] is a way of reducing overfitting in a network by in each forward/backward pass, reduce a pre-decided percentage of the hidden neurons, i.e. temporarily delete those neurons. Then in the next forward/backward pass the network is restored and other hidden neurons (the same amount as earlier) are temporarily deleted (see *Figure 13*). This procedure can be seen as an average scheme after doing this for the whole training.

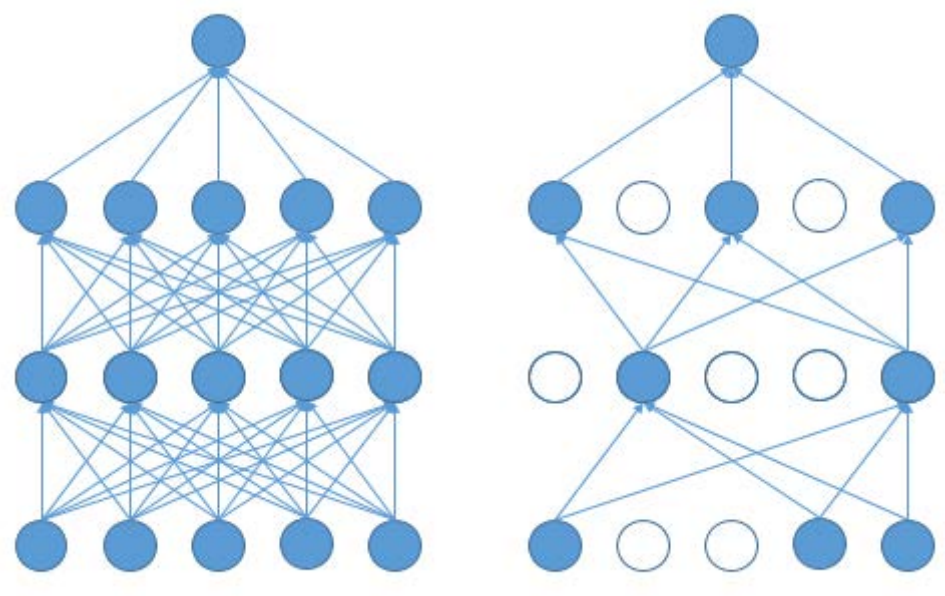

Figure 13 How dropout works, by cancelling out some hidden neurons.

Then when running the network without dropout (which is done on the validation and/or test dataset), the weights coming out from the hidden neurons have to be modified (scaled) since there are much more active neurons than during training.

8.2 Data Augmentation

A cause of overfitting can be because of a too small training dataset. If this is the case, then data augmentation can solve the issue of overfitting. This can be done by either collecting more data (usually very time consuming) or by creating artificial data.

To create artificial data doesn't have to be a tricky thing. It might just mean to rotate and deform (if images), or change in some way the already existing data.

8.3 Regularization Term

To help the network to learn more general results and also to increase its classification accuracy, one can introduce a regularization term to the cost function by simply adding it to the already existing cost function (C_0):

$$C = C_0 + Reg\ , \tag{50}$$

where Reg is the regularization term. There exist different regularization terms, but the two most common are $L1$ regularization:

$$Reg = \frac{\lambda}{N}\sum_w |w| \tag{51}$$

and $L2$ regularization:

$$Reg = \frac{\lambda}{2N}\sum_w w^2\ , \tag{52}$$

where N is the size of the training data set. In the backpropagation algorithm this will only affect the weight update, by introducing a *weight decay*, while not affecting the bias update at all (regularization is used to keep the weights small, so that the network doesn't overfit. Using a regularization on the bias wouldn't make sense, since it would keep the bias of a layer closer to the origin, even though the training data might be strongly biased. That would just limit the learning expressiveness of the network.):

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{N}w \tag{53}$$

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}\ . \tag{54}$$

It is called weight decay since the new learning rule for weights become (the learning rule for the bias remains the same):

$$w \to w - \eta \frac{\partial C_0}{\partial w} - \eta \frac{\lambda}{N} w = \left(1 - \eta \frac{\lambda}{N}\right) w - \eta \frac{\partial C_0}{\partial w} . \tag{55}$$

The regularization term is preferred since it gives smaller weights and thus provides a simpler and more powerful explanation for the data.

8.4 Batch Normalization

Batch normalization [40] is something that performs a normalization for each layer in the network. It is applied before the activation function. The idea is to handle the internal covariate shift, since there might be a difference in the distributions of the input data between the layers in the network.

In deep networks the input to each layer depends in all the previous parameters in the network. Therefore, a small change of the weights can be the cause of a larger error in the end. Without batch normalization this is prevented by using small learning rates and being careful with the weight initializations.

In a convolutional neural network, the normalization is performed as the following:

$$y_{ijcb} = w_k \frac{x_{ijcb} - \mu_c}{\sqrt{\sigma_c^2 - \epsilon}} + b_c \tag{56}$$

$$\mu_c = \frac{1}{HWB} \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{b=1}^{B} x_{ijcb} \tag{57}$$

$$\sigma_c^2 = \frac{1}{HWB} \sum_{i=1}^{H} \sum_{j=1}^{W} \sum_{b=1}^{B} (x_{ijcb} - \mu_c)^2 , \tag{58}$$

where the data batch has the dimension $HxWxCxB$ and H is the height of the features, W is the width of the features, C is the number of channels and B is the batch size. ϵ is to prevent numerical issues.

As a consequence of batch normalization, dropout is often not necessary to use, since it also has regularization properties.

9 The networks

Two different networks are being considered. A 12 convolutional layer (from size 1024x1024 to size 32x32, back to size 1024x1024) network is trained using temporal information ($Network_5$) and no temporal information ($Network_1$) and then analyzed.

The comparison should answer if the use of temporal information increases the performance of the fully convolutional network or not.

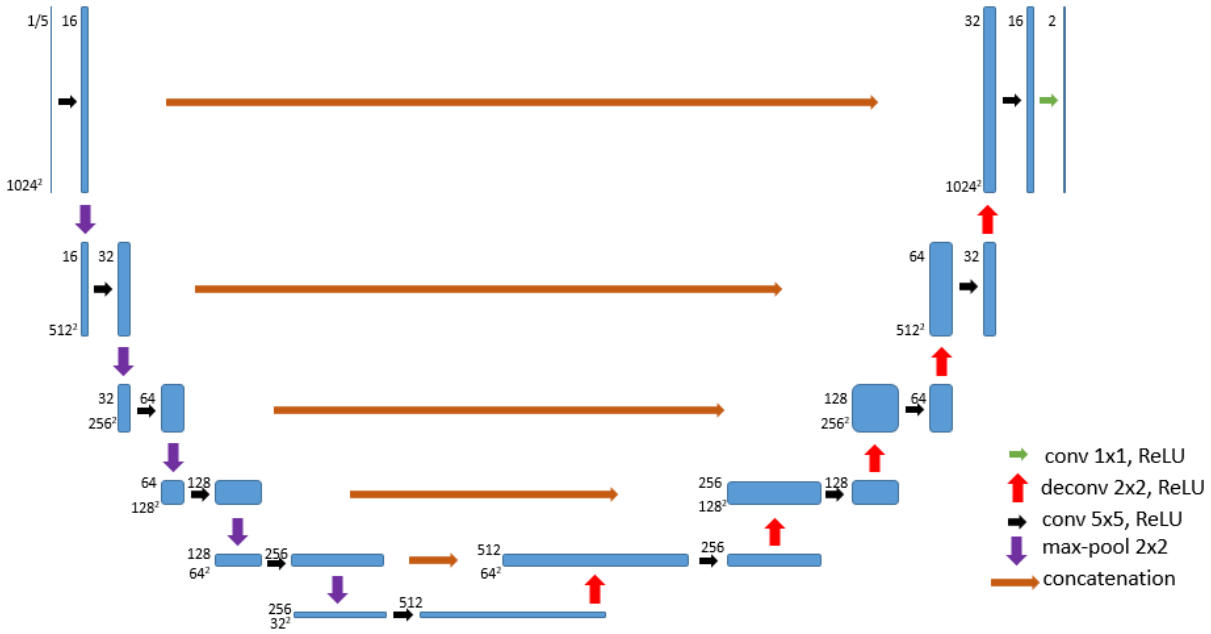

Figure 14 The network architecture. For temporal training a 5 channel input is used, while a 1 channel input is used otherwise.

The fully convolutional network consists of convolutional layers, max-pooling layers, ReLU activations, deconvolutional (transposed convolutions with stride 2) layers where the input is the output from the lower resolution convolution from one step earlier concatenated with the corresponding convolutional output from the downsampling part. The output is a convolutional layer with patch-size 1x1 to do a pixel wise classification, for comparison with the one hot-encoded segmentation mask (it could also have been done with only a one layer output, by doing some modification on the cost function). The output is run through a softmax before calculating the cost (or loss).

The fully convolutional network will be trained using temporal information to see if this approach can achieve better results than without the temporal information, i.e. the fully convolutional network will also be trained without the temporal information. Network_5 and Network_1 will have the same architecture, but the input will be of 5 channels (temporal information) for Network_5 and of 1 channel (grayscale) for Network_1. In more detail, for every input frame of Network_5, frames in the sequence of the time-lapse that are before (two frames) and after (two frames) the input frame will also be used as input with that input frame. If the input frame is denoted as $X_{i,j}$ and the output label for that frame is denoted as $Y_{i,j}$, where i is the sequence number and j is the frame number, then the other frames sent into the network for training will be $X_{i,j-2}$, $X_{i,j-1}$, $X_{i,j+1}$ and $X_{i,j+2}$ even though the label used to compare the output with is only $Y_{i,j}$. See *Figure 14* for the network architecture.

10 Measurements

It is important to measure the results from the network properly, so that the conclusions of each parameter setting and each architecture gets evaluated in a way that makes it easy to see if something is bad or good. One way to measure this is to calculate the loss and the accuracy, both for the training set and for the validation set. Then by comparing the two graphs conclusions can be made, e.g. if the network is overfitting or underfitting (the opposite of overfitting, often occurs if the model is too simple). The standard way of calculating the accuracy is in the following way [41]:

$$TP = True\ Positive$$

$$TN = True\ Negative$$

$$FP = False\ Positive$$

$$FN = False\ Negative$$

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{59}$$

The calculation of the loss depends on which cost function that is being used. A commonly used cost function is the cross-entropy. Though, in the case of image segmentation and the usage of a binary segmentation mask, it might be necessary to use a weighted cost function. This is because if the object to be classified just are very few pixels of the mask and the rest are zeros. Then the accuracy would be very high, even if it didn't find any object. A weighted cost function, if the cross-entropy cost function is used, is (if the ones are the wanted class):

$$C = -\frac{1}{n}\sum_x w_{class} y_{truth} \ln y_{pred} + (1 - y_{truth}) \ln(1 - y_{pred}) , \tag{60}$$

where $w_{class} > 1$, so that the calculation of the error on the wanted class is more sensitive (i.e. a misclassification should cost more for the wanted class than the other class).

Other important measurements are precision and recall, F1 score, and also the error rate:

$$Precision = \frac{TP}{TP+FP} \tag{61}$$

$$Recall = \frac{TP}{TP+FN} \tag{62}$$

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} = \frac{2TP}{2TP+FP+FN} \tag{63}$$

The error rate shows how many pixels correctly classified as a percentage.

In the end, what is important to find out is for every input image, how many pixels does the network classify as stem cell and not a stem cell.

