Steering Angle Prediction by a Deep Neural Network and its Domain Adaption Ability

Anders Hansson

Master's thesis
2018:E26

LUND UNIVERSITY

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

CENTRUM SCIENTIARUM MATHEMATICARUM

Steering angle prediction by a deep neural network and its domain adaption ability

Anders Hansson

May 23, 2018

1 Abstract

The goal of this thesis is to design an artificial neural network for self-driving vehicles in regards to steering the vehicle. The performance of the networks is evaluated on a simulated race track in addition to more conventional metrics such as *cost/loss*. In the main part of the project the networks were trained on data from real-life driving and validated/tested on simulated data, which is an example of *domain adaption*. The simulated data were from the same conditions as in the simulated race track. Two main designs were evaluated, one based on the design proposed by *NVIDIA* and one based on the idea of *multi-task learning* where an *autoencoder* was trained simultaneously with a steering angle predictor.

The resulting network based on the multi-task learning and solely trained on real-life driving data managed to make it around the entire simulated track without driving off the road. The network based on the NVIDIA design on the other hand only managed to stay on the road for a short period of time under the same conditions. The results indicate that the multi-task based design is better at domain adaption than the one based on NVIDIA's design and incites for further research within the area.

Also some additional tests were conducted with real-life data for both training and validation. The results from this part of the thesis were ambiguous with respect to the domain adaption ability.

2 Acknowledgements

I want to express my deepest gratitude to my supervisor Alexandros Sopasakis for his guidance and advice. I also want to thank him for introducing me to the research area and for the opportunity to write my master's thesis on such an interesting subject. Furthermore I want to thank him for letting me do the project in my own pace and for all the quick feedback. Seldom have I met a person that answers emails so rapidly.

I also want to thank Carl-Gustav Werner for helping me with technical issues

I am thankful for the center for scientific and technical computing at Lund University (LUNARC) for letting me use their resources for my thesis.

Contents

1 Abstract **2**

2 Acknowledgements **3**

3 Introduction **6**
3.1 Background . . . 6
3.2 Objectives . . . 6
3.3 Approach and objectives . . . 6

4 Theory **7**
4.1 Artificial Neural Networks . . . 7
4.1.1 Artificial Neuron . . . 7
4.1.2 Feedforward Artificial Neural Networks . . . 8
4.1.3 Layers . . . 9
4.1.4 Deep artificial neural networks . . . 10
4.2 Activation functions . . . 10
4.2.1 Heaviside function . . . 10
4.2.2 Sigmoid function . . . 11
4.2.3 ReLU (Rectified linear unit function) . . . 12
4.2.4 ELU (exponential linear unit function) . . . 13
4.3 Universal Approximation Theorem . . . 14
4.4 Convolutional neural network . . . 15
4.4.1 Convolutional layer . . . 15
4.4.2 Pooling layer . . . 17
4.4.3 Upsampling layer . . . 18
4.4.4 Fully connected layer . . . 18
4.5 Autoencoder network . . . 19
4.6 Data . . . 19
4.6.1 Collection . . . 19
4.6.2 Partitioning of data . . . 20
4.6.3 Over-/undersampling . . . 21
4.6.4 Augmentation . . . 22
4.6.5 Data Preprocessing . . . 23
4.7 Network training . . . 24
4.7.1 Types of learning . . . 24
4.7.2 Loss functions . . . 26
4.7.3 Stochastic gradient descent . . . 26
4.7.4 The Backpropagation algorithm . . . 28
4.7.5 Initialization of network . . . 29
4.7.6 Training algorithm . . . 31
4.7.7 Vanishing gradient problem . . . 31
4.7.8 Overfitting . . . 32
4.7.9 Transfer Learning and domain adaption . . . 35
4.7.10 Multi-task learning . . . 36
4.8 Regularization . . . 37

4.8.1 Early stopping . . . 37
4.8.2 L2-regularization . . . 38
4.8.3 Dropout . . . 39
4.9 Batch Normalization . . . 40
4.10 Hyperparameters . . . 42
4.10.1 Learning rate . . . 43
4.10.2 Batch size . . . 43
4.10.3 Epochs . . . 43
4.10.4 Regularization parameter . . . 44
4.10.5 Dropout rate . . . 44
4.11 Performance measures . . . 44
4.11.1 Loss . . . 45
4.11.2 Accuracy . . . 45
4.11.3 Confusion matrix . . . 45

5 Implementation **46**
5.1 Data . . . 46
5.1.1 Artificial data . . . 46
5.1.2 Real-life data . . . 47
5.1.3 Manipulation of data . . . 47
5.2 Resources . . . 50
5.2.1 Software . . . 50
5.2.2 Hardware . . . 50
5.3 Tasks . . . 50
5.3.1 Task 1 . . . 55
5.3.2 Task 2 . . . 55
5.3.3 Task 3 . . . 59

6 Results **59**
6.1 Data . . . 59
6.2 Task 1 . . . 60
6.3 Task 2 . . . 61
6.3.1 Task 2.1 . . . 61
6.3.2 Task 2.2 . . . 69
6.4 Task 3 . . . 79

7 Discussion **84**
7.1 Data . . . 84
7.2 Task 1 . . . 86
7.3 Task 2 . . . 86
7.4 Task 3 . . . 88
7.5 General conclusions . . . 89
7.6 Further work . . . 89

3 Introduction

3.1 Background

An autonomous or self-driving vehicle is assumed to be able to partially or entirely drive itself. Partial self-driving or supervised driving involves technology which assists and partially or fully reacts on its own without control from humans. Some such technology exists already today and incorporates systems for accident avoidance such as automatic deployment of airbags, automatic breaking, automatic lane keeping, crash warning and avoidance technology, etc. On the other hand a vehicle which is able to entirely drive itself means that the vehicle has the capability to navigate without any human intervention. Such technology does not yet exist on the market but it is generally believed that it might be in the near future.

The possibility of a vehicle driving itself have inspired researchers for decades now. A number of research publications in autonomous driving have appeared from experts ranging from IT to automobile manufacturer to image processing, mathematics and several more. The potential benefits of self-driving vehicles are numerous and among the most mentioned ones are safer and more effective transport.

Among other challenges concerning self-driving vehicles, one is to make the vehicle drive itself in different conditions without having met them before. It is basically impossible to train a system for self-driving vehicles explicitly on all situations it might encounter in reality. Consequently, methods to make the ”knowledge” of a self-driving vehicle very general and applicable to a wide range of conditions are very attractive. Much like how humans can recognize a dog, even without having seen that particular breed before.

3.2 Objectives

The aim of this work is to explore some of the currently used and most effective modeling methods and software approaches responsible for driverless technology today. The research proposed within this thesis therefore aims to understand how current state-of-the-art methodologies within artificial intelligence are used in order to teach a vehicle to drive itself. Here, drive itself refers to the ability to steer a vehicle. In that respect the so called deep learning algorithms, which in combination with neural networks are the main building blocks behind such autonomous systems, will be studied. Two different such deep learning systems will be constructed and compared. Their performance will ultimately be evaluated on a simulated race track.

3.3 Approach and objectives

The aim will be to design a network that can drive itself around the aforementioned simulated race track. The approach will be to train the network by feeding it input images along with correct steering angles. The input images will be both simulated driving data and driving images from real life. In one setting the network will be trained on the simulated driving data and in another on the real-life data. Then design features of the network can be compared between the two settings. The focus of the thesis will be on the second setting where the concept of *domain adaption* is necessary.

4 Theory

4.1 Artificial Neural Networks

As the name reveals an Artificial Neural Network (ANN) is an artificial version of a neural network. The particular neural network that the research within ANNs ultimately is trying to mimic is the human brain. The human brain is incredibly versatile when it comes to solving different tasks, today unmatched by any computer. Thus, trying to mimic the processes going on in the human brain is very appealing and an artificial brain has huge potential when it comes to solving problems that science meet today. In this section the components and the architecture of an artificial neural network will be explained, along with its resemblance to a biological neural network.

4.1.1 Artificial Neuron

As mentioned above, an artificial neural network tries to mimic a biological neural network. Likewise, in the same way as a biological neural network consists of neurons an ANN consists of artificial neurons. A neuron cell in the human brain coarsely consists of a *nucleus*, *dendrites* and an *axon* [11]. An illustration of a neuron can be seen in Figure 1

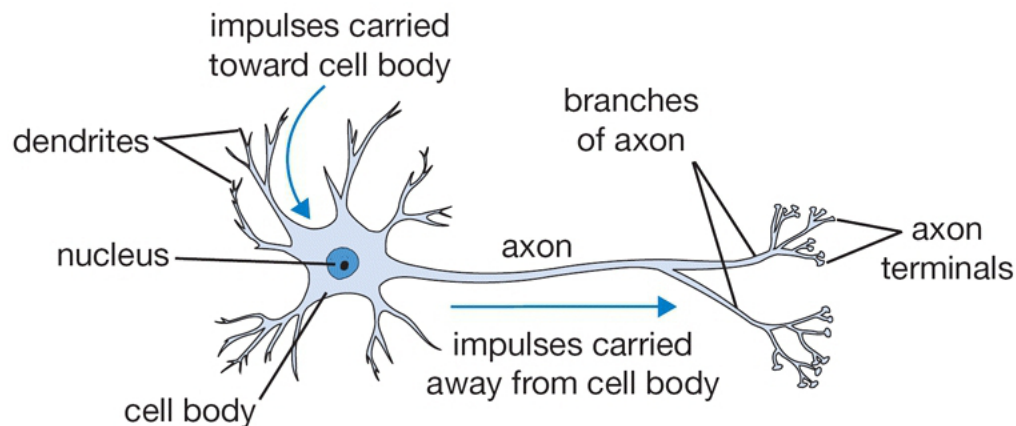

Figure 1: Illustration of a biological neuron in the human brain where important parts are labeled.[1]

The most prominent features of a biological neuron that an artificial neuron tries to mimic are the following. A neuron gets input signals from its *dendrites*, as can be seen in Figure 1 it can get several input signals. In the *nucleus* some kind of processing of the input signals occurs (see section 4.2) that decides what the output should be. The biological neuron has an "all-or-none" property which means that the the output can only have two values, more on this in section 4.2. The output is then transferred to other neurons via the *axon* [11]. The mathematical model of the biological neuron known as an artificial neuron can be seen in Figure 2

[1] http://cs231n.github.io/neural-networks-1/

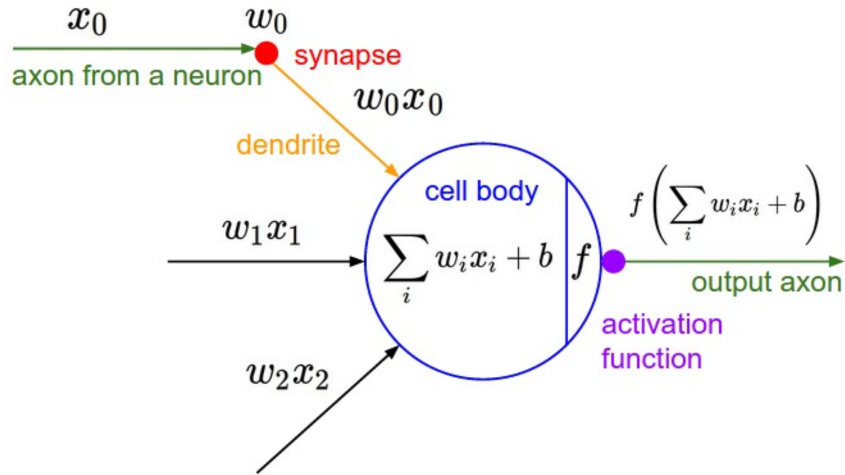

Figure 2: Illustration of an artificial neuron.[2]

Mathematically the artificial neuron can be expressed as follows. The artificial neuron takes as input $\mathbf{x} = (x_1, x_2, ..., x_m)$ that is a vector of all the inputs x_i and m is the number of inputs. It weights each input x_i with a designated weight ω_i and sums up all the weighted inputs. A bias b is added to the sum and the result is fed to an activation function σ (or f in Figure 2). The activation function introduces non-linearity in to the model and is explained further in section 4.2. The result from the activation function is the output of the neuron and is sent via the equivalent of the axon. If we call the output of the neuron o and the activation function σ, the output of the neuron can be written as:

$$o = \sigma(\sum_{i=1}^{m} \omega_i x_i + b). \tag{1}$$

4.1.2 Feedforward Artificial Neural Networks

Once we have the artificial neuron we can start to connect neurons and create artificial neural networks. There are different ways of connecting the neurons, leading to different structures (and possibly different abilities) of the network. The basic structure that is used in this project is called *feedforward* neural networks, which means that information is only sent forward in the structure. Opposed to *recurrent* neural networks where information can be sent backwards in the structure and loops can occur [6].

[2]http://cs231n.github.io/neural-networks-1/

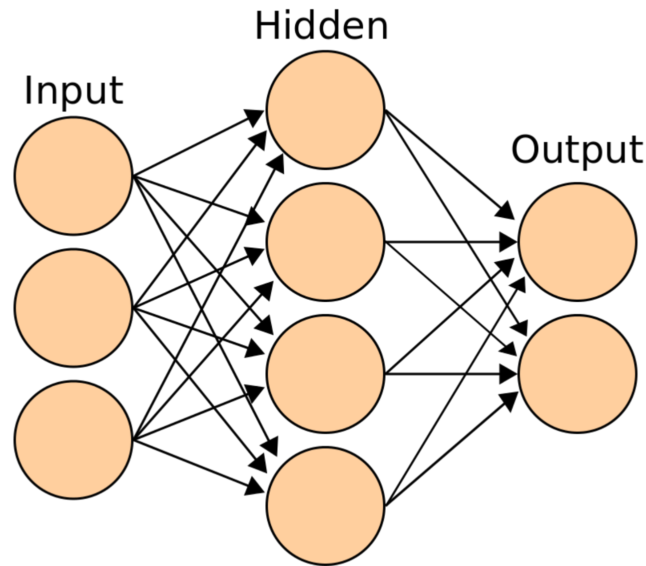

Figure 3: Illustration of a feedforward neural network. The circles represent artificial neurons and the arrows illustrate how output from one neuron acts as input to other neurons. Note how the network is divided in to layers where no information (arrow) goes between neurons in the same layer or backwards to the preceding layer.[3]

4.1.3 Layers

An artificial neural network consists of layers and in the case of *feedforward* networks they are defined so that neurons within the same layer does not have any connections (exchange of information). In general the layers are divided into three different types: input, hidden and output layers.

Input layers don't do any mathematical operations but just represent the input to the network. Consequently the input layer has the same number of nodes as number of input values to the network. For example if the input is a 3x3 matrix the input layer will have nine nodes.

Hidden layers are layers between the input and output layers. The reason they are called hidden is not important but one can imagine that the neural network is a black box with an unknown number of layers acting as a function. As a user of the network you just put in input and get output. What is in the box is hidden from you, hence hidden layers.

An output layer is the last layer in an ANN. Like the hidden layers it usually has an activation function and the outputs from the activation function in the output layer is also the output of the network. It is common to have a different activation function in the output layer than in the hidden layers, i.e. the hidden layers usually share the same activation function and the output layer has a different one. The choice of activation function in the output layer depends on the application of

[3] By Glosser.ca - Own work, Derivative of FileArtificial neural network.svg, CC BY-SA 3.0, httpscommons.wikimedia.orgwindex.phpcurid=24913461

the network. It is for example common that the output should be probabilities and thus must be limited to values between 0 and 1.

Regardless of the position of a layer (input, hidden or output) the layers can have different properties and be categorized based on those properties. The most basic layer is called fully connected (or dense) and is illustrated in the hidden and output layer in Figure 3. Fully connected means that all neurons in the layer have a connection to all neurons in the preceding layer. Other types of layers are for example Convolutional layers (see section 4.4.1), Pooling layers (see section 4.4.2) and Dropout layers (see section 4.8.3).

4.1.4 Deep artificial neural networks

Generally when one talks about deep artificial neural networks the network has more than one hidden layer. For example the network displayed in Figure 3 would not be called deep. Apart from that there are no strict rules that define a deep network. The reason that deep networks in general and deep convolutional neural networks, see section 4.4, in particular have become very popular is that it has been shown that the depth of the network is of great importance for its performance [7]. Today, state of the art networks can have over hundreds of layers [10].

4.2 Activation functions

An activation function is a non-linear function that introduces non-linearity in to artificial neural networks (ANNs). As described in section 4.1.1, an artificial neuron takes a linear combination of outputs from the preceding layer as input to an activation function. If the activation functions in a layer would be linear functions, the layer would only be able to do linear transformations. If all layers in a network would have linear activation functions, linear algebra tells us that a single layer would be enough to be able to perform the same mapping as the whole network. Thus the whole network would be limited to do linear approximations. By using non-linear activation functions, more complex functions and phenomena can be approximated. Actually, with some requirements on the activation functions an ANN is capable of approximating any continuous function. More on that in section 4.3. The activation function is also a way to model the functionality of a neuron in the brain, which connected to other neurons constitutes the computational unit in the brain that artificial neural networks are ultimately meant to mimic.

4.2.1 Heaviside function

The first activation function that was suggested in the early development of ANNs was the *Heaviside function* defined as:

$$\sigma(z) := \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0. \end{cases} \tag{2}$$

Which is a logical choice due to the "all-or-none" nature of the biological neuron. That means that the biological neuron can "fire", equivalent to activate in a artificial neuron, either a constant strength signal or nothing. I.e. no signal with intermediate strength can be fired or higher stimulation (input signals) can not increase the strength of the output signal [11]. An illustration of the Heaviside function can be seen in Figure 4

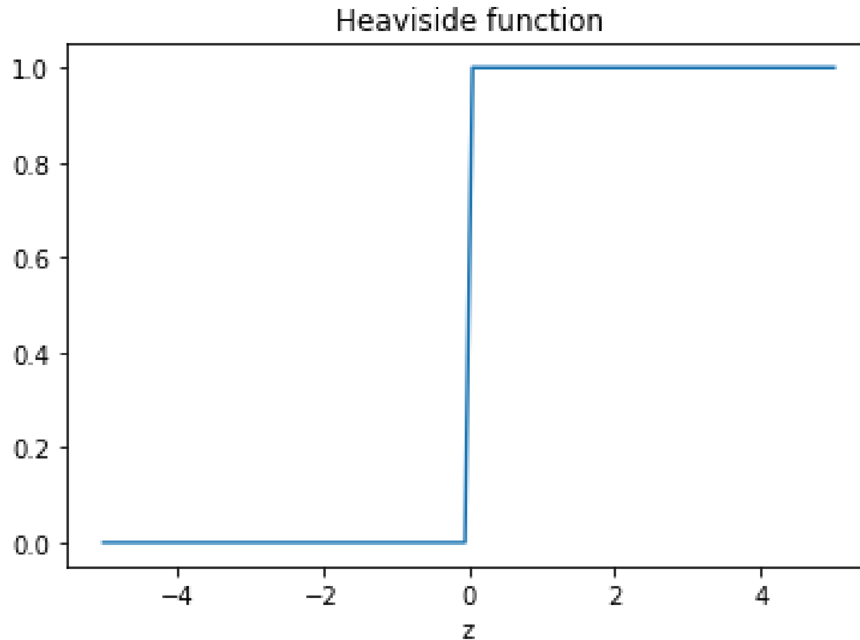

Figure 4: Illustration of the Heaviside function.

However, in order to be able to use the backpropagation algorithm (see section 4.7.4) the activation function needs to be differentiable. Also, the backpropagation algorithm requires the activation function to have the property that a small change in the input leads to a small change in the output [6]. The Heaviside function, (2), does not have that property since a small change in input may lead to either no change of the output at all or a jump from 0 to 1.

4.2.2 Sigmoid function

A remedy to the problem with the Heaviside function mentioned above was to introduce the *Sigmoid function* defined as:

$$\sigma(z) := \frac{1}{1 + e^{-z}}. \tag{3}$$

The Sigmoid function can be seen as a smoothed out version of the Heaviside function that have the required properties that the Heaviside function lacks (see section 4.2.1). An illustration of the Sigmoid function can be seen in Figure 5

As can be seen in (3) and to some extent in Figure 5 the sigmoid function resembles the heaviside function when the magnitude of the input value is large. The sigmoid function goes towards 0 when the input goes towards $-\infty$ and towards 1 when the input goes towards ∞. Thus, the sigmoid function can be seen as approximating the "all-or-none" property (see section 4.2.1) of the biological neuron.

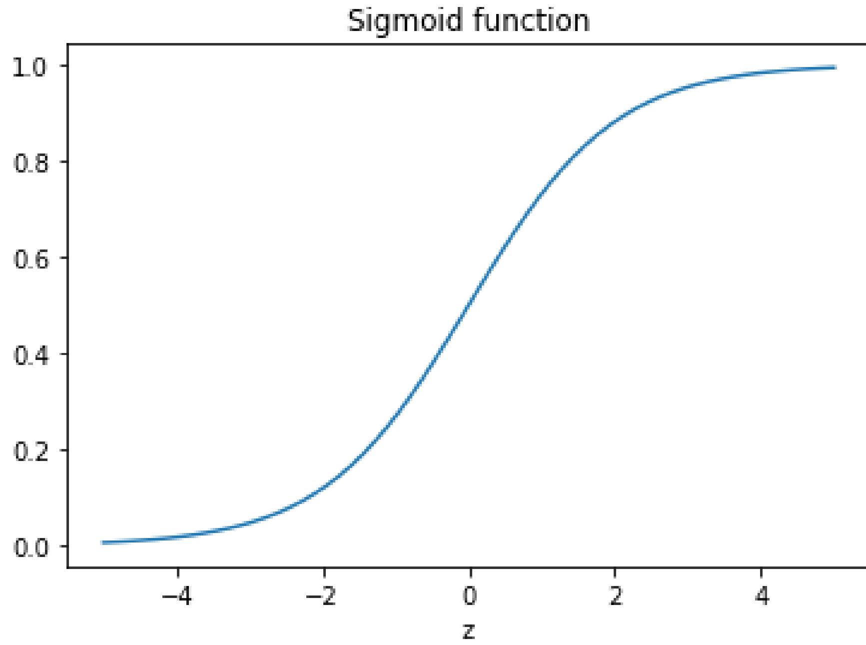

Figure 5: Illustration of the Sigmoid function

4.2.3 ReLU (Rectified linear unit function)

ReLU (or rectified linear unit) is another popular activation function and is defined as:

$$\sigma(z) := \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0. \end{cases} \tag{4}$$

An illustration of ReLU can be seen in Figure 6.

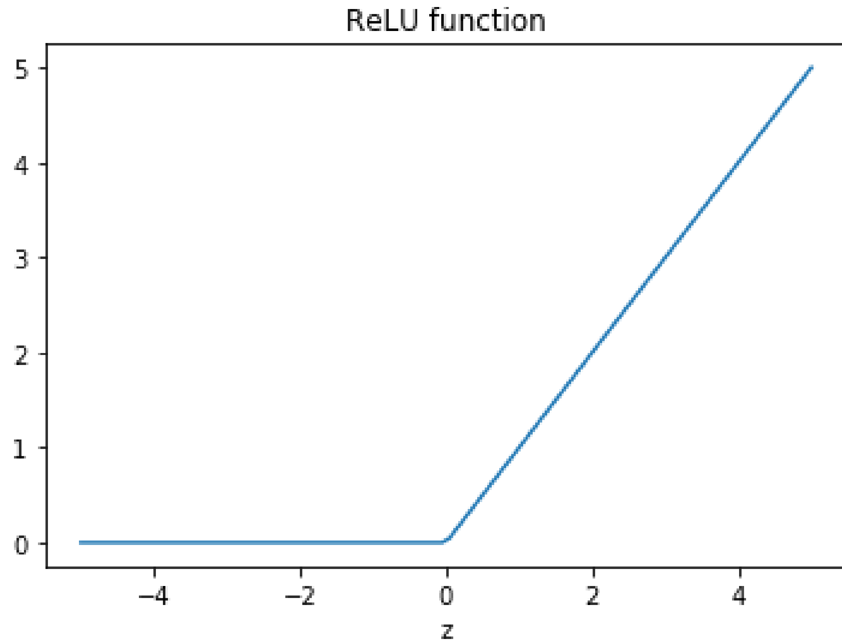

Figure 6: Illustration of the ReLU function

One of the reasons to why ReLU was introduced was because it partly remedies the "vanishing gradient problem" that is described in section 4.7.7 and that it has the property of zero output for negative inputs (to mimic a biological neuron, see section 4.1.1).

4.2.4 ELU (exponential linear unit function)

An extension to ReLU is another activation function called ELU and it is defined as:

$$\sigma(\alpha, z) := \begin{cases} z & \text{if } z \geq 0 \\ \alpha(\exp(z) - 1) & \text{if } z < 0. \end{cases} \tag{5}$$

Where α defines at which negative value the ELU saturates. An illustration of ELU (with α=1) can be seen in Figure 7

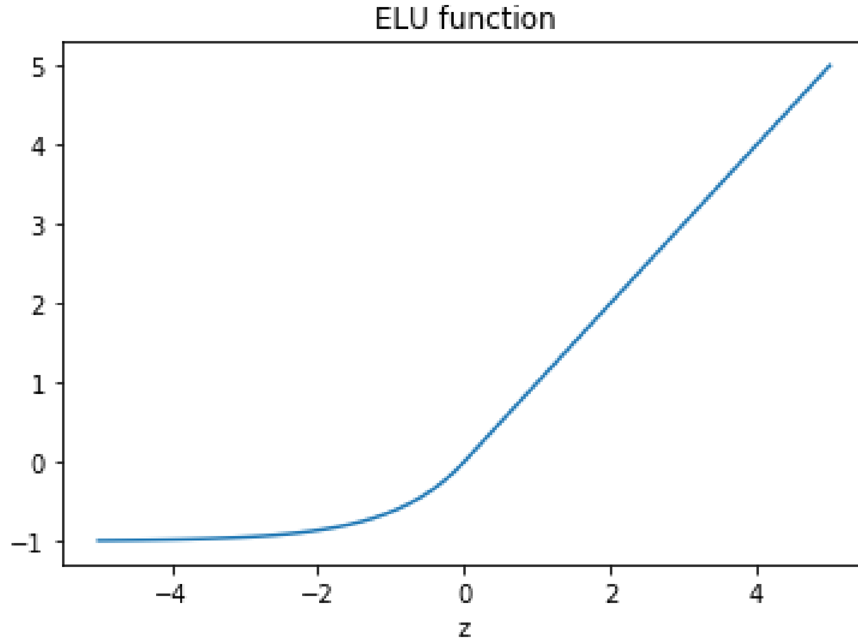

Figure 7: Illustration of the ELU function (with $\alpha = 1$)

ELU is similar to ReLU but with the difference that it has a non-zero gradient for negative inputs, which is advantageous considering the "vanishing gradient problem" (section 4.7.7).

4.3 Universal Approximation Theorem

Ultimately, an ANN is a function approximator that through training is trying to approximate an underlying function as closely as possible. In the field of machine learning the function to be approximated is usually some unknown complicated function. The *Universal approximation theorem* is a mathematical theorem that justifies the use of artificial neural networks to approximate functions. It connects the extensive mathematical theory of approximating functions to neural networks which doesn't have an as extensive mathematical theory yet. In [12] the theorem is stated as follows:

Let g(.) be a non-constant, bounded, and monotone-increasing continuous function. Let I_D *denote the D-dimensional unit hypercube* $[0,1]^D$. *The space of continuous functions on* $[0,1]^D$ *is denoted by* $C(I_D)$. *Then, for all* $f \in C(I_D)$ *and* $\epsilon > 0$*, there exist an integer M and sets of real constants,* $\{\omega_j^{(2)}, \omega_{ji}^{(1)}, \omega_{j0}^{(1)} | i = 1, ..., D$ *and* $j = 1, ..., M\}$ *such that the approximate realization of* $f(.)$

$$F(x_1, ..., x_D) = \sum_{j=1}^{M} \omega_j^{(2)} g\Big(\sum_{i=1}^{D} \omega_{ji}^{(1)} x_i + \omega_{jo}^{(1)}\Big) \tag{6}$$

satisfies

$$|F(x_1, ..., x_D) - f(x_1, ..., x_D)| < \epsilon \tag{7}$$

for all $(x_1, x_2, ..., x_D) \in I_D$.

Equation (6) represents a neural network with one hidden layer. Thus, stating that a single hidden layer NN is capable of approximating any continuous function. As a consequence, also NN's with multiple hidden layers are capable of approximating any continuous function. For example, add a layer to a single hidden layer NN described by (6) that conducts unity mapping of the inputs. I.e. the layer takes the output from the previous layer and matrix multiplies it with the identity matrix and outputs the result. So, the theorem effectively states that a neural network, satisfying the condition described, is capable of approximating a continuous function arbitrarily good. It is an *existence theorem* in the sense that it states that a single hidden layer NN is sufficient for the approximation. However, it doesn't state that a single hidden layer NN is optimal in the sense of learning time, ease of implementation or generalization [12].

In addition, one of the condition on the activation function ($g(.)$) in the theorem is that it should be bounded. However, ReLU for example (see section 4.2.3) does not satisfy that condition. Despite this, Sonoda and Murata [13] was able to show that the *Universal approximation theorem* also holds for neural networks with unbounded activation functions.

4.4 Convolutional neural network

Convolutional neural networks are a type of artificial neural network that uses convolutions in so called convolutional layers. In image recognition tasks, network designs based on the principle proposed by Y. LeCun [14] have been very prominent. Progress in computational resources and further research have led to deeper and more advanced networks and extensions to Y. LeCun's network. One example that has performed very well in the application of autonomous driving is the one created by NVIDIA [15].

The motivation for using convolutional neural networks can once again be found in nature. Hubel and Wiesel [16] showed that the visual cortex contains a complex arrangement of cells where each cell is sensitive to a small region of the visual field. These small regions are called *receptive fields* and are represented by kernels in convolutional neural networks. Together the *receptive fields* cover the entire visual field and are well suited to detect local spatial correlation features.

The reason that convolutional neural networks are especially suited for processing images is that images usually have a spatial structure. It can be illustrated in that pixels in an (natural) image are usually not uncorrelated to adjacent pixels. For example, if a pixel has nine directly adjacent pixels that are blue it is highly probable that also the concerned pixel is blue. This local spatial correlation is taken advantage of by using convolutional layers described next.

The dominating structure for convolutional neural networks has been based on three types of layers; convolutional, pooling and fully-connected layers. Their roles are described in the following sections.

4.4.1 Convolutional layer

The foundation in a convolutional neural network is the convolutional layer. The layer is inspired by the receptive fields of the visual cortex described above. The equivalent to the a receptive field in the visual cortex in a convolution layer is called a kernel (or filter or feature extractor). The kernel is a convolution filter that convolves the input data. Convolution operations can be performed in many dimensions but in this thesis 2D-convolution are performed. Technically the input images in this project will be images represented by 3D matrices with dimensions $h \times w \times d$ where $h \times w$ is the height and the width of the image (resolution in pixels) and d is the number of color channels.

In this thesis RGB images are used, which means that there will be three color channels. However, each color channel can be handled separately in the convolution operation so only 2D convolutions need to be performed. If the input image is described by a two-dimensional image, I, and a kernel, K, with dimensions $h \times w$ the convolution can be described as:

$$(I * K)_{xy} = \sum_{i=1}^{h} \sum_{j=1}^{w} K_{ij} \cdot I_{x+i-1,y+j-1} \tag{8}$$

where $(I * K)_{xy}$ is an element in the resulting matrix $(I * K)$ called feature map. The operation is illustrated in Figure 8

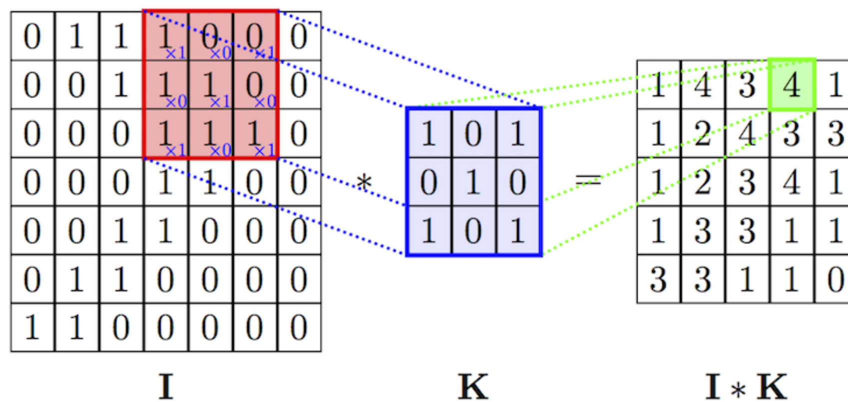

Figure 8: Illustration of the convolutional operation.[4]

Convolutional layers are in many ways similar to fully connected layers. In fact, if certain weights between two fully connected layers are set to be zero, which is equal to that no connection exists, the later layer can be equivalent to a convolutional layer. In Figure 8 every element in matrix I represents an output from a neuron. The elements in the kernel K represents weights between the layers. Observe that in the figure only the nine weights belonging to the kernel are shown. Technically, there are also weights equal to zero between the neuron in question ($(I * K)_{xy}$) and all the neurons in I not marked in the figure. Finally, the elements in matrix $I * K$ represent inputs to neurons in the next layer. By convolving the kernel over the input, I, the resulting feature map, $I * K$, is formed.

There are three parameters to be set for the layer; kernel size, stride and padding. Kernel size is plainly the size of the kernel. For example is the kernel size 3×3 in Figure 8.

Stride is how many steps horizontally and vertically the kernel should take over the input matrix (I) between every input to the feature map ($I * K$). Depending on the kernel size and the stride the kernel will overlap. For example a kernel size of 3×3 with stride 1 will give a large overlap, whereas a kernel size of 3×3 with stride 3 will give no overlap.

Padding means to increase the dimension of the input map, I, by adding elements around its edges. This is usually done by adding zeros (zero-padding) and is a way of limiting reduction in size from input map to feature map without distorting any information.

[4]https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html

To calculate the size of the resulting feature map (along one dimension) for any given convolution the following formula can be used.

$$O = \frac{W - K + 2P}{S} + 1 \tag{9}$$

where O is the feature map dimension, W is the input dimension, K is the kernel dimension, S is the stride and P the padding.

Prominent features of convolutional layers One rational idea behind convolutional layers is the notion of local feature detection. As mentioned in the beginning of this section the spatial correlation between pixels makes it advantageous for a neuron in the convolutional layer to be constrained to inputs within its receptive field (kernel size). Since features usually appear locally (an edge for example) there is no reason for the neuron to take in information from the entire input image in order to detect that particular feature. These local features can then be combined in to higher order features (edges in to shapes for example) in the next convolutional layer. This is why the kernels are also called feature extractors

Another asset of convolutional layers is *weight sharing*. As mention above features are usually local, but they can appear anywhere in the input. The feature extractors (kernels) thus have to be applied over the whole input, hence the convolution. The idea of weight sharing is that the feature extractor (kernel) has the same weights regardless of which area it is convolving over. In other words, the feature extractor is extracting the same feature over the whole input. Which means that all neurons in the feature map share the same set of weights. As a consequence, weight sharing greatly reduces the number of weights in the network.

By applying several different feature extractors on the input, multiple feature maps can be created. These feature maps in turn can be sent in as input to another convolutional layer (with several feature extractors) and create new feature maps and so on. In every layer higher order features can be extracted and in the end combined to form an appropriate output depending on the purpose of the network. For example a steering angle which is the case in this thesis.

4.4.2 Pooling layer

Historically, in convolutional neural networks convolutional layers have usually been followed by pooling layers also sometimes called down-sampling layers. Pooling layers reduce the spatial dimension of the feature maps to decrease the number of parameters (weights and biases) in the network, ideally without loosing important information. The idea is that the feature map can be compressed into a smaller feature map without loosing too much information. The operation can be described as applying a function on a subregion of the feature map and get a scalar output instead. Usually a 2×2 subregion is transformed into a scalar.

The most common operation is called *maxpooling* and can be described as

$$y = max(x_1, x_2, ...x_n) \tag{10}$$

where y is the resulting scalar and n is the number of elements in the subregion. Maxpooling is illustrated in Figure 9. As illustrated in the figure the parameters stride and filter-size, mentioned in relation to convolutional layers, are also applicable in maxpooling operations. The most common configuration is the one depicted in the figure where stride is set to two and filter-size to 2×2. This operation greatly reduces the size of the feature map, as can be seen in the figure.

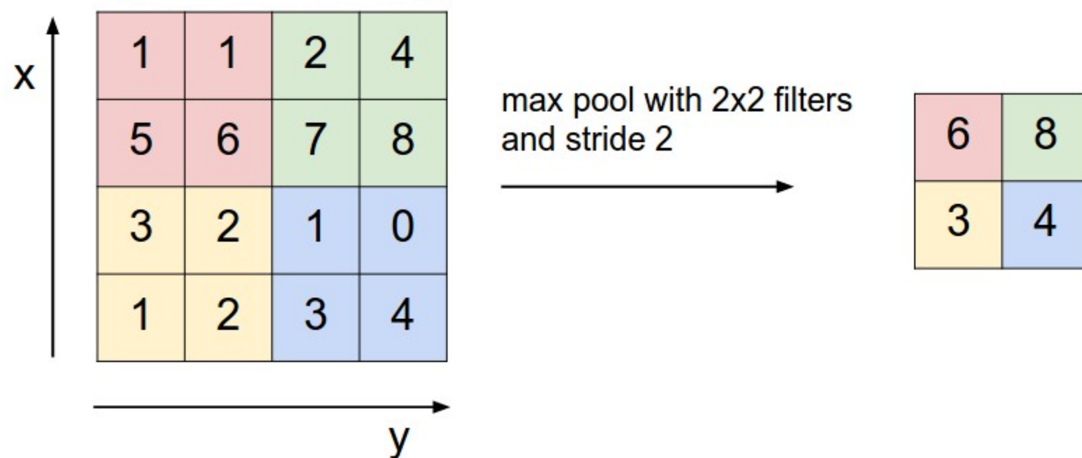

Figure 9: Illustration of a maxpooling operation.[5]

There are other types of pooling operations. For example average pooling where instead of taking the maximum value in the subregion the average of the elements is calculated. Regardless of type of pooling layer it has become more common to exclude them from the network. Instead, in order to reduce the size of the feature maps larger strides in convolutional layers can be used for example [17].

4.4.3 Upsampling layer

An upsampling layer operates in the opposite way of a pooling layer (which is also called down-sampling layer). I.e. it increases the size of the feature map. In its most basic form, for every input pixel, an upsampling layer copies the value to all pixels in a larger matrix [35]. For example, an upsampling layer that expands the input feature map by two in both dimensions (assuming 2D feature maps) would expand a 2×2 feature map filled with ones to a 4×4 feature map filled with ones. If the kernel used by the layer is larger than 1×1 the same kind of operations can be done as in pooling layers. For example taking the maximum value in the kernel and copying it to the larger feature map, taking the average over the kernel etc.

4.4.4 Fully connected layer

As described in section 4.1.3, in fully-connected layers all neurons are connected to all neurons in the preceding layer. In convolutional neural networks (especially for image recognition tasks) the convolutional and pooling layers are usually followed by one or multiple fully-connected layers. The fully-connected layers are often considered to be the classification part of the network, whereas the convolutional/pooling part is considered the feature extraction part. This is at least the idea behind it even though there is no way of separating the tasks in a network that trains itself (see

[5]http://cs231n.github.io/convolutional-networks

section 4.7). Intuitively, one can say that the convolutional part of the network extracts important features and removes unimportant information (noise) from the input. The extracted features are then fed in to a fully-connected network that based on the information about the important features produces an output. Imagine for example that you have a network that is supposed to classify if an image contains a dog or not. A fully-connected network would take the whole image as input and consider every pixel in order to make a classification. On the other hand, a fully connected network preceded by convolutional layers would only get information about important features such as if there is a tail, ears, two eyes, fur etc. in the picture and base its classification on that. That way less information is fed to the fully-connected layer and the network can in its turn have fewer neurons. In this thesis the output is a steering angle.

4.5 Autoencoder network

Another type of network that will be covered briefly is called an autoencoder. An autoencoder is an artificial neural network consisting of an encoder followed by a decoder. In it's essence, an encoder takes input and downsizes it (in the case of images reducing the resolution) and a decoder does the opposite [34]. Usually, the task of an autoencoder is to reproduce the input. This may seem like a futile function but by training the network to reproduce the input, the encoder part of the network might learn how to downsize the input and keep the most important information for reproducing the input. In other words it might learn to compress data without loosing important information. A function that is used extensively when storing data.

4.6 Data

In order for a network to train well and ultimately reach a good performance the data used for training is of vital importance. Especially in the case of *Supervised learning* (see section 4.7.1), which is utilized in this project. Both the quality and the sheer amount of data is important. The quality in this case means that the data contains all necessary information in order for the network to learn correct behavior. For example, if a network for self-driving vehicles is fed exclusively with data representing left turns during its training. It will not be able to handle a right turn properly.

4.6.1 Collection

What the training process is essentially trying to do is to figure out the underlying structure of the data it is fed with. Thus, theoretically it is important that the data used for training represents the reality well. Quantitatively this means that the distribution of the training data has the same features as the distribution of reality. Features in this case can be parameters like mean and standard deviation.

It is also important that the examples constituting the training data can be seen as independent and identically distributed random variables from the underlying distribution representing the reality. If this is fulfilled along with the requirements mentioned in the previous paragraph the model will be able to generalize. In other words, the model will be able to use the structure it has learned from the training data to predict outcomes for unseen data. If the unseen data also shares the same features as the underlying data [20].

4.6.2 Partitioning of data

One important act is to partition the data into appropriate sets in order to be able to evaluate the model properly. The sets are usually called training set, validation set and test set. The fundamental reason for the partition is that the model can not be evaluated on the same data as it has trained on.

To make it clearer let's use an analogy. Suppose you are trying to teach a kid multiplication and prepare him/her for a test. As a resource you have a table of all multiplications from 0×0 to 10×10 (i.e. 100 multiplications). If the kid has a very good memory and is given the whole table, he/she would be able to memorize the entire table. Consequently, if given a test consisting of executing 10 multiplications from the table, the kid would be able to score maximum points. Yet, the kid doesn't have to know anything about how to perform multiplication, he/she simply uses his/her memory. Thus, the test would not be a good measure of his/hers multiplication skills.

Instead, a better method would be to remove a certain number of multiplications from the table and give the reduced table to the kid to practice on. Then use the removed multiplications as a test. Thus, at the test the kid is faced with new unseen multiplications and the test score will be a better measurement of how well the kid has understood the concept of multiplication. In this analogy the reduced table represents the training set and the removed multiplications the test set.

Now, suppose you are a teacher and have a whole class of kids in the same situation as above. You are supposed to select one of them to represent the class in a competition. The competition will consist of performing 10 specific multiplications from the full table. You are given the 10 multiplications beforehand. Let us call these 10 multiplications the competition set. Since it would be cheating to let the kids practice on the competition set you remove those specific multiplications from the full table. Then, as a way to select the kid with the best multiplication skills you remove 10 additional multiplications from the reduced table to form another test just for the class. Let us call these 10 multiplications the class-test set. Now, the twice reduced table (now consisting of 80 multiplications) can be used as a training set for the class. The class-test set can be used to select the kid who should represent the class in the competition. Lastly, the competition set can be used as an ultimate measure of the kid's multiplicative skills.

In the analogy above the twice reduced table represents the training set, the class-test set the validation set and the competition set the test set. The features of the different sets are described further below.

Training set The training set is used in the training algorithm to adjust the parameters (weights and biases) of the network. In the analogy the training set is used to change/develop the kid's brain.

Validation set The validation set is used to evaluate a network with a certain setup of hyperparameters (see section 4.10). The hyperparameter setup can be seen as the design of the network. The validation set doesn't affect the parameters. In the analogy, the validation set is used to measure which kid has the most promising abilities. Note that the analogy is not perfect here. A kid would be able to learn (adjust parameters) from a test consisting of the validation set. Whereas an ANN will not change it's parameters due to the validation set

Test set The test set is used as an ultimate measure of the models performance and consists exclusively of previously unseen data. In the analogy, the test set is used as a performance measure

of how well the best kid in the class has understood the concept of multiplication.

Holdout method The most basic way to partition the data into training, validation and test set is called the holdout method. It is based on an assumption that the total data available is limited. A proportion of the data is then simply set aside as validation data and the same is done for the test data. The proportions the sets are divided into can be considered as hyperparameters [21].

Cross-validation Another method to partition the data is called cross-validation. In contrast to the holdout method, the partitioning of data is done repeatedly. Every time a partitioning is done a split is created, i.e. a split is one possible set-up of training and validation set. Usually the holdout method is used to separate the test set from the other data so the test set will be left out of this paragraph.

There are many ways the cross-validation can be performed. However, the common factor and difference compared to the holdout method is that several splits are created from the original data. So ,the holdout method created one split and train on the training set and validate on the validation set. In contrast, the cross-validation method creates k splits. Train k identical networks on the k training sets and validate on the k validation sets. Then, by averaging the parameters (weights and biases) over the k trained networks a final network is formed [22].

Actually, the holdout method can be considered as the simplest form of cross-validation. If k is set to 1 in the previous paragraph the resulting method will be the holdout method. However, the holdout method is usually considered a separate method since it is a very commonly used version of cross-validation.

The advantage of cross-validation methods are that they makes use of the whole data set in the training process. Thus, it can be advantageous to use cross-validation when the available data is very limited. A disadvantage is the extra computations required due to the multiple networks trained simultaneously. If for example k is set to 10, the number of computations will be 10 times as high compared to if the holdout method is used. Consequently, the holdout method is probably preferable when large neural networks are trained.

4.6.3 Over-/undersampling

Two methods to enhance the quality of the data (see section 4.6) are called over- and undersampling. The methods are remedies to biased data, which means that the data is biased in such a way that it affects the quality of the trained network. All dataset sampled from reality is basically biased more or less. However, when the data is too biased it leads to unwanted effects in the training process. In the case of this project the data consists of images and corresponding steering angles and the range of the input is in theory continuous. I.e. the steering angles can have any value between two limits, where the limits are typically defined by the cars turning radius. Biased data in this case can be for example that steering angles around 0 (i.e. driving straight) constitutes a very big part of the data. Which may lead to that the network learns how to handle curves poorly. If the data is severely biased it may even lead to the network having a constant output. The reason is that if the network puts out a constant value close to the highly over represented steering angle, the cost (see section 4.7.2) can still be very low.

A remedy to biased data is to over or undersample the data. Oversample the data means that examples of the underrepresented class (steering angles within a certain interval in this case) in the data set are duplicated one or several times. Consequently, oversampling entails an expanded data

set. As the name indicates undersampling is the opposite action of oversampling. I.e. undersampling is the action of removing examples from the overrepresented class. In the example above with steering angles around 0 being overrepresented, oversampling would mean duplicate examples with steering angles that differs from 0 (with a certain tolerance). Undersampling would in the same manner mean removing examples with steering angles equal to 0.

4.6.4 Augmentation

Augmentation is a plethora of methods that can be used to increase both the size and quality of a data set. The common factor for the methods is that the data set is augmented by some artificial data. Here artificial means that data are added to the original data set by manipulating examples in the original data set. In other words, the data set is expanded without collecting new data or duplicating original data.

As mentioned above there are a plethora of augmentation methods that can be utilized. As a demarcation only relevant methods utilized in this project are described here.

Horizontal flip A very useful augmentation method to increase both the quality and size of the data set is to perform a horizontal flip on the input. In this project a horizontal flip is the action of interchanging the values of opposite pixels with the same distance to the center of the image, with regards to the horizontal axis. By adding the flipped images to the original data set an augmented data set is created. In addition to increase the size of the data, horizontal flipping inhibits the data set to be biased towards any direction. In theory if horizontal flipping is performed on every example in the original data set, the augmented data set will be unbiased. The logic reason that horizontal flip can be used in this project is that the steering angle of a flipped image should be the same as for the original image but with switched sign.

Translation A common augmentation technique, when the inputs are images, is called translation. The technique is especially useful in feature detection when the exact location of the feature is unimportant. For example, if a neural network is trained to recognize a cat. It should not matter if the cat is located in the center of the image, in the upper left corner etc. The method simply shifts the pixels in an image a certain number of steps horizontally and vertically. The method needs to be further defined for its application. One problem to be solved for example is what to do with the pixels that end up outside of the frame when they are translated. A common solution to this problem is to only use a subsection of the input images as input to the network, preferably the center. This "cropping" is done regardless of if translation is performed or not. Consequently, if translation is to be performed a maximum number of steps for the translation can be set.

However, in the case of self-driving vehicles the location of features is important. Hence, in contrast to classification problems the output of the network should change if the input image is translated.

Shadowing Another way to augment the data set is to add artificial shadows to images. This is yet another way to expand the data set by adding images that mimics real life data. Depending on the time of the day and the surrounding environment, a network for self-driving vehicles may encounter shadows from all possible directions when put in real life. Thus, in order to train the network as good as possible it is beneficial to expose it to as many real world situations as possible

during training. The technical implementation of the method is simply to darken, or even blacken, certain pixels that together form the pattern of a natural occurring shadow.

Brightness With the same reasoning as in the previous paragraph it is useful to augment the data set with images with varying brightness. The idea is that by brighten or darken all pixels in an image, different real life conditions can be simulated. A sunny day would for example generally render brighter input images than a cloudy one. Also, during the course of a day the brightness varies naturally from sunrise to sunset.

4.6.5 Data Preprocessing

In general there are two main ways of preprocessing the input data; zero-centering and normalization. These actions are made to make the training faster and more robust. The type of preprocessing that is made depends on the purpose of the network but in general it is to make the network robust and be able to handle input values with different scales. How the preprocessing is performed also depends on the activation function used in the network (or in the first hidden layer if different activation functions are being utilized throughout the network). Usually, it is preferable to have input values around 0 and with magnitudes not much greater than 1. The reason for these preferences is to avoid slow learning by the network due to the shape of the current activation function utilized. This phenomena is discussed further in section 4.7.5. In that section it is also explained why different measures have to be taken depending on activation function. Note that that section discusses issues that arise in the initialization of the network parameters, but the same reasoning applies to the preprocessing of the data. There are more ways of preprocessing the input data than the two that will be mentioned here. Which methods that are used is highly dependent on the problem and therefore the presentation is restricted to the two methods below since they are the only ones considered relevant for this project. For further reading see for example [23].

Zero-centering Zero-centering is probably the most common preprocessing procedure and can be geometrically interpreted as centering the cloud of input data around the origin along every dimension. Probably the most common method is referred to as *mean subtraction* and is, as the name suggests, the procedure of subtracting the mean along every dimension of the input data from every individual input value [23]. If however the input values are known to be reasonably even spread out within an interval, simpler actions may be satisfactory. As in the case when the inputs are RGB images and thus have pixel values limited to the interval 0 to 255. Then instead of calculating and subtracting the mean, the center of the interval can be used instead. Which in this case would be 127.5.

The main motive for zero-centering the data is to avoid slow learning as mentioned above. This problem can occur when the sigmoid (or any similar) activation function (see section 4.2.2) is used. This problem, as aforementioned, is discussed in section 4.7.5.

Normalization Normalization is the action of normalizing the different dimensions in the input data into the same scale. Two important consequences of normalization are robustness against input values with different scales and avoidance of slow learning. This can be especially important if the input data consists of different quantities with equal importance [23].

Notably, as in the case of this project, when the input consists of images the input values are already basically in the same scale. The input values are pixel values that usually range from

0 to 255. Thus, normalization may be redundant in these situations. A counter example where normalization is vital would be a network that calculates the yield of solar panels based on area and latitude. Here the input values are in square meters and degrees, two different units with totally different ranges.

However, even if the input values are of the same scale normalization may be important due to the second consequence of normalization mentioned above. Namely avoidance of slow learning. Once again the reader is referred to section 4.7.5 for a thorough explanation of the problem.

The normalization can be performed in varying ways. One popular way is to divide each dimension by its standard deviation. Another one is to simply normalize each dimension so that the min and max of the dimensions are -1 and 1 respectively [23]. If the data is fairly consistent within a certain interval, the input values can simply be normalized with regards to that particular interval. For instance when the input consists of RGB images and therefore have values between 0 and 255. Then simply dividing each input value by 255 may be a sufficient normalization.

4.7 Network training

Network training can be seen as a set of rules and algorithms conducted to change the network so it performs better on an assigned task. The rules and algorithms are based on the concept of machine learning. To define the concept of learning in general and in machines in particular is not a trivial matter. Since this report is about neural network learning, a definition of learning in the context of neural network will be presented here. The definition comes from *Neural Networks* by Simon Haykin [12].

Learning is a process by which the free parameters of a neural network are adapted through a process of stimulations by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameter changes take place.

In this section of the report different aspects and considerations of the training process will be presented and explained.

4.7.1 Types of learning

There are several ways to classify learning algorithms. One well established way that has prevailed through the years is to divide them according to what kind of data that is used in the learning process. The three major classes used are *supervised learning*, *unsupervised learning* and *reinforcement learning*. Even though only supervised learning have been used in this project, all three classes will be briefly explained in the following paragraphs for completeness. Note that other classes may exist and that algorithms may be a mixture of more than one class.

Supervised learning In the supervised learning paradigm, prior knowledge about how to interpret the training data is required. This prior knowledge comes in the form of labeled input data, or every input is accompanied by an output. The output belonging to a specific input is regarded as the optimal response to that particular input. The goal of the training is for the network to infer the underlying rules that correlates the features in the input to the output. In mathematical terms one might put it like this. Denote the underlying function $h : \mathbf{R}^m \rightarrow \mathbf{R}$ and call every input

example $\mathbf{x}_i \in \mathbf{R}^m$ and the corresponding correct output $y_i \in \mathbf{R}$. Then, the goal is for the network to, to the furthest extent possible, mimic the function $h(\mathbf{x}_i) = y_i$.

In supervised learning, the learning algorithm lets the network "guess" (at least initially let's call it guess) the output for a specific input. It then compares the correct output with the one from the network and uses the difference or error to adjust the parameters in the network. This procedure is iterated and the idea is that the network will improve its ability to predict the correct answer over time. With quality data sets, the theory then states that not only will the network improve it's performance on the training set, but also on previous unseen data. This because it have learned the underlying rules that determines the output based on the input features.

One negative aspect of supervised learning is that the collection of data might be time consuming and expensive. Since the data contains a correct answer to every example, some kind of expertise generally has to be consulted for the collection of those answers. If the problem the network trains for is not trivial, this might be a costly process. Suppose for example that a network trains to recognize tumors in x-ray images. In this case an expert within the field has to label all the images in the data set correctly.

Unsupervised learning Unlike in supervised learning, in unsupervised learning there are no associated outputs to the inputs in the data set. Thus there is no defined goal for the network to strive towards. Hence, unsupervised learning is commonly used in situations when either no clear answer exists or simply no prior knowledge of the answer exists. What unsupervised learning predominantly is utilized for is clustering, i.e. grouping the input into different categories based on features in the input. Note that these categories are not known in advance, but the user has to pre-set how many categories that should exist. This is an example of a hyperparameter (see section 4.10) [20]. For example can a *competitive learning rule* be used to perform unsupervised learning [12]. Since unsupervised learning is not used in this project it will not be discussed further here, but for the interested reader more information can be found in [12].

Reinforcement learning Reinforcement learning is a separate category of machine learning algorithms since it's not supervised neither unsupervised. It is usually introduced to be inspired by behaviorist psychology. The reason is that it works in a similar manner to a human in certain situations. The network (or machine) learns by trying to optimize some cumulative value by taking action in an environment. The cumulative value is often called reward and the network is either rewarded (increase of the cumulative value) or punished (decrease of the cumulative value) depending on the action it takes [30].

Unlike in supervised learning there is no "correct" action/output corresponding to each input. In that manner it's no supervision of the learning. However, the network is given a certain goal to achieve, but it's not told how to reach that goal. Consequently, instead of comparing every action of the network with a correct one, the network is rewarded for actions that increase the cumulative value and the network has to find the actions itself that ultimately yield the highest cumulative value. So in a way the learning process is supervised, since the environment gives feedback (reward or punishment) depending on the action.

An example that relates to the behaviorist psychology analogy would be how a human learn how to play chess. There is no one telling you exactly what move you should make in every situation, neither is there a correct action to every situation for a network in reinforcement learning. However, by winning or loosing the game you still get feedback on how well you performed. In the same way

will a network under training be rewarded if it wins the game after a sequence of moves. Essentially, the network learns by trial and error.

4.7.2 Loss functions

In order to train a network it's necessary to have some kind of measure of how the network performs. The purpose of the training is to improve that measure. For the measure to be effective it must be able to be used as feedback in order for the network to improve. In supervised learning in general and in this project in particular that measure is called loss (or sometimes cost or error) and is defined by a loss function. The loss function is a function that maps the outputs from a network to a single value, the loss. In the backpropagation algorithm (see section 4.7.4) the loss is a measure of how well the output of the network matches the desired output [1].

For reasons that will be clear in coming sections there are two requirements on the loss function.

1. The loss function needs to be able to be written as an average $Q = \frac{1}{n}\sum_x Q_x$ over loss functions Q_x, for n individual training examples, x [1].
2. The loss function needs to be continuous and differentiable.

