Dmitrii Krasheninnikov

AUTONOMOUS CONTROL OF A RC CAR WITH A CONVOLUTIONAL NEURAL NETWORK

Bachelor's Thesis
Information Technology

2017

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland
University of Applied Sciences

Author (authors)	Degree	Time
Dmitrii Krasheninnikov	Bachelor of Engineering	May 2017

Title

Autonomous Control of a RC Car with a Convolutional Neural Network

43 pages

Commissioned by

The Curious AI Company

Supervisor

Reijo Vuohelainen

Abstract

Autonomous vehicles promise large benefits for humanity, such as a significant reduction of injuries and deaths in traffic accidents, and more efficient utilization of transportation leading to reduced air pollution and vastly reduced costs. However, at the present moment the technology is still in development.

The objective of the thesis was to build a simple and reliable testbed for the evaluation of algorithms for autonomous vehicles and to implement a baseline car control algorithm. For this purpose a system that allows a remote controlled car autonomously follow a track on the floor was developed. This work used the Parrot Jumping Sumo car with a built-in camera as the experimental vehicle. A control system that allows to receive and record the images from the car and send back the control commands was implemented. The baseline car control algorithm chosen in this work was a convolutional neural network (CNN) predicting control commands from the images received in real time from the car's camera.

CNNs are machine learning models achieving state of the art results in a variety of computer vision tasks, and have previously been applied to autonomous driving. Several simple machine learning models were introduced in this thesis, followed by construction of a CNN from these models. Afterwards, the algorithms used to train CNNs were reviewed. The CNN used in this work was trained on one hour of recorded driving data and was able to successfully control the car for over a minute without requiring an intervention by a human driver.

Keywords

autonomous driving, robot operating system, machine learning, deep learning, supervised learning, regression, neural network, convolutional neural network, backpropagation

CONTENTS

SYMBOLS AND ABBREVIATIONS

Symbols

a	A scalar
a	A vector or a random variable
A	A matrix
$\mathbb{A}$	A set
$p(\cdot)$	A probability distribution
θ	The model parameters
$\sigma(\cdot)$	The sigmoid function
$1_{(\cdot)}$	The indicator function

Indexing

$x^{(i)}$	The i-th example from a dataset
$h^{(i)}$	The i-th hidden layer of a neural network
$W^{(i)}$	The weight matrix of the i-th hidden layer of a neural network
a_i	The i-th element of a vector a
$A_{i,j}$	The i-th element of the j-th column of a matrix A

Abbreviations

RC	Remote-controlled
ML	Machine learning
NN	Neural network
FC	Fully-connected
CNN	Convolutional neural network
ROS	Robot operating system
CPS	Command prediction system

1 INTRODUCTION

In the modern world, machine learning (ML) plays a role more significant than ever in many seemingly very different areas such as genetics, pharmacological research, image classification and segmentation, video captioning, speech recognition, natural language processing, robotics and stock market predictions. ML powers the Netflix movie recommendation system and the Google search engine; most of the weather forecasting labs use ML algorithms to make predictions.

This thesis focuses on the application of ML to autonomous driving, a technology expected to redefine the automotive world. The Curious AI Company develops powerful ML algorithms that might operate the self-driving cars of the future. A simple and reliable testbed is needed for evaluating the algorithms in the physical world. For this purpose a system that allows a remote controlled car autonomously follow a color-marked track on the floor using the imagery from the car's built-in camera is developed. As a baseline algorithm a convolutional neural network is used to predict the control commands from the video frames. The theoretical part of this thesis addresses the following questions:

- What is machine learning?
- Which kinds of machine learning algorithms exist?
- What are convolutional neural networks?
- How are neural networks trained?

This thesis is organized as follows. Chapter 2 provides an overview of the basic concepts and assumptions in ML. Chapter 3 delves into parametric models and introduces convolutional neural networks. In Chapter 4 algorithms and techniques used to train parametric models are explored. Chapter 5 briefly reviews applications of ML in autonomous driving, reports the detailed implementation of the car control system and analyzes the results. Chapter 6 is dedicated to conclusions.

2 MACHINE LEARNING

This chapter provides an overview of the basic concepts in ML. First, the definition of ML is introduced, followed by a description of the main paradigms of learning. The chapter closes with a discussion of assumptions about the data generating process that are often embedded into ML models.

2.1 Definition

As opposed to classical computer programs, in which the task is formalized as a predefined sequence of precise instructions, ML algorithms base their decisions on the information extracted from the data. This is especially important in cases where it is not feasible to specify the task explicitly, or the static specifications are not robust enough.

One of the classic definitions of machine learning is provided by Mitchell (1997):

DEFINITION 1. "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**".

Consider an image classification task: a dataset of images with either a cat or a dog on each image is given. For some of the images the labels are given, that is, it is known if there is a cat or a dog in the image, and for another set of images the labels are unknown. The task **T** is to infer the labels for images without them; experience **E** consists of the set of images paired with their known labels, also commonly referred to as *training set*. The set of images with unknown labels is usually called *test set*. The performance measure **P** measures how well the knowledge extracted from the training set is generalized to make predictions about the labels of the images in the test set.

It is easy to see that solving the task with a classical computer program is not practical and likely infeasible. Suppose the images are grayscale with the resolution of 64×64 and have the standard 8 bit color depth. The number of all such images is extremely large: $2^{8 \times 64 \times 64} = 2^{32768} \approx 10^{9860}$, and all images of cats

and dogs are a tiny subset of them. Still, it would take an enormous amount of $\texttt{if} - \texttt{then}$ statements to disentangle the messy pixel input into binary output. On the other hand, a machine learning algorithm will automatically learn the most relevant and informative features of the images, collapsing the input space into a low-dimensional manifold where it is easy to perform classification.

2.2 Paradigms of Learning

Based on the type of signal received with the input data, ML algorithms can be roughly divided into several classes: supervised, unsupervised, semi-supervised and reinforcement learning. These paradigms of learning are described below.

2.2.1 Supervised Learning

Supervised learning is the most mainstream form of ML, and by far the most successful in practical applications (LeCun et al. 2015). In the supervised setting the algorithm learns a mapping between given input-output pairs, as in the cats and dogs image classification example above.

