Norwegian University
of Life Sciences

Master's Thesis 2019 30 ECTS
Faculty of Science and Technology

Fully convolutional neural network for semantic segmentation on CT scans of pigs

Jarand Hornseth Pollestad
Master of Science in Data Science

Abstract

Identifying the shape and location of structures within medical images is useful for purposes such as diagnosis and research. This is a cumbersome task if done manually. Recent advances in computer vision and in particular deep learning have made it possible to automate this task to such an extent that it is comparable to human level performance.
This thesis reviews the components used to construct a fully convolutional neural network for semantic segmentation. It then proposes a modified network architecture based on an existing state-of-the-art fully convolutional neural network called U-net.
The architecture is applied to a binary classification problem involving computed tomography scans of pigs provided by Norsvin SA. The goal is to classify each pixel in the scans as either "a part of the pig which is edible" or "background" which means everything that is not in the edible class.
Each computed tomography scan is too large for the network to process at once. Part of the thesis is therefore devoted to investigating approaches for feeding the information in the scans to the proposed network.
The network is trained on 238 scans and evaluated on 37 scans. The evaluation is done quantitatively using the index over union metric and qualitatively through manual inspection of segmented images. The results show that the best performing network on average obtains an index over union score of 0.962 when given a scan for segmentation.

Acknowledgements

I want to thank my thesis supervisors Oliver Tomic, Ph.D. and Kristian Hovde Liland, Ph.D. for guidance and feedback during this work. I would also like to thank my external supervisor Jørgen Kongsro, Ph.D. and Norsvin SA for providing the thesis topic, data, and guidance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Computer vision in medical image analysis

Medical imaging is the process of building visual representations showing the internal structure of biological systems such as the human body. Vast amounts of medical image data are being generated daily by techniques such as computed tomography (CT), magnetic resonance imaging (MRI), ultrasound and other medical imaging modalities.

Medical image analysis is the process of extracting clinically useful information from medical image data for purposes such as diagnosis and research [29]. It is not feasible for domain experts to manually analyze all the data due to the sheer quantity of data being created. Computer vision is becoming an increasingly important tool to aid medical image analysis, in part due to the deep learning revolution of the past decade.

Computer vision is a field of computer science concerned with giving machines visual perception, i.e. obtaining a high-level understanding of the contents in an image or a video. An area of computer vision that is of particular interest in this thesis is object recognition. Object recognition techniques are capable of recognizing objects in images and videos. It roughly encompasses the following subproblems:

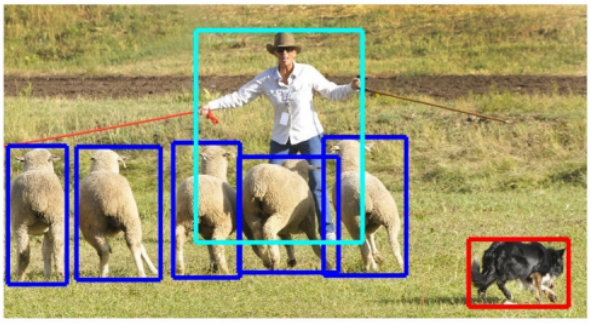
- Image classification: Assign a class label to an image based on its content, see figure 1.1a
- Object localization: Image classification + localization. Assign class labels to objects in the image and mark their location using a bounding box, see figure 1.1b
- Semantic segmentation: Pixel-wise dense prediction of an image, i.e. predicting a class label for each pixel in the image, see figure 1.1c.
- Instance segmentation: Pixel-wise dense prediction of an image while separating between instances of the same class, see figure 1.1d.

1.2 Semantic segmentation

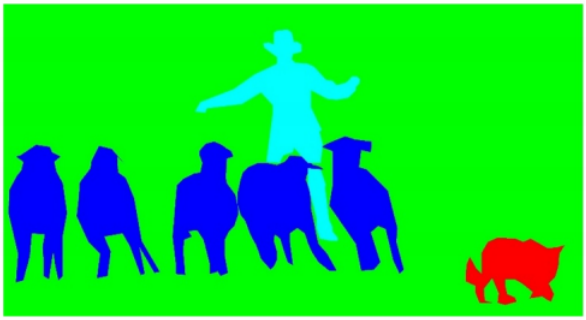
This thesis revolves around the use of convolutional neural networks (CNNs) to do semantic segmentation on CT scans of pigs. Segmentation is the process of partitioning image pixels into "coherent parts" or distinct non-overlapping regions. In medical image analysis this is useful for identifying the shape and boundaries of structures of interest [21].

Simple segmentation techniques partition the pixels using only the pixel intensity values. One example is the watershed segmentation algorithm [2]. This algorithm treats pixel intensity values as a description of elevation in a topographical map. Regions in the image with large pixel intensity values become peaks and regions with small pixel intensity values become valleys. Imagine then

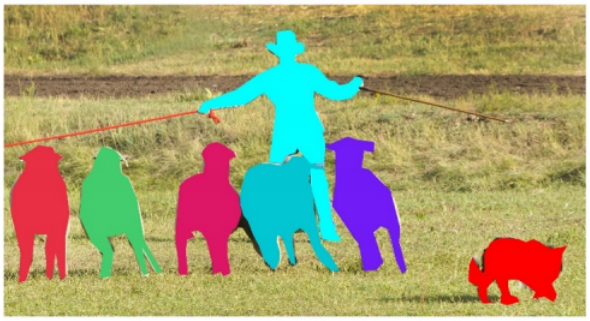
(a) Image classification



(b) Object localization



(c) Semantic segmentation



(d) Instance segmentation

Figure 1.1: Subproblems of object recognition. Image credit [18]

that each valley is gradually filled with water. Water from different valleys will inevitably merge. A separating line is drawn as these merge locations. The process continues until the entire image is flooded and all separating lines are drawn. Ideally, these lines mark the boundaries of objects in the images. However, there is no understanding of what these objects are.

Semantic segmentation attempts to partition the image pixels into semantically interpretable regions. A set of semantically meaningful class labels, for example, cow, grass, sky, etc. are first defined. Each pixel in the image is then assigned one of these class labels based on the content and structure of the image.

As mentioned earlier this thesis will use CNNs to do semantic segmentation. A CNN is a form of artificial neural network (ANN) especially suited for image related classification tasks. The construction of a CNN is described in detail in chapter 2. The CNN is trained using images and their respective segmentation masks (also known as ground truth images), see figure 1.2. Masks are images where each pixel is annotated with the correct class label. It serves as a blueprint for how to correctly classify each pixel in the image. During training, the CNN learns the relationship between the content in each input image and its respective mask. It can then use the learned relationship to predict masks when given new images.

1.3 Aim of thesis

- Utilize state-of-the-art fully convolutional network for semantic segmentation on CT scans of pigs.
- Investigate how to best deal with 3D images when doing semantic segmentation.
- Lay the groundwork for segmenting more complicated structures in the CT scans of pigs.

Figure 1.2: From the left: Input image, mask, predicted mask, input image overlayed with the predicted mask [32]

1.4 Outline of thesis structure

- Chapter 2 provides background knowledge about neural networks and a detailed description of the components used to carry out semantic segmentation using convolutional neural networks.
- Chapter 3 describes the images and masks used to train and evaluate the model.
- Chapter 4 describes the final model architecture and the training process.
- Chapter 5 presents experimental results.
- Chapter 6 is a qualitative evaluation of the model performance.
- Chapter 7 summarizes the work.

Chapter 2

Theory

The purpose of this chapter is to provide the theoretical background for the components used to construct a neural network for semantic segmentation. It will start with a basic building block of neural networks, namely the perceptron, and gradually build on this to construct a multilayer perceptron, a convolutional neural network, and finally a fully convolutional neural network.

2.1 Perceptron

The basic technical ideas of neural networks have been around for decades, but it is only in the last few years that we have seen a widespread application in products and research. Neural network performance is very dependent on large amounts of data, more so than traditional machine learning algorithms. The relatively recent digitization of society has made available a seemingly endless stream of data (sensors, images from smartphone cameras, tracking of online activity, etc). Another limiting factor of neural networks is the enormous number of computations required. This is naturally remedied by the tremendous increase in computational power that is now available to both professionals and amateurs through graphics processing units (GPUs) and deep learning cloud services. There have also been significant algorithmic improvements allowing for deeper networks which have been proven important for tasks involving computer vision, speech recognition, and natural language processing.

Artificial neural networks (ANNs) are inspired by how interconnected nerve cells in the brain (neurons) process- and transmit signals. The neurons consist of three main components, namely a cell body, dendrites, and an axon, see figure 2.1. Input signals from other neurons are received at the dendrites and aggregated within the cell body. If the input exceeds a threshold value the neuron generates an action potential. This action potential is output from the neuron through the axon and is used as input for other neurons [20].

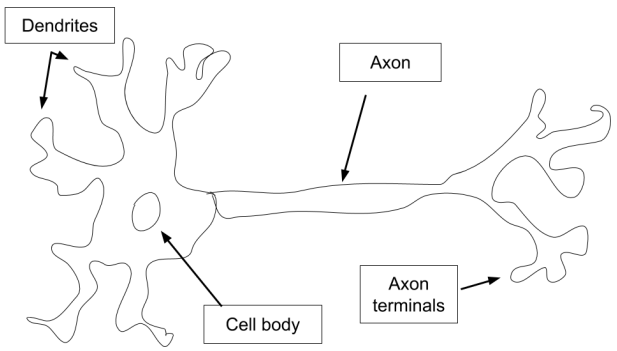


Figure 2.1: Neuron

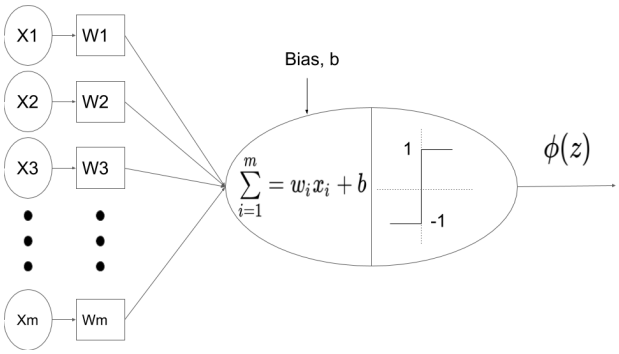


Figure 2.2: Perceptron

The neuron model served as inspiration for an electronic system, the perceptron, capable of recognizing similarities between patterns in information [25]. A perceptron is the simplest form of an artificial neural network and is capable of binary classification given linearly separable classes. A

perceptron can be seen in figure 2.2. The perceptron receives a vector of input values, $x = [x_1 \cdots x_n]$ representing a sample with n variables. These input values are multiplied with their respective weights, $w = [w_1 \cdots w_n]$ and a bias, b is added to produce the net-input, $z = \sum_{i=1}^{n} w_i x_i + b$. A step function, ϕ is then applied to the net-input. If the net-input is larger than zero the step function will assign the sample to class 1, otherwise, it will assign it to class -1 [23].

$$\phi(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ -1, & \text{otherwise} \end{cases} \tag{2.1}$$

Because the perceptron is a binary classifier it has two decision regions which are separated by a hyperplane defined by $z = 0$. The purpose of the bias is giving the perceptron the ability to translate, or shift, this hyperplane.
Optimal weights for predicting the correct class for a set of samples are initially unknown and must be learned. The end goal is to learn weights that best describes the relationship between the samples and their known classes. When these weights are learned the model can be used to predict the class of new samples (assuming the new samples has the same distribution as the original samples). Perceptrons use a process called stochastic gradient descent (SGD) to learn the set of weights that best separates the classes. The weights, w are typically initialized to some small random value. The perceptron receives an input vector $x^{(i)}$ for a sample, i and makes a prediction given w. The predicted class label, $\hat{y}^{(i)}$ is then compared to the true class label of the sample, $y^{(i)}$. Each weight, w_j in w is then updated using a learning rate parameter, η describing how large of an adjustment should be made to the weights:

$$w_j := w_j + \Delta w_j \tag{2.2}$$

Where:

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)} \tag{2.3}$$

The perceptron then receives a new sample and makes a prediction using the updated weights. This process is repeated until the smallest possible error is obtained. Because the output of the perceptron is a linear combination of the input, with an added step function, it will only be able to find a hyperplane that perfectly separates the classes if the classes are linearly separable [9].

2.2 Multilayer Perceptron

The perceptron unit is limited to binary output and simple linear classification tasks and is therefore not applicable to many real-world problems which are non-linear. However, these limitations can be overcome by making some small adjustments to the perceptron unit and then stacking them in a layered structure, thereby creating a multilayer perceptron (MLP). An MLP is composed of three or more layers. An input layer, hidden layers, and an output layer, see figure 2.3. An MLP with a single hidden layer can approximate any function given a sufficient number of units in the hidden layer [12].

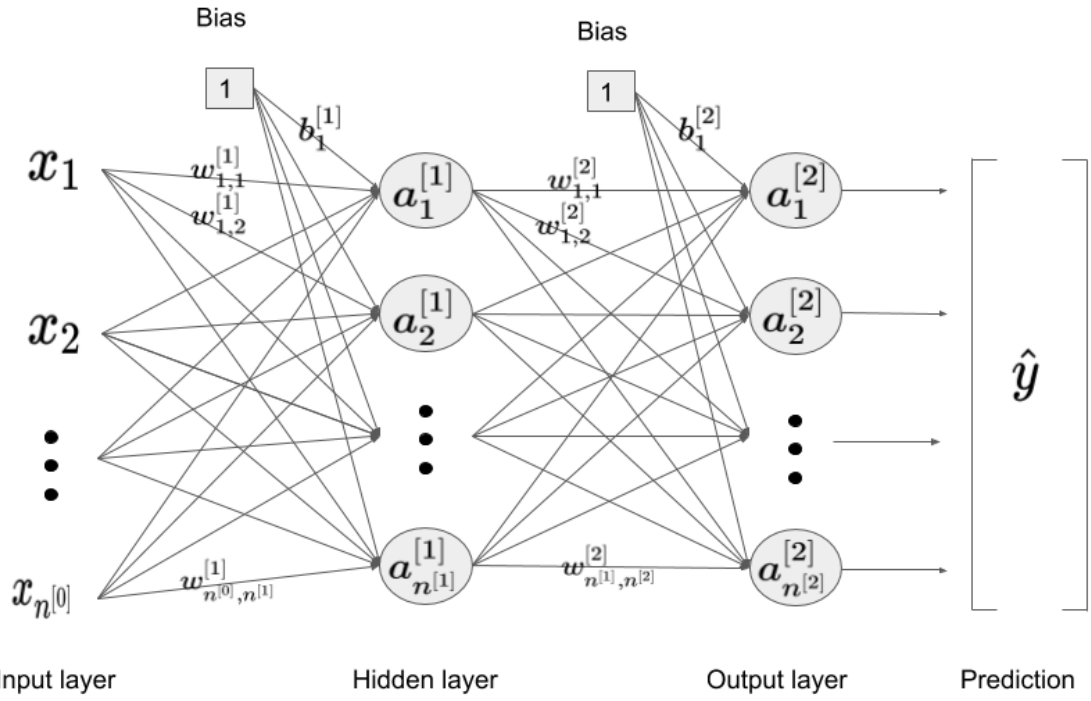


Figure 2.3: Multilayer perceptron with three layers where: a is an activation unit, w is the weight of a connection, n is the number of units in the layer, and b is the bias of a unit. The input layer is layer zero, the hidden layer is layer one, and the output layer is layer two. $a_2^{[1]}$ refers to the activation of unit number two, in layer number one. $w_{1,2}^{[1]}$ refers to the weight of the connection between unit one in layer zero and unit two in layer one.

Each perceptron unit in a layer receives input from all units in the previous layer. As before, a net-input is calculated using the input values in combination with the weights and bias. Instead of applying a step function to the net-input we apply some differentiable non-linear activation function. This function has to be differentiable due to the method which is used to update the weights, namely backpropagation, described in section 2.2.4, and it has to be non-linear to introduce non-linearity to the MLP, see section 2.2.6. The output from the activation function, called the activation of the unit, is then transmitted to all units in the subsequent layer. No connection is made between units in the same layer.

2.2.1 Forward propagation

Forward propagation is the process of calculating the output of a neural network such as an MLP. Given the MLP in figure 2.3 we can calculate the net-input, z of the units in layer one (the hidden layer):

$$z_1^{[1]} = w_{1,1}^{[1]} x_1 + w_{2,1}^{[1]} x_2 + \cdots + w_{n^{[0]},1}^{[1]} x_{n^{[0]}} + b_1^{[1]} \tag{2.4}$$

$$z_2^{[1]} = w_{1,2}^{[1]} x_1 + w_{2,2}^{[1]} x_2 + \cdots + w_{n^{[0]},2}^{[1]} x_{n^{[0]}} + b_2^{[1]} \tag{2.5}$$

$$\vdots$$

$$z_{n^{[1]}}^{[1]} = w_{1,n^{[1]}}^{[1]} x_1 + w_{2,n^{[1]}}^{[1]} x_2 + \cdots + w_{n^{[0]},n^{[1]}}^{[1]} x_{n^{[0]}} + b_{n^{[1]}}^{[1]} \tag{2.6}$$

The activation, a of the units in layer one is then calculated using some activation function, ϕ on the net-input:

$$a_1^{[1]} = \phi\left(z_1^{[1]}\right) \tag{2.7}$$

$$a_2^{[1]} = \phi\left(z_2^{[1]}\right) \tag{2.8}$$

$$\vdots$$

$$a_{n^{[1]}}^{[1]} = \phi\left(z_{n^{[1]}}^{[1]}\right) \tag{2.9}$$

By arranging the weights, activations, and biases in matrices we can express the net-input and activation of a layer, l using matrix multiplication:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{2.10}$$

$$A^{[l]} = \phi\left(Z^{[l]}\right) \tag{2.11}$$

where:

$$\underset{(n^{[l]},1)}{\boldsymbol{Z^{[l]}}} \quad \underset{(n^{[l]},n^{[l-1]})}{\boldsymbol{W^{[l]}}} \quad \underset{(n^{[l-1]},1)}{\boldsymbol{A^{[l-1]}}} \quad \underset{(n^{[l]},1)}{\boldsymbol{b^{[l]}}}$$

$$\begin{bmatrix} z_1^{[l]} \\ z_2^{[l]} \\ \vdots \\ z_{n^{[l]}}^{[l]} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{[l]} & \dots & w_{n^{[l-1]},1}^{[l]} \\ \vdots & \ddots & \\ w_{1,n^{[l]}}^{[l]} & & w_{n^{[l-1]},n^{[l]}}^{[l]} \end{bmatrix} \begin{bmatrix} a_1^{[l-1]} \\ a_2^{[l-1]} \\ \vdots \\ a_{n^{[l-1]}}^{[l-1]} \end{bmatrix} + \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix}$$

2.2.2 Prediction

The number of units in the output layer of a neural network denotes the number of classes we want the network to be able to recognize. In order to convert the net-input of the output layer into interpretable class probabilities it is common to use the softmax function:

$$\hat{Y}(z): \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \rightarrow \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_N \end{bmatrix}; \quad \hat{y}_j = \frac{e^{z_j}}{\sum_{k=1}^{n} e^{z_k}} \quad , \quad \forall_j \in 1 \dots n$$

The output is a vector of probabilities summing to one, where $\hat{y}_1$ is the probability that the sample belongs to class 1, $\hat{y}_2$ is the probability that the sample belongs to class 2, etc.

2.2.3 Loss function

An essential part of training any neural network is the loss function. The purpose of the loss function is to evaluate how well the network models the dataset. A popular loss function for classification problems is cross-entropy (CE).

$$CE(\hat{Y}, Y) = -\sum_i Y_i \log \hat{Y}_i \tag{2.12}$$

where Y is the true class label of the sample and $\hat{Y}$ is the predicted class label.

2.2.4 Gradient descent

After the network has done a prediction and the loss is computed we want to adjust the weights and biases of the network in order to reduce the loss. Gradient descent was briefly mentioned in section 2.1. It is a process used to find the global minimum of the loss function, meaning finding the weights and biases that give the smallest loss and therefore the most accurate prediction. Finding this global minimum is an iterative process where the derivative of the loss with respect to each weight and bias in the network is calculated. This derivative is often referred to as the gradient. Each weight and bias is then adjusted proportionally to the negative of its gradient before a new prediction is made. This continues until the metric used to measure performance reaches some predetermined value, or until the network can't improve anymore. The go-to method for calculating the gradients is **backpropagation** [26]. Backpropagation uses the chain-rule to compute the gradients for each layer in the network. Below is an example of how backpropagation calculates the gradients for the MLP in figure 2.3:

The first step is to calculate the derivative of the loss, in this case CE, with respect to the softmax input, which is the net-input of layer two, $Z^{[2]}$. [27] shows how this can be calculated as:

$$\frac{\partial CE}{\partial Z^{[2]}} = \hat{Y} - Y \tag{2.13}$$

resulting in a column vector of gradients that are used to calculate the gradients of the network parameters in earlier layers using backpropagation:

$$(n^{[2]}, 1) \qquad \frac{\partial CE}{\partial Z^{[2]}} = \hat{Y} - Y \tag{2.14}$$

$$(n^{[2]}, n^{[1]}) \qquad \frac{\partial CE}{\partial W^{[2]}} = \frac{\partial CE}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial W^{[2]}} = (\hat{Y} - Y) A^{[1]T} \tag{2.15}$$

$$(n^{[2]}, 1) \qquad \frac{\partial CE}{\partial b^{[2]}} = \frac{\partial CE}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial b^{[2]}} = \frac{\partial CE}{\partial Z^{[2]}} \tag{2.16}$$

$$(n^{[1]}, 1) \qquad \frac{\partial CE}{\partial A^{[1]}} = \frac{\partial CE}{\partial Z^{[2]}} \frac{\partial Z^{[2]}}{\partial A^{[1]}} = W^{[2]T} (\hat{Y} - Y) \tag{2.17}$$

$$(n^{[1]}, 1) \qquad \frac{\partial CE}{\partial Z^{[1]}} = \frac{\partial CE}{\partial A^{[1]}} \frac{\partial A^{[1]}}{\partial Z^{[1]}} = W^{[2]T} (\hat{Y} - Y) \odot \frac{\partial \phi(Z^{[1]})}{\partial Z^{[1]}} \tag{2.18}$$

$$(n^{[1]}, n^{[0]}) \qquad \frac{\partial CE}{\partial W^{[1]}} = \frac{\partial CE}{\partial Z^{[1]}} \frac{\partial Z^{[1]}}{\partial W^{[1]}} = \left(W^{[2]T} (\hat{Y} - Y) \odot \frac{\partial \phi(Z^{[1]})}{\partial Z^{[1]}} \right) A^{[0]T} \tag{2.19}$$

$$(n^{[1]}, 1) \qquad \frac{\partial CE}{\partial b^{[1]}} = \frac{\partial CE}{\partial Z^{[1]}} \frac{\partial Z^{[1]}}{\partial b^{[1]}} = W^{[2]T} (\hat{Y} - Y) \odot \frac{\partial \phi(Z^{[1]})}{\partial Z^{[1]}} \tag{2.20}$$

The weights and biases in layer l can then be updated:

$$W^{[l]} = W^{[l]} - \eta \frac{\partial CE}{\partial W^{[l]}} \tag{2.21}$$

$$b^{[l]} = b^{[l]} - \eta \frac{\partial CE}{\partial b^{[l]}} \tag{2.22}$$

where η is the learning rate describing how large of an adjustment should be made to the weights and biases given the gradients.