The requirements are due to the *Backpropagation algorithm* that is used in the training process, see section 4.7.4. The *Backpropagation algorithm* uses the gradients of the loss function for individual training examples (hence requirement 2), to calculate the total gradient of the loss function (hence requirement 1).

Quadratic loss function A widely used loss function is the *Quadratic loss function* (also called *mean squared error* or just *MSE*) and is defined as:

$$Q(\omega) = \frac{1}{2n}\sum_x ||t(x) - y(x)||^2 \tag{11}$$

where n is the number of training examples, x are individual training examples, $t(x)$ is the desired output for training example x and $y(x)$ is the output from the network for training example x. The 2 in the denominator is just there to make the expression of the derivative cleaner. Note that the loss function is written as a function of ω (weights), but that it also is a function of b (biases). One could also write it as a function of y, but since y is a function of ω and b it is simply left out of the expression. Also, focus in the training process will be to optimize Q with respect to the weights and biases so the loss function will from now on be denoted $Q(\omega)$ where ω represents both the weights and the biases.

There are a plethora of possible loss functions to be used but no more will be touched upon in this project and hence they will not be mentioned here. For further reading see for example [6].

4.7.3 Stochastic gradient descent

Stochastic gradient descent is an optimization method to find an optima for an objective function $Q(\omega)$ that consists of a sum of differentiable functions, $Q_i(\omega)$, (called summand functions).

$$Q(\omega) = \frac{1}{n}\sum_{i=1}^{n} Q_i(\omega) \tag{12}$$

Standard iterative gradient descent methods approximates the gradient of the objective function from a subset of the summand functions in every step. Thus, by updating the gradient in every step using different subsets the idea is that the updated gradient converges towards the real gradient. This is called *Stochastic gradient descent*.

When the subset of summand functions described above only consists of one function , i.e. the approximated gradient is updated by every summand function's gradient, the method is said to be "on-line" [2].
Note however that it is still a version of *Stochastic gradient descent*.

Consequently, when optimizing with regards to ω the *Stochastic gradient descent* method will update ω according to

$$\omega = \omega - \eta \nabla Q_i(\omega) \tag{13}$$

where η is usually called the learning rate when it comes to machine learning, but in general it can also be called step-size.

Momentum An extension of the Stochastic gradient descent is to add something called momentum. The method stems from the physical analogy and tends to decrease fluctuations of the update of ω. This is done by making the update of ω a linear combination of the gradient, $\nabla Q_i(\omega)$, and the previous update, $\Delta\omega$. The iteration step can then be written as

$$\omega_{n+1} := \omega_n - \eta \nabla Q_i(\omega_n) + \alpha \Delta\omega_n, \tag{14}$$

where $\Delta\omega_n$ is ω_n-ω_{n-1} i.e. the previous update of ω. α is a coefficient that determines how much of the previous update that should be considered when updating ω [2]. α is sometimes called the momentum coefficient, even though it is more like friction in the physical analogy.

RMSProp (Root Mean Square Propagation) RMSProp is another extension of the Stochastic gradient descent. It adapts the learning rate for each parameter based on the magnitude of the gradient of the function evaluated for that particular parameter. The idea is to improve convergence by increase the learning rate when the magnitude of the gradient is small and the opposite when the magnitude is large. This is done by dividing the learning rate by the square root of a running average, $v(\omega, t)$, of the mean square of the magnitude of previous gradients for the parameter in every iteration [2]. The running average is updated as follows:

$$v(\omega_n) = \gamma v(\omega_{n-1}) + (1-\gamma)(\nabla Q_i(\omega_{n-1}))^2 \tag{15}$$

where γ is the forgetting factor. Here $(\nabla Q_i(\omega_{n-1}))^2$ represents the inner product.

By taking the square root of the running average, $v(\omega_n)$, and divide the learning rate by it the updating formula for the weights becomes [2]:

$$\omega_n := \omega_{n-1} - \frac{\eta}{\sqrt{v(\omega_n)}} \nabla Q_i(\omega_{n-1}) \tag{16}$$

Adam (Adaptive Moment Estimation) Adam is a further extension of the Stochastic gradient descent method that combines and uses *Momentum* and *RMSProp*, described above. The idea is to take advantage of the benefits from both methods. If m_n represents the momentum (or even

the exponential moving average) and $v_n = v(\omega_n)$ the mean square running average the updating formulas can be written as [5]:

$$
\begin{aligned}
m_n &:= \beta_1 m_{n-1} + (1-\beta_1)\nabla Q_i(\omega_{n-1}) && (17)\\
v_n &:= \beta_2 v_{n-1} + (1-\beta_2)(\nabla Q_i(\omega_{n-1}))^2 && (18)\\
\hat{m} &:= \frac{m_n}{1-\beta_1^{n-1}} && (19)\\
\hat{v} &:= \frac{v_n}{1-\beta_2^{n-1}} && (20)\\
\omega_n &:= \omega_{n-1} - \eta\frac{\hat{m}}{\sqrt{\hat{v}}+\epsilon} && (21)
\end{aligned}
$$

where ϵ is a small number used to prevent division by zero, β_1 and β_2 are the forgetting factors of m_n and v_n respectively. The updates (19) and (20) above are meant to correct the exponential moving average because of the error that occurs for the first estimates when exponential moving averages are initialized [5].

4.7.4 The Backpropagation algorithm

The stochastic gradient descent method (or any gradient based optimization method) is based on the gradient of an objective function, which in the case of training of deep neural networks usually is a loss function. The Backpropagation algorithm is essentially a method to compute the gradient of the loss function, Q, of neural networks with respect to the weights and biases. Since the gradient is calculated with respect to the weights ω (and the biases b) the backpropagation algorithm calculates the partial derivatives $\frac{\partial Q}{\partial \omega^l_{jk}}$ and $\frac{\partial Q}{\partial b^l_k}$ where ω^l_{jk} is the weight on the connection between neuron j in the $(l-1)^{th}$ layer and neuron k in the lth layer and b^l_k is the bias for neuron k in the lth layer. In order to derive the backpropagation algorithm, let $Q(\omega)$ be a differentiable loss function (see section 4.7.2), σ_k be the activation function for neuron k in the lth layer and z_k the weighted sum that acts as input to that neuron. z_k can be defined as:

$$z_k = \sum_j \omega^l_{jk} o_j + b^l_k. \tag{22}$$

Where o_j are the outputs from the neurons in the $(l-1)^{th}$ layer. Note that if the lth layer is the first hidden layer o_j in equation 22 will be replaced by x_j which are the inputs to the network. The outputs from the neurons in the lth layer can be written:

$$o_k = \sigma(z_k) \tag{23}$$

Then, according to the chain rule the partial derivatives, $\frac{\partial Q}{\partial \omega^l_{jk}}$, can be expressed as:

$$\frac{\partial Q}{\partial \omega^l_{jk}} = \frac{\partial Q}{\partial z_k}\frac{\partial z_k}{\partial \omega_{jk}}. \tag{24}$$

The local errors, usually referred to as *deltas*, are defined as:

$$\delta_k \equiv \frac{\partial Q}{\partial z_k}. \tag{25}$$

Regarding equation 22, the second factor in equation 24 can be calculated as:

$$\frac{\partial z_k}{\partial \omega_{jk}} = o_j. \tag{26}$$

Thus, equation 24 can be rewritten as:

$$\frac{\partial Q}{\partial \omega_{jk}^l} = \delta_k o_j. \tag{27}$$

The outputs for neurons in the output layer are also outputs for the network and are denoted y_k. For output neurons δ can, by once again use of the chain rule, be calculated through:

$$\delta_k = \frac{\partial Q}{\partial z_k} = \frac{\partial Q}{\partial o_k}\frac{\partial o_k}{\partial z_k} = \frac{\partial Q}{\partial o_k}\sigma'(z_k). \tag{28}$$

After a forward pass through the network, both factors on the right hand side of equation 28 are known and therefore can all deltas for the output layer be calculated. Consequently, the derivatives in equation 24 can be calculated for weights connected to the output layer.

For inner nodes, when calculating δ, all the node's connections to the next layer have to be considered. Hence, delta can be written as:

$$\delta_j = \frac{\partial Q}{\partial z_j} = \sum_k \frac{\partial Q}{\partial z_k}\frac{\partial z_k}{\partial z_j} = \sum_k \delta_k \frac{\partial z_k}{\partial o_j}\frac{\partial o_j}{\partial z_j} = \sigma'(z_j)\sum_k \delta_k \omega_{jk}. \tag{29}$$

As mentioned above, all the deltas for the output layer are known after a forward pass. That is true for the other factors on the right hand side in equation 29 to. Accordingly, all the deltas for the layer preceding the output layer can be calculated and recursively the deltas for all neurons in the network can be calculated. Using equation 27 all partial derivatives can be calculated and thus the gradient. This propagation of errors (deltas) is the reason for the method being called backpropagation.

Similarly, the partial derivatives $\frac{\partial Q}{\partial b_k^l}$ can be expressed as:

$$\frac{\partial Q}{\partial b_k^l} = \frac{\partial Q}{\partial z_k}\frac{\partial z_k}{\partial b_k^l}. \tag{30}$$

By looking at equation 22 it's trivial to see that $\frac{\partial z_k}{\partial b_k^l} = 1$. Along with defining δ_k as in 25, equation 30 can be rewritten as:

$$\frac{\partial Q}{\partial b_k^l} = \delta_k. \tag{31}$$

The deltas are calculated as described above and consequently all partial derivatives with respect to the biases, b_k, can be calculated by formula 31 [1].

4.7.5 Initialization of network

One inevitable issue that has to be considered when an artificial neural network shall be trained is how the weights and biases should be initialized. Setting the initial values for the weights and biases

represents setting a starting point in the weight-(and bias) space. I.e. the topological space where the chosen SGD method is trying to find the optimal point in order to minimize the loss. This might seem like an unimportant issue since the optimization method will step away from the starting point anyway (unless in the very unlikely case that the initial point happens to be a minimum). This is indeed true but with thoughtful initialization the learning process may be substantially accelerated. To illustrate the problem, remember how the update of weights depends on the gradient (see section 4.7.3) and consequently the partial derivatives and how they are calculated (see equations 27 to 29). From those equations one can see that the derivative of the activation function for a neuron ($\sigma'(z)$) plays a crucial role in how the weights are updated. A small value on $\sigma'(z)$ leads to small values in the gradient which in turn leads to a small weight update (see equation 13). Thus, a neuron with a small partial derivative might learn extremely slow [6].

Sigmoid neurons Now, suppose you have an ANN with the sigmoid function (see section 4.2.2) as activation function. Regard one arbitrary neuron in the network. Suppose that you initialize the network weights such that the first input to the network causes the linear sum (equation 22) that acts as input to that particular neuron to have an absolute value much greater than 1 (neglect the bias for now, let's set it to zero). By looking at Figure 5 or equation (3) one can see that for such inputs to the sigmoid function, the derivative will be very small. Hence, referring to the discussion above, the particular neuron will learn very slowly.

As a remedy techniques have been developed for initialization to confine the input values to neurons within certain intervals. In order for a sigmoid neuron to learn fast it's optimal if the input to the neuron is close to 0, where the derivative is greatest. Consequently, the developed methods for initialization of the weights and biases have been focused on producing inputs to the neurons with mean zero and little chance of values with large magnitude. One way of achieving this is to try to make the inputs have a normal distribution with mean 0 and a low standard deviation [6].

One popular method of initialization is the one developed by Glorot and Bengio [18]. They suggest setting the bias to 0 and sample the initial weights either from a Gaussian distribution with 0 mean and $\sqrt{\frac{6}{n_{in}+n_{out}}}$ standard deviation. Or sample the weights from a uniform distribution defined as:

$$\omega_{ij} = U\Big[-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\Big]. \tag{32}$$

Where n_{in} is the number of neurons connected from the previous layer to the neuron in question and n_{out} the number of connections from the neuron in question to the next layer. U stands for uniform distribution.

ReLU neurons If ReLU is used as activation function on the other hand other concerns appear. As can be seen from Figure 6 and equation (4), ReLU doesn't suffer from small derivatives for large positive input values. Instead, for large positive input values the derivative is constant 1. On the other hand, for negative input values the derivative is constant 0. Which leads to so called "dead units" since they are not able to learn at all. He, Rang, Zhen and Sun suggest a modified initialization adapted for ReLU activation function [19]. As in Glorot and Bengio's method the bias is set to 0. The weights on the other hand is sampled from a normal distribution with mean 0 and standard deviation $\sqrt{\frac{2}{n_{in}}}$.

4.7.6 Training algorithm

The goal for the training of a network with backpropagation is to minimize some loss function. The learning algorithm can be described in the following steps

1. Set goal
2. Initialize
3. Forward pass an input and save all neuron outputs
4. Calculate loss. **If** goal/stopping criteria reached stop
5. Backpropagate errors
6. Calculate gradient
7. Update weights and biases
8. Repeat from step 3

4.7.7 Vanishing gradient problem

The vanishing gradient problem is a problem that can occur in deep neural networks and cause severe slowdown of the learning process. To understand the problem let's denote the learning speed of neuron j in layer l by δ_j^l. In other words the *delta* of a neuron defined in equation (25). That's a reasonable way of defining the learning speed of a neuron since a large δ leads to larger change for the parameters related to that neuron in the backpropagation algorithm.

First, note that in general does a low learning rate (small δ) not necessarily needs to be an unwanted feature. When a function is close to an optima, which is the ultimate goal, the gradient should be really small. In that case δ will also be small for all neurons. However, right after the initialization of the network or just in the beginning of the training it is very unlike that a minimum is found. Hence, if that is the case the reason for the small deltas is probably malicious.

The problem can be understood by looking at equation (29). For inner nodes δ is a product of the derivative of the activation function, σ', and a sum of products of δs and weights of succeeding layers. The δs in the sum are in their turn products in the same way and thus the sum can be expanded to a sum of products of $\sigma' s$, weights and $\frac{\partial Q}{\partial o_k}$. Where $\frac{\partial Q}{\partial o_k}$ comes from equation (28).

Now, if the weights are initialized as described in section 4.7.5 more or less all of them will fulfill $|\omega| < 1$. Suppose now that an activation function is used with a derivative that doesn't exceed 1 in magnitude for any point i.e. $|\sigma'| < 1$. This means that the condition $|\omega\sigma'| < 1$ will be fulfilled for basically all such products in equation (29). Now again look at equation (29). From these assumptions one can expect that δs in earlier layers will be smaller than in later layers. In other words, early layers will learn much slower than later layers. In severe cases when the network is very deep and $|\sigma'| << 1$ the decrease in learning rate for early layers compared to later can be exponential [6].

Exploding gradient problem In addition to the vanishing gradient problem an opposite phenomenon called *exploding gradient problem* can occur. The problem also occur in deep networks but usually in the case when an activation function is used with the property $|\sigma'| \geq 1$. In that case one can see that the condition $|\omega\sigma'| \geq 1$ might be fulfilled. Again, by looking at equation (29) one can see that δs in previous layers now will be greater in magnitude than δs in succeeding layers. Consequently, if this is the case for most adjacent layers in a deep network the magnitude may increase exponentially.

Choice of activation function As can be concluded from previous paragraphs the choice of activation function plays a crucial role in both the vanishing and exploding gradient problem. The vanishing gradient is a consequence of a small derivative of an activation function. A typical example of that is the sigmoid function (see section 4.2.2) which have a very small derivate for inputs with large magnitude. The vanishing gradient problem is essentially the problem described in section 4.7.5 but multiplied several times. One might think that a remedy to the problem would be to set large values on the weights. However, this is not as straightforward as it may look. Consider that the derivative of the activation function is a function of a weighted sum, i.e. $\sigma' = \sigma'(\omega o + b)$. If ω becomes very large, the risk is that σ' becomes small and we have the same problem again [6].

One remedy to the vanishing gradient problem is to use another activation function, for example ReLU (see section 4.2.3). When ReLU is utilized σ' is either 0 or 1. So for positive values the derivate is 1 and thus the local errors (δs) can be propagated backwards through the network without loosing magnitude. For negative values however the gradient instead disappears totally, which means that no learning can occur for those neurons. This problem is one reason for why activation functions like ELU (see section 4.2.4) was developed. ELU has the same property as ReLU in the positive region but have a small derivative in the negative region to avoid that the learning ceases completely.

On the other hand, networks with ReLU are much keener to the exploding gradient problem. If we ignore the cases when $\sigma' = 0$ the derivate of ReLU will always be equal to 1. Which means that the inequality $|\omega\sigma'| \geq 1$ will entirely depend on the weights and be equivalent to $|\omega| \geq 1$. From the inequality and once again equation (29) one can see that for large values on the weights, δs in early layers may become very large and even "explode". Here "explode" may connote that the value of δ exceeds the numerical limit for the computer and thus makes the whole training process futile.

4.7.8 Overfitting

Overfitting is a frequently occurring issue when networks with a large number of free parameters (weights and biases) are training. The problem in its essence derives from the fact that a mathematical model with a large number of parameters can model a huge range of phenomena very precisely, given enough free parameters to adjust. However, this does not mean that the model is good at generalizing. This only means that the model can describe a given data set very well. If given new unseen data, the model might work terribly. When this is the case, the model is said to overfit the data (or the model has overtrained). In Figure 10 such a situation is illustrated by the development of the error/loss (see section 4.7.2) on the training set and validation set during training progress. At the point when the error/loss on the validation set starts to increase whereas it continues to decrease on the training set, the network starts to overfit.

[6]http://dbigbear.blogspot.com.au/2007/11/overfitting.html

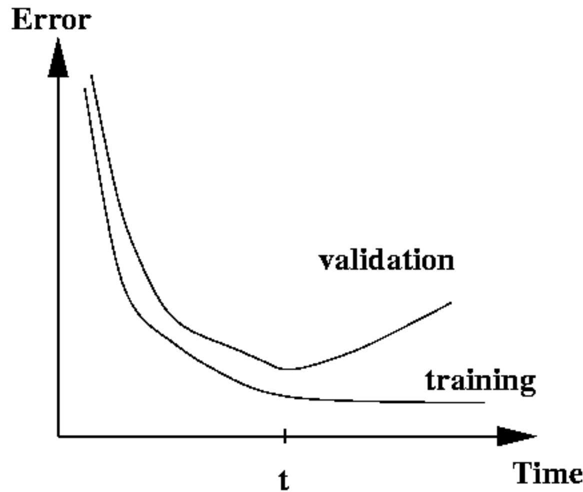

Figure 10: Illustration of the error/loss on the training set and validation set during progress of training. Note that both the training and validation error/loss decrease until time t. After that point the validation loss starts to increase whereas the training loss continues to decrease. This is a typical sign of overfitting.[6]

To further clarify the issue a mathematical analogy can be made with polynomial regression. Polynomial regression is a technique to fit a polynomial to a set of data points as closely as possible based on some kind of measure. For example the *least squares method* can be used as such (compare 4.7.2). Without presenting any proof, it is well known that a polynomial of degree n can perfectly fit $n + 1$ data points. For the cause of reasoning, only two dimensional points are considered in the analogy, still the idea can be extrapolated to higher dimensions. Now, consider that you have a set of noisy data points generated by a quadratic polynomial. The number of data points far outnumbers the number of the underlying polynomial (degree 2). Since the generated data points contain noise, a 2:nd order polynomial will not fit the points perfectly. In contrast, a higher order polynomial will be able to fit the data points better (with regards to the utilized loss measure) and a polynomial of high enough degree will be able to fit them perfectly. Hence, even though a 2:nd order polynomial would be the best approximation of the underlying function, a higher order polynomial will yield a better performance measure. In that case, the polynomial is overfitting the data. The phenomenon is illustrated in Figure 11 where the solid line overfits the data and the dashed line represents a 2:nd order polynomial. As can be seen in the figure the high order polynomial gets really close to some of the data point at both ends of the figure. Note however that in these regions the graph fluctuates heavily and if additional data points would be added in these regions the polynomial would most likely miss them by far. I.e. it would not be a good model of the underlying function.

As a complement, a more intuition based explanation more related to this project will be

[7]Safari Books online: https://www.safaribooksonline.com/library/view/hands-on-machine-learning/9781491962282/ch04.html

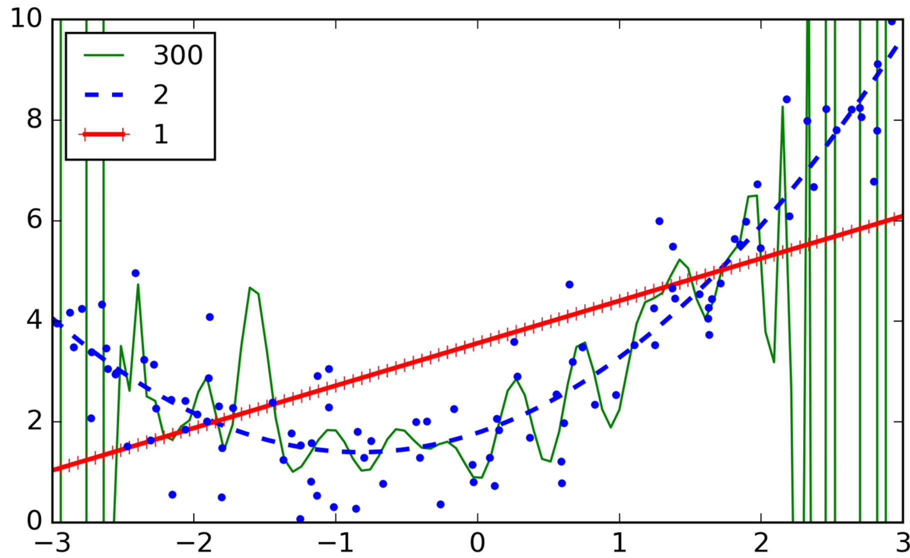

Figure 11: Illustration of overfitting and underfitting when performing polynomial regression on a set of data point.[7]

presented in this paragraph. Suppose we are trying to train a neural network to predict steering angles based on input images from the windshield. Our network consists of a huge number of parameters and the training set is relatively small. In a situation like this the network will be able to adjust its parameters for unique features in every image in the data set. Say for example that only one of the images in the training set contains a yellow trash can next to the road. That image will of course have a corresponding steering angle. In this case, when the network has a great number of parameters, it will be able to detect the yellow trash can and put out a steering angle very close to the correct one only based on the presence of the yellow trash can. Consequently, the loss will be very low even though the yellow trash can has nothing intrinsically to do with the steering angle. If the network then is exposed to a new image containing a yellow trash can (validation set), it may predict a totally wrong steering angle. If the network is large enough and the training data small enough this can basically be the case for all images in the training data. The situation is similar to the analogy with the kid learning multiplication in section 4.6.2.

Underfitting A not as prevalent, but worth to mention, problem when training neural networks is underfitting. Underfitting is the opposite situation to overfitting and occurs when the network doesn't have enough free parameters to model the underlying function. An example of underfitting can be seen in Figure 11. There, a polynomial of degree 1 (the straight line) is fitted to the data points generated by a 2:nd order polynomial. Clearly, the polynomial of degree 1 will be insufficient to model the underlying function.

4.7.9 Transfer Learning and domain adaption

The idea of transfer learning is to transfer "knowledge" between networks and use the acquired "knowledge" for a new task. The "knowledge" in this case is trained parameters. The discussion of the definition of knowledge is left for a more philosophical context. Supervised learning can be very effective for specific tasks, such as predicting steering angle for a car based on an input image. Yet, the performance of a network for that specific task is highly dependent on the data used for training and testing (see section 4.6). It is typically assumed that the training- and test/validation data is generated from the same underlying distribution. In many cases, a trained network will perform poorly if it is tested on a data set from another setting. Suppose for example that a steering angle predictor network is trained and tested in the conventional way on a data set consisting of images from driving on desert roads. Using techniques discussed in earlier sections the network will probably be able to steer a car on a desert road relatively accurate. Yet, if the same network would be allowed to steer a car on a mountain road it is likely that the result would be highly unsatisfactory. Even though the network might perform poorly on the mountain road there should be common features between the desert and mountain road that the network could utilize. For example should features like road curvature and edges between ground surfaces be useful in both environments. On a lower level, some features can even be useful for networks with very different tasks. For example could abilities such as detecting and localizing simple shapes be useful for both steering angle prediction and localizing a cat in an image. Transfer learning is a term for techniques that try to transfer abilities in networks that can be useful for other networks. If such transfers are successful a lot of training can be avoided and the network might be more general [31].

Definition In this section a more technical definition of transfer learning will be presented. First, two concepts will be presented: *domain* and *task*. These are concepts involved in all supervised learning problems.

A *domain* consists of a feature space $\mathcal{X}$ and a marginal probability distribution $P(X)$ over the feature space. Where X is a sample of examples from $\mathcal{X}$ and $X = x_1, x_2, ...x_n \in \mathcal{X}$. In the settings of this project $\mathcal{X}$ would be all possible input images that a self-driving car could face and X would be the examples constituting the data used in the training process. The domain is then defined as the tuple $\mathcal{D} = \{\mathcal{X}, P(X)\}$.

A *task* is also defined as a tuple. A task requires a *domain* and consists of a label space $\mathcal{Y}$ and a conditional probability distribution $P(Y|X)$, where X is the same X as in the paragraph above and $Y = y_1, y_2, ...y_n \in \mathcal{Y}$. The conditional probability distribution $P(Y|X)$ is usually what the network learns during training. $\mathcal{Y}$ is the set of all possible labels for the task and would in this case be a continuous spectrum between the maximum steering angles the actual car can execute.

Now, denote a source domain $\mathcal{D}_s$, a source task $\mathcal{T}_s$, a target domain $\mathcal{D}_t$ and a target task $\mathcal{T}_t$. The objective for the transfer learning is to make it possible to learn the target conditional probability distribution $P(Y_t|X_t)$ in $\mathcal{D}_t$ with the information gained from $\mathcal{D}_s$ and $\mathcal{T}_s$. Here it is assumed that the domains and tasks differ, i.e. $\mathcal{D}_s \neq \mathcal{D}_t$ and $\mathcal{T}_s \neq \mathcal{T}_t$ [31].

There are different scenarios for transfer learning since the source and target conditions can vary in several ways. The case that will be discussed here is also known as *domain adaption* [32] and is the case when the source and target marginal probability distributions differ, i.e. $P(X_s) \neq P(X_t)$. This is the case for the example mentioned in the introduction of this section with the desert and mountain roads. Further reading on other scenarios can be found in [31].

Using pre-trained feature extractors One popular technique is to use parts of a trained network that is known to perform well on a task for a different task. For vision tasks (tasks where the network make a decision based on an input image) abilities like edge detection and localization of shapes are usually useful regardless of the output of the network. Convolutional neural networks are extensively used in vision task contexts and are relatively well known. As mentioned in section 4.4 the convolutional layers are usually considered feature extractors and the fully connected layers the classification/decision layers, even if a clear distinction is not possible to make. Anyway, it is generally known that the early convolutional layers extract basic features (like edges etc.) and following layers extract more and more complex features. Hence, the idea is to take an appropriate part of a well performing CNN and use it as a part of another network. At some level in the well performing network, the convolutional layers should still extract useful features for the new network.

The idea is that by freezing the copied part of the well performing CNN, i.e. not allowing the parameters to change, and then train the rest of the network (typically the classification/decision part) the network will be forced to make decisions based on image features. Contrary to letting it base them on features that are only present in the feature space from the source domain.

4.7.10 Multi-task learning

Another method to improve network performance is called *Multi-task learning*. The idea is to train the network on more than one task simultaneously, even if the ultimate goal still is to maximize the performance on one task. The method is mainly a way to avoid overfitting and make the network more general. The idea can be implemented in many ways but the most common one is called *hard parameter sharing*.

In hard parameter sharing, layers (and their parameters) are shared between different networks with different tasks. It can be seen as one large network that consists of connected sub-networks. For every task there is a specific sub-network. In addition, all or some of the task specific sub-networks are connected to a common sub-network [33]. An example of such a network is illustrated in Figure 12.

In the learning process, the parameters in the task specific sub-networks are trained and tweaked with respect to the performance measures for the individual tasks. Whereas the parameters in the common network are trained with respect to some aggregate of the individual tasks.

Biologically, motivation for the method can be found in the way humans learn. Humans are constantly exposed to a lot of impressions and information and even if we are just focused on one specific task we still process seemingly irrelevant information. Information that might be helpful in the future or in other settings, even though we don't realize it at the moment. Suppose for example that a baby is trying to learn how to recognize cats by looking at photographs of cats, dogs, horses, cars, bikes and apples. The most direct approach would be to just "label" every image with a yes or no (binary classification) whether it is a cat or not. The multi-task approach would be to for example also label every image if it is an animal or not. Then the baby would also acquire some knowledge about animals. If the baby later is exposed to an image of a cat breed he/she hasn't seen before, it is likely that he/she will not classify it as a cat but at least will be able to say that it's an animal. With the first approach he/she will only be able to say that it's not a cat (even though it is).