More formally, the dataset is usually provided in a form of pairs $(x^{(i)}, y^{(i)})$ where $x^{(i)} \in \mathbb{R}^n$ is an input item and $y^{(i)}$ is the item's corresponding correct output. When all samples $x^{(i)}$ have the same dimensions, the input is commonly expressed as a *design matrix* $X = [x^{(0)}, x^{(1)}, ..., x^{(k)}]^T$. Similarly, outputs are often represented as a matrix $Y = [y^{(0)}, y^{(1)}, ..., y^{(k)}]^T$ or a vector y if outputs $y^{(i)}$ are one-dimensional. Each of the elements of an item $x^{(i)}$ is referred to as a *feature*. Additionally, sometimes *feature* can refer to a whole column of the design matrix X.

In supervised learning the goal is to approximate function f such that $f(X) = Y$. Learning f here corresponds to learning the conditional probability distribution $p(y|x)$.

In case the domain of $y^{(i)}$ is a discrete set ($y^{(i)} \in \mathbb{Z}^m$), such as "spam" and "not spam" categories of emails in a spam filter, the modeling task is referred to as *classification*. In the case when $y^{(i)}$ takes continuous values ($y^{(i)} \in \mathbb{R}^m$), such as in a task predicting the price of the house given its area, the task is called *regression*. This thesis deals with a supervised regression problem.

Even though at the first glance supervised ML algorithms can seem very different, and indeed rely on different paradigms of math, computer science and physics, the *function approximators* narrative unifies them. The job of a ML algorithm, be it logistic regression, a decision tree or a neural network, is to construct an accurate mapping from inputs to their corresponding outputs (Ayodele 2010). For the example of image classification mentioned above, an arbitrary ML algorithm is learning how to approximate the function from the domain of provided images to the range of labels.

2.2.2 Unsupervised Learning

A task of inferring some kind of structure in the data without labels is usually referred to as unsupervised learning. The dataset is typically provided in a form of $x^{(i)}$, and the goal is to model the probability distribution of data $p(x)$.

The kinds of tasks where unsupervised learning is used are:

- Cluster analysis;
- Dimensionality reduction for exploratory data mining;
- Generative modelling: the goal is to mimic the data generating process;
- Compression tasks: it is desired to keep as much structure of the data distribution as possible while using a limited amount of memory.

In addition to the above, unsupervised learning can be used as a feature extraction part of a procedure for some other form of learning. For example, we can learn a good representation of animals from a large number of unlabelled pictures of animals, and hereafter use this representation with a small number of labeled pictures of cats and dogs to train an accurate cat vs dog classifier. This kind of approach is called **semi-supervised learning**. Leading machine learning researchers expect unsupervised learning to become significantly more important in the longer term (LeCun et al. 2015).

2.2.3 Reinforcement Learning

Reinforcement learning is concerned with sequential decision making and relies on a much weaker training signal than supervised learning. The system, referred to as the agent, interacts with the environment by making actions based on the observed state of the environment and receiving rewards; the goal is to maximize the overall accumulated and time-discounted reward. As there is no clear feedback about which actions lead to the reward or punishment, the agent has to figure out the correspondence by itself. Examples of tasks that fit the reinforcement learning framework are playing games such as Atari (Mnih et al. 2015) and Go (Silver et al. 2016), control tasks in robotics and even optimization of power usage effectiveness in a datacenter (Evans & Gao 2016).

2.3 Assumptions

A large part of a success of a ML algorithm is a correct set of beliefs about the world incorporated into it. Domingos (2012) phrases this as "every learner must embody some knowledge or assumptions beyond the data it's given in order to generalize beyond it".

Formally this is known as the **No Free Lunch Theorem**, introduced by Wolpert & Macready (1997):

THEOREM 1. *Given a finite set* $\mathbb{V}$ *and a finite set* $\mathbb{S}$ *of real numbers, assume that* $f : \mathbb{V} \to \mathbb{S}$ *is chosen at random according to uniform distribution on the set* $\mathbb{V}^{\mathbb{S}}$ *of all possible functions from* $\mathbb{V}$ *to* $\mathbb{S}$*. For the problem of optimizing* f *over the set* $\mathbb{V}$*, then no algorithm performs better than blind search.*

In other words, averaged across all possible function approximation tasks no algorithm generalizes to the previously unseen data points better than a random algorithm. This suggests that in practice good performance can only be achieved by incorporating the knowledge of the distribution's structure into the ML model as a set of assumptions. From a Bayesian viewpoint these assumptions can be seen as priors. The assumptions used in the majority of machine learning models are described below.

The smoothness assumption for supervised regression states that if two inputs points $x^{(0)}$, $x^{(1)}$ are close, so should be their corresponding outputs $y^{(0)}$, $y^{(1)}$ (Zhu & Goldberg 2009). For supervised classification this translates into similar examples having similar classes. For semi-supervised and unsupervised learning algorithms the smoothness assumption usually holds only for high-density regions of data.

The limited dependencies assumption states that most features do not affect each other to a large extent. For example, the Naive Bayes model assumes that the elements of $x^{(i)}$ are conditionally independent from each other given the output $y^{(i)}$. In graphical models the limited dependencies assumption corresponds to the graph not being densely connected (Sutherland 2015).

The limited complexity assumption states that the true data generating process has a significant amount of structure and can be represented well with a fixed number of parameters. This assumption is somewhat similar to incorporating Occam's razor into the model. Regularization, which is described in Section 4.4, usually also limits model complexity, adding a preference for simpler models.

There is no single machine learning algorithm that works best across all the tasks. Different algorithms add more assumptions about the data generating process to the ones mentioned above, which results in superior performance in tasks where the added assumption is correct. Additionally, when selecting a ML algorithm one always deals with tradeoffs between speed, complexity, accuracy, and interpretability across the algorithms.

3 PARAMETRIC MODELS

This chapter presents parametric models, the family of ML models to which convolutional neural networks belong. First, linear regression and logistic regression, two simple parametric models, are introduced. Next, a fully-connected neural network is constructed from these two models. Finally, the fully-connected neural network is extended to a convolutional neural network.

3.1 Overview

One of the most important dichotomies in ML is between parametric and nonparametric models. The main distinction between these families of models lies in the different ways of approximating the data generating process: in parametric models the number of parameters specifying a model is fixed whereas in nonparametric models the number of parameters grows with the amount of training data.

