Linköping University | Department of Computer and Information Science
Master thesis, 30 ECTS | Datateknik
202018 | LIU-IDA/LITH-EX-A--2018/034--SE

Evaluating Deep Learning Algorithms for Steering an Autonomous Vehicle

Utvärdering av Deep Learning-algoritmer för styrning av ett självkörande fordon

Filip Magnusson

Supervisor : Cyrille Berger
Examiner : Ola Leifler

External supervisor : Åsa Detterfelt

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

With self-driving cars on the horizon, vehicle autonomy and its problems is a hot topic. In this study we are using convolutional neural networks to make a robot car avoid obstacles. The robot car has a monocular camera, and our approach is to use the images taken by the camera as input, and then output a steering command. Using this method the car is to avoid any object in front of it.

In order to lower the amount of training data we use models that are pretrained on ImageNet, a large image database containing millions of images. The model are then trained on our own dataset, which contains of images taken directly by the robot car while driving around. The images are then labeled with the steering command used while taking the image. While training we experiment with using different amounts of frozen layers. A frozen layer is a layer that has been pretrained on ImageNet, but are not trained on our dataset.

The Xception, MobileNet and VGG16 architectures are tested and compared to each other.

We find that a lower amount of frozen layer produces better results, and our best model, which used the Xception architecture, achieved 81.19% accuracy on our test set. During a qualitative test the car avoid collisions 78.57% of the time.

Acknowledgments

I would like to thank MindRoad, and especially Åsa Detterfelt, for the opportunity to work on this thesis. I would also like to thank my supervisor Cyrille Berger and my examiner Ola Leifler at the university for the help and feedback I have recieved during the project.

Contents

List of Figures

List of Tables

1 Introduction

Autonomous vehicles have gathered a lot of interest lately. Several large companies are working on self-driving cars [28]. Many cars on the market now already have several autonomous features. Other vehicles such as drones and other robots are also becoming less reliant on human input. With development of self-driving vehicles comes also many problems that needs to be solved.

An autonomous vehicle must be able to adapt to many different situations, using the data available to it. By using its various sensors, the vehicle must scan the environment in order to make the correct decision at any given point. For example, if the vehicle finds an obstacle in its path, it must be able to avoid it. But how can the vehicle know what an obstacle look like, and what exactly to do when it encounters one? A possible answer to that question is through machine learning.

Machine learning has become more and more popular over the years. In broad terms, machine learning can be described as algorithms which learns from data to become better at some specific task [9]. Machine learning can be used for many different applications, such as speech recognition [6], image classification [15] and playing chess [26].

One machine learning model that has proven successful in many areas, in particular image classification, is *neural networks* [15]. Neural networks are inspired by how the neurons in our brains work. When such a network is large, it is called *deep neural network*, and becomes part of *deep learning*. For image processing in particular, one successful type of network is the *convolutional neural network*.

Using neural networks in robotics, both for perception and for decision-making, is an increasing trend as shown by Tai et al. [34]. Neural networks is having great success in areas such as object detection, environment recognition and steering.

In order to identify the obstacle the vehicle could be equipped with a neural network that is trained to recognize obstacles. By using a camera to take pictures, and then classifying the obstacles, the neural network could decide which action the vehicle should take to avoid collisions.

This method have been used previously by Tai et al. to make a robot equipped with a depth camera explore an indoor environment [33], and by LeCun et al. using a robot equipped with stereo cameras to explore an outdoor environment [16]. In this paper however only a single non-depth camera will be used to study the effectiveness of using limited sensors.

1.1 Motivation

Simultaneous Localization and Mapping (SLAM) is a well studied research area [19]. In SLAM a vehicle explores an unknown environment and simultaneously maps the area and localizes its own position in that area. In their paper, Mur-Artal and Tardos presents a SLAM system called ORB-SLAM2. They do not focus on the exact navigation of the robot, but instead use a prerecorded image sequence for their training and evaluation.

In this study we will look at obstacle avoidance, which is part of the navigation of the robot. Such a system could be used together with SLAM in order to handle both steering and localization. However, we want to study the accuracy of various neural network architectures when applied to obstacle avoidance, so we have chosen to limit the scope of this study to that task only.

MindRoad

MindRoad is a company that works with development of embedded systems and interactive applications. The company also offers education in areas such as embedded systems and agile development. Now MindRoad wants to know more about the possibilities and limitations of neural networks when applied to obstacle detection and avoidance.

MindRoad has a small autonomous robot car equipped with a camera. A previous thesis work investigated the possibility of obstacle avoidance with the help of a neural network. The neural network analyzes the images taken with the camera and classifies it according to which action the car should take to avoid possible obstacles in the image. If no obstacles are detected the car should continue to drive straight ahead.

MindRoad now wants to look at other possible neural network implementations. The currently implemented network is able to identify and avoid obstacles around 70% of the time. By implementing other networks and evaluating them both against each other and the original implementation, the usability of neural networks in obstacle avoidance can be further studied.

General Application

From a general viewpoint, the result of this study can potentially be relevant for any autonomous robot or vehicle that needs to avoid possibly unknown obstacles in its path when not using many different sensors. It also serves as a comparison of a few different neural network architectures.

1.2 Aim

The aim of this thesis is to look at different neural network architectures and compare their accuracy in image recognition and real-time obstacle avoidance. This will be done using MindRoad's autonomous robot car. The resulting network will take images as input and output a steering command that the car should follow in order to avoid any obstacles. If there are no obstacles the car should keep driving forward. In order to do this the network must learn some kind of depth estimation to be able to know the difference between a close obstacle and the far-away background. As the data set needs to be very large when training a neural network, transfer learning will be used to transfer the knowledge from another domain. This is to limit the amount data we will need to collect, since doing so can be hard and time-consuming. This thesis builds upon the paper *Deep Learning for Autonomous Collision Avoidance* by Oliver Strömgren [31]. Here we will use the same basic equipment and perform similar experiments as Strömgren and attempt to find better results by implementing and evaluating other models.

1.3 Research questions

1. What convolutional neural network architectures are most suitable for simple obstacle avoidance?
2. How high accuracy can different neural networks reach when applied to obstacle avoidance?
3. Does transfer learning work well when the target classification problem is very different from the source problem and the target domain has much less training data?

1.4 Delimitations

The neural networks will only be evaluated using MindRoad's small autonomous robot car and the obstacles that it encounters in an indoor office environment.

2 Theory

In this chapter, convolutional neural networks will be described. To be able to understand how they work, first machine learning in general and non-convolutional neural networks will be presented.

2.1 Machine Learning

Machine learning algorithms are a set of algorithms that can learn from experiences [9]. Machine learning methods are commonly divided into supervised learning and unsupervised learning.

In supervised learning, the system is given pairs of input and output (x,y) during training. These are related by some unknown function $y = f(x)$. The system tries to learn f, usually by estimating $p(y|x)$. If successful, the system can then calculate y from a previously unseen x. For example, in the domain of image classification, the input to the system would be images, and the output would be a label that the system tried to learn, like 'cat' or 'dog'. After seeing many labeled images the system would begin to learn the difference between cats and dogs. When the training is complete, the system can be given a new, previously unseen, image and (hopefully) correctly label it as cat or dog.

In unsupervised learning, there is no known output. The system is only given (x) during training. The idea is that the system will then find rules or patterns in the data that was previously not known.

There are also other types of machine learning algorithms. One of them are reinforcement learning, which are based on actions and rewards. The system will take actions that interacts with the environment. Its goal is to find a set of actions (or sequences of actions) that maximizes a reward function.

Machine learning problems can also be further categorized into regression problems and classification problems based on the expected output. In regression problems the output is some real-valued number, such as the distance to an object or the price of an item. In classification problems the output is a discrete value, like an image label or a steering command.

2.2 Artificial Neural Networks

Artificial neural networks (ANN), or just Neural networks (NN), is a popular technique within machine learning [9]. The method is inspired by how biological brains work. It is however important to note that neural networks do not intend to perfectly mimic or model the brain, but only to borrow some insights from what we know about it. Neural networks, like their biological counterparts, are made from several computing units called neurons. Each neuron has several inputs and outputs. The neurons are connected to each other so that the outputs of a neuron forms the input of other neurons. The input of a neuron is typically formed from the output of many other neurons. Each connection between neurons is accompanied with a weight, which essentially tells the neuron how important this connection is. When a neural network "learns", these weights are what changes in value. When calculating the output of a neuron, first each input is multiplied with its corresponding weight and summed together, as shown in equation (2.1). After this sum is calculated, a function is applied to the result, as shown in 2.2. The bias term b is a weight connected to the neuron itself, and not any of its connections. The function used is called an *activation function*.

$$x = \sum w_i \cdot x_i \tag{2.1}$$

$$y = f(x + b) \tag{2.2}$$

In a feed forward neural network, the neurons are structured in layers, with one input layer, one output layer and zero or more hidden layers. The output of each layer forms the input of the next, see figure 2.1 for an illustration. The input layer is where the input values of the network is sent in, and varies with the task. The idea is then that each layer creates its own representation of the data, with deeper layers finding more complex relations between different parts of the input. If the network is correctly trained on the task, the representation in the final layer would be the expected output for your task, like an image label or a distance.

In classification tasks, the output layer have as many neurons as there are classes. The output is then often fed to a softmax classifier which gives the probability that the data belongs to each class [15, 36].

The softmax classifier is used to rescale the output values so that they sum to 1. In this way the output y_i can be interpreted as the probability of the input belonging to class i. The softmax classifier for K different classes is shown in equation 2.3.

$$f(z_j) = \frac{e^{y_j}}{\sum_{k=1}^{K} e^{y_k}} \text{ for } j = 1, ..., K \tag{2.3}$$

2.3 Activation Function

The activation function is important to introduce non-linearity in the network. If we had no activation function, or if we had only a linear activation function, the network would only be able to model linear relations.

Some possible activation are the hyperbolic tangent function and the sigmoid function, shown in equations 2.4 and 2.5.

$$f(x) = \tanh(x) \tag{2.4}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

The choice of activation function can greatly affect the time it takes for a neural network to train. Krizhevsky et al. showed that using the rectifier, as shown in equation 2.6, can make the networks train several times faster than using the hyperbolic tangent as activation function [15]. Neurons using the rectifier function are usually called Rectified Linear Units (ReLUs)

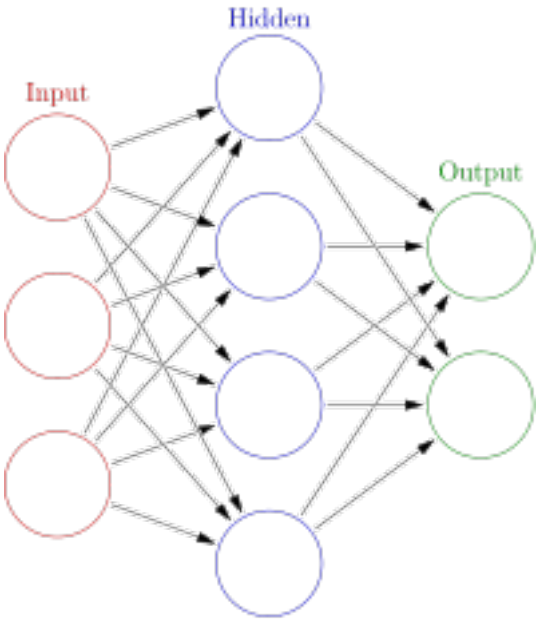


Figure 2.1: The structure of a neural network.[1]

and sometimes the function itself is called ReLU. Unlike the hyperbolic tangent function and the sigmoid function, the rectifier is simple to compute, and it is also easy to calculate its gradient, which will be useful when training the neural network.

$$f(x) = \max(0, x) \tag{2.6}$$

The rectifier is currently the most common activation function used in neural networks [22].

2.4 Training

The performance of a specific neural network model depends entirely on weights of its connections [9]. In order to obtain optimal results, the optimal weight values must be found. In supervised learning, this is done by comparing the output of the network to the expected output, or label, for the current input. The weights are then modified in attempt to minimize future errors. There are several ways to change the weights, and one of the most commonly used is called *stochastic gradient descent*, which is a modification of *gradient descent*.

Loss function

To know how well a model performs, the loss function is used. It is sometimes called the objective function or the cost function. The purpose of the loss function is to measure the error of the output when comparing to the expected output. For classification problems having more than two classes, the categorical cross entropy function is often used, which can be seen in equation 2.7, where y' is the output and y is the expected output of each neuron [35].

$$z(\mathbf{y}, \mathbf{y}') = -\sum_{i=1}^{n} y_i \ln y_i' \tag{2.7}$$

[1] By Glosser.ca [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

Gradient Descent Optimization Algorithms

In gradient descent the weights are updated according to the gradient of the loss function with respect to each weight, as seen in equation 2.8, where w_t is the weights at time t, z is the loss function and η is the learning rate [35].

$$w_t = w_{t-1} - \eta \frac{\delta z}{\delta w_{t-1}} \tag{2.8}$$

The learning rate decides how much the weights should change after each update [13]. It is usually set to a small number like $\eta = 0.001$. Sometimes the learning rate is decreased over time or after the accuracy has stopped increasing.