[8]https://arxiv.org/pdf/1706.05098.pdf, page 2.

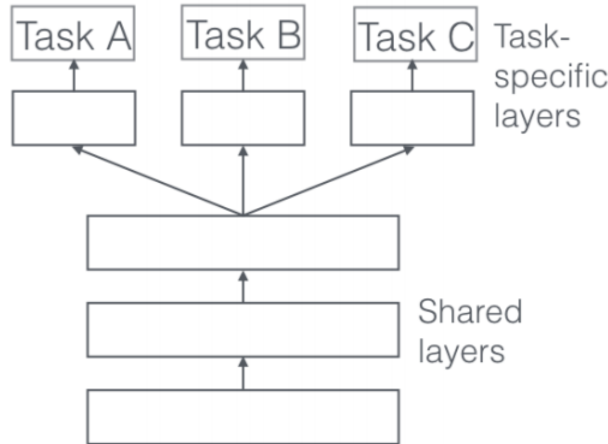

Figure 12: Illustration of a network with hard parameter sharing. The network has three tasks and thus three task-specific sub-networks (or task-specific layers as in the figure) and one common sub-network (shared layers in the figure).[8]

4.8 Regularization

Regularization is the common name for different techniques to counteract overfitting (see section 4.7.8). There are many developed regularization techniques and one can also discuss which methods that should be called regularization techniques. For example, data augmentation, explained in section 4.6.4, can indeed be seen as a regularization technique even though it wont be described in this section.

Referring to section 4.7.8, the fundamental cause for overfitting is the relative amount of parameters in the network in relation to the available training data. Often, it is not possible or very expensive to collect more data. Consequently, regularization techniques are developed under that presumption. As a first thought one might consider simply decreasing the size of the network. That is a reasonable approach and might work for simpler tasks. However, works like [7] and [10] have shown that deep networks outperforms shallow ones. In addition, the human brain consists of about 86 billion neurons. That is equivalent to an artificial neural network with a huge number of parameters. Yet, the human brain is an extremely powerful learner and doesn't seem to suffer from overfitting. So it seems indubitably that larger networks are capable of better performance. Hence, decreasing the size of the network might not be a preferred regularization measure.

In the succeeding sections three popular and well documented regularization techniques will be presented. Namely, *early stopping*, *L2-regularization* and *dropout*.

4.8.1 Early stopping

Early stopping is a technique that can be seen as a way of avoiding overfitting in an existing network design by preventing the network to train for too long. In contrast to the two other techniques below, early stopping is not related to any hyperparameter (hyperparameters are described in section 4.10)

that changes the design of the network. So if the goal is to find a better design of some network, early stopping is usually combined/followed by some change of hyperparameter.

The method is based on the validation loss and can be explained together with Figure 10. Since the loss (error in Figure 10) on the validation set is the measure for how well the network is generalizing, the idea is that there is no or little reason to continue training if the validation loss is not improving. Especially if the training loss is still improving, because then overfitting is most likely occurring. Hence, early stopping would typically stop the training at time t in Figure 10.

Early stopping might sound easy but the part that can be difficult is to decide when the validation loss is not improving any longer, i.e. when the training should stop. In real situations the curves are rarely as smooth and clear as in Figure 10, where it is obvious where the validation loss stops improving. The validation loss might fluctuate between epochs even though the general trend is that it decreases. So if early stopping is activated whenever the validation loss after an epoch not is lower than the previous, it might be too early. The network might still improve given some more epochs of training [6]. Usually one have to define some parameters such as *patience* and *tolerance*. Patience defines for how many epochs the network should be allowed to train without improvement in the validation loss before early stopping is executed. Tolerance defines some kind of difference between validation losses that has to be exceeded between subsequent epochs in order to be considered as an improvement.

4.8.2 L2-regularization

A popular regularization technique is the *L2-regularization* method or *weight decay* as it is also called. It is not obvious why the technique works but an intuitional explanation will be given in this paragraph. The technique is based on the idea that small weights are preferable over large weights in a neural network. This is in order to prevent the network from basing its output too much on single input features and ultimately inhibit overfitting. If the weights are kept small, the network is more prone to make its decisions (outputs) based on several input features/values. If the weights are allowed to be large, the network is more sensitive to noise and unimportant features [6]. Once again the example with the yellow trash can from section 4.7.8 can be used. In that case the yellow trash can can be regarded as noise since it is not important for the steering angle. If the network is allowed to have large weights it may set its weights such that when a yellow trash can occurs in the input, that feature totally dominates the output of the network and other (more important features such as the road curvature) will basically be ignored. On the other hand, if the weights are restricted the network won't be able to base its decision solely on the presence of the yellow trash can. Instead the decision will be based on an aggregate of input features. Small weights also makes the network less sensitive to minor unimportant fluctuations in the input and instead look at the "bigger picture", to use layman terms.

Mathematically, the L2-regularization is implemented by adding an extra term to the cost function used in the training process (see section 4.7.2). If the quadratic loss function is used for example (equation (11)), then the new expression for the loss will be

$$Q(\omega) = \frac{1}{2n}\sum_{x} ||t(x) - y(x)||^2 + \frac{\lambda}{2n}\sum_{\omega} \omega^2. \tag{33}$$

Or, since L2-regularization works regardless of which loss function is used a more general way to express it would be

$$Q(\omega) = C_0 + \frac{\lambda}{2n} \sum_{\omega} \omega^2. \tag{34}$$

Where C_0 is the original unregularized loss function. λ is a coefficient called the *regularization parameter* and has the property $\lambda > 0$ [6]. Before delving deeper into the effects of λ let's clarify what the name *L2-regularization* comes from. L2 comes from the L2-norm and is linked to the fact that the extra term in equation (34) consists of a sum of squared weights. Very much like the L2-norm of the weight vector, except the square root. There is also a *L1-regularization* method which works much in the same way but instead of adding a sum of squares to the original cost function it adds a sum of absolute values. I.e. the terms ω^2 in equation (34) are replaced by $|\omega|$. The *L1-regularization* method will not be presented here but for further reading see [6].

Now, back to the regularization parameter λ. Since the backpropagation algorithm now works with equation (34) instead of the original one, λ will play an important role in how the weights are adjusted. λ can be seen as a factor of the relative importance between having small weights and minimizing the original loss function. For large λ small weights are prioritized whereas small λ encourage minimizing the original loss function. Technically, the difference in the backpropagation algorithm when L2-regularization is used can be found in the partial derivatives $\frac{\partial Q}{\partial \omega}$. The partial derivatives $\frac{\partial Q}{\partial b}$ will be unchanged. Again, denote the original loss function C_0. Then the partial derivative of the loss function with respect to the weights becomes

$$\frac{\partial Q}{\partial \omega} = \frac{\partial Q_0}{\partial \omega} + \frac{\lambda}{n}\omega. \tag{35}$$

The expression for $\frac{\partial Q_0}{\partial \omega}$ can be found in equation (27). So, the L2-regularization doesn't provide any major added complication to the backpropagation algorithm and thus doesn't stop the training from being effectively executed by the backpropagation algorithm.

4.8.3 Dropout

Dropout has in recent years become a widely utilized regularization method. The method in its essence is to randomly "drop" nodes (which means temporarily inactivate) from the network during training. The fundamental idea behind it is that by doing so one prevents the nodes in the network to co-adapt to much. Which means that the nodes in the network becomes more independent and don't rely too much on other nodes. This will ultimately inhibit overfitting since nodes that are not too dependent on other nodes will be less prone to propagate unimportant features and noise through the network.

According to Strivastava et al. [25] the best way to "regularize" a fixed-sized network would be to combine the predictions from all possible parameter settings of the network. However, for large networks this becomes infeasible since the number of calculations becomes too big. Instead, they suggest averaging the predictions for a number of smaller "thinned" networks that shares parameters. In general, combining predictions from different models improves the result. However, training several different networks usually requires much computations and large data sets. The dropout method is a way to do an approximation of training different networks and combining them.

Technically the dropout method works as follows. Every node is assigned a probability p. p is the probability that the node will be retained in the network during one forward-backward pass in the training process. If a node is dropped, all its connections and corresponding weights are

also dropped and thus not part of the training. The weight update for the dropped weights are thus 0. Or equivalently, the elements in the gradient corresponding to those weights are 0. So for every training example (forward-backward pass) a "thinned" network is trained. See Figure 13. If batches are used this procedure is done for every example in a batch. So a different "thinned" network is trained for every example in the batch [25].

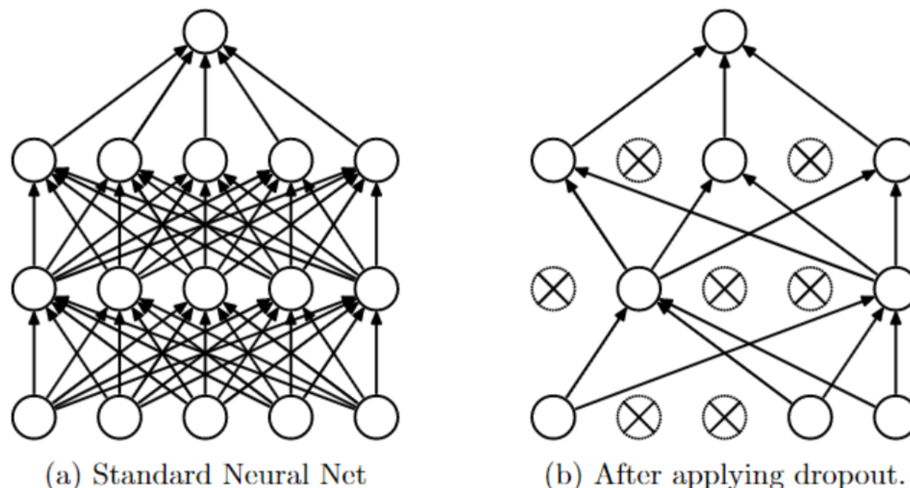

Figure 13: **Left:** A standard neural network with 2 hidden layers. **Right:** An example of a "thinned" network produced by applying dropout to the network on the left. Crossed units have been dropped.[9]

Then, the update for every weight in the full network is averaged over the batch. After the parameter update the same procedure is repeated with a new batch. In that way the training goes on. When it comes to testing you typically want only one network to test on. Thus, an average over multiple "thinned" networks can't be used. Instead, the test is performed on a scaled-down version of the trained full network. The scaling down works in such a way that all nodes are active (i.e. no nodes are dropped) and instead all weights are multiplied with the p corresponding to the node they are connected to. This is in order for the nodes to have the same output at testing time as the expected output at training time [25]. Also see Figure 14.

Networks using dropout have been shown to outperform unregularized networks, but also networks regularized with L2-regularization [25]. Yet, there is nothing that prevents utilization of several regularization techniques and a combination may be the best option.

4.9 Batch Normalization

Batch Normalization is a method that was introduced by Ioffe et al. [8] to speed up the training process of a neural network. The method is intended to prevent *covariate shift* between layers,

[9]Srivastava, N. Hinton, GE. Krizhevsky, A. Sutskever, I. 2014 *Dropout: A Simple Way To Prevent Neural Networks from Overfitting.* In *Journal of Machine Learning Research 15*, pages 1929-1958.

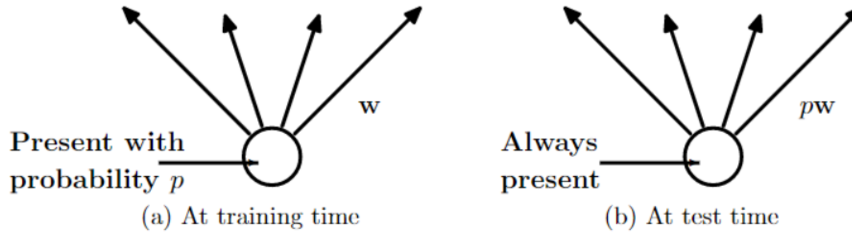

Figure 14: **Left:** A node at training time that is present with probability p and is connected to nodes in the next layer with weights $\mathbf{w}$. **Right:** At test time, the node is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time. [10]

which the authors call *internal covariate shift*. By doing this they show that the training process can be accelerated.

Covariate shift When the input distribution to a learning system changes, while the distribution of the output shall remain constant, it is said to experience covariate shift [8]. This is a problem since the learning system has to keep adjusting to new input distributions. The problem is not exclusively for the whole system but can be applied to subsystems or subnetworks. Even one layer can be seen as a subsystem of the network. When layers in the network experience covariate shift from the previous layer it is called internal covariate shift. Internal covariate shift can be a noteworthy problem in deep networks since the weight update in early layers through training can lead to changed distribution of the input to succeeding layers. The reason that it is a problem is that it slows down learning. This is due to saturating nodes much like the vanishing gradient problem (see section 4.7.7). Because, if a layer experiences covariate shift the inputs to the layer will not have the same distribution as the layer is trained for. Consequently, the inputs to its activation function may end up in the regions where the derivative is close to zero. Which, aforementioned, leads to slow learning.

Method The solution that Ioffe et al. suggest is to normalize the inputs to every layer (or a subset of layers depending on the design of the network) based on the batch. Hence, the name *Batch Normalization*. So, when the network trains a layer with batch normalization the following actions are taken. The activations from the previous layer (outputs, denoted o in section 4.7.4) are *whitened* [17] along every dimension. To make it clearer suppose that the batch size is m and denote a specific activation in the network o_i. Then the whitening will be done based on the resulting activations from the current batch $o_i^1, o_i^2, ..., o_i^m$. This is done for all activations and for all batches. In addition, to avoid that the layer loses representational power the whitened activations go through a linear transformation. If the linear transformation is ignored the layer might start working just as a linear layer and loose its nonlinear properties. The parameters constituting the linear transformation work much like weights and biases. Hence, when using batch normalization those new parameters will be trained too [8].

The meaning of the training though is to result in a deterministic network. Thus, the network

must be able to make a decision (output) for a single input and not be dependent on other values from a batch. So, to accommodate for this the network during testing differs from during training. When it is time for testing the whitening in the normalization process is not based on current values in a batch, but instead based on constants calculated from a large set of the training data (or ideally the entire training set). In a nutshell it works like this. During training all examples of a batch is propagated through the first batch normalized layer. The mean and variance of the outputs from every example along every dimension is calculated and used to whiten the outputs. Then the whitened outputs are linearly transformed and sent to the next layer. During testing on the contrary, a large subset of the training data is sent trough the trained network and outputs for the batch normalized layers are saved. Based on these saved outputs the mean and variance for all outputs along all dimensions are calculated. These calculated means and variances are then set as constants in the network. Now, a deterministic network has been created. If the network gets an input now it will simply normalize the outputs in every batch normalization layer by using the constants as mean and variance.

It is clear that batch normalization adds substantial computations to the training process. Consequently, the execution time for an epoch of training will most likely increase. Nevertheless, the idea is that the method will speed up learning by reducing the number of epochs needed to reach a certain performance level. So, in total the training time will ultimately decrease.

4.10 Hyperparameters

Hyperparameters are, unlike the free parameters, parameters that have to be set before the training starts and are not affected by the training. Here, free parameters are the ones that are adjusted by the backpropagation algorithm (weights and biases) and are just called parameters. An analogy would be the training of an endurance athlete. The structure of the training program would be based on hyperparameters such as distance per week, time in different heart rate zones, hours of strength training per week etc. The parameters on the other hand would be (hopefully) affected by the training and could be for example the heart's stroke volume, aerobic capacity etc.

Optimization Optimization of hyperparameters is a cumbersome and more time consuming process than optimization of parameters. This is due to the fact that there is no clear function from hyperparameter to cost, so a gradient based method can not be used as in parameter optimization. Such a function would include the training itself and thus one forward pass would include a full training process. Therefore, the extent of hyperparameter optimization methods and theory is not as ample as parameter optimization. Two possible methods that can be used are grid search and random search. In these cases different parameters are tested either in a structured way or a random way respectively. Both cases require some set interval of the hyperparameter values though. So, still some kind of knowledge of which hyperparameter values that are reasonable are required. These reasonable values are usually based on previous work and empirical knowledge.

As aforementioned, the process can be very cumbersome. If for example grid search is performed, every extra hyperparameter that is added to the search adds one extra dimension to the grid. Consider that in order to test one setup of hyperparameters, an entire training process has to be conducted. So to test just a few parameter setups can be a very time consuming process. Below, some of the possible hyperparameters involved in the training of a neural network are presented and explained.

4.10.1 Learning rate

One of the most fundamental hyperparameters is the learning rate, presented in section 4.7.3. As mentioned there the learning rate can be seen as a step-size for the gradient descent method. The methods presented in that section are all developed to minimize problems due to a poor learning rate setting. Despite that, a reasonable learning rate is crucial for the convergence of the backpropagation algorithm to a minimum. A too small learning rate can lead to extremely slow convergence. In the other end, a too large learning rate may lead to no convergence or even divergence. This is due to the nature of the stochastic gradient descent. A too large learning rate may cause the minimizing process to jump away from an advantageous point close to a minimum to a point far away from the minimum. Note though that this can occasionally be a good feature to, in the case that the minimizing process jumps out of a local minimum to a global minimum.

4.10.2 Batch size

If stochastic gradient descent is utilized, the batch size is another hyperparameter to consider. The two extremes of the batch size are of course the entire training set and a single example. Meaning that the gradient is updated based on the entire set or after just one example respectively. The most common batch size is something in between the extremes. If one uses the entire training set, the method is guaranteed to converge to the global minimum if the error surface is convex and to a local minimum if the surface is non-convex [26]. Also, a large batch size is more likely to represent the underlying function well than a small one. Which makes the gradient update more likely to be in the correct direction. However, since the method uses the entire training set for every gradient update the method can be very slow. Especially if the training set is large.

At the other end of the spectrum, the gradient is updated after a single example in the training set. On the positive side, this usually entails very fast updates. On the downside, it may lead to heavy fluctuations in the loss. When only one example is used as an approximation of the underlying function, the gradient is more likely to be updated in the wrong direction. Hence the fluctuating loss. In addition, there is no guarantee of convergence to a minimum since the method can keep jumping between minima. Once again though, it is not necessarily an unwanted feature that the method jumps out of minima [26].

As aforementioned, the preferred choice of batch size is usually something in between the extremes. That's usually called a *mini-batch*. Mini-batches take advantage of both extremes. In addition, mini-batches also take advantage of highly optimized matrix multiplication algorithms implemented in most modern deep learning libraries [26]. The size of the mini-batches varies for different applications. Sometimes when using parallel computing the mini-batch size is adjusted to the number of processors working in parallel to optimize the computations.

4.10.3 Epochs

The classical definition of an epoch is one training cycle over the entire training set. I.e. the network sees every example in the training set once and updates its parameters accordingly. One epoch is independent of batch size and if the data is shuffled etc. However, in some training procedures one epoch can not longer be as stringently defined as above. For example, if data augmentation (see section 4.6.4) is used in real-time it is not clear what an epoch is. Real-time in this case means that for every example during training, augmentation is performed before the example is sent to the network. Which means that even if the training set consists of n examples, in reality the network

can be fed with many more examples. Call for example an image in the training set x_i, and say that real-time data augmentation is utilized. Moreover, the training process simply goes through the training set in numerical order. In this case the first time x_i is sent in to the network perhaps horizontal flipping is performed. Next time it is sent in, the brightness is changed etc. Thus, even though the same training example from the training set has been sent in twice to the network, technically the network has not seen two identical inputs. In situations like this, one epoch is usually defined just as a number of training examples.

Anyway, regardless of data handling it is debatable if epoch even is a hyperparameter. If the explanation that was given in the beginning of the hyperparameter section (see section 4.10) is considered one might not regard the number of epochs as a hyperparameter. Mainly because it doesn't always have to be set before the training starts. This is the case in the widely used early stopping technique (see section 4.8.1) where the number of epochs depends on the result from the loss on the validation set. Note here that even if the number of epochs in this case indirectly depends in the training process, it can not be considered a parameter.

Nevertheless, in situations when different networks should be compared on different measures the number of epochs are sometimes pre-set to a constant number to accommodate for equality between trials. It can hence be seen as a hyperparameter.

4.10.4 Regularization parameter

The *regularization parameter*, or λ, was introduced in section 4.8.2 and its role in the training was explained. As described there, the regularization parameter represents a trade-off between minimizing the unregularized loss function and having small weights. As in the case for most hyperparameters the knowledge about what values that are reasonable and works mainly comes from testing and experience. What can be said though is that the regularization parameter should be scaled to account for the mini-batch size. Since the regularization sum is multiplied by $\frac{\lambda}{2n}$ (see equation (34)), the size of λ should be changed to keep the same quota if n is changed. This is done in order to have the same regularization effect (weight decay) regardless of mini-batch size n [27].

4.10.5 Dropout rate

Another hyperparameter that can be tuned is the dropout rate, p, introduced in section 4.8.3. The authors of [25] suggest $p = 0.8$ for input layers when the input is real-valued and $p = 0.5 - 0.8$ for hidden layers. The dropout rate is related to the size of the network. Since a dropout layer with dropout rate p and n neurons effectively has np neurons, the authors suggests keeping the product np constant if n neurons works satisfactory for an unregularized network. What's worth knowing is that large dropout rates have small regularization effects. Whereas small dropout rates induce heavier regularization, but can make the network less powerful and substantially increase the training time [25].

Note that confusion may occur since it is common in some settings to call the opposite of p the dropout rate. I.e. the probability that the node is dropped. This is for example the case in the machine learning package *Keras*, that is used in the project and introduced later in the report.

4.11 Performance measures

To evaluate a neural network's ability some kind of performance measure has to be defined and used in order to quantify it. The performance measure has to be relevant for the network's assigned task

and can differ from the loss (see section 4.7.2). The loss must for example have some properties to work in the backpropagation algorithm, which is not necessary for a performance measure. The loss is a function to control the training process, whereas a performance measure usually more accurately reflects how well the network is at its intended task. Nevertheless, this doesn't exclude the possibility that the loss can be used as a performance measure to. Many performance measures exist in the field of deep neural network training. Naturally, this section will be confined to measures relevant to image processing.

4.11.1 Loss

Loss (or sometimes referred to as cost or error) is described in section 4.7.2. The advantages with loss functions are that they are differentiable functions and that they can be used for *regression* problems [29]. This is for example the case for the networks developed in this project. Networks that based on an input image shall output a steering angle. Here, there are no obvious classes that the outputs can be divided into.

4.11.2 Accuracy

Accuracy is a very common performance measure in for example image classification tasks. In classification tasks the input image is labeled with a category and the network is supposed to label it correctly. A typical example would be that there are three categories; cats, dogs and rabbits represented in a data set. Given one input image, the network is supposed to sort the image into the right category. The accuracy for a given set is defined as the quota between the number of correct labeled images and the total number of images in the set, i.e. $accuracy = \frac{\#correct}{\#total}$. As mentioned in the paragraph above, accuracy might not be a suitable performance measure when no natural classes exist. It is up to the designer to define any eventual classes in these cases.

Furthermore, when the data is unbalanced accuracy might not be an adequate performance measure. Suppose for example a binary classification problem, i.e. only two classes exist to sort the data in to. Now, say that 990 out of 1000 images in the data set belong to class 1 and 10 images to class 2. Then if the network consistently labels all inputs as class 1, the accuracy will be 0.99, which in most cases is considered a very high accuracy. This despite that the there is no actual dependence on the input images. This is of course a very extreme example but the problem still exists. As a remedy several other measurements have been developed. Among others the *confusion matrix* which is described below.

4.11.3 Confusion matrix

A confusion matrix is a more visual performance measure than the previous mentioned ones. Consequently it can reveal aspects of a network's performance that might not be disclosed by for example the loss or accuracy. Like the accuracy it requires that the output can be sorted into classes though, and it is frequently used in classification tasks. The confusion matrix consists of rows that represent the instances in a predicted class and columns that represent instances in a actual class [28]. An example of a confusion matrix can be seen in Figure 15, where once again the three classes cats, dogs and rabbits exist.

All correct predictions are located in the diagonal of the matrix, and thus an optimal outcome is a diagonal matrix. In the example illustrated in Figure 15 however, the model has for example labeled 3 out of 8 cats as dogs.

Confusion Matrix		Actual Class		
		Cat	Dog	Rabbit
Predicted Class	Cat	5	2	0
	Dog	3	3	2
	Rabbit	0	1	11

Figure 15: Example of a confusion matrix with three classes. A total of 27 animals have been labeled (predicted) in to the three classes. The sum of each column represents the number of actual animals in that class in the set. The sum of each row represents the number of animals in that class predicted by the model.

Confusion matrices can be useful in regression problems to, despite that it requires classes. Again, take for example the networks in this project. They are supposed to predict a steering angle based on an input image. There are no obvious classes to divide the output in. Yet, if the steering angles are sorted into for example the nearest integer a confusion matrix can be helpful. If for example the network predicts a steering angle of 14° and the actual steering angle is 15° the accuracy will say that the network was "wrong". Even though, in reality, that is a pretty good prediction. On the other hand, a confusion matrix will in this situation increase a number in a sub (or super) diagonal cell. After an entire set is predicted the confusion matrix might have a lot of high numbers in the sub and super diagonal. Which will indicate that the network is actually performing pretty well, just slightly of the actual steering angles. If the classes are narrow, this might even be negligible. Whereas the accuracy in this situation might be really low and indicate poor performance.

In addition, the confusion matrix might reveal if the network is biased in any way. If for example the network tends to predict steering angles that are slightly to the left of the actual ones, this will be apparent in the confusion matrix. Either the upper right or lower left part of the matrix (depending of how the classes are represented in the matrix) will in this case have cells with higher numbers than on the other side of the diagonal. In comparison, such an unwanted performance feature will not be exposed by the loss or the accuracy.

5 Implementation

5.1 Data

5.1.1 Artificial data

In the project two different underlying data sets were used. The first one contained images created from the *Unity* car simulator in an artificial environment. The data was collected by manually steering a car in the simulator and simultaneously saving the images representing the windshield view along with the corresponding steering angle. There were three cameras recording simultaneously, one in the center, one to the left and one to the right. The corresponding steering angles were in this case with respect to the center images. Examples of images from the three cameras

can be seen in Figure 16. This data set will in this report be referred to as the "artificial data set".

(a) Left camera (b) Center camera (c) Right camera

Figure 16: Examples of images from the artificial data set

The artificial data set consisted of 3767 unique images if left, center and right images (like the three examples in Figure 16) all are considered unique. The size of the data set was about 53 MB.

5.1.2 Real-life data

The second data set used consisted of images from real life driving. The data was collected from a camera mounted in a car driving mostly on highways. It was collected by *comma.ai* and is published under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License [36]. Which implies that also the work in this report is published under the same license. An example from the data set can be seen in Figure 17. This data set will in the report be referred to as the "commaai data set".

Figure 17: Example of image from the commaai data set

The commaai data set consisted of 7 and a quarter hours of driving in total, which adds up to about 80GB.

5.1.3 Manipulation of data

For the artificial data set some manipulation was required. Since only the steering angles corresponding to the center images were available in the log, steering angles for the left and right images

had to be properly added. This was done by adding 5°to the steering angles for the left images and removing 5°to the right. All in order to account for the laterally shifted position of the car. The choice of 5°was based on the length to the horizon and the maximum speed of the car in the simulator.

The original artificial data set was biased towards left turns. As a remedy, all images were flipped and copied which in addition to making the data set unbiased made it twice as large. Also, oversampling was utilized for underrepresented steering angles. There are no strict classes when it comes to the steering angles, but when histograms over the distribution of steering angles in the data set was looked at it could be established that a majority of the steering angles were really small in magnitude. I.e. corresponding to driving straight.

In training, the images were cropped to remove unnecessary part of the image such as the sky and the hood of the car. In addition, the images were resized to 66×200 pixels from originally 160×300. These actions greatly reduced the input size to the network which ultimately decreases the number of parameters. Also, during training the images were converted from RGB to YUV format. This is done in the work by *NVIDIA* [15], who accomplished great results with their network.

Several modifications were made to the commaai data set in order to improve performance of the network. First, like the artificial network the images were cropped and resized to 66×200 for the same reasons. Likewise, they were also converted to YUV. Second, the commaai data contained a lot of examples when the car stood still, it could be due to traffic or an intersection etc. That is relevant data in a more large-scale setting, if the purpose is to train the car not only to predict steering angles but also break and accelerate for example. However, for this project such data is not beneficial for the purpose and was thus removed. Third, the data had some noise when it came to the steering angles probably due to measuring error when collecting the data. This was apparent when steering angles with magnitudes larger than 90°were present in the set. Consequently, steering angles with magnitude larger than 57°were removed. It is reasonable to assume that a car can't turn that sharply. A conversion had to be made since the steering angles given in the data set was with respect to the position of the steering wheel, not the actual steering angle of the vehicle.