2.2.5 RMSprop

There are potentially large differences in the magnitude of gradients across the network. A learning rate that is suitable for adjusting weights and biases in one part of the network may be too small or too large for other parts of the network. If a learning rate is too large the network may diverge from the optimal solution, meaning it will overshoot the weights and biases that give the smallest possible loss (given the network architecture and input data). If the learning rate is too small it will take longer to train the network because the network only makes minor adjustments to the weights and biases during each iteration of gradient descent. It also increases the risk of getting stuck in a sub-optimal solution. A common analogy for gradient descent is a mountainous landscape with peaks and valleys. The optimal weights and biases that give the smallest possible loss are located in the deepest valley, i.e. the global optimum. However, we may at some point during training find ourselves in a shallower valley, a local optimum, surrounded by smaller peaks and without the step size (learning rate) required to get across the peaks and see what's on the other side. We could then mistakenly assume that the local optimum is the global optimum and that no improvements can be made to the weights and biases, which is a sub-optimal solution. For these reasons it is difficult to set a global learning rate that copes with all local issues.

RMSprop is an optimization algorithm which addresses this problem by dividing each gradient by a moving average of its recent gradient magnitudes [11], see equation 2.25 and 2.26, thus scaling the weight and bias updates. This allows us to use large learning rates which increases learning speed while reducing the risk of overshooting the global optimum solution.

RMSprop algorithm

Given some iteration, t of the training process calculate the cross-entropy loss, CE (or any other loss). Use backpropagation to compute the derivative of CE with respect to all weights and biases in the network as was done in section 2.2.4, i.e. compute all $\frac{\partial CE}{\partial w}(t)$ and $\frac{\partial CE}{\partial b}(t)$. The moving average, S for time-step, t is then calculated as:

$$S(t)_{\frac{\partial CE}{\partial w}} = \beta * S(t-1)_{\frac{\partial CE}{\partial w}} + (1-\beta)\left(\frac{\partial CE}{\partial w}(t)\right)^2 \tag{2.23}$$

$$S(t)_{\frac{\partial CE}{\partial b}} = \beta * S(t-1)_{\frac{\partial CE}{\partial b}} + (1-\beta)\left(\frac{\partial CE}{\partial b}(t)\right)^2 \tag{2.24}$$

where β is a parameter specifying how much earlier iterations of the moving average should affect the current iteration. The recommended value for β is 0.9.

The weights and biases can then be updated:

$$w = w - \eta \frac{\frac{\partial CE}{\partial w}}{\sqrt{S(t)_{\frac{\partial CE}{\partial w}}}} \tag{2.25}$$

$$b = b - \eta \frac{\frac{\partial CE}{\partial b}}{\sqrt{S(t)_{\frac{\partial CE}{\partial b}}}} \tag{2.26}$$

2.2.6 Activation functions

As mentioned in section 2.2 the purpose of including non-linear activation functions in the network is to introduce non-linearity to the network, thereby allowing the network to learn a non-linear function. If we instead used a linear activation function, $\phi(z) = z$ between two arbitrary layers l and $l + 1$ in a MLP then:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]} \tag{2.27}$$

$$A^{[l]} = \phi(Z^{[l]}) = Z^{[l]} \tag{2.28}$$

$$Z^{[l+1]} = W^{[l+1]}A^{[l]} + b^{[l+1]} \tag{2.29}$$

$$= W^{[l+1]}Z^{[l]} + b^{[l+1]} \tag{2.30}$$

$$= W^{[l+1]}\left(W^{[l]}A^{[l-1]} + b^{[l]}\right) + b^{[l+1]} \tag{2.31}$$

$$= \left(W^{[l+1]}W^{[l]}\right)A^{[l-1]} + \left(W^{[l+1]}b^{[l]} + b^{[l+1]}\right) \tag{2.32}$$

$$= W^{*}x + b^{*} \tag{2.33}$$

$$A^{[l+1]} = \phi(Z^{[l+1]}) = W^{*}x + b^{*} \tag{2.34}$$

The network will only be able to output a linear function, regardless of network depth and width. This is also the case if there are no activation functions. Non-linear activation functions are therefore necessary if we want to deal with data that isn't linearly separable as is the case in many real-world problems.

One such non-linear activation function, the logistic sigmoid function, see figure 2.4, is often seen applied to the output layer of a neural network used for binary classification. It takes an input value and outputs a value in the range $[0, 1]$. This value can be interpreted as the probability that the sample belongs to the positive class given the weights and biases parameterizing the network. We can then select a threshold, say 0.5, and decide that samples with probability greater than 0.5 are assigned the positive class and samples with probability less than, or equal to, 0.5 are assigned the negative class.

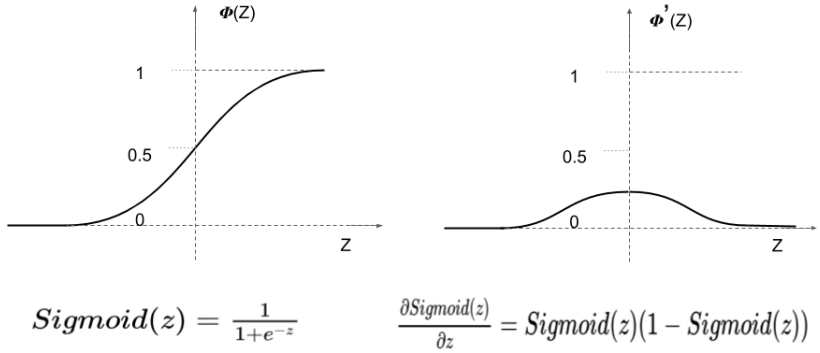


Figure 2.4: The figure illustrates the sigmoid function on the left and the derivative of the sigmoid function on the right.

One drawback of the sigmoid function is that it saturates quickly at either tail 0 or 1. The gradient of the sigmoid function in these regions approaches zero. From equation 2.18, 2.19, and 2.20 we

can see that during backpropagation these (local) gradients are multiplied with the prediction error that is being propagated backwards. If the gradient is too small the error signal will greatly diminish thereby reducing the networks ability to update parameters in earlier layers. This is referred to as the vanishing gradient problem. It is therefore recommended not to use the sigmoid function in intermediate layers of a network.

A popular alternative in recent years that addresses the issue of vanishing gradients is the Rectified Linear Unit (ReLU)[22], see figure 2.5. A network using ReLUs was shown to reach 25 % training error on CIFAR-10 up to six times faster than when sigmoid/tanh activation functions were used in an equivalent network [30]. CIFAR-10 is a popular dataset used to benchmark the performance of many computer vision algorithms. The ReLU function can be applied using a computationally inexpensive thresholding operation on the net-input matrix. Because the gradient is 1 for all positive values it reduces the vanishing gradient problem.

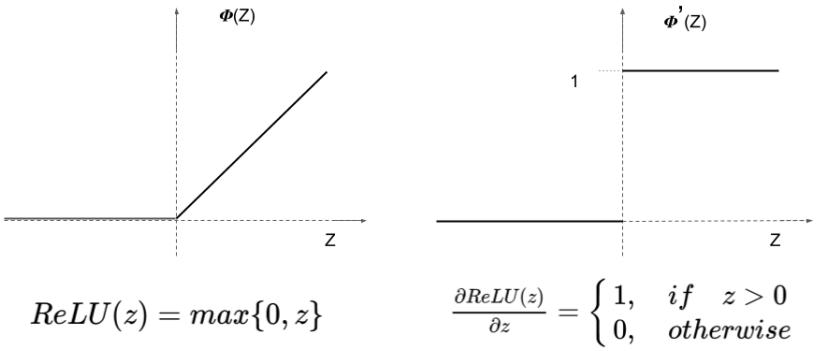


Figure 2.5: The figure illustrates the ReLU function on the left and the derivative of the ReLU function on the right.

2.3 Convolutional Neural Networks

While MLPs can achieve remarkable results they do not scale well for image classification tasks. Take for example a 512x512 RGB image, i.e. an image with three color channels totaling $512 * 512 * 3 = 786432$ input pixels. If such an image is used as input to an MLP with ten neurons in a single hidden layer followed by an output layer with one neuron we would need to train/update $786432 * 10$ weights and 10 biases in the hidden layer and an additional 10 weight and 1 bias in the output layer, totaling ≈ 7.86 million parameters. A realistic network for image classification would have more layers and neurons thus making MLP networks with full connectivity for image classification unfeasible.

Convolutional Neural Networks (CNN's) are a subclass of neural networks. They have some properties that allow them to retain spatial information and significantly reduce the number of trainable parameters (compared to MLPs), making them very suitable for image classification tasks. A key distinguishing element between CNN's and MLPs is the convolution layer. In a convolution layer, the convolution operation is used instead of matrix multiplication to compute neuron activations. The activations are passed from one layer to the next, as in an MLP, and prediction error is backpropagated to update the parameters. An example of a CNN architecture called LeNet-5 can be seen in figure 2.6.

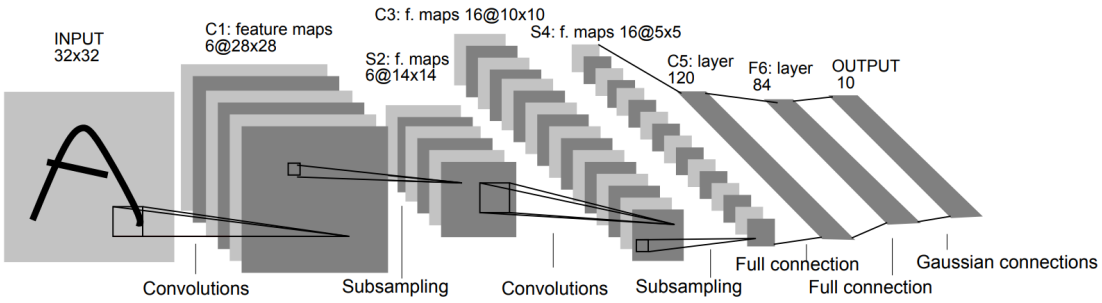


Figure 2.6: The LeNet-5 CNN architecture was used for handwritten character recognition. An image of a character is passed through as series of convolution and pooling (subsampling) operations and subsequently classified using fully connected layers. Image credit [14]

2.3.1 Convolution layer

A key component in a convolutional layer is the filter (also known as a kernel). Examples of filters can be seen in figure 2.7. These filters contain the trainable parameters, or weights, of a convolution layer. A layer can have multiple filters, where each filter is typically spatially small (height and width) and has the same depth as the input volume to the layer, i.e. the same number of channels. Each filter is moved spatially across the input volume taking dot products between each input volume channel and its respective filter channel, and subsequently summing the results and adding a bias. An illustration of this operation with arbitrary input values and weights can be seen in figure 2.8. Notice how the output of a convolution operation only has one channel per filter, regardless of how many channels the filter itself contains. The output is called a feature map. The number of feature maps a convolution layer produces correspond to the number of filters in the layer. Each element in the feature map is only connected to a local patch in the height and width dimension, but along the entire depth dimension, of the input volume, see figure 2.8. This patch is the receptive field of the element and is determined by the filter size.

The feature map contains the response of the filter at the various spatial locations in the input volume. It's intuitive to think of a filter as a feature identifier. A strong response indicates a strong presence of the feature described by the filter within the receptive field the filter is looking at. This could be a color change, an edge or something else the network deems useful. The number of filters in a convolution layer is a design choice. A large number of filters allows the network to learn more feature identifiers but it also increases the number of trainable parameters. The feature maps output by a convolution layer are stacked and used as input for the next layer in the network.

A CNN will typically contain multiple convolution layers. During training, the CNN will construct a feature hierarchy where the filters in the earlier layers of the network learn to identify simple features and subsequent filters learn to identify increasingly complex features [34].

Take for example the LeNet-5 architecture in figure 2.6. Its purpose was to classify handwritten characters. Taking into account upper- and lower case variations of letters, numbers, and other characters it had to distinguish between a large number of classes. This means that during training the network had to learn a set of complex features that are useful for separating between these classes. Given a new image of a character the convolution layers in the network would then produce a set of feature maps describing which of these complex features are present in the input image.

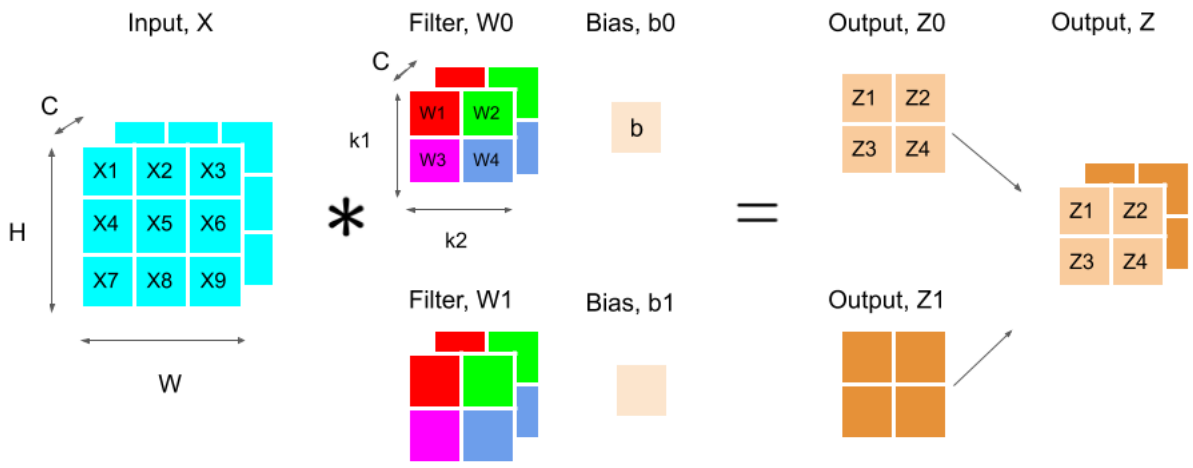


Figure 2.7: Two filters, W0 and W1, of size 2x2 is convolved with the input, X using a stride of 1x1. The output of the operation is one feature map per filter. These are stacked and used as input for the next operation in the CNN

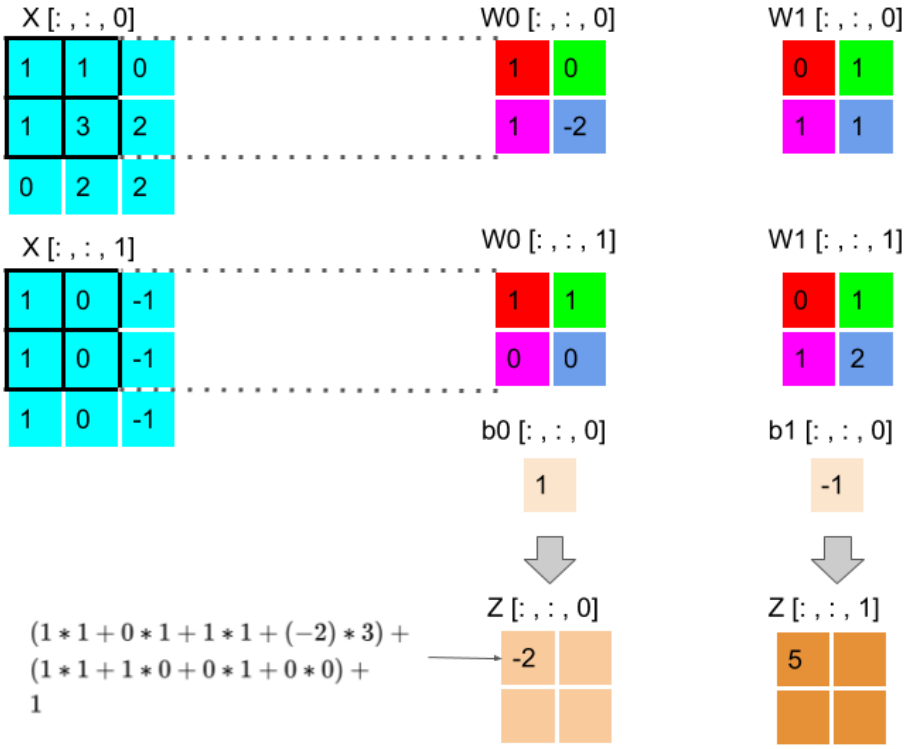


Figure 2.8: The figure shows how an element of the output feature map, Z is calculated by doing element-wise multiplication between a filter, W and an input, X and adding a bias, b

A convolution layer with n filters where each filter has height, k_1 width, k_2 and depth, C has $(k_1 * k_2 * C + 1) * n$ trainable parameters.

The dimensions of the stack of feature maps produced by a convolution layer are determined by five parameters, namely the dimensions of the input volume, filter size, number of filters, stride, and padding.

The stride is the step size of the filter movement across the input. If for example the input is an image and the stride is two, then the filter will do a convolution operation at a location and then move two pixels before performing the next operation. If the stride is equal to the filter size then each pixel is used for computation once by each filter.

In figure 2.7 and 2.8 it can be seen that the elements at the borders of the input will not be included in as many convolution operations as elements closer to the center. The corner elements will only be used one time by each filter, while the element in the center will be used four times by each filter, thus information at the border is lost. In figure 2.7 and 2.8 we can also see that the output is smaller than the input. This is because the filters can only fit twice in both the height and width dimension of the input, given a stride of one. If an input were to be passed through dozens of convolution layers its size will rapidly decrease. In order to preserve information at the borders and control output size, it is common to use **zero-padding**. The input volume is then padded with zeros along the borders, see figure 2.9.

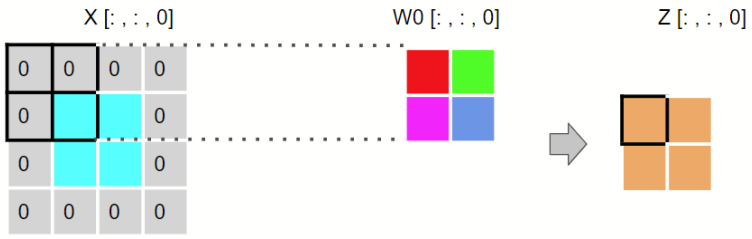


Figure 2.9: A convolution operation with filter size 2x2 and stride 2x2. Zero-padding of the input matrix allows for convolution without a reduction in dimension. The output has the same dimension as the input. It also improves performance along the boundary of the input.

The width of the output volume of a convolution layer can now be calculated as:

$$\text{Output width} = \left\lfloor \frac{\text{Input width} + 2 * \text{Padding} - \text{Filter width}}{\text{Stride}} \right\rfloor + 1 \tag{2.35}$$

similarly, the output height is calculated using input height and filter height.

The convolution layer gives the CNN some characteristics that make it more suitable than MLPs for image related tasks:

Sparse connectivity

In an MLP each neuron in a layer is connected to all neurons in the previous layer, see figure 2.10, while in a CNN each neuron is only connected to its receptive field of neurons in the previous layer, see figure 2.11. This drastically reduces the number of computations required during forward and backward propagation and the number of parameters that have to be stored.

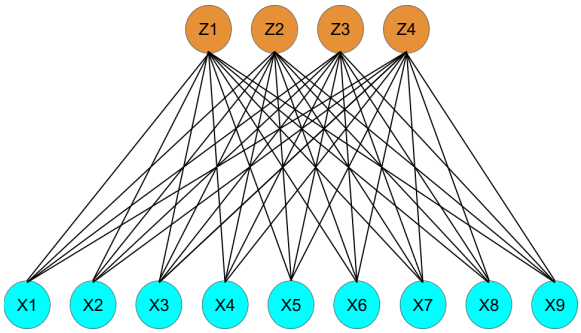


Figure 2.10: Full connectivity between layers. Each unit in output, Z is connected to all units in input, Z

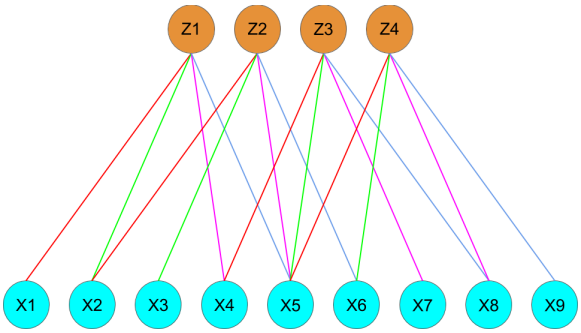


Figure 2.11: Sparse connectivity between layers. Each unit in output, Z is only connected to the units in input, X from which it was computed. From figure 2.7 we can see that Z1 was computed using 4 elements in the upper left quadrant of the input, namely X1, X2, X4, and X5. The number of connections from each Z corresponds with the size of the filter.

Parameter-sharing

Figure 2.11 illustrates how the parameters of a filter, shown as colored lines, are reused as the filter passes over the input feature map. The reasoning for this is that the filter is a feature identifier, and if a feature is useful to compute at one location in the feature map, then it is likely useful to compute at some other location in the feature map. This results in a substantial reduction in the number of parameters that have to be stored.

Equivariance to translation

Translation of a structure (all pixels is shifted the same amount in the same direction) in the input feature map will result in an equivalent translation in the output feature map, see figure 2.12. It is however not equivariant to rotation, see figure 2.13. This is naturally a problem since an object doesn't stop being the same object just because it's pictured at an angle. It is therefore common to train a model using rotated versions of the same image so the model learns different representations of the same objects.

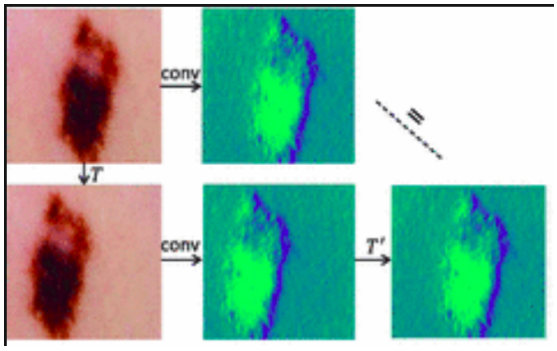


Figure 2.12: An image of a skin lesion is convolved with a filter to create a feature map. The same filter is convolved with a translated version of the same skin lesion. The feature map created by the latter operation is identical to the first one if we reversed the translation [17].

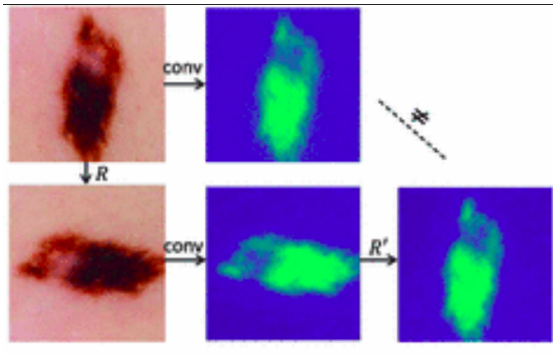


Figure 2.13: An image of a skin lesion is convolved with a filter to create a feature map. The same filter is convolved with a rotated version of the same skin lesion. If we look closely we can see that the feature map created by the latter operation is not exactly the same as the first one when the rotation is reversed [17].