11 Imbalanced Classes

An issue within the area of machine learning, is that the dataset used for training can be unbalanced. Consider a classification of two classes (the same applies for a multi-class problem) and that one of the classes is overrepresented compared to the other class. Then there are imbalanced classes, and the dataset is unbalanced. The consequence of this is that if the algorithm just classifies everything as the overrepresented class, the accuracy will be very high, even though the algorithm has failed its task.

There are several approaches to this issue, and which one to choose is dependent on the task. Some techniques to approach this problem are [42], [43]:

- Add weights, so that the overrepresented class gets a low weight and the underrepresented class gets a high weight (see eq.60)
- Resample the dataset
- Increase the size of the dataset
- Use other metrics than accuracy
- Generate synthetic samples
- Use penalized models

The following descriptions considers the area of image segmentation, two-class classification and deep learning, but the principle is the same for machine learning-algorithms in general.

11.1 Add Weights

When having imbalanced classes, it is important to take this into account when building the network. One way to do this is to add weights in the loss function. If not, the network will think it does fine/well, when it e.g. is actually missing every pixel of one class. In practice this means that all the pixels classified as the overrepresented class gets multiplied with a lower weight than what the pixels of the underrepresented class gets multiplied with. How to decide the weights is case dependent.

One way to decide the weights is to take the pixel frequency median of the whole dataset divided through the pixel frequency of the class [12], [44], for each class. This gives a smaller weight for the larger class and a greater weight for the smaller class. Another way to perform class balancing is to multiply a weight only to the underrepresented class in the loss function, which is the total amount of pixels of

the larger class divided by the total amount of pixels of the smaller class, over the whole training data. Though in some cases, it might be the best to choose an arbitrary weight and by trial-and-error iterate the most proper weight.

11.2 Resample the Dataset

One way to tackle the issue of imbalanced classes is to resample the dataset. This means that we remove the images that are the most overrepresented of one class. This decreases the size of the dataset, but it might help.

11.3 Augmented Dataset

To augment the dataset, if possible, is a way to make sure that either we increase the dataset of images that are more equally represented between the classes, or just that we collect more data so that we don't decrease the size of the dataset too much if we resample the dataset as mentioned above.

11.4 Other Metrics

The issue of class imbalance often leads to the accuracy paradox [45]. Then it can be good to use other metrics, like:

- Confusion Matrix: Shows the correct predictions and the incorrect predictions made
- Precision: Shows the classifiers exactness
- Recall: Shows the classifiers completeness
- F1-Score: Shows a weighted average of precision and recall
- Kappa: Shows the accuracy normalized by the imbalance of the classes in the dataset
- ROC: Shows the accuracy divided into sensitivity and specificity

11.5 Synthetic Samples

It can be possible to create more samples of the underrepresented class. In the case of stem cells, this would mean to add fake stem cells in the images, that make the balance between the classes more equal.

11.6 Penalized Models

Using this approach means that one adds a penalizing term to the cost when the network misclassifies the minority class, so that the network learns that it is more important.

12 Temporal Segmentation & Classification

When using a neural network for the task of segmentation, one want that the output is the input image with a highlighted area of the object to be segmented. To train a neural network to perform this, it is necessary to use binary segmentation masks as labels (here with the same height and width as the input image), so that the network can backpropagate the pixel errors of each corresponding pixel of the

output/prediction. In this way the network will be trained to learn on a single pixel level.

If one has access to sequential data, e.g. a time-lapse recording as in this case, it might be possible to use the temporal information to make the network better learn what is e.g. a stem cell and what is not. Since the stem cells (when looking at this case) are moving and everything else is static in the sequences, the network can potentially more easy understand what is a stem cell and what is not.

13 Experiments

13.1 Experimental Setup

The two different networks, $Network_1$ (1 channel input) and $Network_5$ (5 channel input) were trained with the same setup of hyperparameters and then on a slightly different setup of hyperparameters for $Network_5$, to compare the difference. A momentum optimizer was used to minimize the cross-entropy cost function, which was weighted because of the class imbalance. Dropout was used to prevent overfitting, i.e. no regularization term was added to the cost function and no batch normalization was used since the batch size was one image. The networks were trained on a dataset of 24 997 images of size 1024x1024, where all images had been preprocessed by removing the sequential background and cropped or padded to the size 1024x1024. Each input image had a corresponding segmentation mask as label/truth. 19 998 images were used for training, 4 998 images were used for validation and 1 image was used for test/prediction. The (most relevant) experimental hyperparameter settings can be seen in *Table 1*, which was for $Network_1$ and $Network_5$, and in *Table 2*, which was only for $Network_5$. The networks were trained until the loss converged or that they started to overfit.

Hyperparameter	Value
Dropout	0.25
Learning rate	0.15 (with decay)
Decay rate	0.95
Momentum	0.2
Max-pooling	2x2, stride 2
Deconvolution	2x2, stride 2
Convolution	5x5, stride 1
Cost function weight	2 for the unrepresented class (stem cells)

Table 1 Hyperparameters for $Network_1$ and $Network_5$.

Hyperparameter	Value
Dropout	0.25
Learning rate	0.1 (with decay)
Decay rate	0.95
Momentum	0.2
Max-pooling	2x2, stride 2
Deconvolution	2x2, stride 2
Convolution	5x5, stride 1
Cost function weight	2 for the unrepresented class (stem cells)

Table 2 Hyperparameters for a second run of Network$_5$.

13.2 System Requirements

The programming language used here is python, where the library/API TensorFlow [9], developed by Google, has been used. This saves time to not having to build every small function from scratch, but also gives more free hands to create something unique compared to the usage of a high level-API like Keras [46].

The different networks are trained on a 6 core CPU and a NVidia P100 Tesla GPU.

14 Results

The results of the best performing model of the different networks and hyperparameter setups can be seen in *Table 3*. The training and validation process of the different networks with hyperparameter setup as in *Table 1* can be seen in *Figure 15-Figure 21*.

Measure	Network$_1$ (*Table 1* settings)	Network$_5$ (*Table 1* settings)	Network$_5$ (*Table 2 settings*)
Error rate	0.033	0.054	0.041
F1 score	0.83	0.70	0.82
False white	0.000157	0.000134	0.000227
WP accuracy	0.83	0.68	0.84

Table 3 The results of the different setups of Network1 and Network5. The measure False white means how many pixels that were wrongly classified as a stem cell. The measure WP (White Pixel) accuracy means how many pixels that were correctly classified as a stem cell.

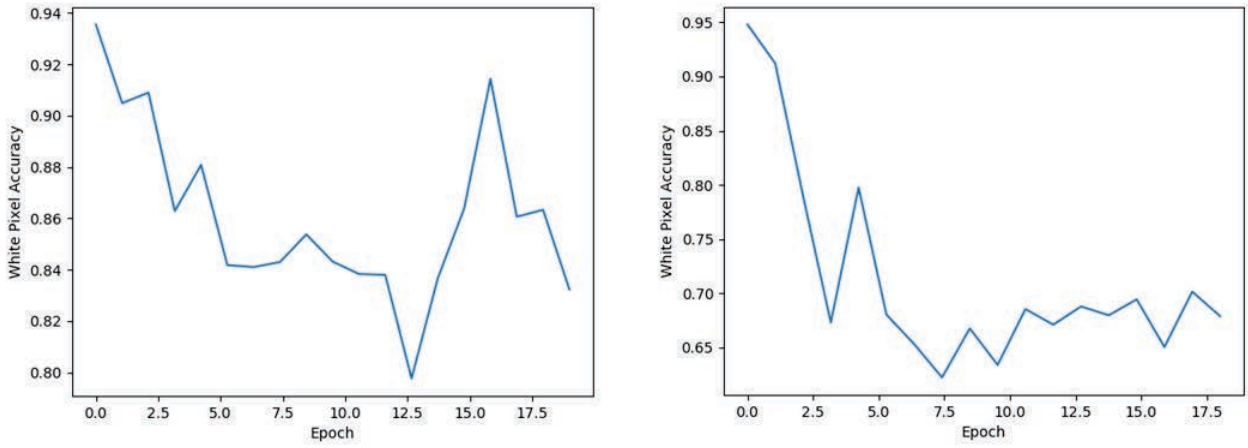

Figure 15 The plots shows how the different networks change their performance during training. The plots correspond to Network1 and Network5, respectively.

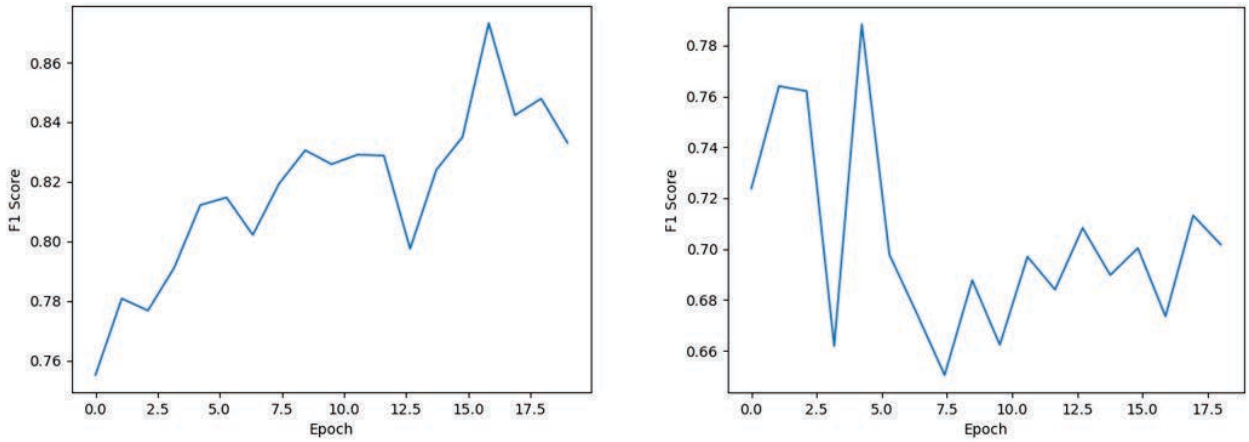

Figure 16 The plots shows how the different networks change their performance during training. The plots correspond to Network1 and Network5, respectively.

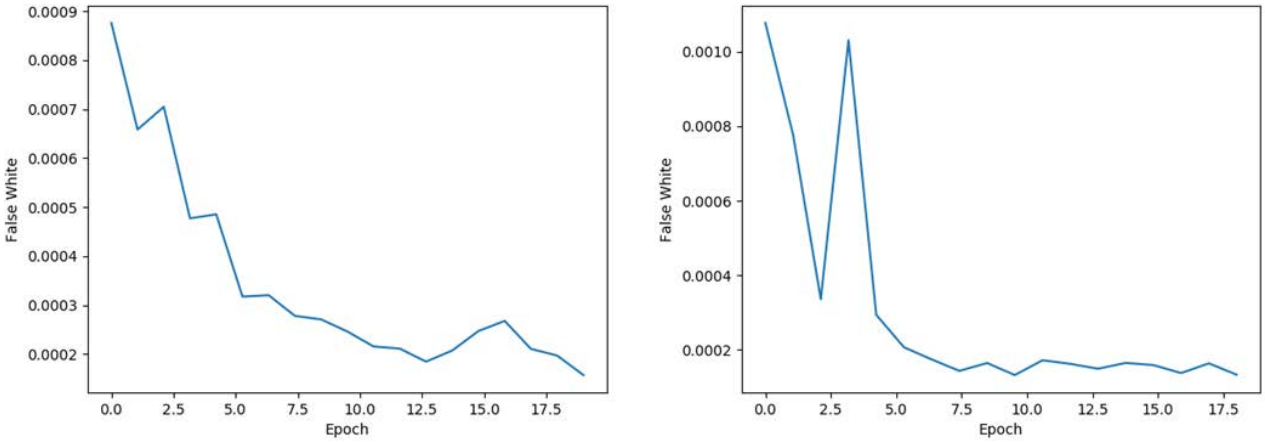

Figure 17 The plots shows how the different networks gets better at sorting out what is a stem cell and what is not during training. The plots correspond to Network1 and Network5, respectively.