Parametric models, which are the focus of this thesis, are usually faster to use and easier to interpret, but they make stronger and sometimes unnecessary assumptions about the nature of the data distributions. Nonparametric models are more flexible, but often computationally intractable for large datasets. (Murphy 2012)

Formally, parametric models assume a finite set of parameters θ. Ghahramani (2015) states that given the parameters, predictions $\hat{y}$, are independent of the observed data, x:

$$p(\hat{y}|\theta,x) = p(\hat{y}|\theta) \tag{1}$$

Therefore θ capture everything there is to know about the data.

In order to evaluate how well a given model describes the observed data and to estimate the model's generalization to unobserved data some kind of performance measure is needed. This performance measure is referred to as a *cost* or *loss function* $J(x,y,\theta)$; in supervised learning the two most commonly used cost functions are mean squared error and cross-entropy loss. Given a family of the model and the cost function, the task of selecting model parameters θ is reduced to finding θ that minimize the cost J:

$$\theta = argmin_{\theta} J(x,y,\theta) \tag{2}$$

The process of finding parameters satisfying the above is referred to as training the model.

The next sections describe several parametric models and their workings starting from the simplest model for regression, linear regression, followed by neural

networks, their different architectures and the various tricks for training them.

3.2 Linear Regression

As mentioned in the section on paradigms of learning, in regression the goal is to approximate a function $y = f(x, \theta)$ where (x, y) is a pair of random variables $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$. A simple example of a problem where one might want to use linear regression is predicting a child's height y based on parents' heights $[x^{(0)}, x^{(1)}]^T$. As the algorithm's name implies, linear regression assumes that there is an approximately linear relationship between x and y parametrized by a weight matrix and a bias term $\theta = [W, b]$. The estimate of y is denoted as $\hat{y}$:

$$\hat{y} = f(x, \theta) = Wx^T + b \tag{3}$$

To simplify the notation Ng (2013) introduces the convention of letting $x^{(i)} \rightarrow [x^{(i)}, 1]$ and $W \rightarrow [W, b]$, so that:

$$\hat{y} = f(x, \theta) = Wx^T \tag{4}$$

The mean squared error cost is typically used to evaluate the performance of the model:

$$J_{MSE}(y, \hat{y}) = \frac{1}{N} \Sigma_{i=1}^{N} (\hat{y}_i - y_i)^2, \tag{5}$$

where N is the number of labeled items in the dataset.

Intuitively, one can see that the cost will be 0 when $\hat{y} = y$. In practice, this almost never happens due to variance and noise in the data, even if the true underlying relationship is linear. The best one can do is to find such parameters θ that the cost is minimized.

Typically to minimize the cost an iterative numerical algorithm such as gradient descent (described in Section 4.1) is used. However, for the linear least squares problem there exists a closed form solution (Goodfellow et al. 2016):

$$W = argmin_W ||y - WX^T||_2 = (X^T X)^{-1} X^T y \tag{6}$$

3.2.1 Polynomial Regression

It is easy to make linear regression accurately approximate more complex relationships between x and y by adding polynomial features thus making the model more expressive. For example, a one-dimensional input x may be extended to include all polynomials of x up to degree D:

$$\hat{y} = f(x^D, \theta^D) = W_D x^D + W_{D-1} x^{D-1} + ... + W_1 x + b \tag{7}$$

Polynomial regression is still linear in features, and the additional expressive power comes from the newly added polynomial features being nonlinear.

3.2.2 Overfitting and Underfitting

In practice, a less expressive model is likely to be unable to accurately model a complex data generating process, while a more expressive model has higher chances of capturing the noise present in the data. These two failure modes are referred to as *underfitting* and *overfitting*. Figure 1 illustrates overfitting and underfitting with an example of approximating a noisy cosine function using polynomial regression models of varying degrees.

When observing subpar performance on the test data, it is important to identify whether underfitting or overfitting is taking place in order to optimally choose a course of action that will address the problem. One of the best ways to diagnose underfitting and overfitting is examining the loss L for the training data and the test data.

- Overfitting is usually diagnosed if the loss computed on the test dataset is significantly higher than the loss computed on the training dataset.
- Underfitting is usually suspected when both the training and the test losses are high.

Overfitting is usually addressed by collecting more data and using regularization techniques further discussed in Section 4.4. Underfitting can often be addressed by improving the machine learning model – incorporating the correct assumptions about the data generating process and increasing the model complexity.

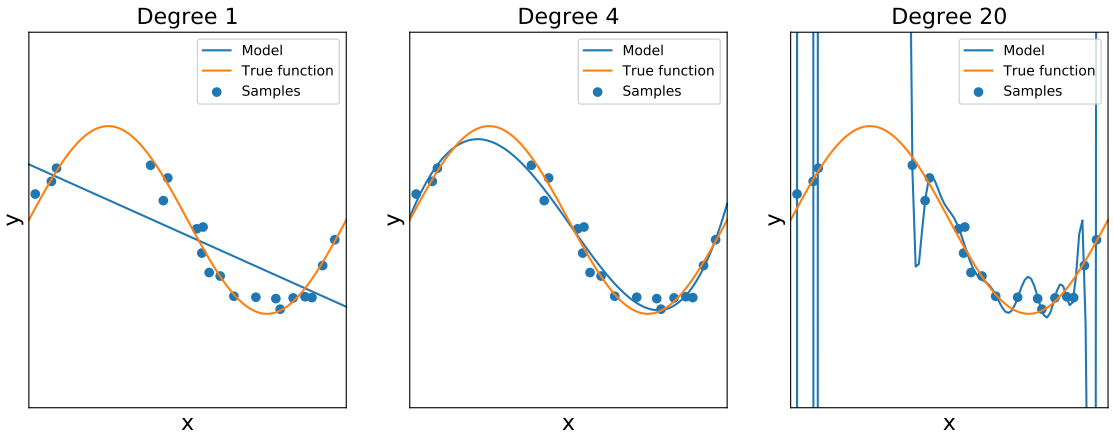

Figure 1. Underfitting (left), good performance (center) and overfitting (right)

3.3 Logistic Regression

Logistic regression generalizes the linear regression model to handle classification, a range of supervised learning tasks where the output is discrete. The *logistic* function, also referred to as *sigmoid* function, "squashes" the arbitrarily real-valued input into an output with values in the range $(0, 1)$.