2.3.2 Pooling layer

A layer where every unit in the input is replaced using some summary statistic on itself and neighboring units is called a pooling layer. Commonly used in CNN's is max pooling [35], see figure 2.14. Immediately notable is the dimension reduction. This reduces the number of parameters that need to be stored. Another benefit is the added invariance to local translations in the input. Small shifts in the locations of a feature will have less effect on the outcome of the classification task. The exact position of the largest element within the upper left quadrant of figure 2.14 is irrelevant. A pooling layer has no trainable parameters.

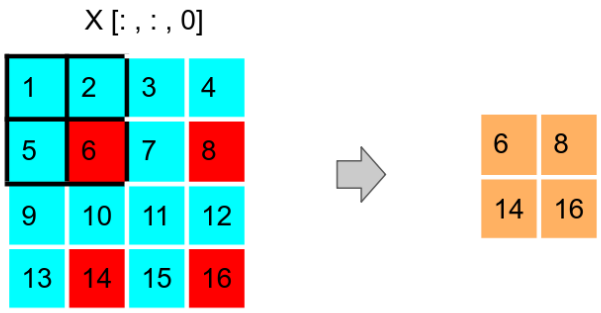


Figure 2.14: A max pool operation with filter size 2x2 and stride 2x2. A 4x4 matrix is reduced to a 2x2 matrix. Each element in the output matrix is the largest value in the corresponding quadrant of the input matrix

2.3.3 Transposed convolution layer

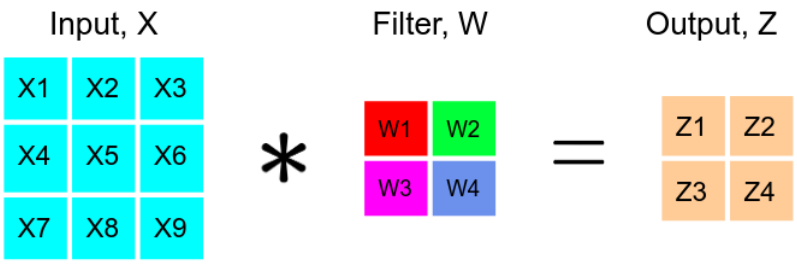


Figure 2.15: Convolving a 3x3 image with a 2x2 filter without zero-padding the image.

There are instances where we wish to upsample feature maps from low resolution to high resolution. This is especially important in semantic segmentation where we want to produce a segmented image of the same dimension as the input image. The need for upsampling for semantic segmentation is discussed in detail in section 2.3.7. Figure 2.16 show some techniques that can be used to do upsampling of a feature map that don't require trainable parameters. These techniques only depend on the content of the feature maps and the desired factor of upsampling.

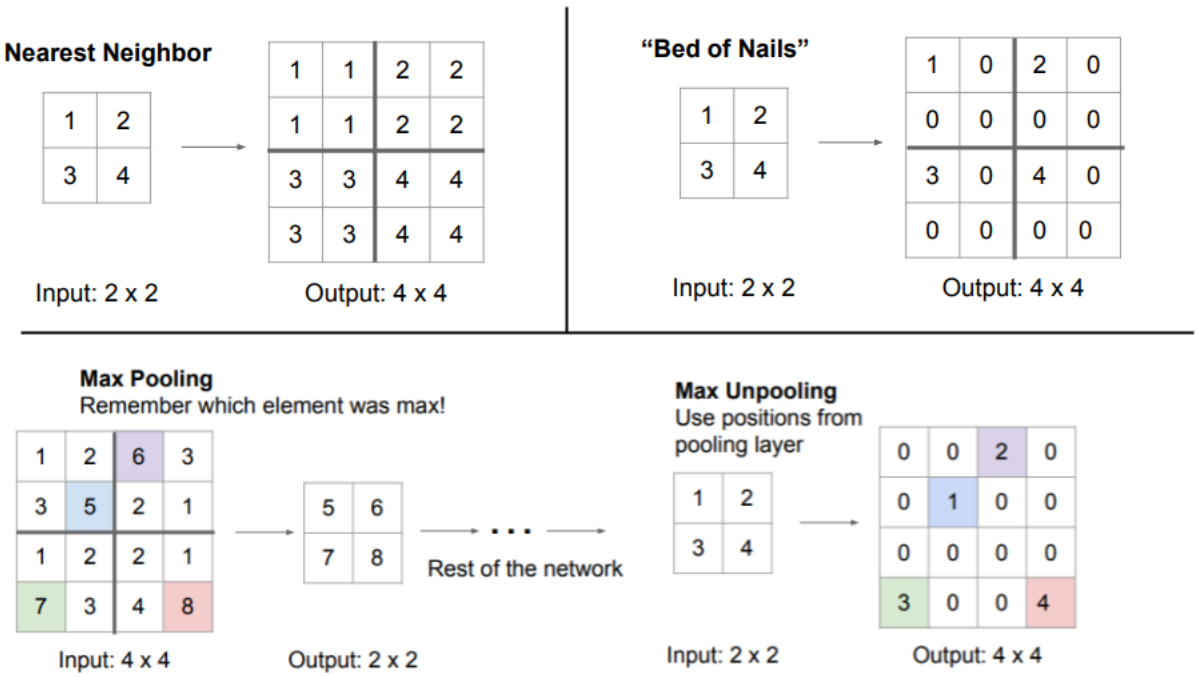


Figure 2.16: Techniques for upsampling images that don't require trainable parameters. Image credit [16].

Alternatively, we can use transposed convolution, also known as fractionally-strided convolution, which is a technique for upsampling that has trainable parameters. In other words, it tries to learn the optimal, or most correct, upsampling while the model is being trained. Take for example a convolution operation between a single channel 3x3 image with a 2x2 kernel using a stride of 1x1 and no padding resulting in a 2x2 output, see figure 2.15. This operation can be expressed as a convolution matrix, C:

$$\underset{\textbf{(4,9)}}{\underbrace{\begin{bmatrix} w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 \\ 0 & 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 \end{bmatrix}}_{\boldsymbol{C}} \underset{\textbf{(9,1)}}{\underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix}}_{\boldsymbol{Input}}} = \underset{\textbf{(4,1)}}{\underbrace{\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix}}_{\boldsymbol{Output}}}$$

A transposed convolution operation is simply a convolution operation using a transposed convolution matrix, C^T. The 2x2 output is then upsampled to a 3x3 matrix.

$$\underset{\boldsymbol{C^T}}{\begin{bmatrix} c_1 & 0 & 0 & 0 \\ c_2 & c_1 & 0 & 0 \\ 0 & c_2 & 0 & 0 \\ c_3 & 0 & c_1 & 0 \\ c_4 & c_3 & c_2 & c_1 \\ 0 & c_4 & 0 & c_2 \\ 0 & 0 & c_3 & 0 \\ 0 & 0 & c_4 & c_3 \\ 0 & 0 & 0 & c_4 \end{bmatrix}} \underset{\boldsymbol{Input}}{\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix}} = \underset{\boldsymbol{Output}}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix}}$$

$$\underset{\mathbf{(9,4)}}{} \qquad \underset{\mathbf{(4,1)}}{} \qquad \underset{\mathbf{(9,1)}}{}$$

Note that the weights in C and C^T are not the same, which is why the new weights are given the notation c. It is merely to show that we can do upsampling using a convolution operation. The important part is the connectivity pattern of C^T. It describes how individual pixels in the input is contributing to multiple pixels in the larger output [6]. A one-to-many relationship as opposed to the many-to-one relationship we see during a standard convolution operation.

2.3.4 Activation layer

The activation layer of a CNN serves the same purpose as described in section 2.2.6, which is to introduce non-linearity to the network. It is typically applied after a convolution layer where it applies a point-wise non-linearity to each element in the feature maps. An activation layer has no trainable parameters.

2.3.5 Batch normalization layer

In a neural network the output of the first layer serves as input for the second layer and the output of the second layer serves as input for the third layer etc. During training the weights and biases parameterizing the network change. The result is that during each forward pass a layer is likely to output activations with a different distribution compared to the last forward pass. A change in distribution in earlier layers can be amplified as it propagates through the network. This change in distribution is called internal covariate shift. Batch normalization [13] has become common in most CNN architectures. The authors argue that it accelerates learning because it reduces internal covariate shift. This reportedly makes the network more robust, allows the use of larger learning rates (thus faster convergence) and makes it less susceptible to the vanishing gradient problem described in section 2.2.6.

It is common in deep learning to feed more than one image into the networks at each forward pass. This set of images is referred to as a mini-batch and it allows us to train the model faster. The 4D tensor containing the input images can be written as $[B, H, W, C]$ where B is number of images in the batch, H, W is the height and width of the images, and C is the number of channels in the image.

The batch normalization algorithm for CNN's first computes the mean value, μ_β from all values, x_i in the batch across all spatial locations for each channel. In other words, C mean values, μ_β are computed and each μ_β is based on $B * H * W$ values, see equation 2.37. μ_β is then used to calculate the variance, μ_β^2, see equation 2.38. μ_β and μ_β^2 is then used to normalize each of the $B * H * W$ values in the batch, see equation 2.39. The normalization of values can reduce the expressive power of the CNN [8]. Two trainable parameters, γ and β, are therefore added to the normalized values, see equation 2.40. γ and β lets the network scale and shift each normalized value, thus giving the network the same expressive power as it had before normalization. The difference is that the mean and variance of each normalized value is now determined solely by γ and β instead of the

distribution of the input to the layer.

$$\mu_\beta = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.36}$$

$$\sigma_\beta^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\beta)^2 \tag{2.37}$$

$$\hat{x}_i = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}} \tag{2.38}$$

$$y_i = \gamma \hat{x}_i + \beta \tag{2.39}$$

$$\tag{2.40}$$

Two trainable parameters, γ and β, and two non-trainable parameters, μ_β and μ_β^2, has to be computed for each C in a batch normalization layer.

2.3.6 Residual connections

The depth of a CNN (number of convolution layers) has a significant impact on its performance. More layers are generally better as it allows the network to build a richer feature hierarchy [28]. One problem with going deeper is that gradients have to propagate through more layers when the network is training, and we're once again faced with the vanishing gradient problem. This can be mitigated, but not solved, through batch normalization. Another issue that arises is the degradation problem. The accuracy of deep CNN's has been shown to be worse than shallow CNN's [10], see figure 2.17.

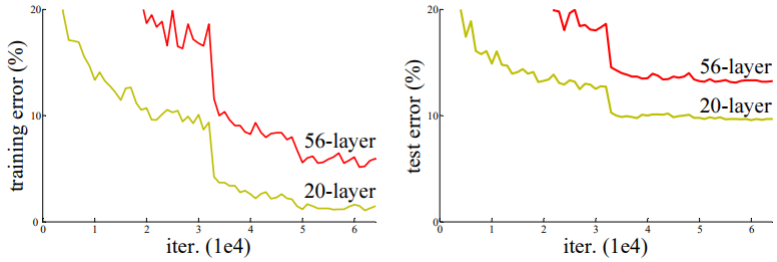


Figure 2.17: Deep CNN is performing worse than shallow CNN when trained on the CIFAR-10 dataset. Image credit [10]

Take for example a shallow CNN performing at some level. If we increase the model by k layers we would expect the deeper network to perform at least as well as the shallow network because the k new layers could simply learn an identity mapping. That is to say, they could learn to output their input without any changes and thereby match the performance of the shallow network. This does not happen in practice as the networks seem to have difficulties learning these identity mappings.

Residual connections, also referred to as short skip-connections, allows information to bypass one or more convolution layers (or any other layers). The information, in the form of feature maps, is added to the output of the convolution layer which is another set of feature maps, see figure 2.18. A requirement for this operation is that the feature maps being added has the same spatial dimensions. This is achieved by zero-padding the input to the convolution layers and adjusting the number of filters in accordance with the number of feature maps going into the convolution layers. The residual connection is the equivalent of an identity mapping and allows for the construction of very deep CNN's.

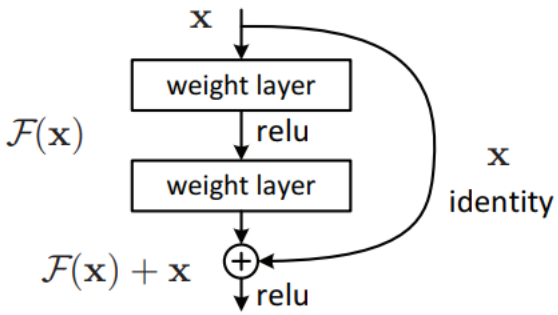


Figure 2.18: A weight layer denotes some layer modifying the input (for example a convolution layer). A residual connection, x bypasses the weight layers and adds the unmodified information to the output of the weight layers. Image credit [10]

2.3.7 Fully connected layer

As seen in figure 2.6 a typical CNN for classification consists of a series of convolution, activation, and pooling layers followed by a couple of fully connected layers. As mentioned in section 2.3.1 the series of convolution and pooling layers produce a set of feature maps describing which complex features are present in an input image (out of a set of learned complex features the network considers useful for separating between classes in the dataset). Different images have different combinations of complex features. During training, the fully connected layers learn to associate different combinations of complex features with the different classes.

The fully connected layers use the feature maps to output a vector of class probabilities as discussed in section 2.2.2. Because of the dense layers all spatial information about the location of features is lost. This is not an issue when the objective is to predict a single class label for an entire image. It is however an issue when doing semantic segmentation where the goal is a pixel-wise dense prediction, i.e. predicting a class label for each pixel in the input image.

2.4 Semantic segmentation using a convolutional neural network

Although problematic, it is possible to use the classical CNN architecture for semantic segmentation by using a sliding-window approach [4]. A pixel is then classified using the window- or patch of pixels surrounding it. The window used to classify neighboring pixels will be slightly different, and the window used to classify a pixel far away is likely to be very different, see figure 2.19. The network will then learn to classify the pixels depending on the content of the window. Some form of padding is necessary when part of the window of a pixel is outside the boundary of the image.

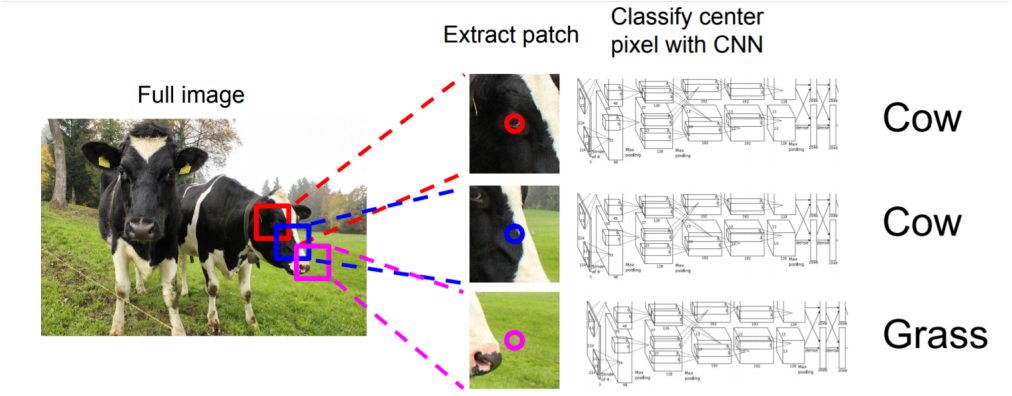


Figure 2.19: Sliding window for doing semantic segmentation using a conventional CNN. Image credit [16]

This approach has a few drawbacks, the most obvious being that it's computationally expensive

to classify each pixel separately. There is also a lot of redundancy due to the overlapping windows of neighboring pixels. In addition, there is a loss of context due to the limited size of the window. A larger window will require more pooling operations which decreases the localization accuracy, thereby making it difficult to accurately draw the borders between classes.

2.4.1 Fully convolutional neural network

A much more efficient approach is to utilize a fully convolutional network (FCN). The FCN takes the entire image as input and classifies all pixels during a single forward pass. The use of FCN's for semantic segmentation was popularized by Long et al. [19] where they used an encoder-decoder structure. As an encoder they used the VGG-16 network [28], see table 2.1, but without the fully connected layers at the end. The output from the encoder is a set of feature maps describing which complex features are present in the input image. However, these feature maps have a much smaller spatial dimension, i.e. image resolution, than the input image due to the pooling layers. These feature maps have to be upsampled, or scaled up, in order to obtain a dense prediction of the input image. This takes place in the decoder.

Convolution block	**VGG-16**
Convolution layer	Input image
Activation layer	Convolution block
Convolution layer	Max pool
Activation layer	Convolution block
	Max pool
	Convolution block
	Max pool
	Convolution block
	Max pool
	Convolution block
	Max pool
	Fully connected layer
	Fully connected layer
	Fully connected layer
	Prediction

Table 2.1: VGG-16 architecture

The decoder takes as input the stack of feature maps output by the 5th and final, max pool operation in the encoder. It then applies a 1x1 convolution layer which is a convolution layer where N 1x1xC filters are used. C is the number of feature maps in the stack and N is the number of class labels. This layer compresses the C feature maps into N feature maps. Each of these feature maps can be interpreted as a heatmap for one of the N classes.

These N feature maps are then upsampled to the same spatial dimension as the output from the 4th max pool using a transposed convolution layer. The output from the 4th max pooled is compressed into N feature maps using a 1x1 convolution layer and subsequently summed with the upsampled feature maps through skip connections. The process is then repeated using the output from the 3rd max pool. The output from this operation is then upsampled until it has the same spatial dimension as the input image.

We now have N feature maps of the same spatial dimension as the input image, each feature map representing a class label. By applying the softmax function across the feature maps for each pixel we obtain class probabilities, see figure 2.20.

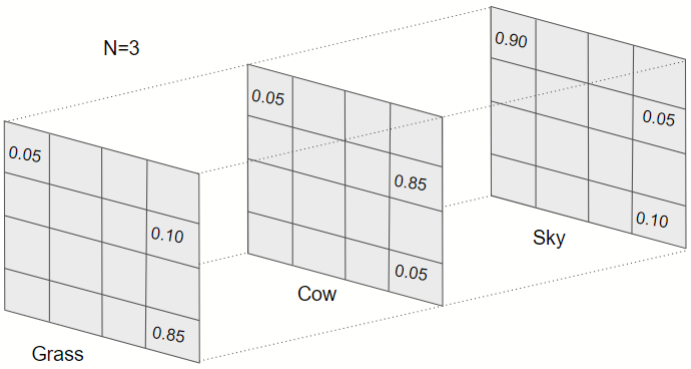


Figure 2.20: Example of prediction map created by FCN-8 for a segmentation problem with 3 classes

The intuition behind the skip connections is that the feature maps created deep in the network have a good understanding of the content in the image, but lack spatial information due to the pooling operations. By adding skip connections from earlier in the network (fewer pooling operations) we provide the feature maps with the spatial information necessary to accurately predict both the classes in the image and their location. We combine the "what" with the "where". This architecture is called FCN-8.

Ronneberger et al. [24] used the insight gained from FCN-8 to create what is commonly known as the U-Net architecture. The U-net have skip connections at every pooling stage, unlike FCN-8 who only had them at the bottom three. The decoder therefore resembles a mirrored version of the encoder, see figure 2.21.

The U-net does not utilize 1x1 convolution layers before each skip connection. The output stack of feature maps from the encoder is upsampled, using transposed convolutions, to the same spatial dimension as the previous pooling step without undergoing compression. The output feature maps from the previous pooling step are then concatenated onto the upsampled stack of feature maps. The resulting stack is used as input for two convolution layers that reduce the number of feature maps and learns how to combine the "what" with the "where". The result is upsampled again and the process repeated until the spatial dimension is equal to the input image. A 1x1 convolution layer and softmax function is then used to create a prediction map as seen in figure 2.20. By increasing the number of skip connections and retaining a large number of feature maps until the very end the U-net routinely outperforms the FCN-8 architecture.

2.4.2 Training a U-net

Training a U-net is similar to training any neural network, a process which was explained in section 2.2. The trainable parameters in the network are randomly initialized. The network is fed an input image and generates a prediction map as seen in figure 2.20 using the random parameters. The output is compared to a one-hot-encoded segmentation mask of the input image, see figure 2.22 and 2.23, showing the blueprint for a perfect segmentation. The comparison is done with some loss function, such as categorical cross-entropy. The loss, describing the prediction error, is propagated backward in the network and used to update the trainable parameters in a direction that will reduce the prediction error. A new image is fed to the U-net and the process is repeated using the updated parameters. This continues for as long as the prediction error is decreasing. The U-net have then learned how to take a new input image and produce an accurate segmentation map using the content of the image.

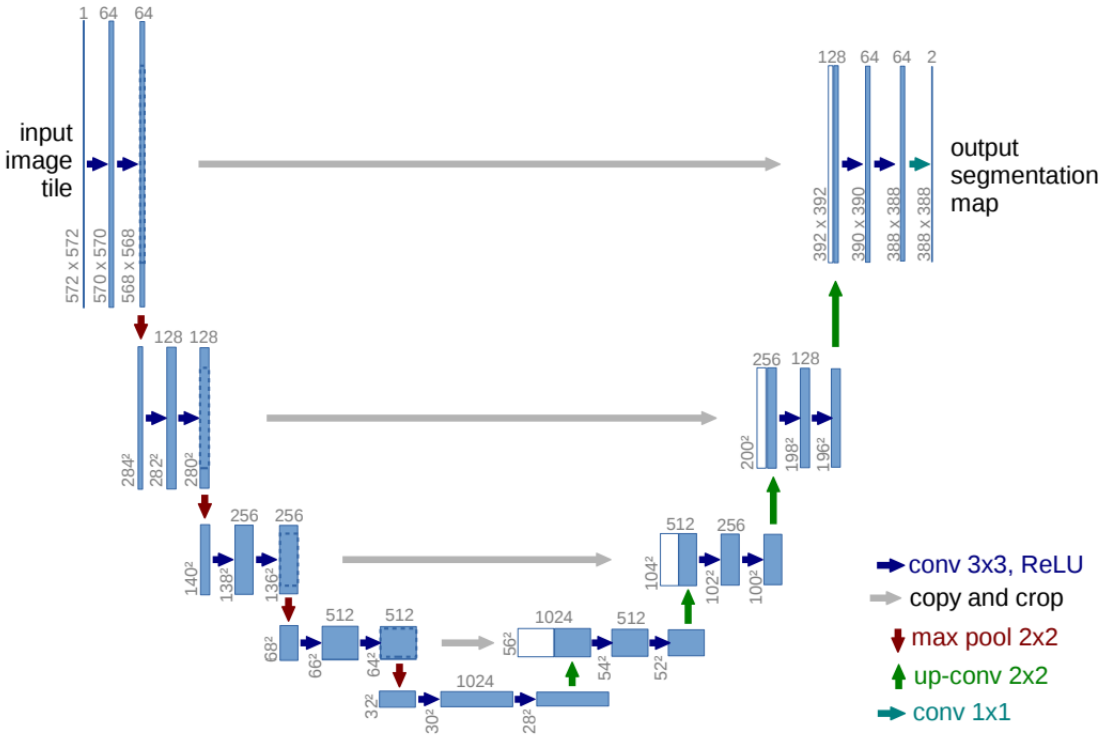


Figure 2.21: Original U-net architecture for semantic segmentation. They used convolution filters of size 3x3 with stride 1x1 and no padding. Without padding there is a reduction in feature map size in every convolution layer. A feature map that is reduced to half its size by a max pool, passed through two convolution layers (without padding), and then doubled in size using transposed convolution will not regain its original size. In order to match the sizes so they could concatenate feature maps from the encoder to the upsampled feature maps in the decoder they decided to crop the feature maps in the encoder. This approach only works if the object of interest is located centrally in the input images. Image credit [24]

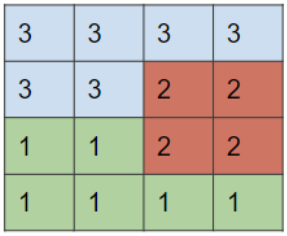


Figure 2.22: An example of a segmentation mask with 3 classes

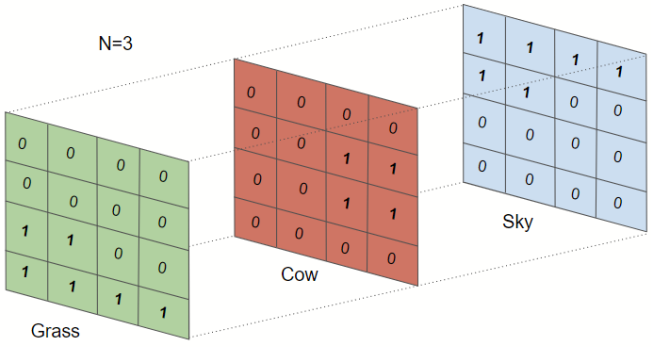


Figure 2.23: A one-hot-encoded version of the segmentation mask in figure 2.22

Chapter 3

Data

This chapter presents the images we want to segment and how they were obtained. It also describes the segmentation masks and how they were created.

