Master of Science Thesis in Computer Science
Department of Electrical Engineering, Linköping University, 2018

Deep Learning for Autonomous Collision Avoidance

Oliver Strömgren

LINKÖPING UNIVERSITY

Master of Science Thesis in Computer Science

Deep Learning for Autonomous Collision Avoidance

Oliver Strömgren

LiTH-ISY-EX--18/5115--SE

Supervisor: **Abdelrahman Eldesokey**
ISY, Linköping University
Åsa Detterfelt
MindRoad AB

Examiner: **Fahad Khan**
ISY, Linköping University

Division of Computer Vision Laboratory (CVL)
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Abstract

Deep learning has been rapidly growing in recent years obtaining excellent results for many computer vision applications, such as image classification and object detection. One aspect for the increased popularity of deep learning is that it mitigates the need for hand-crafted features. This thesis work investigates deep learning as a methodology to solve the problem of autonomous collision avoidance for a small robotic car. To accomplish this, transfer learning is used with the VGG16 deep network pre-trained on ImageNet dataset. A dataset has been collected and then used to fine-tune and validate the network offline. The deep network has been used with the robotic car in a real-time manner. The robotic car sends images to an external computer, which is used for running the network. The predictions from the network is sent back to the robotic car which takes actions based on those predictions. The results show that deep learning has great potential in solving the collision avoidance problem.

Acknowledgments

First, I would like to thank MindRoad for giving me the opportunity to do this thesis, especially on this subject. In addition, I would like to thank all those who work at MindRoad for welcoming me in the way they have. A special thanks to my supervisor at Linköping University. I really appreciated all the help and feedback on the report and the work. Finally, I would like to thank my girlfriend Emma for being patient with me, for all her support and for pushing me when needed during this time.

Linköping, April 2018
Oliver Strömgren

Contents

List of Figures

List of Tables

Notation

Acronyms

Acronym	Meaning
ML	Machine Learning
SL	Supervised Learning
UL	Unsupervised Learning
DL	Deep Learning
NN	Neural Network
DNN	Deep Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
CNN	Convolutional Neural Network
AI	Artificial Intelligence
MLP	Multilayer Perceptron
GD	Gradient Descent
SGD	Stochastic Gradient Descent
TL	Transfer Learning
SVM	Support Vector Machine

1
Introduction

This thesis is the final part of a Master of Science degree in Computer Science at Linköping University, and it was done in collaboration with MindRoad located in Linköping.

1.1 Background

This thesis work is a continuation on an earlier work that was done at MindRoad company in Linköping. The earlier work resulted in a robotic car that can follow a colored ball autonomously. The robotic car does not take its surroundings into consideration. Therefore, it can not detect obstacles and drives into them. It is desired to make this robotic car smarter through Machine Learning (ML).

One subarea of machine learning that has grown rapidly in recent years is Deep Learning (DL). It has been given huge attention for solving problems that contain learning, e.g. pattern recognition, object detection and image classification. This is due to the emergence of large datasets and good computing power from GPUs. Another aspect for the increase of popularity is that deep learning requires very little engineering by hand, which made it easier to build algorithms, since deep learning methods learn the representations needed automatically. Deep learning has also achieved great success in computer vision, especially Convolutional Neural Networks (CNNs) which made processing of images and high-dimensional data feasible in deep networks. This thesis work will look into deep learning as a way to make the robotic car smarter, by trying to solve the problem of autonomous collision avoidance.

1.2 Purpose

The purpose of this thesis is to investigate deep learning and the use of it for autonomous collision avoidance. As part of the work, a deep neural network is to be chosen and implemented and used with the robotic car for avoiding collisions in real-time.

1.3 Problem Formulation

This thesis deals with the following research questions:

- *Is Deep Learning a good approach for this kind of problem?*

One reason to look at deep learning is that traditional machine learning methods are limited in processing data in its raw form. Deep learning methods on the other hand are representation-learning methods, they can handle data in its raw form and automatically discover the representations needed, e.g for detection and classification [23].

- *What type of network is most suitable for the problem?*

There exist a lot of different networks, which are described in the paper by Schmidhuber [30]. Therefore, a study in the literature will be made to conclude what type of network is most suitable to use.

1.4 Limitations

One factor that will have a quite big impact is the lack of a proper GPU. Instead, a CPU will be used to train and evaluate the network. Thus, our experiments will be conducted on a single network architecture. Another reason for using a single network is due to time constraints. There must be time to implement, train the network and integrate it to be used together with the robotic car as well.

The robotic car is run by a Raspberry Pi Model B+ (version 1) and this does not make it possible to do the implementation directly on the robotic car, as it lacks the computational power needed. Therefore, an external computer is used for running the network.

2
Theory

In this chapter, the theory relevant to the thesis will be presented. Starting with the fundamentals of machine learning[1] and then proceeding with deep learning.

2.1 Machine Learning

Machine learning is a field within computer science, and is a form of applied statistics, which studies algorithms that can learn from and make predictions on data. In earlier years in development of Artificial Intelligence (AI), systems were relying on hard coded knowledge about the world, which were described in formal languages. The systems did reason about these statements in formal languages using logical rules. But it is hard to construct formal rules with enough complexity to describe the world in a correct way [9]. AI systems need the ability to acquire the knowledge on their own by extracting patterns from raw data, and this is known as machine learning.

A machine learning algorithm is able to learn from data. A definition of learning is given by Mitchell [25]:

Definition 2.1 (Learning). *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*

[1] Machine learning is a huge research area and only the relevant parts will be covered here.

The definition 2.1 is quite comprehensive, but one example of a task is object recognition, e.g. learn to recognize faces in images. Here a model will be fed with images and will learn to recognize faces. One way to see if it is actually learning is to calculate how many cases the model is wrong, this would be our performance measure. When the model sees more images, it gets more experience. If the performance measure would drop with more experience, it means the model is learning.

The goal with machine learning is to predict the right outcome from data that is not known before, based on earlier known data. The earlier known data that the model has been trained on can either be unlabeled or labeled. This results in that machine learning algorithms can be categorized into different paradigms. When the training data is labeled, it is called Supervised Learning (SL) and Unsupervised Learning (UL) when it is not.

2.1.1 Supervised Learning

The most common paradigm of machine learning is supervised learning [23]. When we use supervised learning, each training example is labeled by a supervisor who shows the model what to do. An example to illustrate this would be a model which we want to classify images containing either a cat or a dog. First we collect images containing cats and dogs and we label those images with its respective category. When the model is training, it is shown an image and it should predict which category, or class, that the image belongs to. For the model to be able to do this it will for every input produce an output in the form of a vector of scores. This vector will contain scores for both a cat and a dog, and we want the correct class to have the highest score.

Another way to describe in a more formal way is to assume that we have input variables, x, and an output variable y. We then use an algorithm to learn a mapping function from the input to the output:

$$y = f(x) \tag{2.1}$$

The goal is to approximate this mapping function well enough so that new inputs can be mapped to the right output.

Supervised learning can be divided into two different categories, classification and regression [4][20]. In regression, the desired output consists of one or more continuous variables, while in classification each input is assigned to one of a finite number of discrete classes. An example would be if we have an image, then regression could be used to determine how many cats are there in an image. While classification can be used to determine whether there is a cat in an image or not.

2.1.2 Classification

A definition of classification is given in the book by Michie et al. [24], and it is defined as follows:

Definition 2.2 (Classification). *Classification has two distinct meanings. We may be given a set of observations with the aim of establishing the existence of classes or clusters in the data. Or we may know for certain that there are so many classes, and the aim is to establish a rule whereby we can classify a new observation into one of the existing classes. The former type is known as Unsupervised Learning (or Clustering), the latter as Supervised Learning.*

The simplest case for visualizing a classification problem is to imagine we have a set of observations **O** that is to be divided into two different classes, as shown in figure 2.1.

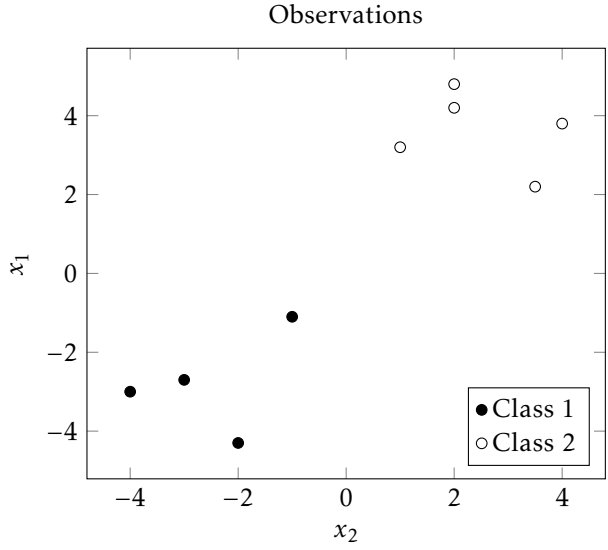


Figure 2.1: *A set of observations.*

We want to find the line that separates the two classes. The line that we are seeking has the following functional form:

$$y = f(w, X) = w^T X + b \tag{2.2}$$

where w is a weight vector and X is the input vector describing one sample in **O**. b is a bias term, which in this case will shift the line left or right along the x_2-axis. For example, in figure 2.1, $X = (x_2, x_1)$ is the coordinates of each observation.

From this we can classify new observations with the help of equation 2.2:

$$y = \begin{cases} \text{Class 1}, & \text{if } f(w, X) < 0 \\ \text{Class 2}, & \text{if } f(w, X) > 0 \\ \text{undefined}, & \text{if } f(w, X) = 0 \end{cases} \tag{2.3}$$

2.1.3 Neural Networks

Neural Networks (NNs) arose from attempts to finding representations of information processing in biological systems. One of the inspirations came from the human brain. However, there are complexities to a biological neural system that are not modeled by a neural network [4][25].