$$\sigma(x) = \frac{1}{1 - e^{-x}} \tag{8}$$

In the case of binary output $y \in \{0, 1\}$, logistic regression models the estimate of the probability that $y = 1$:

$$\hat{p}(y = 1) = \sigma(Wx^T) \tag{9}$$

The binary cross-entropy cost is typically used to evaluate performance of logistic regression. In the equation below $1_{(\cdot)}$ is an indicator function, such that $1_{\text{true statement}} = 1$ and $1_{\text{false statement}} = 0$.

$$J_{CE}(y, \hat{y}) = -\sum_{i=1}^{N} [1_{y^{(i)}=1} \log(\hat{p}(y^{(i)} = 1)) + 1_{y^{(i)}=0} \log(1 - \hat{p}(y^{(i)} = 1))] \tag{10}$$

$$= -\sum_{i=1}^{N} [y^{(i)} \log(\hat{p}(y^{(i)} = 1)) + (1 - y^{(i)}) \log(1 - \hat{p}(y^{(i)} = 1))] \tag{11}$$

Unlike for the linear least squares problem, there is no closed-form solution for finding the parameters that minimize the cross-entropy cost. Instead, iterative numerical algorithms such as gradient descent are typically used.

Logistic regression can be extended to an output with multiple classes. Typically the k classes are represented in the "one-hot" encoding – an input that belongs

to the j-th class ($j \in 1, ..., k$) is labeled with a k-dimensional vector where the j-th element equals 1 and all the other elements are 0.

Analogously to the sigmoid function in the logistic regression, the *softmax* function is used to make the outputs of the multiclass logistic regression interpretable as class probabilities. The *softmax* function is a generalization of the sigmoid function that receives a k-dimensional vector as input and outputs a k-dimensional vector whose elements are positive and sum up to 1. The j-th element of the output vector ($j \in 1, ..., k$) is given by:

$$softmax(z)_j = \frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_i}}, \tag{12}$$

where the subscript j indicates the j-th element of the output vector.

The estimate of the probability that the output belongs to given class j is then:

$$\hat{p}(y = j) = softmax(Wx^T)_j \tag{13}$$

The binary cross-entropy cost can also be generalized to multiple labels. Assuming y is represented using the one-hot encoding, the cross-entropy cost is:

$$J_{CE} = \sum_{i=1}^{N} y_i \log(softmax(Wx^T))^T \tag{14}$$

3.4 Neural Networks

In many interesting cases, such as computer vision and natural language processing, linear models often cannot satisfyingly approximate the function $y = f(x)$. To extend linear models to represent a richer family of nonlinear functions of x, one can apply the linear model not to x itself but to a nonlinearly transformed input $g(x)$ (Goodfellow et al. 2016). Neural networks (NNs) are parametric function approximators $f(x)$ composed of several simpler functions: $f_{NN}(x) = (f_{(n)}(f_{(n-1)}(...f_{(0)}(x)))$. A single layer NN can be defined as

$$\hat{y} = f(x, W^{(2)}, b^{(2)}, W^{(1)}, b^{(1)}) = W^{(2)} g(W^{(1)} x + b^{(1)}) + b^{(2)}, \tag{15}$$

where $g(\cdot)$, usually an element-wise function, is referred to as *activation function* and the vector $h^{(1)} = g(W^{(1)} x + b^{(1)})$ is called *hidden layer*. Elements of the hidden layer are usually referred to as *hidden units*.

The **Universal Approximation Theorem** by Hornik et al. (1989) states that a single layer neural network with a finite number of neurons in a hidden layer and loose assumptions on the activation function $g(\cdot)$ can approximate continuous functions on compact subsets of $\mathbb{R}^n$ to any desired degree of accuracy.

To simplify the notation for the neural network layers one can use the same trick Ng (2013) used for linear regression: $h^{(l)} \rightarrow [h^{(l)}, 1]$ and $W^{(l)} \rightarrow [W^{(l)}, b^{(l)}]$. Using this notation, Equation 15 can be rewritten as:

$$\hat{y} = W^{(2)} g(W^{(1)} x) \tag{16}$$

A single layer neural network can be generalized to an arbitrary number of hidden layers $h^{(l)}, l = 1 : L$ using the following recursive relation:

$$h^{(0)} = x, \tag{17}$$

$$h^{(l)} = g(W^{(l)} h^{(l-1)}) \tag{18}$$

The total number of layers in the network is called *depth* of the model. From this terminology the name *deep learning* arises. Algorithm 1 demonstrates the computation of an output and the loss of a neural network of depth l.

Algorithm 1. Neural Network Forward Propagation

Require: l, the network depth
Require: $W^{(i)}, i \in \{1, ..., l\}$, the weight matrices of the model
Require: $b^{(i)}, i \in \{1, ..., l\}$, the bias parameters of the model
Require: $g^{(i)}(\cdot), i \in \{1, ..., l\}$, the list of activation functions
Require: $J(\cdot)$, the cost function
Require: x, the input to process
Require: y, the target output

$h^{(0)} \leftarrow x$
for $k \leftarrow 1, ..., l$ **do**
  $a^{(k)} \leftarrow b^{(k)} + W^{(k)} h^{(k-1)}$
  $h^{(k)} \leftarrow g^{(k)}(a^{(k)})$
end for
$\hat{y} \leftarrow h^{(l)}$
$L \leftarrow J(\hat{y}, y)$ ▷ In practice, a regularization term is often added to the loss L. Regularization is addressed in detail in section 4.4.
return L

It is common to represent neural networks as directed acyclic graphs. For example, Figure 2 shows a neural network with input $x \in \mathbb{R}^4$, two hidden layers

$h^{(1)} \in \mathbb{R}^5$ and $h^{(2)} \in \mathbb{R}^3$, and output $y \in \mathbb{R}$. Nodes of the graph are elements of the network's layers, and each of the arrows connecting the nodes represents an element of a weight matrix. A weight matrix $W^{(i)}$ "connects" the layers $h^{(i)}$ and $h^{(i+1)}$. For example, the arrow connecting the first element of the input layer to the first element of the layer $h^{(1)}$ corresponds to the element $W^{(1)}_{1,1}$ of the weight matrix $W^{(1)}$. If the layer $h^{(i)}$ has n hidden units and the layer $h^{(i+1)}$ has m hidden units, the shape of the weight matrix $W^{(i)}$ is $n \times m$. As each of the elements of a layer defined by Equation 18 is "connected" to each of the elements of the subsequent layer (by an element of W), such layers are often referred to as *fully-connected* layers. Neural networks that consist only of fully-connected layers are called fully-connected neural networks.