3.1 Images

The dataset used for the analysis consists of 275 computed tomography (CT) scans of pigs provided by Norsvin SA. CT scanning is an imaging procedure that uses X-rays to measure the densities within an object, and in this case pigs. The images are created in a CT scanner where X-rays are emitted in a circular pattern around the pig. A detector on the opposite side of the X-ray source measures the photons that have passed through the biological material. Attenuation values are calculated and used to create a cross-sectional image, or a slice, of the body. The attenuation value describes how easily the organic materials in the pig is penetrated by the X-rays. Large attenuation values indicate that the X-rays were quickly weakened, i.e. they passed through something dense (for example bone). Small attenuation values indicates that the X-rays had little resistance. Attenuation values in CT images are commonly expressed using Hounsfield Units (HU) which is a linear transformation of the attenuation values. HU values for some common materials can be seen in table 3.1. Figure 3.3 shows the distribution of HU values in an example slice.

The X-ray source and detector is then shifted slightly along the transverse plane of the pig and another slice is created. This procedure continues until the entire pig is represented as a stack of slices. Views of this stack of slices seen from the transverse, sagittal, and coronal plane can be seen in figure 3.1, 3.4a, and 3.5a respectively. All images in the transverse plane is a square matrix of 512 x 512 pixels. The length of the scanned pigs varies, meaning that the length dimension of sagittal and coronal slices varies with each pig. The length is typically in the range of 800-1200 pixels.

Material	**HU**
Bone, calsium, metal	1000
Grey matter (brain)	35
Muscle, soft tissue	20-40
Water	0
Fat	-30 to -70
Air	-1000

Table 3.1: Hounsfield Units (HU) for some common materials. A large HU indicates a dense material [15]

3.2 Masks

Norsvin SA also provided segmentation masks for each of the image slices. Examples of these masks can be seen in figure 3.2, 3.4b, and 3.5b. Each mask consists of two classes, one class being the edible parts of the pig (yellow) and the other class being background (dark blue). Mask pixels were given a value of 1 and the background pixels were given a value of 0. Non-edible parts of the pig, namely intestines, testicles, and certain bones are labeled as background. It is important to note that the masks for these images were not manually annotated by domain experts. Norsvin SA has built an anatomical atlas representing the average pig by combining CT scans of 386 pigs [7]. The slices of this average pig were manually annotated with class labels. The labels from this average pig were then projected onto images of the 278 pigs used in this thesis using skeletal- and surface landmarks.

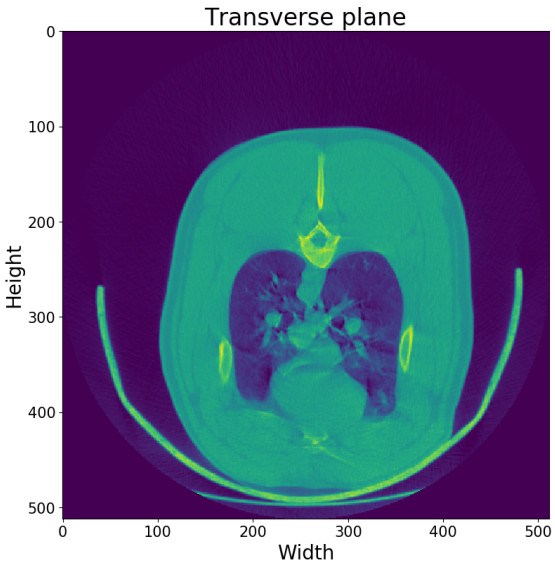


Figure 3.1: Slice of a pig viewed in the transverse plane.

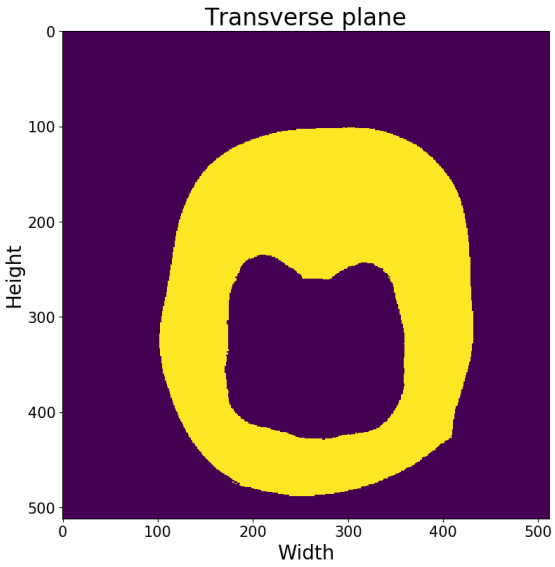


Figure 3.2: Segmentation mask for figure 3.1

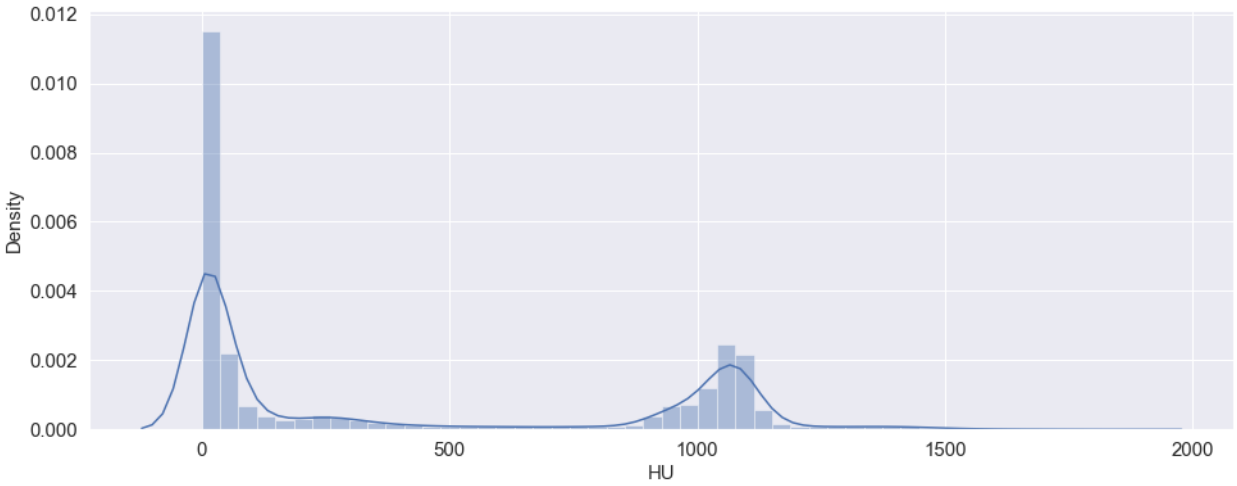
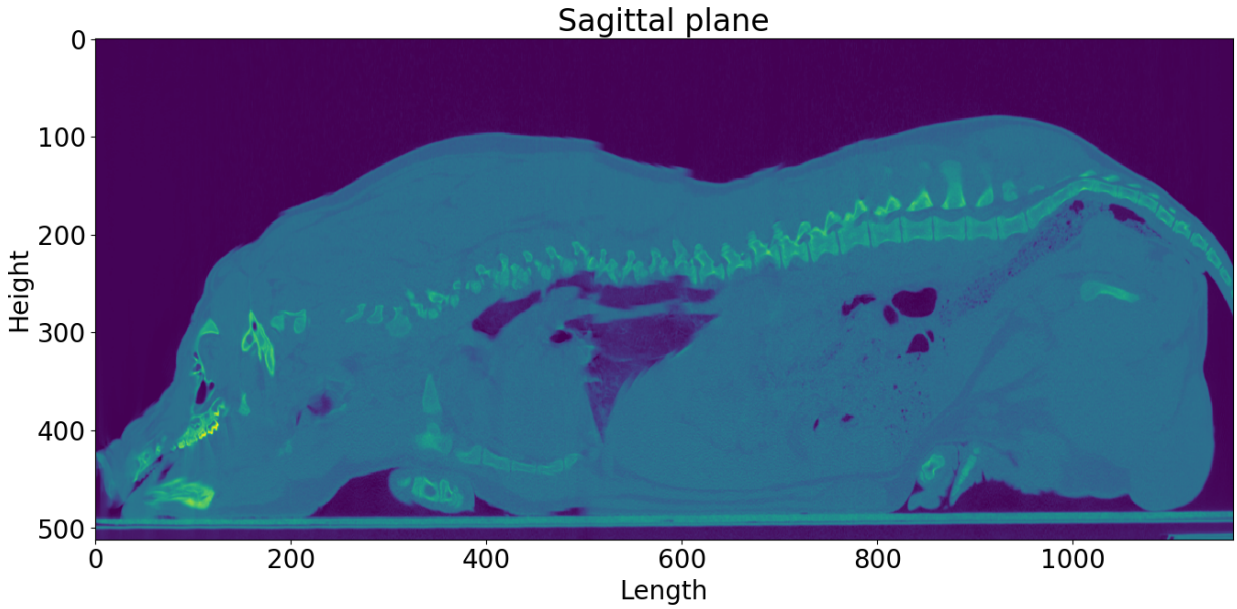
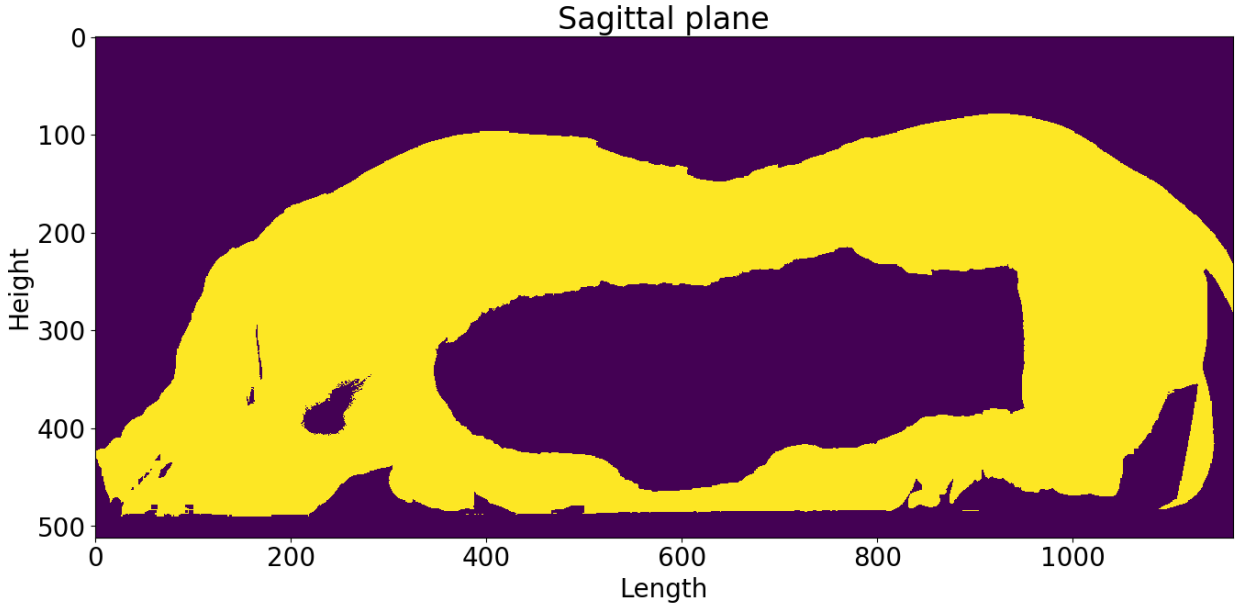


Figure 3.3: Distribution of the HU values in figure 3.1

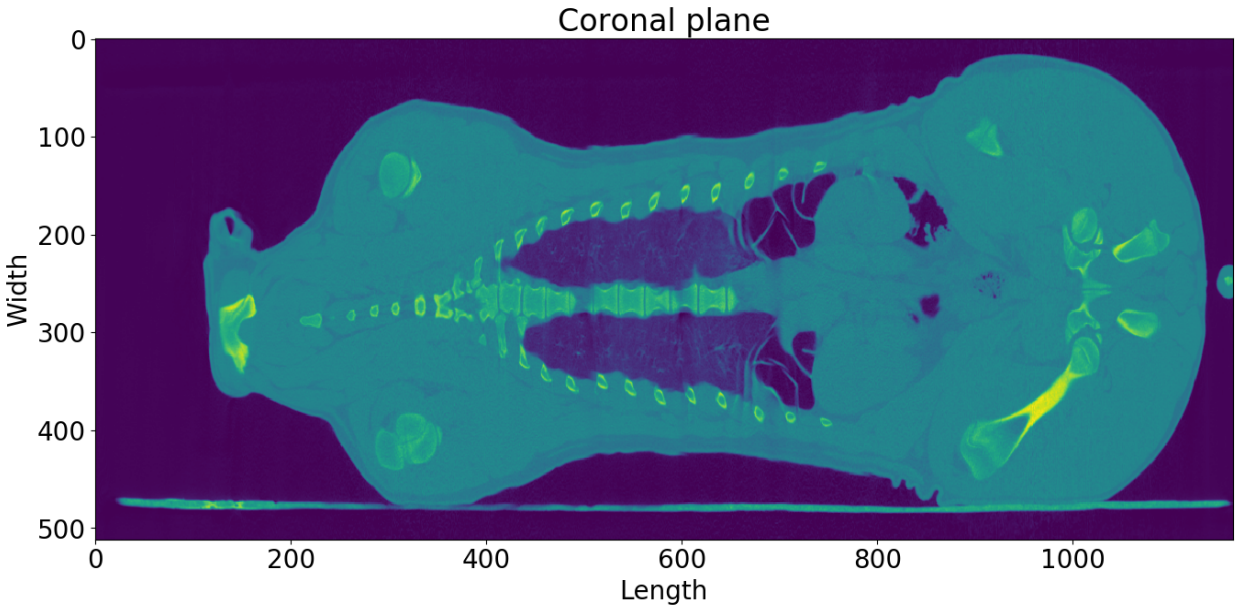


(a)

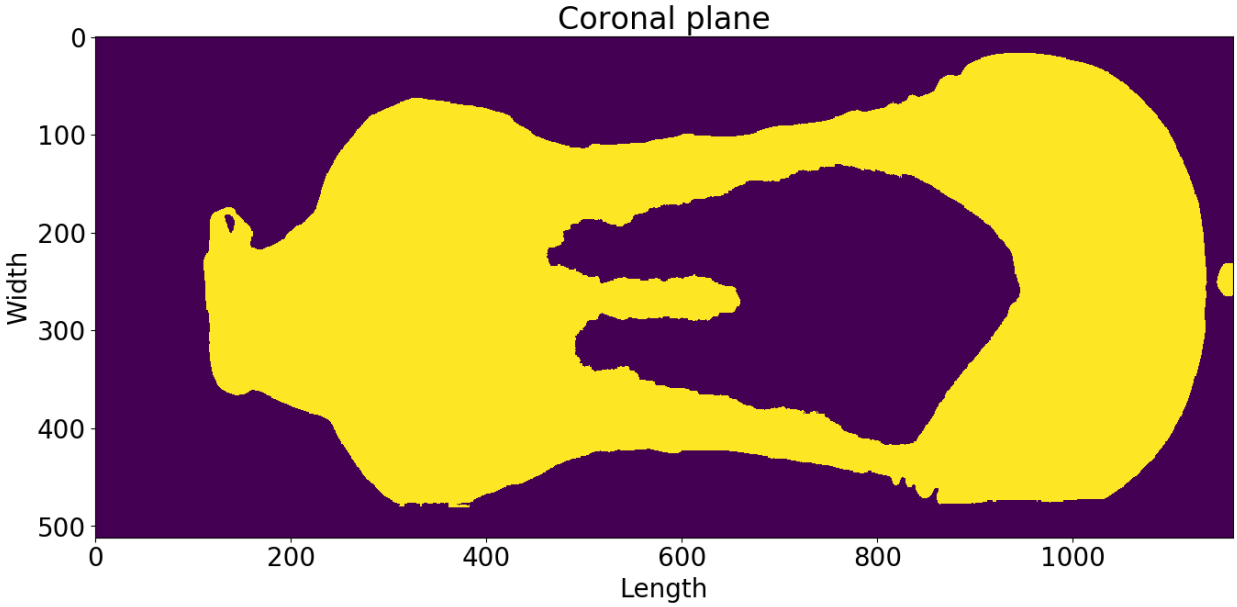


(b)

Figure 3.4: (a) Slice of a pig viewed in the sagittal plane. (b) Segmentation mask for slice in (a)



(a)



(b)

Figure 3.5: (a) Slice of a pig viewed in the coronal plane. (b) Segmentation mask for slice in (a)

Chapter 4

Method

This chapter starts off with a reminder of the problem we want to solve and the challenges associated with this task. It then lists the hardware and software used in this thesis. Afterwards, it describes the network design, how the network was trained, and how the performance of the network was evaluated.

4.1 Problem statement

The CT scan of each pig is a 3D volume. It would be advantageous to use the entire 3D CT scan as input to the network in order to utilize the spatial context in all three dimensions. A pixel could then be classified by using information in the same slice, and also information in neighboring slices. However, because of the size of these volumes, this is not technically feasible due to the limited GPU memory. For this reason, multiple approaches to feeding the 3D CT scan to the network were compared. This chapter contains a 2D section and a 3D section. In the 2D section we describe the process of training a set of networks, all using the same architecture, where each network is trained using images from either the transverse, sagittal, or coronal plane, see figure 3.1, 3.4a, and 3.5a respectively. The purpose was to see if any plane provides more useful information for a semantic segmentation task. In the 3D section we describe the process of training the network using blocks of the 3D volume small enough to fit in GPU memory.

4.2 Hardware and software

All preprocessing of images and training of models were conducted with an Intel Core i5-6600K @ 3.50GHz processor and a NVIDIA GTX 1070 8GB GDDR5 graphics processing unit (GPU). The model itself was built using Keras version 2.2.2 on a Tensorflow 1.10.0 backend.

Tensorflow is an open-source machine learning framework created by Google [1]. It allows the user to build complex neural networks in the form of computational graphs. It also supports GPU accelerated computation. This provides a major performance boost compared to training the model on a central processing unit (CPU), see table 4.1.

Keras [3] is a high-level application programming interface that runs on top of Tensorflow, CNTK [33], or Theano [31] (which are all machine learning frameworks). Keras handles some of the more complex aspects of Tensorflow thus allowing for faster experimentation.

Table 4.1: Comparison of a high-end CPU and GPU

Specifications	**Intel Core i7-6900K Processor Extreme Ed.**	**NVIDIA GeForce GTX 1080 Ti**
Base Clock Frequency	3.2 GHz	<1.5 GHz
Processing units	8	3584
Memory Bandwidth	64 GB/s	484 GB/s
Floating-Point Calculations	409 GFLOPS	11300 GFLOPS
Cost	∼$1000	∼$700

4.3 2D approach

4.3.1 2D U-net architecture

The network designed to carry out the segmentation task is a variation of the U-net architecture described in section 2.4.1. Figure 4.1, 4.2, 4.3, and 4.4 illustrates the network architecture. Appendix A contains the code used to build the network. Deep learning is not an exact science. It is common to try various state-of-the-art networks and best practices until obtaining a satisfactory result. The network was therefore designed using an iterative process experimenting with different network depths, layer combinations, learning rates, optimizers and loss functions.

As seen in figure 4.1 it consists of six downsampling steps and six upsampling steps. It was found that this is an adequate depth for building the necessary feature identifiers. Deeper networks provided no benefit and even degraded performance in some instances. A deeper network would also add to the already substantial number of trainable parameters that have to be computed and stored.

Convolution layers in the topmost blocks (where the spatial dimension of the activation maps are equal to that of the input image) each have $16 * 2^0 = 16$ filters of size 3x3. These filters were moved spatially across the feature maps using a stride of 1x1. The input to each convolution layer is zero-padded to preserve size. The convolution layers in the blocks one step down on the "ladder" each have $16 * 2^1 = 32$ filters, the convolution layers in the blocks another step down each have $16 * 2^2 = 64$ filters, and so on. The convolution layers in the bottom block each have $16 * 2^6 = 1024$ filters.

Batch normalization layers are heavily used as they are proven to improve network performance, see section 2.3.5. They are placed between each convolution layer and activation layer as suggested by the authors of the original paper [13]. All activation layers in the network is using the ReLU activation function, see section 2.2.6.

A key component of this U-net is the residual blocks. All downsampling and upsampling steps have two of these, see figure 4.2 and 4.3. These blocks contain the standard layers of a CNN, namely convolution, activation, and batch normalization, with added residual connections. As explained in section 2.3.6 these mitigate the degradation problem of deep networks. They have also been shown to help U-net architectures learn faster [5].

The feature maps are downsampled using max-pooling operations, see section 2.3.2, and upsampled using transposed convolutions, see section 2.3.3.

The resulting set of feature maps output by the upsampling steps is compressed to a single feature map using a 1x1 convolution layer with a single filter. Because this is a binary classification problem (a pixel is either part of the mask or not) we only need one feature map. This feature map is converted to a prediction map through element-wise application of the sigmoid activation function, see section 2.2.6.