The brain consists of a large amount of neurons that are connected through synapses in multiple ways, which also works in parallel. This view of the brain is the source of the design of a neural network. A neural network is organized in layers and each layer consists of a number of neurons. These layers could be an input layer, hidden layer or an output layer. A hidden layer is a layer that is neither an input nor an output to the network. Depending on the design of the network, there can be one or several hidden layers. Each layer consists of neurons that represent biological neurons in the form of a mathematical function. These neurons can be locally connected or fully connected between layers. Fully connected means that every neuron is connected to every neuron in the previous layer. The edges between the neurons can either be undirected or directed. The latter is the most common one where the edges are directed forward, as the example in figure 2.2. This type of neural network is also called a feedforward neural network, or a multilayer perceptron (MLP) [9][30][25].

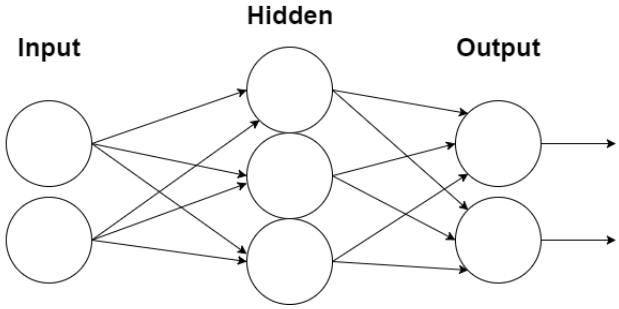


Figure 2.2: *An example of a network (MLP).*

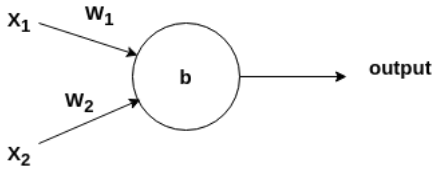


Figure 2.3: *A simplified example of a neuron.*

The goal of feedforward networks is to approximate some function f. If we take a classifier as an example, then it would mean that equation 2.1 maps an input x to a class y. It does this by defining a mapping:

$$y = f(x, w) \tag{2.4}$$

and learns the value of the parameter w that results in the best approximation. Each neuron receives one ore more inputs from the previously layer, see figure 2.3, and sums them to produce an output, the output is also called activation. The activation from a neuron is calculated by the following formula:

$$a_j = f\Big(\sum_i w_i \cdot x_i + b\Big) \tag{2.5}$$

where f is non-linear function, $\cdot$ is dot product, w is neuron's weight, x is the input to the neruon and b is the bias.

The weights in a network represents the importance of the respective inputs to the output. By using these weights each neuron is making a decision by weighing up the results from the previous layer. In this way a neuron in a later layer can make a decision at a more complex and more abstract level than the ones in previous layers.

The purpose of a non-linear function, or activation function, is to map the weighted input to the output (activation of the neuron). There exists several non-linearity functions. For example, the sigmoid, tanh and ReLU functions [15][9]. The sigmoid function takes real numbers as input and squashes them in the range between [0,1]. The mathematical form of the function is as follows:

$$\sigma(x) = \frac{1}{(1 + e^{-x})} \tag{2.6}$$

This means that every large negative value will become 0 and every large positive value will become 1, as shown in figure 2.4. The sigmoid function was used frequently before due to its interpretation of the firing rate of a neuron [15], where 0 represents not firing and 1 for firing. In practice, the function decreased in pop-

ularity because of two major drawbacks. The first drawback is that it saturates and kill gradients. This means that when the neurons activation is close to either 0 or 1, the gradient at these regions is almost 0. This results in that almost no signal will go through the neuron. The second drawback is that the outputs are not zero-centered. This can cause neurons in later layers to always receive only positive inputs.

The tanh function is similar to the sigmoid function. It also squashes the input values, which also are real numbers, but the range for this function is [-1,1], see figure 2.4. As the sigmoid function, tanh saturates but it does not have the drawback of being zero-centered. As can be seen in the equation 2.7 the tanh is just a scaled sigmoid.

$$tanh(x) = 2\sigma(2x) - 1 \tag{2.7}$$

ReLU stands for Rectified Linear Unit, and it has become a very popular activation to use [15]. What the ReLU does is just to put all negative values to 0. In other words, it computes the following function:

$$f(x) = max(0, x) \tag{2.8}$$

One of the positive things about the ReLU is that it is not as computational heavy as the sigmod or tanh. Due to that it can be implemented by just thresholding a matrix at zero. This leads to that the ReLUs typically learns much faster in networks with many layers [23]. One drawback of the ReLU is that it can be fragile during training. If a large gradient comes through the neuron, it could cause the weights to update in such a way that the neuron will never activate again. The gradient coming through will always have the value 0 from that point. What happens is that the ReLU gets knocked off the data manifold [15].

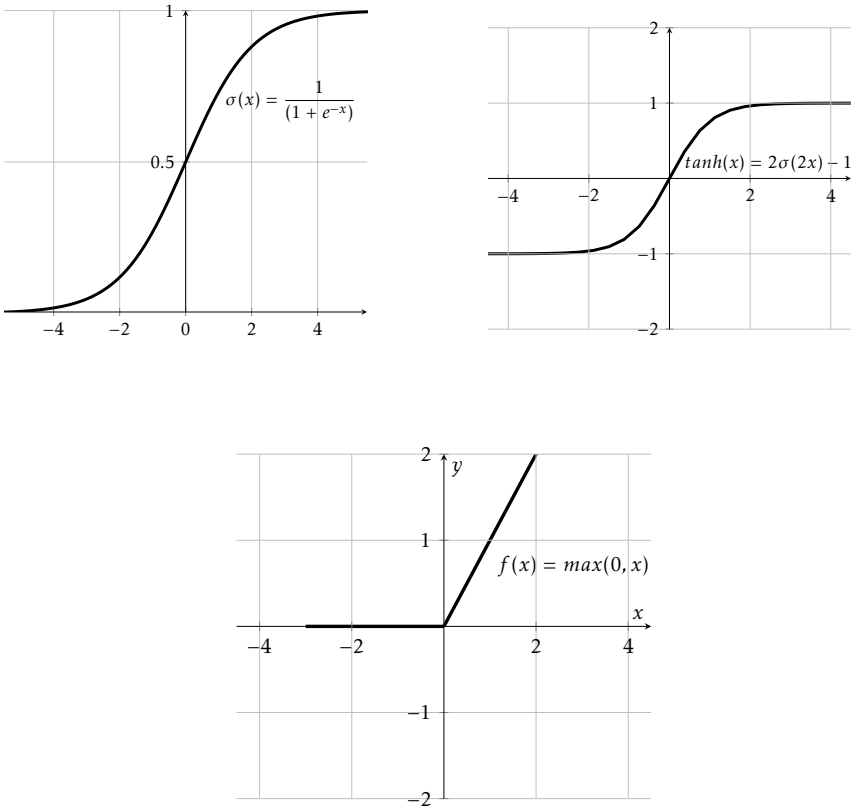


Figure 2.4: *Representation of the different activation functions. Top-left: sigmoid, top-right: tanh and bottom: ReLU.*

More examples on activation functions are described in the online course from Stanford [15], the book by Goodfellow et al. [9] and the book by Gonzales et al. [8].

2.2 Deep Learning

Deep learning has become very popular due to its ability to generalize well, which traditional machine learning algorithms fail to do [9]. If a model is good at generalizing it is good at predicting new unseen data. We can also say that they exhibit small generalization error, which is the difference between training error and test error. The error is usually measured as a percentage of wrong predictions. The problem with generalization is that it becomes more difficult when working with high-dimensional data. Traditional machine learning algorithms are insufficient to learn complicated functions in these high-dimensional spaces, and deep learning is able to overcome this problem.

As mention in section 1.3, deep learning is a subfield of machine learning, and deep learning methods are able to process data in its raw form and automatically discover the representations needed in the data. This is something that is called representation-learning, and conventional machine learning methods has limited ability to do it [23]. This is why deep learning requires very little engineering by hand. The way it works is that it attempts to learn high-level abstractions in the data by utilizing hierarchical architectures. An example of a hierarchical architecture is neural networks. In deep learning, neural networks are called Deep Neural Networks (DNNs), and the difference between a "regular" and deep network is that in a deep network there are many more hidden layers. The number of layers may vary between tens, hundreds and thousands. [10][30][21]. Figure 2.5 shows an example of a deep network.

2.2.1 Deep Neural Networks

Deep neural networks can learn the representation in data by themselves and thereby learn what features are important[2]. The question is, how are they actually learning?

Recall the weights that was introduced in equation 2.5. These weights are representing the importance of the respective inputs to the output. When we say that a network is training, it is the process of fitting the model to the data by optimizing, or adjusting, the weights in the model. This is done by finding the weights that minimizes an error function E, also called loss function. If we would find a perfect fit for the model it would mean that $E = 0$, otherwise $E > 0$.

[2]This is also the case for neural networks described in section 2.1.3, but the theory behind it is covered here instead.