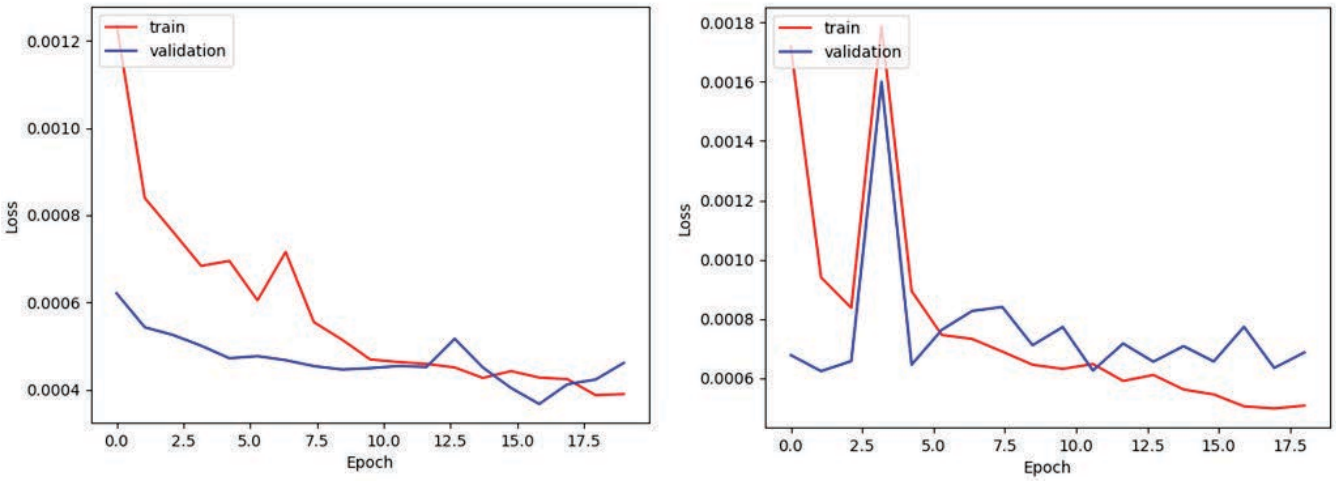

Figure 18 The plots shows how the different networks learns during training. The plots correspond to Network1 and Network5, respectively. In the plot to the right it is fluctuating and starts to overfit early. This is a sign of a too large learning rate.

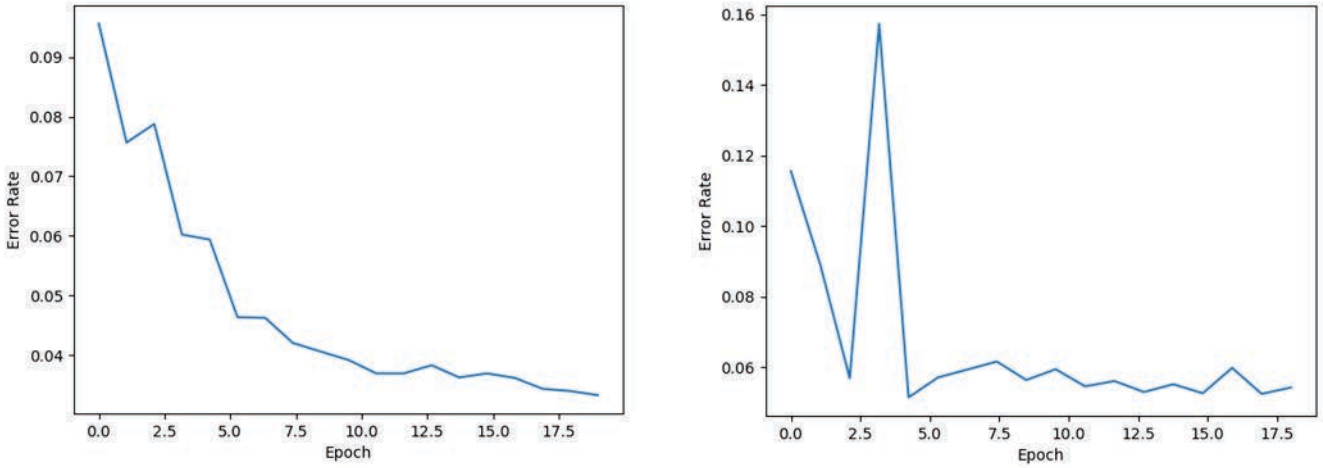

Figure 19 The plots shows how the different networks change their performance during training. The plots correspond to Network1 and Network5, respectively.

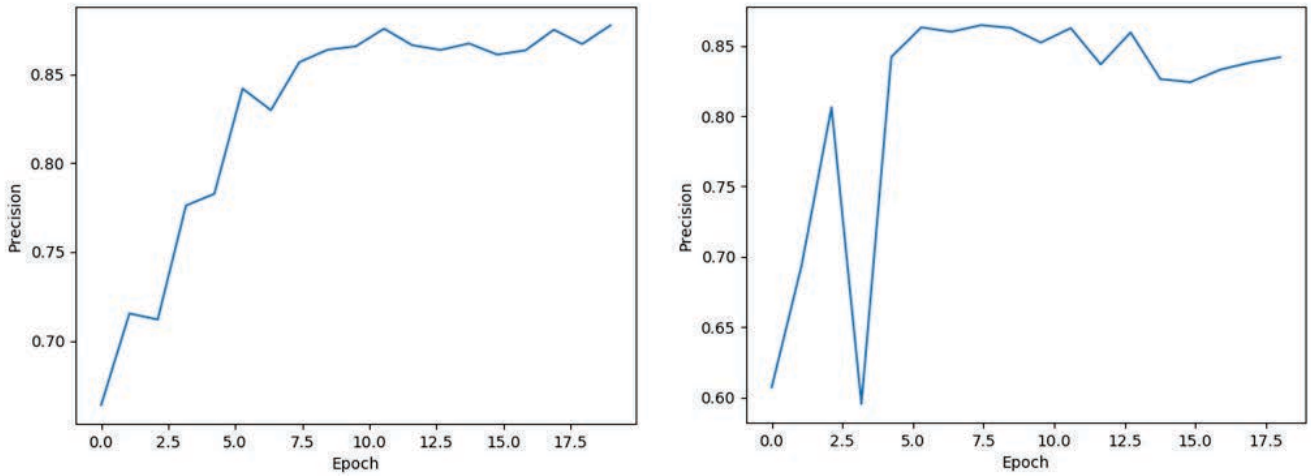

Figure 20 The plots shows how the different networks increase their precision during training. The plots correspond to Network1 and Network5, respectively.

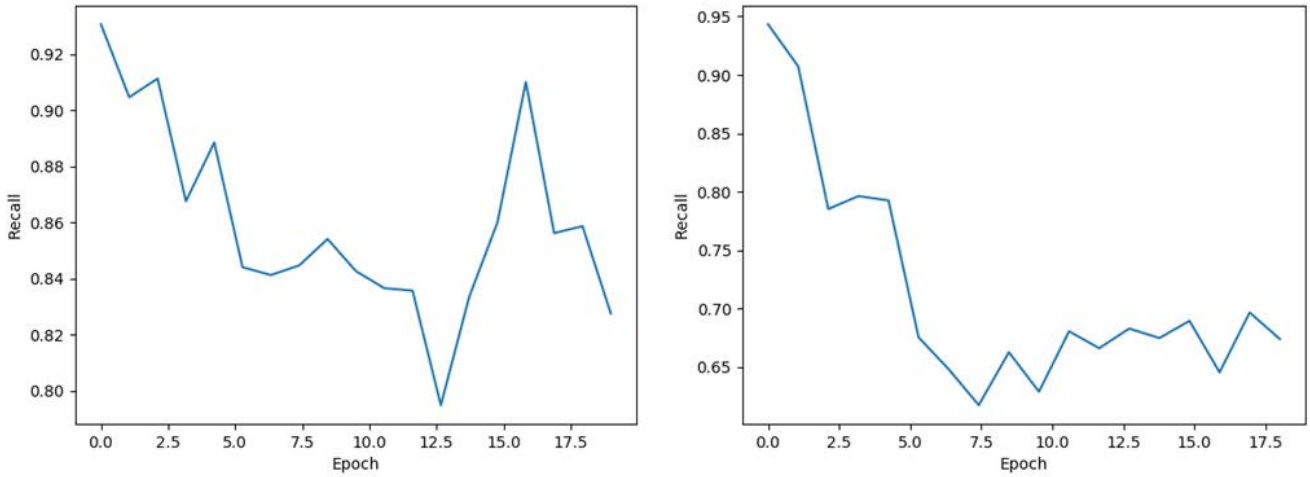

Figure 21 The plots shows how the different networks change their performance during training. The plots correspond to Network1 and Network5, respectively.