During network training we used binary cross-entropy, see section 2.2.3, to compute the loss of the network given the prediction map and the mask. The training was optimized using RMSprop, see section 2.2.5.

The total numbers of parameters in this network is $81,867,505$ of which $81,831,025$ are trainable.

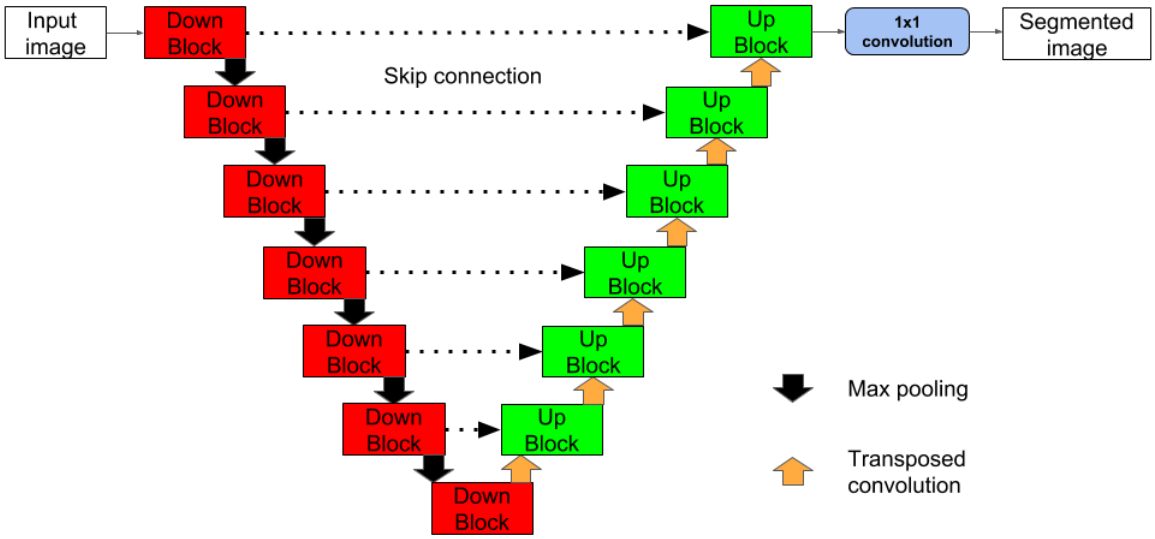


Figure 4.1: U-net for semantic segmentation of images. 6 max-pooling operations are used for downsampling and 6 transposed convolution operations are used for upsampling. The 6 max-pooling operations will reduce a size 512x512 input image to a coarse size 8x8 feature map with limited spatial information. Skip connections are used to transfer spatial information from the encoder to the upsampled coarse feature maps in the decoder.

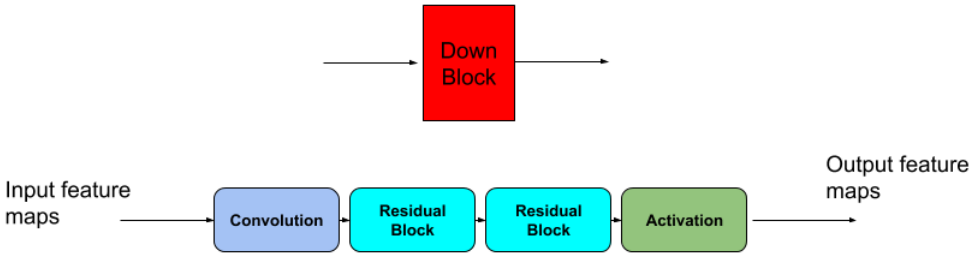


Figure 4.2: Components of a down-block used in the encoder of the U-net.

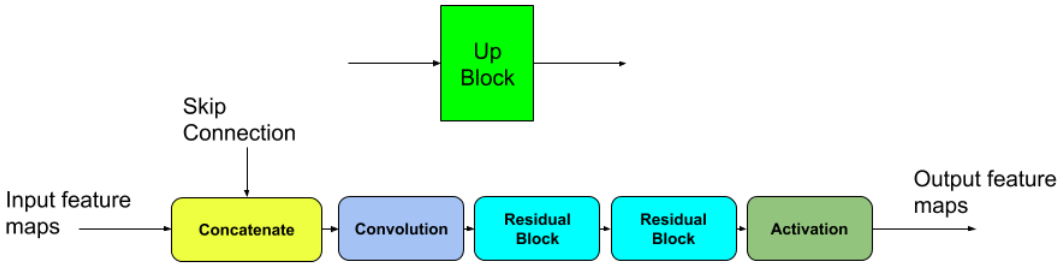


Figure 4.3: Components of a up-block used in the encoder of the U-net.lock

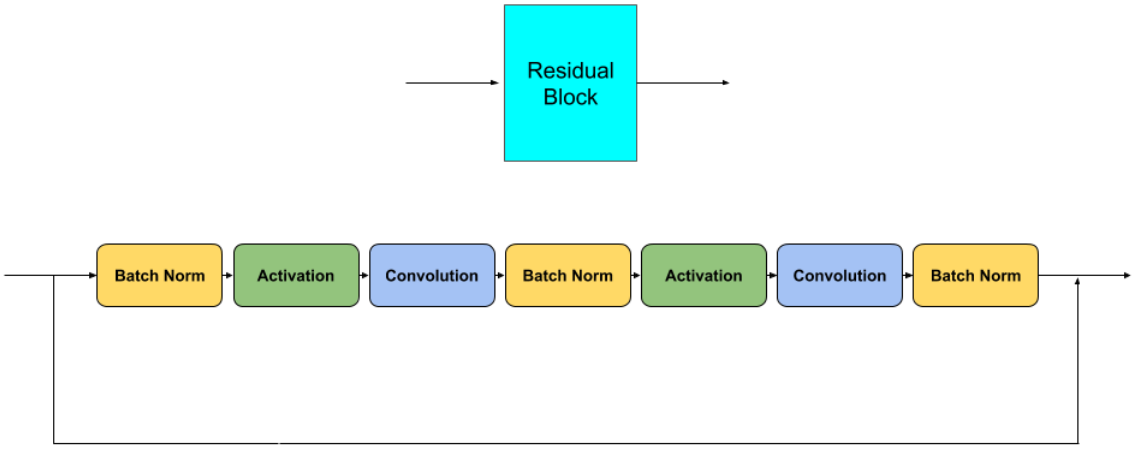


Figure 4.4: A residual connection allows information to bypass the convolution layers in a residual block.

4.3.2 Data preparation

For the 2D approach, three separate U-nets with this architecture were trained using images in either the transverse, sagittal, or coronal plane. These images were not fed into the network as is, they had to be preprocessed.

Padding

A consequence of the chosen architecture is that there is a restriction on acceptable input size. More precisely, the number of pixels in any dimension of the input image had to be divisible by two to the power of max pool operations. The architecture shown in 4.1 has six max pooling operations, meaning all input dimensions had to be divisible by 64. An image of size 512 x 512 is acceptable, but 512 x 600 is not. To meet the requirement, all images were zero-padded, see section 2.3.1, up to the nearest number of pixels where the requirement is satisfied. A 512 x 600 image would become 512 x 640.

Weighting of classes

Class imbalance is a frequent problem in classification tasks. The amount of edible pig marked by the masks varies across the CT scan. In the transverse plane the slices at the very front and back of the pig, i.e. head and tail, contains predominantly background pixels. Slices at the center of the pig tend to be more balanced with an equal amount of mask pixels and background pixels. Class imbalance is problematic because the classifier will put more emphasis on the majority class and neglect the minority class during training [29]. The fraction of mask pixels to background pixels in the entire training set was approximately 0.289. To counter this the loss function was weighted during training. The computed loss for pixels where the true class label is 0 (background) was multiplied by $1/(1 - 0.289) = 1.406$. The computed loss for pixels where the true class label is 1 (mask) was multiplied by $1/0.289 = 3.460$. The network is then penalized more for incorrect predictions on the minority class which in this case was mask pixels.

Training of network

The general process for training a U-net architecture was explained in section 2.4.2. The network trained on images in the transverse plane was fed mini-batches with 4 images in order to speed up training. Each image was randomly selected from all available images. The networks trained on images in the sagittal- and coronal plane were fed single images. This is because the images don't stack due to varying length dimensions.

4.4 3D approach

4.4.1 3D U-net architecture

The U-net architecture used to train on smaller blocks from the CT scan is similar to the one used in the 2D approach, with some minor adjustments. The convolution operation explained in section 2.3.1 is a 2D convolution. The filters are then moved spatially across two dimensions of the input. The filter in a 3D convolution moves spatially across three dimensions of the input. This is also the case for the max-pooling layers and transposed convolution layers. Another adjustment is that the network depth was reduced to four downsampling steps and four upsampling steps due to memory constraints.

4.4.2 Data preparation

The block approach was motivated by the V-Net paper [21] where the authors trained their network using blocks of size 128 x 128 x 64 extracted from the complete CT scan. In this thesis blocks of size 96 x 96 x 96 was used. 96 was chosen due to memory constraints and the restriction of the

network only allowing for input dimensions that are divisible by two to the power of max pool operations. Therefore, block dimensions had to be divisible by 16.

The classes were weighted the same amount as in the 2D case.

Training of network

During training the program extracted two blocks (mini-batch of size two), each of size 96 x 96 x 96, from a random location in a CT scan in the training set. This process was repeated 50 times for each scan in the training set to ensure that the network had seen all parts of the animal.

4.5 Evaluating network performance

After training is completed it is necessary to measure the performance of the network on images the network hasn't seen before. If the network is performing well during training and not on the unseen data it means that the network has failed to properly generalize the relationship between the input image and the segmentation mask. A common practice in machine learning tasks is to divide the available data into a training set and test set. The training set is used to train the model, i.e. update trainable parameters. The trained model is then used to predict the samples in the test set which the model hasn't seen before.

The training set and test set in this thesis were given 238 and 37 CT scans respectively. The 37 CT scans are roughly 15% of all available CT scans (275). This is enough to provide a realistic measure of the performance of the model. Too few samples and there is a risk the model is being tested on outliers.

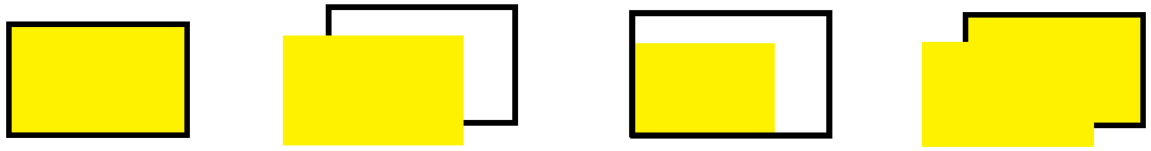
4.5.1 Performance metric

An essential part of evaluating the performance of the network is the performance metric. The most intuitive metric when doing classification is accuracy. The accuracy of a segmented image is calculated using the segmented image and the corresponding mask. The accuracy is then the number of correctly classified pixels in the segmented image (in accordance with the mask) relative to the total number of pixels in the segmented image. If half the pixels are correctly classified then the accuracy of the prediction is 50%. To ensure that this is a representative accuracy for the model it is recommended to conduct this test on a sufficiently large number of images.

One problem with using accuracy as a metric for semantic segmentation, and many other classification tasks, arise in the case of class imbalance. Take for example an image consisting of 90% grass and 10% asphalt. The model could then predict that all pixels in the image is grass and obtain a 90% accuracy, which isn't very useful if the location of asphalt is of interest. A much more helpful metric when quantifying the performance of a semantic segmentation model is the Intersection over Union metric (IoU), also known as the Jaccard index. It measures the similarity between two sets of data:

$$IoU(A, B) = \frac{A \cap B}{A \cup B} \tag{4.1}$$

where A is the mask and B is the prediction. The IoU function returns a value in the range [0,1] where 0 indicates no spatial overlap between the prediction and the mask, and 1 indicates complete overlap between the prediction and the mask. In terms of semantic segmentation, an IoU of 1 means that the predicted pixel class for all pixels is in accordance with the ground truth given by the mask for the respective input image, i.e. perfect segmentation. An illustration of the IoU can be seen in figure 4.5.



(a) Mask (b) Prediction (c) Mask ∩ Prediction (d) Mask ∪ Prediction

Figure 4.5: The black rectangle is the object we want to segment from the surrounding area. The mask in (a) shows what perfect segmentation looks like. The prediction in (b) is slightly off target. In (c) we have the area of the prediction that was on the object according to the mask. In (d) we have the total area marked by both the mask and the prediction. Intersect over union for the prediction is calculated by dividing the yellow area in (c) by the yellow area in (d). If the prediction has the correct size but half of it is outside the object then we will get an IoU of $0.\bar{3}3$

The IoU was implemented in Python as:

```
def iou(mask, prediction):

    intersect = prediction * mask
    union = prediction + mask

    return (intersect.sum() + 1e-6) / (union.sum() + 1e-6)
```

where $1e-6$ was added to the numerator and denominator to prevent dividing by zero in cases where both the prediction and mask only consisted of background pixels which have the value 0.

4.5.2 Evaluation procedure

To summarize, three networks have been trained on 2D images (from different planes) and one network has been trained using 3D blocks. We wanted to compare these networks.

The network trained on 2D slices in the transverse plane was given a CT scan from the test set. It then predicted the segmentation mask for each slice in the transverse plane. These predictions were stacked in a 3D volume. Similarly, the network trained on 2D slices in the sagittal plane was given the same CT scan. It then predicted the segmentation mask for each slice in the sagittal plane and stacked these in a 3D volume. We now had two 3D volumes of the same dimensions (only rotated) that contains dense predictions for the same pig. By calculating the IoU of each of these 3D volumes with respect to the mask for that pig we were able to see which plane was more useful in training the network.

Creating a comparable 3D volume of dense predictions using the network trained on 3D blocks was done by partitioning the CT scan into blocks of size 96x96x96, feed them to the network for prediction, and then put the blocks back together in the correct position. The result was then compared with the results from the 2D approach.

The performance of the various networks on the 37 CT scans in the test set is presented in chapter 5.

Chapter 5

Results

Table 5.1 contains the IoU scores from semantic segmentation on 37 unseen CT scans of pigs. The table contains results obtained using networks trained in the transverse, sagittal, and coronal plane of the CT scans and results obtained using a network trained on sub-blocks of the CT scans. The table also contains the results from additional predictions that were made by combining the predictions made using networks trained in the transverse, sagittal, and coronal plane. The "transverse + sagittal" column in table 5.1 is the result of averaging the two 3D volumes of predictions output by the transverse network and the sagittal network. Similarly, we have a "transverse + coronal" column and a "sagittal + coronal" column. The "combined" column is the result of averaging the predictions from all three networks.

Table 5.2 is the same as 5.1 except all values are seen relative to the values in the "transverse" column. This makes it easier to compare the methods. A qualitative assessment of the segmented images is presented in chapter 6.

Table 5.1: IoU scores for the 37 pigs in the test set

Pig ID	transverse	sagittal	coronal	transverse+ sagittal	transverse+ coronal	sagittal+ coronal	combined	3D
10969	0.951	0.952	0.910	0.952	0.952	0.952	0.960	0.777
19813	0.963	0.966	0.928	0.964	0.964	0.966	0.972	0.769
21797	0.953	0.957	0.922	0.958	0.954	0.957	0.962	0.779
21798	0.946	0.945	0.909	0.950	0.947	0.945	0.951	0.786
21800	0.949	0.953	0.917	0.954	0.949	0.954	0.959	0.784
21811	0.965	0.957	0.919	0.966	0.966	0.958	0.966	0.791
21816	0.958	0.961	0.918	0.966	0.958	0.961	0.965	0.792
21828	0.958	0.963	0.925	0.964	0.959	0.963	0.969	0.774
21887	0.943	0.948	0.907	0.948	0.943	0.948	0.951	0.795
21901	0.965	0.961	0.927	0.966	0.965	0.961	0.971	0.772
21920	0.938	0.945	0.907	0.943	0.938	0.945	0.948	0.793
21943	0.956	0.957	0.914	0.958	0.957	0.957	0.961	0.786
21951	0.956	0.957	0.919	0.959	0.956	0.957	0.961	0.758
21952	0.956	0.960	0.915	0.962	0.957	0.961	0.963	0.782
21953	0.896	0.895	0.855	0.910	0.896	0.895	0.899	0.797
21954	0.956	0.961	0.920	0.959	0.956	0.961	0.965	0.778
21955	0.962	0.963	0.925	0.961	0.962	0.964	0.972	0.761
21967	0.965	0.964	0.926	0.966	0.966	0.964	0.972	0.790
21979	0.963	0.959	0.916	0.962	0.963	0.959	0.964	0.763
21991	0.960	0.964	0.926	0.963	0.960	0.965	0.970	0.778
22004	0.964	0.962	0.922	0.961	0.964	0.962	0.968	0.751
22005	0.948	0.950	0.907	0.952	0.948	0.950	0.954	0.777
22008	0.963	0.967	0.924	0.967	0.963	0.967	0.971	0.772
22009	0.965	0.966	0.928	0.969	0.966	0.966	0.972	0.761
22010	0.966	0.963	0.921	0.967	0.967	0.964	0.971	0.766
22011	0.931	0.919	0.892	0.939	0.931	0.920	0.933	0.811
22012	0.955	0.963	0.919	0.962	0.956	0.963	0.968	0.780
22013	0.953	0.955	0.919	0.961	0.954	0.956	0.961	0.783
22066	0.949	0.955	0.910	0.954	0.949	0.955	0.957	0.756
22216	0.952	0.956	0.915	0.960	0.952	0.957	0.961	0.790
22323	0.955	0.957	0.914	0.958	0.956	0.957	0.964	0.764
23303	0.951	0.957	0.917	0.959	0.951	0.958	0.961	0.782
23494	0.954	0.960	0.919	0.959	0.955	0.960	0.966	0.768
31125	0.956	0.966	0.918	0.965	0.957	0.966	0.969	0.785
31338	0.965	0.961	0.918	0.961	0.966	0.962	0.968	0.778
32385	0.966	0.967	0.923	0.969	0.967	0.968	0.973	0.781
32576	0.962	0.965	0.923	0.965	0.963	0.966	0.971	0.786

Table 5.2: IoU scores for the 37 pigs in the test set seen relative to the scores in the "transverse" column

Pig ID	transverse	sagittal	coronal	transverse+ sagittal	transverse+ coronal	sagittal+ coronal	combined	3D
10969	1.0	1.001	0.957	1.001	1.001	1.001	1.009	0.817
19813	1.0	1.003	0.964	1.001	1.001	1.003	1.009	0.799
21797	1.0	1.004	0.967	1.005	1.001	1.004	1.009	0.817
21798	1.0	0.999	0.961	1.004	1.001	0.999	1.005	0.831
21800	1.0	1.004	0.966	1.005	1.000	1.005	1.011	0.826
21811	1.0	0.992	0.952	1.001	1.001	0.993	1.001	0.820
21816	1.0	1.003	0.958	1.008	1.000	1.003	1.007	0.827
21828	1.0	1.005	0.966	1.006	1.001	1.005	1.011	0.808
21887	1.0	1.005	0.962	1.005	1.000	1.005	1.008	0.843
21901	1.0	0.996	0.961	1.001	1.000	0.996	1.006	0.800
21920	1.0	1.007	0.967	1.005	1.000	1.007	1.011	0.845
21943	1.0	1.001	0.956	1.002	1.001	1.001	1.005	0.822
21951	1.0	1.001	0.961	1.003	1.000	1.001	1.005	0.793
21952	1.0	1.004	0.957	1.006	1.001	1.005	1.007	0.818
21953	1.0	0.999	0.954	1.016	1.000	0.999	1.003	0.890
21954	1.0	1.005	0.962	1.003	1.000	1.005	1.009	0.814
21955	1.0	1.001	0.962	0.999	1.000	1.002	1.010	0.791
21967	1.0	0.999	0.960	1.001	1.001	0.999	1.007	0.819
21979	1.0	0.996	0.951	0.999	1.000	0.996	1.001	0.792
21991	1.0	1.004	0.965	1.003	1.000	1.005	1.010	0.810
22004	1.0	0.998	0.956	0.997	1.000	0.998	1.004	0.779
22005	1.0	1.002	0.957	1.004	1.000	1.002	1.006	0.820
22008	1.0	1.004	0.960	1.004	1.000	1.004	1.008	0.802
22009	1.0	1.001	0.962	1.004	1.001	1.001	1.007	0.789
22010	1.0	0.997	0.953	1.001	1.001	0.998	1.005	0.793
22011	1.0	0.987	0.958	1.009	1.000	0.988	1.002	0.871
22012	1.0	1.008	0.962	1.007	1.001	1.008	1.014	0.817
22013	1.0	1.002	0.964	1.008	1.001	1.003	1.008	0.822
22066	1.0	1.006	0.959	1.005	1.000	1.006	1.008	0.797
22216	1.0	1.004	0.961	1.008	1.000	1.005	1.009	0.830
22323	1.0	1.002	0.957	1.003	1.001	1.002	1.009	0.800
23303	1.0	1.006	0.964	1.008	1.000	1.007	1.011	0.822
23494	1.0	1.006	0.963	1.005	1.001	1.006	1.013	0.805
31125	1.0	1.010	0.960	1.009	1.001	1.010	1.014	0.821
31338	1.0	0.996	0.951	0.996	1.001	0.997	1.003	0.806
32385	1.0	1.001	0.955	1.003	1.001	1.002	1.007	0.808
32576	1.0	1.003	0.959	1.003	1.001	1.004	1.009	0.817

Chapter 6

Discussion

This chapter contains a qualitative analysis of the results. A summary of the results is first presented. An analysis of network performance at various areas of the pigs is then conducted in order to find out where the networks are having difficulties. Afterwards, there is a qualitative assessment of segmentation quality on a selection of segmented images. Finally, there is a section discussing the strengths and weaknesses of the proposed network.

6.1 Summary of the results

Summary statistics for table 5.1 is presented in table 6.1. From this table we can see that, on average, there are relatively minor differences in performance (measured in IoU score) between the networks trained on 2D planes. The term "transverse network" will from here on mean the 2D network trained on slices in the transverse plane. Similarly, "sagittal network" and "coronal network" will mean the 2D networks trained on slices in the sagittal and coronal plane respectively.

The transverse- and sagittal network had an average IoU score of 0.954 and 0.956 respectively, while the average IoU score of the coronal network was 0.916. We can see that although the coronal network performed worse than both the transverse and sagittal network, the best performing network on average was the "combined" network which includes the coronal network. This indicates that all planes contain useful information that is not present in any of the other planes.

The network trained on 3D blocks performed significantly worse than the networks trained on 2D planes.