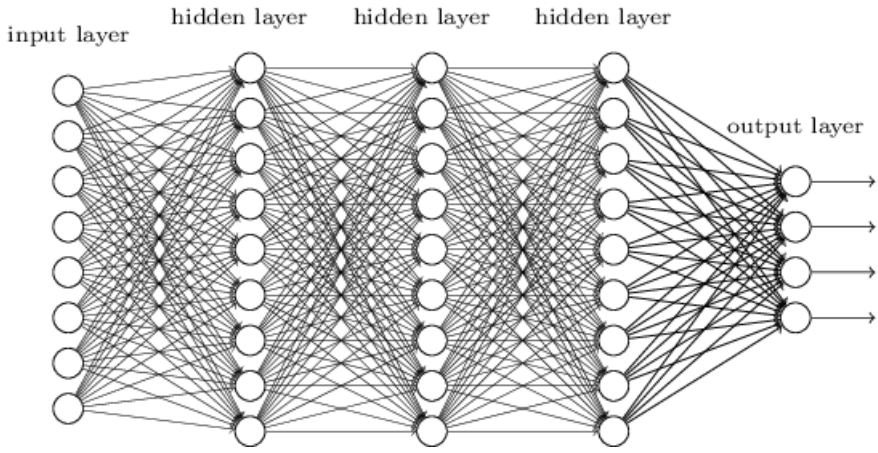


Figure 2.5: *An example of a deep neural network (DNN).*

One commonly used loss function is the cross-entropy function [9][13]. This function is based on maximum likelihood estimations, which means that the loss function is the negative log-likelihood:

$$E = -log\Big(p(y|x)\Big) \tag{2.9}$$

where x is the input and y the labels.

The function measures how good the fit is between two distributions. It tries to approximate the mapping between data and their corresponding labels.

Usually classification is represented using one-out-of-N output encoding. It means that one output node is allocated for each class we have in the data. The targets are then represented by a vector $\boldsymbol{y_i}$, which has all $0's$ except for a 1 at the index that indicates the correct class [25][4]. A classifier that works in a similar way is the softmax. It interprets the scores in the output vector and normalizes the values so that the probabilities sum to one [13]. The softmax classifier uses the cross-entropy as its loss function and it takes a vector of real values and squashes them into a vector of values between 0 and 1:

$$f_j(x) = \frac{e^{x_j}}{\sum_k e^{x_k}} \tag{2.10}$$

Forward- and Backpropagation

A network tries to adjust the weights when it is learning. It can also be seen as the network trying to adjust the weights based on the contribution of each neuron to the total loss value. The weights will, therefore, be adjusted so that the total loss value is minimized. This is done in two stages. In the first stage, the derivatives of the loss function with respect to the weights are calculated. Then calculated derivatives are used in the second stage, where we do adjustments to the weights.

Each neuron in the network outputs a_j, which is a weighted sum of its inputs and is calculated as in equation 2.5 where x is the input and w is the associated weight to the input. This sum is then transformed by a nonlinear activation function $f(\cdot)$, which gives the output a_j. Now suppose that we have calculated the activation for all hidden and output neurons in the network according to equation 2.5, this process of doing so is what we call forward propagation [25]. Since it can be seen as a forward flow of information.

Forward propagation does not include any learning. To allow the network to learn, the derivative of the loss with respect to the input of a neuron should be computed by working backwards from the derivative with respect to the output of the same neuron. It can be applied repeatedly to propagate derivatives through all neurons starting from the output, and this is called backpropagation [17][11][23]. The backpropagation algorithm is an algorithm that computes the chain rule of calculus, with a specific order of operations. The chain rule has the following form:

$$\frac{\partial E}{\partial w_{ji}} = \sum_{i=1}^{N} \frac{\partial E}{\partial x_j} \frac{\partial x_j}{\partial w_{ji}} \tag{2.11}$$

where E is the loss, w are the weight and x is the input. See below for a simplified example to calculate the derivatives.

$$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k} \tag{2.12}$$

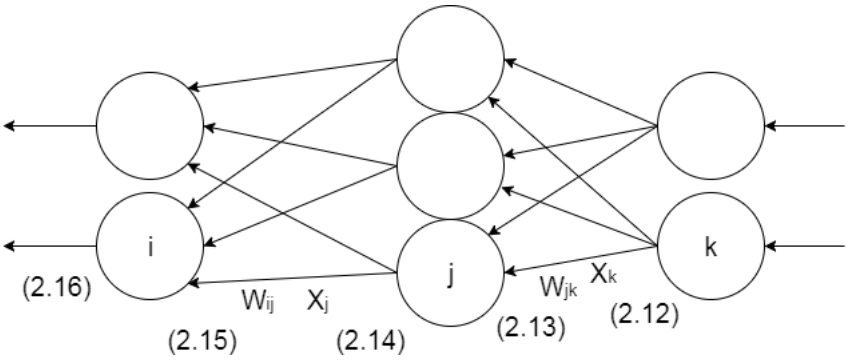
$$\frac{\partial E}{\partial y_j} = \sum_k w_{jk}\frac{\partial E}{\partial z_k} \tag{2.13}$$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j} \tag{2.14}$$

$$\frac{\partial E}{\partial y_i} = \sum_k w_{ij}\frac{\partial E}{\partial z_j} \tag{2.15}$$

$$\frac{\partial E}{\partial z_i} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_i} \tag{2.16}$$

E is the error, y_k is the output of neuron k where $y_k = f(z_k)$ for some linear function $f(\cdot)$, $z_k = \sum(w_{jk} \cdot x_k)$, w_{jk} are the weight on the connection from neuron j to k and x_k is the input of neuron k.



Stochastic Gradient Descent

In the second stage we use the calculated derivatives to adjust the weights by using a training algorithm. A popular algorithm is called Stochastic Gradient Descent (SGD) [4], a procedure to repeatedly evaluate the derivative and perform adjustment of the weights. Stochastic Gradient Descent is a variation of Gradient Descent (GD), which is an optimization algorithm for finding the minima of a cost function [16]. Gradient Descent finds a local minimum by following the gradient calculated with the backpropagation at a current point in time. In other words, it is trying to find the minimum loss by minimizing the loss function. We do this by computing the loss function over the entire training data and perform a single weight update with steps proportional to the negative of the derivative.

One problem with Gradient Descent is that it is computationally expensive to compute the loss function over the entire data [16]. That is why Stochastic Gradient Descent was introduced. The difference is that the gradient is calculated over batches instead and the batch is used to perform a weight adjustment. The batch sizes are usually in powers of 2. Then based on the batch's loss, weights are updated as following:

$$W_{t+1} = W_t - \eta \sum_{m=0}^{M-1} \frac{\partial E_m}{\partial W_t} \tag{2.17}$$

where W_t is the current weight at time t, η is the learning rate, or step size, M is the mini-batch size and E_m is the loss for the m-th training sample.

Regularization

A problem that arises with these networks is how to make them perform well on new unseen data and not just on the training data. The approach to deal with this are to use methods that are designed to reduce the test error, which are known as regularization. In the book by Goodfellow et al. [9] they give the following definition:

Definition 2.3 (Regularization). Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.

Two common terms when dealing with deep neural networks are under- and overfitting. Underfitting happens when the model is not able to get a low enough error value on the training set. While overfitting occurs when the gap between the training error and the validation error is too large. When we say that overfitting occurs we can also say that the model has memorized the training data and will therefore perform badly on new unseen data. This is one reason why overfitting often occurs when the dataset is small. Regularization is one way of preventing the model to overfit, besides splitting the dataset up into a training and a validation set.

One common regularization method is the use of dropout [23], it has been shown that it is a good method for preventing overfitting and improving the performance of neural networks [33]. Dropout refers to dropping out neurons, this means that the neurons in the network will temporarily be removed together with all its incoming and outgoing connections. Which of the neurons that are removed are chosen randomly, where each neuron is kept with a fixed probability. The probability is often set to 50% as it seems to be close to optimal for a variety of networks and tasks [33]. Applying dropout can be seen as training several thinner networks, since for every new batch different neurons will be trained. Dropout also prevents co-adaptions between the neurons. Co-adaption happens when a neuron change in a way to fix mistakes of other neurons. This leads to overfitting due to that co-adaptions do not generalize well to new unseen data.

Dropout prevents this by making the presence of other neurons unreliable. This results in that a neuron is unable to rely on other neurons to fix its mistakes. It makes the neurons more robust to noise in the data.

There exists many forms of regularization, see the book by Goodfellow et al. [9], the paper by Michie et al. [25] and the book by Bishop [4] for more comprehensive descriptions and examples of different methods.

2.2.2 Convolutional Neural Network

One very common deep neural network to use that has proven to be very successful in many classification tasks is the Convolutional Neural Network [21][10][35]. Convolutional neural networks work in similar way as a regular deep neural network. The difference is that the architecture of the network makes the assumption that the inputs are multidimensional data. The regular deep neural networks does not take spatial data structures into consideration. For example, in images nearby pixels are more correlated than pixels far away from each other, this has to be taken into consideration [10][23].

A convolution neural network make use of convolution, which is a special kind of linear operation. The architecture of a typical convolutional neural network is structured as a series of stages, whereas each stage consists of convolution and pooling layers as well as non-linearities [23][22], see figure 2.6. In the earlier layers, the network learns to recognize low-level features like edges in a picture and it combines these features into more complex high-level features in later layers.

The equations that are used in a convolutional layer has the form:

$$a^k = f\Big(\sum w^k * x + b^k\Big) \tag{2.18}$$

$$y^k = maxpool(a^k) \tag{2.19}$$

where x is the input, w^k is the weights of the k-th convolutional filter, $*$ is a convolution operator and b is the bias. These are summed up and passed through a non-linearity, usually a ReLU and this creates a feature map a^k. This is passed to the max pooling and the output from the pooling is passed to the next layer.