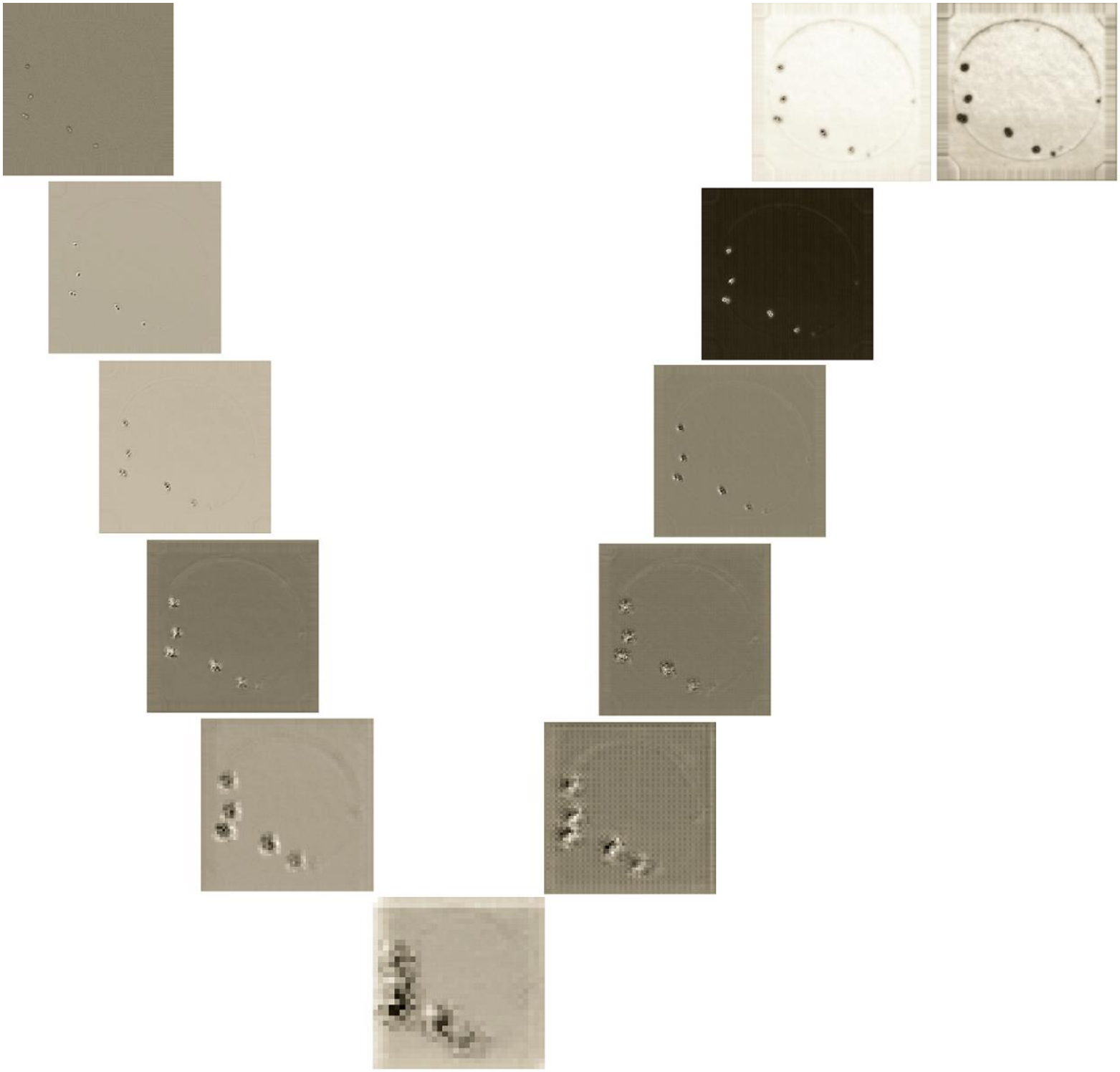

Figure 22 One feature map at each layer. The structure corresponds to the fully convolutional network seen in Figure 14. The input is a frame from the training data, in the beginning of training.

At each layer in the fully convolutional network, there are several feature maps. The number of feature maps, for each layer, is shown in *Figure* 14. In *Figure* 22 one can follow how the network is learning, by observing one (of many) feature map (as an example) for each convolutional layer.

The results of the two different networks can be seen in *Table 3*, where the F1 score is applied to the two pixel classes, foreground (MuSCs) and background.

Examples of the performance can be seen in *Figure 23*.

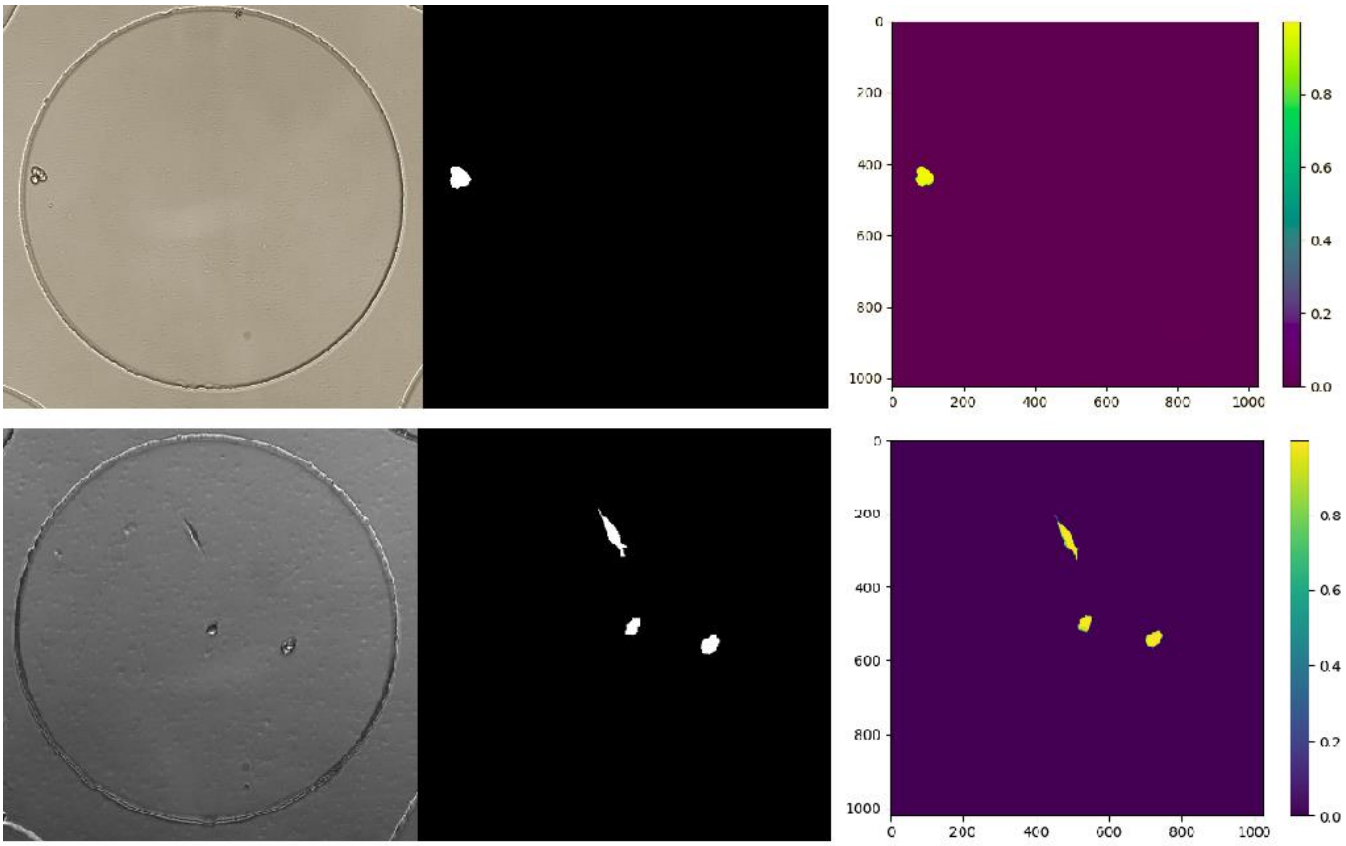

Figure 23 The achieved result as a heat map. The left images is the input (1 channel), the images in the middle is the ground truth, i.e. the segmentation mask, and the right images is the prediction of the fully convolutional network.

15 Discussion & Conclusion

The results, as can be seen in *Table 3*, implies that $Network_1$ and $Network_5$ both perform well. $Network_1$ is slightly sharper, though there are more changes in the hyperparameter settings that could/should be made for $Network_5$ since it has more complexity in its training data.

$Network_1$ learns to find the correct MuSC pixels quite fast, but it takes longer to learn that other objects in the image aren't MuSCs. $Network_5$ is the opposite, it takes more time to find the correct MuSC pixels, but it learns to sort out the other objects in an image quite fast (low *False white*-measure quite fast, see *Figure 17*).

As can be seen in *Figure* 18, both networks are stopped training when converging or overfitting. $Network_5$ shows early overfitting and a lot of fluctuation that indicates on a too high learning rate. The steps of the weight and bias parameter updates are too large, hence the huge spike in the right plot in *Figure 18*. $Network_5$ increased its

performance when trained with a lower learning rate, but it still performs worse than $Network_1$.

To increase the performance of $Network_5$, one could try to:

- Make the network deeper, i.e. add more convolutional layers, since it has more complexity in its training data compared to $Network_1$
- Increase the number of images in the training data to make it more general and to avoid overfitting (this should also improve $Network_1$)
- Increase the weight of the cost function
- Increase the dropout rate to avoid overfitting
- Add a regularizer term to the cost function to avoid overfitting

From *Figure 15, 17* and *21*, one can see that in the beginning both of the networks have a high accuracy on finding the MuSC pixels. Though, the networks also think that other features in the input frames are MuSCs. When it learns that not all the objects in the image are MuSCs, it is harder for the networks to be as accurate as before, but both of the networks still performs well.

In *Figure 22* one can follow the process of the whole fully convolutional network. From input size 1024x1024 to 32x32 and back to 1024x1024. Even though the exact position of the spatial information is discarded in the downsampling, it keeps the relative positions and therefore finds the exact position of the spatial information in the end. A lot thanks to passing over the information from the downsampling to the upsampling.

The temporal approach in training would be interesting to use on data where it is very hard for the network to learn to find the wanted objects otherwise. In this case, the performance of $Network_1$ is good and it can separate MuSCs from other features/objects in an image, as can be seen in the examples of its performance in *Figure 23*. This means that giving the network a short-term memory will not help the network here to perform better, since the way to improve $Network_1$ is a matter of increasing accuracy of classifying the very most outer edge pixels of a MuSC, and not overall if it can, very roughly, segment something as a MuSC correctly or not. If that was the case, then the temporal approach would most probably increase the performance. In this case, $Network_1$ performs well and it is unnecessary to make the network more complex.

16 Acknowledgements

I want to thank my supervisor Joakim Jaldén and my friend and business partner Robert Lundberg for their helpful discussions.

17 References

[1] "Wikipedia," 27 05 2017. [Online]. Available: https://en.wikipedia.org/wiki/HeLa. [Accessed 07 06 2017].

[2] K. Magnusson, "Segmentation and tracking of cells and particles in time-lapse microscopy," School of Electrical Engineering, KTH Royal Institute of Technology, Stockholm, 2016.

[3] P. F. a. T. B. Olaf Ronneberger, "U-Net: Convolutional Networks for Biomedical Image Segmentation," Computer Science Department and BIOSS Centre for Biological Signalling Studies, University of Freiburg, Freiburg, 2015.

[4] S. D. H. Z. K. S. O. V. A. G. N. K. A. S. a. K. K. Aäron van den Oord, "WAVENET: A GENERATIVE MODEL FOR RAW AUDIO," Google DeepMind, London, 2016.

[5] L. M. a. S. C. Alec Radford, "UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS," in *ICLR 2016*, 2016.

[6] "ieee," 2017. [Online]. Available: https://www.ieee.org/conferences_events/conferences/conferencedetails/index.html?Conf_ID=17944. [Accessed 07 06 2017].

[7] "Caffe," 14 04 2017. [Online]. Available: http://caffe.berkeleyvision.org/doxygen/annotated.html. [Accessed 07 06 2017].

[8] C. C. A. L. a. A. R. Joël Akeret, "Radio frequency interference mitigation using deep convolutional neural networks," Department of Physics and Department of Computer Science, ETH Zürich, Zürich, 2017.

[9] Google, "Tensorflow," Google, 2017. [Online]. Available: https://www.tensorflow.org/. [Accessed 07 06 2017].

[10] S. H. a. B. H. Hyeonwoo Noh, "Learning Deconvolution Network for Semantic Segmentation," Department of Coputer Science and Engineering, POSTECH, Korea, 2015.

[11] "Visual Geometry Group," 08 10 2014. [Online]. Available: http://www.robots.ox.ac.uk/~vgg/research/very_deep/. [Accessed 07 06 2017].

[12] A. K. a. R. C. Vijay Badrinarayanan, "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation," IEEE, 2016.

[13] M. Nielsen, Neural Networks and Deep Learning, Determination Press, 2015.

[14] "Neural Networks," 2000. [Online]. Available: http://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html. [Accessed 07 06 2017].

[15] "CS231n Convolutional Neural Networks for Visual Recognition," Stanford, 2017. [Online]. Available: http://cs231n.github.io/neural-networks-1/. [Accessed 07 06 2017].

[16] R. Rojas, Neural Networks - A Systematic Introduction, New York: Springer Verlag, 1996.

[17] C. M. Bishop, Neural Networks for Pattern REcognition, New York, NY: Oxford University Press, 1995.