Table 6.1: Summary statistics of table 5.1

	transverse	sagittal	coronal	tra_sag	tra_cor	sag_cor	combined	3D
count	37	37	37	37	37	37	37	37
mean	0.954	0.956	0.916	0.958	0.955	0.956	0.962	0.778
std	0.013	0.014	0.013	0.011	0.013	0.014	0.013	0.013
min	0.896	0.895	0.855	0.910	0.896	0.895	0.899	0.751
25%	0.951	0.955	0.914	0.958	0.952	0.956	0.961	0.769
50%	0.956	0.960	0.919	0.961	0.957	0.960	0.965	0.779
75%	0.963	0.963	0.923	0.965	0.963	0.964	0.970	0.786
max	0.966	0.967	0.928	0.969	0.967	0.968	0.973	0.811

Figure 6.1 is a boxplot depicting the distributions of the IoU scores in table 5.1. We can see that there are two pigs that have a very negative effect on the average performance, namely pig 21953 and 22011. The deviation in IoU score for these two pigs relative to the mean IoU is quite consistent in all planes, except for the 3D method where there are no outliers. The CT scans of these pigs likely contained something the networks wasn't able to learn. Slices in the transverse plane from these two pigs, and two other pigs for comparison, can be seen in figure 6.2. In this figure we can see that pig 21953 is slightly rotated compared to the other pigs, which is a possible explanation for why the network is struggling. No obvious abnormality can be seen for pig 22011 in this slice.

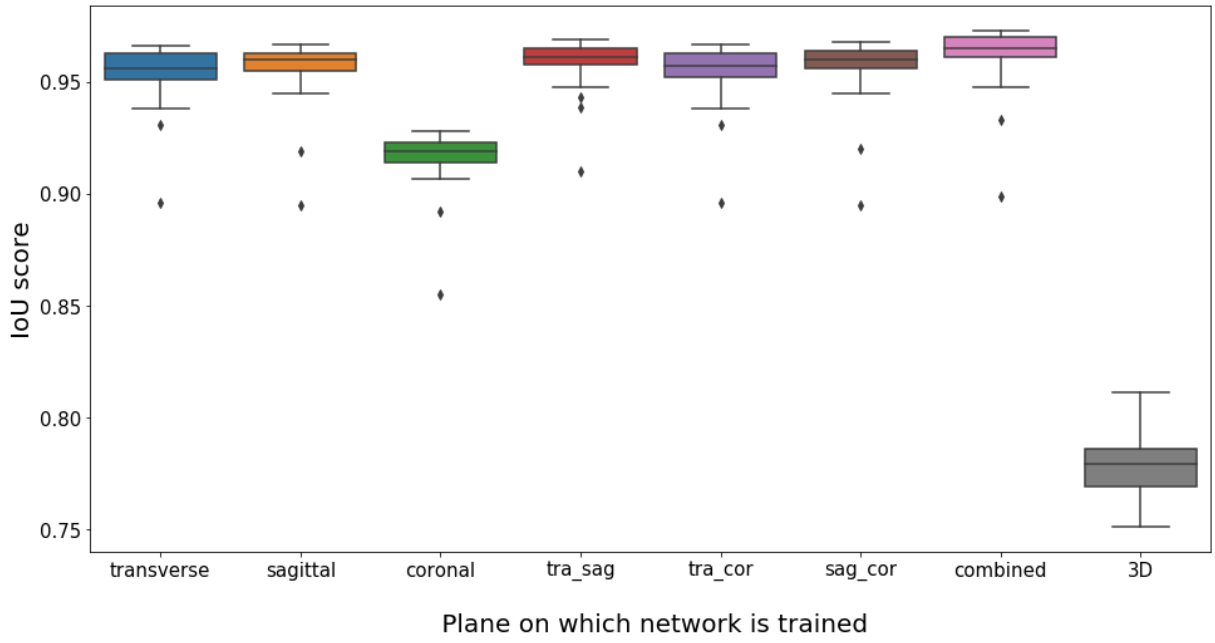
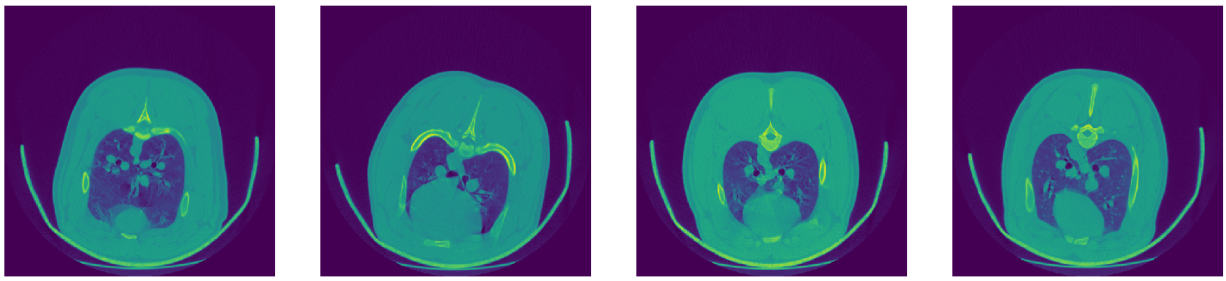


Figure 6.1: Distribution of the IoU scores in table 5.1



(a) ID 19813 (b) ID 21953 (c) ID 22011 (d) ID 32385

Figure 6.2: Image slice nr. 500 in the transverse plane for a selection of pigs. Pig with ID 21953 is slightly rotated which is possibly causing problems for the networks as they are not used to seeing images at this angle.

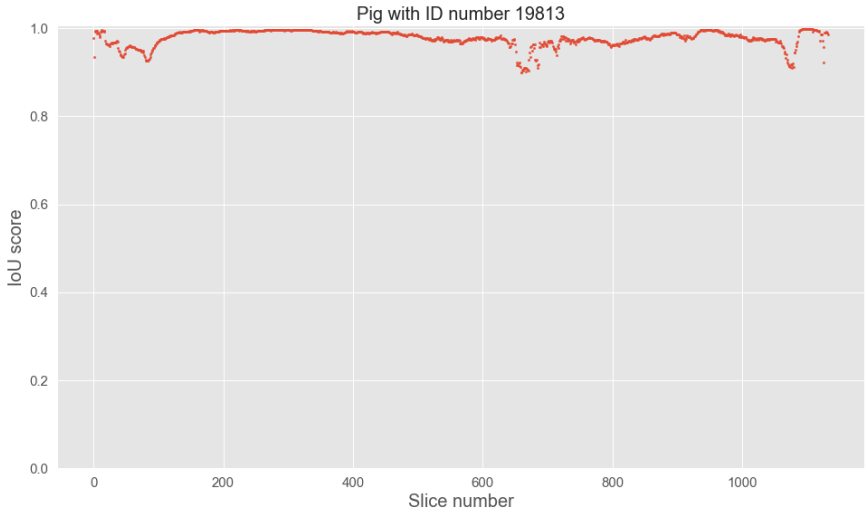
6.2 Analysis of results

In order to get a better understanding of where the networks are struggling we have selected three pigs for further analysis, namely the pigs with ID 19813, 21955, and 31125. Figure 6.3a, 6.3b, and 6.3c show the IoU scores obtained using the transverse network on each slice in the transverse plane along the length of the three pigs. Slice number 0 is the very first slice at the head of the pig. Larger slice numbers mean we are moving towards the back of the pig. We can see that the transverse network struggled to produce a good segmentation at the very front and back of the pigs. There is also a noticeable dip in IoU score between slice numbers 600-800. Figure 6.6 and 6.7 shows how

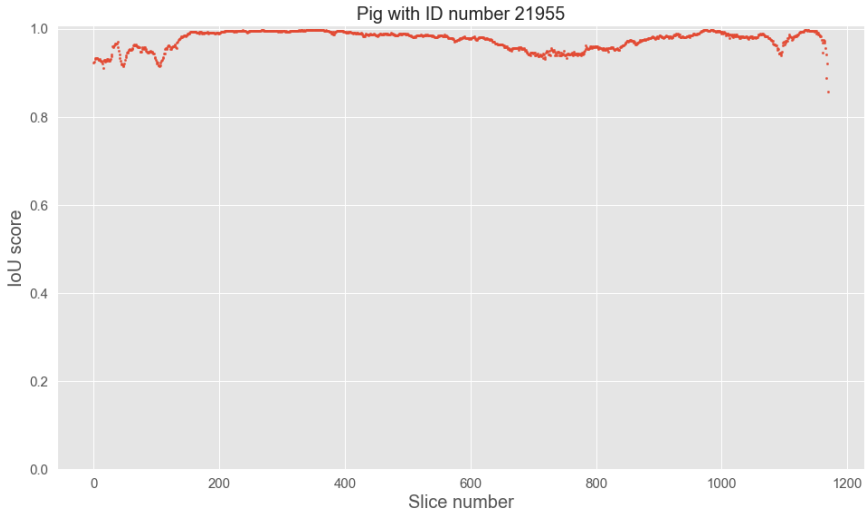
the masks develop as we move through pig 21955 in the transverse plane. We can here see that the problem areas have a larger ratio of background pixels compared to the well-performing areas.

Figure 6.4a, 6.4b, and 6.4c show the IoU scores obtained using the sagittal network on each slice in the sagittal plane along the width of the three pigs. At the sides of the pigs there are areas where the network is achieving perfect segmentation (according to IoU score). From figure 6.8 and 6.9 we can see that these slices contain only background pixels. These are present in the CT scans because the scanner is wider than the pigs. We can see that the sagittal network is struggling with the transition from slices containing only background pixels to slices containing a relatively small number of mask pixels. The IoU metric is heavily punishing cases where the network fails to correctly classify a very minor mask presence in a slice (small intersect over large union in the python implementation of IoU).

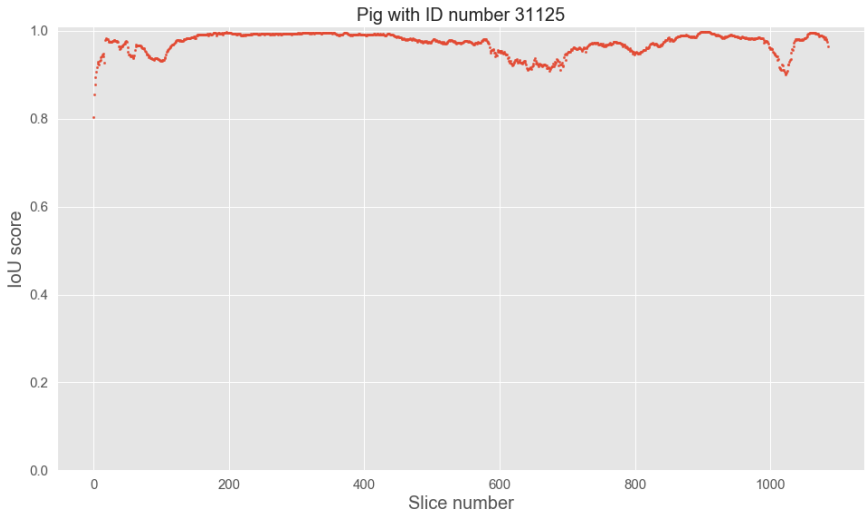
Figure 6.5a, 6.5b, and 6.5c show the IoU scores obtained using the coronal network on each slice in the coronal plane along the height of the three pigs. Slice number 0 is above the pig. As the slice number increases we move towards the bottom of the pig. Because the scanner is taller than the pigs there are slices containing only background pixels where the network is achieving perfect segmentation. The transition from slices with only background pixels to slices with a few mask pixels is again problematic. We observe a slight downward trend in IoU score as we move further down on the pig. At approximately slice number 450 there is a sharp drop in IoU score on all three pigs, which explains why the coronal network on average is performing worse than the other 2D networks. In figure 6.10 and 6.11 we can see that this is roughly where the masks lose any semblance to the outline of a pig. In figure 6.2 we can see that in order to keep the pigs upright in the CT scanner they've used an u-shaped container which is influencing the shape of the pigs. In addition, the legs of the pigs could be placed differently from pig to pig. In general, there seems to be a lot of variation in the bottom coronal slices which makes training difficult. Figure 3.4a back in section 3.1 also illustrates the complexity of slices in the coronal plane at the bottom of the pigs. This figure also shows the gap under the container where slices in the coronal plane will contain only background pixels. The coronal network mistakenly predicts that there are mask pixels in these slices which is why the IoU score for these slices is zero.



(a)

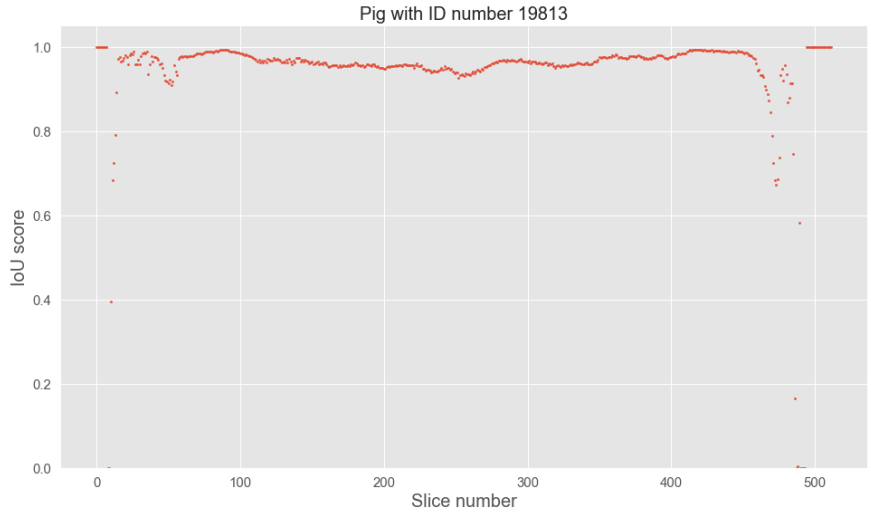


(b)

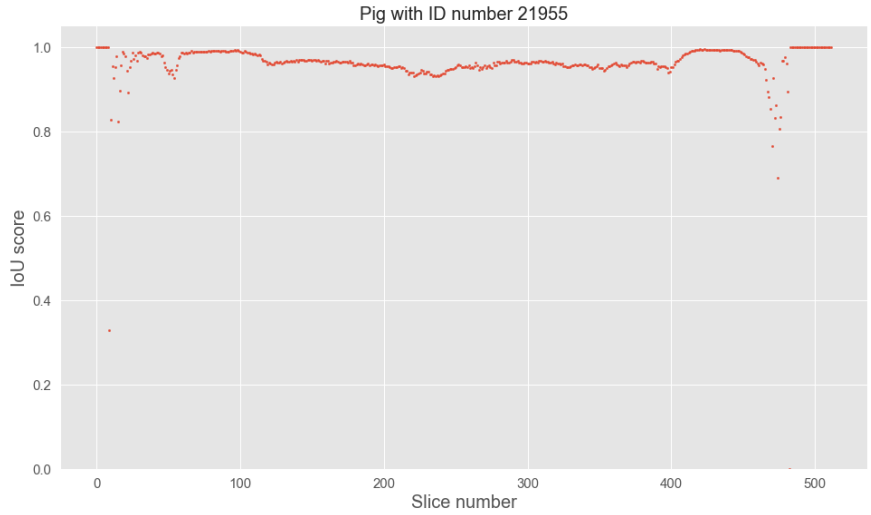


(c)

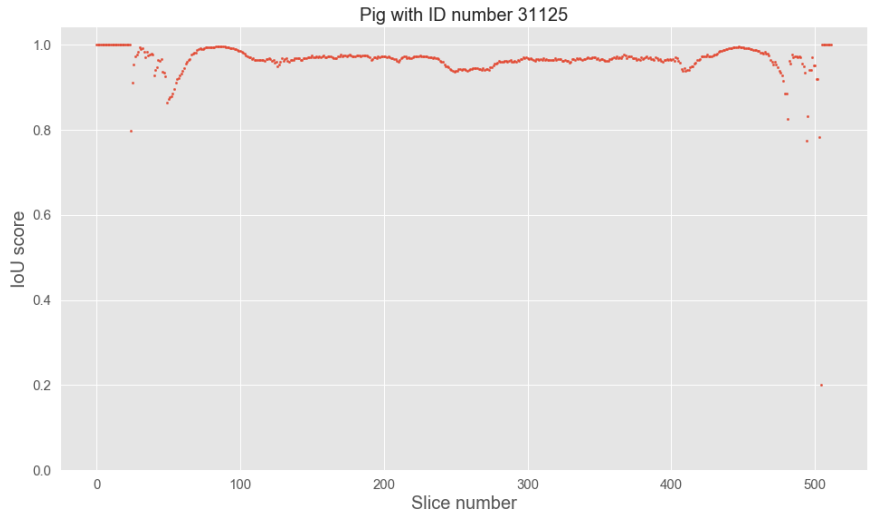
Figure 6.3: IoU scores obtained using the transverse network on each slice in the transverse plane along the length of three pigs. Going from left to right corresponds to going from head to tail.



(a)

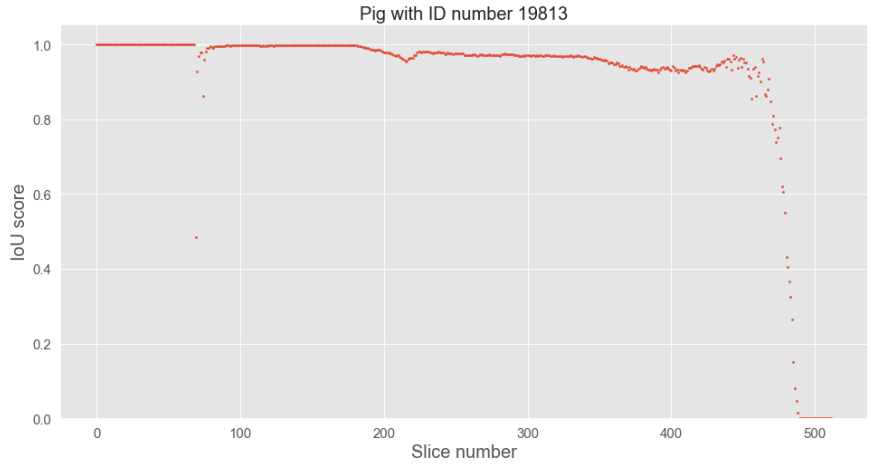


(b)

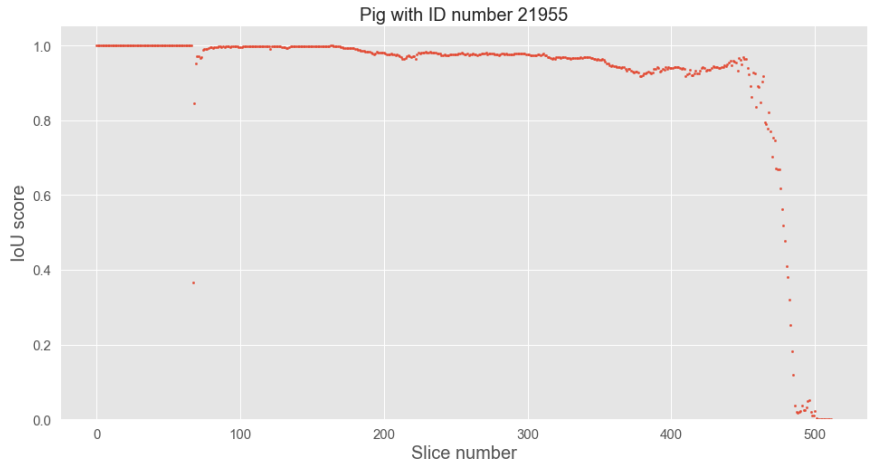


(c)

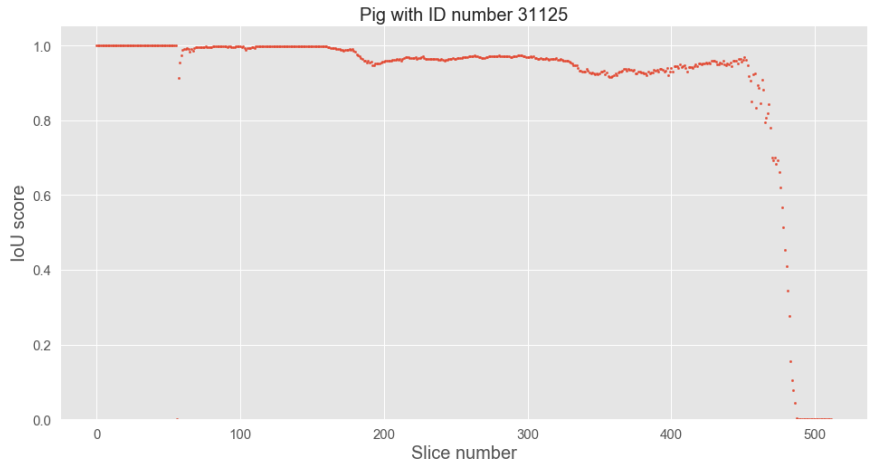
Figure 6.4: IoU scores obtained using the sagittal network on each slice in the sagittal plane along the width of three pigs.



(a)



(b)



(c)

Figure 6.5: IoU scores obtained using the coronal network on each slice in the coronal plane along the height of three pigs. Going from left to right corresponds to going from top to bottom.

6.3 Visual assessment of segmentation versus mask

The IoU score is a well established quantitative measure of segmentation quality. However, a visual assessment should be conducted to get an idea of what types of structures the networks struggled to segment. Visualizing the predicted segmentation masks for all pigs is impractical. The pig with ID number 21955 was selected for this purpose because the networks are performing close to average on this pig according to table 5.1 and 6.1.

6.3.1 Transverse plane

Figure 6.6 and 6.7 shows the predicted mask versus the true mask for a selection of slices in transverse plane. We can see that the prediction is generally very good at correctly segmenting the outer boundaries of the structures, even for the more complex shapes in slice nr. 100 and nr. 200. The main mistakes in this plane seems to be about failure to detect smaller "holes" (areas with background pixels within the larger structure).

Something that should be noticed is that in some of the true mask slices there are mask pixels outside the larger structure. In these particular slices they resemble the u-shaped container used to support the pig. In section 3.2 we explained how the masks are automatically generated. It is likely that this automated procedure has made these mistakes. It is interesting to see that the predicted masks do not include these mistakes. Therefore, we could argue that the predicted masks are more accurate than the true masks in some respects. However, in terms of IoU score the predicted masks will be punished for not including these mistakes.