When we talk about convolutional neural networks we introduce three mechanisms, namely *local receptive fields, weight sharing and subsampling*. In a convolutional layer, each neuron is connected only to a local region of the input, i.e. the neighbouring neurons in a region. In contrast to a fully-connected layer where each neuron is connected to each neuron in the previous layer. It is this region that is called the *local receptive field*. The receptive field enables the neurons to extract features such as edges and corners in an image. In a convolutional layer, the neurons are organized in planes, which are called feature maps. The neurons that belongs to the same feature map are *sharing weights*. If we think of the neurons as feature extractors, all neurons in the same feature map will detect

the same pattern but at different locations in an image. This is the reason why they are sharing weights since the same pattern, or object, can be anywhere in an image. There are generally several feature maps since we usually need to detect different patterns in an image, each feature map will have its own set of weights. The set of weights is often referred to as *filters* or *kernels*. For each feature map in the convolutional layer, there is a plane of neurons in the pooling layer. The pooling layer has two purposes, the first one is to progressively reduce the spatial size to reduce the amount of parameters and computations in the network. The second is to the make the network more invariant to small permutations. The most used pooling operation is the max pooling which saves the maximum of the input numbers from a small region in a feature map, see example 2.4, [14][4][22].

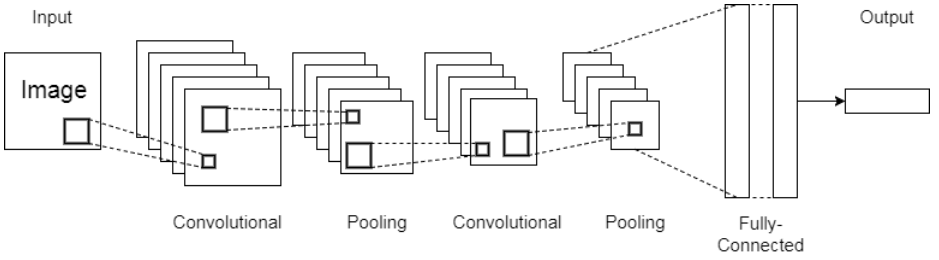


Figure 2.6: *A typical architecture for a convolutional neural network (CNN).*

Example 2.4: Example of max pooling

To the left is a feature map from the convolutional layer and to the right the result when it goes through the max pooling layer. This is done by usually taking a 2x2 filter and sliding it over the feature map. As can be seen is the feature map reduced in size and the maximum in each quadrant is saved.

2.2.3 Transfer Learning

For a model to get sufficient accuracy it requires a lot of training data. One way to solve this is the use of Transfer Learning (TL). Transfer learning is when a network, or model, is first trained on a dataset and a specific task, and then used to train on a different dataset and task [37][27]. This works because when a network is trained on images, the first layers tend to learn low-level features, i.e. edges, lines and corners. It occurs regardless of which dataset is used, therefore these features are called general features. Since it is relatively rare to have a dataset that is big enough, and it can take two to three weeks to train a modern convolutional neural network across several GPUs [18], it is common to use a pretrained network that has been trained on millions of images and use it for the task ahead.

There are two common approaches used in transfer learning. The first one is to use a pretrained network as feature extractor, and the second is to use it with fine-tuning [18][37]. When it is used as a feature extractor, the last layer is removed from the pretrained network and the output, from the layer that becomes last after the removal, is used as features for a trainable classifier. Usually it is the classifier that is removed, and this way the pretrained network can be used for classifying other classes than it was trained to classify. When the fine-tuning approach is used, not only the classifier that is replaced but the weights in the earlier layers are used for weight initialization. Then a new classifier is placed on top and the network is retrained with the new dataset. It is called fine-tuning since the weights are not trained from scratch, but they are adjusted from the point they were in the pretrained network to fit the new dataset. It is also possible to freeze the earlier layers, which mean that they are not trainable, since the earlier layers consists of general features they can be kept as they were. In this case, only the more specific features to the old dataset are adjusted to better fit the new task and dataset.

3
Method

This chapter starts with an overview of the work that this thesis work is a continuation of. After that, it continues with the description of the network used.

3.1 Overview of the Robotic Car System

The robotic car that was used in this thesis work, see figure 3.3, consists of three subsystems: a Raspberry Pi Model B+, a microcontroller and a remote client software. The Raspberry Pi's purpose is to function as the middleware in the system and it has two main tasks. The first main task is to continuously send images from the camera, which is a Raspberry Pi Camera Module, to the client software. The second task is to send movement instructions received from the client software to the microcontroller. The client software can be seen in figure 3.2. The Raspberry Pi runs a version of the Linux distribution Raspbian and the software is written in C and uses OpenCV (version 3.0) as its imaging library [36]. A library is also used for the camera module to be able to work with OpenCV, the library is called raspicam_cv [28]. It communicates with the client software over Wifi and with the microcontroller through a serial port over USB. A flow diagram of the system can be seen in figure 3.1.

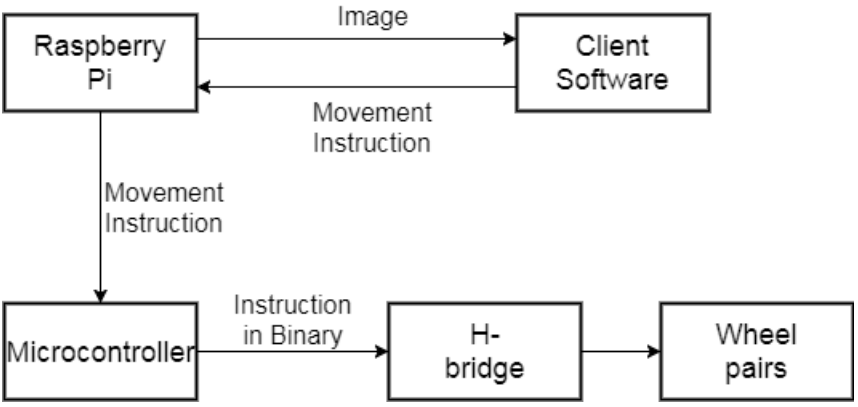


Figure 3.1: *Flow diagram of the system.*

The microcontroller is an Arduino Nano compatible unit (ATmega328P), with a preinstalled Arduino bootloader that controls the motors on the robotic car. It is programmed with the official Arduino software. Its task is to send control signals to the motor for right and left wheel pair. Each wheel pair can be controlled separately and the velocity is controlled with the help of pulse width modulation. It is an H bridge that controls the motors and enables the steering of the wheel pairs in both directions, by writing the right corresponding binary number to the wheel pairs.

The client software is written in C++ and only works in a Windows environment. The software is also using OpenCV (version 3.0) as its imaging library. Through the software we can connect to the robotic car, see the video stream received from it and remotely control it. The robotic car is remotely controlled by the arrow keys on a keyboard. There are other settings that can be controlled through the client software but that will not be covered here.

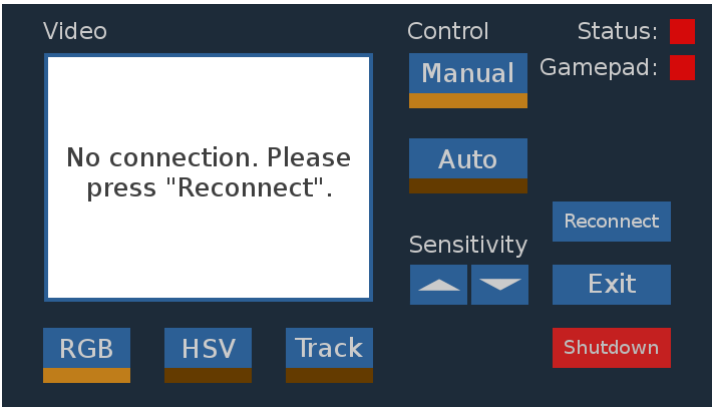


Figure 3.2: *Image of the client software.*

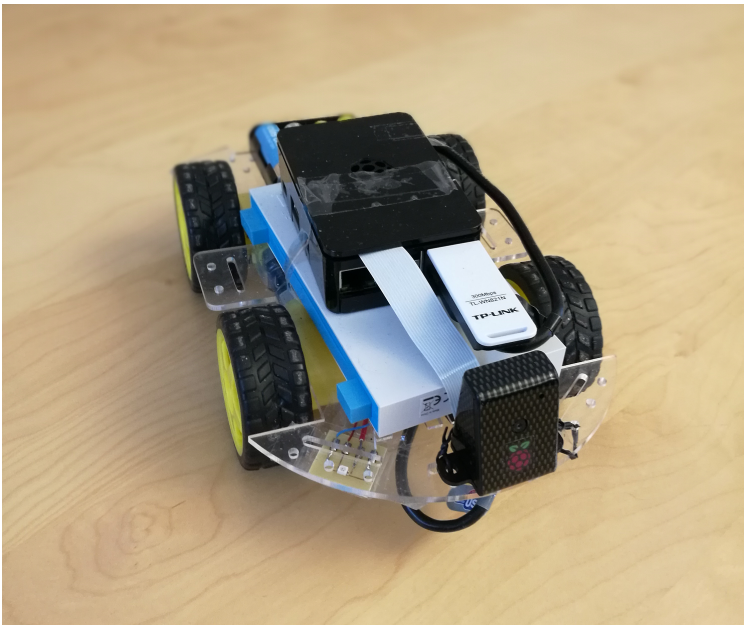


Figure 3.3: *Image of the robotic car used in the thesis work.*

The full documentation and source code are available on Github[1] [34]

3.1.1 Communication

There are two communication protocols developed for the robotic car, one is for the network communication and the other for the communication between the Raspberry Pi and the microcontroller. The video stream is done separately from the protocols. The network communication between the robotic car and the client software is over UDP, hence the robotic car should be able to continuously send images without the need to wait for confirmation from the client software that it has received an image.

The video stream is simulated by the Raspberry Pi sending approximately 30 frames per second to the client software. The images are sent uncompressed over the network and has a resolution of 160 x 120 pixels, they are rescaled in the client software to 320 x 240 pixels to make it easier to see in the graphical interface. Every image is saved in a OpenCV structure called IplImage, which includes the information about the size of the image, the image data and number of color channels.