In gradient descent, all training samples are evaluated before updating the weights. This is called an *epoch*. This leads to an optimal weight update because all samples are taken into account during the update. However it also takes a long time between each update, and so training using this method is very slow.

Stochastic Gradient Descent

A faster alternative is to divide an epoch into smaller parts, or *batches*. This is *stochastic gradient descent*, or SGD [2]. Batch sizes might be between 1 and a few hundred, with 32 being a good default value. Updating the weights after each batch leads to a lot more than just one update per epoch.

There are a few challenges with using SGD, such as the difficulty of choosing a good learning rate [23]. Another issue is that SGD might sometimes get stuck on a saddle point when trying to find the global minimum. Some extensions to SGD have therefore been made in order to solve these problems.

Momentum

One such extension is to add *momentum* to the weight updates [23]. Momentum adds a fraction of the previous change in order to help steer SGD in the right direction and prevents oscillations. Momentum is shown in equation 2.9 where γ is the momentum term. γ is usually set to around 0.9.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \frac{\delta z}{\delta w_{t-1}} \\ w_t &= w_{t-1} - v_t \end{aligned} \tag{2.9}$$

Adagrad

Another issue with SGD is the problem of finding the best learning rate. If the learning rate is too small, it might take a long time to find the optimal weights, while a too large learning rate can make the algorithm update in too large steps and continuously 'miss' the minimum. On top of this, different parameters might need a different learning rate depending on their importance. There are some methods that solves this by introducing an adaptive learning rate.

Adagrad uses a learning rate that adapts to the parameters [23, 38]. Infrequent parameters have larger updates while frequent ones have smaller updates. The update rules for Adagrad is shown in equation 2.10, where we use $g_{t,i}$ to denote the gradient of the loss function with regards to w_i at time t. The denominator here is the square root of the sum of the previous

gradients. ϵ is a smoothing term to avoid division by zero. It should be a small number usually on the order of 10^{-8}.

$$\begin{aligned} v_{t,i} &= -\frac{\eta}{\sqrt{\sum_{j=1}^{t} g_j^2 + \epsilon}} g_{t-1.i} \\ w_{t,i} &= w_{t-1,i} + v_t \end{aligned} \tag{2.10}$$

The advantage of using Adagrad is that there is no need to tune the learning rate manually. Usually a default value of 0.01 is used.

Its weakness is that the denominator contains all the previous squared gradients. As this sum grows, the learning rate becomes smaller and smaller and eventually the algorithm will be unable to learn anything new.

Adadelta

In 2012 Zeiler presented Adadelta as an extension to Adagrad to fix the weakness of the growing denominator [38]. The idea is to instead use the average of the previous w squared gradients. However, storing many gradients in inefficient, so an approximation of this is to use a decaying average as shown in equation 2.11, where ρ is a decay constant.

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2 \tag{2.11}$$

In Adagrad we used the square root of the sum, and by taking the square root of this average we obtain the RMS (Root Mean Square), as seen in equation 2.12. The resulting parameter update is then shown in equation 2.13.

$$\text{RMS}[g]_t = \sqrt{E[g^2]_t + \epsilon} \tag{2.12}$$

$$\begin{aligned} v_t &= -\frac{\eta}{\text{RMS}[g]_t} g_t \\ w_t &= w_{t-1} + v_t \end{aligned} \tag{2.13}$$

Zeiler also noticed that the units of the parameter updates did not match when considering SGD, momentum or Adagrad. In order to make the units match, the learning rate η is replaced with the RMS of the parameter updates at time t. Since $\text{RMS}[v]_t$ is unknown before it is updated it is approximated with the RMS at time $t-1$. The full Adadelta update rules is then shown in equation 2.14. With this replacement, Adadelta does not even need a starting learning rate.

$$\begin{aligned} v_t &= -\frac{\text{RMS}[v]_{t-1}}{\text{RMS}[g]_t} g_t \\ w_t &= w_{t-1} + v_t \end{aligned} \tag{2.14}$$

Adam

Kingma presented Adam, another method that uses adaptive learning rates for each parameter [14]. Adam uses a decaying average of past squared gradients like Adagrad, as well as a decaying average of past gradients, as shown in equation 2.15

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{2.15}$$

As m_t and v_t are initialized to 0, they are biased towards 0 during the initial updates. Therefore a bias-corrected m_t and v_t are computed as shown in equation 2.16

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{2.16}$$

These are then used to create the Adam update rules as shown in 2.17. Kingma proposes the default values of $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{2.17}$$

Weight Initialization

Before any updates are done to the weight values, they must be initialized to some starting values [13]. One naive way to initialize them is the set them all to the same value, like 0. Since we cannot know what values the weights will end up with, and the weights can be positive or negative, setting it to zero seems like a good starting guess. However, if they all have the same value they will also lead to identical gradient calculations and identical updates. Since every weight would be be the same the network would not be able to learn very much.

To solve this it is common to initialize the weights with small random values, to ensure diversity in the neurons but still not have any bias towards positive or negative values. Glorot and Bengio found an increase in performance using a random Gaussian initialization with mean 0 and variance based on the number of connections of a node [8]. This is called the Glorot initializer and can be seen in equation 2.18. There is also a uniform version seen in 2.19 which also scales with the number of connections. n_{in} and n_{out} is the number of input connections and the number of output connections respectively.

$$W \sim N(0, \frac{2}{n_{in} + n_{out}}) \tag{2.18}$$

$$W \sim U(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \sqrt{\frac{6}{n_{in} + n_{out}}}) \tag{2.19}$$

The biases are usually initialized with 0, since the symmetry issue is resolved by having the connection weights randomized [13].

Backpropagation

To be able to update all weights one must then be able to calculate $\frac{\delta z}{\delta \mathbf{w}_i}$ for all layers. This is done by a process called backpropagation. It is reliant on the chain rule for derivatives, as shown in 2.20, where z is a function of y and y is a function of x.

$$\frac{\delta z}{\delta x} = \frac{\delta z}{\delta y}\frac{\delta y}{\delta x} \tag{2.20}$$

In neural networks the output from the final layer is can be written as a function of the weights and inputs of the second-to-last layer [9, 13]. That layer can in turn be described as a function of the previous layer and so on, until the input layer. Using this dependence on the previous layer one can then calculate the partial derivatives one layer at a time by using the chain rule. This is done by first calculating the derivatives for the last layer, and then for each layer multiply with the derivatives of the layer's output with respect to their input. This backwards iteration is why the method is called backpropagation.

Regularization

One important part in machine learning is to make a model that can generalize into unknown cases. Sometimes a model can fit perfectly on the data used for training, but work very poorly on new data, which makes the model not very useful. This is called *overfitting*. There are several methods that tries to minimize overfitting, and these are called *regularization*. [9]

To measure how well a model performs when given new data, the dataset is usually split into a training set and a test set [9]. The training set is used to modify the weights as described above, and the test set is an approximation of new unseen data to test the accuracy of the model. The data in the test set must be completely independent of the data in the training set.

Early Stopping

Generally the model's accuracy on the training data increases as training goes on. But at some point its accuracy on new data might decrease as the model gets too overfitted to the training data. To avoid this a simple method called *early stopping* can be used [9].

A new dataset called validation set is used and the model is tested on the validation test after each epoch. When the accuracy on the validation data stops increasing, the training stops. Since the accuracy does not always increase smoothly, the stopping usually occurs after there have been no increase for several epochs in a row. An alternative to stopping the training is to use the model at the highest validation accuracy. Each time the accuracy on the validation set increases, the weights at that point is saved. After training is concluded it returns the weights where the validation set had the highest accuracy, as opposed to simply using the latest weights.

Dropout

Dropout, as presented by Srivastava et al. [29] is another way to combat overfitting. When using dropout, some neurons are randomly "dropped out" during training. This means that the neuron is temporarily removed from the network. Each neuron has a probability p to be retained during each training case, where p is usually set to 0.5. A neural network with n neurons can be changed into 2^n possible thinned networks with many shared weights by removing different neurons. Generating and training that many networks is of course unfeasible, so dropout can be seen as a way to sample this collection of models and use a different network each time.

When testing the network after training, it would be too computation-heavy to test and average the predictions from exponentially many networks. A simple way to average all the models is then to simply remove the dropout part and use the whole network for testing. Since this network has more weights than the thinned versions used during training, each weight is multiplied with p to obtain a correct scaling.

Dropout discourages situations where some neurons are too dependent on a few other neurons [29, 15]. Instead each neuron must be useful on its own or with many subsets of neurons since it cannot rely on the presence of any specific neurons during training.

Dropout have been used successfully to improve the performance of neural networks and have been shown to greatly prevent overfitting as shown by Krizhevsky et al. and Dahl et al. [15, 6]. However a network using dropout takes around twice as long time to train.

2.5 Data Augmentation

Since a neural network tend to have very many parameters (weights), it needs a lot of training data to achieve good results. Sometimes the amount of data available is limited, or takes a long time to collect. Small changes in existing data could then be made to generate more data with minimal effort. This is called *data augmentation* [9]. For a classification task this is very useful if the changes does not change the class of the input. If the input data is images,

data augmentation could consist of mirroring the images, rotating them, scaling them and/or cropping them. Depending on the exact task, some care must be taken when doing data augmentation. If the task is to classify images, then a mirrored image of a house is still a house. But if the task is to read a text, then mirroring the image would not work.

Sometimes data augmentation can be used to generate data of different classes. Giusti et al. [7] used data augmentation to generate "turn-left"-images from "turn right"-images (and vice versa) by mirroring them.

Chen et al. [21] uses a similar technique to generate more data when training a system used in a self-driving car that estimates the distances to nearby lanes and cars. They were able to extend their dataset containing 14 000 images to about 62 000 images.

2.6 Transfer Learning

Instead of generating more training data, methods can be used to lower the amount of needed data. One way to reduce the number of training samples needed is to start with a previously trained network. If the network has learned similar categories to the ones needed, then much of that knowledge can be transferred to this new domain [1].

The first layer of neural networks that have been trained on natural images usually look very similar. Yosinski et al. [37] first trained a network on one dataset, then transfered some or all of the weights to a new network. Training this network on a new, different, dataset (but similar to the first one) resulted in an even better network than one that had only been training on the new dataset.

Transfer learning when used on a dataset that is less similar than the original was not too surprisingly shown to be less effective, but could still be very helpful if the new dataset is small [37]. Oquab et al. [20] trained a network on the large ImageNet database with over 1 million images, and then had success with the network on the much smaller VOC 2007 and VOC 2012 image data.

2.7 ImageNet

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual competition as well as a publicly available dataset [24]. The goal of the dataset is to provide a large image dataset for development and research of image recognition algorithms. The competition is a way to promote discussion and see the progress made each year.

The total number of images in the ImageNet database of about 14 million, and the number of images used in the ILSVRC competition is around 1.2 million [24, 30].

2.8 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of neural network which is particularly good at image related tasks [20, 15]. In a typical ANN, every neuron in a layer is connected to every neuron in the next layer. This leads to very many weights when the network is large, which makes it very difficult to train. In a CNN this is solved by introducing a new type of layer, the convolutional layer [17].

A CNN is usually divided into several separate parts, each with a different layout and purpose. There are many types of CNNs with special layers, but all of them usually has convolutional layers, pooling layers and fully connected layers.

Convolutional Layer

A convolutional layer consists of several filters, or kernels. Each filter is a matrix that is generally much smaller than the input matrix. The filter is moved across the input matrix and

the overlap is multiplied at each point to produce an output matrix [35]. A simple example with a $3 * 3$ input and a $2 * 2$ filter is shown in figure 2.2.

Mathematically, this is described by equation 2.21, where O is the output matrix, I is the input matrix, F is the filter matrix, and $k * l$ is the size of the filter [9].

$$O(i,j) = \sum_{m=1}^{k} \sum_{n=1}^{l} I(i+m, j+n)F(m,n) \tag{2.21}$$

For simplicity, this is shown using two-dimensional matrices, but both the input and the filters are usually three-dimensional. The reason for this is that many different filters are applied in each layer, and each filter produces its own output matrix which are then stacked to create a three-dimensional output matrix [13]. The filter is only moved along the width and height of the input. Their depth is the same as the depth of the input matrix, which is only 1 in the example.

As can be seen in the example, the output becomes smaller than the input. This will happen when using any filter larger than $1 * 1$. If this is repeated for several layers we could end up with a very small matrix where a lot of information at the borders are gone [13].

To solve this, *zero-padding* can be used in order to preserve the matrix size between the input and the output [13, 9]. *Zero-padding* is used to make the input matrix larger by simply adding adding extra rows and columns at the edges and fill them with zeros. Depending on how much zero-padding is used, the output matrix can be made to have any size, but it is usually used to keep it the same size as the input.

Another variable in a convolutional layer is *stride*. Stride is how many steps the filter is moved between each multiplication. In the example the stride was 1, but the stride can be larger than that. Note that using a larger stride will lead to a smaller output matrix, since fewer matrix multiplications are made.