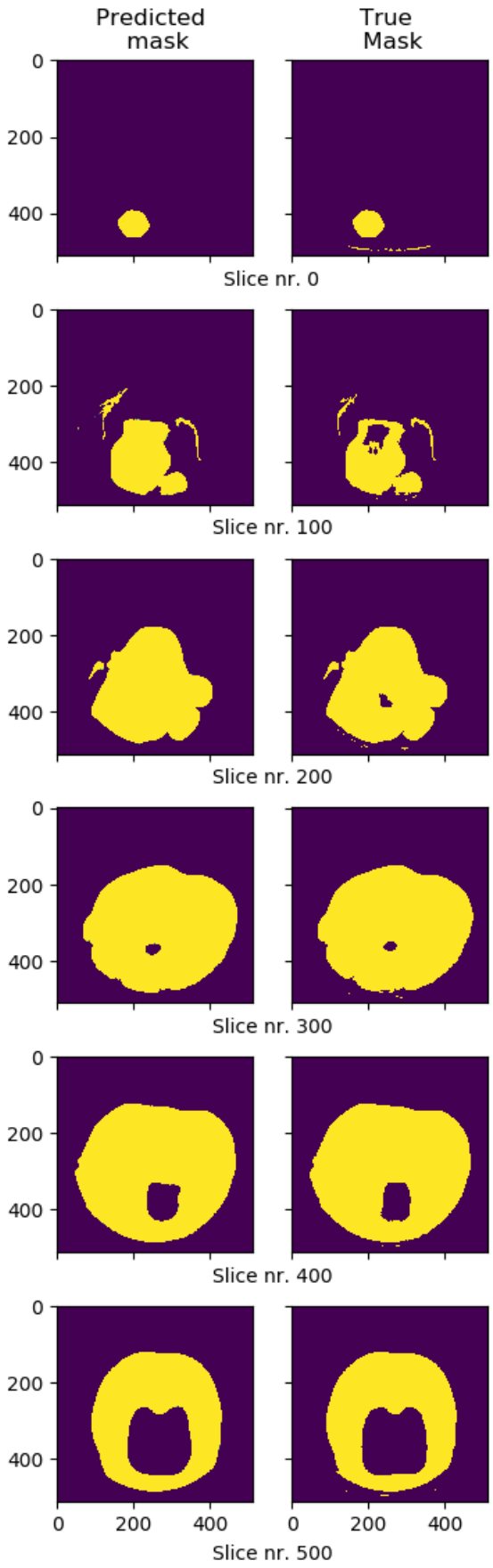


Figure 6.6: Predicted- and true masks in the transverse plane for pig with ID 21955

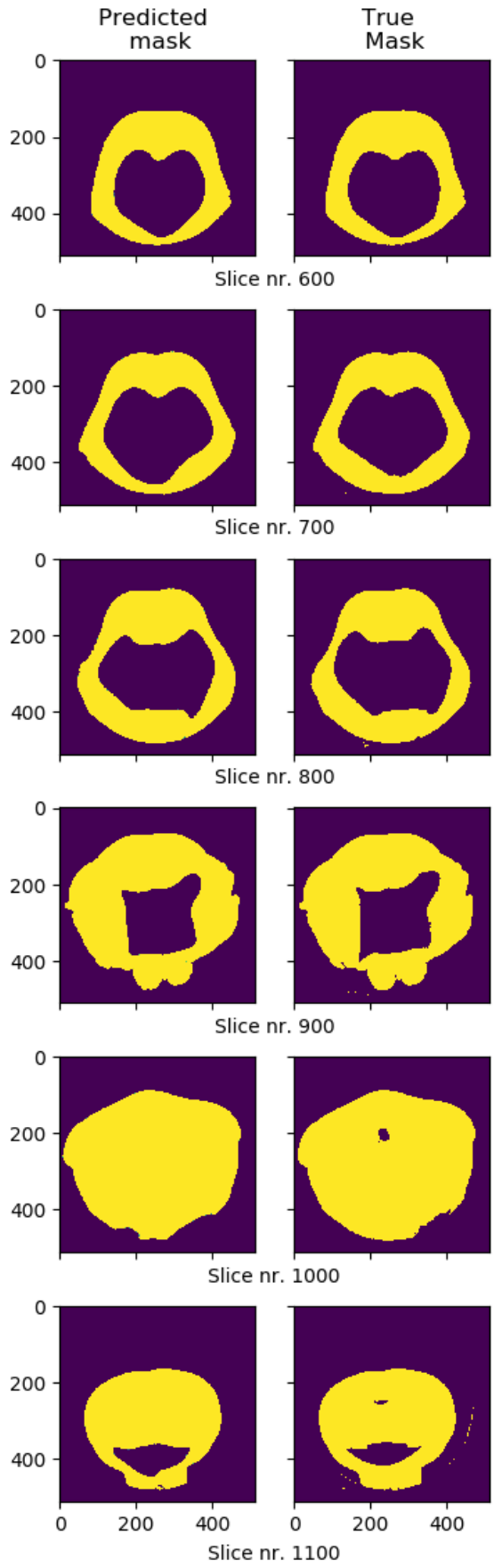


Figure 6.7: Predicted- and true masks in the transverse plane for pig with ID 21955

6.3.2 Sagittal plane

Figure 6.8 and 6.9 shows the predicted mask versus the true mask for a selection of slices in sagittal plane. We observe the same tendencies as earlier where the predicted outer boundaries of the structures are looking great, while the network seems to struggle on smaller "holes" within the larger structure. Boundaries within the structure are more rounded off than in the true masks. Remnants of the u-shaped container can also be seen in this plane.

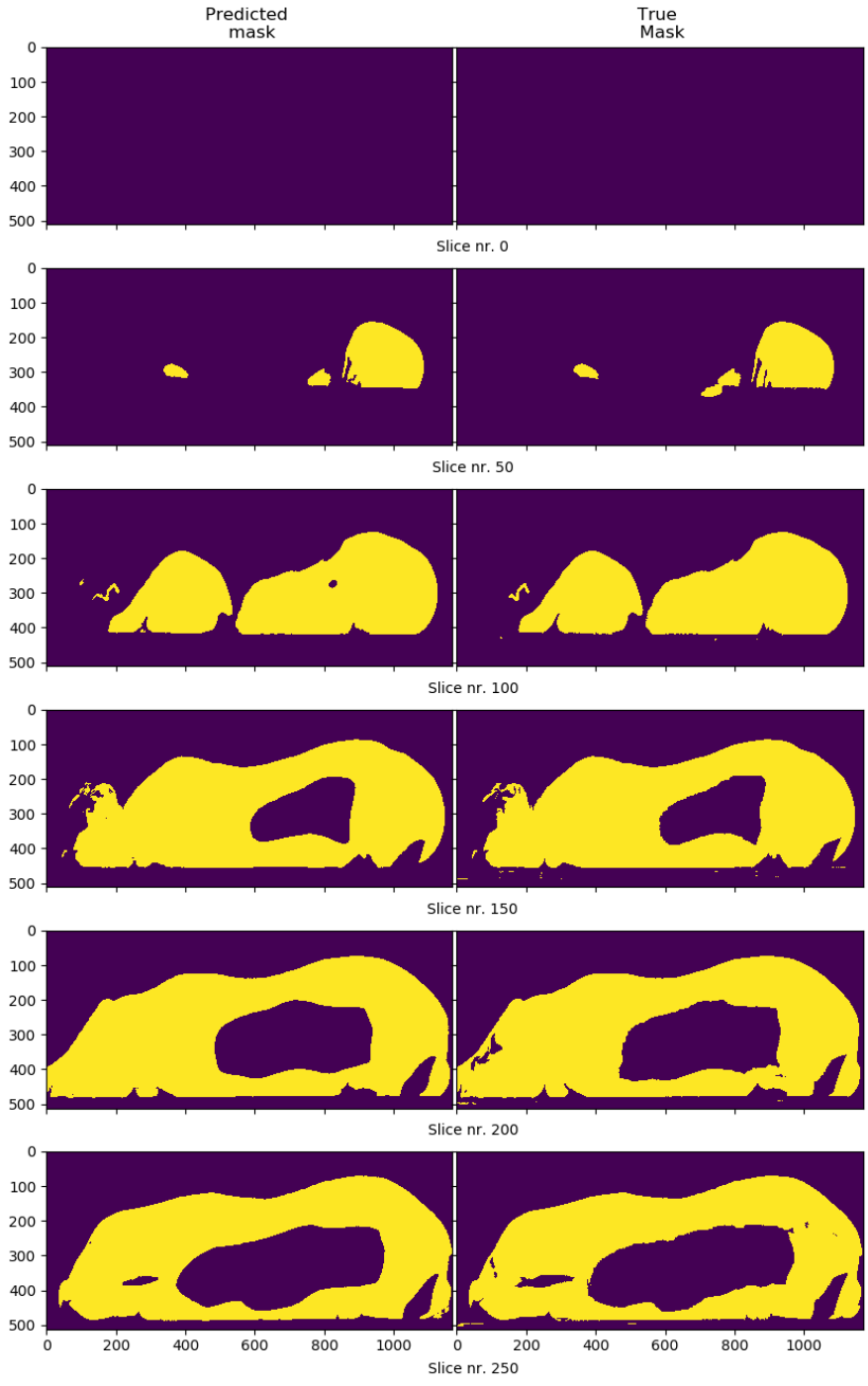


Figure 6.8: Predicted- and true masks in the sagittal plane for pig with ID 21955

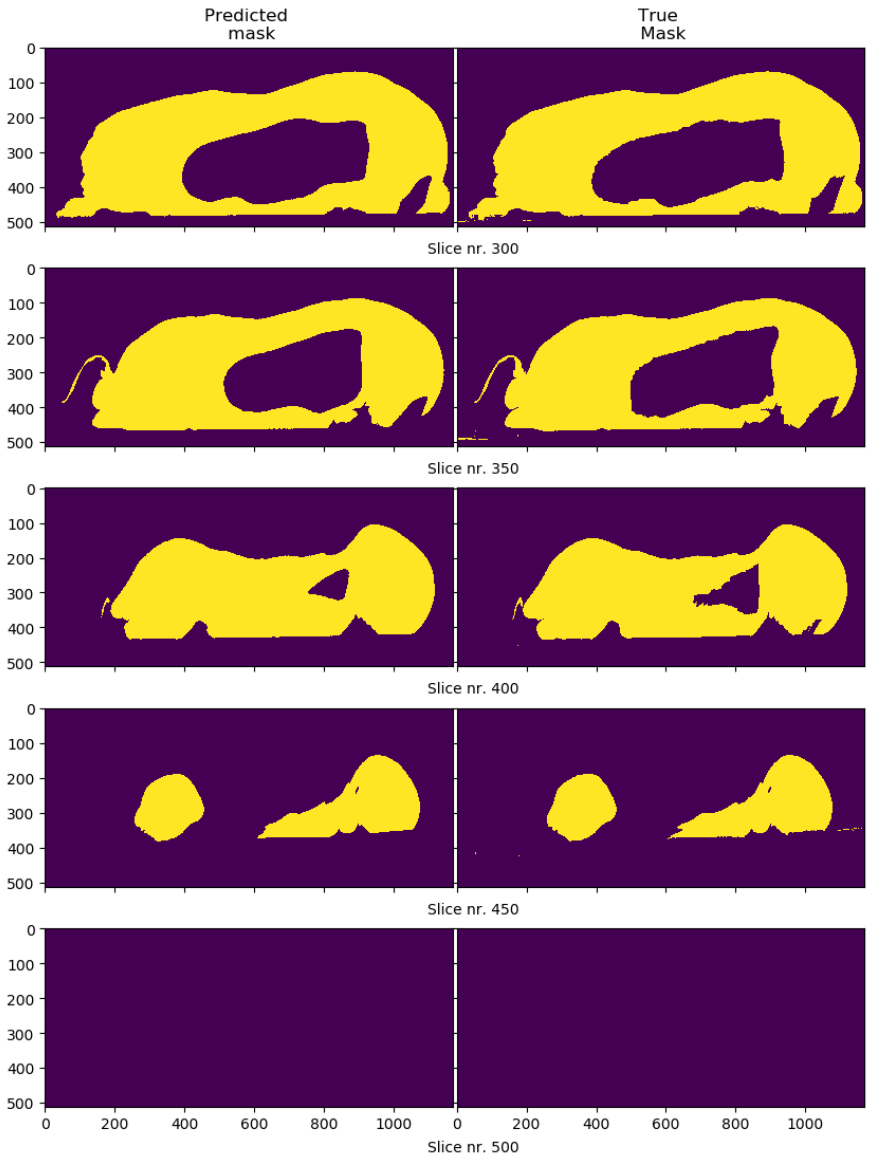


Figure 6.9: Predicted- and true masks in the sagittal plane for pig with ID 21955

6.3.3 Coronal plane

Figure 6.10 and 6.11 shows the predicted mask versus the true mask for a selection of slices in coronal plane. Nothing noticeably different from the other planes except for the serious mistakes on the bottom slice.

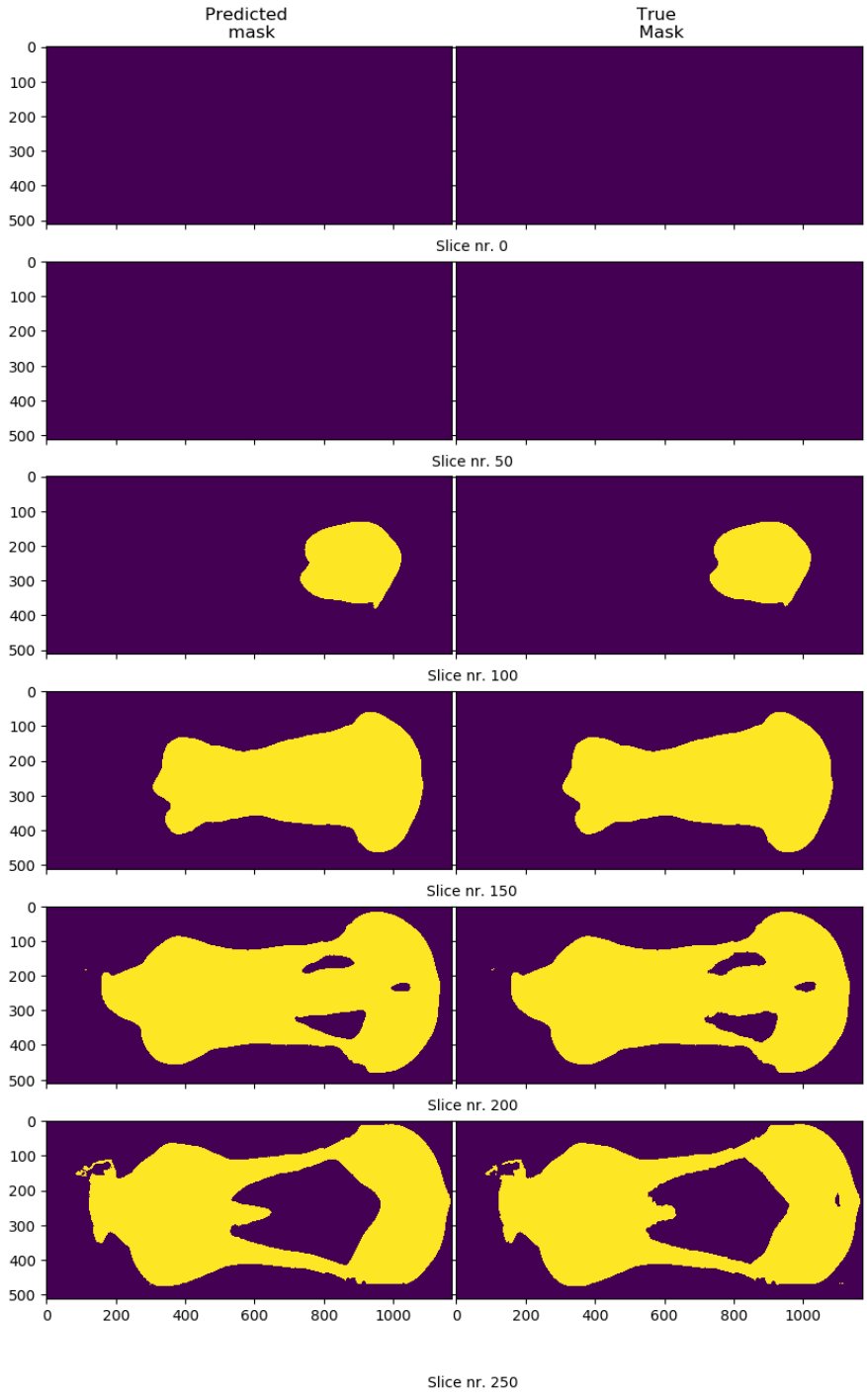


Figure 6.10: Predicted- and true masks in the coronal plane for pig with ID 21955

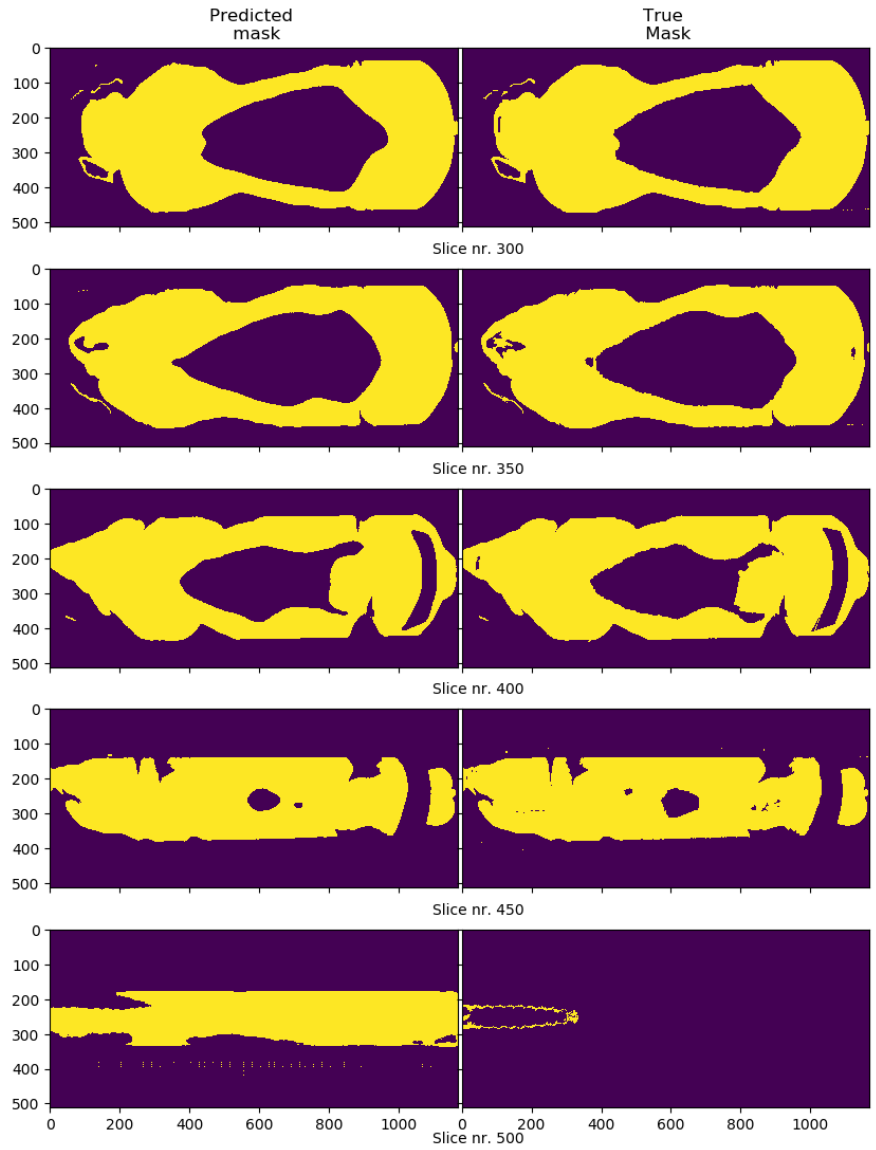


Figure 6.11: Predicted- and true masks in the coronal plane for pig with ID 21955

6.3.4 3D blocks

Figures 6.12, 6.14a, and 6.15a show how the 3D U-net architecture performed on the pig with ID 21955. As mentioned in section 4.5.2 a complete CT scan of the pig is too large for the 3D U-net. The CT scan is instead partitioned into blocks that the 3D U-net uses to predict masks. These predictions are then stitched back together. The figures shown are slices extracted from the resulting volume of predictions.

The boundaries of the blocks are clearly noticeable. The predicted masks show that the outer boundaries are decent, but that the 3D approach fails to provide more than a general location of "holes" within the structure. It is likely that the blocks (of size 96x96x96) were not large enough to provide sufficient context for the segmentation task. It is interesting to see that the network is performing well on the outside and poorly on the inside. It could be that it has a harder time distinguishing between the edible part of the pig (which the masks are marking) and the intestines (which are not included in the masks) than edible parts of the pig and the surrounding air.

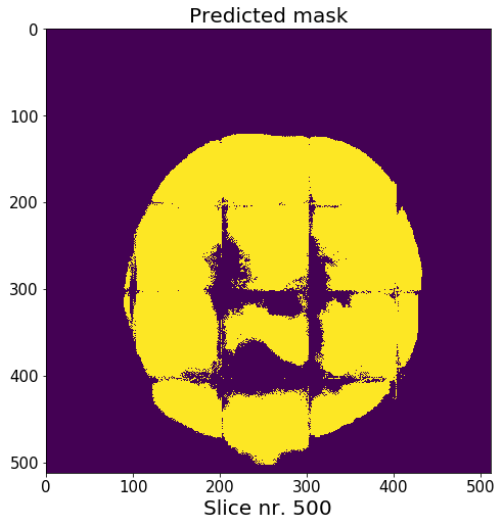


Figure 6.12: Predicted mask in transverse plane using the 3D U-net architecture.

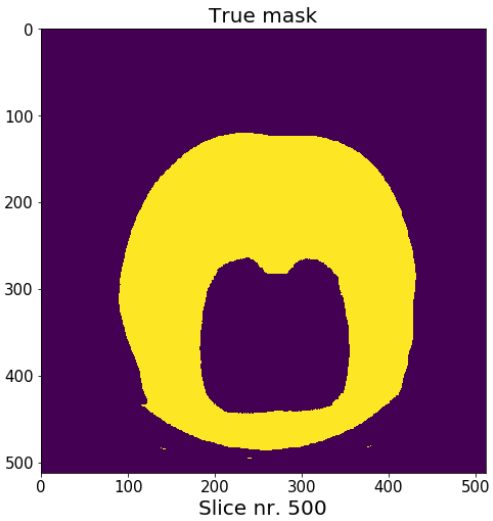
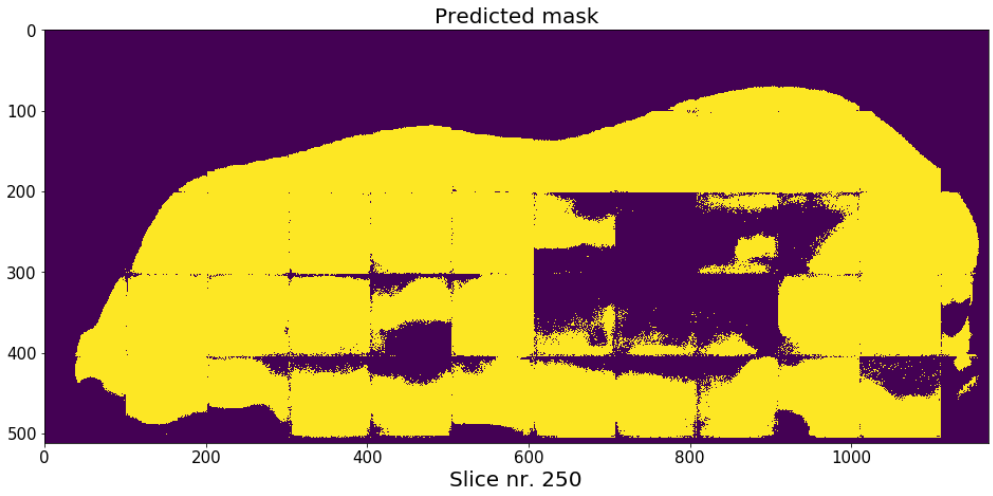
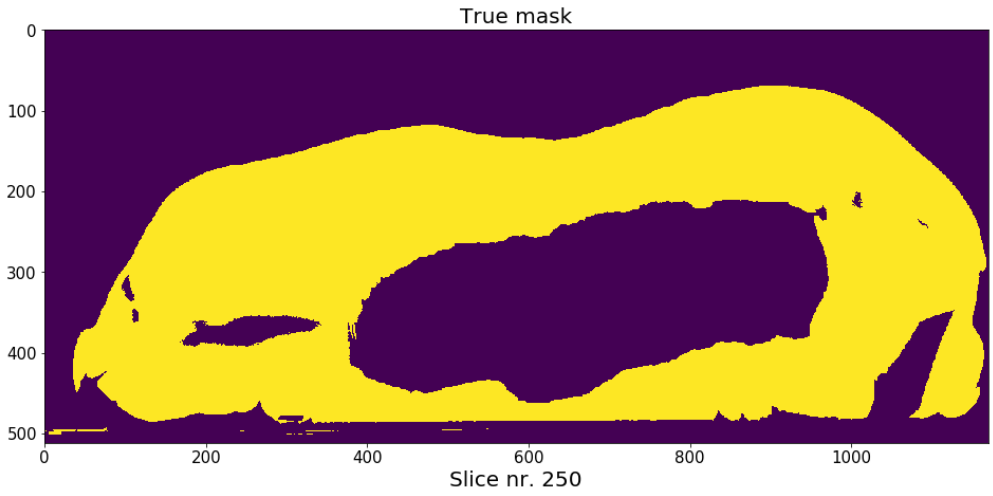


Figure 6.13: True mask for the slice in 6.12.