The communication protocol allows each instruction to be 16 bytes, with a header and up to 15 bytes of data per instruction. Each instruction must contain a header and 0-15 bytes of data. The header contains information about what kind of instruction is being sent, this is the first four bits, and how many bytes of data

[1] Unfortunately, the documentation is only available in Swedish

the instruction contains, which is the last four bits. This allows a total of 16 instructions. The instructions sent from the client software when controlling the robotic car remotely can be seen in table 3.1. The protocol between the Raspberry Pi and the microcontroller works the same way, the only difference is what type of instructions are being sent.

Table 3.1: *The instructions that are sent when controlling the robotic car.*

Instruction	**Bytes sent**
`Forward`	00100010 01111111 00000000
`Forward-right`	00100010 01111111 01100000
`Forward-left`	00100010 01111111 10100000
`Backward`	00100010 10000000 00000000
`Backward-right`	00100010 10000000 01100000
`Backward-left`	00100010 10000000 10100000
`Rotate right`	00100010 00000000 01111111
`Rotate left`	00100010 00000000 10000000

3.2 The Choice of the Deep Network

The first part of the thesis work was to investigate deep learning and determine what type of network that would be used. Basically there are three architectures that are most popular. The architectures are: feedforward neural network, Recurrent Neural Network (RNN) and the convolutional neural network. The recurrent neural network is similar to a feedforward network, the difference is that a recurrent network is cyclic [30] in contrast to a feedforward network which is acyclic. This gives them the ability to create and process memories of arbitrary sequences of input patterns. It does this by keeping a vector in the hidden neurons that keep track of states, the vector contains information about the history of all past elements of a sequence [23].

A reason why a recurrent neural network was not chosen is because they have proven to be problematic to train, the reason is that the gradients typically explode or vanish [23][3]. This is called the exploding gradient problem and the vanishing gradient problem respectively. What happens when the gradients are vanishing is that the neurons in earlier layers in the network are learning much slower than neurons in later layers. This causes the gradient to be smaller when we move backwards through the hidden layers. What happens in the exploding gradient problem is the opposite, when we move backwards through the hidden layers the gradient is instead growing exponentially. There are various ways to

handle these problems, such as using an Long Short-Term Memory (LSTM) network or gradient clipping. LSTM networks use a special kind of memory unit, and gradient clipping means keeping the gradients between two numbers to prevent them from getting too large. Another reason is that a RNN takes longer time to train, since they are more complex and more computations are done in the network.

One motivation for not going with a regular fully-connected network is the amount of parameters that needs to be learned by the network, which would take a long time to train. Another motivation is that the convolutional neural network is a type of a deep fully-connected network, which is easier to train and generalizes better than a network with only fully connected layers [23]. A convolution neural network is also by design made to handle inputs that have spatial structures, like images. Since 2012, convolutional neural networks have become very common to use in computer vision [35][23], and they also give good results in various benchmarks for classification tasks [29]. In the paper by Sharif Razavian et al. [31] they experiment with a convolutional neural network for different recognition tasks, and came to the conclusion that *"Thus, it can be concluded that from now on, deep learning with CNN has to be considered as the primary candidate in essentially any visual recognition task."*

3.2.1 Related Work

There are two papers that are related to this thesis work, the first one is by Lecun et al. [26] and the second by Bojarski et al. [5]. In the first paper they describe a vision-based obstacle avoidance system for off-road mobile robots. The difference here, compared to this thesis work, is that they use an outdoor environment, and they also use two wireless color cameras. In the second paper, they use three cameras and at the same time collecting the steering angle when acquiring data. What these two have in common is that they are using a convolutional neural network. This also adds to the motivation of why to use a convolution neural network.

3.3 Implementation details

The main task for the network was to do classification. Since the instructions that are sent from the client software to the robotic car are a finite set, these can be represented as classes. The network was therefore given the task to classify images to one of these classes. Only five of the instructions were used, every instruction that causes the robotic car to go backwards were decided not to be used. The motivation is that it will be problematic to predict if an image belongs to the instruction of going forward or backwards, these images will look the same but will belong to two different classes. Instead of moving backwards the robotic car need to rotate.

This section is divided into three parts, collecting of data, the implementation and training of the network and the last part is about how to make the robotic

car drive according to the predictions from the network.

3.3.1 Collecting the Data

The robotic car was used to collect the data required to train the network. The code for the client software was modified to be able to save the images that were received from the robotic car. The images that were saved are the unscaled RGB images with a resolution of 160 x 120, with the motivation that it is these images that the robotic car actually sees. Examples of an typical image can be seen in figure 3.4. To be able to label the saved images, the code was also modified to save a text document containing the name of the image and which instruction that was applied while capturing the image.

Figure 3.4: *Examples of images sent by the robotic car.*

When collection the data, a person was driving the robotic car at different times of the day and with different lighting conditions to be able to get a variety of samples. When navigating the robotic car, it was done by watching the video stream in the client software. The driver also tried to have a consistent driving behavior as possible. When there was no obstacle the driver drove straight a head and only turned right or left in the presence of an obstacle.

If an image is flipped horizontally, it becomes a mirror image to the original image. Due to this property was a script written in Python that took an image that was labeled either with turn right or left and flipped it horizontally. This created a lot more of samples for these two classes without the need to actually drive the robotic car.

3.3.2 Network Training

For the implementation and training part, Keras [6] and TensorFlow [1] were used. TensorFlow is an open-source software library for machine intelligence, or rather numerical computations. Keras is a high-level neural networks API written in Python, and runs on top of TensorFlow. Everything was done in a Windows environment. Using an Intel(R) Core(TM) i7-4790 CPU with a processor frequency at 3.60 GHz.

Since it takes a long time to train a convolutional neural network from scratch, even with a GPU, transfer learning was used. The approach used was fine-tuning, which is described in section 2.2.3. Keras has several networks that are available alongside pretrained weights that can be used for prediction, feature extraction and fine-tuning. Every model for image classification that is available is trained on ImageNet, which is a large image database [7]. The chosen model was the VGG16 network that contains 16 layers with trainable weights, and it generalizes well to a wide range of tasks and datasets [32]. Worth mentioning is that other models were tried out too, but some of them have a restriction that they need images larger than 160 x 120, and others took much longer time to train than the VGG16 model. In the latter case the models contained more weights, which caused the longer training time.

The top of the VGG16 model contains fully-connected layers and a softmax classifier. These were removed and replaced with fully-connected layers containing less neurons than the original layers. Another classifier was added, since the original network is trained for predicting 1000 classes and only 5 was needed for this work. Cross-entropy was used as loss function together with a softmax classifier. Stochastic gradient descent was used, with learning rate set to $1 \cdot 10^{-4}$ and momentum to 0.9. There were also an attempt to use a Support Vector Machine (SVM) [19] instead of softmax. A visual representation of VGG16 can be seen in figure 3.6.

The top of the model was first fed with with the bottleneck features, the activation maps before the classifier, that was obtained by running the data through the convolutional layers of the VGG16 model. After that the model was fine-tuned with the first 11 trainable layers in the VGG16 model frozen.

When training a network, there are a lot of hyperparameters to choose that control the models behavior, like batch size, number of neurons to use in a layer and learning rate to name a few. There exists no good method for selecting these hyperparameters, since it depends on the task and dataset at hand. Some recommendations exist for how to set these parameters like in the paper by Bengio [2], therefore some experiments were made in attempt to increase the accuracy and decrease the loss of the network. Table 3.2 summarizes the different experiments. To the left in the table is the name of the model. For simplicity they are named *model#* and to the right are the layers and different settings.

The notations in table 3.2 are described as they are in Keras, a dense layer for example is a fully connected layer. In every configuration the network was trained for 50 epochs, one epoch is one pass through the training set. Different sizes of the dataset was used and split as close to a ratio of 80/20 as possible, i.e. 80% of the data for training and 20% of the data for validation.

3.3.3 The Robotic Car

Two things were done to make the robotic car able to drive according to the predictions of the network. First, the implementation of the client software was modified. Secondly, a Python program was implemented for doing the predictions. A flow diagram can be seen in figure 3.5.

The implementation of the client software was modified to also send the images to a Python program. The Python program loads the trained network and receives images sent from the client software. Each image is then fed to the network that outputs a prediction of what class it thinks the image belongs to. The result from the network is then sent back to the client software, which checks the result and based on that sends the right movement instruction back to the robotic car. The communication between the client software and the Python program is done through sockets, were the Python program function as a server and the client software as a client. The model that gave the best result was the one used when testing the robotic car.