The size of the output matrix can be calculated by using equation 2.22, where W is the input size, F is the filter size, P is the amount of zeros you add on each side, and S is the stride [13].

$$(W - F + 2P)/S + 1 \tag{2.22}$$

If the input or filter matrices are not quadratic each dimension have to be calculated separately. For example, in figure 2.2 we have $W = 3$, $F = 2$, $P = 0$ and $S = 1$. The output size would then be $(3 - 2 + 2 \cdot 0)/1 + 1 = 2$ which is also what we get in the figure.

There are several reasons to use convolutional layers [9]. The filters are much smaller than the input and the weights of the filter are reused at every position of the input. Therefore substantially less storage is needed for the weights of a a convolutional layer compared to that of a fully connected layer, even if many filters are user in each layer. Less weights also leads to less computation and training time.

The convolutional layer is also equivariant to translation [9, 17]. Since the filter is applied to all different parts of the input, the result for a certain part of the input will be the same regardless where it is. If the input is an image, this means that the CNN will, for example, still recognize a face even if it is moved to another part of the image.

Pooling Layer

Usually some convolutional layers are followed by a pooling layer. In the pooling layers several outputs from a previous layer is summarized into a a single output [9, 4]. One purpose of the pooling layers is to make the system invariant to small changes in the input. It is also useful in order to lower the amount of data in the network and thus lower the amount of computation needed.

There are a few different ways to summarize a group of outputs, and one such way is max-pooling [4]. In max-pooling the value selected is simply the maximum value among

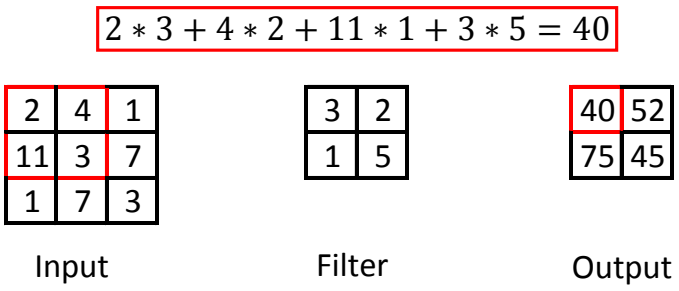


Figure 2.2: An example of how a filter is applied to the input in a convolutional layer. To produce the red cell in the output, the filter is applied to the red cells in the input. Each cell in the filter is multiplied with the corresponding cell in the input, and the sum of that becomes the output.

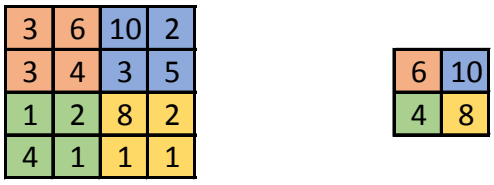


Figure 2.3: An example of how max pooling works in a convolutional layer. Each cell of the output to the right is the max of the corresponding colored area of the input to the left.

all outputs in the section. Another pooling method is average-pooling, which calculates the average value.

An example of how max-pooling works can be seen in figure 2.3.

The pooling layer works with each depth slice independently and does not combine any data along the depth dimension [13]. Therefore the output depth stays the same as the input depth even though the size otherwise decreases.

Commonly the size of the pooling units and their stride is the same, which means there is no overlapping. However Krizhevsky et al. [15] had success with setting a size larger than the stride and obtaining overlapping pooling. It was found that models with overlapping pooling was slightly harder to overfit.

Fully Connected Layer

At the end of the CNN there are fully connected layers. These layers are structured in the same way as layers in traditional neural networks are. Each neuron takes as input every

neuron in the previous layer [9]. Sometimes no fully connected layers except the output layer are used in a CNN [36].

Depthwise Separable Convolution

Depthwise separable convolution is a special type of convolutional layer [10]. The idea is to separate the depth and the spatial parts of a convolutional filter. In a normal convolutional layer the depth of the filter is the same as the depth of the input, while the output from each filter is only a single layer. This means that a filter both combines all the different channels of the input (depth) and combines the spatial surroundings (width and height) at the same time. *Depthwise separable convolution* instead makes separate filters for depth and spatial surroundings.

In a depthwise separable convolution layer, first a series of 1-depth filters is applied to the input, with one filter for each channel. This is called depthwise convolution and is responsible for finding the spatial correlation in the data. After those, a series of $1 * 1$ filters are applied. This is called pointwise convolution and is responsible for combining the different input channels of the data. A depthwise separable convolution later typically does not have any activation function between the depthwise and the pointwise convolution.

By separating into different filters, less total computations and weights are needed. It also makes intuitive sense to separate the cross-channel correlation and the spatial correlations. [10, 5].

Depthwise separable convolution has been used by Chollet to create the Xception architecture [5] and by Howard et al. to create the MobileNet architecture [10].

Xception is in turn based on the Inception architecture which was created by Szegedy et al. [32]. Inception implements "inception modules", which concatenates feature maps created from different filter sizes in each layer. Each convolution with larger filter size is preceded by pointwise convolution. Chollet et al. [5] uses these inception modules but replaces the convolution with depthwise separable convolution and produces better result than the Inception model.

2.9 Evaluation Metrics

When evaluating results to a classification problem, a confusion matrix can be used. A confusion matrix is a matrix where each row and each column is labeled with one of the possible output classes. One axis represents the true answer, and the other axis represents the predicted answer. The number in cell (x,y) is then the number of times the algorithm predicted class x when the answer was class y. The sum of the diagonal is the number of times the algorithm predicted correctly. See table 2.1 for an example.

For each class **x**, *true positive* (tp), *false positive* (fp), *false negative* (fn) and *true negative* (tn) can be calculated. *True positive* is the number of correct predictions for class **x** and *false positive* is the number of predictions for class **x** that were actually of another class. *False negative* is the number of predictions for different class that is actually of class **x** and *true negative* is the number of predictions for different class that is not of class **x**.

A confusion matrix was used by Tai et al. to evaluate the results in a similar study where a robot used a depth camera to navigate an indoor environment [33].

From the confusion matrix *accuracy*, *precision* and *recall* can be read. Accuracy is simply the number of times the algorithm predicted the correct answer divided by the total number of predictions, see 2.23. Accuracy can be calculated per class or for the whole result.

While accuracy can seem reliable, it can also be very misleading in some cases. Consider an model that tries to predict whether or not a person has a certain disease that 0.1% of humans have. If the model always predicts 'no, this person does not have the disease', then it would still be correct almost every time (since the disease is so rare). The model would have

Table 2.1: Example of a confusion matrix

		Predicted		
		Class A	Class B	Class C
Actual	Class A	45	8	3
	Class B	10	36	7
	Class C	3	11	50

99.9% accuracy, but since the model always gives the same answer regardless of input, we can see that the model would be useless in any practical situation.

In an attempt to solve this, precision and recall can be used as well to evaluate a model. Precision and recall is calculated for each class. Precision is the number of true predictions divided by the total number of instances that were predicted for that class. Recall is the number of true predictions divided by the total amount of correct instances for that class. See equations 2.24 and 2.25.

$$Accuracy = \frac{tp + tn}{tp + fp + tn + fn} \tag{2.23}$$

$$Precision = \frac{tp}{tp + fp} \tag{2.24}$$

$$Recall = \frac{tp}{tp + fn} \tag{2.25}$$

The F_1 score, or F-score, is sometimes used to produce a single value that takes both precision and recall into account. It is the harmonic mean of the two values and ranges from 0 (worst) and 1(best) just like accuracy, precision and recall. The formula for F_1 score is shown in equation 2.26.

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.26}$$

2.10 Related work

Image Classification

In 2012 Krizhevsky et al. developed a deep convolutional network to classify images in the ImageNet competition [15]. Using techniques like multiple GPUs, dropout and data augmentation, they won the competition and reached a top-5[2] accuracy of 83.0%. The second-place in the competition only reached 74.3%. This was a drastic increase in accuracy, and following this many more CNNs were used in the ImageNet competition with great success.

He at al. presented *deep residual learning* in 2015 [36]. Deep residual learning uses shortcut connections between layers. These shortcut connections are neuron connections that skips one or several layers. Using their residual net they won several tracks in the ImageNet competition and got 9% top-5 error rate on the ImageNet localization task. The shortcut connections also allowed to train very deep network and still have no optimization difficulty. To test this they explore a 1202-layer network, and while it has worse accuracy than their 110-layer network it still gets good results. Shortcut connection has since been used in several other successful networks such as Xception [5] and DenseNet [11].

[2]When the true label is among the top 5 guesses from the network.

Autonomous Driving

Chen et al. describes two major paradigms for autonomous driving systems [21].

In the *mediated perception approach*, several sub-systems gather information about many things in the vehicles environment, such as nearby cars, street signs and traffic lights. All this information is combined to create a full representation of the surrounding world. The AI-based system will then take the full representation into account when making decisions.

The *behavior reflex approach* instead simplifies this process by making the sensor data a direct input to the AI-system which then produces the driving actions from that input. To learn this mapping, the system records the inputs (sensor-data) and outputs (the driving commands) while a human drives the vehicle.

Chen et al. also presents a third paradigm, *direct perception approach*, which lies between the other two paradigms. In this approach, there is a mapping from an image to several meaningful indicators, such as the distance to the nearest car in front of you, the distance to the lane markings on both sides, and the angle that the road is turning. This compact representation of the world is then used as input to a relatively simple controller.

Huval et al. trains a CNN to detect vehicles and lanes in real-time while driving [12]. The input images were taken while driving around in the San Fransisco Bay Area. The images were then labeled with the location of vehicles and lane marking using combination of human annotators and LIDAR mapping. For lane detection the system is very accurate up to 50 meters away from the car. Beyond 65 meters the accuracy starts to drop fast, mainly due to the image resolution failing to capture the markings at that distance. The vehicle detection also performs well but produces several false positives in the forms of trees and overpasses. Its accuracy was compared to to a mid-range radar and performed better.

Behaviour Reflex Approach

As mentioned in the introduction, Tai et al. [34], LeCun et al. [16] and Giusti et al. [7] have done similar studies in the past.

In the study by Tai et al. [34] a Microsoft Kinect sensor was used to measure depth. A set of depth images from an indoor environment with corridors was then used as training data. The output classes were five steering commands such as "full-right" and "straight-forward". A softmax-classifier was used to the output to make the final decision on how to steer. An overall accuracy of 80.2% were reported and a high similarity between robot and human decisions was noted.

LeCun et al. [16] used two cameras to produce stereo images of an outdoor environment. The images were then given as input to a neural network which produced a turning angle in order to avoid obstacles in the image. The images were produced by a human driver of a small robot car where the steering angle for each image were recorded. An 83% similarity between the system output and the human driver were recorded. However LeCun at al. noted that many of the errors were because the robot would turn slightly later than the human, or in situations were both a left turn and a right turn could avoid the obstacle, and the human and the robot chose differently. The rate at which the robot would avoid obstacles could thus be higher than the reported accuracy.

In the study made by Giusti et al. [7] aerial outdoor images were produced in order to make a robot follow a trail in a forest environment. Each image was classified according to which direction the trail was heading (left, forward or right). The images were taken by a hiker equipped with three cameras, one directed straight forward and the other two at 30 degrees to the left and right. The accuracy of the neural network were compared against two other algorithms as well as two human observers. The accuracy reported was 85.2%, similar to that of the humans. The algorithm was also tested with flying drones on an actual trail. However the cameras used produced images of lower quality compared to those of the

training set, and the drone did not do well on narrow paths because it turned too late. In good lighting conditions and wide trails it managed to follow the trail for a few hundred meters.

Depth Estimation

Saxena at al. states that depth estimation using a single monocular image is a difficult task for a machine, as opposed to using binocular vision or multiple images [25]. However they also note that humans are very good at this task, by using cues such as texture variations and known object sizes. Most of those cues are contextual, and cannot be inferred by only looking at a small part of the image. Saxena et al. collects images and their depth maps using a 3-D laser scanner. They then trained a Markov Random Field using this data. Their model was tested on both outdoor and indoor images and performs quite well on predicting relative depth between objects, but make more errors when it comes to their absolute distance.

Liu et al. studied the use of a CNN to estimate depth with a monocular camera [18]. They use a deep CNN paired with Conditional Random Fields. They divide an image into groups of pixels called superpixels, and the depth of each superpixel is then predicted. They compare their results to other work and produce equivalent or better results on estimating depth on both indoor and outdoor images.

3 Method

3.1 Equipment

The robotic car used in the experiment was equipped with a Raspberry Pi Model B+ and a microcontroller. It was also equipped with a camera which can take images with a resolution of 160 x 120 pixels. See figure 3.1 for an image of the car. The Raspberry Pi was deemed too slow to handle the image classification itself, so the images were sent to a different computer where the neural network resided. After the classification was done, the predicted action was sent back to the car which then followed the command. The training of the neural network was made on an Amazon EC2 server equipped with four NVIDIA K80 GPUs and 61 GB RAM [1].