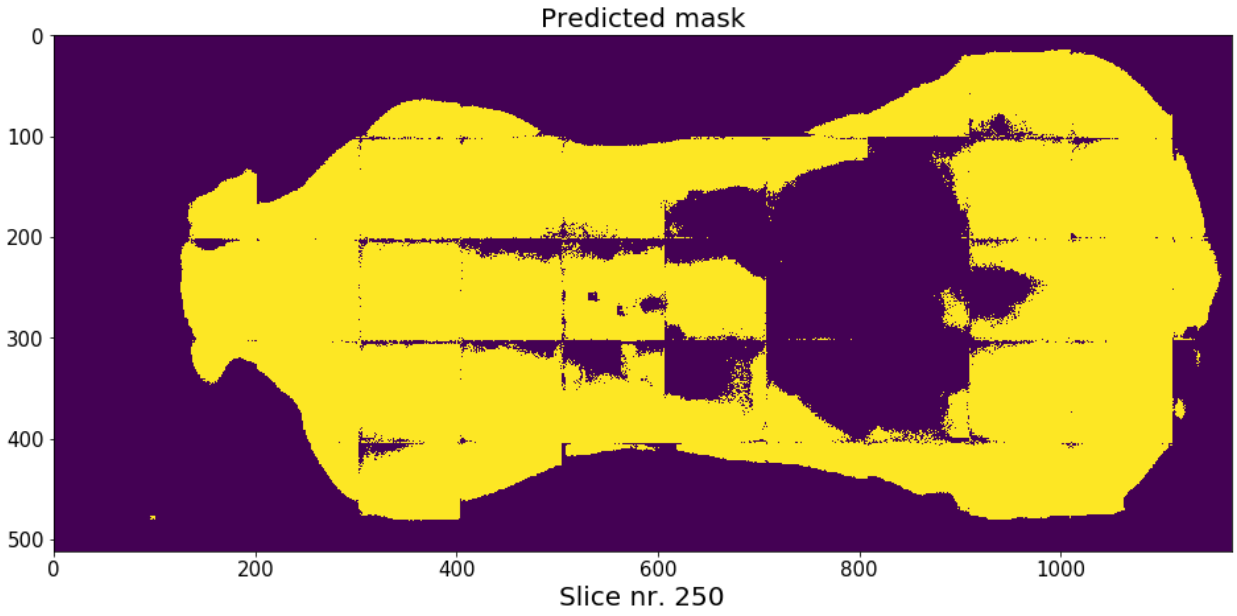


(a)

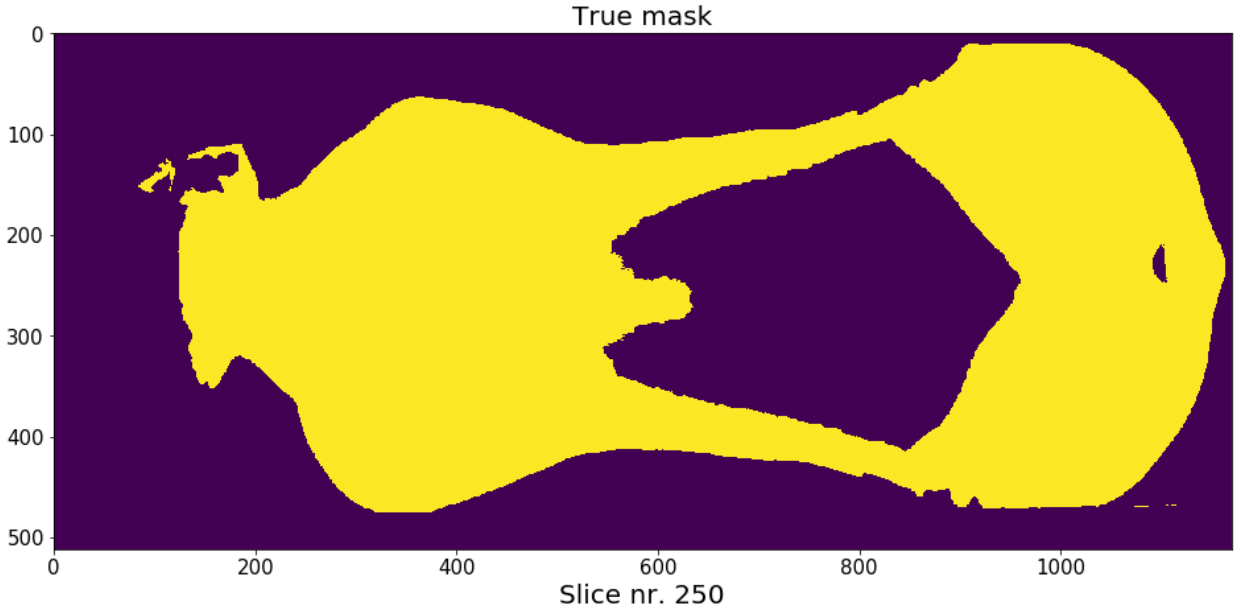


(b)

Figure 6.14: (a) Predicted mask in sagittal plane using the 3D U-net architecture. (b) True mask for the slice in (a)



(a)



(b)

Figure 6.15: (a) Predicted mask in coronal plane using the 3D U-net architecture. (b) True mask for the slice in (a)

6.4 U-net architecture comments

As seen in table 6.1 we objectively achieve overall good segmentation, but from the analysis and visual assessment we see a recurring performance drop in slices where there are mostly background pixels. This could possibly be attributed to the choice of loss function, namely cross-entropy. A brief reminder, the network is trained to minimize the cross-entropy loss and not the IoU score. The IoU score is merely a more intuitive evaluation metric used to measure the performance of the trained network. The cross-entropy function implemented in Keras evaluates the performance for each pixel in the prediction separately (using the mask) and computes the mean over all pixels. This means that each pixel is given equal weight in the learning process. Because there is a large majority of background pixels in the CT scans the cross-entropy function will cause the network to focus on correctly predicting the background class. The class weighting, see section 4.3.2, was meant to alleviate this effect, but it is possible that this was not sufficient to counter the effect.

In general, the chosen architecture is subjectively performing very well on the majority of slices, indicating that the architecture has enough expressive power to learn the relationship between the images and the masks.

Chapter 7

Conclusions and Future work

7.1 Conclusions

This thesis has shown that a modified U-net architecture (based on the work by [24]) with residual connections can effectively be applied to a semantic segmentation task on CT scans of pigs. It is difficult to clearly gauge the performance of the network without access to masks annotated by domain experts. However, quantitative and qualitative measures presented in this thesis clearly shows the effectiveness of using fully convolutional networks for semantic segmentation on this type of data.

U-net architectures were trained on 238 CT scans and subsequently evaluated on 37 CT scans. The objective was to develop an architecture that obtains the best possible segmentation according to the Index over Union metric. It seemed plausible that leveraging the information in the entire CT scan would result in the best segmentation. Because the GPU memory couldn't fit the entire scan we investigated multiple approaches of feeding the CT scan to the network. Networks were trained on 2D slices extracted from the 3D CT scan from either the transverse, sagittal, or coronal plane. A network was also trained using blocks of size 96x96x96 extracted from the CT scans.

It was found that on average the best performance was achieved by combining the predictions created by networks trained on 2D slices from all three planes. It is worth noting that combining the predictions yielded a relatively minor performance boost compared to networks only trained on slices in either the transverse- or sagittal plane.

The network trained on 3D blocks failed to achieve comparable performance, probably due to the limited receptive field of the blocks.

7.2 Future work

There were two pigs on which all networks performed noticeably worse because they were slightly different than what the networks had seen before. More training samples will almost always be advantageous to improve the general performance of any network and reduce the risk of encountering unseen data. Utilizing data augmentation techniques would be an effective method of generating these samples without collecting additional CT scans. Additional samples are then generated by modifying existing samples through rotation, cropping, mirroring, scaling and other methods.

In the discussion we saw how the automatically generated masks contained flaws. These flaws have likely impacted training. Better performance could be achieved by manually annotating masks or improving the automatic generation of the masks. These improved masks in combination with more samples could resolve some of the issues we saw in the bottom coronal slices where the network struggled.

There are slices in the CT scans that only contain background pixels because the scanner is larger than the pigs. If the network learns to recognize this they will easily obtain perfect segmentation on these slices which skews the mean IoU score for the scan. Removing these slices during pre-processing is something that could be investigated.

The U-net trained on 3D blocks performed poorly. Training on a machine with larger GPU memory where larger blocks can be used is likely to improve performance. In addition, it could be interesting to investigate interpolation techniques for combining overlapping blocks. Extracting overlapping blocks, make predictions, and effectively combine them could boost performance.

Using cross-entropy as loss function proved beneficial for the overall segmentation score on each pig, but could be the reason for why the network made errors on slices where there was a large class imbalance. A more advanced class weighting scheme or possibly another more specialized loss function for semantic investigation could reduce these errors.

The suggested network performs well on the binary segmentation task presented in this thesis. The next step would be to adapt the suggested architecture to a multiclass segmentation problem.

Bibliography

[1] Martin Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems." In: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467.

[2] Serge Beucher et al. "The watershed transformation applied to image segmentation." In: *SCANNING MICROSCOPY-SUPPLEMENT-* (1992), pp. 299–299.

[3] François Chollet et al. *Keras*. https://keras.io. 2015.

[4] Dan Ciresan et al. "Deep neural networks segment neuronal membranes in electron microscopy images." In: *Advances in neural information processing systems*. 2012, pp. 2843–2851.

[5] Michal Drozdzal et al. "The Importance of Skip Connections in Biomedical Image Segmentation." In: *CoRR* abs/1608.04117 (2016). arXiv: 1608.04117. URL: http://arxiv.org/abs/1608.04117.

[6] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning." In: *arXiv preprint arXiv:1603.07285* (2016).

[7] Lars Erik Gangsei et al. "Building an in vivo anatomical atlas to close the phenomic gap in animal breeding." In: *Computers and Electronics in Agriculture* 127 (2016), pp. 739–743.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[9] Simon S. Haykin. *Neural networks and learning machines*. Vol. 3. Pearson Education, Inc., Upper Saddle River, 2009.

[10] Kaiming He et al. "Deep Residual Learning for Image Recognition." In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[11] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. *Neural Networks for Machine Learning: Lecture 6a Overview of mini-batch gradient descent*. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, Retrieved on 2019-03-25.

[12] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks." In: *Neural networks* 4.2 (1991), pp. 251–257.

[13] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[14] Yann LeCun et al. "Gradient-based learning applied to document recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[15] M.H. Lev and R.G. Gonzalez. "17 - CT Angiography and CT Perfusion Imaging." In: *Brain Mapping: The Methods (Second Edition)*. Ed. by Arthur W. Toga and John C. Mazziotta. Second Edition. San Diego: Academic Press, 2002, pp. 427–484. ISBN: 978-0-12-693019-1. DOI: https://doi.org/10.1016/B978-012693019-1/50019-8. URL: http://www.sciencedirect.com/science/article/pii/B9780126930191500198.

[16] Fei-Fei Li, Justin Johnson, and Serena Yeung. *Lecture slides in detection and segmentation*. http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture11.pdf, Retrieved on 2019-04-08.

[17] Xiaomeng Li et al. “Deeply Supervised Rotation Equivariant Network for Lesion Segmentation in Dermoscopy Images.” In: *OR 2.0 Context-Aware Operating Theaters, Computer Assisted Robotic Endoscopy, Clinical Image-Based Procedures, and Skin Image Analysis*. Ed. by Danail Stoyanov et al. Cham: Springer International Publishing, 2018, pp. 235–243. ISBN: 978-3-030-01201-4.

[18] Tsung-Yi Lin et al. “Microsoft COCO: Common Objects in Context.” In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

[19] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully Convolutional Networks for Semantic Segmentation.” In: *CoRR* abs/1605.06211 (2016). arXiv: 1605.06211. URL: http://arxiv.org/abs/1605.06211.

[20] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity.” In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

[21] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. “V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation.” In: *CoRR* abs/1606.04797 (2016). arXiv: 1606.04797. URL: http://arxiv.org/abs/1606.04797.

[22] Vinod Nair and Geoffrey E Hinton. “Rectified linear units improve restricted boltzmann machines.” In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[23] Sebastian Raschka and Vahid Mirjalili. *Python machine learning*. Packt Publishing Ltd, 2017.

[24] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-net: Convolutional networks for biomedical image segmentation.” In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241.

[25] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton (Project Para)*. Cornell Aeronautical Laboratory, 1957.

[26] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. “Learning representations by back-propagating errors.” In: *Cognitive modeling* 5.3 (1988), p. 1.

[27] Peter Sadowski. “Notes on backpropagation.” In: (2016). URL: https://www.ics.uci.edu/~pjsadows/notes.pdf.

[28] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition.” In: *arXiv preprint arXiv:1409.1556* (2014).

[29] Naman D. Singh and Abhinav Dhall. “Clustering and Learning from Imbalanced Data.” In: *CoRR* abs/1811.00972 (2018). arXiv: 1811.00972. URL: http://arxiv.org/abs/1811.00972.

[30] Ilya Sutskever, Geoffrey E Hinton, and A Krizhevsky. “Imagenet classification with deep convolutional neural networks.” In: *Advances in neural information processing systems* (2012), pp. 1097–1105.

[31] Theano Development Team. “Theano: A Python framework for fast computation of mathematical expressions.” In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[32] SH Tsang. *Review: FCN (Semantic Segmentation)*. https://towardsdatascience.com/review-fcn-semantic-segmentation-eb8c9b50d2d1, Retrieved on 2019-04-014.

[33] Dong Yu et al. “An introduction to computational networks and the computational network toolkit.” In: *Microsoft Technical Report MSR-TR-2014–112* (2014).

[34] Matthew D Zeiler and Rob Fergus. “Visualizing and understanding convolutional networks.” In: *European conference on computer vision*. Springer. 2014, pp. 818–833.

[35] Yi-Tong Zhou and Rama Chellappa. “Computation of optical flow using a neural network.” In: *IEEE International Conference on Neural Networks*. Vol. 1998. 1988, pp. 71–78.

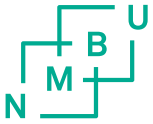
Appendix A

FCN architecture

```
from keras.models import Model
from keras.layers import Input, BatchNormalization, Activation, Add
from keras.layers.convolutional import Conv2D, Conv2DTranspose
from keras.layers.pooling import MaxPooling2D
from keras.layers.merge import concatenate
from keras.optimizers import RMSprop
from keras.activations import relu


def conv_block(conv_block_input, n_filters, kernel_size, strides=(1, 1), padding='same', activation=True):
    conv_block_input = Conv2D(filters=n_filters, kernel_size=kernel_size, strides=strides, padding=padding)(conv_block_input)
    conv_block_input = BatchNormalization()(conv_block_input)
    if activation:
        conv_block_input = Activation(relu)(conv_block_input)
    return conv_block_input


def residual_block(residual_block_input, n_filters):
    x = Activation(relu)(residual_block_input)
    x = BatchNormalization()(x)
    x = conv_block(conv_block_input=x, n_filters=n_filters, kernel_size=(3, 3))
    x = conv_block(conv_block_input=x, n_filters=n_filters, kernel_size=(3, 3), activation=False)
    x = Add()([x, residual_block_input])
    return x


def build_model(shape=(None, None, None), n_filters=16):
    input_layer = Input(shape)

    down_1 = Conv2D(filters=n_filters * 1, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(input_layer)
    down_1 = residual_block(residual_block_input=down_1, n_filters=n_filters * 1)
    down_1 = residual_block(residual_block_input=down_1, n_filters=n_filters * 1)
    down_1 = Activation(relu)(down_1)
    maxpool_1 = MaxPooling2D(strides=(2, 2))(down_1)

    down_2 = Conv2D(filters=n_filters * 2, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(maxpool_1)
    down_2 = residual_block(residual_block_input=down_2, n_filters=n_filters * 2)
    down_2 = residual_block(residual_block_input=down_2, n_filters=n_filters * 2)
    down_2 = Activation(relu)(down_2)
    maxpool_2 = MaxPooling2D(strides=(2, 2))(down_2)
```

```
down_3 = Conv2D(filters=n_filters * 4, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(maxpool_2)
down_3 = residual_block(residual_block_input=down_3, n_filters=n_filters * 4)
down_3 = residual_block(residual_block_input=down_3, n_filters=n_filters * 4)
down_3 = Activation(relu)(down_3)
pool3 = MaxPooling2D(strides=(2, 2))(down_3)

conv4 = Conv2D(filters=n_filters * 8, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(pool3)
conv4 = residual_block(residual_block_input=conv4, n_filters=n_filters * 8)
conv4 = residual_block(residual_block_input=conv4, n_filters=n_filters * 8)
conv4 = Activation(relu)(conv4)
pool4 = MaxPooling2D(strides=(2, 2))(conv4)

down_5 = Conv2D(filters=n_filters * 16, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(pool4)
down_5 = residual_block(residual_block_input=down_5, n_filters=n_filters * 16)
down_5 = residual_block(residual_block_input=down_5, n_filters=n_filters * 16)
down_5 = Activation(relu)(down_5)
pool5 = MaxPooling2D(strides=(2, 2))(down_5)

down_6 = Conv2D(filters=n_filters * 32, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(pool5)
down_6 = residual_block(residual_block_input=down_6, n_filters=n_filters * 32)
down_6 = residual_block(residual_block_input=down_6, n_filters=n_filters * 32)
down_6 = Activation(relu)(down_6)
pool6 = MaxPooling2D(strides=(2, 2))(down_6)

down_7 = Conv2D(filters=n_filters * 64, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(pool6)
down_7 = residual_block(residual_block_input=down_7, n_filters=n_filters * 64)
down_7 = residual_block(residual_block_input=down_7, n_filters=n_filters * 64)
down_7 = Activation(relu)(down_7)

transposed_conv_6 = Conv2DTranspose(filters=n_filters * 32, kernel_size=(3, 3), strides=(2, 2), padding="same")(down_7)
up_6 = concatenate([transposed_conv_6, down_6])
up_6 = Conv2D(filters=n_filters * 32, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(up_6)
up_6 = residual_block(residual_block_input=up_6, n_filters=n_filters * 32)
up_6 = residual_block(residual_block_input=up_6, n_filters=n_filters * 32)
up_6 = Activation(relu)(up_6)

transposed_conv_5 = Conv2DTranspose(filters=n_filters * 16, kernel_size=(3, 3), strides=(2, 2), padding="same")(up_6)
up_5 = concatenate([transposed_conv_5, down_5])
up_5 = Conv2D(filters=n_filters * 16, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(up_5)
up_5 = residual_block(residual_block_input=up_5, n_filters=n_filters * 16)
up_5 = residual_block(residual_block_input=up_5, n_filters=n_filters * 16)
```

```
up_5 = Activation(relu)(up_5)

transposed_conv_4 = Conv2DTranspose(filters=n_filters * 8, kernel_size=(3, 3), strides=(2, 2), padding="same")(up_5)
up4 = concatenate([transposed_conv_4, conv4])
up4 = Conv2D(filters=n_filters * 8, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(up4)
up4 = residual_block(residual_block_input=up4, n_filters=n_filters * 8)
up4 = residual_block(residual_block_input=up4, n_filters=n_filters * 8)
up4 = Activation(relu)(up4)

transposed_conv_3 = Conv2DTranspose(filters=n_filters * 4, kernel_size=(3, 3), strides=(2, 2), padding="same")(up4)
up_3 = concatenate([transposed_conv_3, down_3])
up_3 = Conv2D(filters=n_filters * 4, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(up_3)
up_3 = residual_block(residual_block_input=up_3, n_filters=n_filters * 4)
up_3 = residual_block(residual_block_input=up_3, n_filters=n_filters * 4)
up_3 = Activation(relu)(up_3)

transposed_conv_2 = Conv2DTranspose(filters=n_filters * 2, kernel_size=(3, 3), strides=(2, 2), padding="same")(up_3)
up_2 = concatenate([transposed_conv_2, down_2])
up_2 = Conv2D(filters=n_filters * 2, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(up_2)
up_2 = residual_block(residual_block_input=up_2, n_filters=n_filters * 2)
up_2 = residual_block(residual_block_input=up_2, n_filters=n_filters * 2)
up_2 = Activation(relu)(up_2)

transposed_conv_1 = Conv2DTranspose(filters=n_filters * 1, kernel_size=(3, 3), strides=(2, 2), padding="same")(up_2)
up_1 = concatenate([transposed_conv_1, down_1])
up_1 = Conv2D(filters=n_filters * 1, kernel_size=(3, 3), strides=(1, 1), activation=None, padding="same")(up_1)
up_1 = residual_block(residual_block_input=up_1, n_filters=n_filters * 1)
up_1 = residual_block(residual_block_input=up_1, n_filters=n_filters * 1)
up_1 = Activation(relu)(up_1)

output_layer = Conv2D(filters=1, kernel_size=(1, 1), strides=(1, 1), padding="same", activation="sigmoid")(up_1)

model = Model(inputs=[input_layer], outputs=[output_layer])

model.compile(optimizer=RMSprop(lr=1e-4), loss='binary_crossentropy', metrics=[iou])

return model
```



Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway