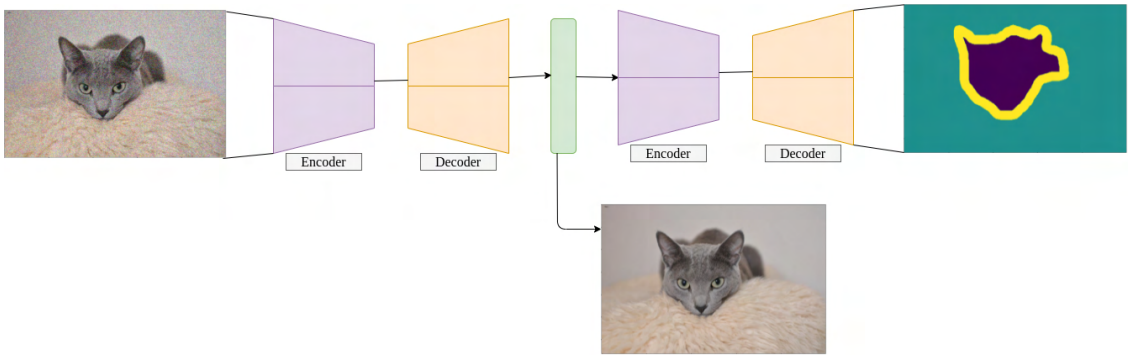
CHALMERS
UNIVERSITY OF TECHNOLOGY



Semantic Segmentation with Joint Segmenting and De-noising

Encoder-Decoder Networks with Dense Connected Layers

Master's thesis in Complex Adaptive Systems

Marios Aspris

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021

Master's thesis 2021

Semantic Segmentation with Joint Segmenting and De-noising

Encoder-Decoder Networks with Dense Connected Layers

Marios Aspris

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2021

Semantic Segmentation with Joint Segmenting and De-noising
Encoder-Decoder Networks with Dense Connected Layers
Marios Aspris

Supervisor: Professor Irene Yu-Hua Gu, Department of Electrical Engineering
Examiner: Professor Irene Yu-Hua Gu, Department of Electrical Engineering

Master's Thesis 2021
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Segmentation and denoising algorithm representation for predicting the final image and segmenting the relevant objects.

Typeset in LaTeX

Gothenburg, Sweden 2021

Semantic Segmentation with Joint Segmenting and De-noising

Encoder-Decoder Networks with Dense Connected Layers

MARIOS ASPRIS
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Convolutional networks have not yet been understood fully, and no mathematical theory or proof exists today why they work, nor why they are able to produce really good results. Early successful attempts in segmentation of images have used neural networks that downsample an image and upsample it hierarchically, and as a final output provide per pixel classification of a segment. In this method, an algorithm is constructed that uses two hierarchical encoder-decoder networks that are stacked, to jointly predict and reconstruct an input image with added Gaussian noise, and at the later stage to segment the image. The results show that the network is able to perform both tasks, and produce good overall performance.

Keywords: deep learning, convolutional networks, semantic segmentation, denoising, encoder-decoder.

Acknowledgements

For carrying out this report and research project I would like to thank first my supervisor, Professor Irene Yu-Hua Gu for allowing me to participate in this project and her useful guidance. I would like to thank her for her patience towards me and this project, and her push to towards results and completing the report.
I would also like to thank Chalmers University of Technology and Gothenburg University for the excellent quality of education they offer, and the diverse courses we are allowed to participate in.
Lastly, I want to thank my girlfriend, friends and family for their support and for listening to me explaining what this project was about, even without explicitly asking for information.

Marios Aspris, Gothenburg, July 2021

Contents

List of Figures

1

Introduction

Starting on the topic one can ask the question, what is segmentation? The problem of segmentation has a long history in the field of Computer Vision, with early attempts in 1970 to view the problem from a scientific perspective, and apply mathematical modelling. Segmentation of an image is the process of partitioning a numeric representation of an image, a digital image, into multiple sets of pixels depending on their properties. The process can be addressed by working with a compact representation of the interesting image data, that emphasizes the properties that make it interesting. Of what is interesting in the image and what is not, depends on the application of the problem.

In the process of working with a digital image, we want to assign a label $w\epsilon\{1, ..., M\}$ to each pixel indicating which of M objects is present, based on the local image data x. For example consider the Figure 1.1 below, that represents a road with automotive vehicles and a city in the background. In a segmentation task we want to separate for example the trees from the cars, the buildings from the road and so on. Therefore, we want to group in some way the pixels that specify the property of the object.

In the early attempts of solving and mathematically formulating the problem, the approach took a non-variation method. The first attempts consisted of trying to take all the pixels of an image and divide them in regions Ω_i, and then compare the variance of every union of pairs of regions Ω_i, Ω_j and merge them according to some threshold λ. This approach was not very successful, as there was no control of how the threshold was chosen. Modern and more recent attempts, make use of machine learning techniques.

The idea of grouping the parts of the image together originates from Gestalt psychology. Gestalt psychology is an attempt to understand the ability to acquire meaningful perceptions from a chaotic environment. Consider for example the Figure 1.2, where it represents three arrows with different directions. The image is split into two parts, the upper and lower part. If we consider observing just the upper part of the image, we can argue that the three arrows appear to be in different size, an assumption that can be proven false if we can see the lower part of the image where they are measured and compared. This optical illusion happens due to the fact that we cannot observe an individual arrow, without grouping the horizontal line with the two vertical lines at each end, since they appear continuous and form an individual object. Therefore, as an explanation that was given on this illusion, the trick happens due to depth that is recognized in the image. The arrows that

Figure 1.1: The above figure is taken from the BDD100K dataset that is made available by Berkeley University. On the figure we can see a full-frame segmentation of a road with automotive vehicles.

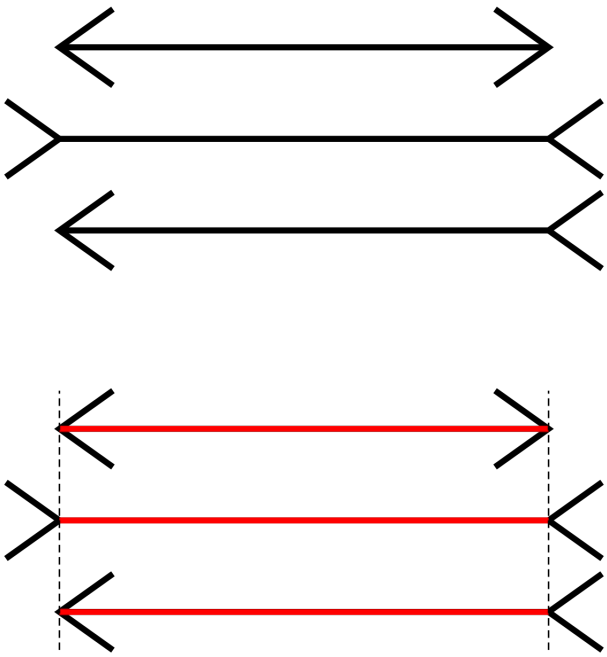


Figure 1.2: The above figure represents three arrows that appear to be of different sizes, which can be proven false if we consider the lower part of the image, where the arrows are actually measured in red color. This is also called the Müller-Lyer illusion. The figure is taken from the Wikipedia page.

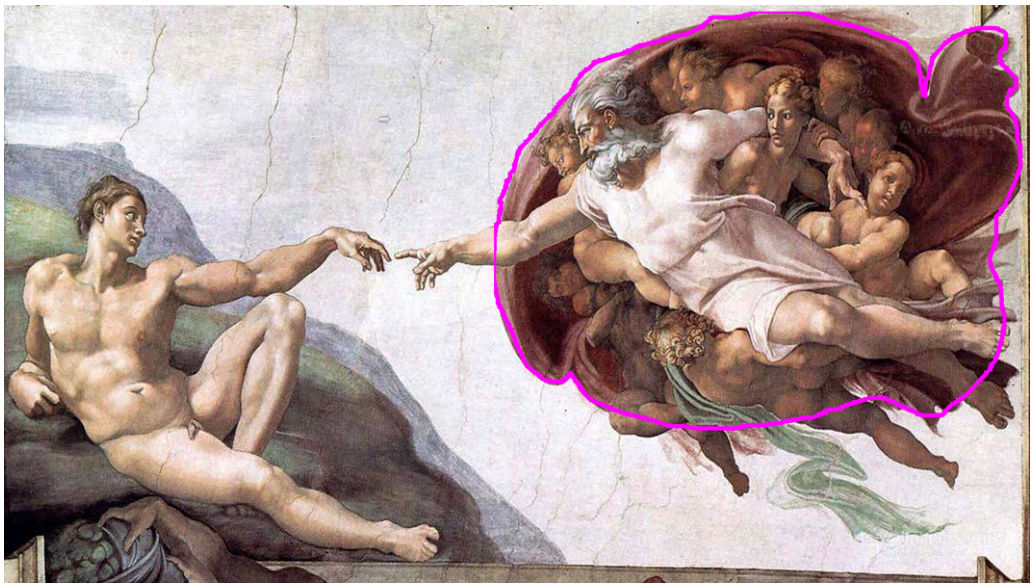


Figure 1.3: The above image is a painting made by the famous polymath Michelangelo in the year 1512. The painting is part of the Sistine Chapel ceiling, and it illustrates the Biblical creation narrative from the book Genesis in which God gives life to Adam, the first man. One might argue that the highlighted boundary can be seen as a depiction of the human brain. The highlighted boundary was created manually by the author.

angle in, give a sense of being closer to each other than the arrows that angle out, which we interpret as being further. This is known as the Müller-Lyer illusion.
It is worth noting that Gestalt theories are considered as experimental psychology, as quantitative research results to support the Gestalt ideas are lacking. In addition, even today nobody can claim to understand how perception works in human beings or animals and how we comprehend our surroundings, hence human or animal intelligence is not something we can declare to have properly defined or understood. For example, if we observe the representation on the Figure 1.3, one can argue that the highlighted boundary portrays a human brain. In order to observe and articulate such argument about the depiction of the human brain, we need to understand the context of the image and some form of associative learning takes place. It can also be ambiguous what the artist wanted to present.
Even if we have not currently reached the state to rigorously define intelligence, research in algorithms and mathematical formulations for Computer Vision have achieved significant results. As specified before, modern algorithms make use of machine learning, which in comparison to the early approaches, are based on the assumption that an algorithm is optimized with data, ie real images, rather than modelled explicitly. The name 'learning' has nothing to do with what we as humans can experience as acquiring new skills, knowledge or behaviors, it rather refers to the optimization of a function over a set of observations that we acquire and collect for a specific problem.
A sub part of machine learning, a method called deep learning has been a successful implementation and achieved important results on one of the hot topics of today,

Computer Vision.

1.1 Deep Learning; Success and modern research

Deep Learning[1] refers to artificial neural networks, which are computational models that are composed of multiple layers to learn representations of the data input, with multiple levels of abstractions. The model is optimized with the back-propagation algorithm, which allows it to change the internal parameters that are used to compute the representation in each layer from previous layers. What makes these models different from conventional and other successful machine learning algorithms, is their ability to be optimized on data without specific manipulations, and automatically discover the representations needed for the task. Previously, for data to be used with a machine learning model, they were manually manipulated and engineered in order to be transformed to a suitable internal representation. Successful implementations of these deep neural networks today make use the back propagation algorithm to be optimized. However, specialized implementations of these multi-layer abstractions are being used today, with two specific classes being successful; Recurrent Neural Networks and Convolutional Neural Networks.

1.1.1 Recurrent Neural Networks

Recurrent neural networks are a family of neural networks for processing sequential data. A RNN, is a neural network that is specialized and made for processing a sequence of values $x^{(1)}, x^{(2)}, ..., x^{(n)}$ and can scale to much longer sequences than would be practical for networks without sequence specialization. An example of such sequences would be representations of sentences of a text as numerical data. A sentence is considered as sequential data based on the distributional hypothesis, that "a word is characterized by the company it keeps". For example, consider the sentence "I went to the library in the morning". If this is the input data of a machine learning model and the task is to extract where did I go in the morning, it needs to understand the occurrences of the words in the sentence to recognize that I went in the library. A RNN takes advantage of parameter sharing across different parts of the model that makes it possible to apply the model to examples of different lengths. Parameter sharing is important for sequential data, as information for a particular point can occur at multiple positions within the sequence.
RNN operate on a sequence that contains vectors $x^{(t)}$. For example the equation below,

$$h^{(t)} = f(h^{(t-1)}, x^{(t-1)}; \theta) \tag{1.1}$$
$$= g^{(t)}(x^{(t)}, x^{(t-1)}, ..., x^{(1)}; \theta) \tag{1.2}$$

represents the unfolded recurrence after t steps. The function g^t takes the whole past sequence as input and produces the current state.
The forward propagation of the model is derived as follows and it can be observed

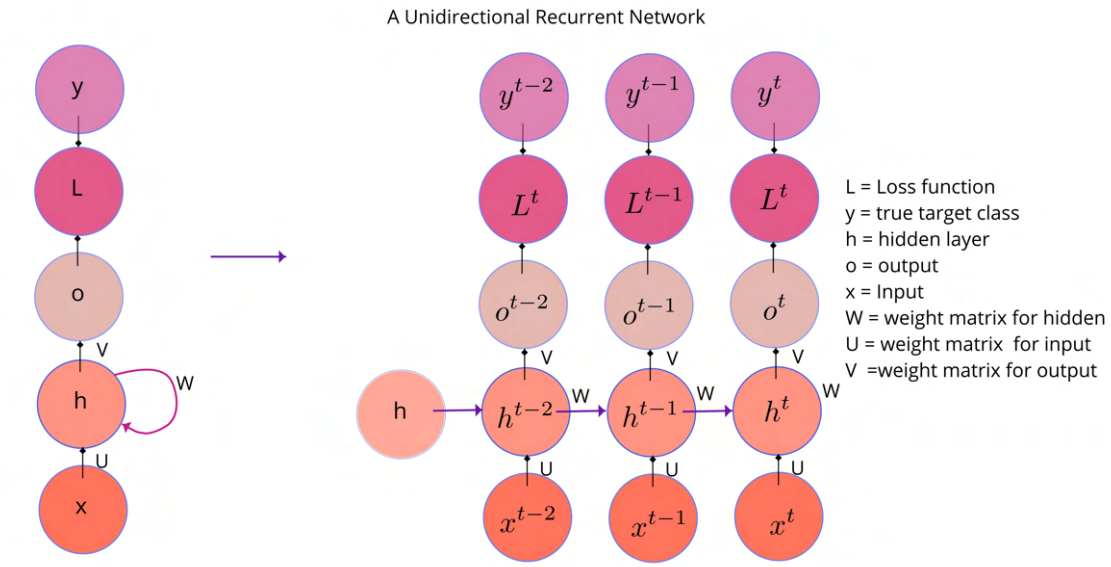


Figure 1.4: The above image represents the forward propagation of the unidirectional RNN that maps inputs x to corresponding outputs o. The Figure is created by the author.

visually on the figure 1.4,

$$i^{(t)} = b + Wh^{(t-1)} + Ux^t, \tag{1.3}$$

$$h^{(t)} = tanh(i^{(t)}), \tag{1.4}$$

$$o^{(t)} = c + Vh^{(t)}, \tag{1.5}$$

$$\hat{y}^{(t)} = softmax(o^{(t)}) \tag{1.6}$$

The equations are updated for each time step, and they begin with a specification of the initial state $h^{(0)}$. The parameters are the bias vectors **b**, **c** and weight matrices **W**, **U**, **V** respectively for hidden-to-hidden connections, input-to-hidden, and hidden-to-output connections. The loss function **L** is then paired with the output $\hat{y}$ and the true labels y, and summed for computing the total loss. The gradient computation for optimization involves performing a forward propagation pass moving from left to right as represented in the figure, followed by a backward propagation moving in the opposite direction.
The RNN depicted in the figure is unidirectional and has a causal structure, meaning that it captures only information from the past and the present input, left to right direction in the figure. In many applications, we want to capture the whole input sequence. RNN have been extended to bidirectional to address the need to input the whole sequence not depending on its direction. As the name suggests, it consists of two unidirectional RNN, one from left to right direction and one from right to left. They are stacked on top of each other both producing one output together.
The most demanding situation for RNN, is to build models that are robust and they need to have rich information about a state $h^{(t)}$. To address this, other forms

of the forward propagation of the sequence models have been considered, the most effective ones called gated RNN. These include the long short-term memory network and the more recent one, the gated recurrent unit.

1.1.2 Convolutional Neural Networks

Convolutional neural networks, much as recurrent neural networks, are a family of networks for processing values X on a grid, for example a digital image. CNN are not much different from traditional feed forward neural networks. They work more or less the same, and they have multiple layers of abstraction with non-linearities functions between them. What is fundamentally different is that instead of having a traditional matrix multiplication between the weights and the input, the weights are two dimensional convolution kernels and we have a mathematical operation called convolution operation between the input. This is shown on the Figure 1.5. For example, on the same figure a digital image of a dog consists of red green blue channel that gets concatenated and gets pixel values that represent the dog. If our weights are a five channel kernel, we do a convolution with the image and the output will be a five channel tensor with height and width given by the equation $[H, W] = [\frac{H+2p-f_H}{s} + 1, \frac{W+2p-f_W}{s} + 1]$. Then this is passed through a non linear function, usually a smooth function. The previous sequential operations would consist one layer of a CNN. The output of the CNN is sometimes also referred as a feature map.

The convolution operation mentioned here is the following formula for a two dimensional input such us a digital image,

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{1.7}$$

The symbol K here stands for the two dimensional kernel, such as the one used above. The convolution operation here has a lot of useful properties that have been exploited in the implementations of CNNs. First the operation is associative, meaning the following,

$$(a * b) * c = a * (b * c) \tag{1.8}$$

This might seem simple, but is very powerful as it allows us to use parallel implementations to compute such operations. This makes convolution operations parallel and ideal for such hardware, such as powerful heterogeneous hardware available today. More operations of the convolution are, the commutative property and the distributive property. The distributive property is,

$$a * b + a * c = a * (b + c) \tag{1.9}$$

and the commutative property is,

$$a * b = b * a \tag{1.10}$$

The operation described above make CNN an "embarrassingly" parallel task, a name that is given to tasks that are memory efficient and easy to implement in parallel tasks. This is very useful as it has provided a lot of tools to easily implement ideas and run successful implementations. On the bad side, it requires specialized hardware for implementation, which might come at high costs.

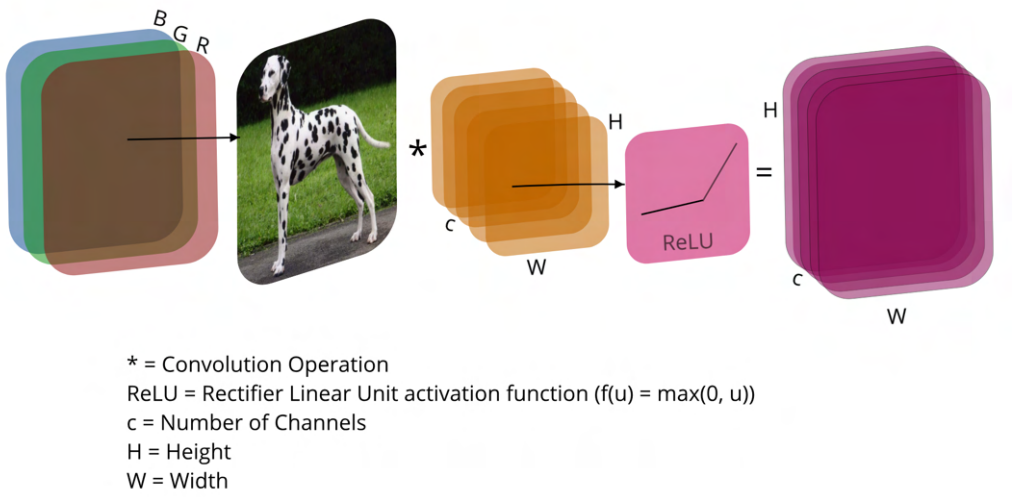


Figure 1.5: The above image represents a layer of a Convolutional Neural Network. The convolution operation is performed on the digital image of the dog, which is constituted from RGB channels that are concatenated to form the image. The 3 dimensional input, after convolution and activation operation takes place with the 5 channel kernel, it is transformed to a 5 channel grid. The height and width are then given from the equation $[H, W] = [\frac{H+2p-f_H}{s} + 1, \frac{W+2p-f_W}{s} + 1]$, where the parameter p is the padding, s is the stride, and $[f_W, f_H]$ are the height and width of the convolution kernel. The Figure was created by the author.

Figure 1.6: A representation of a cat in different positions in an image. This is to demonstrate that we can observe and understand that there is a cat in this image, regardless of its position. This means that we have translation invariance, a property of convolutional networks as well. The figure was taken from the web and modified by the author.

Convolutional networks, due to the fact that kernels constitute a layer as opposed to traditional multilayer perceptron networks, exploit the spatial locality as they enforce a local connectivity with the kernels. When a lot of these layers are stacked together with non-linearities in between, they become increasingly global.
Just as a recurrent network, they also have the weight sharing property. All units in a feature map share the same filter bank. This means that all the units in a given convolutional layer respond to the same motif within their specific response field. In a data array such as a digital image, local groups of values are often highly correlated. Replicating multiple units in this way allows for motifs to be detected regardless of their position in an image. This makes the CNN translation invariant which is demonstrated on Figure1.6. Often in a computer vision problem, we want a convolutional network that we will implement to be also rotation and illumination invariant, therefore it is common to augment the dataset we have and add more images to it, but also to increase the data to get more performance to it.

1.2 Different types of Segmentation in Computer Vision

Since the success of CNN in computer vision on supervised classification, it soon has been applied to segmentation tasks and has been very successful. This has allowed researchers to explore segmentation even further, which today when we refer to this task we can differentiate it in three different categories, Semantic Segmentation, Instance Segmentation and Panoptic Segmentation.

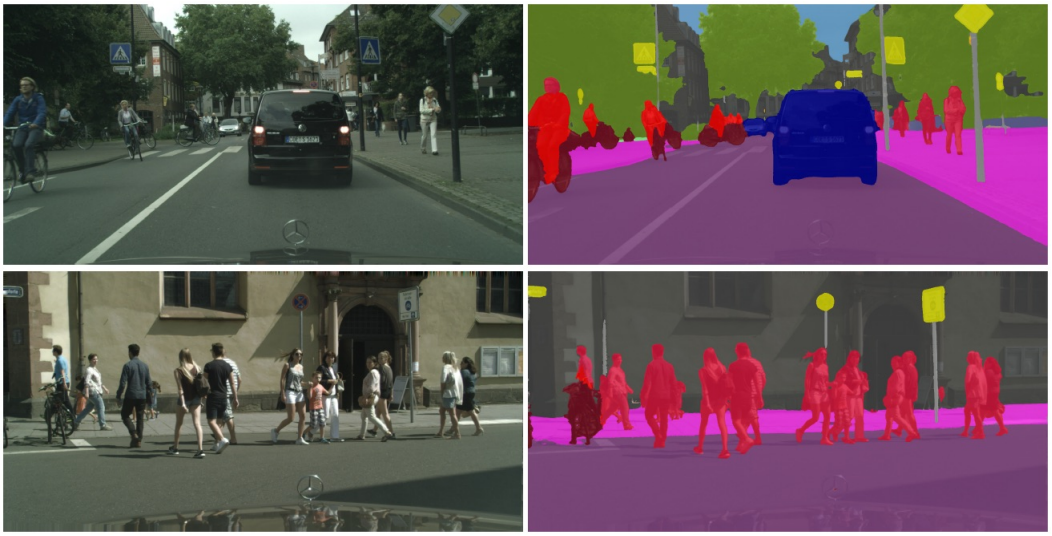


Figure 1.7: A picture taken form the Motion-based Segmentation and Recognition Dataset. On the left side of the figure you can observe an image taken from the real word, and on the right side is the segmented image. All the pixels in the image are assigned to a probability value of belonging to a specific class.

1.2.1 Semantic Segmentation

Now that we have an introduction to grouping of objects on images and to the early segmentation attempts, it is easy to define what is Semantic Segmentation. Semantic Segmentation in modern deep learning algorithms means that for each pixel of a digital image we assign a probability value to it, that belongs to a specific finite number of classes. Consider the Figure 1.7.

You can observe on the left side, two images from the real world and on the right side the images segmented. It is to observe that groups of objects, such as people, are separated into coherent parts in the images. This is achieved by the algorithm predicting the probability of each pixel to belong to each class.

The first approach that described a Convolutional Neural Network approach in segmentation, was to use a fully CNN. As the title specifies, throughout the network we use convolutions, with the final classification of the network being also a convolution. Specifically, a one-to-one convolution, meaning that we perform a convolution of the previous layer with kernel of dimension one on height, one on width and a finite number of channels that we specify. This happens on the final layers of the network, when the input has been down-sampled. With this operation, we can either increase or decrease the channel dimension and give an output of height and width rather than a vector. This helps in giving a probability value to every pixel and producing segmentation maps on the image. The Figure1.8 shows schematically the architecture of the CNN on this implementation.

1.2.2 Instance Segmentation

Instance Segmentation works somehow different than semantic segmentation. Here the requirements for a model are somehow different; we are not interested in seg-

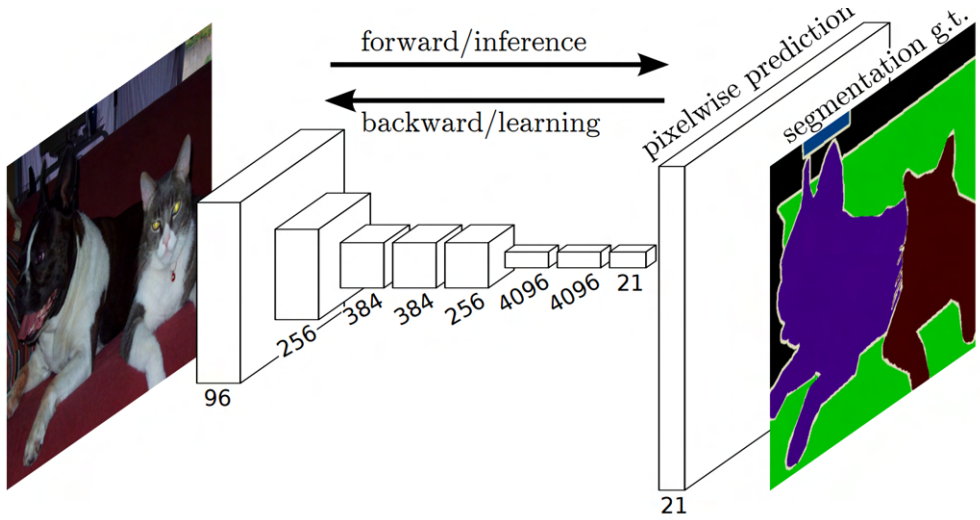


Figure 1.8: The image is taken from the paper "Fully Convolutional Network for Semantic Segmentation" [7]. It describes the architecture of the CNN to produce segmentation maps.

Figure 1.9: The image is taken from the paper "Fully Convolutional Instance-aware Semantic Segmentation" [15]. It shows their results on the test set of the MSCOCO dataset.

menting the whole image, but rather in only segmenting specific parts of it. This can be easily understood from the Figure 1.9. This problem is rather harder to surpass, as we cannot consider the previous model referred before. Instance segmentation requires to use a two stage model; on the first stage we need to find bounding box predictions of the targets and the second stage is to predict in the bounding box the pixels that belong to the object. In this way we can overlay a mask on the object we are interested in, such as the ones on the previous figure that segments the animals only and not the background.

1.2.3 Panoptic Segmentation

The last category of segmentation and probably the most interesting one is Panoptic Segmentation[8]. A panoptic segmentation dataset was recently been released with

the research starting from January 2018. This type of segmentation is considered a combination of the two types described before. For example we use semantic segmentation to segment the people on Figure 1.7, which are shown as one class meaning that every person in the image would go in a specific class, but in the Figure 1.9 each person in the image is considered a different class. For example they are countable.

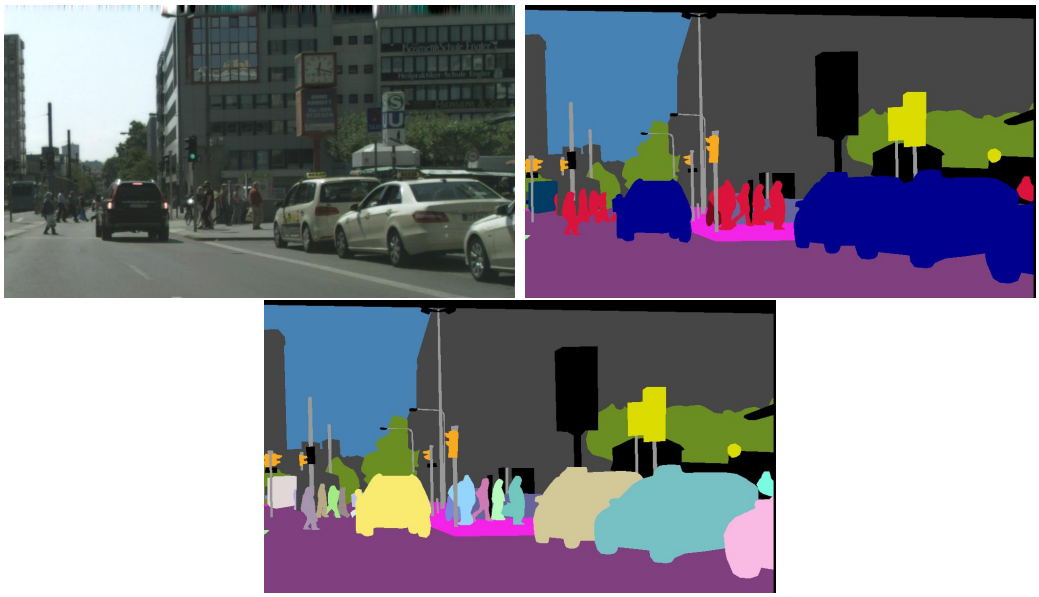


Figure 1.10: Panoptic segmentation of the image on the left. The upper right image represents semantic segmentation maps. On the bottom image you can clearly observe that every car is considered a different class, as well as the pedestrians on the street. Figure taken from the paper "Panoptic Segmentation" [8].

Consider now the Figure 1.10. On the upper right side is a semantic segmentation of the original image and the image below is a panoptic segmentation. You can observe the difference between them, for example on the bottom image you can count how many distinct cars are there, but with semantic segmentation it is possible but harder. You can observe that there is overlays of the segmented map of the car on to another. So in a way, panoptic segmentation combines instance segmentation and semantic segmentation. There is not much research on how this is done, since this challenge has arisen recently and most research is currently being carried out.

2

Theory

In the following sections, I will cover the basic theory in Semantic Segmentation using the proposed method. The proposed algorithm is a convolution network based on encoder and decoder building blocks that on a first stage predicts the input image with an added Gaussian noise and tries to recover the image by predicting the noise free pixels, and at a second stage, predicts the pixels that belong to a specific segment.

2.1 Classification Pre-trained Networks

2.1.1 Residual Networks and the vanishing gradient problem

The convolution operation in neural networks has been very successful, due to its properties and the flexibility it provides as a building block. Because of hardware acceleration and advancements in heterogeneous hardware accelerators such as General Purpose Graphics Programming Units, it has become possible to develop neural networks with a very large amount of layers stacked together, in order to decrease error rate and get better accuracy on specific tasks. The convolution operation is ideal for such hardware due to its nature as a so called "embarrassingly parallel problem", and also ideal for SIMD (Single Instruction - Multiple Data) based operations.
Since neural networks are optimized with gradient-based methods, the update of each layer in a neural network is received from a partial derivative of the error function, that is proportional since many layers exist before it. Hence, the proportionality effect of the partial derivative can be sometimes infinitesimal due to to many layers, therefore it would not update the weights of subsequent layers. This problem was described first in 1991, and it has been addressed differently in many cases, but a simple approach to it, is the residual network.
Residual networks are regular convolution networks that down sample an input image of specific dimensions, but are composed of residual convolution units where previous layers in the sequential process are added to subsequent layers. A descriptive figure of a residual unit is shown in the Figure 2.1. Therefore a forward propagation to the network for such a unit will be the subsequent:

$$x_l = H_l(x_{l-1}) + x_{l-1}. \tag{2.1}$$

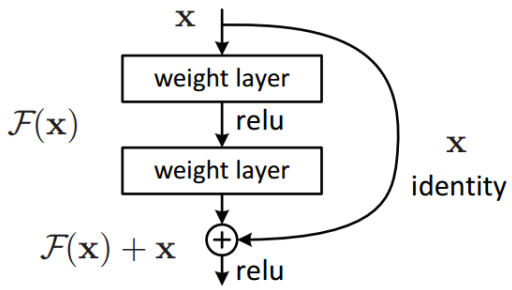


Figure 2.1: The above figure is taken from the paper "Deep Residual Learning for Image Recognition" [12]. The image is supposed to describe how a residual unit in the network would work.

where H_l is a composite function that consists of three consecutive operations: normalization, followed by a non-linear function of rectified linear units and a 3x3 convolution. The notation l denotes the layer index in the network. In the paper for the specific implementation, the authors named the addition x_{l-1} as identity mapping. In the backward propagation of the network for such a unit, for some defined loss function L will be:

$$\begin{aligned} y &= H_l(x_{l-1}) + x_{l-1} \\ &= H(x) + x. \end{aligned} \tag{2.2}$$

$$\begin{aligned} \frac{\partial L}{\partial y} &= \frac{\partial L}{\partial y} * \frac{\partial y}{\partial x} \\ &= \frac{\partial L}{\partial y} * (H_l'(x) + 1) \\ &= \frac{\partial L}{\partial y} * H_l'(x) + \frac{\partial L}{\partial y}. \end{aligned} \tag{2.3}$$

This simple approach using addition of previous layers has been proven to be effective in reducing loss in classification error, and also it was effective in adding more layers to networks. The Figure 2.2 demonstrates how their implementation looks like, and how it compares to another simpler implementation of a neural network.

2.1.2 Densely Connected Networks

Since Residual networks have demonstrated that an addition of the previous layers to subsequent layers can be simple and effective, "Densely Connected Networks" expanded on this idea. With residual networks, network architectures with many layers have been proven to give better results with decreased error rates and to even surpass human performance in classifying images on the classification competition called "ImageNet".
The idea in Dense Networks is to introduce a different way to add information from previous layers. All the previous layers would be connected to subsequent layer, by concatenating the feature maps together in comparison to Residual Networks that are added. A visual implementation of this can be seen in the Figure 2.3. By

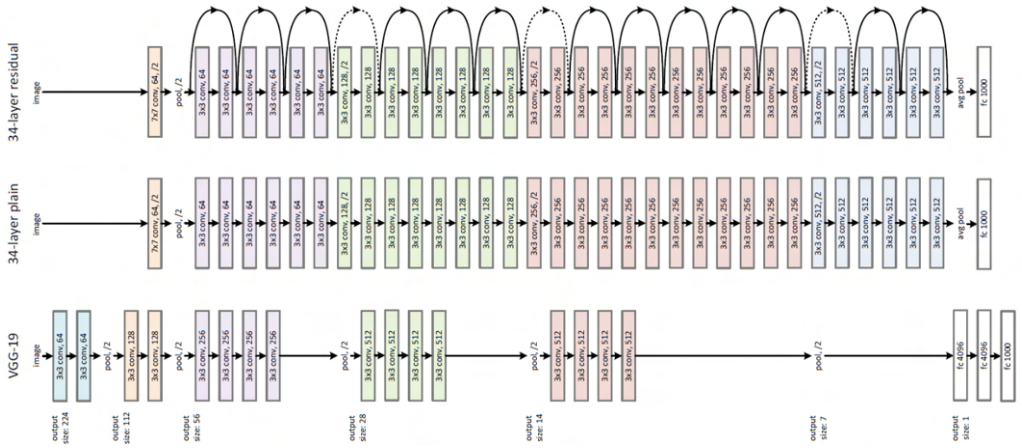


Figure 2.2: The above figure is taken from the paper "Deep Residual Learning for Image Recognition" [12]. In the figure show visually how their network compares to a different and more simpler network called VGG-19 which has 19 layers. They also compare how their implementation of 34 layer Residual Network would compare with plain layers.

concatenating feature maps here, each preceding layer will have more channels and a bigger tensor as input. A *lth* layer will receive and concatenate the feature maps from the previous *0, 1, ..., lth-1* layers.
As previously mentioned, if the same composite function H is defined for a layer then in Dense Networks will be defined as:

$$x_l = H_l([x_0, x_1, x_2, ..., x_{l-1}]). \tag{2.4}$$

where the operation for the layers up to x_{l-1} is concatenation, with the operation $[x_0, x_1, x_2, ..., x_{l-1}]$. The authors have shown that this approach can be effective in terms of error metrics, and allows to build models with higher number of layers. For example, 169 convolution layers with this approach will have lower number of total parameters than a Residual Network with 50 layers, while achieving a better error results on the ImageNet dataset. At the same time, they show better results in terms of the algorithm over-fitting a dataset, since layers can access features from its preceding layers.
In the algorithm implementation, I used this approach as a pre-trained network.

2.1.3 Transfer Learning

Transfer learning is the idea that when a function $f : X \rightarrow Y$ is optimized on some specific domain or dataset D, and is used to predict corresponding outputs $f(x)$ for some x, then the same optimized function can be further optimized on a different domain or dataset *D1*, and perform better than doing the optimization from scratch.
Similarly, this technique can easily be added in convolutional networks. Neural networks can be described as a function,

$$H(x) = g_l(g_{l-1}(...g(W * a + b))). \tag{2.5}$$

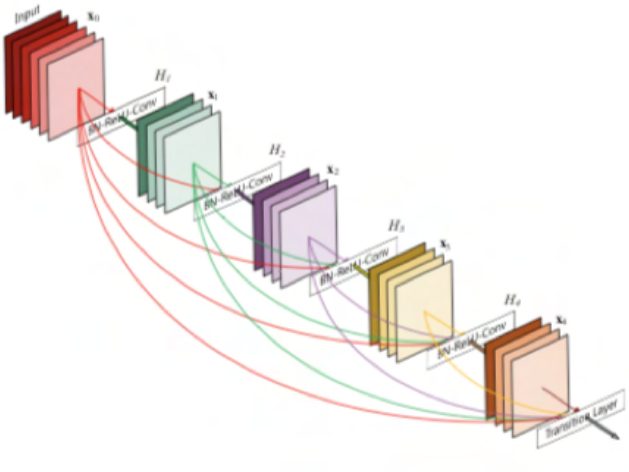


Figure 2.3: The Figure above is taken from the paper "Densely Connected Convolutional Networks" [13]. This demonstrates the connectivity of preceding layers with each other. Here, layers are concatenated together.

where
g_l denotes a non linear function, W is weight matrix that would be optimized, a is the input and b is a term that is added, usually called a bias term. Usually, the weight matrix values are initially set randomly from a normal probability distribution that is 0 centered and variance $\sqrt{\frac{2}{fan-in}}$, where $fan-in$ is the number of input units in the weight tensor. Therefore, we denote as $W \sim \mathcal{N}(0, \sqrt{\frac{2}{fan-in}})$. Once the algorithm is optimized on a specific dataset, the weight matrix will be optimized and can be saved in a specific format, where it can be reused again. An algorithm can be also extended once for example is initially optimized on a classification task, and further optimization can be created from reusing parts of the original algorithm.

2.2 Encoder Decoder Modules for Segmentation and Optimization

So far in the previous chapter, neural network algorithms that have been described were used and suited for a classification task, were the algorithm predicts a probability where a specific input image belongs to a certain class. This algorithms usually down-sample an input image of specific dimension to a specific size, which can be calculated based on how are the size of the convolution operations set on the different layers.
One of the first approaches to use convolutional networks for semantic segmentation is demonstrated in the Figure 1.8. The authors used a a pre-trained network that was optimized on the classification task of the ImageNet dataset, and then they further trained it to segment images. They took the original implementation, and removed from the algorithm the final layer of predicting probabilities of specific classes, and replaced it with a transposed convolution operation, that up-samples the final output back to the original size of the input image. The final output is

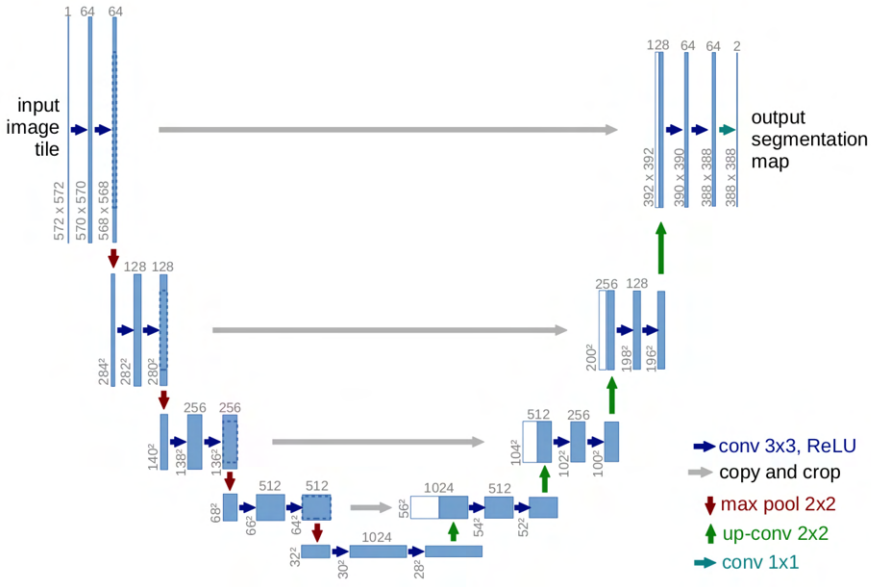


Figure 2.4: The Figure above is taken from the paper "U-Net: Convolutional Networks for Biomedical Image Segmentation" [11]. This demonstrates the connectivity between the layers that down-sample and up-sample the input image, and the encoder-decoder implementation.

then compared with a per-pixel logistic loss. The implementation has shown promising results. One other improvement in this implementation is that the algorithm does not have linear layers, therefore any matrix multiplication operation do not exist. This makes the algorithm faster to optimize or run the forward propagation, because matrix multiplication cannot be parallelized as efficient as the convolution operation.
Building on this approach, a similar implementation was used for segmentation called U-Net, but instead of just adding one up-sampling layer in the end to reconstruct the output to the original size of the image but with channel dimension as the number of classes, they have added several layers. The architecture of this approach can be seen in the Figure 2.4.

The network gets the name U-Net, most from the U shape that creates as shown in the Figure. The important part taken from this is the design of this approach, where the authors used an encoder-decoder architecture with interconnected layers in between with additive operation. The encoder is down-sampling the image, and the decoder is up-sampling the image. The implementation has been effective in this way, and transfer learning can be applied easily here in the encoder.

2.2.1 Up-sample Convolution Layers

Using a decoder for up-sampling operations, there are two types of layers that can be used to do this: transposed convolution or regular convolution. Transposed convolution can be considered as the gradient of convolution, since convolution matrix is transposed. The output of the operation in terms of dimensions would be given

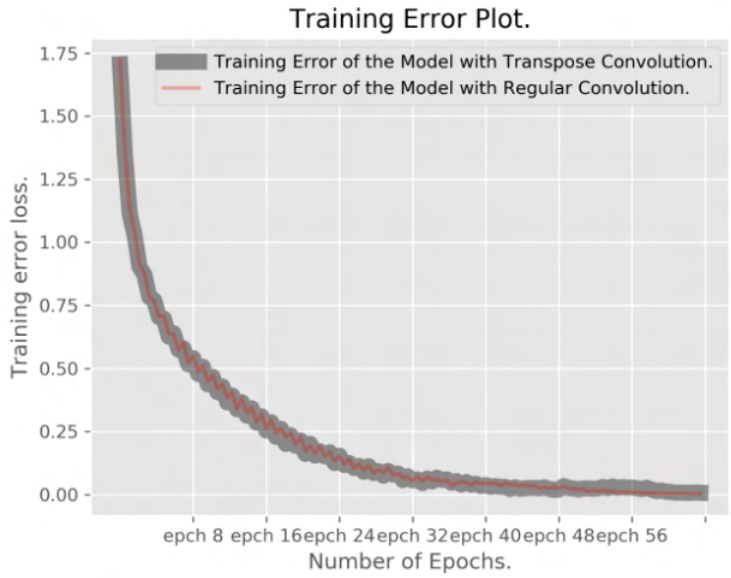


Figure 2.5: The Figure above demonstrates the error rate obtained by training a 3 layer classification network, with either regular convolutions or transposed convolutions. The network was optimized on the CIFAR-10 dataset. The results show that the operation are in general, similar and can produce comparable results. The figure is created by the author.

by the following: $[H, W] = [s * (H - 1) + f_H - 2 * p,\ s * (W - 1) + f_W - 2 * p]$, where H, W is height and width of the input image, s is the stride, p is padding and f is the size of the convolution kernel. With this in mind, the operation can be flexible in adjusting how the size grows.
Some further research on the usability of the transposed convolution has shown that it can be effective, but a better approach is to use first a bi-linear interpolation to up-sample a two dimensional image and then use a convolution operation. This approach has shown improved results.
This is mostly experimental, it has not been proven why it should be preferred. To compare the transposed convolution with the regular convolution, I have performed an experiment with a simple three layer classification network, where I used the same network with regular convolution and with transposed convolution and try to compare the error rates. The results can be seen on Figure 2.5. The classification network was optimized on the CIFAR-10 dataset, and the error rate seems to converge to similar results. This is not enough however to conclude which operation is to be used.

2.2.2 Optimization based on predicting pixel values

To effectively use the final algorithm based on the encoder-decoder architecture, the final layer of the decoder, that has an output size the same as the input size, either regress each value in the final output buffer to a specific continue value, or to a probability value. Therefore, the network outputs a $\hat{y}$ which it would be an estimate

of the true y. If the pixel outputs from the network are probability values, then this is considered a classification network. The final layer of the network would pass through either a softmax function $s \in \mathbb{R} \rightarrow [0, 1]$ given as

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_J}}. \tag{2.6}$$

with K, the number of classes in this case that we are trying to classify into. If the number of classes that exist in the segmentation are more than 2 then this function is used, for multi-class classification. With multi-class classification, a pixel value can belong to either the foreground, background, or the different types of objects.
If the pixel outputs are binary, then the final layer of the network will pass pass through a sigmoid function $\sigma \in \mathbb{R} \rightarrow [0, 1]$ given as,

$$Sigmoid(x_i) = \frac{1}{1 + e^{x_i}}. \tag{2.7}$$

This would classify for example pixel values of probability of belonging either to background or foreground in an image.
To optimize the network based on the final classification layer, for the softmax function we use the equation,

$$-\sum_{c=1}^{K} y_c log(\hat{y_c}). \tag{2.8}$$

and for the sigmoid the binary cross entropy loss function,

$$-(ylog(\hat{y}) + (1 - y)log(1 - \hat{y})). \tag{2.9}$$

However, apart from classifying each pixel value into a probability, the network can be used to predict the pixel values. By doing this, you can keep the final layer of the decoder as it is, and output continues values. The estimate of y, $\hat{y}$ would then be compared with the Smooth L_1 Loss function given from the equation,

$$L(y, \hat{y}) = \left\{ \begin{array}{ll} \frac{0.5*(y-\hat{y})^2}{beta}, & if|y - \hat{y}| < beta \\ |y - \hat{y}| - 0.5 * beta, & otherwise \end{array} \right\} \tag{2.10}$$

The reason why this might be preferred over mean squared difference $mean(y - \hat{y})^2$ is because L_1 smooth loss is less sensitive to outliers. This function is also called Huber loss. This way of estimating a loss and optimizing the model was used for object detection in images, for finding the bounding box around an object of interest. It is also worth mentioning that this method is be faster in terms of optimization or running the forward propagation rather than the classification functions, due to the fact that it does not add the expensive operation of the exponential.

2.2.3 De-noising an Image and Segmenting

De-noising an image in general has been explored by a lot of previous research. A successful algorithm can have a large number of applications in computer vision or

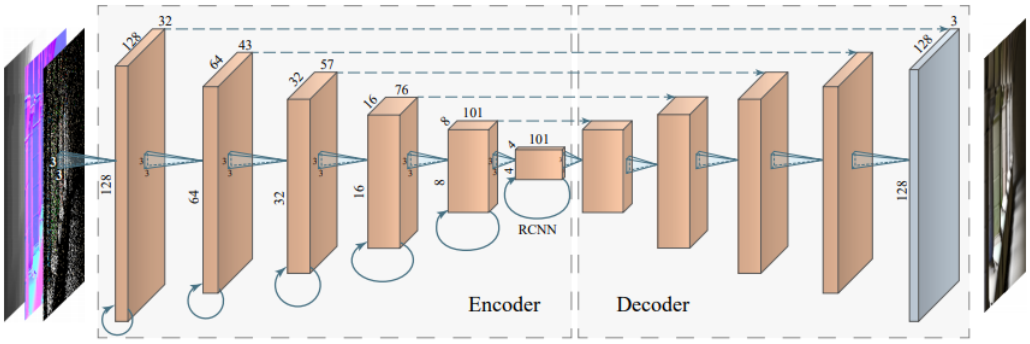


Figure 2.6: The Figure above is taken from the paper "Interactive Reconstruction of Monte Carlo Image Sequences using Recurrent Denoising Autoencoder" [9]. The Figure demonstrates the encoder-decoder architecture used, and how they used the algorithm to denoise rendered images using different input buffers.

computer graphics. One application where de-noising was used, based on an encoder decoder architecture, was in computer graphics were the researchers try to optimize an algorithm to de-noise an image that was rendered using path tracing, based on 3D scene. The algorithm implementation can be demonstrated in the Figure 2.6.

In this implementation, the authors rendered a 3D geometry with a path tracing algorithm which produced a noisy image. For their input to the network, they had the path traced high dynamic range RGB noisy image, the view-space shading normal maps, the depth of the scene as a heat map, and an image that represents material roughness of the geometry in the scene. In total they used 7 channels as inputs. The reason for using such a high input buffer, it's because this provides useful information about the scene in order for it to be denoised better. As a loss function, they used a multi task loss function consisting of,

$$L = 0.8 * L_{1smooth}(y, \hat{y}) + 0.1 * L_{1smooth}(\frac{\partial y}{\partial t}, \frac{\partial \hat{y}}{\partial t}) + 0.1 * L_{1smooth}(\nabla P, \nabla \hat{P}). \quad (2.11)$$

where ∇P is a form of Laplacian smoothing on the ground truth image in order to get better edges, and $\frac{\partial y}{\partial t}$ is considered the discrete difference between the previous and current frame. The authors used this multitask loss as they explained it produced better results and helped to solve better artifacts produced in an image. Also, the encoder of their network had recurrent connections due to the fact that their dataset was sequential, therefore they used a loss function for comparing previous and current frame.

Following this implementation of encoder decoder architecture of denoising an image, we can construct and take a similar approach in segmenting an image based on first denoising an image and secondly using that output to the same network to segment the image. In simple terms, stacking two encoder decoder networks together, with the first having a task in denoising the image based on predicting pixel values, and the second to segment the image based on also predicting pixel values based on segments. Then the loss of the function would be as follows,

$$L(y, \hat{y}) = L_{1smooth}(y, \hat{y}_{denoised}) + L_{1smooth}(y_{segmented}, \hat{y}_{denoised+segmented}). \quad (2.12)$$

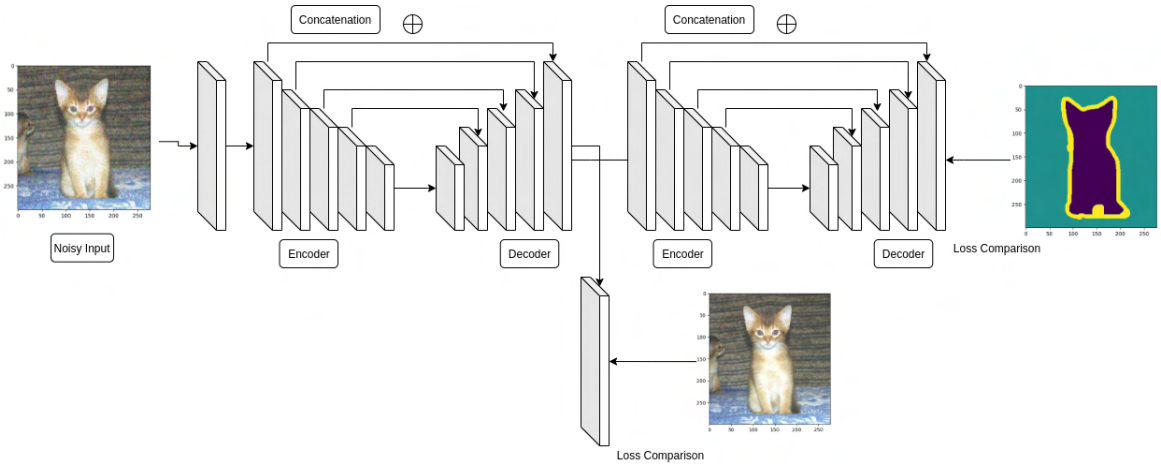


Figure 2.7: The Figure above demonstrates how the final algorithm is implemented. The algorithm takes a noisy image as input, predicts and denoises the image, and as a final stage segments the image. The figure is created by the author.

A descriptive figure for of this algorithm is shown at the Figure 2.7.

3
Methods

In the following part I describe the methods and implementation I used to create the algorithm based on the theory described before. In the Figure 3.1, I demonstrate the pipeline for optimizing the network using the software library PyTorch on the current dataset and at the same time monitor the hardware.

3.1 The Oxford-IIIT Pet Dataset

Selecting a dataset for optimizing neural networks can have huge impacts on the final algorithm and how it will perform or generalize. That is because neural networks are like a compression of the dataset in the algorithm. Generalization is a big topic, in general to make useful algorithms and avoid bias. Not considering diversity in a dataset, or what it represents, can have a negative impact and sometimes harmful effects if an algorithm is released as a product that everyday people might use.
For this implementation, I have considered a dataset that contains cats and dogs from 37 categories with around 200 images for each class. All the data contain annotations at the pixel level, and the annotations are trimaps on pixel level. Trimaps for segmentation of the images mean that the annotation contains pixels that are marked as the background of the images, foreground in the images, and an intermediate region separating foreground and background. In Figure 3.2 this is shown on a specific species of a cat.
Figure 3.3 demonstrates a similar sample from the dog category. In total there are 7384 number of samples from both directories.

3.2 Hardware

Hardware and heterogeneous compute units are essential for the optimization of the algorithm. In my current implementation I have tried different approaches for the task, and in the section I am describing free online resources as well as paid or local ones that I have tried.

3.2.1 Local Heterogeneous Compute Units

As a first approach I used a local resource hardware, which I own and consists of a 32GB Ram 6 core Intel CPU, and a Nvidia RTX 2070 that has a total memory of 8GB of memory. Local hardware has the benefit of having the data locally,

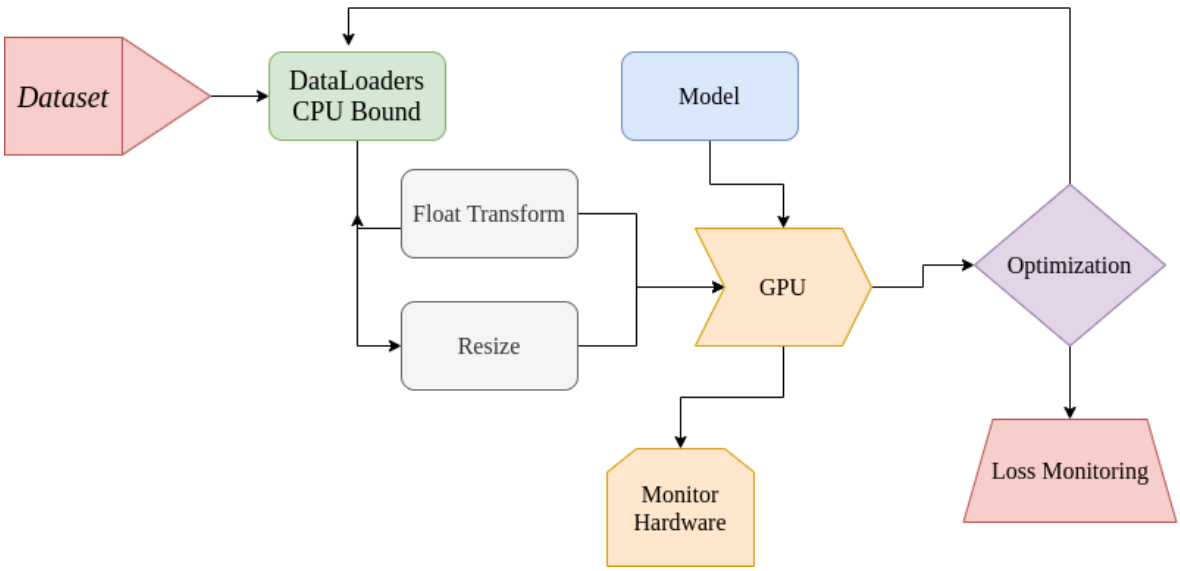


Figure 3.1: The Figure above demonstrates the pipeline of the methods for optimizing the algorithm on the dataset. Here is demonstrated how the optimization loop is constructed by using the CPU to fetch data to the GPU and use the model to optimize it.

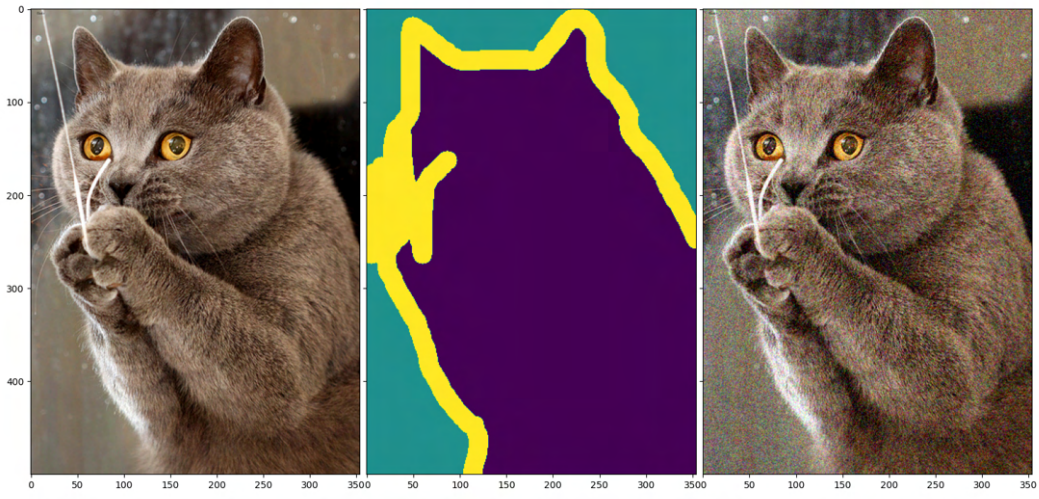


Figure 3.2: The Figure above demonstrates a sample from The Oxford-IIIT Pet Dataset. From left to right one can see the unfiltered image, the trimap segmentation annotations and the unfiltered image with added Gaussian noise to each channel. The figure is taken from the dataset and modified.

Figure 3.3: The Figure above demonstrates a sample from The Oxford-IIIT Pet Dataset from the dog category. From left to right is shown the unfiltered image, the trimap segmentation annotations and the unfiltered image with added Gaussian noise to each channel. The figure is taken directly from the dataset and modified.

which in the case of pictures can be viewed and run test appropriately. Another benefit of owning local hardware, as oppose to cloud resources, is that hardware are not virtualized which means that you have benefits over locality on the CPU and the GPU. Processes would generally run faster. At the same time you can control drivers for the external hardware and easily install the latest API libraries to use the hardware efficiently.

3.2.2 Free Cloud Resources

Since hardware is rather expensive, for an individual to own such hardware, can have very high costs with not that much benefit. There has been an effort from big companies such as Google to offer some source of constrained powerful hardware for everyone to use. One such product is called Google Colab, that offers for 12 hours free access to a Linux server that already runs the software Python. It comes with a lot of software libraries already installed. The hardware consists of a virtual container that runs on a 2 core CPU, with 12 GB of RAM and a virtual Nvidia T4 GPU which consists of 16GB of RAM. It also contains disk space of around 55GB. This can be a good platform to experiment and try different things, for free. The Figure 3.4 shows an implementation I have used.

Another great resource offered for free online is the platform called Kaggle. Kaggle is a platform that offers competitions online for the general public about data science projects in general. Competitions span from segmenting cars in images, to predicting stock prices. In order to host competitions that are somehow fair among

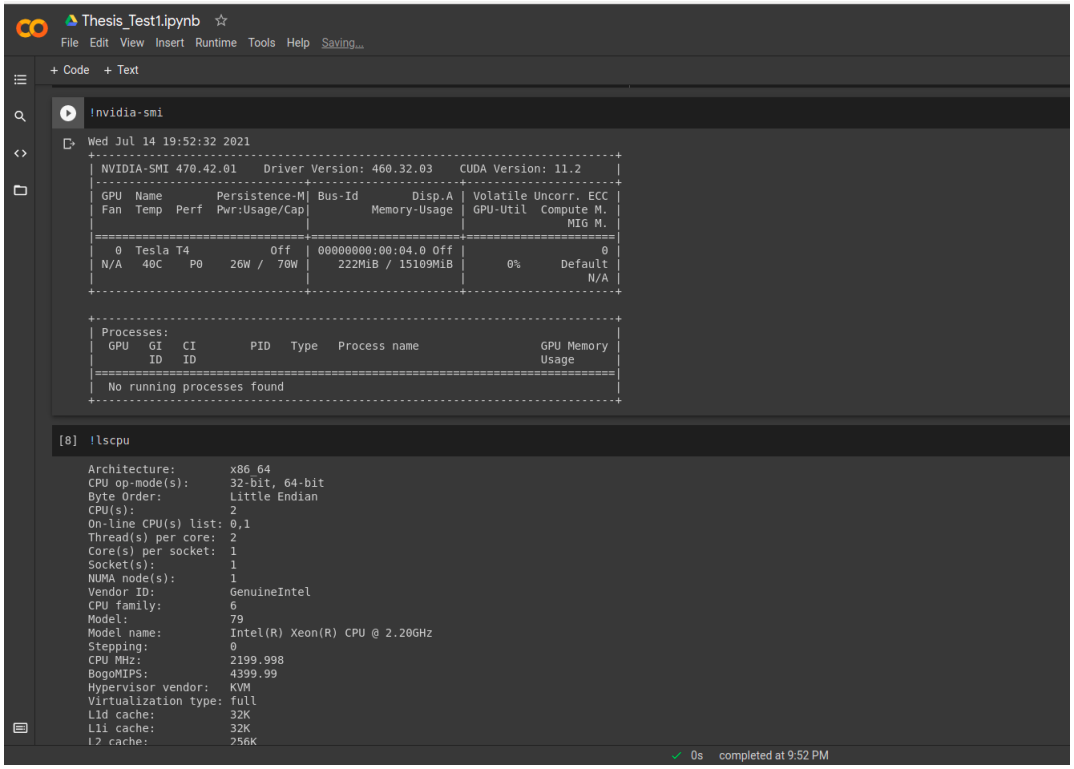


Figure 3.4: A picture of the free online Linux container called Google Colab, offering constrained access to GPU hardware. Figure created by the author.

the competitors, the platform offers free limited time hardware resources, such as a 2 core CPU, 12GB of RAM and a Nvidia P100 with 16GB of RAM. The host operating system is Linux based with Python and relevant software pre-installed. The time constraint consists of 39 hours usage per week. One of the example environments I have used are shown with the Figure 3.5.

3.2.3 Cloud Providers with Limited Free Resources

As of today there are many cloud providers in the market offering a large range of products, for many needs that business would require, from databases, to networking, to web hosting.
Google Cloud offers a free tier for new clients to its cloud resources. In order to access powerful hardware units on their platform, you need to submit a credit card and register an account. You will still have access first to the free tier which consists of a total sum of 300 dollars, which it would be charged first. After this has run out, the payment method submitted on the platform would be charged. Therefore, an advice would be to exhaust first the free alternatives and optimize the code on those platforms, before using more powerful hardware on Google Cloud.
The Google Cloud product that is offering a selection of GPU units, is called Google Compute Engine. You can configure a server as you want, with the latest hardware available by manufacturers.

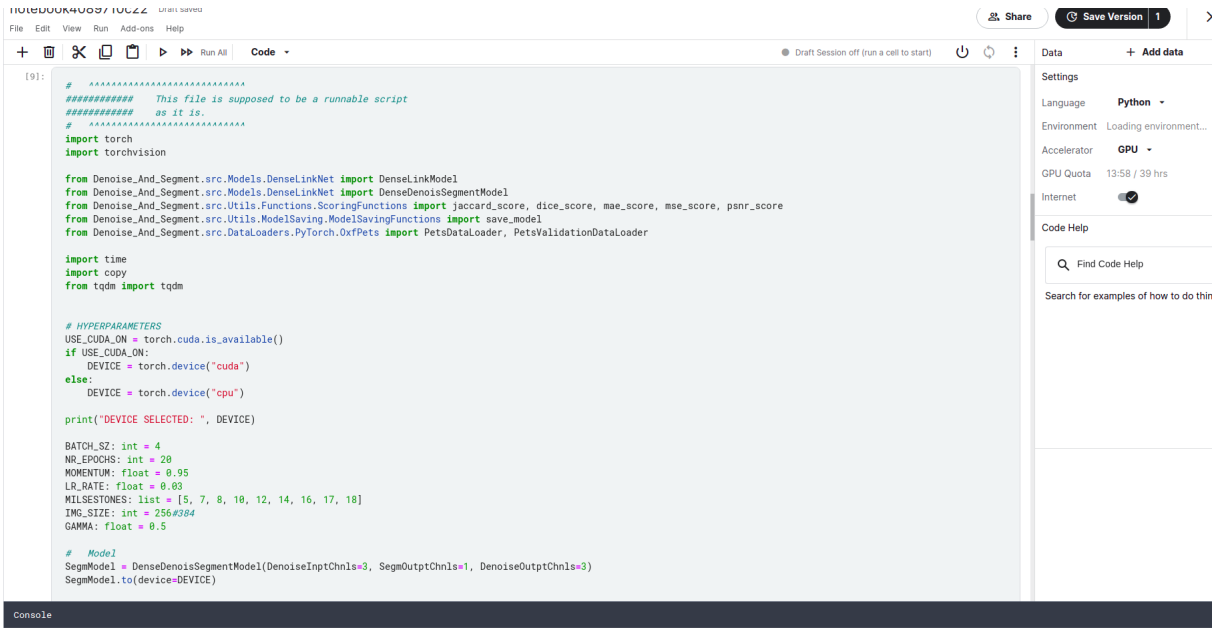


Figure 3.5: A picture of the free online Linux container offered by the online competition host website Kaggle, offering constrained access to GPU hardware. The figure is taken by the author.

3.2.4 Useful Linux command line tooling and GPU monitoring

Linux command line tools can help a lot in monitoring the optimization of an algorithm. One general useful tool, is a terminal multiplexer called tmux. Since working on a cloud provider Linux container, usually by opening a secure shell connection to the container, you only have access to one command line terminal. Terminal multiplexer allows for multiple terminal sessions to be open from the same window. In this way you can run more than one command line program at the same time, such as a tool to monitor the CPU and RAM, tool to monitor the GPU and a tool to run the program, usually a Python process. The Figure 3.6 demonstrates a tmux based session with 3 different terminal sessions running on the same window.

Apart from tmux, another command line tool that can be used is the htop command line, which is an interactive process viewer. It provides useful information on how many CPU cores are being used, how much RAM memory is available, and the process ids for the current processes.

Another useful Linux command line tool, that is available only on systems that include Nvidia hardware and the CUDA Software Development Kit, short SDK, is the command nvidia-smi. It provides information about the GPU, for example how much memory is loaded on the GPU from different processes, the temperature of the GPU and how much power is currently using. This could be useful in order to check how much GPU RAM memory an algorithm is currently using, in order to estimate if more is required.

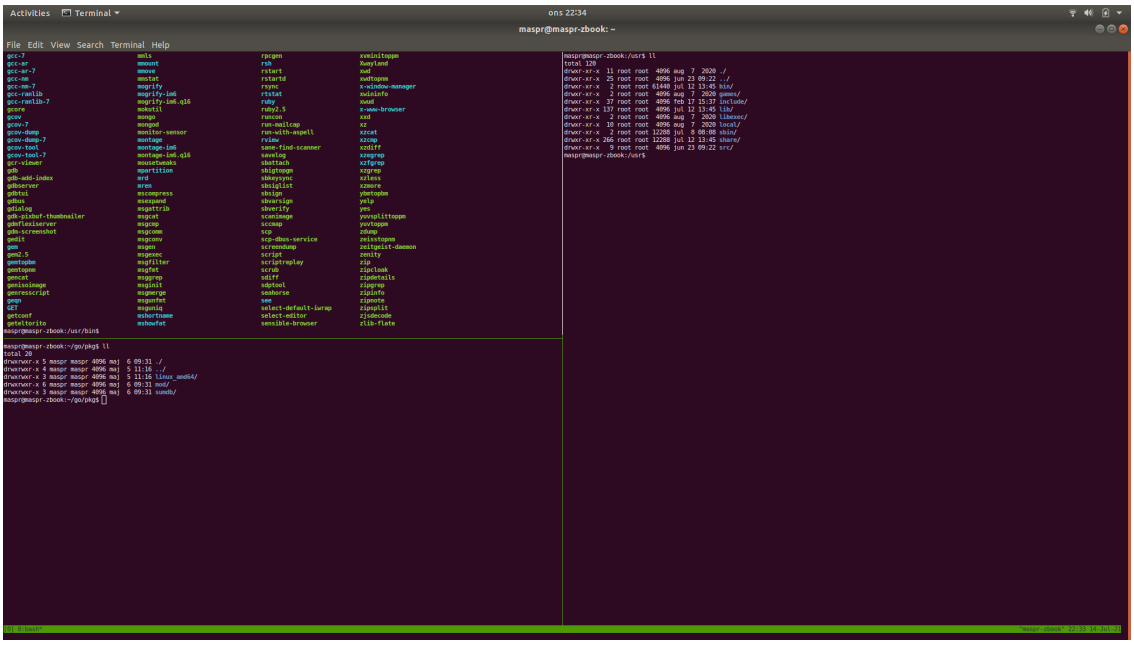


Figure 3.6: A picture of the tmux based command line tool that can run multiple terminal sessions at the same time. The figure is created by the author.

3.3 Software libraries and tools

In the following section, I described the software that I have used in order to build the algorithm and optimize it. The programming language that I have selected throughout the project is Python, version 3.8.

3.3.1 CUDA

Compute Unified Device Architecture, in short CUDA Software consists of a hardware driver, an application programming interface, short API, and mathematical libraries for running on Nvidia hardware. In general, it is free and will only work for Nvidia specific hardware. The CUDA API is an extension of the C programming language, that can be used to specify GPU operations like transfer data between the CPU and GPU or for thread level parallelism on the GPU. CUDA also consists of some profiling tools and compiler, and an SDK with examples.
CUDA can be used to create and optimize neural networks for Nvidia hardware, and since the Python programming language can be used as a wrapper on top of the C programming language, Python can be used to call compiled CUDA instructions. This is something that has great benefits for a wide audience, since with Open Source implementations of such practices, people that do not specialize in algorithms on specific hardware accelerators can use optimized based code for locality. This can efficiently exploit the CPU and GPU data transfer and thread level parallelism.
I have not used the CUDA API explicitly during the development of this report, due to its complexity, but in general I have investigated using available examples of how it works and what can be done to access the hardware. Modern tools to efficiently use this such as TensorFlow and PyTorch exist, that provide ease of access.

3.3.2 TensorFlow

TensorFlow is a free open source software library for machine learning that is being maintained by the Google Research team. TensorFLow was developed as a Python library, that provides an easy way to use heterogeneous hardware such as GPU and to use efficiently vectorized operations. Lately, it has been expanded in order to follow more software engineering practices, to be able to be ported to other platforms such as mobile, or microcontrollers.
Optimizing neural networks can be very tedious, especially when there is a need to write high performance code. TensorFlow provides an automated method to do this, that is called auto-differentiation. This is particularly useful, as you can avoid this process and write only the forward propagation of the algorithm. For this I used TensorFlow to make quick design and tests.
Apart from that, TensorFlow also offers useful commands to download datasets, that can be easily extracted. The Oxford Pats dataset is included, then it can be easily requested on any freely available online resource. In the final method, I used TensorFlow for this purpose.

3.3.3 PyTorch

PyTorch is another Python machine learning library that is open source and is maintained by the Facebook Research team. PyTorch provides similar functionality as TensorFlow with auto-differentiation, or using the library on other platforms such as mobile. The Python library has greater flexibility for creating neural networks, and to write optimization loops that can provide a more controllable environment.

3.3.4 PyTorch DataLoaders

PyTorch DataLoaders, as the name states, are a specific type of class to use in order to create instructions on getting the data that are saved on disk in a loop in order to be used by the model to be optimized. This is particularly useful, as it allows for transformation of those data at the same time that they are fetched from the hard drive. Transformations I have included in the data loading and loop process, are resizing the input image from each original size down to 256 by 256 pixels, and converting the unsigned integers 8-bit integers in the range 0-255, into floating point because they will be loaded on the GPU. At the same time, the image that is loaded alongside the annotation, another image is created with added Gaussian noise.

3.3.5 PyTorch Modularity and Vectorization

Vectorization is important when doing matrix algebra. PyTorch offers a simple way to use matrix algebra by creating tensors either on the CPU or the GPU. This can be beneficial to create metric functions measuring for example accuracy of the network Apart from vectorization, writing modular code is relatively easy. Through the SDK and the way the inheritance model works in Object Oriented Programming, the code can be easily split in different parts, which was particularly useful for constructing

the algorithm with separated operations, and splitting the encoder and decoder in different parts.

3.4 Modular Architecture of the Algorithm

3.4.1 Encoder

The encoder in this case, was a Densely Connected Convolutional Network that was pre-trained on the Imagenet dataset. This comes with the software and is ready to use. Therefore it simplifies a lot the work for the algorithm, and pre-trained weights for it can be downloaded. The final layer of the network, that performs matrix multiplication to output a 1000-dimension vector, is removed because it is obsolete. Only the convolutional feature maps are used.

3.4.2 Decoder

The decoder consists of custom made layers. Each layer, consists of two feature maps that include a convolution operation, non-linear map and a normalization layer. After the two layers, there is an operation for bilinear interpolation that up samples an input by doubling it. In total there are 7 layers like this. Each of these layers is concatenated with encoder layers, as shown and described before. The final layer of the network, consists of one convolutional layer that outputs either 3 channels or 1 channel depending on what you specify. This is because this network is used two times to output first the denoised image, and next to output the segmentation annotation.

3.4.3 Stacking Denoising and Segmentation Algorithm

The encoder and the decoder can be re-used since there are flexible. Whatever is the input dimension in the network, it outputs the same dimension. Therefore, the same encoder-decoder model can be stacked together with another encoder-decoder model and can produce the final algorithm. This is relatively simple, but effective.

3.5 Measuring Results During Optimization Loop

In the following, I specify different metrics to quantify the quality of the optimization of the algorithm on the data. It should be noted that calculation of this metrics are being done on the GPU, in order to eliminate the latency between exchanging data to the CPU.

3.5.1 Mean Absolute Error

Using PyTorch vectorization is easy to write functions to measure for example the mean absolute error. Mean absolute error is calculated as,

$$MAE = \sum_{i=1}^{N} \frac{|y_i - \hat{y}_i|}{N}. \tag{3.1}$$

It is the absolute difference between the predicted and the actual, and this is used during the training loop to measure the difference in the segmentation results. Lower is better, and usually this is also called as the L_{1norm}.

3.5.2 Mean Squared Error

Mean squared error is another useful measure to calculated the difference between the predicted and the true label. It is calculated as,

$$MSE = \sum_{i=1}^{N} \frac{(y_i - \hat{y}_i)^2}{N}. \tag{3.2}$$

This is also called the squared L_{2norm} and is used to calculate the difference between the segmentation label and the predicted segmentation. Lower is better. Note that this calculation is more computationally expensive than mean absolute error, due to the power 2 calculation that needs to be performed on a tensor. I used this as an extra measure to complement the mean absolute error to obtain a more detailed quantification of the network.

3.5.3 Peak Signal to Noise Ration

Peak Signal to Noise Ratio, in short PSNR, is a measure that is used to quantify the reconstruction of an image. It is calculated as the ratio between the MSE and the maximum possible value a signal can take, which is 1 since the maximum pixel value in an image if it is converted to float from 255 unsigned int, is 1. PSNR is calculated in the logarithmic scale of base 10. It is calculate as,

$$PSNR = \log_{10}(\frac{1}{\sum_{i=1}^{N} \frac{(y_i - \hat{y}_i)^2}{N}}) = \log_{10}(\frac{1}{MSE}). \tag{3.3}$$

The metric can be used to measure the true label of the denoised image, against the predicted denoise image. In the absence of noise, two images are identical therefore MSE would be zero and PSNR infinite. Therefore higher PSNR means better reconstruction. Usually a value between 30 and 50 is considered good for images.

3.6 Source Code Availability

The source code is currently available on the web host Github with free access, in the repository Denoise_And_Segment. There is no restrictive license in using it. In the Figure 3.7 the source code directory is viewed in a tree structure.

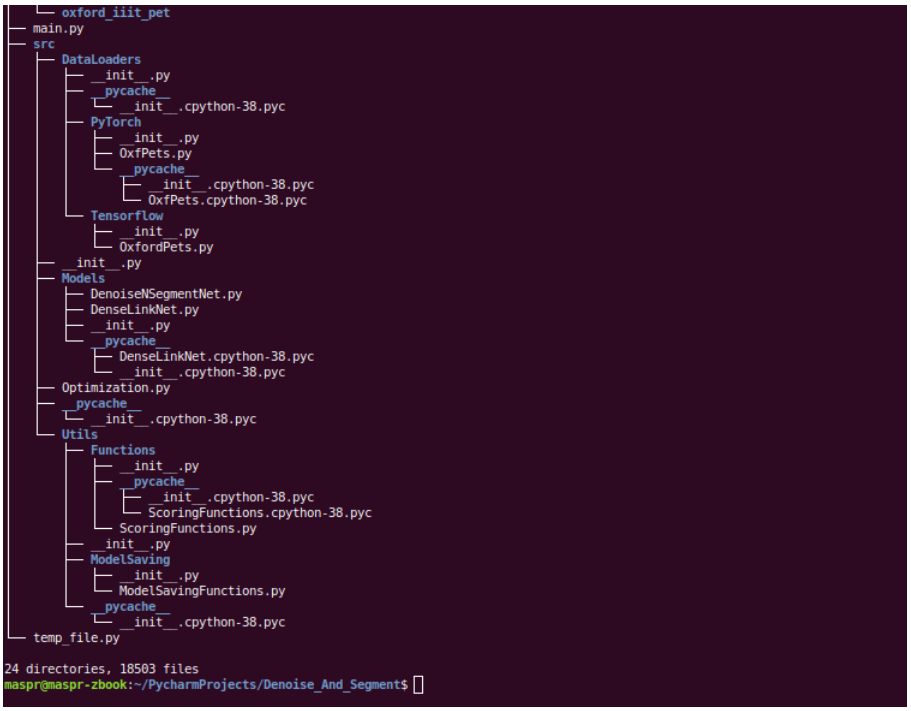


Figure 3.7: The figure represents the tree structure of the source code of this project. The figure is created by the author.

4

Results

After optimizing the network, the metrics as described before are collected and saved. Relevant plots are produced from the training and validation loss and the network is trained for 30 epochs with a batch size of 8 images per batch. These were the most ideal parameters for the available hardware, but it required a lot of time to be trained, approximately 500 minutes.

4.1 Training Error vs Validation Error

Training error is produced by optimizing the network on 70% of the dataset, and the rest was used as a validation evaluation of the network which accounted for 20% and 10% of the dataset was used for testing the overall performance. The Figure 4.1 shows the evolution of the training error over the validation error for 30 epochs. From the figure, we can conclude that the overall optimization is working as the validation error is dropping, but at the same time because of its volatile behavior, the validation error does not constantly decrease at a stable rate, therefore more epochs for training are required. Because the epoch number of training is relatively small, I have not applied an early stopping condition during training. Taking more time to train the network is a bottleneck for this study due to hardware limitations. Improving this condition is costly, and is beyond the scope of this study.

4.2 Quantitative Measures

As described in the methods section of the report, quantitative measures of how the network is progressing are being recorded during the optimization loop.

4.2.1 Mean Absolute Error

The absolute difference between the prediction of the segmentation and the true label, shows some interesting results. The training error seems to slowly decrease, and continue the same trend, while the validation metric is rather erratic but generally decreasing. This is on par with the optimization error on the loss function, that indicates that the network needs to be trained for more epochs. There is no clear indication that the network is over-fitting.

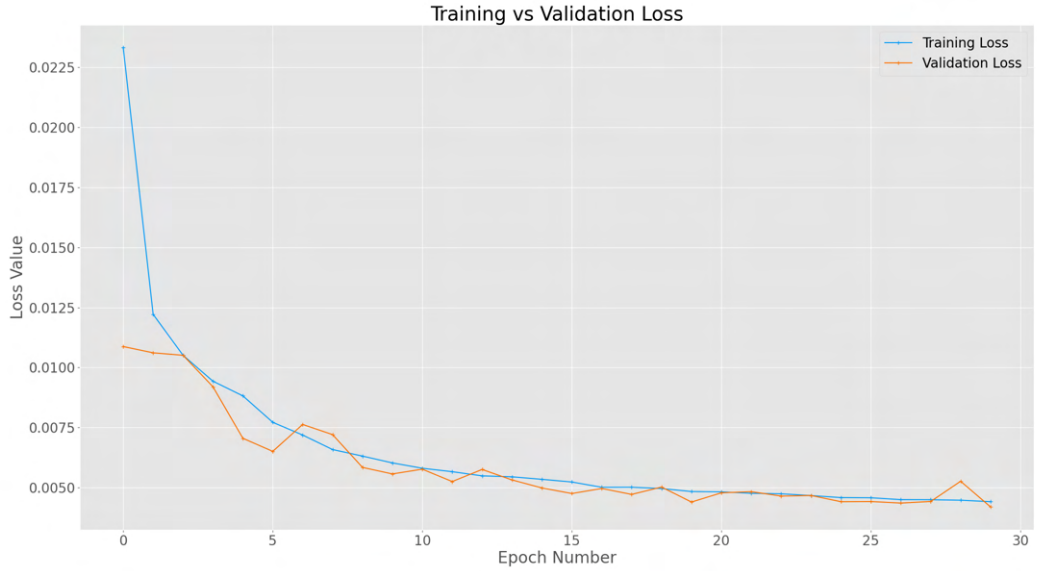


Figure 4.1: The figure represents the error loss during the optimization loop, for the dataset that is split between validation and training.

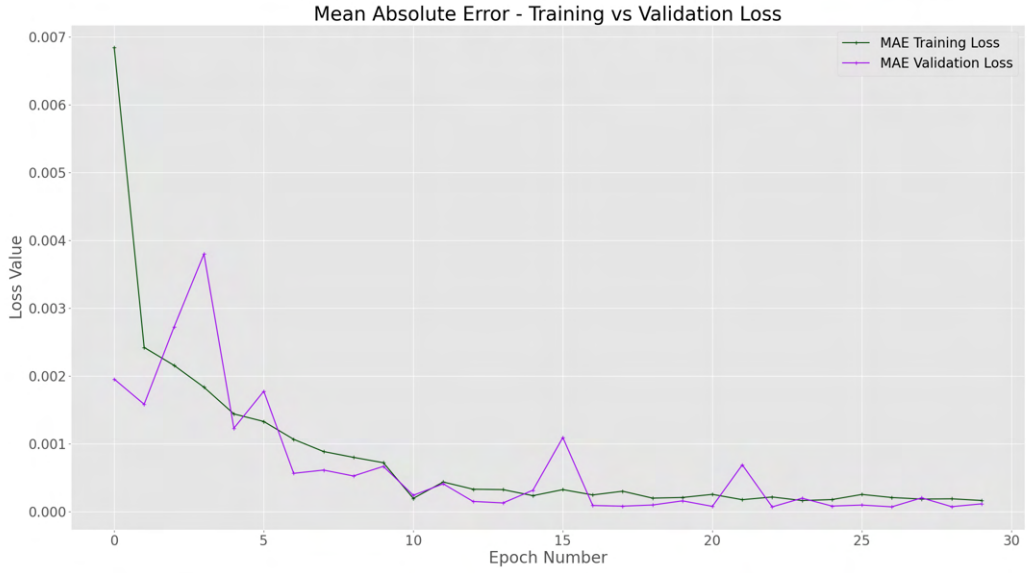


Figure 4.2: The figure represents metric Mean Absolute Error, as an absolute difference between the segmentation prediction and the true label.

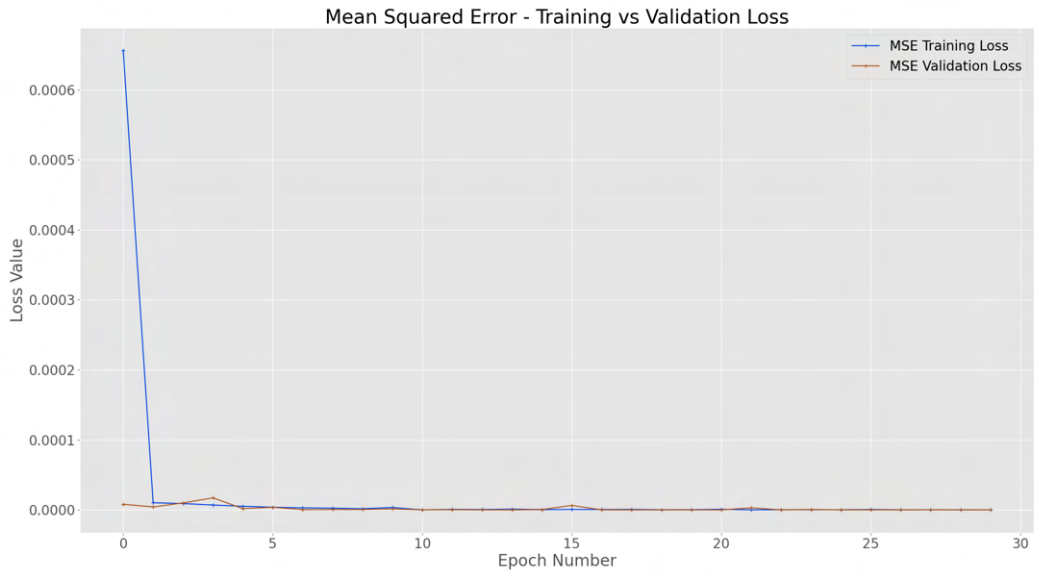


Figure 4.3: The figure demonstrates the Mean Squared Error during training and validation of the algorithm. As mentioned abover, the metric does not seem to give any clear indication of issues the algorithm may have, indicating that other metrics are more suitable for this task.

4.2.2 Mean Squared Error

The mean squared error quickly converges to infinitesimal results as seen in Figure 4.3. This is simply an additional metric to the mean absolute error to measure the performance of the network. However, it does not provide relevant input regarding the learning of the network nor over-fitting. Therefore, we can conclude that mean absolute error is a more useful metric to evaluate the algorithm.

4.2.3 PSNR

The PSNR metric in Figure 4.4 demonstrates that the network is learning to denoise the images, and the reconstruction quality during the training task seems to steadily rise. Validation seems erratic, which could be problematic. Nevertheless, the final task of using the algorithm is for segmentation, therefore the output quality of images from denoising, is secondary.

4.3 Visual Qualitative Results

Below I add some successful results where the network was able to predict the segmentation results, in comparison with the actual label. Additionally, I added some unsuccessful cases to illustrate some issues of the algorithm that remain unsolved. The images are taken from the validation dataset.

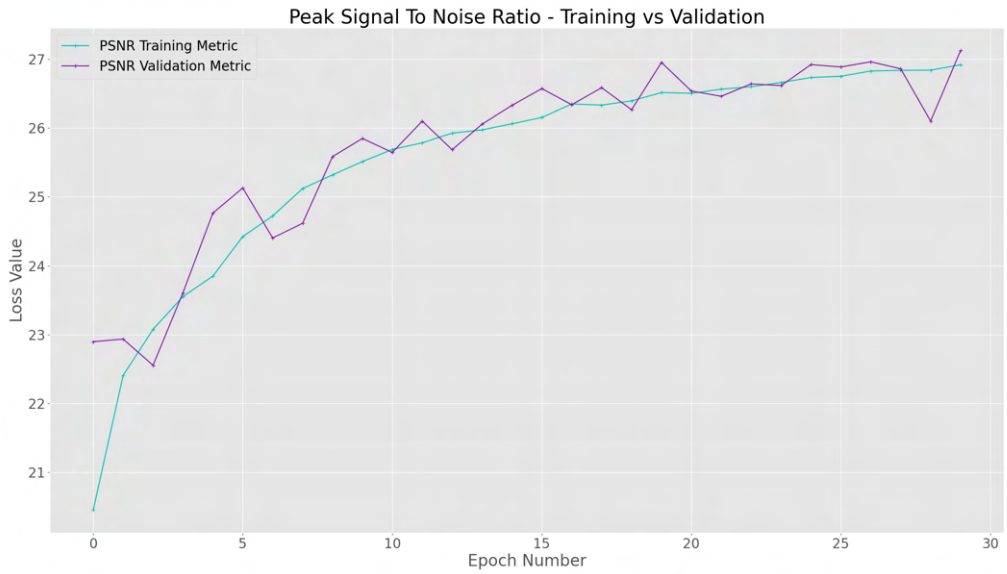


Figure 4.4: The figure shows how the PSNR between the actual image and the denoised prediction evolves with each epoch. There is a clear indication that it is improving. A good PSNR value for an algorithm is considered between 30 and 50 points.

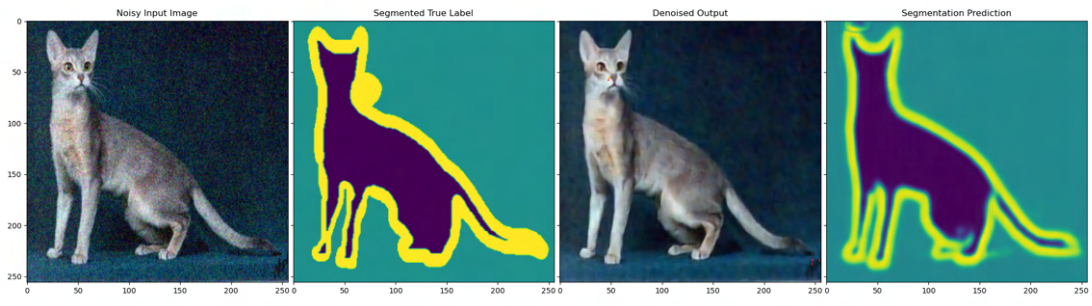


Figure 4.5: In the figure the network predicts a denoised image and a prediction segmentation map, marked on the further right of the image grid. The segmentation seems to work well and there is a prediction of the outline of the image which is belongs to a different class. At the same time, the network denoises the input image, but looks a bit blurry.

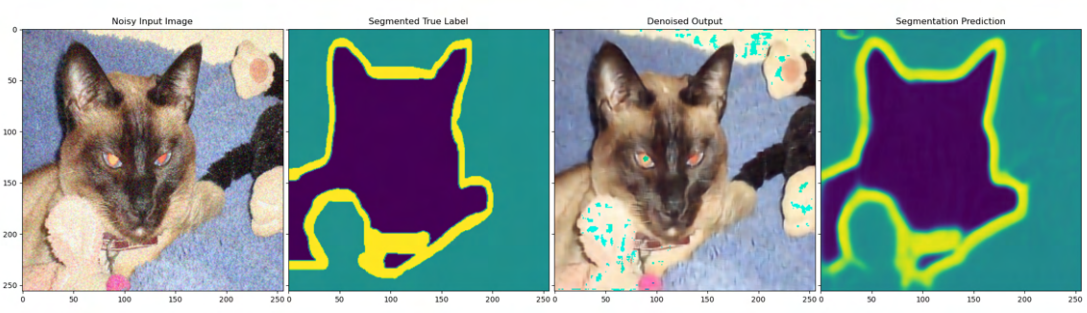


Figure 4.6: The figure shows another good result for the network, where it removes the noise in the image but leaves some artifacts and predicts the segment correctly.

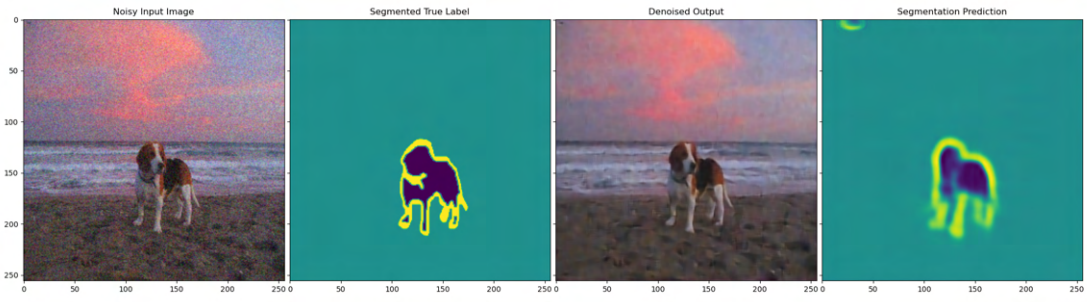


Figure 4.7: The figure shows a slight incorrect segmentation result from the dog class. There is a soft artifact on the top and at the same time the outline seems a bit blurred, which might be hard to miss by binarizing the image.

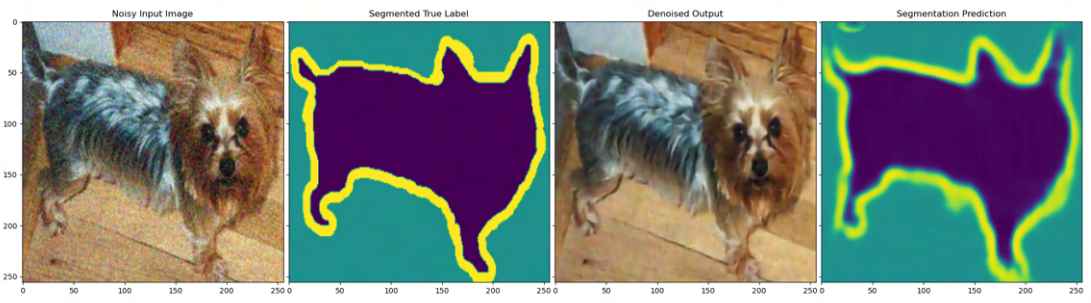


Figure 4.8: The figure shows an encouraging result proving that the algorithm works well. Here, it is demonstrated that the algorithm is able to find the correct outline of the dog, from the separated legs, a correct prediction of the head outline, and also somehow the tail. The denoised prediction has no artifacts as well, demonstrating good results.

Figure 4.9: The figure shows another example that has some slight artifacts on the segmentation prediction. However, it seems to be minor. They can be filtered out by thresholding, but this operation was not performed due to complexity. Nonetheless, there is an accurate prediction of the outline of the dog, and no artifacts in denoising.

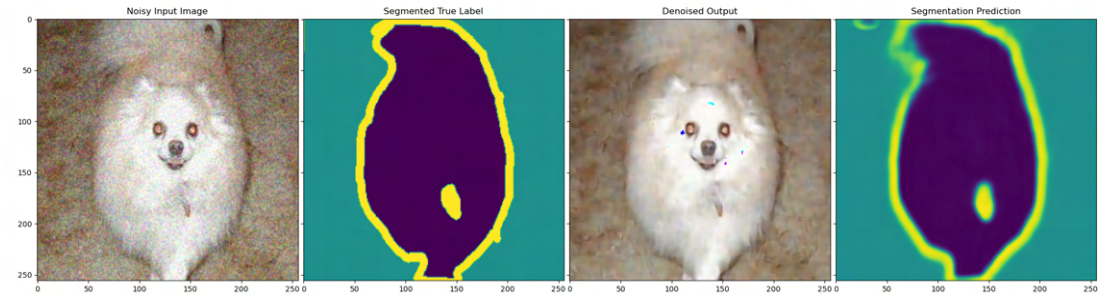


Figure 4.10: The figure shows a successful instance in denoising with very minor artifacts and at the same time segmenting. Here the segmentation is very accurate on the dog. This is rather challenging because of the tail, but the algorithm is able to perform well.

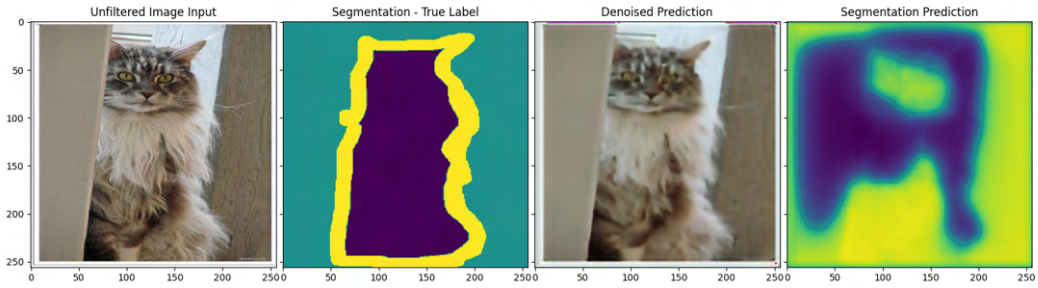


Figure 4.11: The figure shows a complete failure of the algorithm. Here the image has a rather challenging color scheme, and the brightness does not seem to vary, which might be a problem for the segmentation part. It is worth noting that the input image was an unfiltered image.

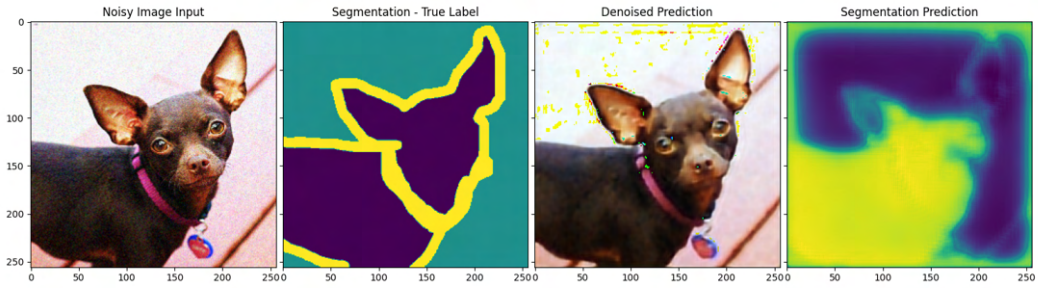


Figure 4.12: The figure shows another complete failure of the algorithm for the dog class.

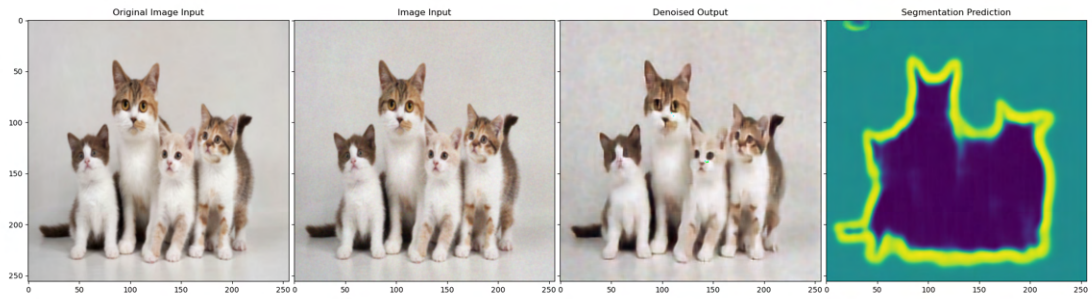


Figure 4.13: In the figure above, there is multiple instances of a cat that are correctly predicted, but as one segment all together. This is expected, as this method works for semantically segmenting one object.

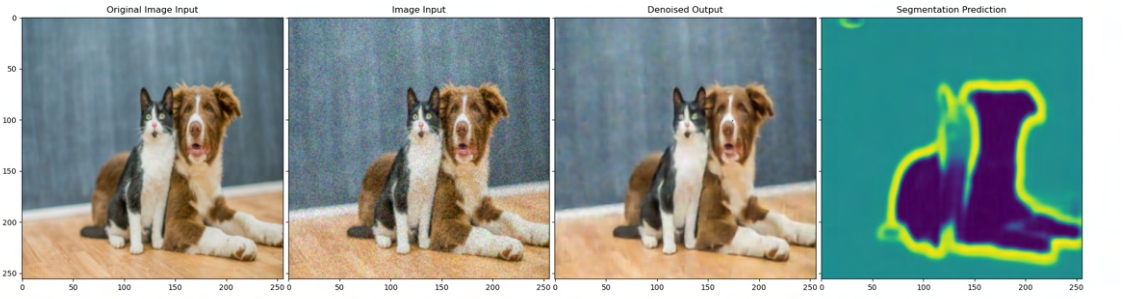


Figure 4.14: In the figure the algorithm fails in some way to predict the head of the cat, but correctly finds the legs.

4.4 Prediction on random images taken from the internet

Below I provide some results using this algorithm on random images collected from the internet. This would be useful to demonstrate how the algorithm is able to perform on unseen images with a different background, and more than one class of dog for example in the image. In some results the algorithm seem to work well for segmenting the exact object with its boundary.

4.5 Comparison with a simple Encoder-Decoder Network and Overall Segmentation Metrics

Judging from the visual qualitative results on the segmentation tasks, there is a clear problem that is observed. When the network tries to segment a picture, it produces artifacts. These artifacts seem to be on regions that there is clear distinctive noise. The network successfully removes the noise, but leaves artifacts on the segmentation results. This could have been addressed by penalizing the loss function for denois-

Figure 4.15: In this figure again the object from the cat class is missed completely, and only the dog gets segmented.

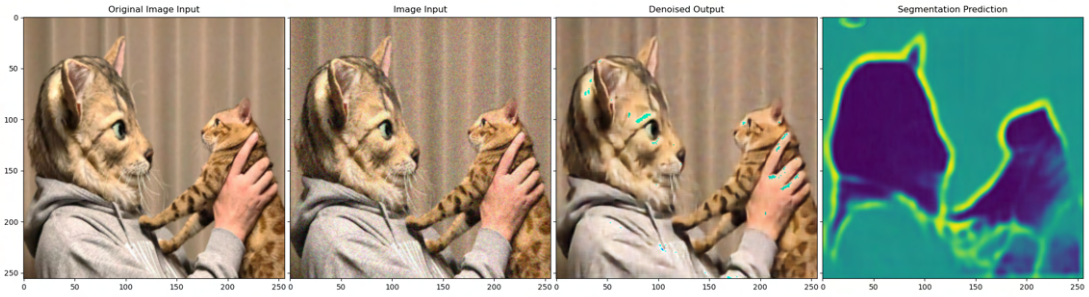


Figure 4.16: Here the network predicts sort of a fake cat in the image, which is a person disguised as a cat. This is rather conflicting, but a hard example for an algorithm to understand the meaning behind this image.

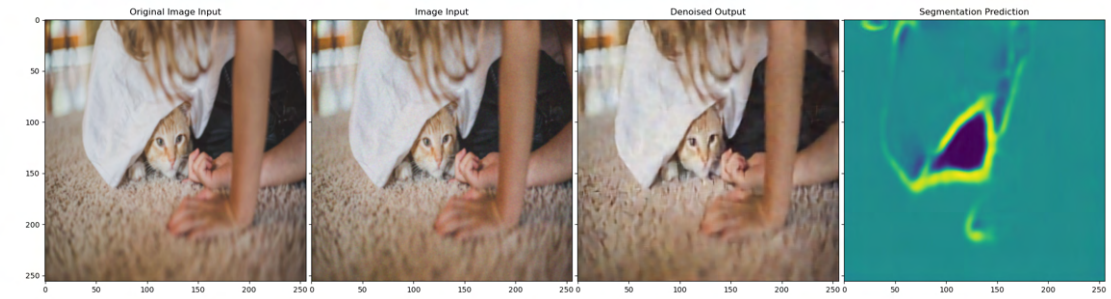


Figure 4.17: In this figure the cat is correctly segmented, even if it is just the face of it and rather a small object in the overall image.

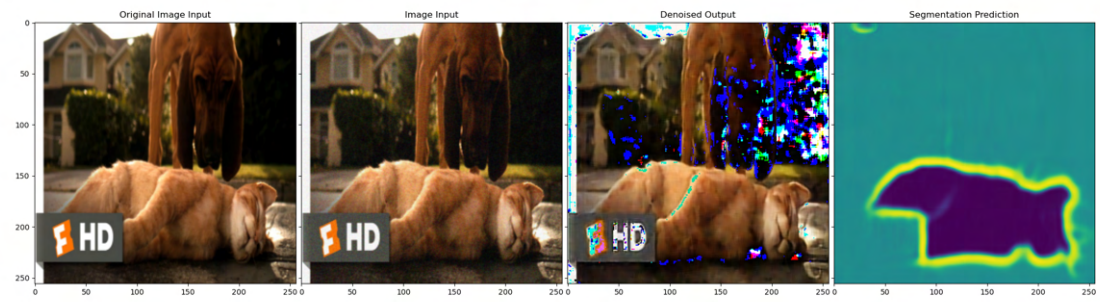


Figure 4.18: The dog in this figure is missed completely, even from denoising the image. The cat is correctly segmented, and also the algorithm avoids the text in the image.

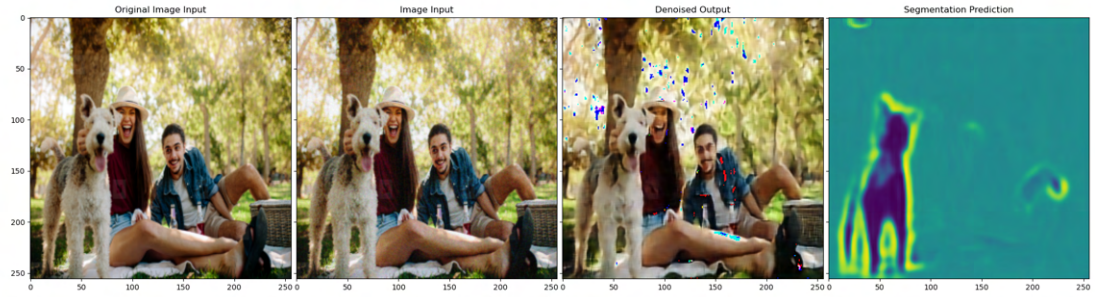


Figure 4.19: In this figure, despite the background with many unknown objects the algorithm is able to find the outline of the dog in the image.

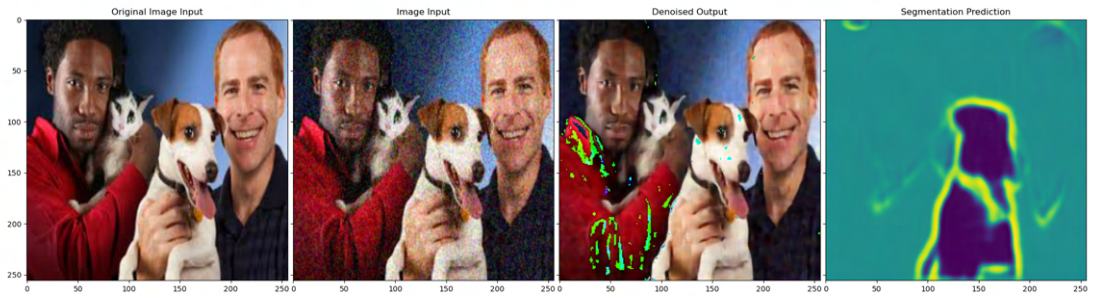


Figure 4.20: In this figure the cat is missed completely, but there is an exact outline of the dog found and segmented.

Figure 4.21: In this figure the dog outline is correctly predicted, but at the same time the algorithm correctly predicts the outline of the person, which is problematic.

ing. In the optimization loop a weight could be set, for example a value of 0.2 and 0.8 weight on denoising loss and segmentation loss respectively. This would have produced better results. the weight could have been set as a hyper-parameter, and optimize it further on in the process.

4.5.1 Training and Validation Error metric comparison

Apart from this, it would be useful to compare the original double network architecture of de-noising and segmenting, with a simple network that just performs segmentation. The simplified version is just one encoder and one decoder, similar to the original that is trained on the dataset. Th Figure 4.22 demonstrates the training and validation error during the optimization loop. The error rate for the simplified network decreases faster, but this does not prove better results. First the denoise and segment network, the loss function takes into account the denoise error value during the optimization loop, therefore it will be expected to be higher.
A better figure for the comparison of the two networks is demonstrated on Figure 4.23. Here it is clearly observed that the denoise-segment network performs better since the error metric drops early on at lower values with less fluctuation in the validation curve, which indicates that this could be a better architecture.