3.2 Dataset

In Strömgren's previous study 13 241 images had already been collected. The images were collected by having a person drive the robotic car around in an indoor environment. Examples of images collected can be seen in figure 3.2. Each image was automatically classified with the action that was taken during driving. The possible image labels were: 'forward', 'forward-left', 'forward-right', 'rotate-left' and 'rotate-right'. To make good training data, the driver had to consistently make the same decisions when faced with similar situations. Otherwise images where the car is expected to make the same decision could end up with different labels.

The behavior during data collecting was to drive forward while there were no obstacles in front of the car, and to turn left or right when an obstacle got too close. Rotating were only used if the obstacle was too large to avoid in another way, e.g. a wall.

Data augmentation were then used to mirror every collected images. Mirrored images where the label was 'forward-left' or 'rotate-left' had their label changed to 'forward-right' or 'rotate-right', and vice versa. Mirrored 'forward' images were also labeled as 'forward'.

It was noted in the Strömgren's study that the accuracy of the network increased when the dataset size was increased from 7 500 images to the final collection of 13 241. It was therefore decided to further increase the size of the dataset.

[1] https://aws.amazon.com/ec2/instance-types/p2/

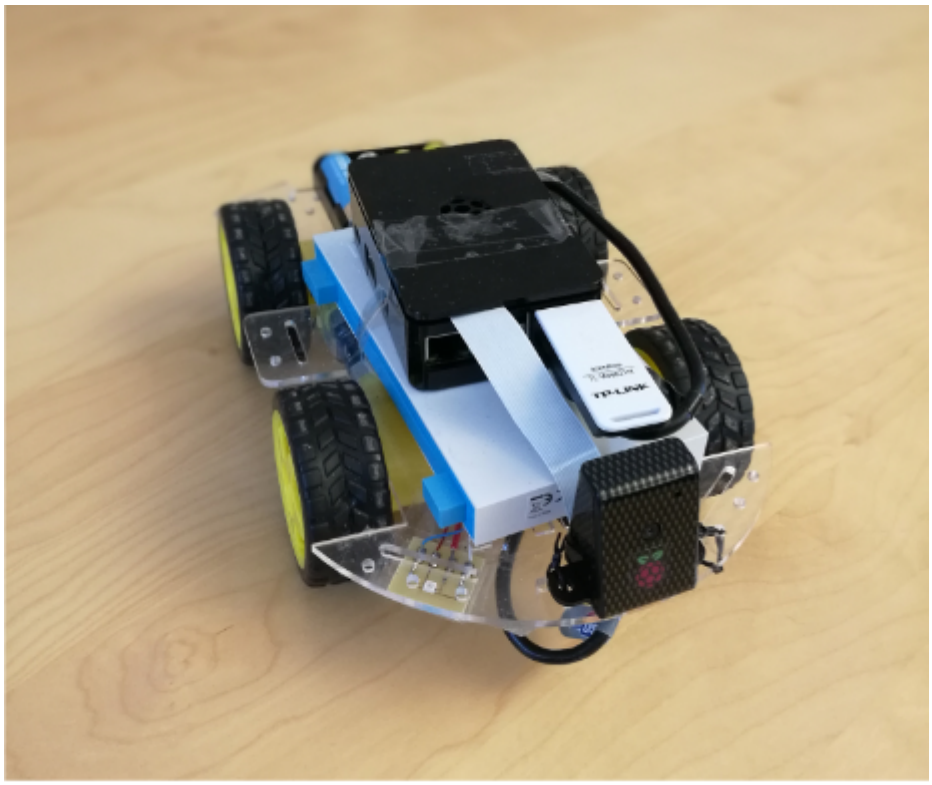


Figure 3.1: The robotic car used in the experiments.

Table 3.1: Collected data

Dataset	Size
Old Training Set	10 593
Old Validation Set	2 648
New Training Set	10 267
New Validation Set	2 614
Full Training Set	20 860
Full Validation Set	5 262
Test Set	4 900
Total Images	31 022

Another 13 000 new images were collected, bringing the total up to around 26 000 images. The images were collected in the same way as the original dataset. To make the result as good as possible, different rooms and obstacles were included in the images. Since the classification depended a bit on the person driving them, there could be consistency differences between the original dataset and the new dataset. Due to the behavior during data collection, most images were collected with the 'forward' label, and fewest from the 'rotate' labels.

The old data set was split into a training set and a validation set, with 20% of the data being in the validation set. The new training data was split in the same ratio by taking every 5th image and put it in the validation set. Another set of data was also collected to form a test set with which to evaluate the model after training was complete.

The different sets and their size can be seen in table 3.1. Note that the 'Full' datasets are the old and new combined and that only the last three datasets were used. The size of the other sets are only provided for reference.

Figure 3.2: Example of images captured by the robotic car.

Zhang et al. presents the possibility to obtain training images though simulation to easier collect data compared to using a real-world dataset [39]. Their environment only contained a robotic arm reaching a target in a single location as opposed to driving a vehicle around in a more varied environment. Therefore creating a simulation would be much more work in our scenario and was deemed unfeasible.

3.3 Implementation

The networks to be trained and evaluated were implemented in Tensorflow and Keras. Tensorflow is an open-source software library used for numerical computations. Keras is a a high-level API for neural networks that runs on top of Tensorflow. One network was already implemented by Strömgren, this network was the baseline of all comparisons. The already existing network was a model called VGG16.

VGG16

VGG is a set of networks originally made and trained by Simonyan and Zisserman in 2014 for use in the ImageNet Large Scale Visual Recognition Challenge 2014 [27]. VGG16 is a model with 16 layers, and can be imported in Keras. The architecture of the model can be seen in table 3.2. The convolutional layers are denoted by 'conv#<number of filters>' and the fully connected layers are denoted by 'FC#<number of neurons>'. All the filters in the convolutional layers are $3 * 3$.

The original VGG16 model has 4096 neurons in the first two fully connected layers and the final layer has 1000 neurons, one for each class to be predicted. To make the training faster, the fully connected layers were replaced by two layers with 256 neurons each. The final layer had 5 neurons since there are only 5 classes to predict.

To aid with training, transfer learning was used as described in section 2.6. The VGG16 was pretrained on the ImageNet dataset. The model had also been previously fine-tuned using the original dataset described above.

MobileNet

MobileNet was the second model selected to be tested. Howard et al. presented MobileNet in April 2017 as

> "a class of efficient models [...] for mobile and embedded vision applications."[10]

Table 3.2: VGG16 layer architecture

VGG16	Modification
conv#64	
conv#64	
maxpool	
conv#128	
conv#128	
maxpool	
conv#128	
conv#128	
conv#128	
maxpool	
conv#256	
conv#256	
conv#256	
maxpool	
conv#512	
conv#512	
conv#512	
maxpool	
conv#512	
conv#512	
conv#512	
maxpool	
FC#4096	FC#256
FC#4096	FC#256
FC#1000	FC#5
soft-max	

MobileNet consists of 28 layers as seen in table 3.3. Convolutional layers and fully connected layers are denoted in the same way as above and 'conv dw#<number of filters>' denotes depthwise convolutional layers as explained in 2.8. Each convolutional and depthwise convolutional layer is followed by a batch normalization and a ReLU activation function. Like in VGG16 above, the final layer with 1000 neurons was changed to a 5-neuron layer due to the number of target classes. The final layer is preceded with a dropout layer with $p = 0.001$. MobileNet was also available as a pretrained model in Keras.

Xception

Xception was presented in 2016 by Chollet [5]. The Xception model is 36 layers deep, not counting the fully connected layers in the end. The model contains depthwise separable layers like MobileNet, and it also contains "shortcuts" where the output of certain layers are summed with the output from previous layers. See figure 3.3.

On top of the convolutional layers a 5-neuron fully connected layer was added, just like for MobileNet and VGG16.

3.4 Transfer Learning

All the models used were pretrained on the ImageNet dataset. When using transfer learning you finetune the pretrained network on your target data. Finetuning the whole network might not be needed, instead a number of layers can be 'frozen' during the training on the

Table 3.3: MobileNet layer architecture

MobileNet
conv#32
conv dw#32 conv#64
conv dw#64 conv#128
conv dw #128 conv#128
conv dw#128 conv#256
conv dw#256 conv#256
conv dw#256 conv#512
conv dw#512 conv#512
conv dw#512 conv#512
conv dw#512 conv#512
conv dw#512 conv#512
conv dw#512 conv#512
conv dw#512 conv#1024
conv dw#1024 conv#1024
avg-pooling
FC#1000
softmax

target dataset. This means that the weights of those layers are not changed any further. In this study, the number of frozen layers during training was one of the hyperparameters that were changed to see how it affected the result.

3.5 Models

The network architecture, the number of frozen layers and the optimizer were all parameters that were changed to obtain several different models. See table 3.4 for a full list of the models and their parameters.

3.6 Training

The models were trained on a server as described in chapter 3.1. First the model was used only as a feature extractor, training only the final layer. This was done regardless of how many frozen layers had been chosen. This was done to to have the newly added final layer start a bit pre-trained just as the rest of the model. After this, the whole model was trained, excluding the frozen layers.

Due to a limit in the possible input sizes for the pretrained networks in Keras, the images were resized to $128 * 128$ and $221 * 221$ for the MobileNet and Xception models respectively.

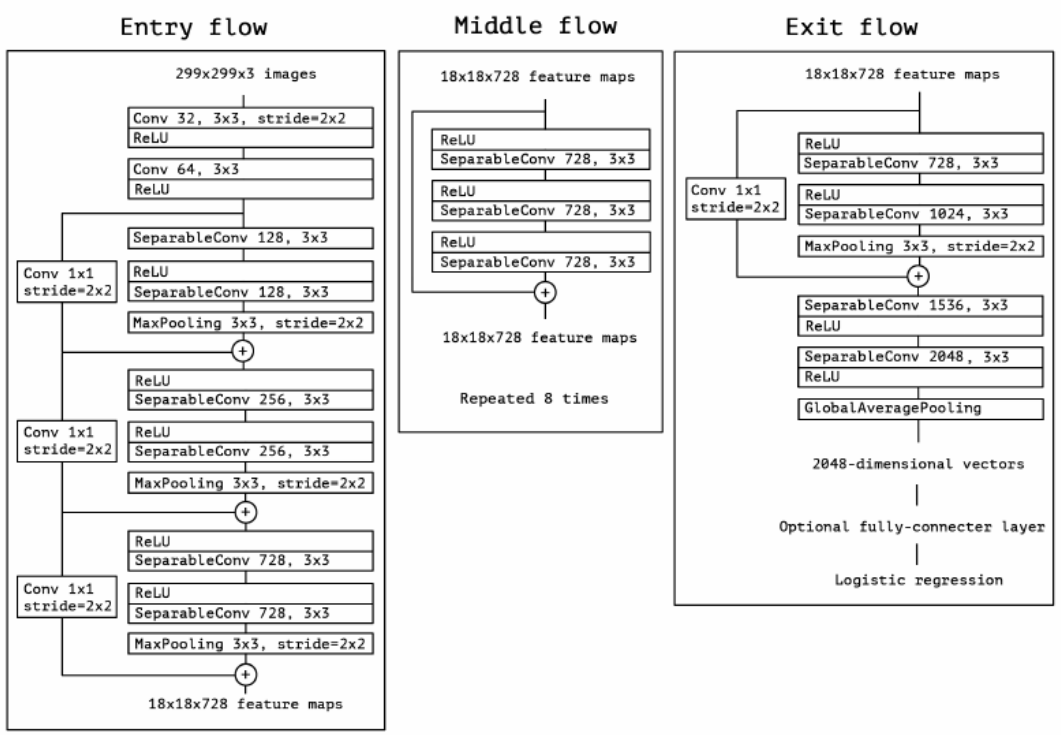


Figure 3.3: Xception layer architecture

The MobileNet models were trained for 50 epochs, while the Xception models were only trained for 40 epochs. The reason for the lower amount of epochs was that the Xception models are larger and take a longer time to train. This is also the reason there are fewer Xception models than MobileNet models. After every epoch the model was tested on the validation data. After the training was concluded, only the version with the highest accuracy on the validation data were saved. This means that the model after 50 (or 40) epochs of training were not necessarily the one used.

3.7 Evaluation

The evaluation is divided into two different parts. One quantative part, where the accuracy of the networks are tested against the test set, and one qualitative part where the robotic car drives by itself in an indoor environment and the amount of collisions with obstacles is measured.

Similar evaluations were made by Tai et al. although instead of measuring the amount of collisions they measured differences in angular velocity, something we lack the equipment to do [33].

The selected evaluation method is the same as in Strömgren's study [31].

Quantitative evaluation

the quantitative evaluation was made by having each model predict the correct action for each image in the test set. The accuracy, defined as the number of correct predictions divided by the number of total images, were measured for each combination of model and dataset. For the best model, also the precision, recall and F_1 score was measured.

Table 3.4: The tested models

Model Name	Network Architecture	Optimization Function	Frozen Layers
Model 1	MobileNet	SGD with learning rate = 0.001	0
Model 2	MobileNet	SGD with learning rate = 0.001	1
Model 3	MobileNet	SGD with learning rate = 0.001	5
Model 4	MobileNet	SGD with learning rate = 0.001	9
Model 5	MobileNet	SGD with learning rate = 0.001	13
Model 6	MobileNet	SGD with learning rate = 0.001	17
Model 7	MobileNet	SGD with learning rate = 0.001	21
Model 8	MobileNet	SGD with learning rate = 0.001	25
Model 9	MobileNet	SGD with learning rate = 0.0001	0
Model 10	MobileNet	SGD with learning rate = 0.0001	1
Model 11	MobileNet	SGD with learning rate = 0.0001	5
Model 12	MobileNet	SGD with learning rate = 0.0001	9
Model 13	MobileNet	SGD with learning rate = 0.0001	17
Model 14	MobileNet	SGD with learning rate = 0.0001	25
Model 15	MobileNet	Adadelta	0
Model 16	MobileNet	Adadelta	1
Model 17	MobileNet	Adadelta	5
Model 18	MobileNet	Adadelta	9
Model 19	MobileNet	Adadelta	17
Model 20	MobileNet	Adadelta	25
Model 21	MobileNet	Adam	0
Model 22	MobileNet	Adam	1
Model 23	MobileNet	Adam	9
Model 24	MobileNet	Adam	17
Model 25	MobileNet	Adam	25
Model 26	Xception	SGD with learning rate = 0.001	20
Model 27	Xception	SGD with learning rate = 0.001	29
Model 28	Xception	SGD with learning rate = 0.001	34
Model 29	Xception	SGD with learning rate = 0.0001	20
Model 30	Xception	SGD with learning rate = 0.0001	29
Model 31	Xception	SGD with learning rate = 0.0001	34
Model 32	Xception	Adadelta	20
Model 33	Xception	Adadelta	29
Model 34	Xception	Adadelta	34
Model 35	Xception	Adam	11
Model 36	Xception	Adam	20
Model 37	Xception	Adam	29
Model 38	Xception	Adam	34
Model 39	VGG16	SGD with learning rate = 0.0001	11

Qualitative evaluation

In the qualitative evaluation the car was driving around in the indoor environment. The tests were only made on the model that had performed the best in the quantitative testing. Obstacles such as chairs and backpacks (the same as the networks had been trained on) were placed 1-2 meters in front of the car and the car was instructed to move forward. It was measured how many times the car collided with the obstacle compared to the total number of runs. The car was also driving around more freely in the environment and its general behavior was observed.

4 Results

In the following chapter the results from the experiments will be presented.

4.1 Quantitative Evaluation

Accuracy

The accuracy for each model is shown in Table 4.1. Both the accuracy on the validation set and the test set is shown. The model numbers are the same as in Table 3.4. As we can see, model 35 had the highest accuracy on the test set. The **bolded** models are the model with the highest accuracy for each architecture. The models in *italics* are the models which did not learn anything, which will be discussed further down.

Table 4.1: Accuracy for the tested models

Model	Architecture	Frozen layers	Validation Accuracy	Test Accuracy
1	MobileNet	0	85.23%	76.14%
2	*MobileNet*	1	*64.23%*	*66.95%*
3	*MobileNet*	5	*64.23%*	*66.95%*
4	*MobileNet*	9	*64.23%*	*66.95%*
5	*MobileNet*	13	*64.23%*	*66.95%*
6	*MobileNet*	17	*64.23%*	*66.95%*
7	*MobileNet*	21	*64.23%*	*66.95%*
8	*MobileNet*	25	*64.23%*	*66.95%*
9	MobileNet	0	65.00%	68.79%
10	*MobileNet*	1	*64.23%*	*66.95%*
11	*MobileNet*	5	*64.23%*	*66.95%*
12	*MobileNet*	9	*64.23%*	*66.95%*
13	*MobileNet*	17	*64.23%*	*66.95%*
14	*MobileNet*	25	*64.23%*	*66.95%*
15	**MobileNet**	0	**86.97%**	**77.43%**
16	*MobileNet*	1	*64.23%*	*66.95%*
17	*MobileNet*	4	*64.23%*	*66.95%*
18	*MobileNet*	9	*64.23%*	*66.95%*
19	*MobileNet*	17	*64.23%*	*66.95%*
20	*MobileNet*	25	*64.23%*	*66.95%*
21	MobileNet	0	85.96%	77.02%
22	*MobileNet*	1	*64.23%*	*66.95%*
23	*MobileNet*	9	*64.23%*	*66.95%*
24	*MobileNet*	17	*64.23%*	*66.95%*
25	*MobileNet*	25	*64.23%*	*66.95%*
26	Xception	20	79.46%	78.53%
27	Xception	29	76.30%	75.73%
28	Xception	34	69.25%	74.24%
29	Xception	20	77.34%	77.43%
30	Xception	29	73.88%	75.73%
31	Xception	34	68.06%	72.94%
32	Xception	20	87.06%	80.02%
33	Xception	29	82.60%	79.58%
34	Xception	34	71.68%	72.95%
35	**Xception**	11	**86.59%**	**81.19%**
36	Xception	20	86.43%	79.64%
37	Xception	29	83.06%	81.17%
38	Xception	34	74.54%	75.47%
39	**VGG16**	11	**87.69%**	**78.39%**

Accuracy during training

During the training of each model, its accuracy on the training set and the validation set was measured after each epoch. As mentioned in Section 3.6, the accuracy reported is not the accuracy after all epochs are completed, but rather the top validation accuracy. Here the accuracy over time is shown for some of the models.

Figure 4.1: Model 2 training

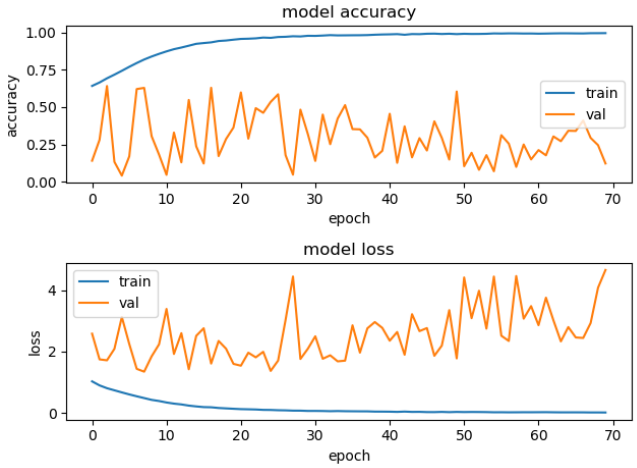


Figure 4.2: Model 4 training

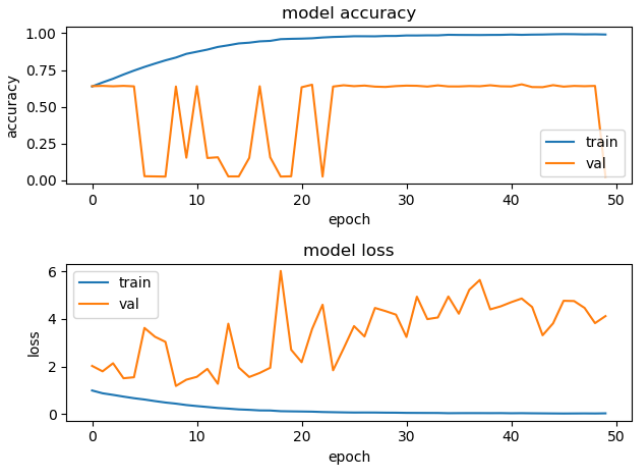


Figure 4.3: Model 18 training

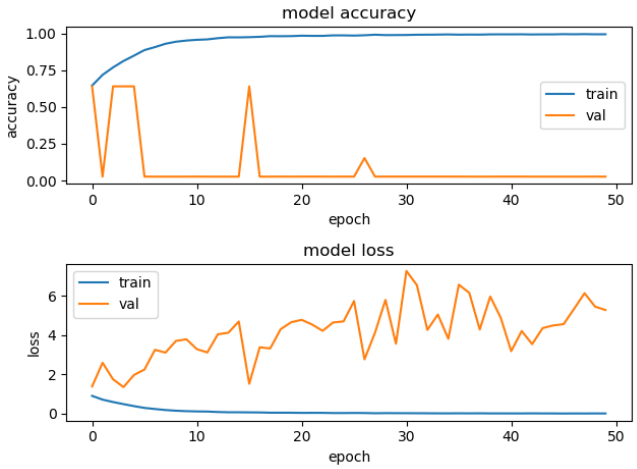
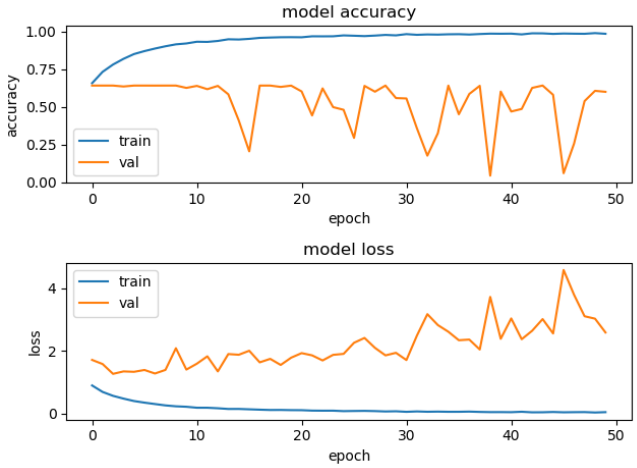


Figure 4.4: Model 22 training



In Figures 4.1, 4.2, 4.3 and 4.4 the accuracy during training is presented for some of the MobileNet models that performed worst. As we can see their training accuracy increases over time but their validation accuracy does not. The models are unable to generalize their knowledge and does not really learn anything.

Figure 4.5: Model 15 training

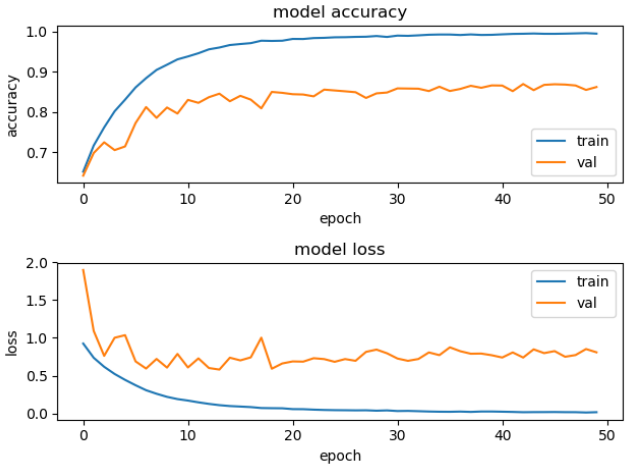
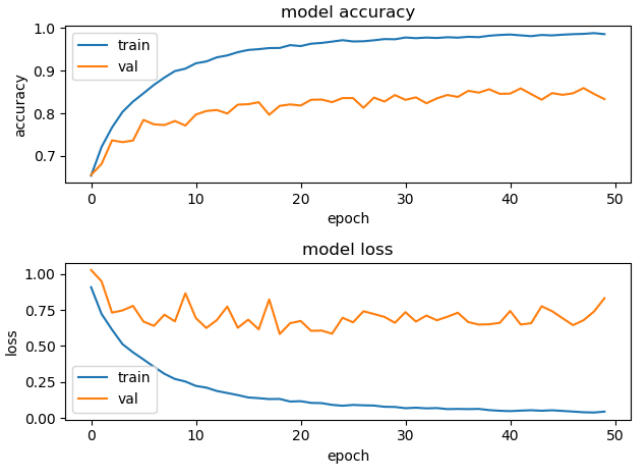


Figure 4.6: Model 21 training



Figures 4.5 and 4.6 show the training of the best MobileNet models. These have no frozen layers unlike the models above that were unable to learn anything. They appear to continously learn during many of the epochs.

Figure 4.7: Model 35 training

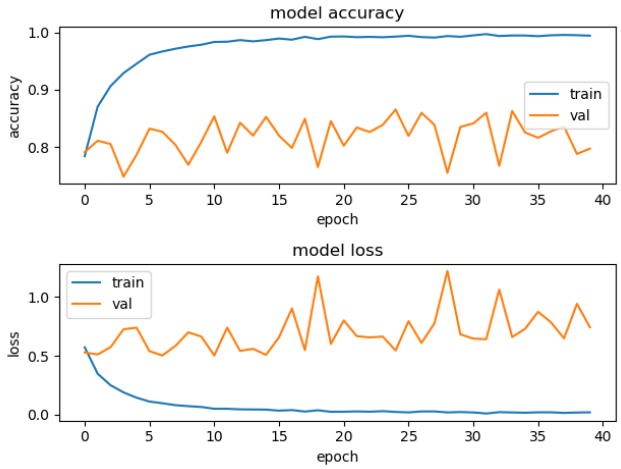
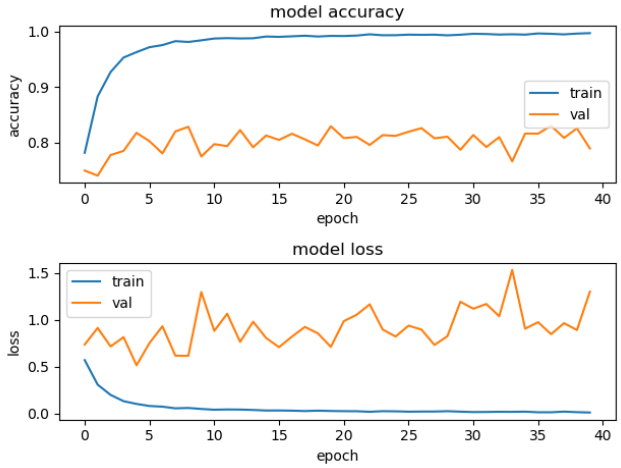


Figure 4.8: Model 37 training



In figures 4.7 and 4.8 we can see the training of the Xception models that performed the best. Unlike the MobileNet models their training process is much less smooth, with their validation accuracy going up and down a lot during the training. Nevertheless they perform better overall than the MobileNet models and gives a higher accuracy on both the validation set and the training set.

Figure 4.9: Model 39 training

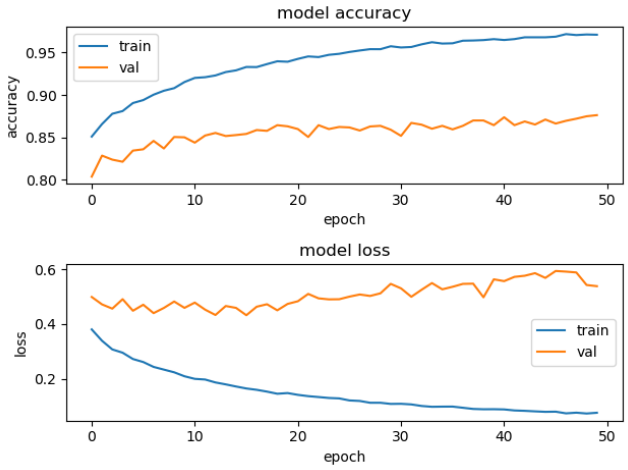


Figure 4.9 show the training for the VGG16 model that we compared our work to. Its validation accuracy increases over time and shows a steady learning process similar to the MobileNet models.

Confusion Matrix

After testing the model on the test set we can see what it predicted for each class compared to the correct answer. Here is the results presented as a confusion matrix for some of the models.

Table 4.2: Confusion matrix for many MobileNet models

		Predicted				
		Rotate-Right	Rotate-Left	Forward	Forward-Right	Forward-Left
True	Rotate-Right	0	0	57	0	0
	Rotate-Left	0	0	57	0	0
	Forward	0	0	3278	0	0
	Forward-Right	0	0	754	0	0
	Forward-Left	0	0	750	0	0

In Table 4.2 we can see the confusion matrix for several MobileNet models. The same matrix was produced for all the models that had a validation accuracy of 66.95%. These models simply had 'Forward' as their output regardless of input.

Table 4.3: Confusion matrix for Model 21

		Predicted				
		Rotate-Right	Rotate-Left	Forward	Forward-Right	Forward-Left
True	Rotate-Right	17	6	21	9	4
	Rotate-Left	5	21	12	4	15
	Forward	9	21	2883	186	179
	Forward-Right	9	1	240	440	64
	Forward-Left	0	14	278	39	419

Table 4.3 shows the confusion matrix for model 21, which was the best performing MobileNet model. This model performs much better than the models shown above, but it still has some problems with seeing the distinction between the 'Forward' images and the 'Right/Left' images.

Table 4.4: Confusion matrix for Model 35

		Predicted				
		Rotate-Right	Rotate-Left	Forward	Forward-Right	Forward-Left
True	Rotate-Right	39	1	9	0	8
	Rotate-Left	0	29	8	6	14
	Forward	11	2	2989	113	163
	Forward-Right	44	4	260	425	21
	Forward-Left	0	30	203	24	493

Table 4.5: Confusion matrix for Model 37

		Predicted				
		Rotate-Right	Rotate-Left	Forward	Forward-Right	Forward-Left
True	Rotate-Right	28	5	7	13	4
	Rotate-Left	4	16	13	2	22
	Forward	3	0	2859	151	265
	Forward-Right	18	1	206	506	23
	Forward-Left	1	6	157	21	565

Model 35 and 37, as shown in Table 4.4 and Table 4.5 respectively, were the best Xception models. Model 35 has the highest overall accuracy, but as we can see model 37 has slightly higher accuracy for the 'Forward-Left' and 'Forward-Right' classes.

Table 4.6: Confusion matrix for Model 39

		Predicted				
		Rotate-Right	Rotate-Left	Forward	Forward-Right	Forward-Left
True	Rotate-Right	32	10	4	6	5
	Rotate-Left	3	29	3	5	17
	Forward	44	31	2702	240	261
	Forward-Right	32	0	163	524	35
	Forward-Left	3	27	154	15	551

Model 39 as seen in Table 4.6 was the VGG16 model from Strömgren's study.It has slightly lower accuracy than the Xception models above, but it performs on par with model 37 on the 'Forward-Left' and 'Forward-Right' classes.

Precision and Recall

From the confusion matrix of the most accurate model (model 35), its precision, recall and F_1-score for each class were calculated and is presented below.

Figure 4.10: Model 35 Precision

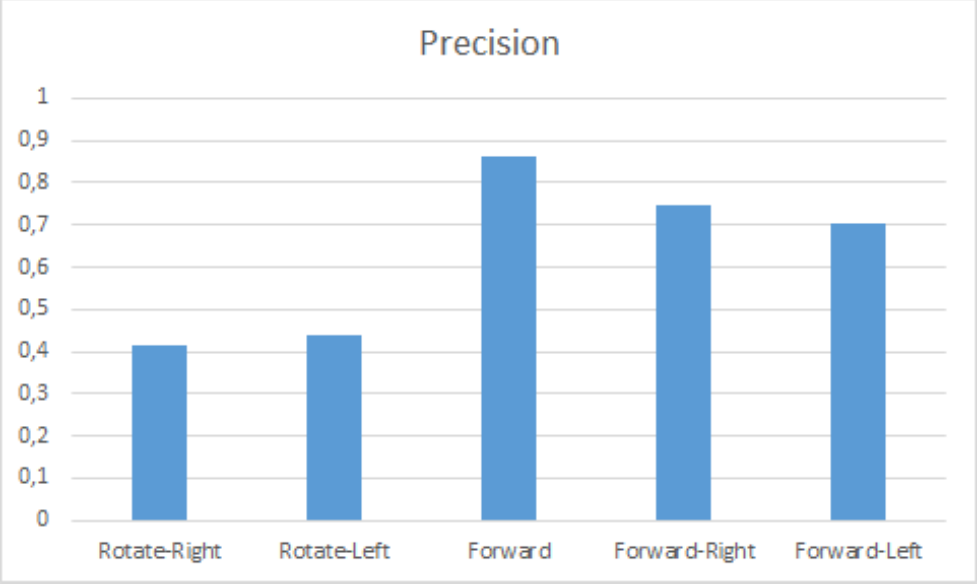


Figure 4.11: Model 35 Recall

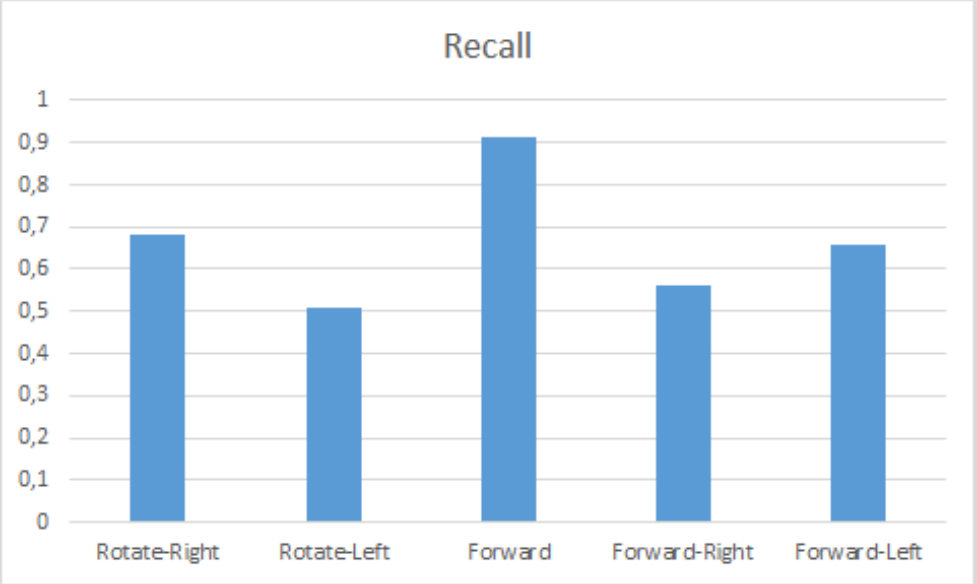
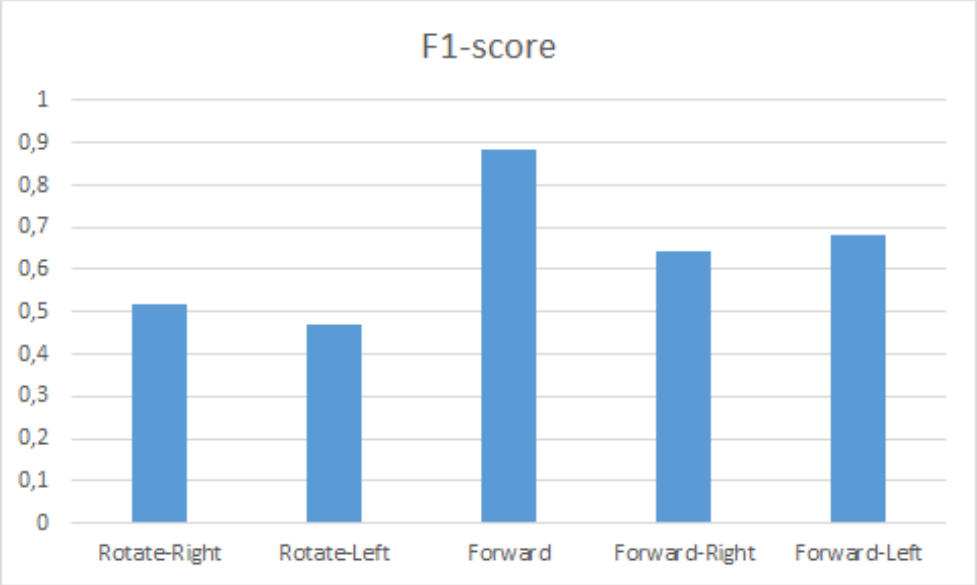


Figure 4.12: Model 35 F_1 score



In Figure 4.10 and 4.11 the precision and recall for each class respectively is presented. The precision and recall for the 'Forward' class is high, but for the 'Forward-Right' and 'Forward-Left' classes their recall is lower than their precision. This means that most of the 'Forward-Right/Left' labels the model predicts are indeed of the 'Forward-Right/Left' class, but we also output many values which are actually another class. For the 'Rotate' classes the opposite behavior is observed instead.

Figure 4.12 show the F_1 score for each class is presented. As it is the harmonic mean of precision and recall is provides a measure on how good the model is at predicting each class. Here we can see that the 'Forward' class performs the best while the 'Rotate' classes performs the worst.

Table 4.7 shows the exact values for each measure displayed in the above figures.

Table 4.7: Precision and Recall for Model 35

	Precision	Recall	F_1 Score
Rotate-Right	0.415	0.684	0.517
Rotate-Left	0.439	0.509	0.472
Forward	0.862	0.912	0.886
Forward-Right	0.748	0.564	0.643
Forward-Left	0.705	0.657	0.680

4.2 Qualitative Evaluation

In Table 4.8 the results of the qualitative evaluation are presented. The obstacles the car was tested were also present in the training set. The avoidance rate is measured by how often the robot car were able to detect and avoid the obstacle in front of them. If the car did not collide with the object, it was counted as a successful run. 10 test runs for each obstacle were performed.

Table 4.8: Avoidance rate for the robot car

Obstacle	Avoidance rate
Trash can	90%
Fire extinguisher	90%
Bar stool	80%
Sofa	60%
Backpack	90%
Chair	60%
Wall	80%
Total	78.57%

5 Discussion

This chapter is divided into two parts. In the first part the results are discussed and compared to results from similar studies. In the second part the method is discussed to see what could have been done differently.

5.1 Results

In this part the results are discussed from a few different perspectives.

Transfer Learning

All models were trained using transfer learning, and was first pretrained on the ImageNet database. The models were trained on the ImageNet image classification task, where for each image, the model were to tell what object was in the image, regardless of the location of the object. However, on our task the location of the obstacle did matter for the classification. Simply recognizing and obstacle was not enough, but the model has to know how far away it is and if it is mostly to the left or to the right in order to make a meaningful choice on if and how to steer away from it.

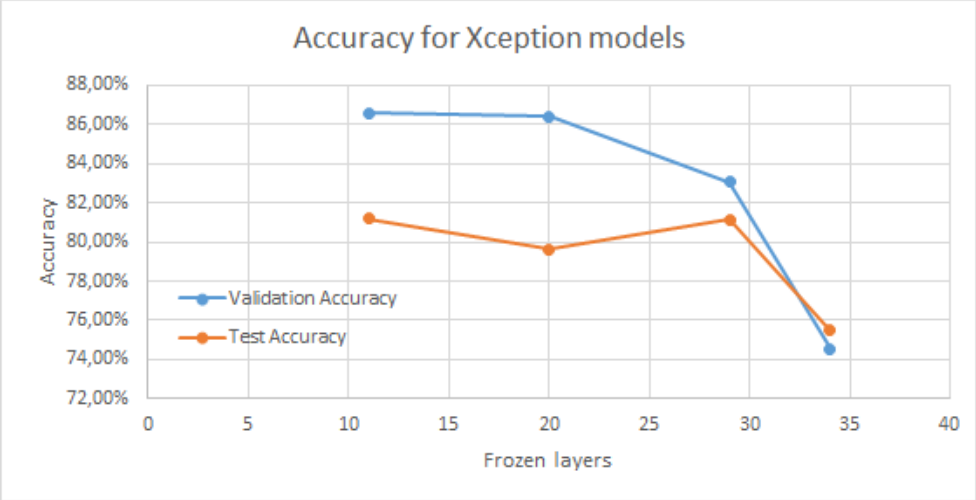
Since the model would now have to learn to recognize the location of objects as well there might not be much transferable knowledge between the different problem domains. Yosinski et al. noticed that the first few layers of deep neural networks tend to learn similar features regardless of dataset or task, which would mean that transfer learning is still useful when working with a different problem [37].

Yosinski et al. also showed that freezing layers might in some cases drop the performance due to fragile co-adaptations, i.e. that the neurons in different layers are too dependent on each other. This is also observed in our experiments. The MobileNet models were unable to learn anything when some of their layers were frozen, perhaps because each layer depended a lot on the other layers, so that only training some of them with new data destroyed their connections. These "untrainable" models only predicted the same class for every input as seen in Section 4.2, which is of course not useful at all. This could be because of a combination of fragile co-adaptations as described by Yosinski et al. as well as an unbalanced dataset. Only when all the layers in the MobileNet models were allowed to be trained did the models manage to learn anything.

For the Xception models, their accuracy goes up and down a lot during training as we can see in Figures 4.7, 4.7 and 4.8. However they do show an upward trend during the first 10-15 epochs. However their loss does not necessarily go down as the accuracy increases, instead we see the loss staying the same or increasing for the Xception models.

However, unlike the MobileNet, all Xception models were able to learn, but we see a loss in performance as the number of frozen layers grow. In Figure 5.1 we plot the accuracy for the Xception models using the Adam optimizer function. We can see the the both the validation and the test accuracy are highest for the model with the fewest frozen layers. However fewer frozen layers also took longer time to train, and unfortunately Xception models with even fewer or no frozen layers were not trained.

Figure 5.1: Layer accuracy

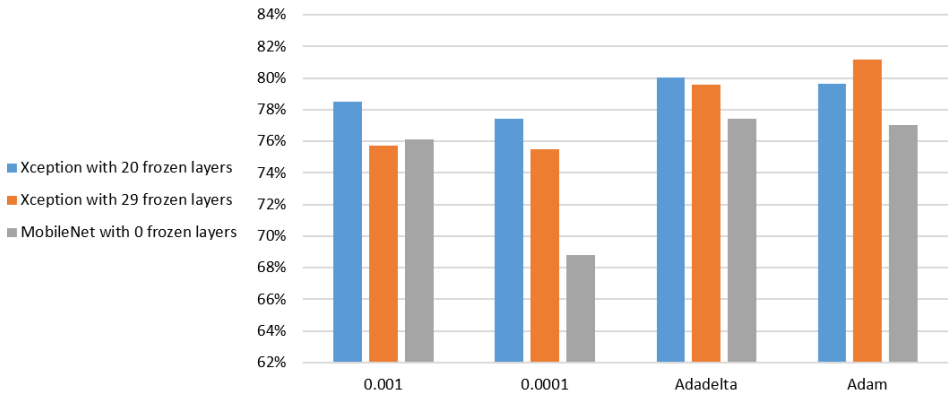


Oquab et al. trained a convolutional neural network on the ImageNet database. The final layer was then removed and replaced with two new layers, and the resulting network was then trained on another image database. Thus most of the layers were frozen. Their second dataset was much smaller and contained fewer classes than the ImageNet database. However the images was still labeled according to what object the image contained, as opposed to having labels related to the location of the object as we have. Their best approach had an increase in accuracy from 70.9% to 82.8% compared to using no pretraining at all. On a different dataset Oquab et al. also tested to freeze fewer layers and also train last previously frozen layer. Their network with 5 frozen layers achieves an accuracy of 70.2% while their network using 7 frozen layers only reaches 68.4%. This increase in accuracy when lowering the amount of frozen layers is consistent with our results from the Xception models.

Optimizer

While not the main focus of the study, different optimizers were tested as a hyperparameter to the models. In Figure 5.2 the accuracy for different models is reported for each optimizer, where 0.001 and 0.0001 refers to SGD with the respective learning rate.

Figure 5.2: Accuracy for optimizers



The two optimizers with adaptive learning rate (Adadelta and Adam) perform better in general than SGD does. SGD is more likely to get stuck in local minima and saddle points since it has a constant learning rate [23]. Adam and Adadelta perform very similar however. Adadelta gives a slightly higher accuracy in general, but the highest accuracy is given by a model using Adam. Because the difference in performance is so small, it is possible that it can be explained by random variation during training, in which case their performance is more or less equal.

Comparison with similar studies

The model that achieved the highest accuracy was a Xception model using the Adam optimizer with 11 frozen layers. It reached an accuracy of 81.19% on the test set. In a study by Lecun et al. a CNN were trained on a similar problem but using two cameras instead of one when recording the data [16]. They reported an accuracy of 64.2%. This is significantly lower than in our experiment, but the environment used for training were outdoors, as opposed to indoors. An outdoor environment is likely much more varied and difficult to traverse than a single office location. They also note that when driving, their robot is able to avoid obstacles more often than their accuracy would suggest.

Tai et al. used a depth camera in an office environment and reported a classification accuracy of 80.2%, which is almost the same as achieved in our experiment [33]. They also used many fewer images for their training data. 750 images were used for training compared to our 20 000 images. However a major part of the task is to be able to avoid obstacles as the robot gets close to them. This is substantially easier with a depth-camera compared to using a non-depth camera. In the former case the model will learn depth by looking at the input values directly, while in the latter case some other less obvious cues must be used to determine the distance to the obstacle.

Comparison with Strömgren's study

The object avoidance rate for our robot car was measured as 78.57%. This a slightly higher than the accuracy reported by Strömgren in his study [31], which was 72.53%. Strömgren also reports a validation accuracy of 82.44%, while the validation accuracy of our best model was 87.06%. Training the VGG16 model that Strömgren used on our data set gives a validation accuracy of 87.69%, even higher than our best model. But since the validation set is used during training to select the top epoch for each model, it can misleading to test the final accuracy on that set. If we instead look at the test set performance, our best model has an accuracy of 81.19% compared to 78.39% for the VGG16 model. The best model uses the Xception architecture, which has more parameters than the VGG16 model. The increased number

of parameters mean it has a potential to learn more information, which is a possible explanation for its higher performance. Another reason could be that the Xception architecture uses special techniques such as shortcut layers and depthwise separable convolution, both which have shown to be effective in convolutional neural networks [5].

Qualitative Results and F_1-score

The robot car were able to avoid obstacles most of the time, and most of the time when it did collide with something it was able to rotate away and continue driving. It also rarely decided to turn left or right when it did not need so, i.e. when the path in front of the car was clear. As can be seen in the confusion matrix in Table 4.4, it was more common for the model to predict forward when the response should have been left or right, than it was to predict left or right when it should have gone forward.

This can also be seen in the precision/recall measure in Figures 4.10 and 4.10, where we can see that the model's precision is higher for left/right than its recall. Looking at the F_1-score in Figure 4.12 we see that Forward performs the best, and the Rotate-classes performs the worst. This reflects the number of training examples for each class, where Forward had the most and the Rotate-labels had the least.

As seen in Table 4.8 the sofa and the chair had the worst avoidance rates at 60%. For the chair obstacle it should be noted that it is the legs of a chair that we are trying to avoid. Compared to the other obstacles, chair legs are thin and cover a much smaller part of the image. This could be a possible reason why the car had problems to avoid chairs. A chair also has several chair legs, so an attempt to avoid one of them might lead to a crash into another.

The sofa is on the other side of the spectrum, with most images of the sofa instead filling up the whole image. This means that while the car is driving towards the sofa, but before it is within the range where the human drive would steer away from it, the sofa would still cover most or all of the image. This means that images of driving straightforward towards the sofa looks very similar to images taken while beginning to steer away from it.

All training and evaluation took place in MindRoad's office, so it is hard to say if the robot would perform equally good or worse in another environment. However all obstacles except the sofa were moved around to various locations in the office during both training and evaluation in an attempt to minimize any reliance on the background while making decisions.

It might be possible that there are diminishing returns on the performance able to be reached using a single camera for a task requiring depth perception. Using two cameras instead of one, or a depth-sensor could lead to better results with little effort.

5.2 Method

In this part the different parts of the method is discussed. The collection of the dataset, the training of the neural networks and the evaluation methods are all discussed in the corresponding section.

Dataset

The size of the dataset is fairly limited when comparing to other image classification studies. The networks were pretrained on ImageNet, which contains millions of images, compared to our dataset of around 26 000 images.

Since the dataset has been collected manually, and there exists no 'objective' truth on when exactly the car should turn when seeing an obstacle, inconsistency in the images is a possibility. While care was taken to try to turn at the same distance to an object every time, the car is not equipped with any depth sensor, so it could not be strictly enforced. The dataset

was also collected by two different people which might have estimated the distance a bit different.

All of this makes the resulting dataset, and the true labels for each image, somewhat subjective. In situations where the obstacle is right in the middle of the image, going left or right to avoid it might be equally valid. However each image has only one label, so only one of the directions would be 'correct'. Before training each image is also mirrored and a 'left' label is changed to a 'right' label (and vice versa) on the mirrored version. This means that for these ambigous situations there would be two images, one where 'right' is the correct label and one where 'left' is the correct answer. This noise in the data would make it even harder for the neural network to learn a consistent driving style itself. Perhaps this could be alleviated by introducing a new image class for those images to signify that both left and right is a correct action.

Another issue with the dataset is the unbalanced image classes. Due to how the images were collected many of the are of the 'straight-forward' label, while few of them are of the two 'rotate' labels. This is because of the time spent driving forward was a lot more than the time spent rotating. Not only is rotating rarely needed, but the time it takes to rotate away from an obstacle is less than a second. This makes it hard to react fast while recording data as well, which could make the 'rotate' images unreliable.

Training

Due to limited time, only a limited amount of models and hyperparameters were considered. While the training of the MobileNet models only took about an hour, the training of the Xception models often took more than 5 hours. The Xception training started later in the project and therefore not as many hyperparameters could be explored.

Not counting the two different model architectures, two hyperparameters were explored: The optimizer function (and its learning rate where applicable) and the number of frozen layers. Especially the Xception models would merit from exploring the number of frozen layers more. Given more time, other numbers of layers could have been frozen. Since the MobileNet models ended up not able to learn anything when freezing layers regardless of the number of frozen layers or their optimization function, ideally less MobileNet models should have been trained in favor of instead training more Xception models. A smaller network could also have been considered, using less parameters means less training data would be needed, and then perhaps transfer learning would not be needed.

For example, Tai et al. uses a much smaller network than the models we tested with only five layers [33]. They do not use transfer learning and only 750 images for training, although they do use depth-images. A smaller network would lead to less training time and more time could be spent exploring both hyperparameters and the structure of the network.

Evaluation

The evaluation was split into two part, one quantitative part and one qualitative part. In the quantitative part each model was tested using a test set. The test set was not part of the training set, but the the two data sets were similar. They had been recorded in the same office environment using mostly the same obstacles. This was a limitation due to the fact that the study was constrained to MindRoad's office and the robot car required a stable connection to a router since the image classification happened on a different computer.

The qualitative evaluation measured what we really wanted to find out: if the car was able to avoid obstacles. In this evaluation the robot car was placed in front of different obstacles, and then the collision rate of those obstacles were recorded. There can of course be other ways to do this type of evaluation. As a simple example, many obstacles could have been placed at various location and the robot would drive until it collided with any of them. Unlike the

quantitative evaluation, there is no one obvious way to as objectively as possible measure obstacle avoidance rate.

5.3 The work in a wider context

As mentioned in the introduction, many companies are working on developing self-driving cars. Aside from cars, smaller machines are also becoming autonomous, such as delivery drones and factory robots. Depending on the application, the autonomous capabilities of the vehicle must be more or less exact. For example a car designed to drive humans in traffic must be extremely safe, while a factory robot operating inside a limited area might not have as high safety concerns.

Those safety levels will decide what type of software and hardware we need on the vehicle. In this study a small robot car equipped with a single camera has been explored, and the results are not yet fit for most real-world applications. Furthermore the system evaluated in this thesis is only used for obstacle avoidance and would have to be combined with other system in order to be more useful, such as a path-finding system. When the obstacle avoidance only becomes a part of a larger system, great care must be taken to balance and prioritize the different components in order to provide the required level of safety.

In 2016 Bojarski et al. [3] used a CNN to map images to steering commands, which is same approach as used in our study, on a real car driving on roads. The car was able to drive autnomousely (without human intervention) around 98% of the time. The CNN used in their study contained convolutional layers as well as fully connected layers. However in our study we have shown that Xception, a network that uses more advanced features such as shortcut connections and depthwise separable convolution, produces better results at the same task than a network not using those features. This implies that even better results than what Bojarksi et al. got is possible through more advanced network architectures.

6 Conclusion

In this report we set out to answer questions about Deep Learning and the user of transfer learning. For convenience we repeat our research questions below:

1. What implementations of convolutional neural networks are most suitable for simple obstacle avoidance?
2. How high accuracy can different neural networks reach when applied to obstacle avoidance?
3. Does transfer learning work well when the target classification problem is very different from the source problem and the target domain has much less training data?

The task was to have a neural network steer a small robot car. The network were to take images from a camera equipped on the car as input, and output a steering command. The goal was to have the network detect obstacles in front of it, and then steer away. If there were no obstacles in front of it, the car should drive straight forward. The car only have a single camera, which makes depth perception difficult.

We have trained several models using two different network architectures, MobileNet and Xception. Both these architectures make use of depthwise separable convolution as described in Section 2.8. They were compared to a VGG16 model which was MindRoad's previous implementation. VGG16 uses only standard convolutional layers and fully connected layers. Both MobileNet and Xception are larger networks than VGG16. Xception performed best of these three architectures and reached 81.19% accuracy however they were all close in accuracy, ranging from 77% to 81%. It is hard to generalize and say something about architetures not tested in this study, but Xception performs best out of the tested ones. Xception is a large network architecture with 36 convolutional layers. It also utilizes shortcut connections and depthwise separable convolution, which has shown promise in other image classification tasks [23, 40].

When using the network to steer the robot car the Xception model were able to avoid obstacles 78.57% of the time. It performed better than previously measured with the VGG16 model, however it were not able to sense and avoid the obstacles perfectly. It is possible that the accuracy might not get much higher using the equipment and approach used in this study. In the next section we present a few ways to continue the research.

Transfer learning was used for all models. They were pretrained on ImageNet, and then a number of layers were frozen. The models were then trained on a dataset containing around 20 000 images taken while driving the car around manually in an office environment. The MobileNet showed an inability to learn anything while it had frozen layers. This is likely because of fragile co-adaptations between different layers, which would mean they all needed to be trained together. The ImageNet training data consists of images that is classified with labels depending on what objects are inside the image. In our problem however the label is also dependent on where in the image the object is. This is another problem related to using transfer learning and could make it even harder for the models to learn this new problem with frozen layers.

The Xception model showed difficulty in learning as well, however it was clear that better results were achieved with fewer frozen layers. The trade-off here is longer training time. Since we had better results using fewer frozen layers we can conclude that while early layers might be more generalized that later layers, they still need to be retrained for best effect at our task. The differences between the two data sets and their related task are too large to be able to use transfer learning without also using significant training on the target data set.

6.1 Future Work

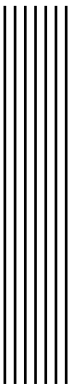
The neural network was only trained and evaluated in MindRoad's office. It is not clear how well the model are able to generalize to other environments, even similar indoor environments. Most of the evaluation was performed in a single large room. The generalization aspect of the classification can be further studied.

The images used for training the network were not evenl distributed among the output classes. The 'Forward' class had the most images, while the two 'Rotate' classes had the fewest. Having better distributed training data could increase the performance.

In this study we utilized transfer learning and then froze various amounts of layers. While less frozen layers took longer time to train, they also produced better models in general. For larger models like Xception models with no frozen layers can be attempted. Howerver with more layers to train even more training data might be needed.

Another way to expand on the work is to not do any transfer learning and instead only train on original data. This way the strange non-learning behaviour that happened with the MobileNet models could be avoided. This will likely need either more training data, a smaller neural network, or both.

Finally, a new method entirely could be attempted. The approach used in this study was to map the input images to a steering command directly. Adding intermediate steps could help the algorithm. If the monocular camera is kept the system could be split into two: one depth-estimation part that outputs a depth-map, and one steering part which takes the depth-map and outputs a steering command. Another way is to attempt to more precisely localize any objects in the image, and then have a different system that does the decision-making.



Bibliography

[1] Yusuf Aytar and Andrew Zisserman. "Tabula rasa: Model transfer for object category detection". In: *2011 International Conference on Computer Vision*. IEEE, Nov. 2011, pp. 2252–2259. ISBN: 978-1-4577-1102-2. DOI: 10.1109/ICCV.2011.6126504.

[2] Yoshua Bengio. "Practical recommendations for gradient-based training of deep architectures". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7700 LECTU (2012), pp. 437–478. ISSN: 03029743. DOI: 10.1007/978-3-642-35289-8-26. arXiv: 1206.5533.

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. "End to End Learning for Self-Driving Cars". In: *Proceedings of the International Joint Conference on Neural Networks* 3 (Apr. 2016), pp. 1460–1462. ISSN: 1938-7228. DOI: 10.1109/IJCNN.2005.1556090. arXiv: 1604.07316.

[4] Y Boureau, J Ponce, and Y LeCun. "A theoretical analysis of feature pooling in visual recognition". In: *ICML 2010 - Proceedings, 27th International Conference on Machine Learning* (2010).

[5] François Chollet. "Xception: Deep learning with depthwise separable convolutions". In: *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017* 2017-Janua (2017), pp. 1800–1807. ISSN: 1063-6919. DOI: 10.1109/CVPR.2017.195. arXiv: 1610.02357.

[6] George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. "Improving deep neural networks for LVCSR using rectified linear units and dropout". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, May 2013, pp. 8609–8613. ISBN: 978-1-4799-0356-6. DOI: 10.1109/ICASSP.2013.6639346.

[7] Alessandro Giusti, Jerome Guzzi, Dan C. Ciresan, Fang-Lin He, Juan P. Rodriguez, Flavio Fontana, Matthias Faessler, Christian Forster, Jurgen Schmidhuber, Gianni Di Caro, Davide Scaramuzza, and Luca M Gambardella. "A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots". In: *IEEE Robotics and Automation Letters* 1.2 (2016), pp. 661–667. ISSN: 2377-3766. DOI: 10.1109/LRA.2015.2509024.

[8] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *PMLR* 9 (2010), pp. 249–256. ISSN: 15324435. DOI: 10.1.1.207.2059. arXiv: arXiv:1011.1669v3.

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

[10] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Adam Hartwig. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *ArXiv* (Apr. 2017), p. 9. DOI: arXiv:1704.04861. arXiv: 1704.04861.

[11] Gao Huang, Zhuang Liu, L v. d. Maaten, and Kilian Q Weinberger. "Densely Connected Convolutional Networks". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 2261–2269. ISBN: 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.243. arXiv: arXiv:1608.06993v1.

[12] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, Fernando Mujica, Adam Coates, and Andrew Y. Ng. "An Empirical Evaluation of Deep Learning on Highway Driving". In: *CoRR* (2015). ISSN: arXiv:1504.01716. arXiv: 1504.01716.

[13] Andrej Karpathy. *Convolutional Neural Networks for Visual Recognition*. 2017. URL: http://cs231n.github.io/ (visited on 02/21/2018).

[14] Diederik P Kingma. "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION". In: *Pattern Recognition Letters* 94 (2017), pp. 172–179. ISSN: 01678655. DOI: 10.1016/j.patrec.2017.03.023. arXiv: arXiv:1412.6980v9.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*. 2012. DOI: http://dx.doi.org/10.1016/j.protcy.2014.09.007. arXiv: 1102.0183.

[16] Y LeCun, Urs Muller, Jan Ben, Eric Cosatto, and B Flepp. "Off-road obstacle avoidance through end-to-end learning". In: *Advances in neural information processing systems* 18 (2006), p. 739. ISSN: 1049-5258.

[17] Yann LeCun and Yoshua Bengio. "Convolutional networks for images, speech, and time-series". In: *The handbook of brain theory and neural networks*. 1995, pp. 255–258. ISBN: 9780874216561. DOI: 10.1007/s13398-014-0173-7.2. arXiv: arXiv:1011.1669v3.

[18] Fayao Liu, Chunhua Shen, and Guosheng Lin. "Deep Convolutional Neural Fields for Depth Estimation from a Single Image". In: *Proceedings of the IEEE International Conference on Computer Vision* (2015), pp. 1–13. ISSN: 10636919. DOI: 10.1109/CVPR.2015.7299152. arXiv: arXiv:1411.6387v2.

[19] Raul Mur-Artal and Juan D. Tardos. "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras". In: *IEEE Transactions on Robotics* 33.5 (Oct. 2017), pp. 1255–1262. ISSN: 15523098. DOI: 10.1109/TRO.2017.2705103. arXiv: 1610.06475.

[20] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. *Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks*. 2014. DOI: 10.1109/CVPR.2014.222.

[21] Princeton VisonRobotics. *DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving*. 2018. URL: http://deepdriving.cs.princeton.edu%20http://deepdriving.cs.princeton.edu/ (visited on 02/09/2018).

[22] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: *CoRR* (2017). arXiv: 1710.05941.

[23] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* (2016). ISSN: 0006341X. DOI: 10.1111/j.0006-341X.1999.00591.x. arXiv: 1609.04747.

[24] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C Berg, and Li Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252. ISSN: 15731405. DOI: 10.1007/s11263-015-0816-y. arXiv: 1409.0575.

[25] Ashutosh Saxena, Sung H Chung, and Andrew Y Ng. "Learning Depth from Single Monocular Images". In: *Advances in Neural Information Processing Systems* 18 (2006), pp. 1161–1168. ISSN: 0920-5691. DOI: 10.1007/s11263-007-0071-y.

[26] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* (2017). ISSN: 23289503. DOI: 10.1002/acn3.501. arXiv: 1712.01815.

[27] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations (ICRL)* (Sept. 2015), pp. 1–14. ISSN: 09505849. DOI: 10.1016/j.infsof.2008.09.005. arXiv: 1409.1556.

[28] *Some of the companies that are working on driverless car technology - ABC News*. URL: https://abcnews.go.com/US/companies-working-driverless-car-technology/story?id=53872985.

[29] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958.

[30] Stanford Vision Lab. *ImageNet*. URL: http://image-net.org/ (visited on 03/07/2018).

[31] Oliver Strömgren. "Deep Learning for Autonomous Collision Avoidance". PhD thesis. 2018.

[32] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Vol. 07-12-June. 2015, pp. 1–9. ISBN: 9781467369640. DOI: 10.1109/CVPR.2015.7298594. arXiv: 1409.4842.

[33] Lei Tai, Shaohua Li, and Ming Liu. "Autonomous exploration of mobile robots through deep neural networks". In: *International Journal of Advanced Robotic Systems* 14.4 (July 2017), pp. 1–9. ISSN: 17298814. DOI: 10.1177/1729881417703571.

[34] Lei Tai, Jingwei Zhang, Ming Liu, Joschka Boedecker, and Wolfram Burgard. "A Survey of Deep Network Solutions for Learning Control in Robotics: From Reinforcement to Imitation". In: *CoRR* (Dec. 2016). arXiv: 1612.07139.

[35] Jianxin Wu. *Introduction to Convolutional Neural Networks*. 2017. DOI: 10.1007/978-3-642-28661-2-5. arXiv: 1111.6189v1.

[36] Songtao Wu, Shenghua Zhong, and Yan Liu. *Deep residual learning for image recognition*. 2017. DOI: 10.1007/s11042-017-4440-4. arXiv: 1512.03385.

[37] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. "How transferable are features in deep neural networks?" In: *NIPS'14 Proceedings of the 27th International Conference on Neural Information Processing Systems* (2014). ISSN: 10495258. DOI: 10.1109/IJCNN.2016.7727519. arXiv: 1411.1792.

[38] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* (2012). ISSN: 09252312. DOI: http://doi.acm.org.ezproxy.lib.ucf.edu/10.1145/1830483.1830503. arXiv: 1212.5701.

[39] Fangyi Zhang, Jürgen Leitner, Michael Milford, and Peter Corke. "Modular Deep Q Networks for Sim-to-real Transfer of Visuo-motor Policies". In: *Australasian Conference on Robotics and Automation* (Oct. 2016). arXiv: 1610.06781.

[40] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices". In: *CoRR* (2017). DOI: 10.1109/CVPR.2018.00716. arXiv: 1707.01083.