The modification mentioned above could be done automatically in different script given the information in the data set. However, some modifications were also made manually. In this setting that means that some sorting out was made by looking at data and discarding it. For example the original data set contained images from when the car stood in the garage, which is not helpful for the project (or rarely in a more large-scale setting either). Thus, such data was removed from the set.

For the purpose of task 3 (see section 6.4) some modifications had to be made to make the two data sets compatible. Specifically, to make the simulator run a network trained on the commaai data set, adjustments had to be made. As mentioned above, resizing of the input images to the same size was of course necessary. Then, the steering angles in the commaai data set had to be normalized in the same way as in the artificial set. The normalization was done based on 25°, i.e. a 25°left turn in the simulator was represented by an output of -1 in the simulator. In addition, to further benefit the learning process in task 3 adequate data was sorted out manually of the commaai data set. This was done by removing images that contained, for the purpose, too much information that could possibly stall the learning process. Example of such data is city driving in heavy traffic, multiple lane highway driving, intersections etc. Since the car simulator is executed in a country road setting with no other traffic (single lane road only and no intersections) such data is irrelevant. In addition to the data being irrelevant, it's much easier to work with a smaller data

set.

One of the main problems with task 3 is the different distributions of the training and validation data. As an approach to mitigate this issue the distributions of the pixel values for different color formats and channels were compared for the two sets. The idea was to find and use color formats and channels were the distributions of the two sets overlapped the most. An $66 \times 200 \times 3$ image is a rather high dimension input and if the comparison should be done properly all pixels should be compared individually between the sets. This is a very cumbersome process so instead all pixels in all images where plotted in the same histogram and compared between the sets.

Over- and undersampling (see section 4.6.3) was heavily utilized on the commaai data set. For example, in the original commaai data set a large part of the steering angles were in the interval -0.5°to 0.5°. Even after the aforementioned sorting the distribution looked like in Figure 18.

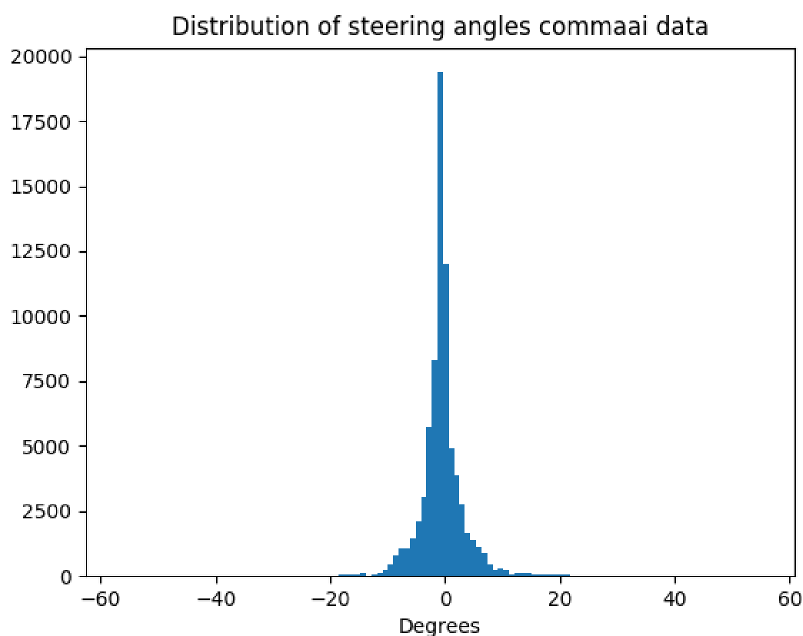

Figure 18: Distribution of steering angles in the commaai data set after sorting.

As a remedy, steering angles close to 0 was undersampled and other angles oversampled. After the over- and undersampling the distribution of steering angles was close to uniform between -25°and 25°and about one fourth of the amount for angles with larger magnitudes. Additionally, the data was augmented by horizontal flipping, translation, shadowing and brightness (see section 4.6.4). After the modifications the distribution looked like in Figure 19

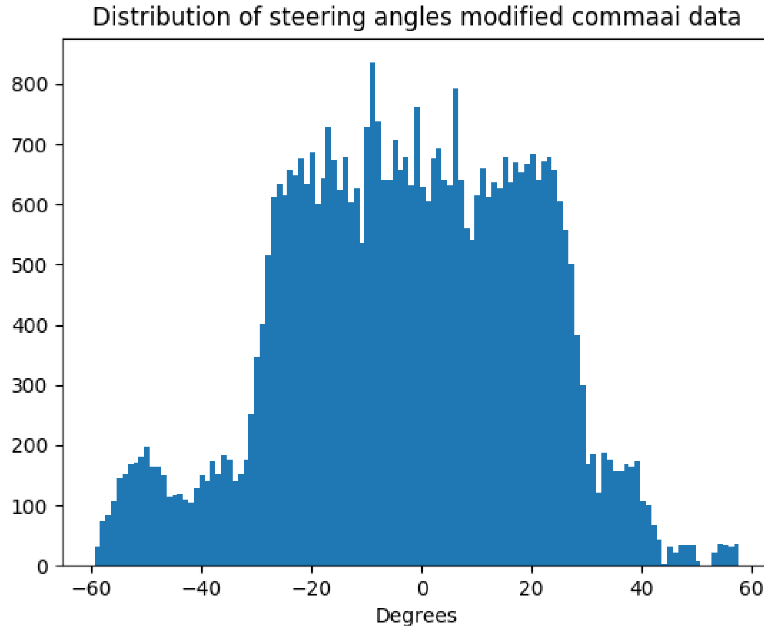

Figure 19: Distribution of steering angles in the commaai data set after over- and undersampling and other augmentations.

5.2 Resources

5.2.1 Software

All networks in the thesis was implemented with the *Python* package *Keras* [39] which was run on top of the open source machine learning framework *Tensorflow*. For hyperparameter optimization the *Hyperas* package was used [41]. Hyperas runs on top of Keras.

The car simulator used was the one created by *Udacity* [40]

5.2.2 Hardware

Keras is capable of running on GPU's as well as CPU's. However, in this project only CPU's were used. Mainly two computational resources were used. The first one was a computer provided by the Centre for Mathematical Sciences at Lund University. The second one was the resources provided by the center for scientific and technical computation at Lund University, also known as LUNARC.

5.3 Tasks

The project can be divided into three main tasks with increasing complexity. Task 1 was to design and train a neural network to perform decently on the artificial data set. I.e. the network would train on one part of the artificial data set and it's performance would be measured on another part of the data set.

Task 2 was to design and train a neural network to perform on the commaai data set. The task was divided into two sub tasks, task 2.1 and 2.2. Task 2.1 was to do precisely the same thing as in Task 1, but on the commaai data set. In task 2.2, the domain of the training data and the data that the performance was measured on would differ in some way. See *domain adaption* in section 4.7.9. The different domains in this case was data from driving during day and driving during night respectively.

Task 3 was an even more distinct case of domain adaption. The network would train on data from the commaai data set and it's performance was measured on the artificial data set.

Activation functions The ELU activation function (see 4.2.4) was consistently used in all networks in all three tasks due to its advantages regarding vanishing gradients and dead units. Minor insignificant experiments were conducted with other activation functions. For example, since the steering angles were limited to between -1 and 1, a bounded activation function in the output layer such as the sigmoid function might be preferable over an unbounded such as ELU. However, this slows down learning more than it helps the process.

Preprocessing of input The input images were zero-centered and normalized (see section 4.6.5). The zero-centering was done by subtracting 127.5 from every input value. Motivated by the fact that the input consisted of images with pixel values ranging between 0 and 255. With the same motivation normalization was done by dividing the zero-centered values by 127.5. Resulting in input values between -1 and 1.

In an attempt to improve the domain adaption necessary in task 2.2 and 3 another preprocess method was briefly experimented with. The method normalized the input pixels with regards to the 95 percentiles of all pixel values in the current input image. I.e. the 95 percentiles over all pixel values over all channels in the input image was calculated. Consequently, 95% of all input values were within that value (in magnitude). Then, that calculated value represented 1 (and -1) and the input values were zero-centered and normalized accordingly. The idea was to make the input during validation/test more similar to the input during training. In Figure 20 the distribution of pixel values over all three channels over a subset of images from both the artificial data set and the commaai data set are plotted. In the figure "Validation" denotes the artificial data and "Real life" the commaai data.

The result of the preprocess with respect to the 95 percentiles on the same two subsets are plotted in Figure 21.

From Figure 20 and Figure 21 it looks indeed like the network would be exposed to more similar values during training and validation if the normalization with respect to the 95 percentiles where used. Yet, no significant improvement could be observed in the training process so the method was not utilized in the succeeding work and left for further discussion and exploration.

Loss function Throughout the project the quadratic loss function (see section 4.7.2) was used in the training of the networks. The quadratic loss function is a well established reliable loss function that works with the ELU activation function. No major considerations or work have been put down to try different loss functions simply due to limitations. Testing different options when training neural networks is a time consuming process and limitations have to be set. The choice of loss function was one such. So to the extent of this report it is unclear whether an alternative loss function would have yielded better results. For task 2 and 3 (see 6.3 and 6.4) networks with several outputs were used. In these cases based on the importance of the outputs the outputs can have

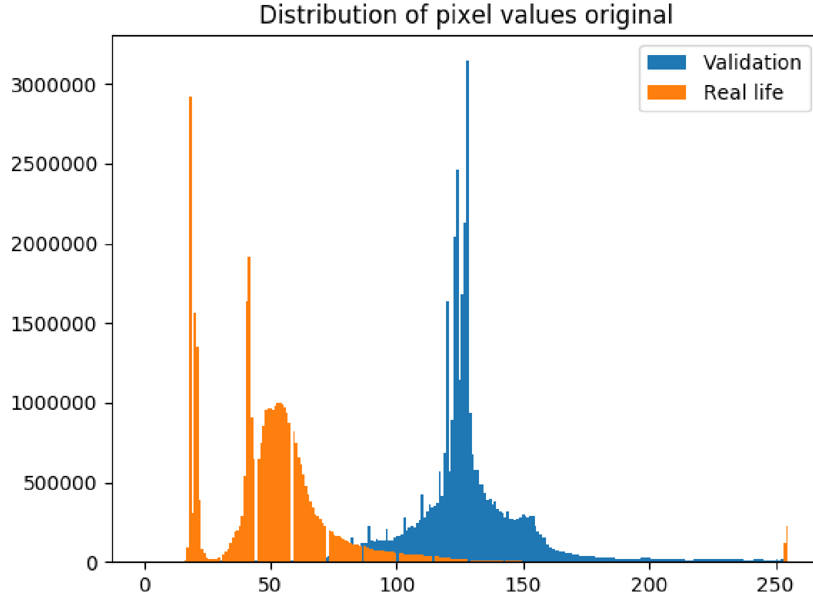

Figure 20: Distribution of pixel values over a subset of images from the artificial and commaai data set respectively. Here "Validation" denotes the artificial data and "Real life" the commaai.

different weights. Note that these weights are not trainable parameters like the weights on the different connections in the network. The weights on the outputs are defined as coefficients to multiply the loss with from respective output. The total loss that the backpropagation algorithm is then based on is the weighted sum of the individual losses. For example, if a network has two outputs and the weights are set to 0.5 and 1. Then the second output is twice as important as the first and the total loss will be $0.5\times$ first loss $+ 1\times$ second loss.

Optimizer In a similar manner to the choice of loss function the use of optimizer (stochastic gradient descent method, see section 4.7.3) was restricted to the *Adam* version. The Adam method has performed well compared to other methods in tests [5] and is considered a good choice. Other options may of course have produced better results and further testing of alternative optimization methods are welcome.

Regularization Dropout is the dominating method for regularization used in the project. Initially some brief test with L2-regularization was conducted but was replaced by dropout. To the extent of the knowledge of the author, L2-regularization doesn't offer any important advantages over dropout. In addition, according to Srivastave et al. [25] dropout outperforms L2-regularization in their tests. Since other techniques such as batch normalization and multi-task learning indirectly also works as regularizers no extra regularization technique was considered.

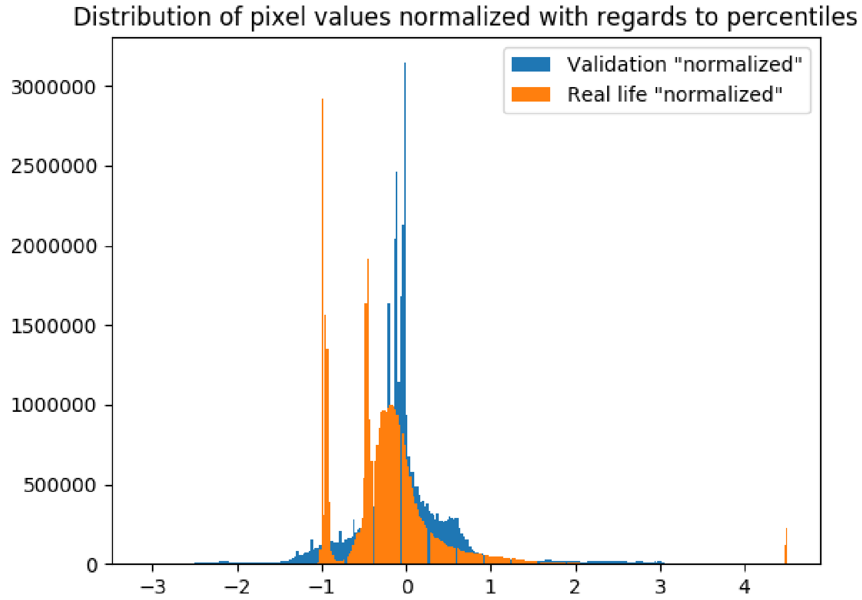

Figure 21: Distribution of pixel values over a subset of images from the artificial and commaai data set respectively after normalized with respect to the 95 percentiles. Here "Validation" denotes the artificial data and "Real life" the commaai.

Hyperparameters For task 2 and 3 the two hyperparameters learning rate and dropout rate was optimized with the *Hyperas* package. Hyperparameter optimization is a very time consuming process and consequently limitation had to be set regarding which parameters to optimize, intervals for the hyperparameters to evaluate and number of epochs to evaluate over etc. In the trade off between speed and accuracy/quality the following limitations were set:

1. Evaluate over 10 epochs with 10 000 samples per epoch
2. Maximum number of evaluation runs: 5 (i.e. 5 different combinations of hyperparameter were tested in total.
3. The interval for the dropout rates were set to between 0 and 1. The learning rate could be either 1.0e-3 or 1.0e-4.

The batch size used was narrowed down to an interval of about 40-200 through a combination of theoretical and empirical knowledge. The interval showed a good balance between speed and ability to converge. Too small batch sizes tended to give very fluctuating loss (and validation loss) curves and not always converge. It also made networks more prone to putting out a constant value as steering angle. I.e. regardless of input a network yields the same output steering angle. Too large batch sizes on the other hand made the training unstable and could give validation losses towards infinity.

The number of epochs was usually set to adapt the training time so that results could be evaluated during working hours. An option was to use for example early stopping (see section 4.8.1), but the chosen approach didn't pose any troubles so no reason was found to change it.

Initialization of network The initialization of the network parameters was done according to the suggestion by Glorot and Bengio [18] (see 4.7.5). The method is the default method in *Keras*. Brief tests were conducted with the initialization suggested by He et al. [19] without any significant difference in the results. Hence, no further tampering with the initialization method was done. Note though that only brief tests were performed and more extensive ones might render different results.

Performance measures Essentially three performance measures have been used throughout the project; loss/cost, confusion matrix and performance in the simulator. The loss have been monitored for both the training data and validation data but the loss on the validation data is of course the important one. The loss is the most objective performance measure and have been utilized in all parts of the project.

Confusion matrix is a good complement to loss since it can give indications on if the output of the network is biased in any way. For example can a confusion matrix show that a model constantly predicts steering angles that are too large (i.e. turning too much to the right), unlike the mean squared error loss that will just show the magnitude of the difference between correct and predicted steering angle. A confusion matrix requires that the output can be divided into classes (see section 4.11.3). Since the steering angle can be within a continuous range the range was discretized into whole degrees between -30 and 30. Hence the outputs were discretized into 61 "categories" representing steering angles from -30°to 30°. Confusion matrix was not applied in task 1 though since it was not prioritized.

The third performance measure utilized was the performance in the simulator. Naturally, it was applied to task 1 and 3. The performance in the simulator is a more subjective measure than the other two. The performance was an aggregate of time and distance before/if the car drove off the road along with reactions to changes in the environment, placement on the road etc. For example, two models could drive of the road at the same location of the track. However, the first one puts out an almost constant steering angle close to zero but since the track starts of almost straight the car doesn't drive of the road until the first curve. The second on the other hand smoothly zig-zags between the two lines defining the road until it reaches the first curve. There, it overestimates the curve and turns too sharply and drives off the road. Clearly, even though the two models made it equally far the second one performed better and is more promising. To perform well in the simulator was the ultimate goal for the models in task 1 and 3.

General tests/precautions Some initial tests on the network used were executed in order to make sure that it had some basic necessary properties.

First the network was trained exhaustively on a small data set (typically a couple of hundred images or fewer) to see if it could overfit the data. If not, there is no point in continuing training it on more data since something is wrong in the design or training process.

Second, the initialization of the parameters in the network was tested. This was done by comparing the output of the initialized network, i.e. not trained at all just initialized, with chance. Chance in this case was a uniformly randomly picked steering angle between the minimum and

maximum steering angle in the test batch. If the initialization is done properly the resulting loss from the network output should at least have the same magnitude as the loss from chance.

5.3.1 Task 1

Model Architecture The model architecture used for task 1is illustrated in Figure 22. Different hyperparameters was tested such as batch size and learning rate. Also, batch normalization was introduced between layers in order to speed up training and possibly improve the performance. Specifications for the best performing network are presented in the results section.

For task 1 the artificial data set was split with the hold out method into training data and validation data with the ratios 0.7 and 0.3 of the full data set respectively. The partition was made randomly. Consequently, the training set consisted of 5271 images and the validation set of 2259 (Remember that the original artificial data set was doubled in size). In addition to this, as mentioned above, oversampling was utilized. The performance in the simulator was then used as a test.

During training further data augmentation was utilized (see section 4.6.4) such as horizontal flipping, translation, shadowing and brightness.

The performance of the network was measured with the validation loss and how well it performed in the simulator. I.e. how well the network managed to steer the car in the simulator.

In the early stages of the project the design of the network was based on the one suggested by Ibrahim in [37]. On the same data set (artificial data set) she reports reaching a validation loss lower than 0.01. Details about hyperparameters used such as samples per epoch for example is not reported in her paper. Minor efforts was made to reproduce her results, without success.

In the next step the design of the network was based on NVIDIA's network that has been proven to work really well in real-life driving [15].

5.3.2 Task 2

Model Architectures For task 2 the approach was to base the design of the network on NVIDIA's design since it has been proven to work well on the type of data that the commaai data set consisted of. However, some adjustments might be beneficial since NVIDIA had access to a much larger data set. According to their report they had access to 72 hours of driving [15], compared to the full original commaai data set consisting of 7.25 hours. Add to that that the commaai data set was heavily scaled downed for training. Consequently, one might expect a network trained on the commaai data set more prone to overfitting than if it was trained on the data set that NVIDIA had access to. The first design used for task 2 (referred to as *NVIDIA* or *basic*) is illustrated in Figure 23. As an alternative to the NVIDIA design, a multi-task learning network was trained under the same conditions throughout task 2. The design of the multi-task learning network can be seen in Figure 24, but the idea was to combine NVIDIA's network with an autoencoder (see section 4.5). Hence, the network would train to predict steering angles simultaneously as learning to reproduce the input image. This was done by using hard parameter sharing (see section 4.7.10) between the convolutional layers in the NVIDIA network, which also worked as an encoder network. This was done in line with the hypothesis that multi-task learning might improve the performance of the network with regards to predicting steering angles. The theory is that it might be especially beneficial in task 2.2, where domain adaption is necessary. For completeness the two design were also compared for task 2.1. In different scripts etc. the networks based on the NVIDIA design have

also been referred to as *basic* or *NVIDIA_basic*. The multi-task networks on the other hand have been referred to as *NVIDIA_autoenc* or "the autoencoder network" or something similar. Thereof some of the titles on figures in the report.

Task 2.1 Two data sets were used for task 2.1, one training set and one test set. The training set, called *train data*, was the modified commaai data set whose distribution is displayed in Figure 19. The test set, called *test data*, was a fraction of the sorted commaai data set. The distribution of the test data can be seen in Figure 25. The training data set size was about 6 GB and the validation data set about 4 GB.

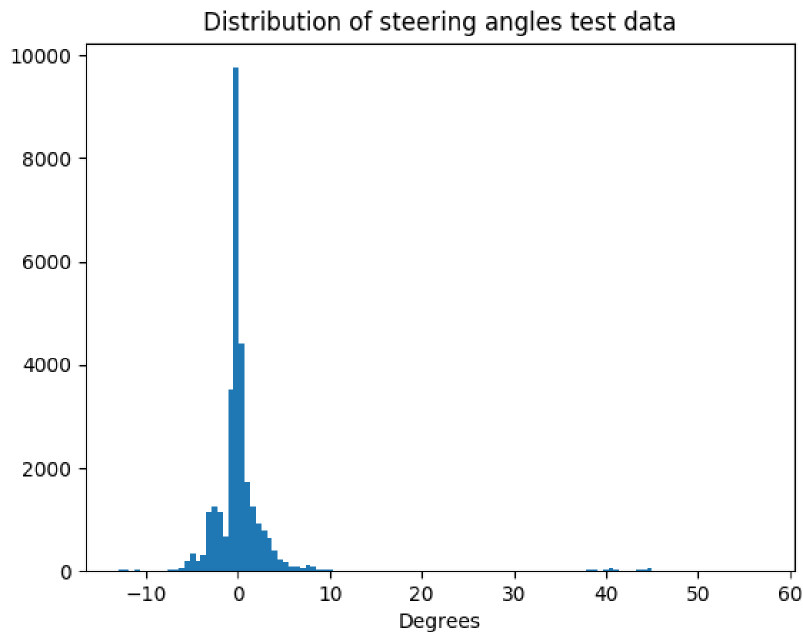

Figure 25: Distribution of steering angles in the test data set used for task 2.1

As can be seen in Figure 25 the test data is very concentrated towards steering angles close to 0°. Consequently, in order to make the test set represent a more diverse range of steering angles, and ultimately better represent what the model might face in reality, the test data was modified in a second comparison. In the second comparison, the data used for performance measures was selected from the test data randomly but with higher probability to pick steering angles with large magnitude. Basically a version of over- and undersampling. An example of how the resulting distribution of steering angles could look can be seen in Figure 26. When this kind of modified test data was used it was sometimes referred to as *selective* or *selective data* or similar. Thereof some of the titles on figures in the report.

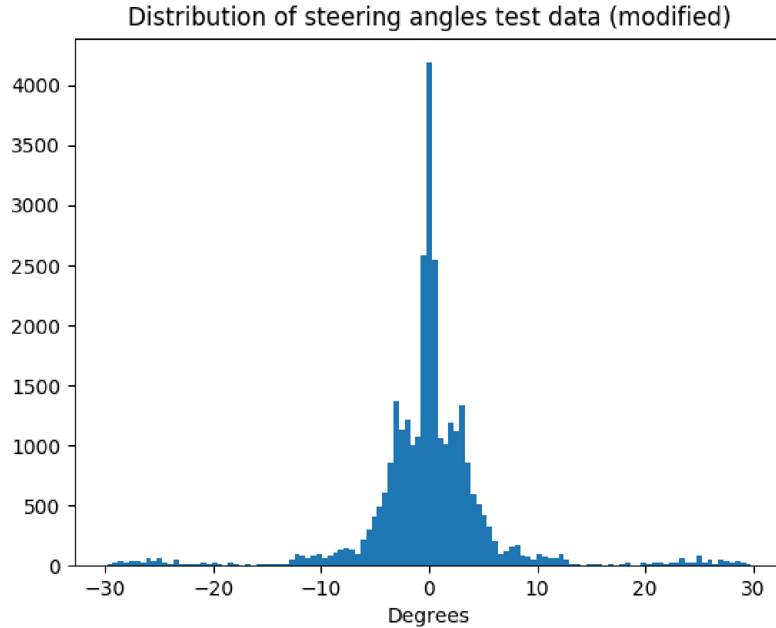

Figure 26: Example of distribution of steering angles in the test data set used for task 2.1 after over- and undersampling.

The following approach was taken for task 2.1:

1. Optimize the NVIDIA design with respect to the validation loss on the test data.
2. Train the networks on the train data.
3. Compare validation loss and confusion matrix between the networks on the test data.
4. Compare validation loss and confusion matrix between the networks on the modified test data.

The optimization mentioned above refers to hyperparameter optimization with *Hyperas* and with respect to the dropout rates for the two dropout layers and the learning rate.

Task 2.2 For task 2.2 two different data sets were used. A subset of the commaai data set was used for training. The data set consisted exclusively of images from day driving. Here day driving means that the data was collected during a time of the day when the sun was up. This data set was called *day data*. The day data was modified in the same way as the modified commaai data set whose distribution is plotted in Figure 19. I.e. over/undersampling and different augmentation techniques were used.

As validation data a subset of the commaai data set was utilized. The data set only contained images from night driving and was called *night data*. For the first comparison the night set was left

untampered. Then for a second comparison, like in task 2.1. the night data was modified to better represent the reality. This was done with the same method as in task 2.1. To further evaluate the performance of the models a third comparison was done on a small subset of the night data. This subset was collected in such a way that all steering angles between -25 and 25 was represented equally many times in the set. The set is referred to as *uniform* or similar. The set was used to get clearer confusion matrices and better see how the models handled curves.

Since the training and validation data differed distinctively in distribution of pixel values, (see Figure 27), the networks domain adaption ability could be evaluated.

The same structure on the approach as in task 2.1 was used in task 2.2, i.e.:

1. Optimize the NVIDIA design with respect to the validation loss on the night data.
2. Train the networks on the day data.
3. Compare validation loss and confusion matrix between the networks on the night data.
4. Compare validation loss and confusion matrix between the networks on the modified night data.
5. Compare validation loss and confusion matrix between the networks on the uniform night data.

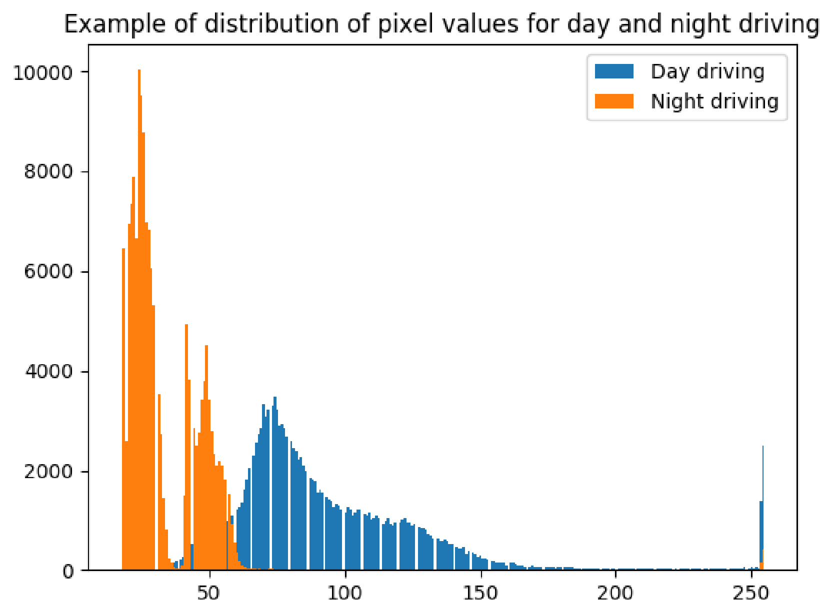

Figure 27: Example of distribution of pixel values for an image from the day and night data set respectively.

5.3.3 Task 3

Model Architecture The same designs on the networks as in task 2.2 were used (see Figure 23 and 24) with the same motivation.

The approach for task 3 was to, except for some minor tweaking, use the best performing NVIDIA and multi-task designs from task 2.2 including hyperparameters. With the motivation that task 3 is in theory the same as task 2.2 only with different data sets.