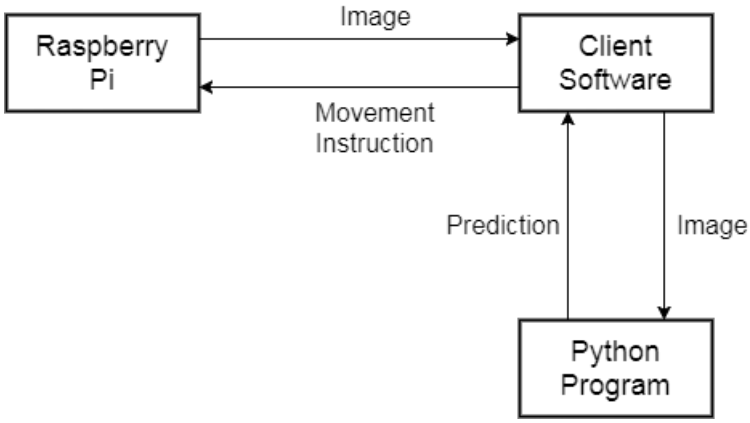


Figure 3.5: *Flow diagram of the communication between the robotic car and the external programs.*

Table 3.2: *Settings for the different experiments.*

Model configurations			
model#	**Layer(#neurons,activation)**	**Batch size**	**Learning rate**
*model*1	Dense(256, ReLU) Dropout(0.5) Dense(256, ReLU) Dense(5, softmax)	16	$1 \cdot 10^{-4}$
*model*2	Dense(256, ReLU) Dropout(0.5) Dense(256, ReLU) Dense(5, softmax)	32	$1 \cdot 10^{-4}$
*model*3	Dense(256, ReLU) Dropout(0.5) Dense(256, ReLU) Dropout(0.5) Dense(5, softmax)	16	$1 \cdot 10^{-4}$
*model*4	Dense(256, ReLU) Dropout(0.5) Dense(256, ReLU) Dropout(0.5) Dense(5, softmax)	32	$1 \cdot 10^{-4}$
*model*5	Dense(256, ReLU) Dropout(0.5) Dense(256, ReLU) Dropout(0.5) Dense(5, softmax)	16	$1 \cdot 10^{-4}$
*model*6	Dense(256, ReLU) Dropout(0.5) Dense(256, ReLU) Dropout(0.5) Dense(5, softmax)	32	$1 \cdot 10^{-4}$
*model*7	Dense(256, ReLU) Dropout(0.5) Dense(256, ReLU) Dropout(0.5) Dense(5, softmax)	64	$1 \cdot 10^{-4}$
*model*8	Dense(256, ReLU) Dropout(0.5) Dense(256, ReLU) Dropout(0.5) Dense(5, SVM)	64	$1 \cdot 10^{-4}$

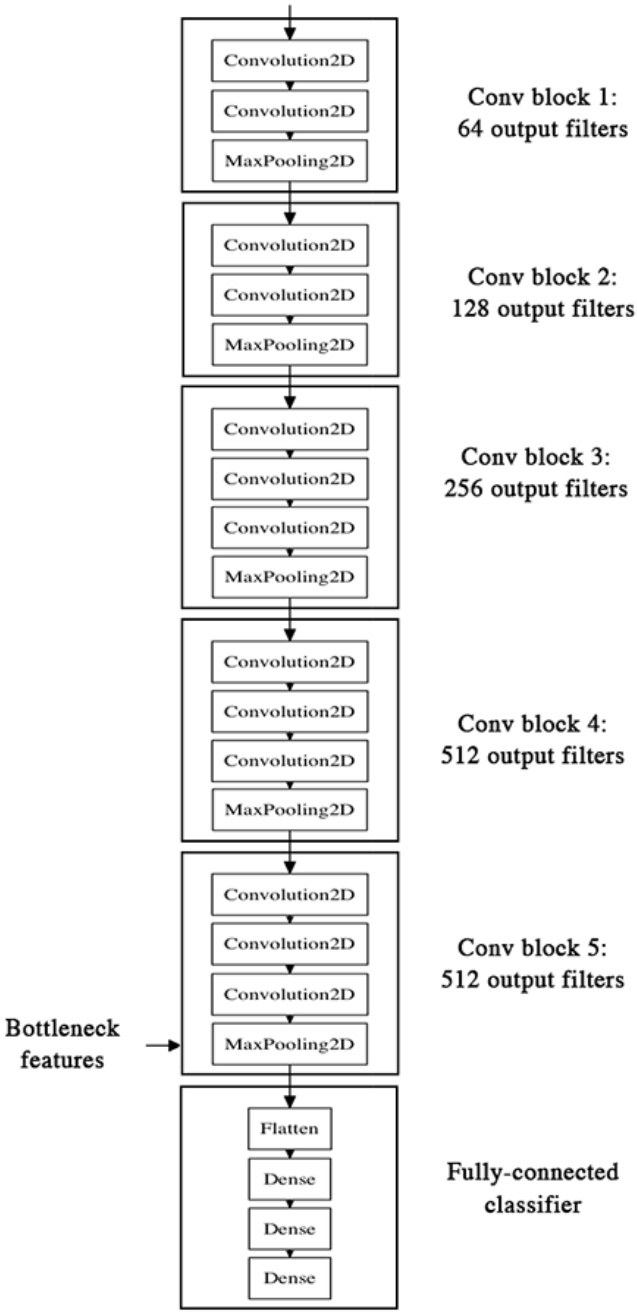


Figure 3.6: *The VGG16 model in Keras.*

4
Results

This chapter will present the results from different convolutional neural network models when evaluated offline and real-time on the robotic car.

4.1 Dataset

Two datasets were collected at two different occasions. The second dataset is an extension of the first. It was expanded to get a greater variety of images, such as difference in lighting and more samples of how to avoid a collision. The number of images in each dataset can be seen in table 4.1 below and examples from two sets can be seen in figure 4.1.

Figure 4.1: *dataset*1 *to the left and dataset*2 *to the right.*

Table 4.1: *The two datasets used.*

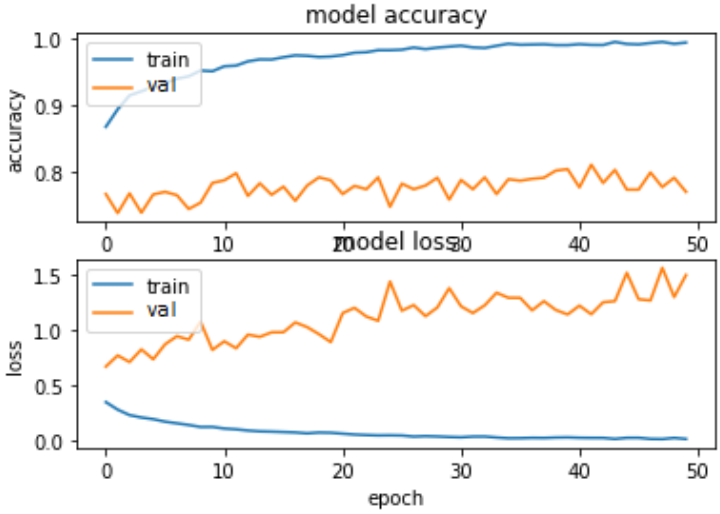
Amount of data			
dataset#	**Total**	**Training**	**Validation**
*dataset*1	7584	5792	1792
*dataset*2	13241	10593	2648

4.2 Quantitative Results

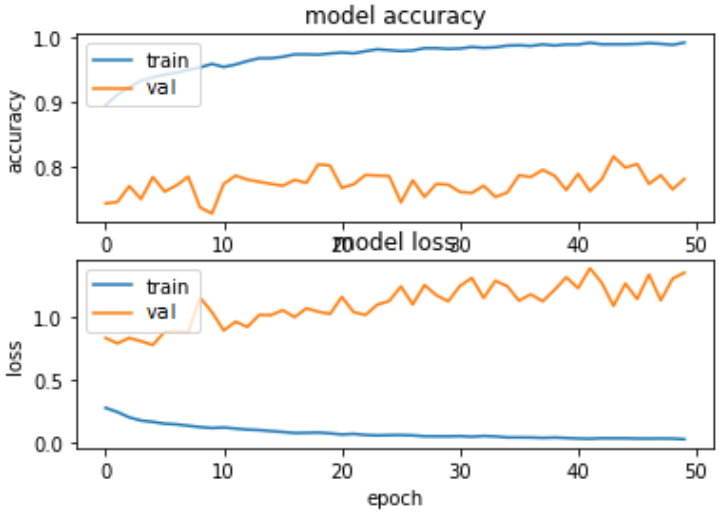
Different models that were described in previous chapter are evaluated on our datasets. A summary of the achieved accuracies are shown in table 4.2. Additionally, figure 4.2 shows the learning curves for different models while training with respect to the training and the validation sets. In the graphs the loss and accuracy are given on the y-axis and epochs on the x-axis.

Table 4.2: *The network's results.*

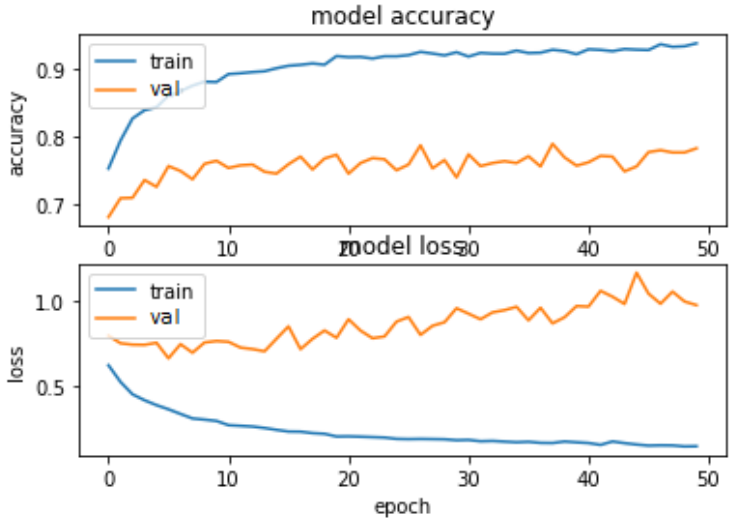
Validation results			
model#	**Accuracy(%)**	**Loss**	**dataset#**
*model*1	76.00	1.5972	*dataset*1
*model*2	77.40	1.3339	*dataset*1
model3	77.62	0.8929	*dataset*1
model4	**78.21**	**0.8837**	**dataset1**
*model*5	81.36	0.8554	*dataset*2
*model*6	82.15	0.9736	*dataset*2
model7	**82.44**	**0.8602**	**dataset2**
*model*8	51.08	0.8029	*dataset*2