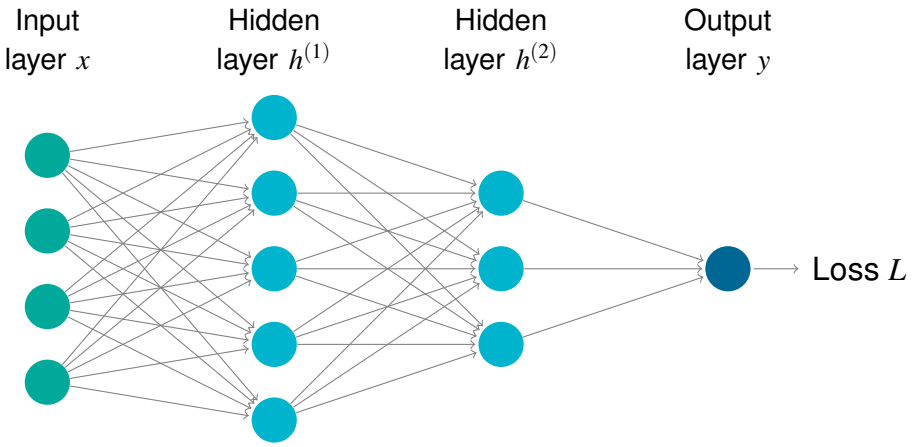

Figure 2. A fully-connected neural network with two hidden layers

A fully-connected layer $g(Wx+b)$ performs the following transformations of the input x:

1. A linear transformation by the weight matrix W.
2. A translation by the vector b.
3. Application of $g(\cdot)$, usually a pointwise nonlinear function.

Thus a neural network of an arbitrary depth can be viewed as a sequence of linear and nonlinear transformations of the input x. In classification tasks these transformations simplify the job of an output layer by making different classes linearly separable. For regression tasks the relationship between the transformed input $h^{(l-1)}$ and y can be modeled much easier than the relationship between x and y.

The most common choice of the output activation function for networks performing regression is the identity function $g(x) = x$. The softmax function is typically used in the output layer of a NN performing classification. This way the output layer can be viewed as performing linear regression or multiclass logistic regression with the last hidden layer $h^{(l-1)}$ as input and the output y.

The de-facto standard activation function $g(\cdot)$ for a neural network's hidden layers is *rectified linear unit* (RELU).

$$\text{RELU}(x) = \max(x, 0) \tag{19}$$

Before RELU was popularized by Glorot et al. (2011), the most common choices for the activation function were sigmoid $\sigma(x)$ and hyperbolic tangent $2\sigma(2x) - 1$ functions. RELU is superior to both of them in several ways, most notably in the efficiency of computation as only comparison, addition and multiplication operations are used. Additionally, RELUs are scale invariant as $max(0, \alpha x) = \alpha max(0, x)$.

The cost functions used to train and evaluate the performance of NNs are the same as those used for linear and logistic regression: common choices are the mean squared error (Equation 5) for regression and the cross-entropy cost (Equation 14) for classification. Similarly to logistic regression, there is no closed-form solution for minimizing the cost, and numerical optimization algorithms are used instead.

In the name "neural networks" the word *neural* is due to NNs' functional similarities with biological neural networks. Each of the elements of a hidden layer resembles a neuron, in a sense that it receives inputs from many other units, sums them up and uses the sum to produce its own activation, an output. Because of this the elements of the hidden layer are sometimes referred to as *neurons*. Layers of neurons act in parallel, processing information and sending their activations to the next layer of neurons. Thus, a neural network is composed.

3.5 Convolutional Neural Networks

For many machine learning tasks in computer vision, volumetric and time series data analysis one wants to incorporate more structure of the task into our model in order to make it more accurate and easy to train. This can be viewed as adding more assumptions about the data generating process to the assumptions mentioned in Section 2.3. Convolutional neural networks (CNNs) introduced by LeCun et al. (1998) incorporate the translation invariance assumption, which is useful for the data with established, grid-like topology. For computer vision tasks translation invariance means that an object would be recognized as that object independent of its location in the picture. Figure 3 shows an example of two images invariant under translation. When given as input to a CNN performing classification these images would produce the same output.

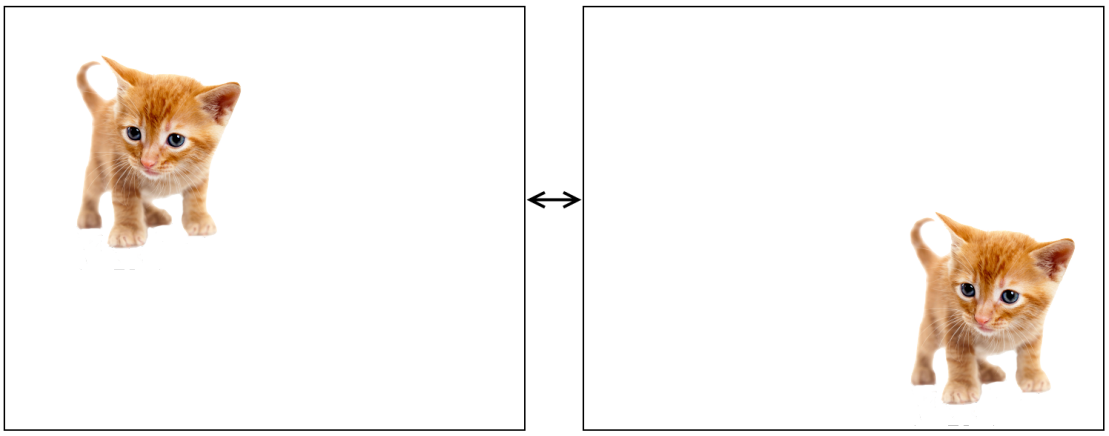

Figure 3. Two images invariant under translation

This thesis focuses on convolutional networks for computer vision, as predicting the car control commands from the images is essentially a computer vision task. Each input image $x^{(i)}$ is typically provided in a form of a 3-dimensional tensor $x^{(i)} \in \mathbb{R}^{n \times m \times c}$, where (n, m) are the image's width and height and c is the number of color channels in the image. Usually the number of color channels is either 1 for grayscale images or 3 for RGB images.