For task 3 brief attempts of using parts of a pre-trained VGG network [7] was conducted. More specifically the VGG16 network [38] available as a pre-trained model in Keras was used. However, the parameters in the convolutional layers were set and could not be changed by the training process. Only the parameters in the finishing fully connected layers was tweaked during training. The idea was to transfer the "knowledge" from the extensively trained VGG16 network to a network for task 3. According to the transfer learning theory 4.7.9 this might reduce training time and enhance performance. The VGG16 network is a deep convolutional neural network that has been trained to classify images in the *ImageNet* data set in to categories such as "apple", "horse", "car" etc. The ImageNet data set is a huge data set containing images in a wide range of categories. VGG16 is renowned to perform well when it comes to classify images in ImageNet. The idea of using the convolutional part of the trained VGG16 is that it works as the feature extractor of the network. The following fully connected layers then works as classifiers. Hence, the features extracted by the convolutional layers may be useful for the purpose of predicting steering angles to. Consequently, training only fully connected layers after the pre-trained convolutional layers effectively trains a fully connected network to predict steering angles from the features extracted by the pre-trained layers.

However, fully connected networks with a lot of nodes are time consuming to train and even though the fully connected layers were reduced for task 3 from the original ones in VGG16 the training process was slow. In addition, when the trained VGG16 modified networks didn't perform better than the other networks trained (NVIDIA and multi-task networks) the approach was not prioritized.

Two data sets were used for task 3. To train the network the same data set was used as in task 2.1 (see Figure 19) and for validation the artificial data set was used. As mentioned earlier the steering angles in the artificial data set was very concentrated around 0. Hence, in order to get the validation loss to better represent the networks performance a "uniform" set was used as validation. The set was created in the same way as the "uniform" set in task 2.2 (see section 6.3).

6 Results

6.1 Data

In the attempt of finding different color channels and formats that made the pixel values of the two data sets (artificial and commmaai) overlap the most, the S and V channels in HSV format looked most promising. Despite this no significant improvement could be observed in the results when those channels were used instead of YUV. Hence, due to restrictions no further attempts were made. In addition, in order to compare the different data sets one need access to both of them. One of the main goals of the project was to see how network can handle domain adaption where one of

the data sets are unknown. So having access to both data sets is a presumption that is not satisfied in this case. Ultimately you want a method that generalizes regardless of test data (unknown).

6.2 Task 1

As mentioned in section 5.1 the artificial data set was biased and also consisted to a large extent of steering angles close to 0. The actions taken to remedy this helped against it but not completely. The effect of the modification of data could be seen in some of the tests. One observation was the following. A network was trained first on the original artificial data set, and then on a modified/augmented version, resulting in two models. Both models were then evaluated on a validation set, consisting of original images from the artificial data set. The result was that the model trained on the original set reached a lower validation loss than the model trained on the modified/augmented version. However, the second model outperformed the first one in the simulator. Since the driving in the simulator is the ultimate goal, the second model can be considered the best one even though it reached a higher validation loss. This was an example of how the performance measure can be insufficient and misleading. The first model probably reached a lower validation loss because the set it trained on had a more similar distribution to the validation set, compared to the second model. When the models were exposed to the images in the simulator though, which probably consisted to a higher extent of larger steering angles, the second model was better trained.

The best performing network on task 1 had 252 691 trainable parameters and the design is illustrated in Figure 22. The network was trained with the following hyperparameters:

Dropout rate:	0.5
Number of epochs:	100
Samples per epoch:	50 000
Batch size:	100
Learning rate:	0.001

Note that here the Dropout rate means probability to drop a node, opposite to what is described in the theory section (and in the original paper [25]). Even though they in this particular case are equal. For convenience, this definition of dropout rate will be used throughout the rest of this section.

With these hyperparameters the model reached the following values:

Loss:	0.013
Validation loss:	0.036
Performance in simulator:	Drove several laps before it was manually stopped.

A graph over the loss and validation loss curves can be seen in Figure 28.

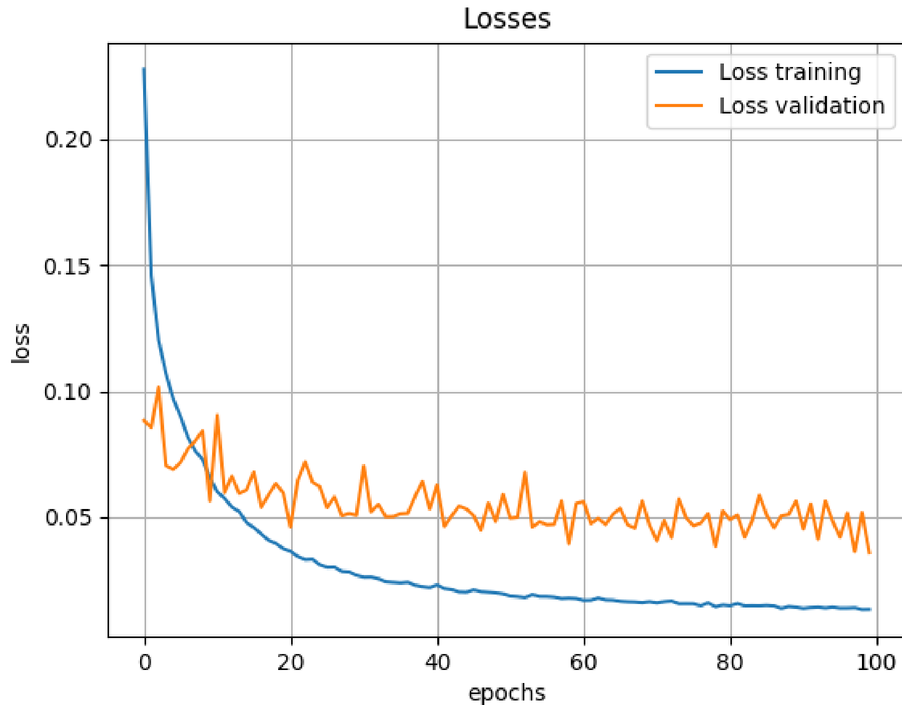

Figure 28: Training loss and validation loss for the best performing network on task 1. The training loss is the mean squared error on a subset of the artificial data set. The images in the data set have been augmented to increase the effective size of the set. The validation loss is the mean squared error on another subset of the artificial data set. In the validation set no images were augmented. An epoch represents 50 000 images. Both curves have a downwards trend during the whole training period, even though it looks like the validation loss might stagnate towards the end. The fact that both curves have a downwards trend is a good sign and indicate that neither under or overfitting occurs. Note that the loss starts with a higher value than the validation loss but eventually reaches a lower value. This is probably due to how the program calculates the loss and an effect of dropout.

6.3 Task 2

6.3.1 Task 2.1

NVIDIA design Here the results for the NVIDIA design which performed best on task 2.1 are presented. The network had 252 219 trainable parameters and the design is illustrated in Figure 23.
The optimization of the NVIDIA design gave the following hyperparameters:

Dropout rate layer 1:	0.23
Dropout rate layer 2:	0.91
Learning rate :	0.001

Which were applied along with the following hyperparameters:

Number of epochs:	20
Samples per epoch:	100 000
Batch size:	100

With these settings the model reached the following losses:

Loss:	0.02
Validation loss:	0.007
Validation loss (modified):	0.10

Where loss refers to the loss on the training data, validation loss on the test data and validation loss (modified) on the modified test data.

The progress of the loss and validation loss are plotted in Figure 29 and the resulting confusion matrix in Figure 30. The confusion matrix on the modified test data is displayed in Figure 31.

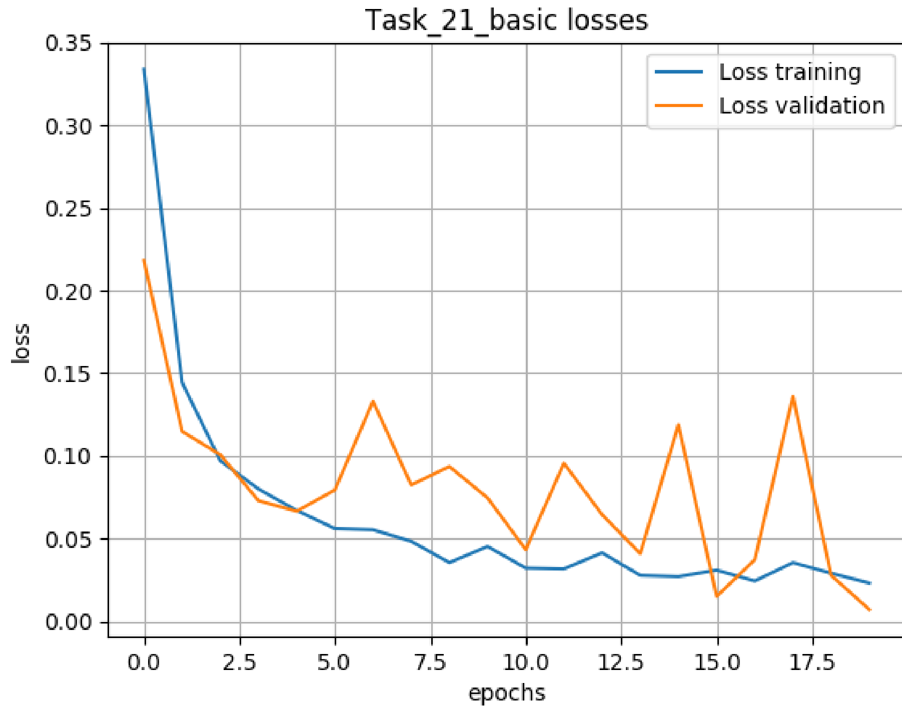

Figure 29: Training loss and validation loss (mean squared error) for the NVIDIA network. The training loss is with respect to a subset of the commaai data set (images from real life driving) that has been augmented to increase the effective size of the set. The validation loss is with respect to another subset of the commaai data set without augmentation. One epoch represents 100 000 images. Both curves have an overall downwards trend, even though the validation loss fluctuates heavily. The fluctuations make the validation loss less reliable and consequently additional tests/metrics should be considered before making any conclusions

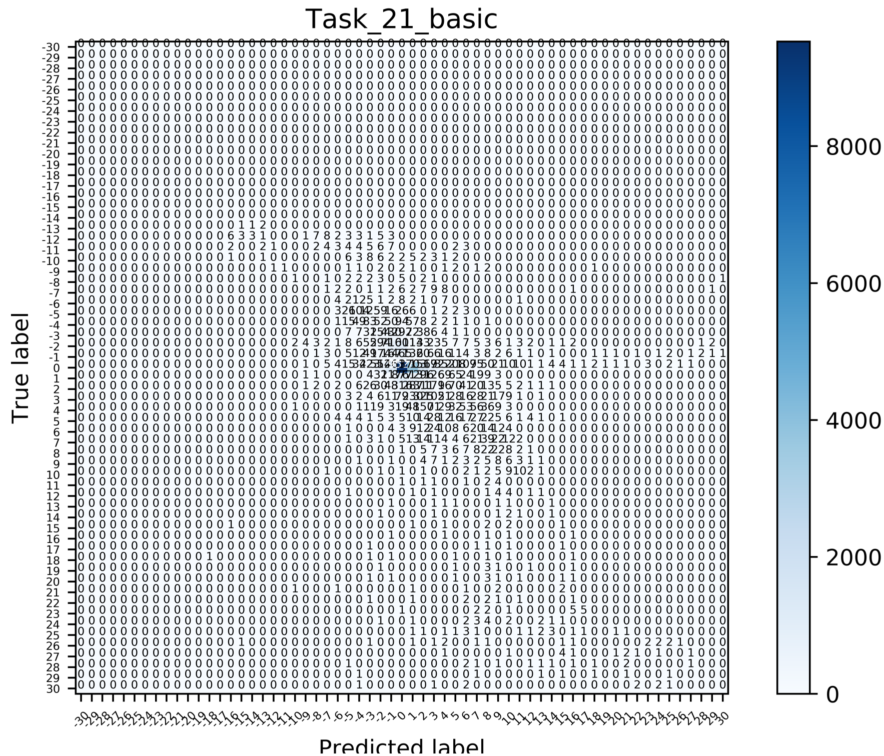

Figure 30: The figure shows the resulting confusion matrix of the trained NVIDIA network evaluated on the test data set. The test data set is a subset of the commaai data set, which consists of images from real life driving. For every image in the test data set there is a correct label (steering angle) which are represented by the rows in the confusion matrix. The columns represent the output steering angle that the network gave for every image in the test data set. An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. As can be seen in the figure the confusion matrix is not diagonal, but it has tendencies towards being so. That is a good sign and shows that the network is able to somewhat predict correct steering angles in the test data set.

Figure 31: The figure shows the resulting confusion matrix of the trained NVIDIA network evaluated on a subset of the test data set. The subset was chosen in a way that favors images with large magnitude steering angles as labels. So compared to the set used in Figure **??**, the set used here has a higher share of images corresponding to curves. For every image evaluated there is a correct label (steering angle) which are represented by the rows in the confusion matrix. The columns represent the output steering angle that the network gave for every image in the set evaluated. An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. As can be seen in the figure the confusion matrix is not diagonal, but it has tendencies towards being so. That is a good sign and shows that the network is able to somewhat predict correct steering angles. The appearance of the confusion matrix is similar to the one in Figure **??**, which is reassuring.

Multi-task design Here the results for the multi-task network that performed best on task 2.1 are presented. The network had 407 138 trainable parameters and the design is illustrated in Figure 24.

Since the autoencoder part of the network works as a regulizer for the steering predictor, a network with heavily decreased dropout rates yielded better results for the multi-task design. The following hyperparameters were used:

Dropout rate layer 1:	0.01
Dropout rate layer 2:	0.01
Learning rate :	0.001

Which were applied along with the following hyperparameters:

Number of epochs:	8
Samples per epoch:	25 000
Batch size:	100
Loss weights [autoencoder loss, steering angle loss]:	[0.2, 1]

With these settings the model reached the following losses:

Loss:	0.028
Validation loss:	0.069
Validation loss (modified):	0.12

Where loss refers to the loss on the training data, validation loss on the test data and validation loss (modified) on the modified test data.

The progress of the losses are plotted in Figure 32 and the resulting confusion matrix in Figure 33. The confusion matrix on the modified test data is displayed in Figure 34.

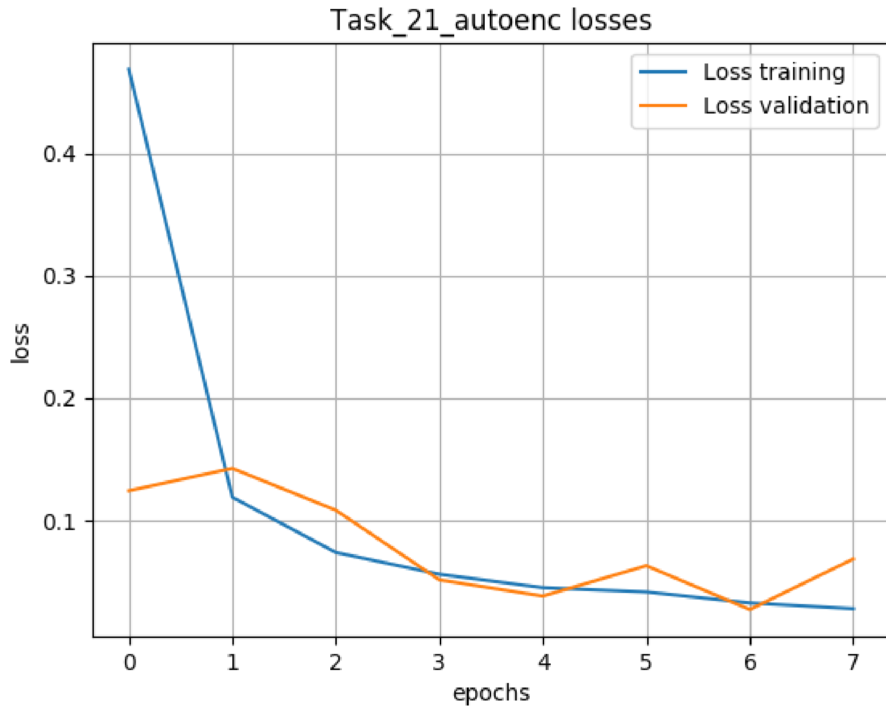

Figure 32: Training loss and validation loss (mean squared error) for the multi-task network. The multi-task network technically has two training and validation losses each and a total aggregated loss, but the ones displayed here are the losses for the steering angles only. The training loss is with respect to a subset of the commaai data set (images from real life driving) that has been augmented to increase the effective size of the set. The validation loss is with respect to another subset of the commaai data set without augmentation. An epoch represents 25 000 samples. The training loss has a clear downward trend during the whole training period. The validation loss has an overall downward trend but not as clear as the training loss. The final training loss is similar in size to the one attained by the NVIDIA network (Figure 29)
. The validation loss though is much higher. However, considering the fluctuations in Figure 29) the difference is not too reliable.

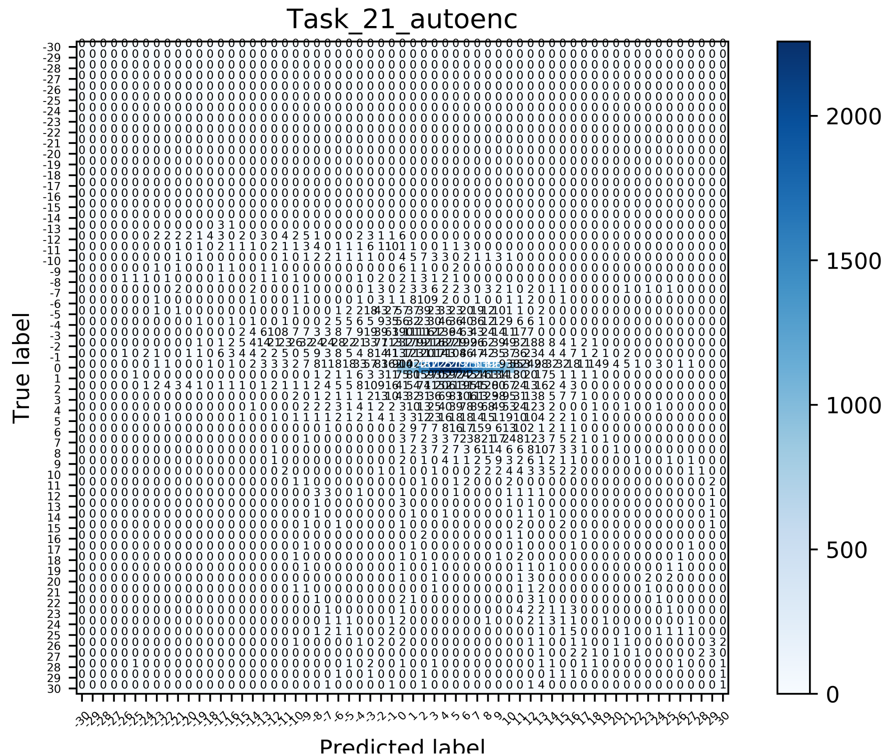

Figure 33: The figure shows the resulting confusion matrix of the trained multi-task network evaluated on the test data set. The test data set is a subset of the commaai data set, which consists of images from real life driving. For every image in the test data set there is a correct label (steering angle) which are represented by the rows in the confusion matrix. The columns represent the output steering angle that the network gave for every image in the test data set. An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. As can be seen in the figure the confusion matrix is not diagonal but looks rather like a cluster around true label 0 and predicted label 6. Entries above the diagonal means that the network predicted a too large steering angle for that input and entires below the opposite. The majority of entries in the matrix displayed are above the diagonal, which indicated that the network is biased towards large steering angles. In this situation it means biased towards turning right. Compare Figure 30

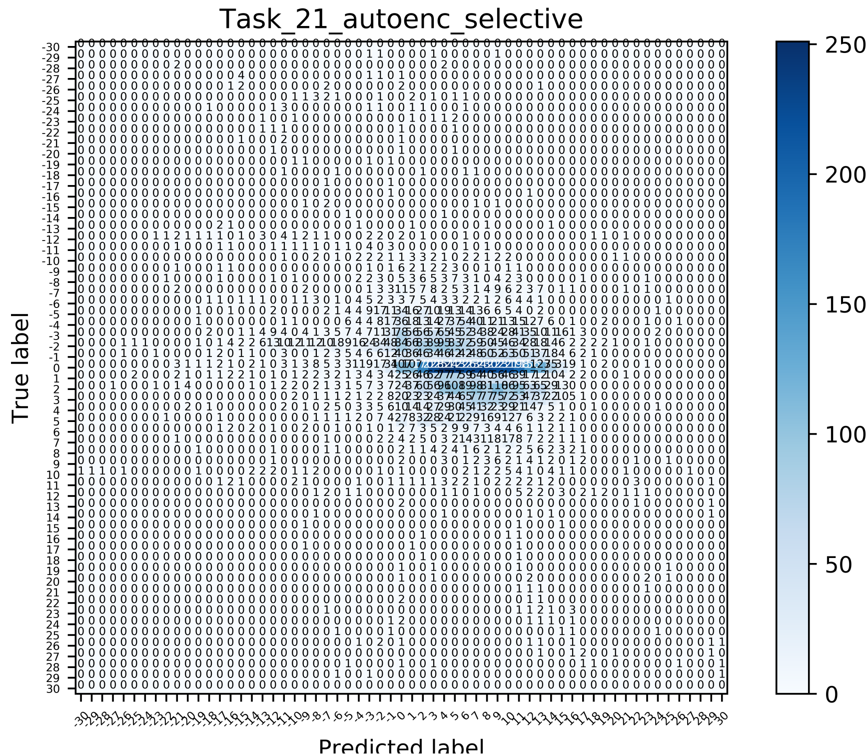

Figure 34: The figure shows the resulting confusion matrix of the trained multi-task network evaluated on a subset of the test data set. The subset was chosen in a way that favors images with large magnitude steering angles as labels. So compared to the set used in Figure 33, the set used here has a higher share of images corresponding to curves. For every image evaluated there is a correct label (steering angle) which are represented by the rows in the confusion matrix. The columns represent the output steering angle that the network gave for every image in the set evaluated. An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. As can be seen in the figure the matrix looks similar to the one in Figure 33. I.e. the network seems to be biased towards right turns. Compare Figure 31.

6.3.2 Task 2.2

NVIDIA design Here the results for the NVIDIA design on task 2.2 are presented.
Compared to task 2.1 much less regularization was used for the NVIDIA design since it yielded better results. The following hyperparameters were used:

Dropout rate layer 1:	0.01
Dropout rate layer 2:	0.01
Learning rate :	0.001

Which were applied along with the following hyperparameters:

Number of epochs:	21
Samples per epoch:	20 000
Batch size:	100

With these settings the model reached the following losses:

Loss:	0.005
Validation loss:	0.058
Validation loss (selective):	0.073
Validation loss (uniform):	0.31

Where loss refers to the loss on the day data, validation loss on the night data, validation loss (selective) on the modified data and validation (uniform) on the uniform data set.

The progress of the losses are plotted in Figure 35 and the resulting confusion matrices on the three validation sets in Figure 36, 37 and 38 respectively. Note that the validation loss in Figure 35 is with respect to the uniform data set. Also note that the best validation loss is reached after 21 epochs (the graph starts with 0), even though the graph shows 30 epochs.

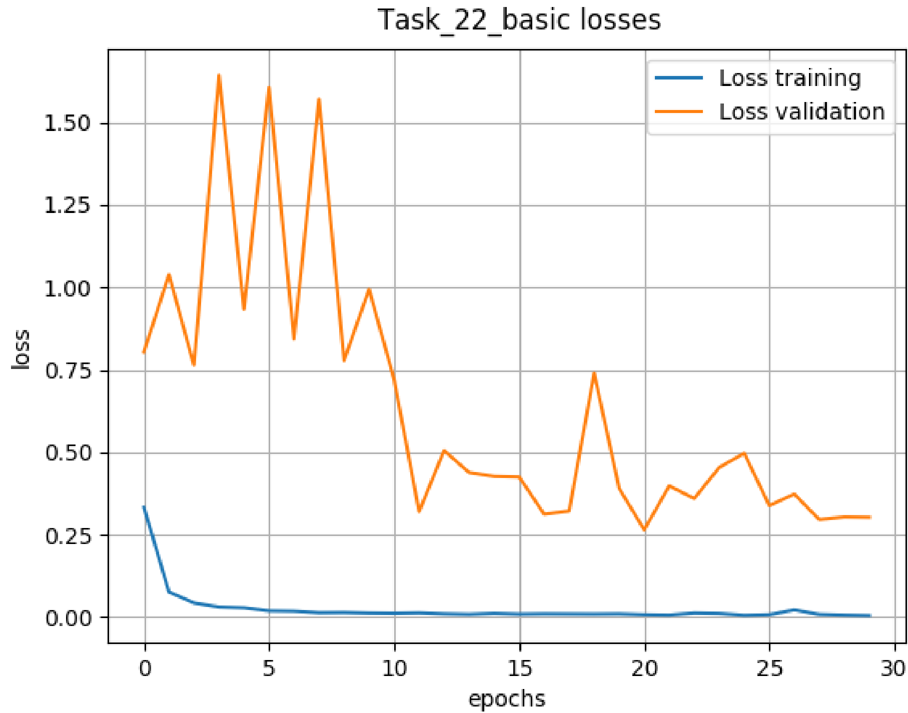

Figure 35: Training loss and validation loss (mean squared error) for the NVIDIA network on the day and uniform night data respectively. The blue graph shows the loss during the training of the network. The loss is with respect to a subset of the commaai data set consisting of images from driving in day light exclusively. Further, the images in the set was augmented for increased quality. The orange graph shows the loss on the validation set. The validation set here consists exclusively of images from driving during dark hours. The distribution of the steering angles in the validation set is uniform between -25°and 25°. One epoch represents 20 000 samples. The training loss clearly goes down initially and even if it's hard to tell it continues to go down. The validation loss fluctuates heavily and is much larger in magnitude than the training loss. Overall it seems to have a downwards trend, but after about epoch 11 it seems to stagnate. Stagnating validation loss along with decreasing training loss is often a sign of overfitting.

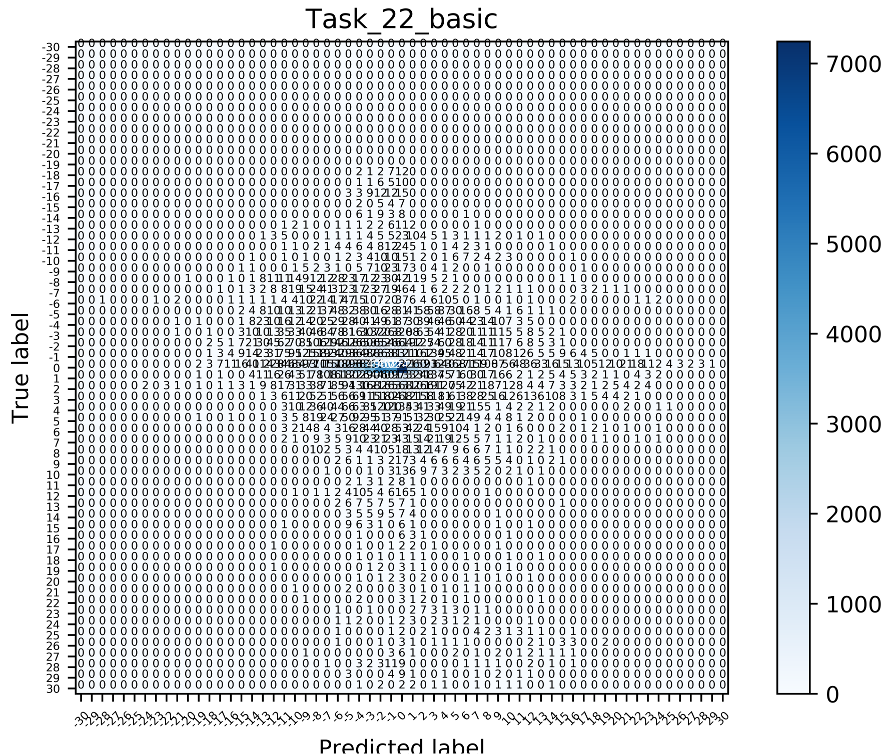

Figure 36: The figure shows the resulting confusion matrix of the trained NVIDIA network evaluated on the night data set. The night data set is a subset of the commaai data set, that consists of images from night driving exclusively. For every image in the night data set there is a correct label (steering angle) which are represented by the rows in the confusion matrix. The columns represent the output steering angle that the resulting network gave for every image in the night data set. An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. From the confusion matrix one can see that the evaluated network is not particularly good at predicting steering angles in the night data set. Instead of being diagonal most entries are located close to the center of the matrix (0,0).