[18] Y. B. a. A. C. Ian Goodfellow, Deep Learning, MIT Press, 2016.

[19] J. L. a. T. D. Evan Shelhamer, "Fully Convolutional Network for Semantic Segmentation," Department of Electrical Engineering and Computer Science, UC Berkley, Berkley, 2016.

[20] D. K. G. W. T. a. R. F. Matthew D. Zeiler, "Deconvolutional Networks," Department of Computer Science, New York University, New York, 2010.

[21] M. D. Z. a. R. Fergus, "Visualizing and Understanding Convolutional Networks," Department of Computer Science, New York University, New York, 2013.

[22] G. W. T. a. R. F. Matthew D. Zeiler, "Adaptive Convolutional Networks for Mid and High Level Feature Learning," Department of Computer Science, New York University, New York, 2011.

[23] J. C. L. T. F. H. A. A. A. T. J. T. C. L. a. Z. W. Wenzhe Shi, "Is the deconvolution layer the same as a convolutional layer?," Twitter, 2016.

[24] V. D. a. F. Visin, "A guide to convolutional arithmetic for deep learning," MILA, Université de Montréal and AIRLab, Politecnico di Milano, 2016.

[25] D. Batra, "ECE 6540 Deep Learning for Perception," 2015. [Online]. Available: https://computing.ece.vt.edu/~f15ece6504/. [Accessed 07 06 2017].

[26] "KDnuggets," 2016. [Online]. Available: http://www.kdnuggets.com/2016/03/must-know-tips-deep-learning-part-1.html. [Accessed 07 06 2017].

[27] ”Wikipedia,” 11 05 2017. [Online]. Available: https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient. [Använd 07 06 2017].

[28] T. Amaral, "Using Different Cost Functions to Train Deep Networks with Supervision," Instituto de Engenharia Biomédica, Portugal, 2013.

[29] R. L. a. R. Collobert, "Word Embeddings through Hellinger PCA," Idiap Research Institute, Martigny, Switzerland, 2017.

[30] ”Wikipedia,” 22 05 2017. [Online]. Available: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence. [Använd 18 06 2017].

[31] T. Yamano, ”A generalization of the Kullback-Leibler divergence and its properties,” Department of Physics, Ochanomizu University, Tokyo, Japan, 2009.

[32] ”Wikipedia,” 09 06 2017. [Online]. Available: https://en.wikipedia.org/wiki/Itakura%E2%80%93Saito_distance. [Använd 18 06 2017].

[33] D. P. K. a. J. L. Ba, "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION," in *ICLR 2015*, 2017.

[34] ”climin.readthedocs,” climin developers, 2013. [Online]. Available: http://climin.readthedocs.io/en/latest/rmsprop.html. [Använd 18 06 2017].

[35] E. H. a. Y. S. John Duchi, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research,* vol. 12, pp. 2121-2159, 2011.

[36] G. H. D. S. M. Y. D. E. J. G. L. N. T. P. E. D. D. G. S. C. D. L. M. W. A. M. H. T. B. a. J. K. H. Brendan McMahan, ”Ad Click Prediction: a View from the Trenches,” Google, Inc., Chicago, Illinois, USA, 2013.

[37] M. D. Zeiler, ”ADADELTA: AN ADAPTIVE LEARNING RATE METHOD,” Google Inc., New York University, USA, 2012.

[38] J. D. a. Y. Singer, ”Efficient Learning using Forward-Backward Splitting,” Google, University of California, USA, 2009.

[39] G. H. A. K. I. S. a. R. S. Nitish Srivastava, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research,* vol. 15, pp. 1929-1958, 2014.

[40] S. I. a. C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," Google, 2015.

[41] ”Wikipedia,” 31 05 2017. [Online]. Available: https://en.wikipedia.org/wiki/Precision_and_recall. [Använd 07 06 2017].

[42] M. K. a. S. Matwin, ”Adressing the Curse of Imbalanced Training Sets: One-Sided Selection,” Department of Computer Science, University of Ottawa, Ottawa, Ontario, Canada, 1997.

[43] "Machine Learning Mastery," 2017. [Online]. Available: http://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/. [Accessed 07 06 2017].

[44] D. E. a. R. Fergus, "Predicting Depth, Surface Normals and Semantic Labels with a Common Multi-Scale Convolutional Archtitecture," Department of Computer Science, New York University and Facebook AI Research, New York, 2015.

[45] ”Wikipedia,” 18 01 2017. [Online]. Available: accuracy paradox. [Använd 07 06 2017].

[46] "Keras Documentation," 2017. [Online]. Available: https://keras.io/. [Accessed 07 06 2017].

18 Appendix

18.1 Classes & Functions

18.1.1 data_handler.py

```
from __future__ import print_function, division, absolute_import, unicode_literals

import numpy as np
import tensorflow as tf
from PIL import Image
import glob
import sys
import random
import os.path
import re

numbers = re.compile(r'(\d+)')


def numericalSort(value):
    parts = numbers.split(value)
    parts[1::2] = map(int, parts[1::2])
    return parts

def read_data(search_path_frame, search_path_mask):
    # Subract 127 from each frame. The size is 1024x1024
    all_frames = sorted(glob.glob(search_path_frame), key=numericalSort) # The path names
    all_masks = sorted(glob.glob(search_path_mask), key=numericalSort) # The path names
    data_name = []
    if len(all_frames) == len(all_masks):
        for idx in range(len(all_frames)):
            frame_name = all_frames[idx]
            mask_name = all_masks[idx]
            data_name.append((frame_name, mask_name))
    else:
        sys.exit("Different number of frames compared to number of masks")
    return data_name

def split_data(data_name, test_size, pred_size=None):
    # test_size and pres_size a number between 0 and 1
    #if (1 - test_size - pred_size) == 0 or (1 - test_size - pred_size) < 0 or (1 - test_size - pred_si
ze) > 1:
    #    sys.exit("Wrong sizes of the data-splits")
    data_len = len(data_name)
    if pred_size is not None:
        train_data_name = data_name[0:round((1 - test_size)*(data_len-1))]
        test_data_name = data_name[round((1 - test_size )*(data_len-1)) + 1:-2]
        pred_data_name = data_name[-1]
        # shuffle the data
        random.shuffle(train_data_name)
        random.shuffle(test_data_name)
        return train_data_name, test_data_name, pred_data_name
    else:
        train_data_name = data_name[0:round((1 - test_size)*data_len)]
        test_data_name = data_name[round((1 - test_size)*data_len) + 1:-1]
        # shuffle the data
        random.shuffle(train_data_name)
        random.shuffle(test_data_name)
        return train_data_name, test_data_name

def data_provider(data_name, batch_size, idx, epoch, channels, n_class, shuffle, pred):
    # If idx == 0, shuffle data_name
    if idx == 0 and epoch != 0 and shuffle == True:
        random.shuffle(data_name)
```

```
    # Get the data and give the batch as output
    if batch_size == 1:
        if channels == 5:
            if pred is True:
                frame = np.zeros((1024, 1024, 5), dtype=np.float32)
                frame[:,:,0] = np.array(Image.open(data_name[0]), np.float32) - 127
                frame[:,:,1] = np.array(Image.open(data_name[0]), np.float32) - 127
                frame[:,:,2] = np.array(Image.open(data_name[0]), np.float32) - 127
                frame[:,:,3] = np.array(Image.open(data_name[0]), np.float32) - 127
                frame[:,:,4] = np.array(Image.open(data_name[0]), np.float32) - 127
                mask = np.array(Image.open(data_name[1]), np.bool)
            else:
                frame_name3 = data_name[idx][0]
                frame_name_parts = frame_name3.split('_')

                frame_nr1 = str(int(frame_name_parts[2].split('.')[0]) - 2)
                frame_nr2 = str(int(frame_name_parts[2].split('.')[0]) - 1)
                frame_nr4 = str(int(frame_name_parts[2].split('.')[0]) + 1)
                frame_nr5 = str(int(frame_name_parts[2].split('.')[0]) + 2)

                if os.path.exists(frame_nr1):
                    frame_name1 = frame_name_parts[0] + frame_name_parts[1] + frame_nr1
                    frame_name2 = frame_name_parts[0] + frame_name_parts[1] + frame_nr2
                else:
                    frame_name1 = frame_name3
                    frame_name2 = frame_name3
                if os.path.exists(frame_nr5):
                    frame_name4 = frame_name_parts[0] + frame_name_parts[1] + frame_nr4
                    frame_name5 = frame_name_parts[0] + frame_name_parts[1] + frame_nr5
                else:
                    frame_name4 = frame_name3
                    frame_name5 = frame_name3

                frame = np.zeros((1024, 1024, 5), dtype=np.float32)
                frame[:,:,0] = np.array(Image.open(frame_name1), np.float32) - 127
                frame[:,:,1] = np.array(Image.open(frame_name2), np.float32) - 127
                frame[:,:,2] = np.array(Image.open(frame_name3), np.float32) - 127
                frame[:,:,3] = np.array(Image.open(frame_name4), np.float32) - 127
                frame[:,:,4] = np.array(Image.open(frame_name5), np.float32) - 127
                mask = np.array(Image.open(data_name[idx][1]), np.bool)
        else:
                    if pred is True:
                        frame = np.array(Image.open(data_name[0]), np.float32) - 127
                        mask = np.array(Image.open(data_name[1]), np.bool)
                    else:
                        frame = np.array(Image.open(data_name[idx][0]), np.float32) - 127
                        mask = np.array(Image.open(data_name[idx][1]), np.bool)
            else:
                frame = []
                mask = []
                if idx != 0:
                    idx = idx + batch_size
                for i in range(batch_size):
                    frame.append(np.array(Image.open(data_name[idx + i][0]), np.float32) - 127)
                    mask.append(np.array(Image.open(data_name[idx + i][1]), np.bool))
            batch_x = frame
            nx = batch_x.shape[1]
            ny = batch_x.shape[0]
            # The placeholder takes 2 channels in as n_class, therefore, change the mask
            masks = np.zeros((ny, nx, n_class), dtype=np.float32)
            masks[..., 1] = mask
            masks[..., 0] = ~mask
            batch_y = masks
            if channels != 5:
                return batch_x.reshape(1, ny, nx, 1), batch_y.reshape(1, ny, nx, n_class), mask
            else:
```

```
125.                 return batch_x.reshape(1, ny, nx, 5), batch_y.reshape(1, ny, nx, n_class), mask
```

18.1.2 preprocess.py

```
1.  import numpy as np
2.  import tensorflow as tf
3.  from PIL import Image
4.  import statistics
5.
6.  def get_freq_median(filenames_training_set):
7.      freq_med_vec = []
8.      max_white = 0
9.      max_black = 0
10.     for i in range(len(filenames_training_set)):
11.         mask = np.array(Image.open(filenames_training_set[i][1]), np.bool)
12.         n_white = mask.sum(axis=1).sum(axis=0)
13.         n_black = (1024*1024) - n_white
14.         freq_med_vec.append(n_white)
15.         freq_med_vec.append(n_black)
16.         max_white += n_white
17.         max_black += n_black
18.     freq_med = np.median(np.array(freq_med_vec))
19.     weight_ones = freq_med/max_white
20.     weight_zeros = freq_med/max_black
21.     return weight_ones, weight_zeros
22.
23. def get_weight(filenames_training_set):
24.     max_white = 0
25.     max_black = 0
26.     for i in range(len(filenames_training_set)):
27.         mask = np.array(Image.open(filenames_training_set[i][1]), np.bool)
28.         n_white = mask.sum(axis=1).sum(axis=0)
29.         n_black = (1024*1024) - n_white
30.         max_white += n_white
31.         max_black += n_black
32.     weight_ones = max_black/max_white
33.     return weight_ones
```