3.5.1 Convolutional Layer

For a neural network to be convolutional one or more of the network's hidden layers has to use the *convolution operation* instead of the traditional linear transformation by a weight matrix used in a fully-connected layer. This kind of layer is referred to as convolutional layer. In the convolutional layer, similarly to a

fully-connected layer, after the convolution operation the input is translated by a bias b and then transformed with an activation function $g(\cdot)$. Usually convolutional networks are composed by multiple convolutional layers followed by several fully-connected layers.

The *2D convolution* operation in images is in essence multiplying the color intensity of a small patch of the image by a small matrix, commonly referred to as *kernel* or *filter*. Given the convolutional kernel K with dimensions $k \times k$ and a $k \times k$ patch of the input image I, the 2D convolution operation is defined as:

$$O_{p,q} = I \times K = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} I_{p-i,q-j} K_{i,j}. \tag{20}$$

In practice machine learning libraries often implement 2D convolution with the kernel K flipped vertically and horizontally, as shown in Figure 4. Formally, this operation is referred to as *cross-correlation*. Computation of cross-correlation is shown in Figure 5.

$$\begin{bmatrix} K_{0,0} & K_{0,1} & K_{0,2} \\ K_{1,0} & K_{1,1} & K_{1,2} \\ K_{2,0} & K_{2,1} & K_{2,2} \end{bmatrix} \Rightarrow \begin{bmatrix} K_{2,2} & K_{2,1} & K_{2,0} \\ K_{1,2} & K_{1,1} & K_{1,0} \\ K_{0,2} & K_{0,1} & K_{0,0} \end{bmatrix}$$

Figure 4. 2D kernel flipped vertically and horizontally

Neurons in a convolutional layer perform convolutions of their input with trainable weights used as convolutional kernels. For example, one may have a convolutional kernel that detects salient features of the cat's face. The CNN would use this kernel to see whether there is a cat's face in different parts of the input image by convolving this kernel with different parts of the input. This process would produce a *feature map*, a matrix with entries corresponding to the similarity of the convolutional kernel to the patch of the original image in the matching location. The hidden layer of a CNN consists of multiple feature maps generated using different convolutional kernels.

Unlike in a traditional fully connected network, neurons in CNNs share parameters. This has an intuitive explanation: when determining whether there is a cat in the picture, one would not care if the cat is at the top or the bottom of the picture. In addition to incorporating the translation invariance assumption, weight sharing results in a reduced number of trainable parameters in the model, cutting down the training time and making the model more compact.

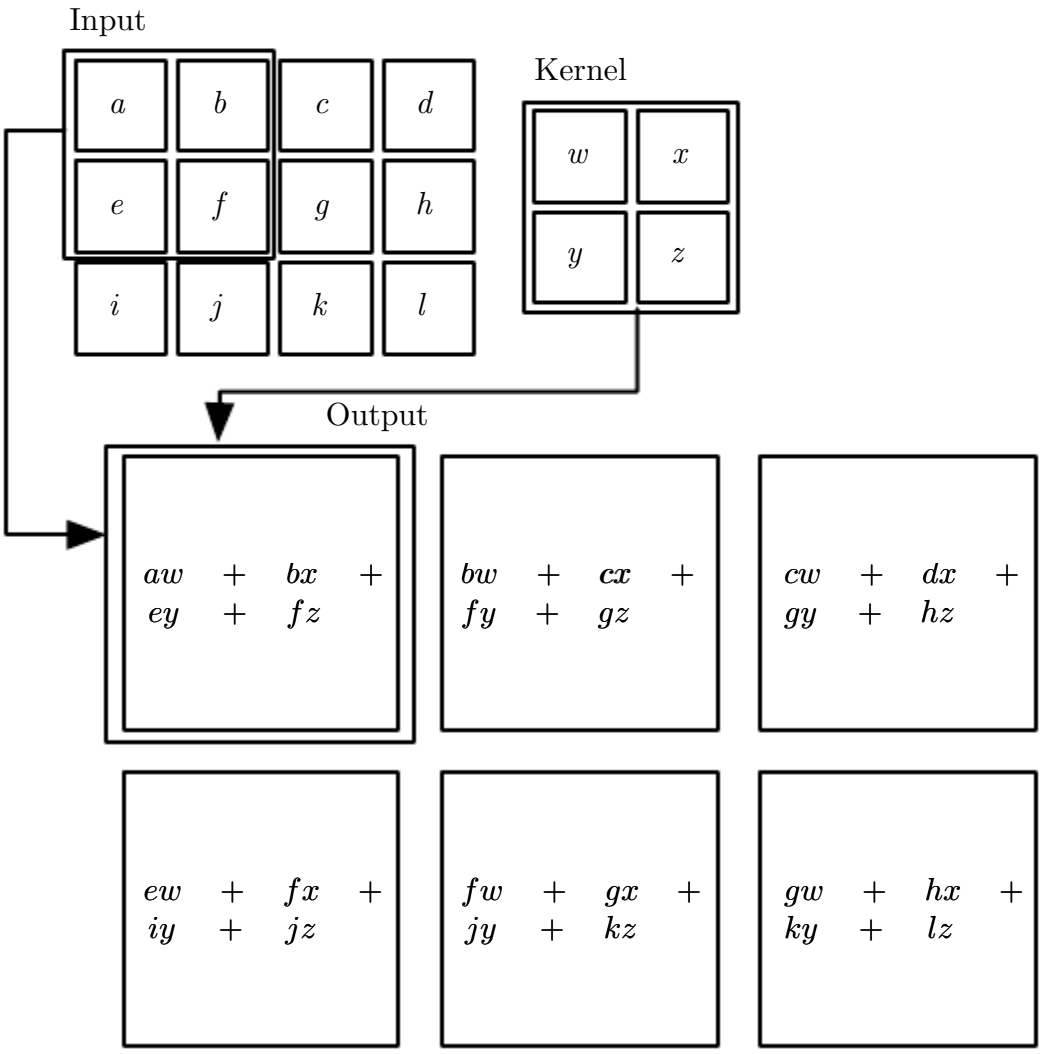

Figure 5. 2D cross-correlation operation (Goodfellow et al. 2016)

3.5.2 Pooling Layer

Pooling layers are commonly used after convolutional layers. The *pooling* operation outputs its nonlinearly downsampled input. The purpose of using a pooling layer is to reduce the number of parameters in the network, hence speeding up the training, preventing overfitting and forcing the network to learn useful representations.