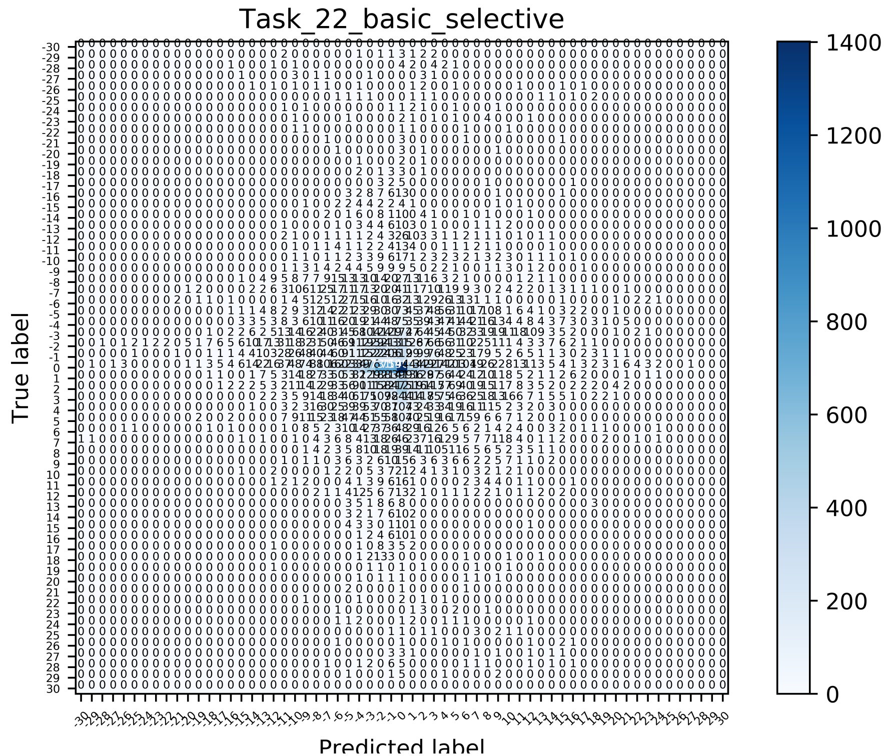

Figure 37: The figure shows the resulting confusion matrix of the trained NVIDIA network evaluated on a subset of the night data set denoted ”selective”. The ”selective” set is chosen from the night data set in a way that favors large magnitude steering angles. In other words does the ”selective” set consists of a higher percentage images corresponding to curves than the night data set. The rows in the matrix correspond to correct steering angles (labels) and the columns to steering angles predicted by the evaluated network An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. The figure clearly indicates that the network predicts steering angles poorly. A lot of steering angles predicted by the network are 0 despite that the true label is not. Note though that the matrix looks relatively symmetric in the x-axis which indicates that the network doesn’t seem to be biased towards any direction.

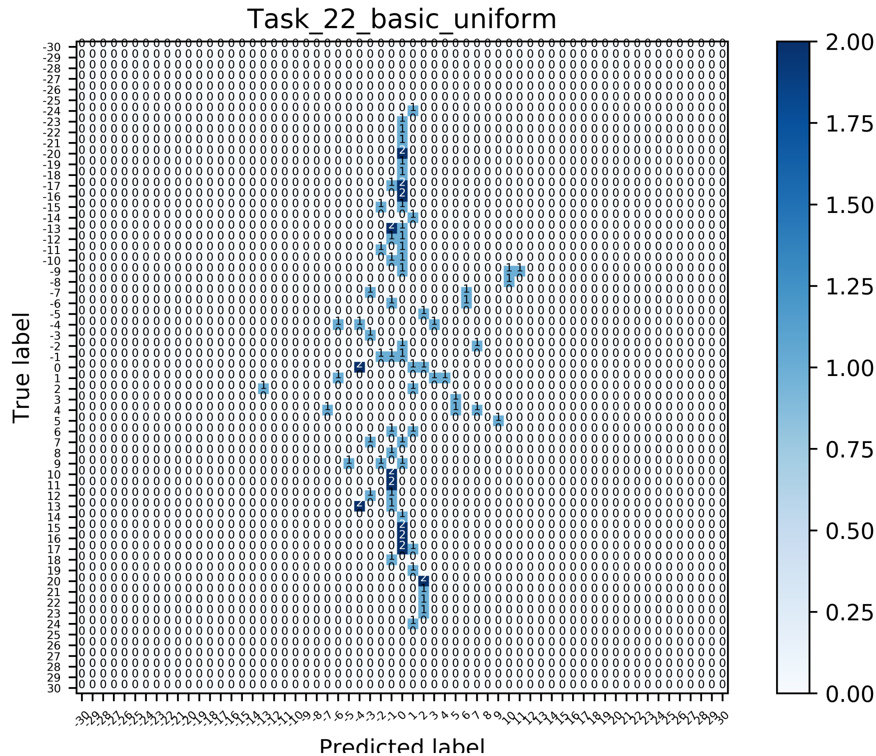

Figure 38: The figure shows the resulting confusion matrix of the trained NVIDIA network evaluated on a subset of the night data set denoted "uniform". The "uniform" data set has a uniform distribution of samples with regards to steering angles (labels) between -25°and 25°. I.e. it consists of equally many images with label -3 as 12 for example. The rows in the matrix correspond to correct steering angles (labels) and the columns to steering angles predicted by the evaluated network An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. The resulting matrix is far from diagonal and a lot of entries are concentrated around column 0. This confirms the pattern that could be discerned in Figure 41.

Multi-task design Here the results for the multi-task design on task 2.2 are presented. The following hyperparameters were used:

Dropout rate layer 1:	0.23
Dropout rate layer 2:	0.91
Learning rate :	0.001

Which were applied along with the following hyperparameters:

Number of epochs:	22
Samples per epoch:	20 000
Batch size:	100
Loss weights [autoencoder loss, steering angle loss]:	[1, 1]

With these settings the model reached the following losses:

Loss:	0.039
Validation loss:	0.14
Validation loss (selective):	0.18
Validation loss (uniform):	0.27

Where loss refers to the loss on the day data, validation loss on the night data, validation loss (selective) on the modified data and validation (uniform) on the uniform data set.

The progress of the losses are plotted in Figure 39 and the resulting confusion matrices on the three validation sets in Figure 40, 41 and 42 respectively. Note that the validation loss in Figure 39 is with respect to the uniform data set. Also note that the best validation loss is reached after 22 epochs (the graph starts with 0), even though the graph shows 30 epochs.

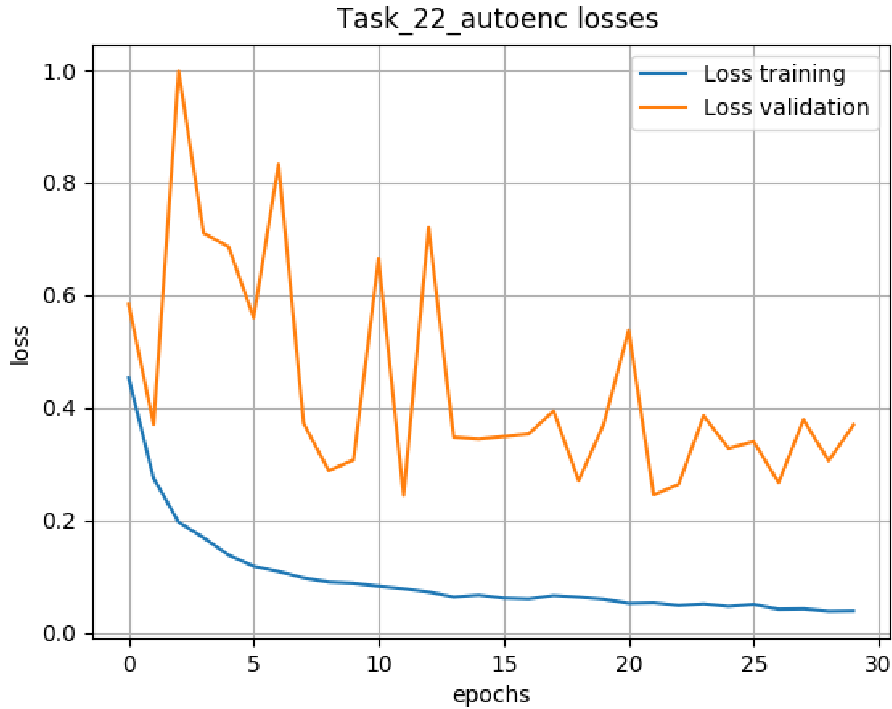

Figure 39: Training loss and validation loss (mean squared error) for the multi-task network on the day and uniform night data respectively. The blue graph shows the loss during the training of the network. The loss is with respect to a subset of the commaai data set consisting of images from driving in day light exclusively. Further, the images in the set was augmented for increased quality. The orange graph shows the loss on the validation set. The validation set here consists exclusively of images from driving during dark hours. The distribution of the steering angles in the validation set is uniform between -25°and 25°. One epoch represents 20 000 forward/backward passes in the backpropagation algorithm. The training loss clearly decreases during the whole training period, whereas the validation loss has a more unclear development. It seems to have an overall decreasing trend but it fluctuates substantially and after approximately epoch 15 it is unclear if it still decreases or not. Compare Figure 35.

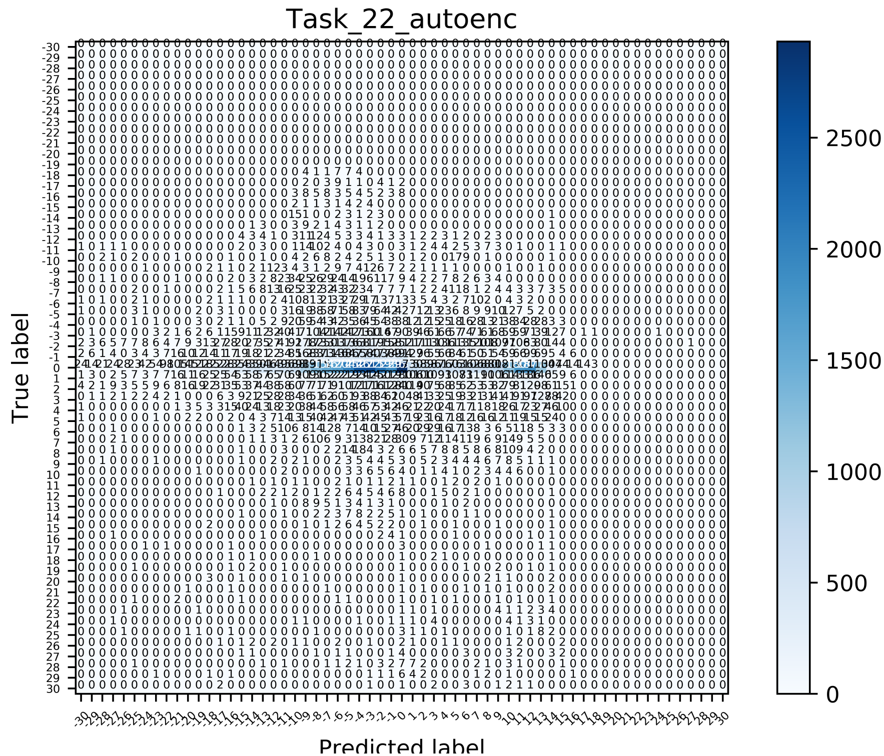

Figure 40: The figure shows the resulting confusion matrix of the trained multi-task network evaluated on the full night data set. The night data set is a subset of the commaai data set that consists of images from night driving exclusively. For every image in the night data set there is a correct label (steering angle) which are represented by the rows in the confusion matrix. The columns represent the output steering angle that the resulting network gave for every image in the night data set. An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. The resulting confusion matrix displayed is far from diagonal. Instead most entries are concentrated around row 0. This indicates that the network is very prone to predict large magnitude steering angles, even though the correct one is small. The network also seems to be inclined towards turning left.

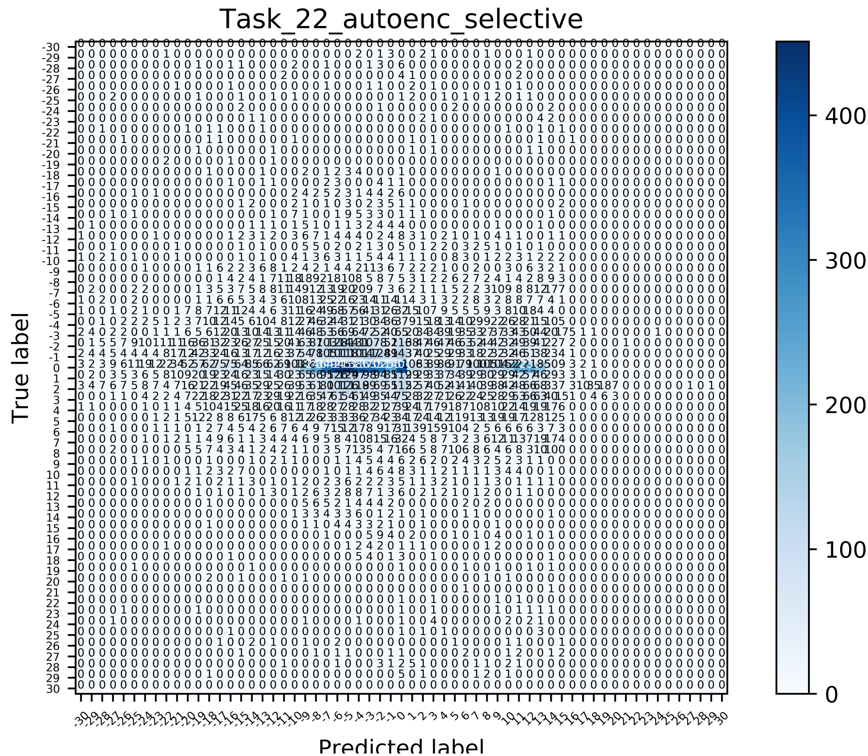

Figure 41: The figure shows the resulting confusion matrix of the trained multi-task network evaluated on a subset of the night data set denoted "selective". The "selective" set is chosen from the night data set in a way that favors large magnitude steering angles. In other words does the "selective" set consists of a higher percentage images corresponding to curves than the night data set. The rows in the matrix correspond to correct steering angles (labels) and the columns to steering angles predicted by the evaluated network An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. Much like in Figure 40, the network is biased towards left turns and also predicts too large steering angles in magnitude. Based on the appearance of the matrix the evaluated network is most likely a poor steering angle predictor. Compared to the NVIDIA network evaluated in Figure 38 though it is closer to being diagonal.

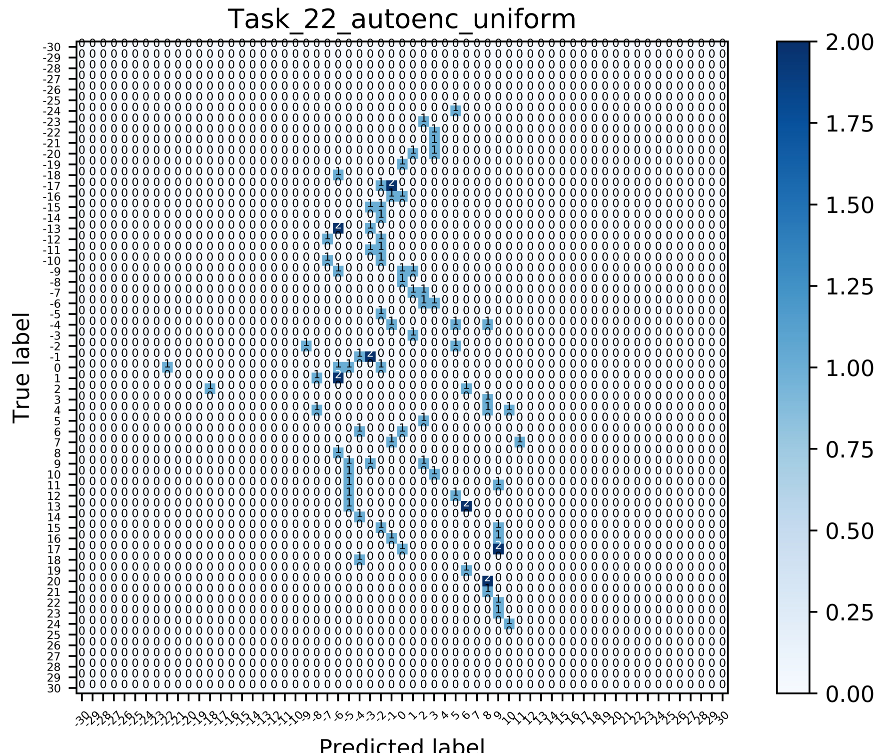

Figure 42: The figure shows the resulting confusion matrix of the trained multi-task network evaluated on a subset of the night data set denoted "uniform". The "uniform" data set has a uniform distribution of samples with regards to steering angles (labels) between -25°and 25°. I.e. it consists of equally many images with label -3 as 12 for example. The rows in the matrix correspond to correct steering angles (labels) and the columns to steering angles predicted by the evaluated network An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. The entries in the resulting matrix are scattered and are far from forming a diagonal matrix.

6.4 Task 3

NVIDIA design Here the results for the NVIDIA design on task 3 are presented.

The following hyperparameters were used:

Dropout rate layer 1:	0.01
Dropout rate layer 2:	0.01
Learning rate :	0.001

Which were applied along with the following hyperparameters:

Number of epochs:	25
Samples per epoch:	20 000
Batch size:	100

With these settings the model reached the following losses:

Loss:	0.02
Validation loss (uniform):	0.41
Performance in simulator:	Drove for about 1 minute before it drove off the road. It drove off in the first sharp turn that it faced.

Where loss refers to the loss on the training data and validation (uniform) on the uniform artificial data set. The progress of the losses are plotted in Figure 43 and the confusion matrix in Figure 44

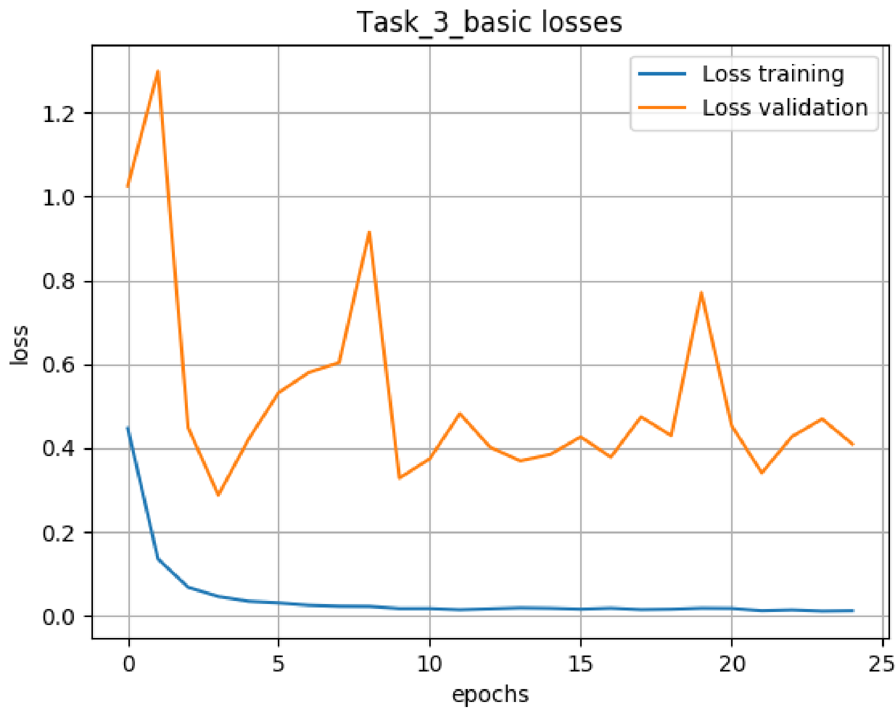

Figure 43: Training loss and validation loss (mean squared error)for the trained NVIDIA network in task 3. The training loss is with respect to the commaai data set, augmented for increased quality. The validation loss is with respect to a uniform subset of the artificial data set. The subset is uniform in the sense that all steering angles (labels) between -25°and 25°are represented equally many times in the set. One epoch is when 20 000 images have been trained upon. The training loss decreases steadily during the entire training period. The validation loss on the other hand fluctuates heavily and seems to stagnate after epoch 12. That might be a sign of overfitting.

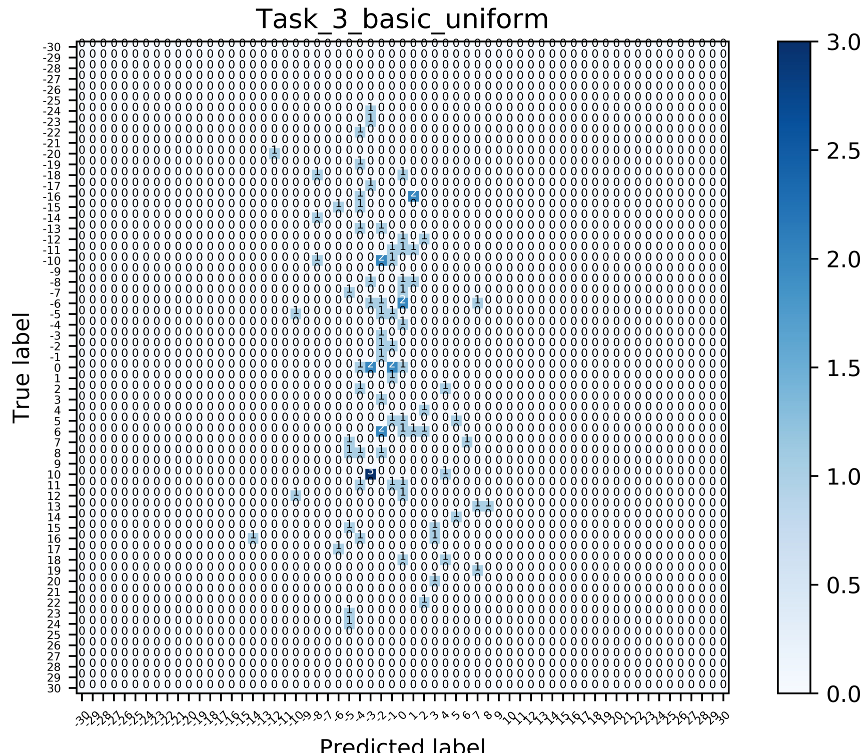

Figure 44: Confusion matrix for the resulting NVIDIA network evaluated on the uniform artificial data set. The set is uniform in the sense that all steering angles (labels) between -25°and 25°are represented equally many times in the set. The rows represent correct steering angles (true labels) and the columns the angles set by the network. An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. The resulting confusion matrix is obviously not diagonal and doesn't show any signs of being close to it either. This suggests that the evaluated network is deficient at predicting correct steering angles and ultimately steering a car in the simulator.

Multi-task design Here the results for the multi-task design on task 3 are presented.

The following hyperparameters were used:

Dropout rate layer 1:	0.23
Dropout rate layer 2:	0.91
Learning rate :	0.001

Which were applied along with the following hyperparameters:

Number of epochs:	13
Samples per epoch:	44 252
Batch size:	92
Loss weights [autoencoder loss, steering angle loss]:	[0.2, 1]

With these settings the model reached the following losses:

Loss:	0.142
Validation loss (uniform):	0.16
Performance in simulator:	Drove several laps before stopped manually.

Where loss refers to the loss on the training data and validation (uniform) on the uniform artificial data set. The progress of the losses are plotted in Figure 45 and the confusion matrix in Figure 46

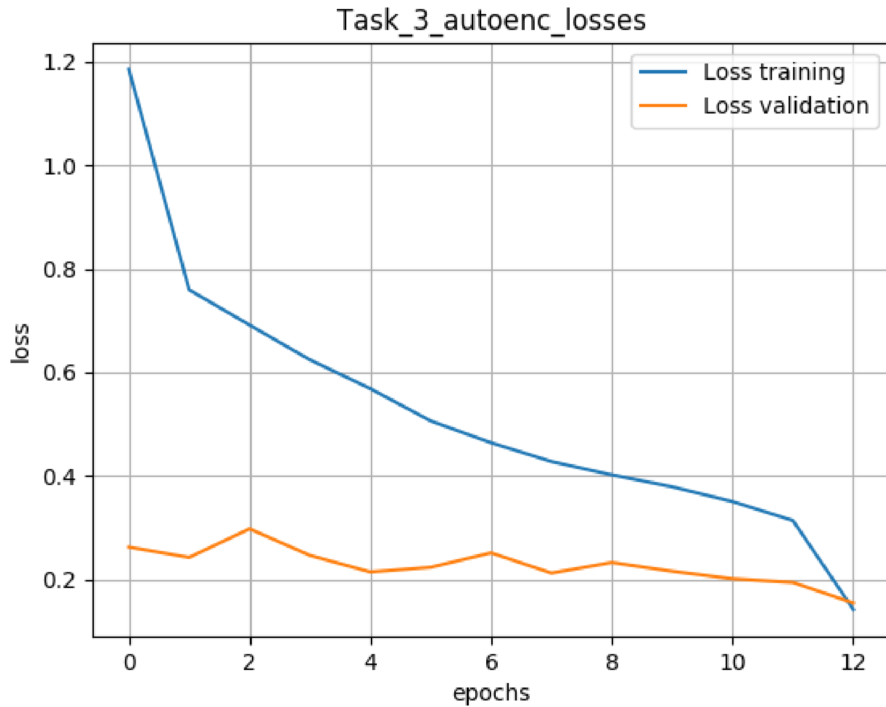

Figure 45: Training loss and validation loss (mean squared error) for the trained multi-task network in task 3. The training loss is with respect to the commaai data set, augmented for increased quality. The validation loss is with respect to a uniform subset of the artificial data set. The subset is uniform in the sense that all steering angles (labels) between -25°and 25°are represented equally many times in the set. One epoch is when 44 252 images have been passed through the backpropagation algorithm. Both the training loss and validation loss decrease during the whole training period (compare Figure 43. That is a good sign and indicates that neither under- or overfitting has occurred. Note that the validation loss is lower than the training loss up until the last epoch. This is probably due to heavy dropout regularization applied.

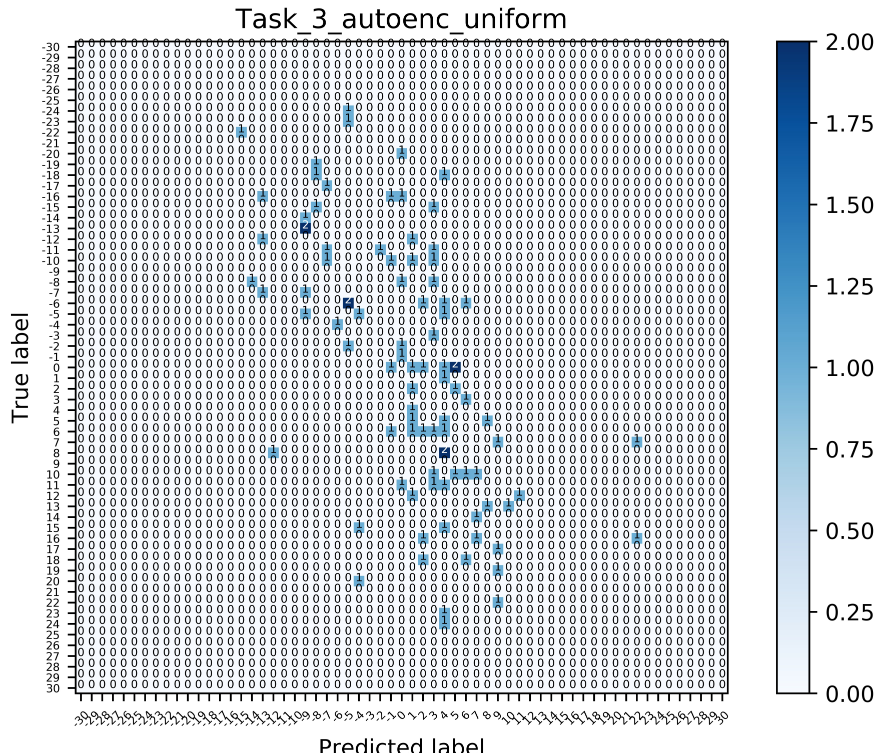

Figure 46: Confusion matrix for the resulting multi-task network evaluated on the uniform artificial data set. The set is uniform in the sense that all steering angles (labels) between -25°and 25°are represented equally many times in the set. The rows represent correct steering angles (true labels) and the columns the angles set by the network. An optimal confusion matrix is diagonal and the closer a confusion matrix is to being diagonal the better. The resulting confusion matrix displayed is not diagonal in the mathematical sense but it has tendencies towards being so. Especially compared to the confusion matrix in Figure 44. The pattern in the matrix displayed suggests that the network evaluated is better at prediction steering angles in the uniform artificial data set than the one in Figure 44.

7 Discussion

7.1 Data

Color channels As mentioned in the implementation section some experimentation was done with using different image formats and color channels. The most common format is RGB. The format used in this thesis is YUV, solely based on that *NVIDIA* did it in their paper. Why they use YUV is not mentioned in their paper [15] and it is not obvious that it's the best choice. On the other hand they don't deal with domain adaption, where the choice of format might be more important. It is tempting to convert the images to gray-scale both in training and testing. Intuitively, features that a steering predictor should consider would probably not get lost in such a

conversion and the features might also be more similar between different domains. For example in Task 3 where the training data is from real-life driving and the validation data from the simulator. In this case features like road curvature and lanes might differ substantially in color but not in gray-scale. On the other hand, other important information about objects/surfaces might get lost. For example can water and asphalt be very similar in gray-scale, which might have devastating consequences.