18.1.3 wrappers.py

```
1.  from __future__ import print_function, division, absolute_import, unicode_literals
2.
3.  import tensorflow as tf
4.
5.  def weight_variable(shape, stddev=0.1):
6.      initial = tf.truncated_normal(shape, stddev=stddev)
7.      return tf.Variable(initial)
8.
9.  def weight_variable_deconv(shape, stddev=0.1):
10.     return tf.Variable(tf.truncated_normal(shape, stddev=stddev))
11.
12. def bias_variable(shape):
13.     initial = tf.constant(0.1, shape=shape)
14.     return tf.Variable(initial)
15.
16. def conv2d(x, W, keep_prob_):
17.     conv_2d = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
18.     return tf.nn.dropout(conv_2d, keep_prob_)
19.
20. def deconv2d(x, W, stride):
21.     x_shape = tf.shape(x)
22.     output_shape = tf.pack([x_shape[0], x_shape[1]*2, x_shape[2]*2, x_shape[3]//2]) # Reduce the number
    of channels by half for each deconvolution layer
23.     return tf.nn.conv2d_transpose(x, W, output_shape, strides=[1, stride, stride, 1], padding='SAME')
```

```
24.
25. def max_pool(x, area):
26.     return tf.nn.max_pool(x, ksize=[1, area, area, 1], strides=[1, area, area, 1], padding='SAME')
27.
28. def concat(x1, x2):
29.     return tf.concat(3, [x1, x2])
30.
31. def softmax_per_pixel(output):
32.     exponential_map = tf.exp(output)
33.     sum_exp = tf.reduce_sum(exponential_map, 3, keep_dims = True)
34.     tensor_sum_exp = tf.tile(sum_exp, tf.pack([1, 1, 1, tf.shape(output)[3]]))
35.     return tf.div(exponential_map, tensor_sum_exp, name="predicter")
```

18.1.4 stemnet.py

```
1.  from __future__ import print_function, division, absolute_import, unicode_literals
2.
3.  import tensorflow as tf
4.  import numpy as np
5.  from collections import OrderedDict
6.
7.  from wrappers import weight_variable, weight_variable_deconv, bias_variable, conv2d, deconv2d, max_pool
    , concat, softmax_per_pixel
8.  from data_handler import read_data, split_data
9.
10. class Stemnet:
11.     """
12.     A fully convolutional network to segment stemcells
13.
14.     :param channels: number of channels in the input image
15.     :param n_class: number of classes
16.     :param test_size: how big part of the dataset to use for validation
17.     :param data_size: how many images to use in the dataset
18.     :param summaries: True for using TensorBoard, otherwise False
19.     :param cost: name of the cost function (default is 'cross_entropy')
20.     :param class_weights: how much to weight the underrepresented class
21.     """
22.
23.     def __init__(self, channels=1, n_class=2, test_size = 0.05, data_size=-
    1, summaries=False, cost="weighted_cost", class_weights=2):
24.
25.         # Input data
26.         data_name = read_data('/media/misakss/DATAPART1/Frames_more/*.png', '/media/misakss/DATAPART1/M
    asks_more/*.png')
27.         self.train_data_name, self.test_data_name, self.pred_data_name = split_data(data_name[0:data_si
    ze], test_size, pred_size=True)
28.         self.n_training_data = len(self.train_data_name) # as many number of training-images
29.         self.n_test_data = len(self.test_data_name) # as many number of test-images
30.         self.class_weights = class_weights
31.         #class_weights = get_weight(train_data_name) # change default in _get_cost if size of training
    set changes
32.
33.         self.channels = channels
34.         self.n_class = n_class
35.         self.summaries = summaries
36.
37.         self.x = tf.placeholder("float", shape=[None, None, None, channels], name="x")
38.         self.y = tf.placeholder("float", shape=[None, None, None, n_class], name="y")
39.         self.keep_prob = tf.placeholder(tf.float32, name="keep_prob") #dropout
40.
41.         self.network_output = self.fully_convolutional_network()
42.
43.         self.cost = self._get_cost(cost)
44.
45.         self.predicter = softmax_per_pixel(self.network_output)
```

```
46.         self.correct_pred = tf.equal(tf.argmax(self.predicter, 3), tf.argmax(self.y, 3))
47.         self.accuracy = tf.reduce_mean(tf.cast(self.correct_pred, tf.float32))
48.
49.     def _get_cost(self, cost_name):
50.         """
51.         The cost function, either cross_entropy, weighted cross_entropy or dice_coefficient.
52.         """
53.
54.         flat_logits = tf.reshape(self.network_output, [-1, self.n_class])
55.         flat_labels = tf.reshape(self.y, [-1, self.n_class])
56.         if cost_name == "cross_entropy":
57.             loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=flat_labels, logits=fl
    at_logits))
58.
59.         elif cost_name == 'weighted_cost':
60.             eps = 1e-5
61.
62.             class_weights = tf.constant(self.class_weights, dtype=tf.float32)
63.
64.             y_argmax = tf.to_float(tf.reshape(tf.argmax(self.y, 3), [-1]))
65.
66.             weight_pos_tmp = tf.multiply(class_weights, y_argmax)
67.             weight_pos = tf.add(weight_pos_tmp, tf.to_float(tf.ones(tf.shape(weight_pos_tmp))))
68.
69.             cost_cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=flat_labels, logits=fla
    t_logits)
70.
71.             weighted_cost = tf.multiply(weight_pos, cost_cross_entropy)
72.             loss = tf.reduce_mean(weighted_cost)
73.
74.             #loss = tf.div(cost, tf.constant(10, dtype=tf.float32))
75.             #loss = tf.clip_by_value(cost, eps, 1.0-eps)
76.
77.         elif cost_name == "dice_coefficient":
78.             eps = 1e-5
79.
80.             prediction = softmax_per_pixel(self.network_output)
81.             intersection = tf.reduce_sum(tf.mul(prediction, self.y))
82.             union =  eps + tf.reduce_sum(tf.mul(prediction, prediction)) + tf.reduce_sum(tf.mul(self.y,
     self.y))
83.             cost_tmp = (2 * intersection/ (union))
84.             cost_clip = tf.clip_by_value(cost_tmp, eps, 1.0-eps)
85.             loss = 1 - cost_clip
86.
87.         else:
88.             raise ValueError("Unknown cost function: "%cost_name)
89.
90.         return loss
91.
92.     def fully_convolutional_network(self):
93.
94.         #----------------------- Definition and Initialization -----------------------#
95.
96.         # Data variables
97.         dtype = tf.float32
98.
99.         # Placeholder for the input image
100.                 nx = tf.shape(self.x)[1]
101.                 ny = tf.shape(self.x)[2]
102.
103.                 # To apply layer, reshape input to 4D tensor
104.                 x_image = tf.reshape(self.x, tf.pack([-
    1,nx,ny,self.channels])) # [batch, width, height, channels]
105.                 in_layer = x_image
106.
107.                 # Variables
```

```
                layers = 7 # number of layers in each sampling direction
                patch_size = 5 # the size of the patch in each convolutional layer
                features_init = 16 # number of feature maps from the first layer
                area = 2 # for pooling, use a 2x2 area, which reduces the size by half
                stride = 2 # for deconvolution

                # Store convolutional output for concatenation and remember the order
                convs = []
                pools = OrderedDict()
                deconvs = OrderedDict()
                dw_h_convs = OrderedDict()
                up_h_convs = OrderedDict()

                # Strings
                l_name = 'Layer'

                #---------------------------------- Downsampling ---------------------------------#

                for layer in range(0, layers):
                    print('Layer: ', layer)
                    # How many feature maps per layer
                    features_out = 2**layer*features_init
                    stddev = np.sqrt(2 / (patch_size**2 * features_out))

                    # Weights and bias
                    if layer == 0:
                        weight = weight_variable([patch_size, patch_size, self.channels, features_out], stddev)
                    else:
                        weight = weight_variable([patch_size, patch_size, features_in, features_out], stddev)

                    bias = bias_variable([features_out])

                    # Convolutional layer
                    conv = conv2d(in_layer, weight, self.keep_prob)

                    # Apply ReLU
                    h_conv = tf.nn.relu(conv + bias)
                    dw_h_convs[layer] = h_conv

                    if layer < layers - 1:
                        # Pooling layer
                        pools[layer] = max_pool(h_conv, area)
                        in_layer = pools[layer]

                    # Update
                    features_in = features_out
                    convs.append(conv)

                #---------------------------------- Bridge ---------------------------------#

                # The input to the first deconvolutional layer
                in_layer = dw_h_convs[layers-1]

                #---------------------------------- Upsampling ---------------------------------#

                for layer in range(layers-2, -1, -1):
                    print('Layer: ', layer)
                    # How many feature maps per layer (layer + 1 since we count/loop backwards)
                    features_in = 2**(layer + 1)*features_init
                    stddev = np.sqrt(2 / (patch_size**2 * features_in))
                    features_out = features_in//2

                    # Weights and bias deconv
```

```
                    weight_d = weight_variable_deconv([stride, stride, features_out, features_in], stddev)

                    bias_d = bias_variable([features_out])

                    # Unpooling layer
                    deconv = deconv2d(in_layer, weight_d, stride)

                    # Apply ReLU
                    h_deconv = tf.nn.relu(deconv + bias_d)

                    # Concatenate deconvolutional layer output with corresponding downsampling output
                    h_deconv_concat = concat(dw_h_convs[layer], h_deconv)

                    # Weights and bias conv
                    weight = weight_variable([patch_size, patch_size, features_in, features_out], stddev) # switched places of features_in and features_out

                    bias = bias_variable([features_out]) # switched places of features_in and features_out

                    # Convolutional layer
                    conv = conv2d(h_deconv_concat, weight, self.keep_prob)

                    # Apply ReLU
                    h_conv = tf.nn.relu(conv + bias)
                    in_layer = h_conv

                    # Update
                    convs.append(conv)
                    deconvs[layer] = h_deconv_concat
                    up_h_convs[layer] = in_layer

                #---------------------------------- Output ----------------------------------#

                # Update
                patch_size = 1

                # Weights and bias
                weight = weight_variable([patch_size, patch_size, features_init, self.n_class])
                bias = bias_variable([self.n_class])

                # Convolutional layer
                conv = conv2d(in_layer, weight, tf.constant(1.0)) # No dropout here

                # Apply ReLU
                output = tf.nn.relu(conv + bias, name="output")

                # Update
                up_h_convs["out"] = output
                convs.append(conv)

                print('Output done')

                if self.summaries is True:

                    #---------------------------------- Summaries ----------------------------------#

                    for i, conv in enumerate(convs):
                        tf.summary.image('summary_conv_%02d'%i, self.image_summary(conv))

                    for k in pools.keys():
                        tf.summary.image('summary_pool_%02d'%k, self.image_summary(pools[k]))

                    for k in deconvs.keys():
```

```
                    tf.summary.image('summary_deconv_concat_%02d'%k, self.image_summary(deconvs[k])
    )

                for k in dw_h_convs.keys():
                    tf.summary.histogram("dw_convolution_%02d"%k + '/activations', dw_h_convs[k])

                for k in up_h_convs.keys():
                    tf.summary.histogram("up_convolution_%s"%k + '/activations', up_h_convs[k])

            return output

        def image_summary(self, img, idx=0):

            # Make an image summary for 4d tensor image with index idx

            V = tf.slice(img, (0, 0, 0, idx), (1, -1, -1, 1))
            V -= tf.reduce_min(V)
            V /= tf.reduce_max(V)
            V *= 255

            img_w = tf.shape(img)[1]
            img_h = tf.shape(img)[2]
            V = tf.reshape(V, tf.pack((img_w, img_h, 1)))
            V = tf.transpose(V, (2, 0, 1))
            V = tf.reshape(V, tf.pack((-1, img_w, img_h, 1)))

            return V
```

18.1.5 train_and_eval.py

```
from __future__ import print_function, division, absolute_import, unicode_literals

import tensorflow as tf
import numpy as np
import tkinter
import matplotlib.pyplot as plt

from wrappers import weight_variable, weight_variable_deconv, bias_variable, conv2d, deconv2d, max_pool
, concat, softmax_per_pixel
from data_handler import data_provider

class Train_and_Eval(object):
    """
    Trains and evaluate Stemnet. Uses final model for prediction.

    :param net: the stemnet instance to train
    :param batch_size: size of training batch
    :param optimizer: name of the optimizer to use (momentum or adam)
    :param learning_rate: the learning rate
    """

    def __init__(self, net, batch_size=1, opt="momentum", learning_rate=0.1):
        self.net = net
        self.batch_size = batch_size
        self.opt = opt
        self.learning_rate = learning_rate
        self.optimizer = self._get_optimizer(self.net.n_training_data)

    def _get_optimizer(self, training_iters):
        if self.opt == "momentum":
            global_step = tf.Variable(0)
            start_learning_rate = self.learning_rate
            momentum = 0.2
            decay_rate = 0.95

```

```
35.             self.learning_rate_momentum = tf.train.exponential_decay(learning_rate=start_learning_rate,
     global_step=global_step, decay_steps=training_iters, decay_rate=decay_rate, staircase=True)
36.
37.             optimizer = tf.train.MomentumOptimizer(learning_rate=self.learning_rate_momentum, momentum=
    momentum).minimize(self.net.cost, global_step=global_step)
38.
39.         elif self.opt == "adam":
40.             learning_rate = self.learning_rate #0.001
41.             self.learning_rate_adam = tf.Variable(learning_rate)
42.
43.             optimizer = tf.train.AdamOptimizer(learning_rate=self.learning_rate_adam).minimize(self.net
    .cost, global_step=global_step)
44.
45.         return optimizer
46.
47.     def train_and_evaluation(self, test_batch_size = 1, epochs=20, dropout=0.75, display=20, save_model
    _from_epoch=19, save_model_every=2):
48.         """
49.         Training and evaluation process
50.
51.         :param test_batch_size: number of images in a batch for validation
52.         :param epochs: number of epochs
53.         :param dropout: dropout rate (the keep probability)
54.         :param display: steps where display stats
55.         :param save_model_from_epoch: how many epochs to wait before saving models
56.         :param save_model_every: steps where it saves a model
57.         """
58.
59.         training_iters = self.net.n_training_data
60.         test_iters = self.net.n_test_data
61.
62.         init = tf.global_variables_initializer()
63.
64.         t_p, f_p, t_n, f_n = self.evaluation() # (network_output, y_)
65.
66.         # Initializing for results
67.         l_train = []
68.         l_test = []
69.         a_train = []
70.         a_test = []
71.         prec = []
72.         rec = []
73.         score = []
74.         error = []
75.         train_white_acc_list = []
76.         test_white_acc_list = []
77.         false_white_list = []
78.
79.         # Save model
80.         saver = tf.train.Saver()
81.
82.         with tf.Session() as sess:
83.             sess.run(init)
84.
85.             if self.net.summaries is True:
86.                 # Merge all the summaries and write them out to /home/misakss/Desktop/Project
87.                 merged = tf.summary.merge_all()
88.                 train_writer = tf.summary.FileWriter('train', sess.graph)
89.                 test_writer = tf.summary.FileWriter('test')
90.
91.             for epoch in range(epochs):
92.                 print('Epoch: ', epoch)
93.                 total_loss = 0
94.                 total_test_loss = 0
95.                 total_acc = 0
96.                 total_F1 = 0
```

```
97.                 total_calc_accuracy = 0
98.                 total_prec = 0
99.                 total_rec = 0
100.                        total_err = 0
101.                        total_test_white_acc = 0
102.                        total_n_false_white = 0
103.
104.                        for step in range(training_iters):
105.                            batch_x, batch_y, train_mask = data_provider(self.net.train_data_name, self
    .batch_size, step, epoch, self.net.channels, self.net.n_class, shuffle=True, pred=False)
106.
107.                            # Run optimization (backprop)
108.                            _ = sess.run(self.optimizer, feed_dict = {self.net.x: batch_x, self.net.y:
    batch_y, self.net.keep_prob: dropout})
109.
110.                            if step % display == 0:
111.                                print('Training-step: ', step)
112.                                if self.net.summaries is True:
113.                                    summary, train_acc, loss, train_pred_arg = sess.run([merged, self.n
    et.accuracy, self.net.cost, tf.argmax(self.net.predicter, 3)], feed_dict = {self.net.x: batch_x, self.n
    et.y: batch_y, self.net.keep_prob: 1.0})
114.                                    train_writer.add_summary(summary, step)
115.                                else:
116.                                    train_acc, loss, train_pred_arg = sess.run([self.net.accuracy, self
    .net.cost, tf.argmax(self.net.predicter, 3)], feed_dict = {self.net.x: batch_x, self.net.y: batch_y, se
    lf.net.keep_prob: 1.0})
117.                                total_loss += loss
118.                                if step % 100 == 0:
119.                                    print(loss)
120.
121.                        l_train.append(total_loss/(training_iters/display))
122.                        a_train.append(train_acc)
123.
124.                        if (epoch % save_model_every == 0) and (epoch > save_model_from_epoch):
125.                            saver.save(sess, "Model/model.ckpt", global_step=epoch)
126.
127.                        for step in range(test_iters):
128.                            test_x, test_y, test_mask = data_provider(self.net.test_data_name, test_bat
    ch_size, step, epoch, self.net.channels, self.net.n_class, shuffle=False, pred=False)
129.
130.                            if self.net.summaries is True:
131.                                summary, acc, test_loss, pred, test_pred_arg = sess.run([merged, self.n
    et.accuracy, self.net.cost, self.net.predicter, tf.argmax(self.net.predicter, 3)], feed_dict = {self.ne
    t.x: test_x, self.net.y: test_y, self.net.keep_prob: 1.0})
132.                                test_writer.add_summary(summary, epoch)
133.                            else:
134.                                acc, test_loss, pred, test_pred_arg = sess.run([self.net.accuracy, self
    .net.cost, self.net.predicter, tf.argmax(self.net.predicter, 3)], feed_dict = {self.net.x: test_x, self
    .net.y: test_y, self.net.keep_prob: 1.0})
135.                            total_test_loss += test_loss
136.                            total_acc += acc
137.
138.                            err = self.error_rate(pred, test_y)
139.
140.                            test_white_acc, n_false_white = self.acc_no_correct_black(test_pred_arg, te
    st_mask)
141.
142.                            true_positives, false_positives, true_negatives, false_negatives = sess.run
    ([t_p, f_p, t_n, f_n], feed_dict = {self.net.x: test_x, self.net.y: test_y, self.net.keep_prob: 1.0})
143.                            if true_positives+false_positives != 0:
144.                                precision = float(true_positives) / float(true_positives+false_positive
    s)
145.                            else:
146.                                precision = 0.0
147.                            if true_positives+false_negatives != 0:
```

```
                                recall = float(true_positives) / float(true_positives+false_negatives)

                            else:
                                recall = 0.0
                            if precision != 0 and recall != 0:
                                F1_score = 2*(precision*recall)/(precision+recall)
                            else:
                                F1_score = 0.0
                            if true_positives+true_negatives+false_positives+false_negatives != 0:
                                calc_accuracy = (true_positives+true_negatives)/(true_positives+true_negatives+false_positives+false_negatives)
                            else:
                                calc_accuracy = 0.0

                            total_F1 += F1_score
                            total_calc_accuracy += calc_accuracy
                            total_prec += precision
                            total_rec += recall
                            total_err += err
                            total_test_white_acc += test_white_acc
                            total_n_false_white += n_false_white

                            if step == (test_iters - 1):
                                print('Test-steps are done')

                                l_test.append(total_test_loss/test_iters)

                                a_test.append(total_acc/test_iters)

                                score.append(total_F1/test_iters)

                                prec.append(total_prec/test_iters)
                                rec.append(total_rec/test_iters)

                                error.append(total_err/test_iters)

                                test_white_acc_list.append(total_test_white_acc/test_iters)
                                false_white_list.append(total_n_false_white/test_iters)
                        print(test_iters)
                        print('----------------------------------------')
                        print('****************************************')
                        print('Precision: ', total_prec/test_iters)
                        print('Recall: ', total_rec/test_iters)
                        print('.....................................')
                        print('False White: ', total_n_false_white/test_iters)
                        print('.....................................')
                        print('Error Rate: ', total_err/test_iters)
                        print('.....................................')
                        print('Accuracy at step %s: %s' % (step, total_acc/test_iters))
                        print('Calculated Accuracy: ', total_calc_accuracy/test_iters)
                        print('White Pixel Accuracy: ', total_test_white_acc/test_iters)
                        print('.....................................')
                        print('F1-Score: ', total_F1/test_iters)
                        print('.....................................')
                        print('Train Loss: ', total_loss/(training_iters/display))
                        print('Test Loss: ', total_test_loss/test_iters)
                        print('.....................................')
                        new_im = self.net.predicter.eval(feed_dict = {self.net.x: test_x, self.net.y: test_y, self.net.keep_prob: 1.0})
                        print("Saving test image to Result", epoch, ".png")
                        print("Saving test mask to Result_mask", epoch, ".png")
                        plt.imshow(new_im[...,1].reshape((1024,1024)))
                        name = "Result" + str(epoch) + ".png"
                        plt.savefig(name)
                        plt.imshow(test_mask.reshape((1024,1024)))
                        name_mask = "Result_mask" + str(epoch) + ".png"
```

```
                        plt.savefig(name_mask)
                        print('Saved')
                        print('****************************************')
                        print('----------------------------------------')

                        if epoch == epochs-1:
                            x_pred, _, mask_pred = data_provider(self.net.pred_data_name, test_batch_size, -1, epoch, self.net.channels, self.net.n_class, shuffle=False, pred=True)
                            y_dummy = np.empty((x_pred.shape[0], x_pred.shape[1], x_pred.shape[2], self.net.n_class))
                            prediction, pred_arg = sess.run([self.net.predicter, tf.argmax(self.net.predicter, 3)], feed_dict={self.net.x: x_pred, self.net.y: y_dummy, self.net.keep_prob: 1.0})
                            if self.net.channels == 5:
                                self.segmentation(prediction, pred_arg, x_pred[:,:,:,2], mask_pred)
                            else:
                                self.segmentation(prediction, pred_arg, x_pred, mask_pred)
                            print('The paths to the frame and the mask are: ', self.net.pred_data_name)

                self.plot_figure('Epoch', 'Loss', 'Loss', l_train, l_test, epochs, train_val=True)
                self.plot_figure('Epoch', 'Accuracy', 'Accuracy', a_train, a_test, epochs, train_val=True)
                self.plot_figure('Epoch', 'Precision & Recall', 'Precision & Recall', prec, rec, epochs)
                self.plot_figure('Epoch', 'Precision', 'Precision', prec, None, epochs)
                self.plot_figure('Epoch', 'White Pixel Accuracy', 'White Pixel Accuracy', train_white_acc_list, test_white_acc_list, epochs, train_val=True)
                self.plot_figure('Epoch', 'False White', 'False White', false_white_list, None, epochs)

                self.plot_figure('Epoch', 'F1 Score', 'F1 Score', score, None, epochs)
                self.plot_figure('Epoch', 'Error Rate', 'Error Rate', error, None, epochs)

                # Finish off sess and writers
                if self.net.summaries is True:
                    train_writer.close()
                    test_writer.close()
                sess.close()
                print('The program is done!')

            def error_rate(self, pred, test_y):

                error = 100 - 100*(np.sum(np.argmax(pred, 3) == np.argmax(test_y, 3)) / (pred.shape[0]*pred.shape[1]*pred.shape[2]))

                return error

            def evaluation(self):
                #logits, labels = without_correct_zeros(network_output, y)

                "Returns correct predictions, and 4 values needed for precision, recall and F1 score"
                #logits = tf.to_float(logits)

                #labels = tf.argmax(labels, 1)
                labels = tf.argmax(self.net.y, 3)
                logits = softmax_per_pixel(self.net.network_output)
                labels = tf.to_float(labels)
                # Step 1:
                # Let's create 2 vectors that will contain boolean values, and will describe our labels

                # Imagine that labels = [0,1]
                # Then
                # is_label_one = [False,True]
                # is_label_zero = [True,False]
                # Step 2:
```

```
266.                # get the prediction and false prediction vectors. correct_prediction is something that
     you choose within your model.
267.                logits = tf.reshape(logits, [-1, 2])
268.                labels = tf.reshape(tf.to_int32(labels), [-1])
269.                is_label_one = tf.cast(labels, dtype=tf.bool)
270.                is_label_zero = tf.logical_not(is_label_one)
271.
272.                correct_prediction = tf.nn.in_top_k(logits, labels, 1, name="correct_answers")
273.                false_prediction = tf.logical_not(correct_prediction)
274.
275.                # Step 3:
276.                # get the 4 metrics by comparing boolean vectors
277.                # TRUE POSITIVES
278.                true_positives = tf.reduce_sum(tf.to_int32(tf.logical_and(correct_prediction, is_label_
    one)))
279.
280.                # FALSE POSITIVES
281.                false_positives = tf.reduce_sum(tf.to_int32(tf.logical_and(false_prediction, is_label_z
    ero)))
282.
283.                # TRUE NEGATIVES
284.                true_negatives = tf.reduce_sum(tf.to_int32(tf.logical_and(correct_prediction, is_label_
    zero)))
285.
286.                # FALSE NEGATIVES
287.                false_negatives = tf.reduce_sum(tf.to_int32(tf.logical_and(false_prediction, is_label_o
    ne)))
288.
289.                return true_positives, false_positives, true_negatives, false_negatives
290.
291.            def acc_no_correct_black(self, pred, mask):
292.                pred = pred.reshape(1024*1024)
293.                mask = mask.reshape(1024*1024)
294.                mask = mask.astype(float)
295.                pred = pred.astype(float)
296.                #zero_idx_pred = np.where(pred==0)
297.                #zero_idx_mask = np.where(mask==0)
298.                ##comp_idx = np.where(zero_idx_pred[0] and zero_idx_mask[0])
299.                #comp_idx = np.where([a and b for a, b in zip(zero_idx_pred[0], zero_idx_mask[0])])
300.                comp_idx = np.where(mask==1)
301.                false_white = np.delete(pred, comp_idx)
302.                n_false_white = np.sum(false_white)
303.                #mask_new = np.delete(mask, comp_idx)
304.                #n_correct = np.sum(pred_new==mask_new)
305.                n_correct = np.sum(pred[comp_idx]==mask[comp_idx])
306.                #n_total = len(pred_new)
307.                n_total_white = np.sum(mask)
308.                if n_total_white != 0:
309.                    accuracy_white = n_correct/n_total_white
310.                else:
311.                    accuracy_white = 1
312.                return accuracy_white, n_false_white
313.
314.            def plot_figure(self, str_x, str_y, str_head, data1, data2=None, epochs=None, train_val=Non
    e):
315.                plt.figure()
316.                if epochs is not None and data2 is not None and train_val is not None:
317.                    plt.plot(np.linspace(0, epochs, len(data1)), data1, 'r', np.linspace(0, epochs, len
    (data2)), data2, 'b')
318.                    plt.legend(['train', 'validation'], loc='upper left')
319.                elif epochs is not None:
320.                    plt.plot(np.linspace(0, epochs, len(data1)), data1)
321.                elif epochs is None and data2 is not None:
322.                    plt.plot(np.linspace(0, epochs, len(data1)), data1, 'r', np.linspace(0, epochs, len
    (data2)), data2, 'b')
323.                    plt.legend(['precision', 'recall'], loc='upper left')
```

```
                plt.ylabel(str_y)
                plt.xlabel(str_x)
                plt.savefig(str_head)

            def segmentation(self, predict, pred_arg, truth, mask):
                predict = predict[...,1].reshape((1024,1024))
                pred_arg = pred_arg.reshape((1024,1024))
                pred_arg_idx = np.where(pred_arg == pred_arg.max())
                bck_grnd_val = pred_arg.min()
                pred_arg_row_idx = pred_arg_idx[0] - 1
                pred_arg_col_idx = pred_arg_idx[1] - 1
                truth = truth.reshape(truth.shape[1],truth.shape[2]) + 127

                edge_pixl = []
                for pixel in range(len(pred_arg_row_idx)):
                    i = pred_arg_row_idx[pixel]
                    j = pred_arg_col_idx[pixel]
                    if (pred_arg[i+1,j] == bck_grnd_val) or (pred_arg[i-
    1,j] == bck_grnd_val) or (pred_arg[i,j+1] == bck_grnd_val) or (pred_arg[i,j-
    1] == bck_grnd_val) or (pred_arg[i+1,j+1] == bck_grnd_val) or (pred_arg[i-1,j-
    1] == bck_grnd_val) or (pred_arg[i+1,j-1] == bck_grnd_val) or (pred_arg[i-1,j+1] == bck_grnd_val):
                        edge_pixl.append((i,j))

                file = open('edge_pixels','w')
                file.write(str(edge_pixl).strip('[]'))
                file.close()

                plt.figure()
                plt.imshow(predict)
                plt.colorbar()
                plt.savefig('Prediction')
                plt.figure()
                plt.imshow(pred_arg)
                plt.colorbar()
                plt.savefig('Argmax Prediction')
                plt.figure()
                plt.imshow(truth)
                plt.colorbar()
                plt.savefig('Truth')
                plt.figure()
                plt.imshow(mask)
                plt.colorbar()
                plt.savefig('Mask')
```