A typical pooling function reduces a $n \times m$ region of the input feature map to a single value in the output feature map, where n and m are small integers such as 2 or 3. The most widely used pooling function is *max-pooling* that returns the maximal value of each $n \times m$ region of the input. Figure 6 shows an example of max-pooling with a 2×2 kernel. Sometimes functions other than max-pooling are used, such as average pooling and L2-norm pooling.

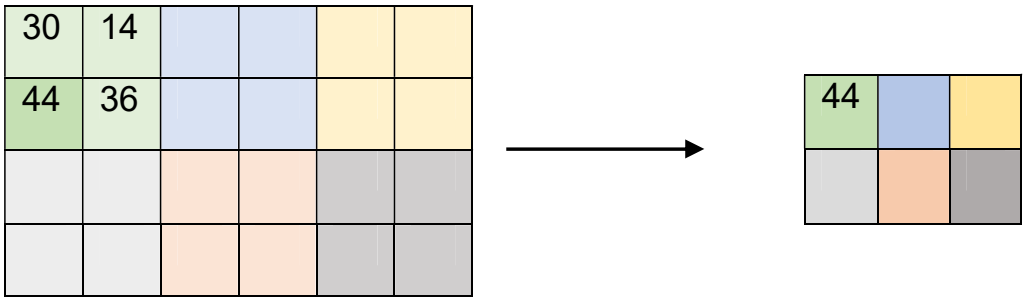

Figure 6. Example of max-pooling

4 TRAINING PARAMETRIC MODELS

This chapter presents the methodology used for training parametric models that include CNNs. First, the gradient descent algorithm is introduced, followed by the backpropagation algorithm that allows neural networks to be trained with gradient descent. Finally, L2-norm regularization and dropout, two regularization techniques often used when training CNNs, are discussed.

4.1 Gradient Descent

As stated in the introduction of the chapter on parametric models, given a parametric model and the cost function $J(x, y, \theta)$, the task of selecting model parameters θ is reduced to finding such parameters θ that minimize the cost J:

$$\theta = argmin_{\theta} J(x, y, \theta) \tag{21}$$

For the parametric models with nonlinearities such as logistic regression or neural networks there is no closed form solution to minimize J. Instead, various iterative algorithms are usually used. A single step of an iterative optimization algorithm, also referred to as update, can be viewed as

$$\theta_{t+1} = \theta_t + \eta_t D_t, \tag{22}$$

where D is the direction of the update and η is the step size (also referred to as the learning rate). The parameters are usually updated for a fixed number of iterations or until the criteria for convergence are met. An example of the con-

vergence criteria would be J getting close enough to zero or the improvement dropping below a predefined threshold for several consecutive updates.

Gradient descent (GD) algorithm is an iterative algorithm most commonly used for minimizing the objective function in machine learning tasks. Gradient descent uses partial derivatives of the cost J with respect to parameters θ to linearly approximate the cost function and determine the direction in which it decreases fastest, thus determining the direction of an update:

$$\theta_{t+1} = \theta_t - \eta_t \nabla J(\theta_t). \tag{23}$$

In practice, for large datasets that contain hundreds of thousands or more of training samples the time to compute a single weight update from the whole dataset becomes prohibitively long. A standard way to address this problem lies in estimating the gradient from a small subset of samples, called a *minibatch* (Goodfellow et al. 2016). The training procedure of an arbitrary parametric model using the minibatch GD is shown in Algorithm 2.

Algorithm 2. Minibatch Gradient Descent

Require: η_k, the learning rate
Require: θ, the initial parameters of the model
Require: J, the objective being minimized
Require: X, the training examples
Require: Y, the targets
Require: m, the minibatch size

while stopping criteria not met **do**
  $(x^{(1,\ldots,m)}, y^{(1,\ldots,m)}) \leftarrow \texttt{sampleMinibatch}(X, Y)$
  $\hat{\delta} \leftarrow \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} J(f(x^{(i)}, \theta), y^{(i)})$
  $\theta \leftarrow \theta - \eta \hat{\delta}$
end while

There are multiple techniques that help to improve GD-based training, such as:

- Using an adaptive learning rate η: shrinking η over time, for example by multiplying it with $\gamma \in (0,1)$ after every few iterations. This usually leads to convergence around better minima.

- Using momentum: adding a fraction of the gradient computed at the previous iteration to the weight update. This strategy smooths out the descent trajectory and often leads to faster convergence.

- Meta-learning: replacing a hand-crafted update rule, usually a combination of adaptive learning rate and momentum, by a *learned* update rule: $\theta_{t+1} = \theta_t + f_t(\nabla_\theta J(\theta_t), \phi)$. Here f_t is a learned function approximator such as a recurrent neural network parametrized by ϕ. (Andrychowicz et al. 2016)

Ruder (2016) provides an extensive overview of GD-based optimization algorithms and concludes that Adam is likely the best overall choice. Adam was introduced by Kingma & Ba (2014) and uses both the adaptive learning rate and momentum learning.

4.2 Backpropagation

The backpropagation algorithm is a way to compute the gradients of the nodes in composite functions such as neural networks, and is a standard technique for training the neural network parameters. The algorithm was reinvented multiple times across different fields, notably by Kelley (1960) and Dreyfus (1962) in the context of control theory and Linnainmaa (1970) in the context of automatic differentiation. The backpropagation algorithm for neural networks consists of two stages:

1. Forward propagation: given parameters theta, input x and correct output y, compute the loss $L = J(x, y, \theta)$ (Algorithm 1).

2. Backward propagation: compute the partial derivatives of the loss L with respect to parameters $b^{(i)}$ and $W^{(i)}$ starting from the output layer using the chain rule of calculus. As soon as the parameters' gradients are computed, update the parameters using the GD update rule (Algorithm 3).