Normalization with respect to 95 percentile As mentioned in the implementation section (5.3) brief attempts were made to normalize the input data with respect to the 95 percentiles of the input image. Even though no significant improvements in the training could be seen it can not be concluded that the method doesn't have potential. Far more tests and evaluation has to be done before such a conclusion can be done.

Commaai data The original full commaai data set was a bit noisy and contained a lot of, for the scope of this project, irrelevant data. Which was mentioned in section 5.1.3. After the sorting that was conducted the quality of the data set can probably be regarded as relatively good. Even though more is always better in the case of supervised learning the quantity of the commaai data set is not bad. After the sorting it was about 15GB, which corresponds to about 100 000 images.

Artificial data The artificial data set on the other hand was insufficient both in quality and quantity. As mentioned in section 5.1.3 a too little fraction of the set contained images corresponding to curves. In addition it contained images of the car driving of the road, which is counterproductive to the learning process. The number of such images was very small though so it shouldn't affect too much.

The sheer number of images in the set, 3767, was small. Especially if the set is used for training as in task 1. If it is only used for validation/testing as in task 3 the number of images should be sufficient. Notable is that 3767 is the number of unique images if the images from the same point in time but from the left, center and right cameras (see section 5.1.1) are considered unique. This is a generous interpretation since these images are very similar in many ways. The images in the set was not even from one full lap in the simulator. The small size of the data set could be noticed in some cases when networks that had lower validation loss performed worse when evaluated in the simulator than networks with higher validation loss. I.e. the data set didn't represent the situations/images that the network might face in the simulator. Several models that have performed well on task 1 have had training sets corresponding to several laps in the simulator.

Data modification Augmentation of the training data proved beneficial for all tasks. Improvements in performance measures could be seen in general when different kind of augmentation was utilized (see section 4.6.4). The same was true for over/undersampling.

When it comes to augmentation of validation data it is a more complicated issue to decide what's best. For task 3, where the ultimate goal was to perform well in the simulator, the validation set used was basically uniform with respect to number of images corresponding to each steering angle between -25°and 25°. This, since such validation set showed to best reflect the performance in the simulator. On the other hand, in task 2 where no simulator was available several sets were used as validation sets in order to try to estimate a models performance. In any case a validation set should have a distribution that somewhat spans all possible "classes", in this case steering angles. Otherwise the network might reach good validation loss despite being very bad in reality. This

can be the case with very biased sets (see section 4.6.3). If the validation set consists mainly of steering angles in a very narrow range, the validation loss might be low for a network that puts out a constant value as steering angle within that range. This phenomenon was observed in some cases during the project.

7.2 Task 1

The validation loss reached in task 1 is not very impressive (0.036) compared to the recorded ¡ 0.01 by Ibrahim in [37]. In addition, the network used for task 1 had 251 219 trainable parameters compared to 119 711 in Ibrahim's network, which is substantially larger. Yet, the model manages to drive around the track in the simulator without driving of the road, which was the ultimate goal. Notable is though that in the simulator a restriction on the speed of the car was set to 15 mph. At higher speeds the model might not be able to stay on the road. In Figure 28 the loss is initially higher than the validation loss. This is in part due to that the loss that is plotted after every epoch is an average of the loss after each batch during the epoch. Whereas the validation loss is calculated for a subset of the validation data only on the resulting network after the first epoch. Also, during training dropout is used and a thinned out network is used whereas when validating the full network is used.

Task 1 was designated very little time compared to 2 and 3 since it was not as interesting as the other two. Examples of networks that perform well on task 1 can be found in multitude online and there is no doubt that it's possible to train an artificial neural network to perform well on task 1. Task 1 was mostly used as an introduction to the programs and packages used as a preparation for task 2 and 3. This is also the reason for less performance measures in task 1 than in 2 and 3. For example are no confusion matrices plotted for task 1.

Conclusion For the purpose of steering a car around the track in the simulator with the speed restriction of 15 mph, the network trained in task 1 is sufficiently good.

7.3 Task 2

The approach taken for task 2 described in 6.3 was to optimize the hyperparameters with respect to the validation loss for the NVIDIA design and then use the same hyperparameters for the multi-task network. This might seem like a partial and biased approach, since the optimization was only done on the NVIDIA network. There were three reasons for that; One, hyperparameter optimization is a very time consuming process and the larger the network the more time is needed. Since the multi-task network was much larger than the NVIDIA the optimization took much longer. Two, when brief optimizations on the multi-task network was done they yielded very similar results to the ones on the NVIDIA. Hence, the motivation to run very time consuming optimizations on the multi-task networks were low. Three, when conducting tests for task 2.1 and 2.2 other dropout rates than the ones calculated in the optimization were used since they yielded better results. Which means that the values calculated by *Hyperas* in the optimization were not as optimal as one would have hoped. Still, the optimization gave values in the right ballpark that turned out to be useful. Especially in task 3.

Task 2.1 Task 2.1 worked much as a reference to task 2.2 and 3. The idea was to see if the design and features of networks performing well on task 2.1 were the same as in task 2.2 and 3. Or could

any patterns of differences be seen between networks performing well on the different tasks. If task 2.1 was left out and only 2.2 and 3 were carried out, then (if successful) conclusions about which aspects and features that is best for domain adaption could be made. However, whether these aspects and features were beneficial for generalizing in general would still have to be investigated. Hence, the motivation for carrying out task 2.1 even though task 2.2 and 3 were the main objective of the project.

The validation losses for the NVIDIA design is lower than for the multi-task design (both for the test set and the modified test set). The validation loss is about 10 times lower for the NVIDIA design, whereas it is about the same on the modified test set. However, the validation loss curves for the two models (Figure 29 and 32) fluctuates substantially, This is due to that the training algorithm chooses different subsets every time from the test set that it validates on, Anyway, this fluctuation means that the values presented in the tables are not very reliable. In this case the values in the tables are from the last epoch. If the values instead were taken from the second or third last epoch the validation loss for the NVIDIA network might be about 0.13 and for the multi-task about 0.03. In that case suddenly the multi-task network outperforms the NVIDIA. With this in mind no major conclusions about the performance of the networks should be made solely based on the validation losses.

The respective confusion matrices on the other hand definitively indicate that the NVIDIA network outperforms the multi-task. The confusion matrix from an optimal network would be diagonal. The discretization of the steering angle range made in this project (whole degrees from -30 to 30) is relatively strict. For example, a predicted steering angle of -13 is still good even if the correct angle is -11. Especially since the correct steering angles from the data set is based on a human driver and as such you can't expect it to be the perfect steering angle. Anyway, a confusion matrix that is close to diagonal is typically a sign of a good network. If the confusion matrices from the two designs are compared it is clear that the ones from the NVIDIA design are closer to being diagonal than the ones from the multi-task design.

Conclusions Based on the major difference in the appearances of the confusion matrices for the two design it's valid to say that, to the extent of what was tested in this project, the NVIDIA design outperforms the multi-task design for task 2.1.

Task 2.2 Task 2.2 was interesting in the sense that the training and validation data sets differed distinctively in distribution, which is called covariate shift (see 4.9). The results from task 2.2 are ambiguous regarding which design on the network that performs best. If the validation loss and the validation loss (selective) are compared between the two models the NVIDIA design clearly reaches lower losses. For the validation loss (uniform) the two models have similar values, even though the multi-task model reached a lower value. Once again though, as can be seen in the plots over the loss and validation loss (uniform) in Figure 35 and 39, the validation loss (uniform) fluctuates substantially. Thus, the difference between the validation losses (uniform) presented in the results (0.31 vs 0.27) can not be considered significant. With only the different validation losses in mind it seems indeed like the NVIDIA design outperforms the multi-task design on task 2.2.

On the other hand, if the different confusion matrices for the two designs are compared it is not so clear which model that performs best. The confusion matrices from the NVIDIA model for validation loss and validation loss (modified) looks pretty much like clusters around 0 and not at all diagonal. The corresponding confusion matrices for the multi-task model doesn't look much better.

They are also pretty much clusters around zero, even though they are more oval horizontally . This indicates that the multi-task model is more prone to predict larger steering angles in magnitude.

The confusion matrices with regards to the validation loss (uniform) however seem to indicate that the multi-task model are superior. Whereas the NVIDIA model predicts most angles in the uniform data set as 0 and looks much like a vertical stripe, the confusion matrix for the multi-task model has tendencies to being diagonal. The tendencies are very subtle though and to call it a good result would be to exaggerate.

A noteworthy observation is that for the NVIDIA design, the validation loss (selective) is lower in task 2.2 than in 2.1. Yet, it is obvious from the respective confusion matrices that the model trained in task 2.1 is much better than the model trained in task 2.2. This indicates how important it is with different/reliable performance measures and how the validation loss can be insufficient as such.

Conclusions Based on the paragraphs above and the ambiguous results it can't be decided which design that handled task 2.2 best in the scope of this project. It is safe to say though that neither of the designs performed particularly well on the task, which suggests the difficulties with domain adaption. The designs (at least the NVIDIA design) performed substantially better on task 2.1 so the design is definitely capable of predicting steering angles when the validation set is similar to the training set. Basically the only difference between task 2.1 an 2.2 is the data sets so the deterioration in results are most certainly due to the problem of domain adaption.

7.4 Task 3

Since the simulator was available as a performance measure in task 3, only the validation loss on the uniform set is presented in the results. The choice of the uniform set was made since it was the best indicator of how the model performed in the simulator, which was the ultimate performance measure. I.e. during the work with task 3 it seemed like the validation loss (uniform) indicated performance in the simulator pretty well, compared to validation loss and validation loss (selective). This is probably due to the large fraction of images of curves in the uniform set compared to the others. Curves which were the main problem for models driving in the simulator. Hence, a network that could predict curves well was crucial in order to stay on the road in the simulator.

The multi-task model reached a lower validation loss (uniform) than the NVIDIA model in task 3. Unlike in task 2.2 it is the authors belief that it is valid to say that the difference in validation loss (uniform) is significant. Because if the graphs for the two validation losses (uniform) (Figure 43 and 45) are compared one can see that the validation loss (uniform) for the multi-task model is basically constantly lower than for the NVIDIA model. For the NVIDIA model it fluctuates considerably but barely goes below 0.3. Whereas for the multi-task model it constantly stays below 0.3.

Both confusion matrices looks quite scattered but the one from the multi-task model is closer to being diagonal (which is desirable). This is in accordance with the difference in validation loss aforementioned. Such an accordance is not obvious though as has been pointed out in earlier sections.

The difference in performance in the simulator is also in accordance with the difference in validation losses (uniform) and confusion matrices. I.e. the multi-task model outperforms the NVIDIA. Even if it drives slowly and sometimes are close to driving off the road, the multi-task

model manages to drive around the track. Unlike the NVIDIA model that drives off in the first sharp curve.

Conclusions To the extent of what have been tested in this task it is safe to say that the multi-task design is superior to the NVIDIA. In all three performance measures it yields better results. When it comes to driving the car in the simulator the multi-task model in task 3 is in parity with the model from task 1. That is very interesting considering that the model in task 1 was trained on data from the artificial data set, whereas the multi-task model in task 3 was trained solely on images from the commaai data set.

7.5 General conclusions

In the classic approach of training a steering predictor on data from the same distribution as the test data, a NVIDIA based design can certainly be successful. This is mainly confirmed by their own paper [15] but also by the results from task 1 and 2.1 in this thesis. There is nothing in the results from this thesis that indicates that a multi-task design would be better in such settings.

The results from task 2.2 and 3 suggest that the multi-task design (Figure 24) tested in this thesis is better at domain adaption than the NVIDIA design tested (Figure 23). However there is a huge number of ways to alter the designs and to say that a multi-task network is always better at domain adaption is not legitimate based on the results in this thesis. Yet, the results are interesting and appeals for further work on the subject

7.6 Further work

The results from especially task 3 is indeed interesting and further work within the area is welcome. More research is needed in order to make a certain conclusion about how well multi-task networks such as the one tested in this thesis versus classic convolutional neural networks (which the NVIDIA design is a version of) handle domain adaption.

More work is needed to conclude whether the difference in performance in task 3 depends on the actual design or not. One explanation to the difference can be the different levels of regularization. Note that in task 3 the NVIDIA network uses the dropout rates 0.01 for both dropout layers, whereas the multi-task network uses 0.23 and 0.91. Remember that here the dropout rate means the probability that a node in the layer is dropped. I.e. a dropout layer with 0.91 is a heavy regularizer compared to a layer with 0.01. In addition the decoder part of the network in the multi-task network has a regularizing effect on the steering angle predictor (see 4.7.10), which means that the multi-task network has even more regularization. Furthermore, the decoder output is an image with pixel values between 0 and 255 compared to the steering angle that usually is between -1 and 1. With the loss weights set to 0.2 and 1 respectively for the decoder and steering angle losses the ratio between the two losses will be about $255 \cdot 0.2 : 2 \cdot 1 = 51 : 2$. This means that in the backpropagation algorithm the loss from the decoder will be about 25 times larger than the loss from the steering angle predictor. So in task 3 there is no doubt that the multi-task network is more regularized than the NVIDIA network. A sign of the different levels of regularization is the loss and validation loss (uniform) curves presented in the results. There, the loss curve for the NVIDIA model goes down during the whole training period and reaches a value of 0.02. The validation loss (uniform) on the other hand seems to have a downwards trend initially but soon it starts to fluctuate and stagnate around 0.4. A decreasing loss and stagnating validation loss is

usually a sign of overfitting. As a comparison the corresponding curves for the multi-task model both seem to have a downwards trend during the whole training period. Hence, potentially the difference in performance might be due to different levels of regularization rather than the different designs. Remember though that the hyperparameters presented in the results are the one that yielded the best results to the extent of what was tested. I.e. more regularization on the NVIDIA design was tested without improvement. Still, the testing conducted was far from exhaustive and it can not be ruled out that more regularization on the NVIDIA network would have given better results.

The decision to train networks on real-life data and test it on artificial was because the real-life data available (commaai) was much larger than the artificial and because the performance could be tested in the simulator. However, an even more interesting and perhaps useful approach (at least in the case of self-driving cars) would be to do the other way around. I.e. train on artificial data and validate on real-life. If such experiment were successful that would mean that networks could be trained on data that could be easily and cost effectively produced in huge quantities. Then the networks could be used for driving cars in reality. Actually, even with the data used in this thesis more tests can be conducted to confirm or dismiss the domain adaption difference noticed in task 3. For example, the day and night data in task 2.2 could be interchanged (i.e. train on night data and validate on day data). The training data in task 3 could be swapped against the day data or night data etc.

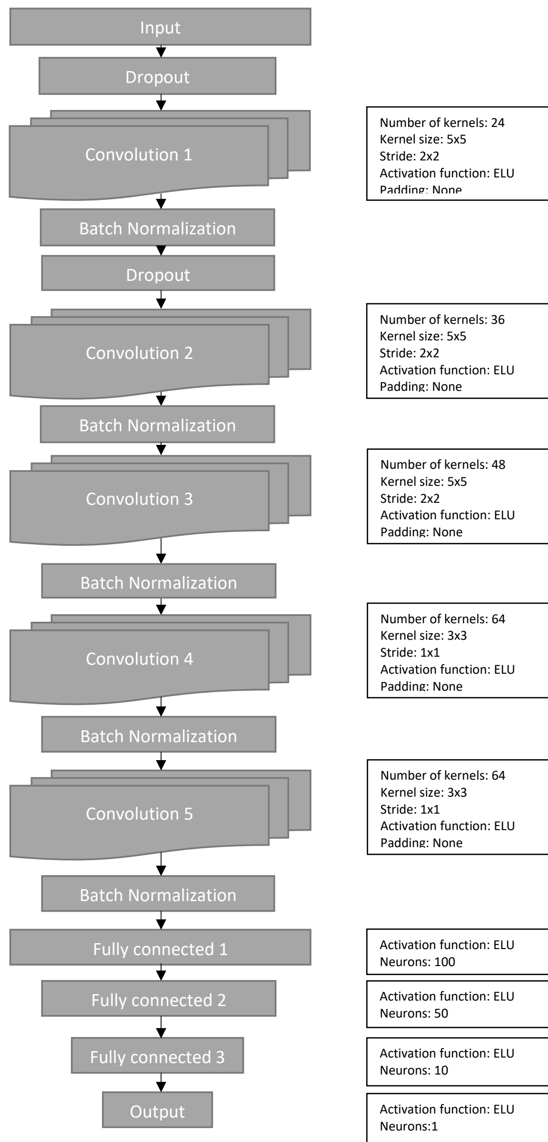

Figure 22: Network design for task 1

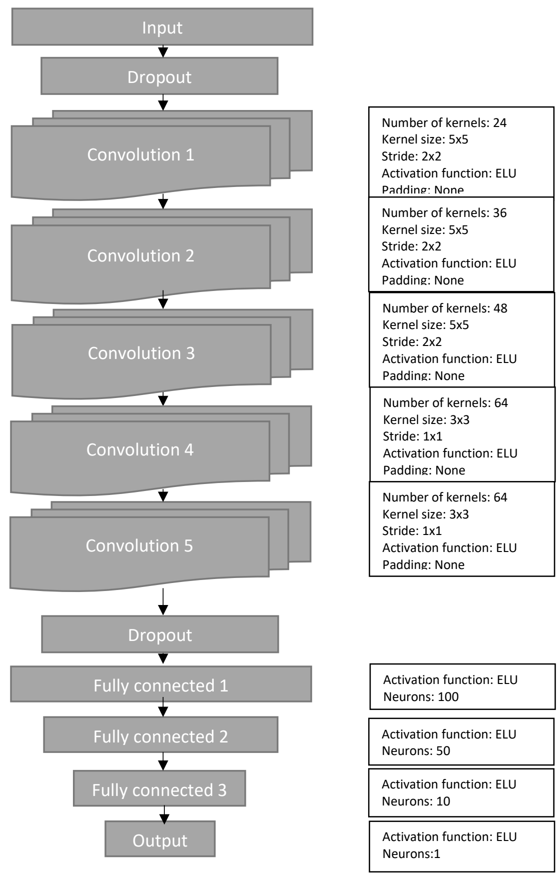

Figure 23: First network design for task 2 referred to as *NVIDIA* or *basic*

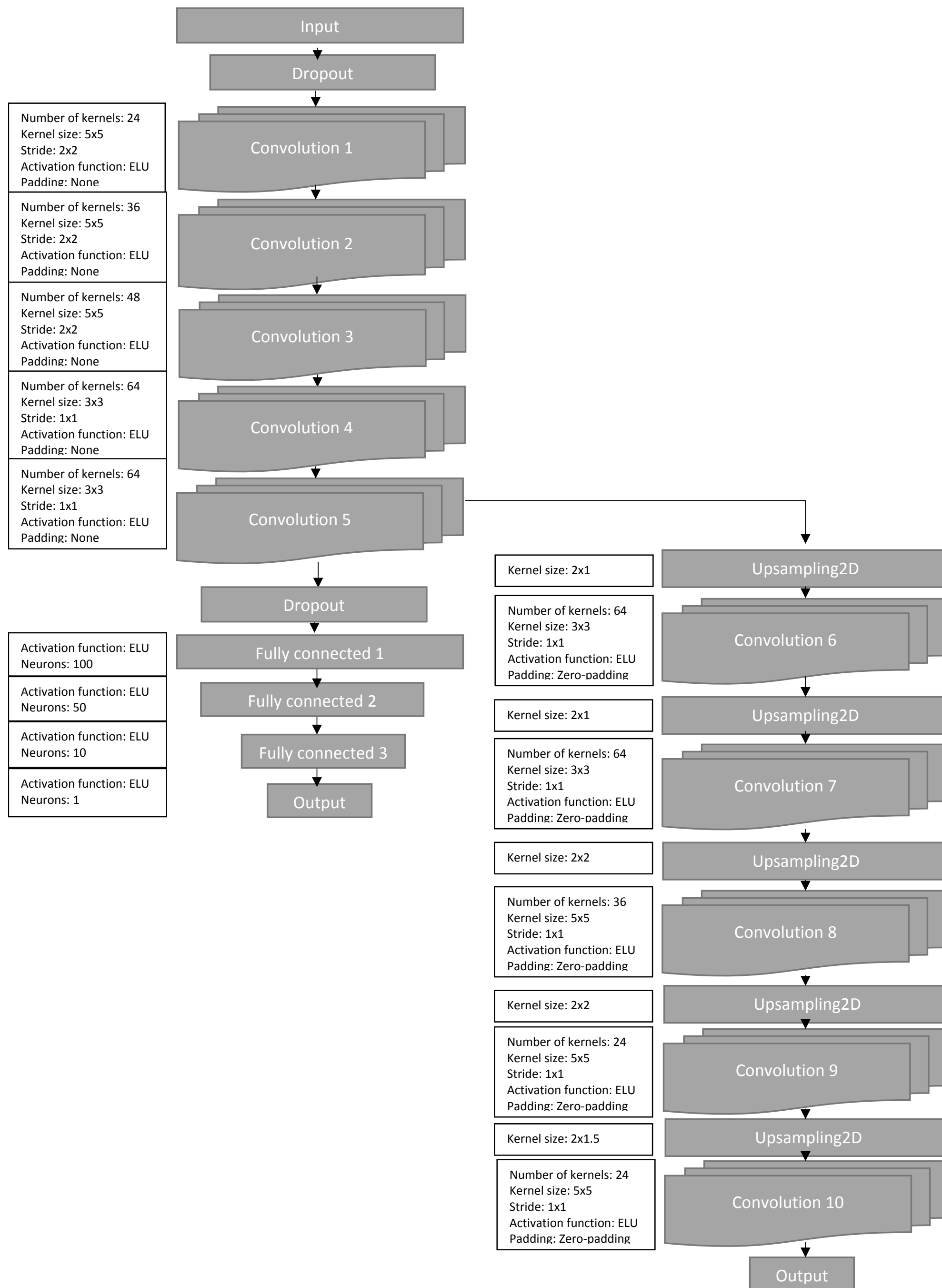

Figure 24: Second network design for task 2 referred to as *multi-task network* or *autoenc*

References

[1] Wikipedia. *Backpropagation.* Available from: *https://en.wikipedia.org/wiki/Backpropagation.* [Accessed: 2017-11-20].

[2] Wikipedia. *Stochastic gradient descent.* Avaliable from: *https://en.wikipedia.org/wiki/Stochastic_gradient_descent.* [Accessed: 2017-11-20].

[3] Pomerleau, Dean A. 1995. *Neural Network Vision For Robot Driving.* Carnegie Mellon University

[4] Wikipedia. *Moving average.* Available from: *https://en.wikipedia.org/wiki/Moving_average.* [Accessed: 2017-11-23].

[5] Diederik, Kingma; Ba, Jimmy. 2014. *Adam: A method for Stochastic Optimization.* arXiv:1412.698

[6] M. A. Nielsen. *Neural Networks and Deep Learning.* Determination Press. 2015. URL: *http://neuralnetworksanddeeplearning.com.*

[7] Simonyan, K. Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition.* arXiv: 1409.1556.

[8] Loffe, S. Szegedy, C. 2015. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* arXiv:1502.03167.

[9] Szegedy, C. 2014. *Going deeper with convolutions.* arXiv: 1409.4842.

[10] He, K. Zhang, X. Ren, S. Sun, J. 2015. *Deep Residual Learning for Image Recognition.* arXiv: 1512.03385.

[11] Wikipedia. *Neuron.* Available from: *https://en.wikipedia.org/wiki/Neuron.* [Accessed 2017-10-13].

[12] Haykin, S. 1998. *Neural Networks: A comprehensive Foundation.* Hamilton, Ontario Canada. Pearson Education. 2nd edition.

[13] Sonoda, S. Murata, N. 2015. *Neural Network with Unbounded Activation Function is Universal Approximator.* Faculty of Science and Engineering, Waseda University. Working paper arXiv:1505.03654.

[14] LeCun. Y, Boser. B, Denker. J. S, Henderson. D, Howard. R.E, Hubbard. W, Jackel. L.D. 1989. *Backpropagation applied to handwritten zip code recognition.* Neural Computation 1(4):541-551.

[15] Bojarski. M, Del Testa. D, Dworakowski. D, Firner. B, Flepp. B, Goyal. P, Jackel. L.D, Monford. M, Muller. U, Zhang. J, Zhang. X, Zhao. J. 2016. *End to End learning for Self-Driving Cars.* arXiv:1604.07316.

[16] Hubel. D, Wiesel. T. 1969. *Receptive fields and functional architecture of monkey striate cortex.* Journal of Physiology. London, 195, 215-243.

[17] S.C. class CS231n: *Convolutional Neural Networks for Visual Recognition.* Available from: *http://cs231n.github.io/convolutional-networks.* [Accessed: 2017-12-13].

[18] Glorot. X, Bengio. Y. 2010. *Understanding the difficulty of training deep feedforward neural networks.* In: Proceedings of the International Conference on Artificial Intelligence and Statistics (AIS-TATS'10). Society for artificial Intelligence and Statistics.

[19] He. K, Zhang. X, Ren. S, Sun. J. 2015. *Delving Deep in to Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* arxiv:1502.01852v1.

[20] Dang. J. *Classification in Bone Scintigraphy Images Using Convolutional Neural Networks* Master's Thesis, Center for Mathematical Sciences Lund University, Lund, 2016.

[21] Wikipedia. *Training, test, and validation sets.* Available from: *https://en.wikipedia.org/wiki/Training,_test,_and_validation_sets.* [Accessed 2018-01-10].

[22] Wikipedia. *Cross-validation (statistics).* Available from: *https://en.wikipedia.org/wiki/Cross-validation_(statistics)* [Accessed 2018-01-10].

[23] S.C. class CS231n. *Convolutional Neural Networks for Visual Recognition.* Available from: *http://cs231n.github.io/neural-networks-2/.* [Accessed: 2018-01-15].

[24] Wikipedia. *Human brain* Available from: *https://en.wikipedia.org/wiki/Human_brain.* [Accessed: 2018-01-20].

[25] Srivastava, N. Hinton, GE. Krizhevsky, A. Sutskever, I. 2014 *Dropout: A Simple Way To Prevent Neural Networks from Overfitting.* In *Journal of Machine Learning Research 15*, pages 1929-1958.

[26] Ruder. S. 2017. *overview of gradient descent optimization algorithms* arXiv:1609.04747

[27] Bengio. Y. 2012. *Practical Recommendations for Gradient-Based Training of Deep Architectures.* arXiv:1206.5533v2.

[28] Wikipedia. *Confusion matrix* Available from: *https://en.wikipedia.org/wiki/Confusion_matrix.* [Accessed: 2018-01-30].

[29] Wikipedia. *Regression analysis* Available from: *https://en.wikipedia.org/wiki/Regression_analysis.* [Accessed: 2018-01-31].

[30] Wikipedia. *Reinforcement learning* Available from: *https://en.wikipedia.org/wiki/Reinforcement_learning.* [Accessed: 2018-02-01].

[31] Ruder. Sebastian. *Transfer Learning - Machine Learning's Next Frontier.* Available from: *http://ruder.io/transfer-learning/.* [Accessed: 2018-04-02].

[32] Wikipedia. *Domain adaption.* Available from: *https://en.wikipedia.org/wiki/Domain_adaptation.* [Accessed: 2018-04-02].

[33] Ruder. Sebastian. 2017. *An Overview of Multi-task Learning in Deep Neural Networks.* arXiv:1706.05098v1

[34] Wikipedia. *Autoencoder* Available from: *https://en.wikipedia.org/wiki/Autoencoder*. [Accessed: 2018-04-18].

[35] Keras. *UpSampling2D* Available from: *https://keras.io/layers/convolutional*. [Accessed: 2018-04-20].

[36] Commaai. *Commaai research* Available from: *https://github.com/commaai/research*. [Accessed: 2017-12-15].

[37] Ibrahim, Nora. 2017. *Designing a deep learning network for self-driving vehicles*. Lund: Lund University.

[38] Chollet, François and others. *Keras-Applications-VGG16*. Available from: *https://keras.io/applications*. [Accessed: 2018-04-22].

[39] Chollet, François and others. 2015. *Keras*. Available from: *https://keras.io*. [Accessed: 2017-10-11].

[40] Udacity. *Udacity-self-driving-car-sim*. Available from: *https://github.com/udacity/self-driving-car-sim*. [Accessed: 2017-10-11].

[41] Maxpumperla. *Hyperas*. Available from: *https://github.com/maxpumperla/hyperas*. [Accessed: 2018-03-02].

Master's Theses in Mathematical Sciences 2018:E26
ISSN 1404-6342

LUTFMA-3349-2018

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/