4.5.2 Jaccard Coefficient, Dice Score, and Accuracy on the Test Set

Jaccard index metric, Dice Score and Accuracy are commonly used metrics for evaluating segmentation results and classification. The reason is, that there are statistical measures to check the similarity of two samples.
The Accuracy measure is calculated by comparing for example two sets $\hat{y}$ and y, and defined as:

$$Accuracy = \frac{1}{N}\sum_{i=1}^{N} 1(y_i = \hat{y}_i) \tag{4.1}$$

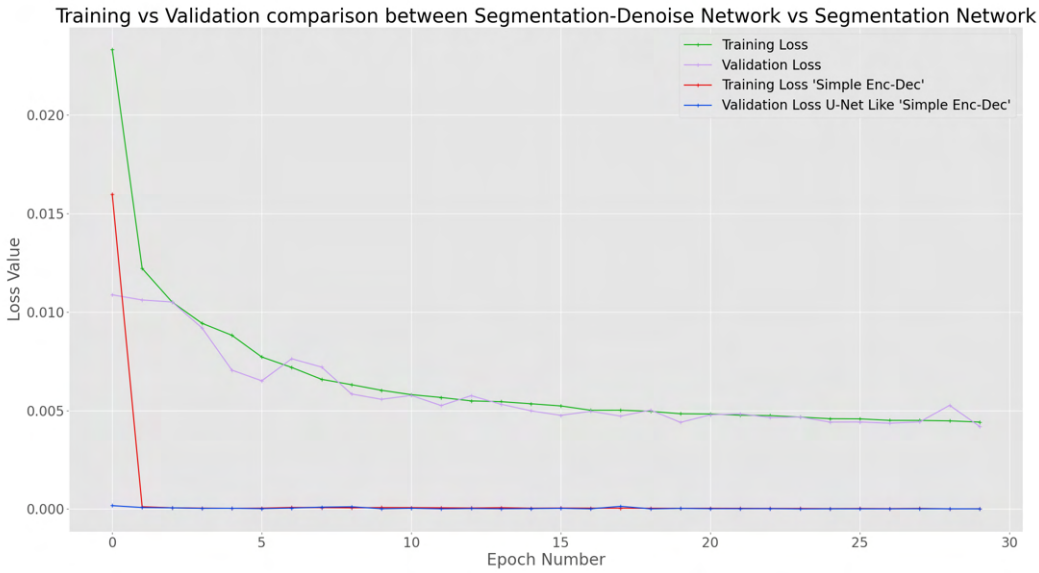


Figure 4.22: The figure demonstrates the training and validation error for the two different networks.

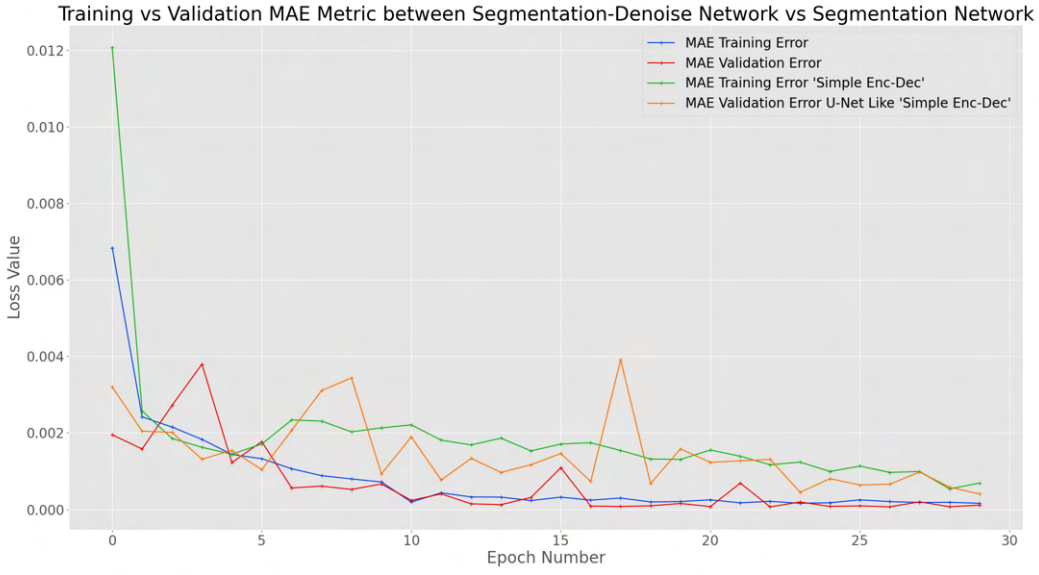


Figure 4.23: The figure demonstrates the MAE metric for the training and validation phase for the two different networks during the optimization loop.

and respectively Jaccard index is calculated as:

$$Jaccard = \frac{|X \cap Y|}{|X| + |Y| - |X \cap Y|} \tag{4.2}$$

From the Jaccard coefficient the Dice score can be derived:

$$DiceScore = \frac{2 * Jaccard}{1 + Jaccard} \tag{4.3}$$

The above measures are calculated as averages on the 10% test data that were left out during the optimization loop, which amount to roughly around 720 images, which are randomly selected from the dataset in the first split. The results are summarized on the Table 4.1 Judging from the results, it seems that both networks perform well

	Denoise-Segment Network	Encoder-Decoder Network
Accuracy	0.9152	0.8875
Jaccard Index	0.9095	0.8834
Dice Score	0.9526	0.9380

Table 4.1: Table Comparing the Accuracy Coefficient, the Jaccard Index and the Dice Score respectively for the two different Networks, to observe quantitatively which one performs better on the test dataset.

on the task for segmenting, but the joint task network performed slightly better.
Overall performance seems to be good for both networks, for the two accuracy metrics. While if the results would have been more closer to 1.0, the established denoise-segment could have competitive results. This denotes that there might be a better choice for parameters for the network, that could have yielded better performance.

4.6 Results Discussion

Commenting on the final results produced, the joint task algorithm did perform well on the dataset, but it also outperformed the simplified algorithm.
One of the approaches that could have improved the algorithm, was data augmentation which could have increased the dataset up to five or more times. This was not very feasible though, as computational resources were not available to handle the large memory requirement. Another approach to show improved results, is as previously discussed, to penalize the denoise task with lower weight, since it has proven to cause artifacts later on. One final approach that could have improved the overall metrics, is handling the class imbalance in a picture with weighting regions. An image for example would have much less amount of pixels that belong to the boundary of the object, than the background or the object itself. This demonstrates class imbalance, that can affect the algorithm further on.
Apart from these methods that could help, during the optimization loop I could have added the calculation of the Accuracy Coefficient and the Jaccard Index that could spot early on how the accuracy increases. I have chose to omit this, since it

was computationally expensive during the optimization process, and could break the optimization loop at any time, which could have been very problematic for producing results and slow down the overall effort. However this could have been addressed better, as it is of high importance.
As mentioned before, the overall accuracy of the method according to the metrics, seems good enough as a baseline for the experiment, and that the established algorithm works in this case. This shows that the proposed method, with a multi task prediction approach can be comparable with a simplified one. What can be further explored, is to use the network in a classification setting rather than regressing to pixel values, and compare the two outputs.

5

Conclusion

Convolutional Networks have been successful in a large variety of tasks, in computer vision, natural language processing and speech recognition and other fields. They have shown impressive results compared to previous methods, and are able to compress and be optimized on specific datasets. As with the increase in computer power, there is a trend towards increasingly more complex and bigger algorithms that exceed previous scores. But, currently there is no single theory that explains, proves or concludes why they are able to achieve such great results, therefore a large majority of algorithms are mostly experimental.

In this study, I have attempted to segment an image based on a joint task in an algorithm, to denoise first an image and at a later stage to segment. Multi-task prediction has been explored in other approaches, such as instance segmentation where classification of a class with joint bounding box prediction for an object and later on segmentation, showed better results compared to other methods. This approach is described in the paper Mask R-CNN.

In addition, lately there has been an attempt for self supervised learning, which in simple words is a convolutional network that tries to reconstruct in specific ways the input image. One example is to split the image in a jigsaw grid puzzle and use a convolutional network to predict the correct assembly. Then the feature maps are used for transfer learning, for example. As an inspiration from this example, I used a method to denoise with artificial pseudo randomness an image and as a later stage to segment the image. Therefore, the network learns the context of the image first and then can find segments.

The network effectively learns to predict segments, yet failing in certain cases. Interestingly, most of the examples for which the network was not able to predict segments represented a cat. A reason for this might be the many edges in pictures, surrounded by noisy pixels, which cause the network to fail to predict correctly the boundary.

To improve the results, there are multiple ways one can think of. One way to improve the results is to raise the size of the image that the network will use. In this report, I used size 256 by 256 pixels for each image. With a size of 512 by 512 would give better results since the images are larger, and convolutions will run on more detailed areas. However, this approach has problems, since there will be more memory required by the GPU, which I was not able to afford. As for many machine learning problems, there is an existing trade-off between economic cost and computational cost that is available.

Further ways to increase the accuracy of the model and overall performance, would have been to augment the dataset. This would include rotating images, changing

brightness, flipping or cropping. In this way, the algorithm would perform better with variation in lighting conditions for example.
In addition to the augmentation of the dataset, if more detailed data regarding the geometry of the images was available, such as normal vector maps or depth heat maps, the network could perform better since there would be an availability for more geometrical input about the image. The network could be used to predict edges and penalized for this, in order to segment better. The final loss could be weighted more towards the segmentation output, in order to penalize early stages. Unfortunately, due to time constraints it was not possible to include such experiments as part of this thesis. However, it would be a good point from which to carry on future research on the topic, and investigate further approaches.

Bibliography

[1] Yann LeCun, Yoshua Bengio and Geoffrey Hinton (2015) Deep Learning.

[2] Ian Goodfellow, Yoshua Bengio and Aaron Courville (2015) Deep Learning.

[3] Firth, John R. (1957). "A synopsis of linguistic theory 1930-1955". Studies in Linguistic Analysis.

[4] M. Schuster, K.K. Paliwal (1997). Bidirectional recurrent neural networks.

[5] J. Schmidhuber and S. Hochreiter (1997). Long Short Term Memory.

[6] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio (2014). Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.

[7] Evan Shelhamer, Jonathan Long, Trevor Darrell (2016). Fully Convolutional Networks for Semantic Segmentation.

[8] Alexander Kirillov, Kaiming He, Ross Girshick, Carsten Rother, Piotr Dollár (2018). Panoptic Segmentation.

[9] Chakravarty R. Alla Chaitanya, Anton Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, Timo Aila (2017). Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder.

[10] Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick(2017). Mask R-CNN.

[11] Olaf Ronneberger, Philipp Fischer, Thomas Brox(2015). U-Net: Convolutional Networks for Biomedical Image Segmentation.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun(2015). Deep Residual Learning for Image Recognition.

[13] Gao Huang, Zhuang Liu, Laurens van der Maaten, Kilian Q. Weinberger(2016). Densely Connected Convolutional Networks.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun(2016). Identity Mappings in Deep Residual Networks.

[15] Yi Li, Haozhi Qi, Jifeng Dai, Xiangyang Ji, Yichen Wei(2016). Fully Convolutional Instance-aware Semantic Segmentation.

A

Appendix 1

A.1 Algorithm

The Joint Task Algorithm in one single Python script.

```
from torchvision import models
from torch import nn
import torch


class ConvEluGrNorm(nn.Module):
    def __init__(self, inp_chnl, out_chnl):
        super(ConvEluGrNorm, self).__init__()
        self.conv = nn.Conv2d(in_channels=inp_chnl, out_channels=
    out_chnl, kernel_size=3, padding=1, bias=False)
        self.norm = nn.GroupNorm(num_groups=16, num_channels=
    out_chnl)
        self.celu = nn.CELU(inplace=True)

    def forward(self, x):
        out = self.conv(x)
        out = self.norm(out)
        out = self.celu(out)
        return out


class UpsampleLayer(nn.Sequential):
    def __init__(self, in_chnl, mid_chnl, out_chnl, transp=False):
        super(UpsampleLayer, self).__init__()
        if not transp:
            self.block = nn.Sequential(
                nn.Upsample(scale_factor=2, mode="nearest"),
                ConvEluGrNorm(in_chnl, mid_chnl),
                ConvEluGrNorm(mid_chnl, out_chnl)
            )
        else:
            self.block = nn.Sequential(
                ConvEluGrNorm(in_chnl, mid_chnl),
                nn.ConvTranspose2d(in_channels=mid_chnl,
    out_channels=out_chnl,
                                   kernel_size=4, stride=2, padding
    =1, bias=False),
                nn.CELU(inplace=True)
            )

```

```
37
38 class DenseSegmModel(nn.Module):
39     def __init__(self, input_channels, num_filters=32, num_classes
   =1, pretrained=False):
40         super(DenseSegmModel, self).__init__()
41         encoder = models.densenet121(pretrained=pretrained).
   features
42         self.layer1 = nn.Sequential(
43             nn.Conv2d(in_channels=input_channels, out_channels=64,
   kernel_size=7, stride=2, padding=3, bias=False),
44             nn.GroupNorm(num_groups=16, num_channels=64),
45             nn.CELU(inplace=True),
46             encoder[3]
47         )
48         self.layer2 = encoder[4:6]
49         self.layer3 = encoder[6:8]
50         self.layer4 = encoder[8:10]
51         self.layer5 = encoder[10]
52         self.gn = nn.GroupNorm(num_groups=16, num_channels=1024)
53         self.pool = nn.AvgPool2d(kernel_size=3, stride=2, padding
   =1)
54
55         self.center = UpsampleLayer(in_chnl=1024, mid_chnl=
   num_filters * 8, out_chnl=num_filters * 8)
56         self.dec5 = UpsampleLayer(1024 + num_filters * 8,
   num_filters * 8, num_filters * 8)
57         self.dec4 = UpsampleLayer(512 + num_filters * 8,
   num_filters * 8, num_filters * 8)
58         self.dec3 = UpsampleLayer(256 + num_filters * 8,
   num_filters * 8, num_filters * 8)
59         self.dec2 = UpsampleLayer(128 + num_filters * 8,
   num_filters * 2, num_filters * 2)
60         self.dec1 = UpsampleLayer(64 + num_filters * 2, num_filters
   , num_filters)
61         self.dec0 = UpsampleLayer(num_filters, num_filters,
   num_filters)
62         self.final = nn.Conv2d(num_filters, num_classes,
   kernel_size=1)
63
64     def forward(self, x):
65         conv1 = self.layer1(x)
66         conv2 = self.layer2(conv1)
67         conv3 = self.layer3(conv2)
68         conv4 = self.layer4(conv3)
69         conv5 = self.layer5(conv4)
70         out = self.pool(self.gn(conv5))
71
72         center = self.center(out)
73
74         dec5 = self.pool(self.dec5(torch.cat([center, conv5], 1)))
75         dec4 = self.dec4(torch.cat([dec5, conv4], 1))
76         dec3 = self.dec3(torch.cat([dec4, conv3], 1))
77         dec2 = self.dec2(torch.cat([dec3, conv2], 1))
78         dec1 = self.dec1(torch.cat([dec2, conv1], 1))
79         dec0 = self.dec0(dec1)
80         return self.final(dec0)
```

```
81
82
83 class DenoiseSegmModel(nn.Module):
84     def __init__(self, denoise_input_chnls: int=3, segm_outpt_chnls
       : int=1, denoise_outpt_chnls: int=3):
85         super(DenoiseSegmModel, self).__init__()
86
87         self._DenoiseNet = DenseSegmModel(input_channels=
       denoise_input_chnls, num_filters=32, num_classes=
       denoise_outpt_chnls, pretrained=True)
88         self._SegmNet = DenseSegmModel(input_channels=
       denoise_outpt_chnls, num_filters=32, num_classes=
       segm_outpt_chnls, pretrained=True)
89
90     def forward(self, x):
91         denoise_out = self._DenoiseNet(x)
92         segm_out = self._SegmNet(denoise_out)
93
94         return denoise_out, segm_out
95
96
97 def DenseDenoisSegmentModel(DenoiseInptChnls: int=3, SegmOutptChnls
       : int=1, DenoiseOutptChnls: int=3):
98     return DenoiseSegmModel( denoise_input_chnls=DenoiseInptChnls,
       segm_outpt_chnls=SegmOutptChnls, denoise_outpt_chnls=
       DenoiseOutptChnls )
99
100
101 def DenseLinkModel(input_channels: int, pretrained: bool=False,
       num_classes: int=1):
102     return DenseSegmModel(input_channels=input_channels, pretrained
       =pretrained, num_classes=num_classes)
```