UNIVERSITY OF COPENHAGEN

Neural networks and their application on images

Theory and application of neural networks

Camilla Kergel Pedersen - wkl125 - Supervised by: Oswin Krause and Kasra Arnavaz

2019-12-06

1 Abstract

This report describes the theory behind neural networks and an application of a neural network in Python using Keras [1]. This project is based on a bachelor thesis from the fall working with images of a tubular structure [2]. This structure was a muscle in a pancreas from a mouse and the goal of that project was to segment out structure from noise and background. That goal of this project is to try and improve these previous results and study neural networks. Now there is access to more training-data of a higher quality and instead of using pre-processing and a random forest classifier, a neural network is implemented.
In order to make the implementation, a lot of theory behind neural networks has been studied, and the conclusion of this studying resulted is the decision that a convolutional neural network with a U-net structure was the way to implement this.
The results of the implementation in this project show an impressively high accuracy of over 99%. However the measure of specificity still implies problems with correctly identifying foreground-pixels because of a rather large imbalance in the dataset. Moreover the results are evaluated as visual images where the similarity between predictions and the ground truth are undeniable.
The biggest problem with this project is the computational issues that come along with such a huge data-set and a convolution neural network of many layers. This is reducing the size of the training set. However, the results show that this can be accepted since the network still performs a lot better than the classifier in the previous project.

Contents

2 Motivation

The motivation for writing this project lies in the discovering from a previous bachelor project. The base for this project was in short to segment out structure on images of a tubular structure in the pancreas of mice. The pancreas produce beta-cells which are responsible for the production of insulin. The scientists that have produced the data set have a hypothesis stating that, when the structure in these images change, the beta cells are produced. A correct binary segmentation of these images will make the structure a lot easier to investigate with regards to this hypothesis.
The segmentation in the previous project was done by using some methods for edge detection to aid the learning of the classifier, and then train a random forest classifier to segment out the structure. The goal was to make predictions for new data based on the training from both edge detected and original images. The project was overall successful but it contained some problems which a lot of the motivation for this project is based on. One of the major problems was the training data, which consisted of 15 annotated images made with the k-means clustering algorithm. It had a lot of inconclusive clusters containing both structure and background, and these where eliminated before training. This resulted in a severe lack of important training data since the pixels in these clusters was the ones hardest to classify. Furthermore this reduced the amount of training data severely.
The bachelor thesis ended up with a prediction accuracy of 97.5% but the huge question was if this result could be trusted due to the quality of the annotations, and the small amount of the training- and test data.
Now there is access to more and better annotations that are made partly manually and with a software that is designed for, among other things, processing of biological images. A more detailed description of the making of the annotations is in the next section.
The goal of this current project is to learn about neural networks and investigate their general use. Furthermore the goal is to create a convolutional neural network in python trained on these new annotations, and see if it is possible to get better and more trustworthy predictions.

3 Data

There are a lot of differences between this new data-set, and the one used in the bachelor thesis. This new data is constructed with a whole new approach. Firstly the microscope analysis software, Imaris, is used to segment the images. This is a tool especially designed for biological image processing. After this initial segmentation a manual removal is performed. Since it is known that certain structures, such a for example round objects, are often noise this knowledge can be used to manually segment out noise. Though a great problem with Imaris is that it is not possible to add structure to the processed image, only to remove it. This problem is currently being solved, but the resolving data from this has not been available for this project. The structure of the data is described in timesteps and depth of the structure. Each 2D-image has a time, a layer and is 821x523 pixels large. It should be noted that the size of the images differ from the bachelorproject where the resolution was 1024x1024.

4 Theory of neural networks

Before diving into the practical part and implementing the neural network, a basic introduction will be provided in order to achieve the understanding needed for the creation of the code. This is also needed to be able to make choices regarding the design and form of the network based on the given problem.
A neural network is a computer system whose design is inspired by the real biological brain. The human brain contains around 100 billion neurons which are used for basically everything that goes on in the human mind.

In a neural network neurons are considered as an object that holds a number, and that number indicates how close the neuron is to fire. The neuron can be considered as a function with a binary output indicating whether it fires or not. The connections between neurons in the brain are called synapses, and it is estimated that an average human have around 10^{15} synapses. Not all neurons are connected and some connections can be stronger than others. Each neuron has around 5.000 connections in average, and the output depends on these connections.

This knowledge is used when it is tried to model the human brain with a computer system. The neurons and the synapses are modelled, and it is attempted to teach the model how to adjust these connections while learning. The input to a neuron is most often described by a vector representing the input values. Since the neuron is described as a function it uses the input $\vec{x}$ a vector of weights $\vec{w}$ and a bias b to calculate an output value by the formula:

$$\vec{w} \cdot \vec{x} + b$$

If this is connected to the further described model of the human brain it can be said that $\vec{w}$ describes the synapses and their strength, $\vec{x}$ defines the output of the other neurons and b adjusts the sensitivity on when the neurons should fire.

The function which determines whether a neuron should fire or not is in some cases just a threshold that splits the two options. This threshold is also known as a decision boundary. When have this is the case it is called a linear classifier since the output becomes divided by a line.

So far the considerations has only been on functions with binary output, because the model takes inspiration from the biological neuron. I this case we also know the function as a "Heaviside" function.
In this project the intuitive output is binary since there are only two options for a pixel - Foreground or background, however, since this is a computer system, which design we are allowed to choose we do not have to restrict our self to this. The problem of having more classes can easily be solved by modifying the loss function. This is also done here. Instead of just classifying the pixels as 1 or 0, they are given a number and the higher number the higher chance of foreground, and vice versa. The final classification can then be achieved by thresholding this value.

4.1 Loss functions

In order to decide how to design the function in the neurons, it is needed to take a look at $\vec{w}$ and b and define a **loss function**. $\vec{w}$ and b should be chosen such that the loss function is minimized, meaning that the model predicts as close to the real predictions as possible. This loss function can be chosen according to the relevant problem, but is mostly based on some sort of distance from the prediction to the real result, and designed so that it is not to sensitive to potential outliers. For this project we specify loss as the function known as binary cross-entropy, since we are training a binary classifier.
The point of this loss function is to predict the probability for the class of a given pixel. Since we know that probabilities sum to 1, the predicted probability for each pixel will contain information from both classes. The formula for the

binary cross-entropy loss function is given as:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

where y is the label (background = 0 or foreground = 1) and p(y) is the predicted probability of a random point being foreground. [3]
The point of any loss function is to penalize bad predictions and reward good ones. In this case the desire is for false predictions to be characterized with a really large output of the loss function. This is why the logarithm is used, as the results of this fits this criteria. Since taking the logarithm of values between 1 and 0 results in negative values a minus is placed in front of the loss function.

4.2 Feeding forward information

Looking at this from a practical view with the current problem in mind of classifying pixels in an image. The theory above concludes how a neural network is structured in layers. An input layer, a number of hidden layers and an output layer. The thing that still needs resolving is how these layers communicate and what they consist of. In this case the input layer is all the pixels in a given image, and the output is a classification of each image. For each of the neurons in the hidden layers we calculate the activation function inside the neurons by applying a known function such as the Sigmoid og ReLU activation function (see below) to the formula in the neurons: $wx + b$. When feeding forward information is used the neurons just fire or don't fire down though the network and when the output layer is reached the pixel is classified as the best "fit" i.e. the neuron in the output layer with the best value, in most cases, the highest value.
The way the network learns how to approach this is by learning the best w and b for each neuron. A neural network will often have a lot of neurons and therefore a lot of weights. When it is given training data with labels, it adjusts the w's in each neuron to fit the label. The hope is that the weights after a certain amount of training-data are so generalized that they'll almost always predict correctly on new data. [4]

4.3 Gradient descent

It is recalled that the weights of each neuron corresponds to the strengths of a synapsis to another neuron, and the bias indicates whether the current neuron

tends to be active or not. When the network is initialized the weights and biases are determined completely randomly, and needless to say, this network will perform very badly. In order to adjust these values the loss-function is computed in each of the output neurons. The loss-function is essentially a function taking in a vector of all the weights in the neural network, and the value of the loss function is an indicator of how wrong this particular training was. Since the desire is for the training to be correct, the loss will be low is this case.
This means that the desire is to minimize this loss-function, and that is done by using gradient descend and taking steps in the opposite direction of the gradient since the minimum should be reached. In order to take these steps the vector w is adjusted so $L(w)$ returns the lowest possible loss. After training the goal is to have an average loss-function that is fitted to a lot of different cases and therefore have adjusted the weights to work in most general situations. [4]

4.4 Back propagation

Above it is described that the gradient descend is needed to compute the minimum loss. However it is not fully described how this is done. In order to achieve those minimum weights an algorithm called backpropagation is used. The goal here is to determine how a single training example should change the weights and biases. The magnitude of each element in the vector w indicates how sensitive the changes in this weight is to the value of the loss function. Often the rule is, that the closer to zero the magnitude is, the less sensitive.
In order to probably explain how backpropagation works, an example will be shown. It is assumed that the network in this example is badly trained and does not perform well. So changes in the weights should happen. For inclusion of the current problem, let's assume a foreground pixel is predicted to have a very low value in the output (indicating background). If this had been correct, the desire would be to keep the current weights. However it isn't, so they should be changed. It is illegal to just change the output, so the only possible thing is to adjust the weights in all the layers, which means a way to go back trough the network is needed. The desire here is to adjust the weights so the value of the output neuron choosing the wrong label (pixel value in this case) gets nudged down and the other value is nudged up.
Zooming in on this, the focus is on the output-neuron with the low value for the foreground pixel and correspondingly high value for background which is wrong. These values should be changed. It is recalled that these current values is a

result of a weighted activation sum of all the neurons in the previous layer plus a bias which is then plugged in to an activation function such as for example the sigmoid. This means that there is three factors that one can change. The weights, the inputs and the bias. Loosely speaking this is achieved by comparing the values with the desired values which should be 0 or 1 depending on the class. Track is kept on these previous values as we go back though the network and adjust the weights according to their sensitivity. [4]

4.5 Calculus behind backpropagation

In order to explain the backpropagation a bit more detailed, the math should be considered. This topic is pretty advanced, and cannot be fully covered here, but demonstration of a small example should be considered. It is assumed that the network is very small with three weights and three biases. The neuron in the last layer is denoted as a_L and the neuron in layer just before is labelled as a_{L-1} and so on. It is assumed that the desired value y for a_L is 1. These two values are plugged into the binary cross-entropy loss-function and the loss is calculated. This is denoted as L_0. It is recalled that a_L is determined by its w w_l it's bias b and the previous activation a_{L-1} plugged into an activation function. This can be written as:

$$a_L = sigmoid(w_L a_{L-1} + b_L)$$

and the correlation between all these mentioned values can be seen in figure 1. Since the desired knowledge is to know how sensitive the loss function is to small changes in w_L, this can be expressed mathematically by asking for the derivative of the loss function with respect to w_L. This can be calculated (by going down the tree in figure 1), and defining a $z = w_{L L-1} + b_L$. This results in three derivatives, and we write this down as:

$$\frac{\partial C_0}{\partial w_L} = \frac{\partial z}{\partial w_L} \frac{\partial a_L}{\partial z} \frac{\partial C_0}{\partial a_L}$$

This is where the chain rule comes in. The derivatives are written down:

$$\frac{\partial z}{\partial w_L} = loss'(a_L, y)$$

$$\frac{\partial a_L}{\partial z} = sigmoid'(a_L, y)$$

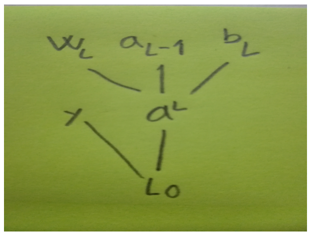


Figure 1: Connection between weight activations and loss function in an output neuron.

$$\frac{\partial z}{\partial w_L} = a_{L-1}$$

So we get:

$$\frac{\partial C_0}{\partial w_L} = \frac{\partial z}{\partial w_L}\frac{\partial a_L}{\partial z}\frac{\partial C_0}{\partial a_L} = loss'(a_L, y) \cdot sigmoid'(a_L, y) \cdot a_{L-1}$$

If the network is expanded to have more than one neuron in each layer this simply means that we need to compute this for each a_{L-1} in the previous layer and sum up the difference between those values and y. So we write this as:

$$L_0 = \sum_{j=0}^{n_{L-1}} loss((a_{L-1})_j - y)$$

This results in the definition on how these weight and biases in all the neurons are changed during training. The next sections will go through the activation functions and the kind of neural network that is used in this specific assignment. [4]

4.6 Activation functions

4.6.1 The sigmoid function

The sigmoid function ranges between 1 and 0 and is therefore ideal to use when calculating with probabilities. The sigmoid function is non-linear and is therefore also ideal for classification since it is very sensible for input containing the middle values (around 0.5). The shape of the curve causes this activation function to often place output in either end of the curve and thereby it resembles a classification.

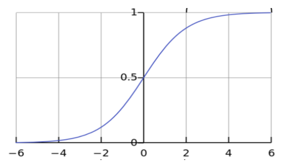


Figure 2: Sigmoid function

The sigmoid function is described by:

$$\frac{1}{1 + e^{-v}}$$

and is drawn in figure 2.

4.6.2 ReLU

Another activation function used in this project is the ReLU activation function. This function simply activates the neuron if the input to the neuron is larger than zero. ReLU has the huge benefit over the sigmoid function that it is a lot faster to compute since no real calculations has to be made. This can make the network a lot faster and that is very important when the data set is as large as the one in this project. A problem with ReLU can be that there is no difference in the reaction of a neuron receiving a very large negative input and a very small negative input. This can result in a huge part of the network being very passive.

5 Convolutional neural networks

A specific kind of neural networks is called convolutional neural networks. This is the kind that is used in the implementation of this project. Convolutional neural networks are often used in connection with image recognition and segmentation. A quick way to describe convolution is to say that it measures how much two functions overlap while the pass each other. It gives a view on how similar they are. The function that is passing is known as a filter and the convolution sends back a signal telling how much they are alike.
For image processing this is very useful. This idea is transferred to be used on a matrix, which is the mathematical definition of an image on a computer. Knowing about the filter, it can be possible to say something about what is on the image. At early processing this could be something that identifies edges or corners and later on it will then be possible to recognize objects or faces. [5]

The principle in a convolutional network is to find the patterns in the image. By finding similarities in where different features, such as edges, occur for a given object it will then be able to identify which object the image might contain. This is done by going over small parts of the image like when doing normal convolution and convolving with the known feature. This results in a feature map that the network can use.
Since we look for many different features and therefore convolve parts of the image with a lot of different filters a convolutional neural network is much more complex than just doing a simple convolution.
Normally when doing a convolution we have 3 different channels R, G and B representing the colour value of a pixel. But since this project is only based on images in greyscale the focus will be on that case, which means that there is only one channel.
When applying a filter to a patch of the image, the dot product of these two is calculated, since this value will be really high when there are similar values in the same places, and this is how the patterns are spotted.
All the found dot products are placed in a map consisting of a matrix of the dot product for each move. The position of the dot product gives the placement of this pattern in the original image. Each filter will have a different map. This is the general idea of convolution.
After the convolution down sampling and maxpooling is performed. In the implementation from this report a 2x2 patch is used. This means that for each 2x2 patch in each of the maps the max-value is pulled out and saved while the rest of the information is lost. This preserves space and therefore also the speed of the computations. These maps is what the neural networks use to train on. In this way, the neural network can learn which values to look for in which positions in order to find the specified object or pixel.

6 U-net structure

For the implementation of this project a U-net structure is used. The U-net structure is a special kind of convolutional neural network that is designed to do segmentation on biomedical images. Since the images in this project bear much similarities to images of veins, it was thought that this method would be very efficient for the problem.
The principle of the U-net structure is to have a contracting path and an

extracting path. The contracting path does exactly that same as a convolutional network described above. Convolution of a small patch, in this case 3x3, is performed, and then the ReLU activation function is applied. Afterwards down sampling and maxpooling is performed. The purpose of this is to map a context of the image in order to do segmentation.
Furthermore batch normalization is performed for each step. This simply means to normalize the values of the feature map, so the correlation of the values make sense according to what is searched for. This increases the stability of the neural network, and is simply performed by subtracting a batch mean and dividing with a standard deviation. Both these values are based on previous values from a convolution. The extracting path does the opposite of the contracting path. Here the information gathered by contracting is used in order to reconstruct the image used the feature maps. The enables the model to take the context of each pixel into account, in stead of just considering one pixel at a time.

7 Slumr GPU

In order to train the neural network an external GPU has been used called Slurm. In order to run the code with enough training data and epochs to get something useful Slurm has to be used. The .py file should be copied onto a node and then it can run using at batch script. The batch script used for this project is attached in appendix A. It takes around 10 minutes with 100 epochs to run. The output is a .h5 file containing the weights obtained by the training. In order to show the results a Jupyter notebook is used with matplotlib and the weights are copied back to the harddrive. This is to avoid having to use the Slurm-GPU for visualisation as this takes a lot of time and requires many steps. If this where to be done, it would require saving the images as .tif files and copying them back each time they should be shown. That takes a lot of time compared to just showing the images in Jupyter with matplotlib [6].

8 Implementation

8.1 How to run the code

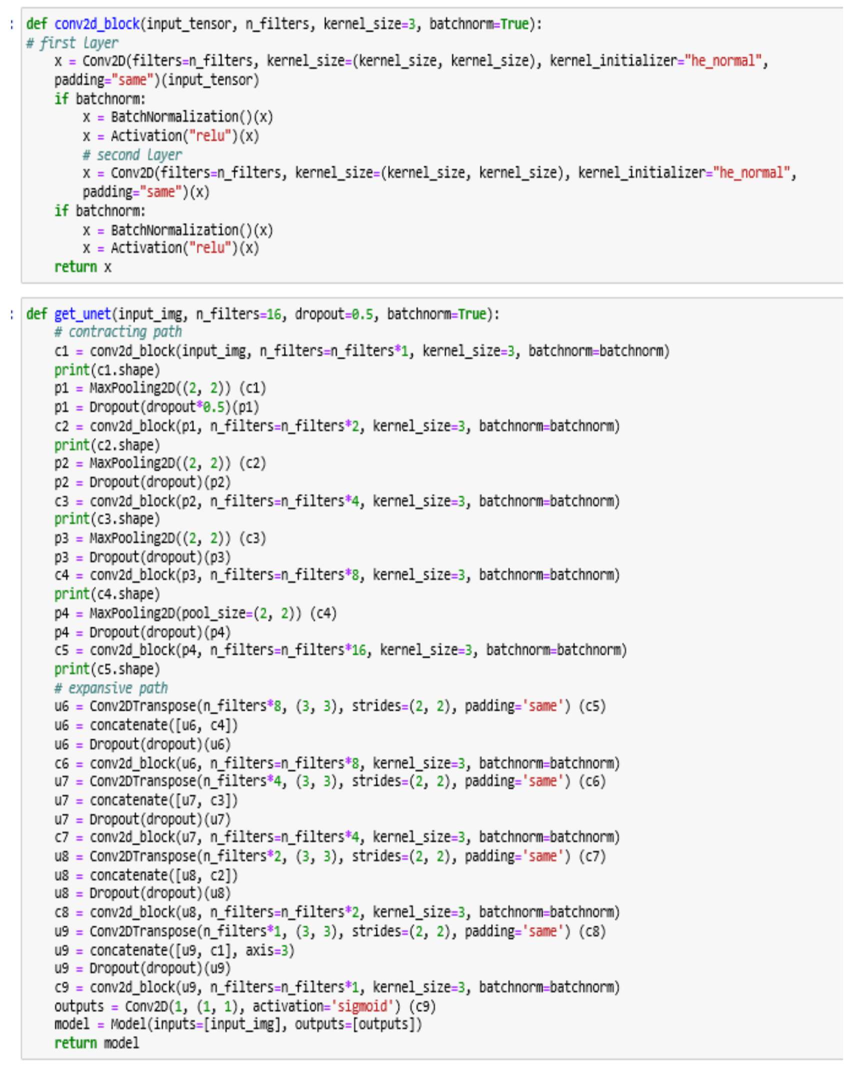
If there is no desire to retrain the network, the code can simply be run by laughing a Jupyter notebook and running the file named "neural network". Make sure that the file w.h5, which contain the pretrained weights, is in the

same folder.
If the desire is to retrain the network the Slurm-GPU mentioned above should be used. The GPU can be accessed by running abc123@ssh-diku-image-science.ku.dk in the command prompt where abc123 is a ku-id with access to the GPU. Access must be granted by the supervisor of this project. Then the "scp" option can be used to copy the files on to the GPU. Afterwards a node should be entered. For this project, the node named a00552 is used, and it is accessed by running the command "ssh a00552". After ssh'ing into Slurm the code files should be copied over again using "scp", and now the code can be run by using the batch-script in appendix A. This script should be copied on to the node, and now it is possible to type the command "sbatch run.sh". After the code is run, the resulting files, in this case the file containing the new weights, can be copied back to the hard drive and used for evaluation of the results. Currently it is assumed that the pretrained weights are sufficient, and therefore it has not been prioritized to go into depth about how to retrain the network.
Another option is to use the library tifffile and save the images on the node and copy them back. However it was found that it was a lot more convenient to just copy the weights, and then play with the images using matplotlib [6] and Jupyter notebook avoiding the use of Slurm for this part.

8.2 The neural network

The implementation of the convolutional neural network is based on an example that detect salt on stones [7]. The code was modified to fit this dataset and redesigned for this purpose, but a lot of the structure remains the same as the problems are similar.
The implementation consists of one .py file for training and a short file to try out the random forest classifier. However the implementation of the neural network will be what these results are focused on. Firstly the data is loaded in and structured in a 4D-array of pixels where each of the axes represent a detail of the pixels. The are structured as follows: (time, depth, height, width).
Afterwards a test-set and a training set is created. It is divided so that there is 100 images in total and 75 of them are used for training and 25 are used for testing. By testing on data, that is not used for training, overfitting is prevented. Overfitting can result in a model that is to well designed for specific data, and therefore will perform poorly on new data.
Then the U-net is constructed by applying the convolution layers with 16 different

filters using the ReLU activation function and batch normalization. Then max-pooling and dropout is performed and the whole thing is repeated now convolving with the result of the previous layer. This is done 4 layers down. Afterwards, the convolution is reversed using the theory mentioned above. See a code snippet from the implementation of this in figure 3.
Now the model can be created by using this U-net, and the training data can be used to fit the model. The training takes a while, so most of the time the function load_weights is used for "training" since we can save the trained weights in a .h5 file for later use. For this we use the h5py package in python [8]. This means that one is able to play around with the results without constantly having to retrain the model in order to rerun the code.
Lastly the results are evaluated and saved in an array of predictions. This array is then what is used to show the final results.
Since the network spits out images in values from 0 to 1, and the real annotations are binary only containing the values 0 **or** 1, the predictions are applied with a threshold. The intuitive threshold of 0.5 is chosen.



```
def conv2d_block(input_tensor, n_filters, kernel_size=3, batchnorm=True):
# first layer
    x = Conv2D(filters=n_filters, kernel_size=(kernel_size, kernel_size), kernel_initializer="he_normal",
    padding="same")(input_tensor)
    if batchnorm:
        x = BatchNormalization()(x)
        x = Activation("relu")(x)
        # second layer
        x = Conv2D(filters=n_filters, kernel_size=(kernel_size, kernel_size), kernel_initializer="he_normal",
        padding="same")(x)
    if batchnorm:
        x = BatchNormalization()(x)
        x = Activation("relu")(x)
    return x
```

```
def get_unet(input_img, n_filters=16, dropout=0.5, batchnorm=True):
    # contracting path
    c1 = conv2d_block(input_img, n_filters=n_filters*1, kernel_size=3, batchnorm=batchnorm)
    print(c1.shape)
    p1 = MaxPooling2D((2, 2)) (c1)
    p1 = Dropout(dropout*0.5)(p1)
    c2 = conv2d_block(p1, n_filters=n_filters*2, kernel_size=3, batchnorm=batchnorm)
    print(c2.shape)
    p2 = MaxPooling2D((2, 2)) (c2)
    p2 = Dropout(dropout)(p2)
    c3 = conv2d_block(p2, n_filters=n_filters*4, kernel_size=3, batchnorm=batchnorm)
    print(c3.shape)
    p3 = MaxPooling2D((2, 2)) (c3)
    p3 = Dropout(dropout)(p3)
    c4 = conv2d_block(p3, n_filters=n_filters*8, kernel_size=3, batchnorm=batchnorm)
    print(c4.shape)
    p4 = MaxPooling2D(pool_size=(2, 2)) (c4)
    p4 = Dropout(dropout)(p4)
    c5 = conv2d_block(p4, n_filters=n_filters*16, kernel_size=3, batchnorm=batchnorm)
    print(c5.shape)
    # expansive path
    u6 = Conv2DTranspose(n_filters*8, (3, 3), strides=(2, 2), padding='same') (c5)
    u6 = concatenate([u6, c4])
    u6 = Dropout(dropout)(u6)
    c6 = conv2d_block(u6, n_filters=n_filters*8, kernel_size=3, batchnorm=batchnorm)
    u7 = Conv2DTranspose(n_filters*4, (3, 3), strides=(2, 2), padding='same') (c6)
    u7 = concatenate([u7, c3])
    u7 = Dropout(dropout)(u7)
    c7 = conv2d_block(u7, n_filters=n_filters*4, kernel_size=3, batchnorm=batchnorm)
    u8 = Conv2DTranspose(n_filters*2, (3, 3), strides=(2, 2), padding='same') (c7)
    u8 = concatenate([u8, c2])
    u8 = Dropout(dropout)(u8)
    c8 = conv2d_block(u8, n_filters=n_filters*2, kernel_size=3, batchnorm=batchnorm)
    u9 = Conv2DTranspose(n_filters*1, (3, 3), strides=(2, 2), padding='same') (c8)
    u9 = concatenate([u9, c1], axis=3)
    u9 = Dropout(dropout)(u9)
    c9 = conv2d_block(u9, n_filters=n_filters*1, kernel_size=3, batchnorm=batchnorm)
    outputs = Conv2D(1, (1, 1), activation='sigmoid') (c9)
    model = Model(inputs=[input_img], outputs=[outputs])
    return model
```

Figure 3: Codesnippet from the implementation of the convolutional neural network and the U-net structure.

9 Results

9.1 Applying a random forest classifier

Since one of the main subjects of the bachelor thesis was to apply a random forest classifier to the pre-processed images it was thought to be interesting to train a the random forest classifier from sk-learn [9] on these new annotations.

However this did not yield anything useful. An example of a result can be seen in figure 4. This is probably due to the lack of training-data, and the accuracy calculated. However it was not possible to include more training data since it made everything crash. This could be solved by running the code on the Slurm GPU, but this was not possible since the sk-learn library could not be installed on the GPU. Another reason is the high accuracy on the result on over 98%. It is very likely that the classifier ends up on this result by the high accuracy, but visually this fails completely. The issues here is the huge difference between the amount of foreground and background pixels. The random forest classifier is in this examples trained on pixels from 2 different images which probably is not enough and they might be bad examples not containing much strcutre. However the notebook kept freezing with more data. Resizing the images by cropping out completely black background might have freed more space for better training-data, but this was not prioritized as the focus was put on the neural network, and the hypothesis is that the result of the neural network will be significantly better, no matter what is done to improve the classifier, simply because this problem is to complex for a classifier, and there are to many features.

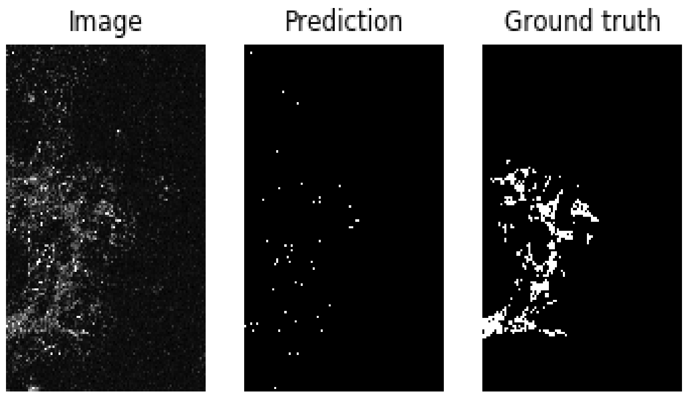


Figure 4: Example data from the implementation of the random forest classifier. Timestep nr 2, depth 15. From left: The image, the prediction and the annotation.

9.2 The neural network

The results of the implementation of the neural network are evaluated both with visual examples shown in figure 5 and 6 and with a calculation of accuracy,

Measure	Accuracy	Specificity	Sensitivity
Result	99.58%	99.87%	81.14%

Table 1: Evaluation of the predictions from the neural network.

specificity and sensitivity. The accuracy can be a bit misleading since there are so many background pixels that are labelled correctly just because they are "to easy" for the network. Even though a high accuracy is measured it does not imply a satisfying visual result. It is seen before that working with this data set, and different machine learning algorithms, can result in completely black predictions and still give a pretty high accuracy. This was also seen in the bachelor-thesis, when the weights where not balanced in the classifier. Therefore specificity and sensitivity is measured as well. The results are shown in table 1. and are measured from a part of the test set containing 10 random 2D-images. The sensitivity measures the models ability to correctly identify foreground pixels, and the specificity measures the ability to correctly identify background pixels. Sensitivity and specificity are defined as:

$$Se = \frac{TP}{TP + FN}$$

$$Sp = \frac{TN}{TN + FP}$$

where T and F are true and false and P and N are positives and negatives.

These results show that, as expected, the ability to correctly identify foregrounds pixels is significantly lower than the actual accuracy. This is probably due to the before mentioned problem with the very low amount of foreground-pixels in the dataset compared to the amount of background pixels. These results are from predicting on the first 10 images in the test-set which corresponds to the images in timestep nr 2 from depth 10 to 20, when the sample size is set to 50. This detail is just to recreate the results as the vary a bit depending on the test set. But in general the numbers are very stable except from the sensitivity that varies a bit more and lays around 70% to 80 %.

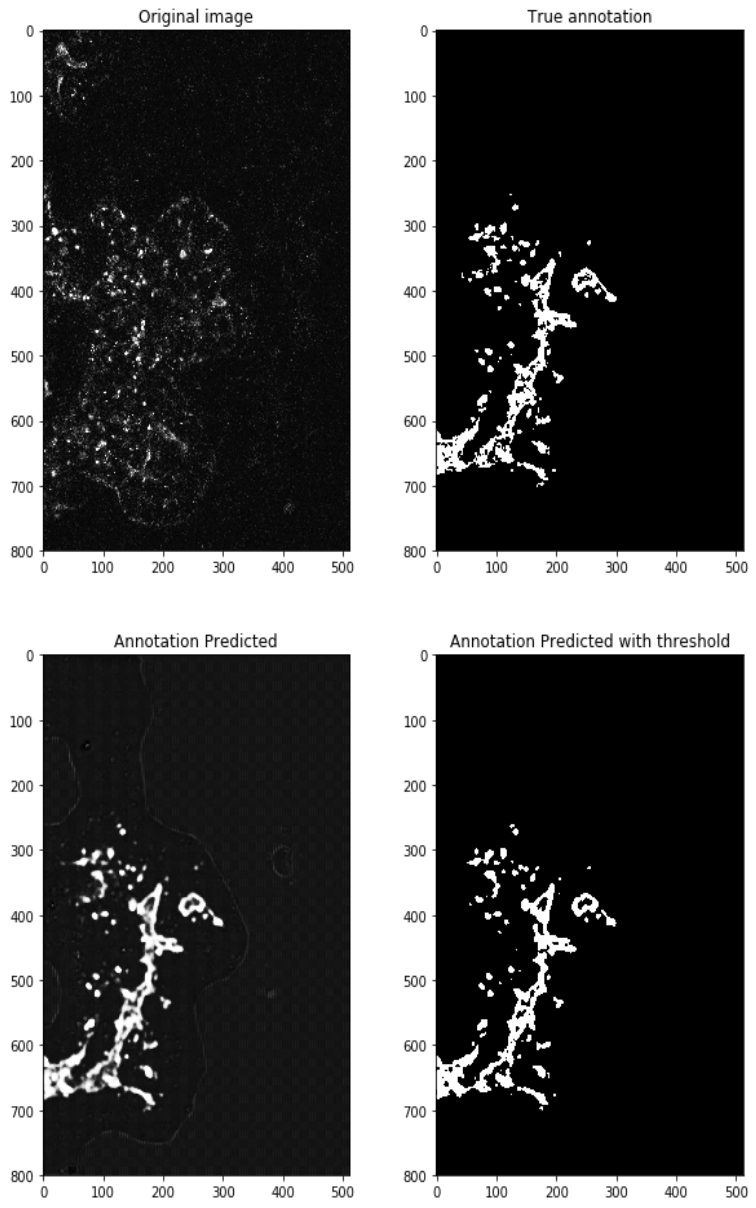


Figure 5: Timestep 24 depth 15.
Example data from the implementation of the neural network. Upper left corner show the original image, upper right corner show the ground truth. Lower left is the prediction before the threshold, and lower right is the final prediction of the image applied with the threshold.

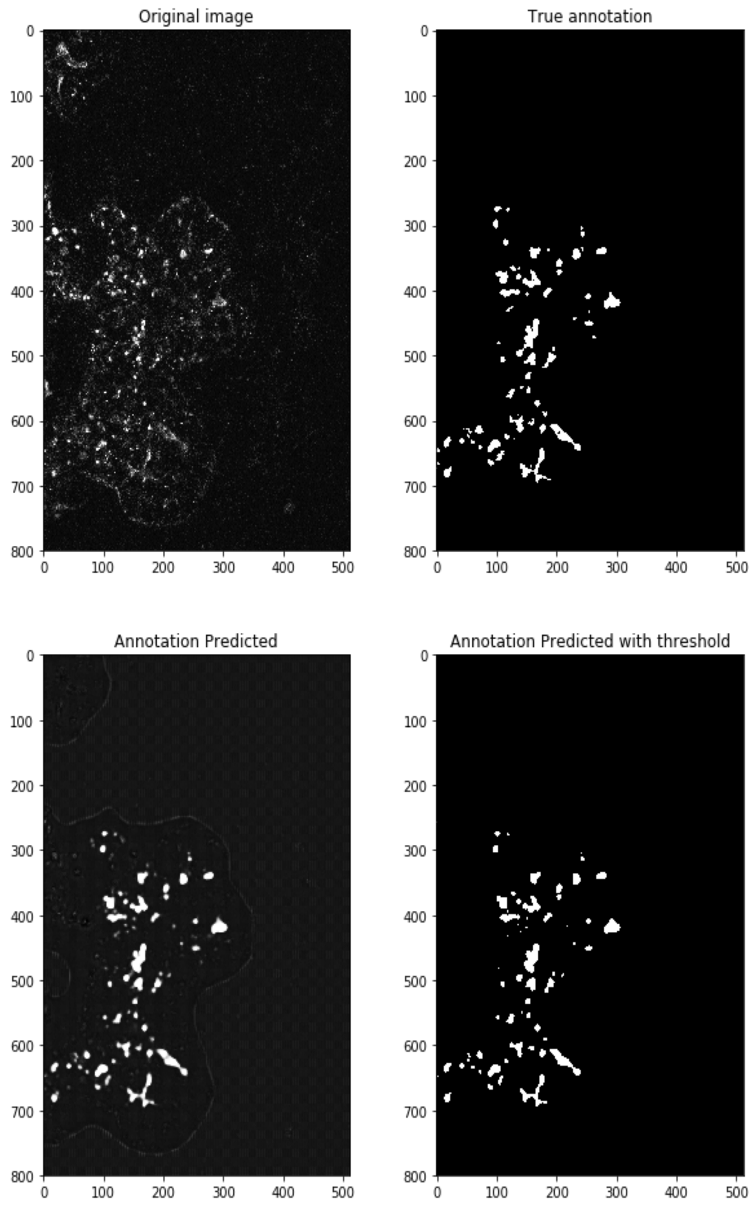


Figure 6: Timestep 2 depth 15.
Example data from the implementation of the neural network. Upper left corner show the original image, upper right corner show the ground truth. Lower left is the prediction before the threshold, and lower right is the final prediction of the image applied with the threshold.

10 Discussion

A huge problem in the implementation of this project was the need to use the Slurm-GPU, and the fact that working with this much data is very computationally exhausting in general. In order to test the actual network during the implementation it was needed to run it on the external GPU which takes around 10 minutes, and then use time on copying the images back to the harddrive and then show them on the screen. For each time the images did not match the expected output this step was repeated taking around 15 minutes in total for each test run. When the network was finally trained correctly, it was possible to load the weights into a .h5 file, so the images where available to play around with, show and threshold. However each image still contains a lot of data and predicting on more than 10 images at a time made the Jupyter notebook crash. This might be a hardware problem with this computer, but it unfortunately limited the results a bit.
The Slurm-GPU furthermore had the issue that it's installation of pip was outdated, and could not be upgraded without administrative access which was not provided for this project. This made it impossible to install certain packages for python including matplotlib and sk-learn. It would have been interesting to train the random forest classifier on the GPU as this would make it possible to provide much more training data, and discover how this classifier actually performs on these new annotations. That would make a clear comparison back to the previous project in order to really show that this new training data is of a much better quality.
It should also be mentioned that the thresholds on the final images, that is set to 0.5 is just a very intuitive, however a bit random, choice. It could be considered to change this threshold or maybe make it adaptive depending on the structure on a given image. It is very possible that different predictions might have a different "perfect" threshold depending on the brightness of the structure in the individual image.
However looking at the final results of accuracy, specificity and sensitivity it becomes very clear that the results are very satisfying, and that all in all this model is very safe to work with. This is also one of the reasons why none of the above issues have been resolved in this project.

11 How could we continue this project?

In connection with the bachelor thesis some pre-processing of the images was done before doing the actual segmentation. Different techniques were used to enhance the edges on the original images in order to try to make it easier for the classifier to segment out the structure. This proved to increase the score from around 92% to the final 97,5 % It is likely to think the same pre-processing could have been useful here. However it was to large for this project to reimplement this part since the images have changed size since the last project, and the code therefore needs a lot of rewriting. Furthermore the network already takes a while to train, and to much training data results in memory-bugs, so to include the edge enhanced images as well might result in the program only running on very few data images. The edge enhancing is not precise, and contain it's own problems as described in the previous project [2], and therefore the original images holds more value in this case and should be prioritized for the training.

12 Applications of neural networks in the real world

Application of neural networks and especially convolutional networks is very common in the industry of working with image processing. They can be used for recognizing certain objects and classifying them. This is very useful in many cases, and a very current example to mention could be self-driving cars where it is important to determine what is in front of the car in order to act accordingly.
In the biological imaging industry convolutional neural networks are used for almost everything that has to do with image processing. Most intuitive things to mention could be recognizing and segmenting out tumours and foreign objects in the body. Another example, that lies very close to the problem in this report is to segment out vessels in different parts of the body.
The U-net structure was developed in Germany in 2015 with the paper about segmentation of biological images. [10] It is used today to segment out different structures in biological images just as in this report. This could for example be segmenting out brain on MR scans or segmenting out the liver on an image of the intestines.
The U-net structure is in general very useful when the desire is to segment out certain structures and an ordinary convolutional network is used when

classification and recognizing of an object is the goal.

13 Conclusion

The goal of this project was to segment out structure on images in a dataset, and determine whether it was possible to improve the results from a different project with the same goal. The hypothesis was that because there is access to new and better training data, and because a neural network normally performs better than a classifier, that the results could be improved sufficiently.
The dataset was collected in a study of the mouse' pancreas and contained images of a certain structure that produces the beta-cells. Since the goal was to investigate the change of this structure over time, a neural network was implemented to solve this by segmenting out the structure on each of the images. In order to implement this neural network it has been studied how these work and how it should be constructed for this particular problem. It was concluded that a convolutional neural network with a U-net structure was the way to solve this. The results of the implementation in this study show a very impressive result both visually on figure 5 and 6 and theoretically with an accuracy of 99,4%, a specificity of 99,87% and a sensitivity of 81,14%. Therefore it is concluded that this study has been very successful, and have indeed improved the previous results.

References

[1] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015.

[2] C. K. Pedersen, "Preprocessing of images," 2019.

[3] D. Godoy, "Binary cross-entropy." `https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a`.

[4] G. Sanderson, "Video series about neural networks." `https://www.3blue1brown.com/`.

[5] Skymind.ai, "Convolutional neural networks." `https://skymind.ai/wiki/convolutional-network`.

[6] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[7] MIT, "U-net for segmenting seismic images with keras." https://www.depends-on-the-definition.com/unet-keras-segmenting-images.

[8] A. Collette, *Python and HDF5.* O'Reilly, 2013.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[10] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *CoRR*, vol. abs/1505.04597, 2015.

14 Appendix A - Batchscript

```
#!/bin/bash
#SBATCH --ntasks=1 --cpus-per-task=10 --mem=32000M
# we run on the gpu partition and we allocate 1 titanx gpu
#SBATCH -p gpu --gres=gpu:titanx:1
#We expect that our program should not run langer than 2 hours
#Note that a program will be killed once it exceeds this time!
#SBATCH --time=2:00:00
python code.py
```