18.1.6 restore_and_use_model.py

```
import tensorflow as tf
import numpy as np
import tkinter
import matplotlib.pyplot as plt
import numpy as np
from data_handler import data_provider
#from wrappers import softmax_per_pixel

class Restore_and_Use_Model:
    """
    Restore the model (Stemnet) and use it for prediction

    :param frame_path: the path of the frame
    :param mask_path: the path of the mask
    :param meta_model_path: the path of the meta-model
    """

    def __init__(self, frame_path, mask_path, meta_model_path):
        self.frame_path = frame_path
```

```
        self.mask_path = mask_path
        self.meta_model_path = meta_model_path
        self.path = meta_model_path.rsplit('/', 1)[0]

    def run_session(self):
        data_name = (self.frame_path, self.mask_path)

        ckpt = tf.train.get_checkpoint_state(self.path) # Argument should be the path where the model is stored

        saver = tf.train.import_meta_graph(self.meta_model_path)

        init = tf.global_variables_initializer()

        with tf.Session() as sess:

            sess.run(init)

            saver.restore(sess, ckpt.model_checkpoint_path)

            graph = tf.get_default_graph()

            x = graph.get_tensor_by_name("x:0")
            y_ = graph.get_tensor_by_name("y_:0")
            keep_prob = graph.get_tensor_by_name("keep_prob:0")

            #output = graph.get_tensor_by_name("output:0")
            predicter = graph.get_tensor_by_name("predicter:0")

            #predicter = softmax_per_pixel(output)

            x_pred, _, mask_pred = data_provider(data_name, 1, -1, 0, 1, 2, shuffle=False, pred=True)
            y_dummy = np.empty((x_pred.shape[0], x_pred.shape[1], x_pred.shape[2], 2))
            prediction, pred_arg = sess.run([predicter, tf.argmax(predicter, 3)] ,feed_dict={x: x_pred, y_: y_dummy, keep_prob: 1.0})
            self.segmentation(prediction, pred_arg, x_pred, mask_pred)

        sess.close()

    def segmentation(self, predict, pred_arg, truth, mask):
        predict = predict[...,1].reshape((1024,1024))
        pred_arg = pred_arg.reshape((1024,1024))
        pred_arg_idx = np.where(pred_arg == pred_arg.max())
        bck_grnd_val = pred_arg.min()
        pred_arg_row_idx = pred_arg_idx[0] - 1
        pred_arg_col_idx = pred_arg_idx[1] - 1
        truth = truth.reshape(truth.shape[1],truth.shape[2]) + 127

        edge_pixl = []
        for pixel in range(len(pred_arg_row_idx)):
            i = pred_arg_row_idx[pixel]
            j = pred_arg_col_idx[pixel]
            if (pred_arg[i+1,j] == bck_grnd_val) or (pred_arg[i-1,j] == bck_grnd_val) or (pred_arg[i,j+1] == bck_grnd_val) or (pred_arg[i,j-1] == bck_grnd_val) or (pred_arg[i+1,j+1] == bck_grnd_val) or (pred_arg[i-1,j-1] == bck_grnd_val) or (pred_arg[i+1,j-1] == bck_grnd_val) or (pred_arg[i-1,j+1] == bck_grnd_val):
                edge_pixl.append((i,j))

        file = open('edge_pixels','w')
        file.write(str(edge_pixl).strip('[]'))
        file.close()

        plt.figure()
        plt.imshow(predict)
        plt.colorbar()
        plt.savefig('Prediction')
```

```
        plt.figure()
        plt.imshow(pred_arg)
        plt.colorbar()
        plt.savefig('Argmax Prediction')
        plt.figure()
        plt.imshow(truth)
        plt.colorbar()
        plt.savefig('Truth')
        plt.figure()
        plt.imshow(mask)
        plt.colorbar()
        plt.savefig('Mask')
```

18.2 Demo Scripts

18.2.1 Network1.py

```
import stemnet
import train_and_eval

net = stemnet.Stemnet()

run = train_and_eval.Train_and_Eval(net)

run.train_and_evaluation()
```

18.2.2 Network5.py

```
import stemnet
import train_and_eval

net = stemnet.Stemnet(channels=5)

run = train_and_eval.Train_and_Eval(net)

run.train_and_evaluation()
```

18.2.3 predict.py

```
import restore_and_use_model

frame_path = 'Frames/frame2009_1_2.png'
mask_path = 'Masks/mask2009_1_2.png'
meta_model_path = '/media/misakss/DATAPART1/Network1_model4/model-normal.ckpt-4.meta'

model = restore_and_use_model.Restore_and_Use_Model(frame_path, mask_path, meta_model_path)

model.run_session()
```

TRITA TRITA-EE 2017:128
ISSN 1653-5146