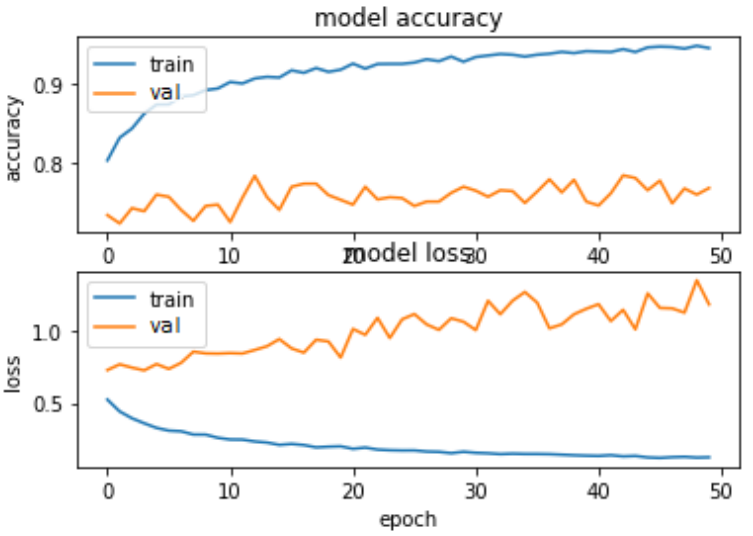
(a) *model1*



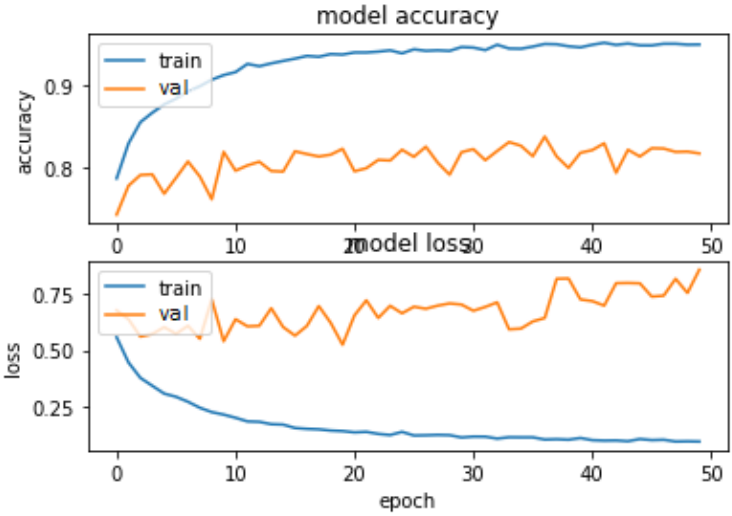
(b) *model2*



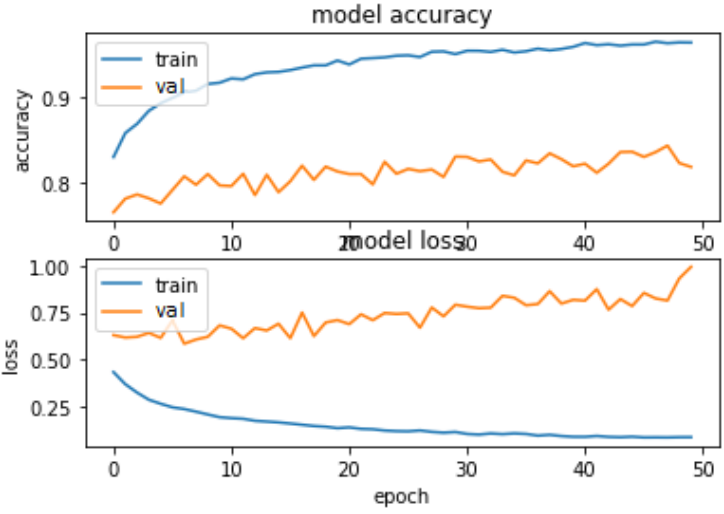
(c) model3



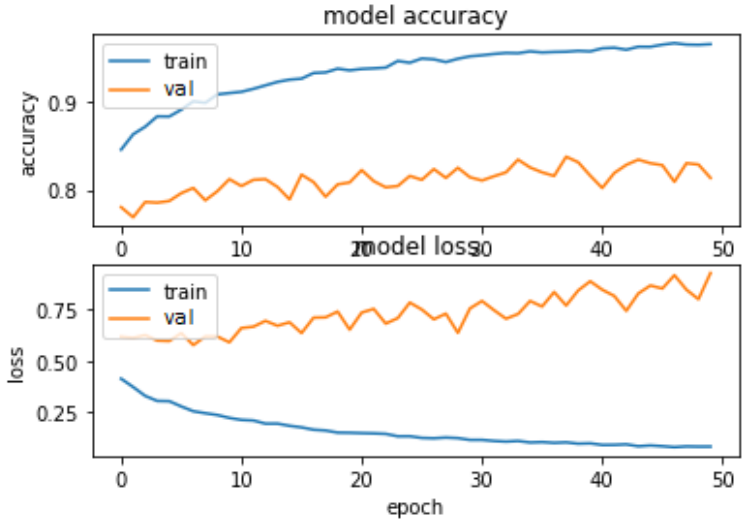
(d) model4



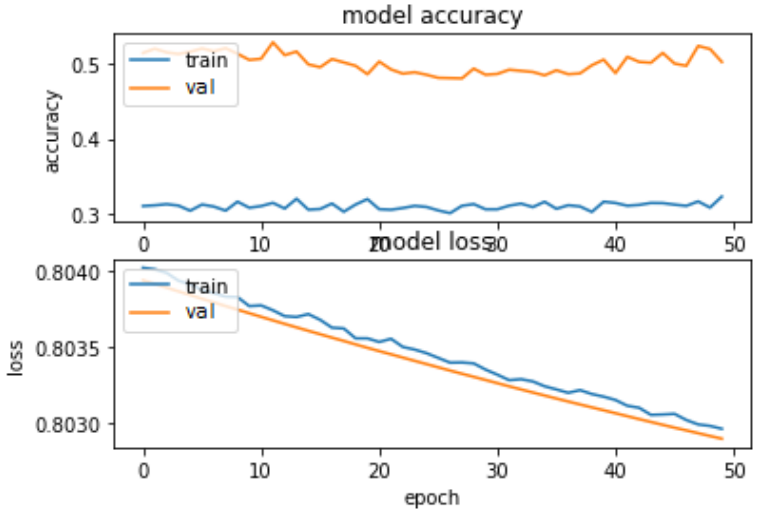
(e) *model5*



(f) *model6*



(g) model7



(h) model8

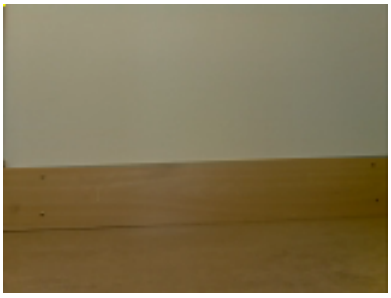
Figure 4.2: *Accuracy and loss per epoch for the models. (a)-(h) are models 1-8 respectively.*

4.3 Qualitative Results

The robotic car was tested by driving around in the office and the obstacles that were used were the same as the network had been trained on, shown in figure 4.3. The number of passed tests for each obstacle together with an overall score can be seen in table 4.3. A test was considered passed if the robotic car did not hit the obstacle, and a total of 91 tests were made.

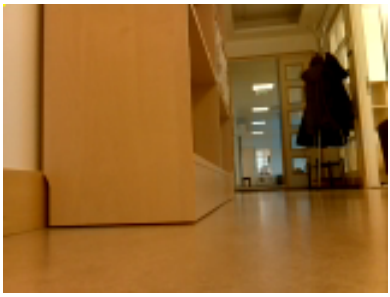
Table 4.3: *Scores from different test cases with model7 used.*

Test results individually	
Obstacle	**Passed(%)**
wall	90.00
chair leg	69.23
bookshelf	66.67
bar stool	52.94
couch	100.00
bin	57.14
conference table	83.33
backpack	70.00
fire extinguisher	91.67
Test results total	
Passed	72.53%



(a) *wall*

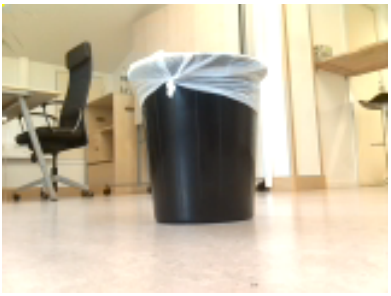
(b) *chair leg*



(c) bookshelf

(d) bar stool

(e) couch



(f) bin

(g) conference table

(h) backpack

(i) fire extinguisher

Figure 4.3: *The obstacles used in the tests. Sorted according to table 4.3*

5
Discussion

This chapter is divided into two parts. The first part is a discussion of the method and the second discusses the results.

5.1 Method

One could argue that using the approach where a pretrained network is used as feature extractor would be a better idea than using fine-tuning. This is because it takes less time to train, since only the new classifier would need training. To use this method instead would require that the dataset is more similar to the one that VGG16 has been trained on [18], which is not the case in this work. We did a test and it results in a worse accuracy of the network, which can be seen in figure 5.1 and table 5.1.

The reason why the layers were changed in the VGG16 network was due to the total training time. The VGG16 model has two fully-connected layers with 4096 neurons in each layer, which was replaced by fully-connected layers with only 256 neurons. If we compare the amount of weights that needed to be learned it would be 4096 x 4096 weights for the original layers and only 256 x 256 weights for the added layers. While this decreases the training time it will decrease the accuracy of the network, but this was a needed trade-off since the training was done on a CPU. One might argue that a shallower network could have been used instead of the deep VGG16 network. However, VGG16 is one of the simple state-of-the-art networks that are widely utilized in lots of applications.

There was a point when the search of parameters for the different experiments needed to be stopped. One could continue this search to try do improve the accuracy and loss of the network. The same applies to the collection of data, which would also yield improvements for the network.

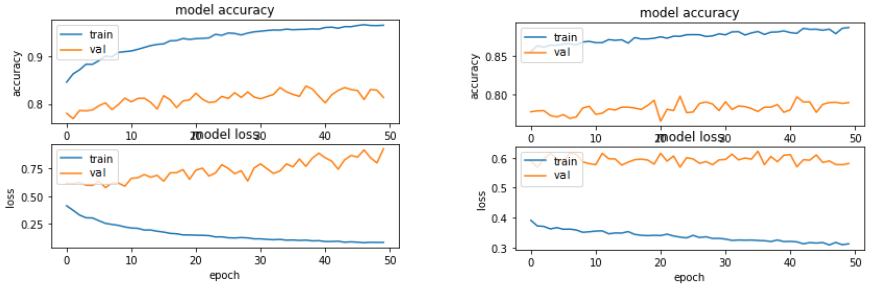


Figure 5.1: *Fine-tuning to the left and only the top trained to the right.*

Table 5.1: *Difference regarding the scores between the two methods with model7 used.*

Difference in accuracy and loss		
Method	**Accuracy(%)**	**Loss**
Fine-tuning	82.44	0.8602
Only top trained	78.39	0.5975

5.2 Results

The discussion about the results will be divided into three parts, the first part will discuss the dataset, the second part will discuss the network and lastly the robotic car.

5.2.1 Dataset

A short discussion about the quality of the dataset can be made. The datasets used was collected by a person driving around in an indoor environment. To be able to get consistent data it requires that every time the driver approaches an obstacle, the driver need to turn at the same distance from the obstacle. This is something that is hard to do, because sometimes the video stream freezes due to network deficiency. Even if it is just for less than one second, it becomes hard to turn at the right time.

Another aspect is data replication on the training set. Since every consecutive image that was received from the robotic car were labeled, there are a fair amount of images that could look the same. Another way to do this could be to label for example every 10th image instead, and use more images in total. A consequence of doing it this way is that it would take more time to collect the data.

When dealing with deep neural networks, it is known that they benefit from being trained on a large dataset. This can clearly be seen here in this work too, by looking at graphs 4.2c and 4.2e. The only difference between these two models is the increased size of the dataset used. This yielded an increase in accuracy by 3.74% points, which is a quite big improvement.

5.2.2 Network

As can be seen in table 4.2, models trained on the bigger dataset achieved better results. What can be seen if we look at graphs 4.2a, 4.2b and 4.2c, is that adding another dropout before the classifier did increase the accuracy and decreased the loss of the network. The way this can be explained is that dropouts makes the network more generic and prevents overfitting in the network. The graphs 4.2a and 4.2b shows that the gap between the accuracy on the training set and the validation set is relatively big, which may indicate that the network is overfitting. The dropout layer was added in an attempt to fix that. This is the reason why it was used throughout the rest of the experiments.

What can also be noted is that an increase in batch size also improves the network. Ideally all the samples would be used to calculate the gradient, but this is not efficient. What could also be mentioned is that when increasing the batch size it will affect the training time, a bigger batch size results in longer training time. In theory a bigger batch size will be more accurate than a smaller batch size, which is what we see in graphs 4.2e, 4.2f and 4.2g. There is a small improvement for each increase. However, if the step size is too big, it can cause the validation accuracy to go down again. This is due to that the error function would get stuck

in a local minimum and not get out of it again. Or, it could be due to an overshoot that will cause it to pass the minima.

When using fine-tuning a small learning rate should be used, which is the motivation for the chosen learning rate used in the experiments. The reason is that we expect that the pretrained network weights to be relatively good, and we do not want to manipulate them too much [18]. There were attempts to use a larger learning rate but this gave worse results by the network. Neither the loss nor accuracy were improved.

The result of using SVM instead of softmax can be seen in graph 4.2h and table 4.2. It did not yield a better result than using softmax. The plotted graph on the accuracy does indicate that the network is not learning, even if the loss is decreasing. It should be mentioned that no real search of parameter tuning was done for the SVM. The reason for using softmax for most of the experiments was also due to its popularity in the literature.

What can be noted in all of the graphs is that the loss value does slightly increase instead of dropping. What could be done to try to improve this is to use an adaptive learning rate. There exists some different ways to do this. One example would be to drop the learning rate after some epochs. Another would be to decrease the learning rate when the validation loss stops decreasing.

5.2.3 Robotic Car

In the implementation of the Python program, not every image received by the client software was taken care of. The reason was that the Python program received images faster than what it could predict an image. It was sometimes noted when testing with the robotic car, and did thereby have a small impact on the results. What could happen was that the robotic car did start to turn too late when approaching an obstacle, and did therefore hit the obstacle. If the first image that should be classified as turn right or left are missed, it will drive forward until the next image is predicted. Which will result in a late turn.

Each obstacle was moved around during testing to see how well the network was generalizing. The network had problems with predicting that it should turn and the robotic car went straight ahead instead. If an obstacle was standing on the same place as it did when the data was collected, the network was quite good at predicting the right movement instruction. This can especially be seen in table 4.3 when looking at the results for the couch. What might also have contributed to the success rate of the couch, is that the couch is quite big and covers almost the whole picture. In contrast to some other obstacles where the surroundings can affect the results more. For example, the bar stool is quite narrow and the surroundings have greater impact for this obstacle. It can be noted that the network was better at managing some obstacles than others. This can be explained by that the dataset did contain more samples of some obstacles.

The total success rate of avoiding collisions were 72.53%. This may be seen as a low result for autonomous collision avoidance. Even though the success rate is low, we state that deep learning shows good potential for solving the problem. As can be seen in the results, the network did perform better with the larger dataset. Therefore, the network will perform better if more data and more variance in data should be collected. This is based on the results that were collected, and that the robotic car did manage to avoid some obstacles well. There could also be more optimization done and larger search for hyperparameters, which would also improve the accuracy of the network.

6
Conclusion

The purpose of this thesis was to examine if deep learning is a good method for solving the problem of autonomous collision avoidance.

The first part of the work was to conclude what type of network to use. After doing the literature study, it became clear that an convolutional neural network was the obvious choice.

The second part of this work shows that using deep learning for solving the problem is possible. It is however not completely successful at generalizing, it has problems when the obstacles are moved around. However, deep learning shows good potential to solving the problem with greater success than what the collected results shows in this work. Thus, the network should be evaluated on a larger dataset.

6.1 Future Work

There are a couple of things that could be done to improve the network's performance. A larger dataset should be collected for the network to be trained on. As shown in this work, the network did improve quite much by just adding more images to the dataset. It should be an easy task that will improve the performance. A larger search of parameters could be done, as it can also can improve the performance of the network. The methods about adaptive learning rate discussed in section 5.2.2, are things that are worth trying out too.

Another thing to look into would be preprocessing of the images in the dataset. No preprocessing of the images was done in this work, and the entire image was used for training and prediction. What could be done is to remove the upper part

of the image, since there is essentially no relevant information in there.

There are several other networks that could be used for fine-tuning. It would be a good idea to test these networks as well and compare them to VGG16. During this work, Keras included a new network called MobileNet [12]. This network is made for mobile and embedded vision applications. There was no time to test this network, but it would be really interesting to see how it performs.

One could also experiment with different optimization function, there exists a lot more than just stochastic gradient descent. For example Adam, RMSprop and Adagrad to name a few. This is also the case for loss functions.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

[2] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[3] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[4] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006. ISBN 9780387310732.

[5] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[6] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[8] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd*

Edition). Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 013168728X.

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

[10] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48, 2016.

[11] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[12] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[13] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition, Accessed: 2017-08-20. URL http://cs231n.github.io/linear-classify/.

[14] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition, Accessed: 2017-08-20. URL http://cs231n.github.io/convolutional-networks/.

[15] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition, Accessed: 2017-08-20. URL http://cs231n.github.io/neural-networks-1/.

[16] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition, Accessed: 2017-08-20. URL http://cs231n.github.io/optimization-1/.

[17] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition, Accessed: 2017-08-20. URL http://cs231n.github.io/optimization-2/.

[18] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition, Accessed: 2017-08-20. URL http://cs231n.github.io/transfer-learning/.

[19] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition, Accessed: 2017-09-25. URL http://cs231n.github.io/linear-classify/#svm.

[20] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. Supervised machine learning: A review of classification techniques, 2007.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[22] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[23] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[24] D. Michie, D. J. Spiegelhalter, and C.C. Taylor. Machine learning, neural and statistical classification, 1994.

[25] Tom M. (Tom Michael) Mitchell. *Machine Learning*. McGraw-Hill, 1997. ISBN 0070428077.

[26] Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp, and Yann L Cun. Off-road obstacle avoidance through end-to-end learning. In *Advances in neural information processing systems*, pages 739–746, 2006.

[27] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[28] robidouille. raspicam cv library, Accessed: 2017-08-28. URL `https://github.com/robidouille/robidouille/tree/master/raspicam_cv`.

[29] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[30] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[31] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

[32] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[33] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[34] Oliver Strömgren. Documentation and source code for the robotic car, Accessed: 2017-08-28. URL `https://github.com/sansloes/master-thesis`.

[35] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In

The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2016.

[36] OpenCV team. Open source computer vision library, Accessed: 2017-08-28. URL `http://opencv.org/`.

[37] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.