The names "forward propagation" and "backward propagation" refer to the graph representation of neural networks. As shown in Figure 7, the "forward" direction corresponds to the left-to-right computation in the graph, and "backward" corresponds to the right-to-left computation.

Algorithm 3. Neural Network Backpropagation

Require: l, the network depth
Require: $W^{(i)}, i \in \{1,...,l\}$, the weight matrices of the model
Require: $b^{(i)}, i \in \{1,...,l\}$, the bias parameters of the model
Require: $g^{(i)}(\cdot), i \in \{1,...,l\}$, the list of activation functions
Require: $J(\cdot)$, the cost function
Require: y, the target output
Require: $\hat{y}$, the estimate of the output computed in the forward propagation

▷ Compute the gradient of the loss w.r.t the output layer.

$$\delta \leftarrow \frac{dL}{\mathrm{d}\hat{y}} = \frac{\mathrm{d}J(\hat{y}, y)}{\mathrm{d}\hat{y}}$$

for $k \leftarrow l, l-1, ..., 1$ **do**

▷ Propagate the gradient through the nonlinearity – convert the gradient w.r.t the layer's output $h^{(k)} = g^{(k)}(a^{(k)})$ into a gradient w.r.t the layer's pre-nonlinearity activation $a^{(k)} = W^{(k)}h^{(k-1)} + b^{(k)}$. Element-wise multiplication if $g^{(k)}$ is element-wise.

$$\delta \leftarrow \frac{\mathrm{d}L}{\mathrm{d}a^{(k)}} = \delta \odot \frac{\mathrm{d}g^{(k)}}{\mathrm{d}a^{(k)}}$$

▷ Compute the gradients w.r.t the parameters. Hereafter the gradients can be used to immediately update the parameters using the GD update rule. It is common to store the values of $a^{(i)}$ and $h^{(i)}$ in memory after the forward propagation, such there is no need to recompute them when computing the gradient.

$$\frac{\mathrm{d}L}{\mathrm{d}b^{(k)}} = \delta$$

$$\frac{\mathrm{d}J}{\mathrm{d}W^{(k)}} = \delta h^{(k-1)T}$$

▷ Propagate the gradient through the linear part of the layer – convert the gradient w.r.t the layer's pre-nonlinearity activation $a^{(k)}$ into the gradient w.r.t the next lower-level layer's output $h^{(k-1)}$.

$$\delta \leftarrow \frac{\mathrm{d}L}{\mathrm{d}h^{(k-1)}} = W^{(k)T}\delta$$

end for

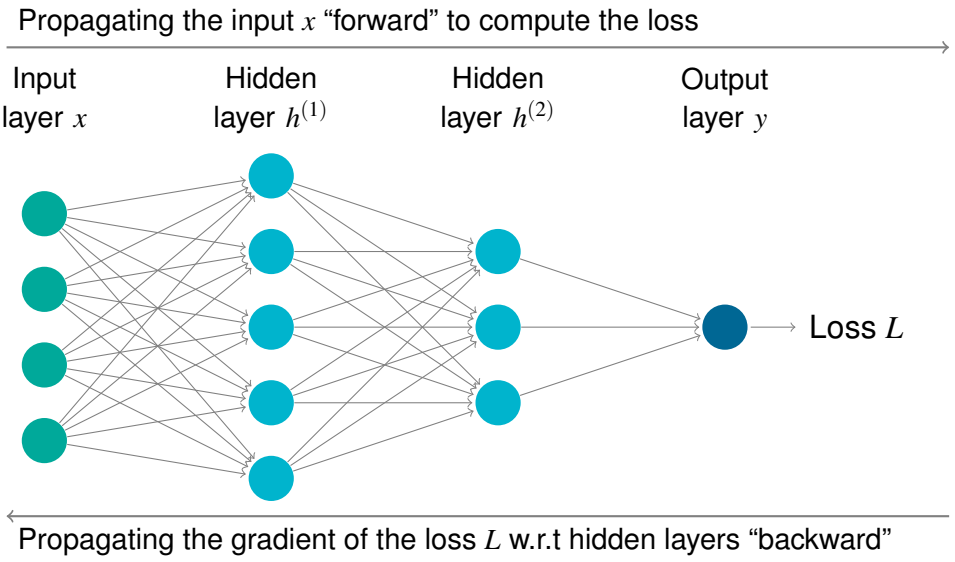

Figure 7. Forward and backward propagation

4.3 Normalization

Normalization is a common technique used to improve GD-based training. The core idea of normalization is to scale the values of the data to be in the same fixed interval. In the machine learning context one usually normalizes the features, making the range of values taken by the elements of columns of the design matrix X be the same for each column.

There are multiple kinds of normalization in statistics. In machine learning *min-max normalization* and *standard score normalization* are commonly used.

Min-max normalization adjusts the values of a vector x to be in a range $[a,b]$. In the vector form the adjustment is:

$$x_{normalized} = a + \frac{(x - x_{min})(b-a)}{x_{max} - x_{min}}, \tag{24}$$

where x_{max} and x_{min} are correspondingly the largest and the smallest elements of x. Often the desired interval $[a,b]$ is the interval $[0,1]$, in which case the adjustment is simply:

$$x_{normalized} = \frac{(x - x_{min})}{x_{max} - x_{min}}. \tag{25}$$

Standard score normalization adjusts the values of a vector x to have mean 0 and standard deviation 1. The adjustment consists of subtracting the mean μ of x from each of the elements of x, and dividing the result by the standard deviation σ. In the vector form this can be written as:

$$x_{normalized} = \frac{(x - \mu)}{\sigma}. \tag{26}$$

The reason normalization is often used in machine learning is its stabilizing effect on GD-based training. Figure 8 shows two hypothetical gradient descent trajectories with and without data normalization prior to training. Updates after each GD iteration are shown with black arrows. The length of a black arrow corresponds to the learning rate η at that iteration. Normalizing the input's features usually indirectly leads to the parameters θ being roughly on the same scale, which results in smoother descent trajectories and fewer iterations until convergence. This can be seen on Figure 8 comparing the GD trajectory without input normalization (left) with the GD trajectory on the normalized input (right).

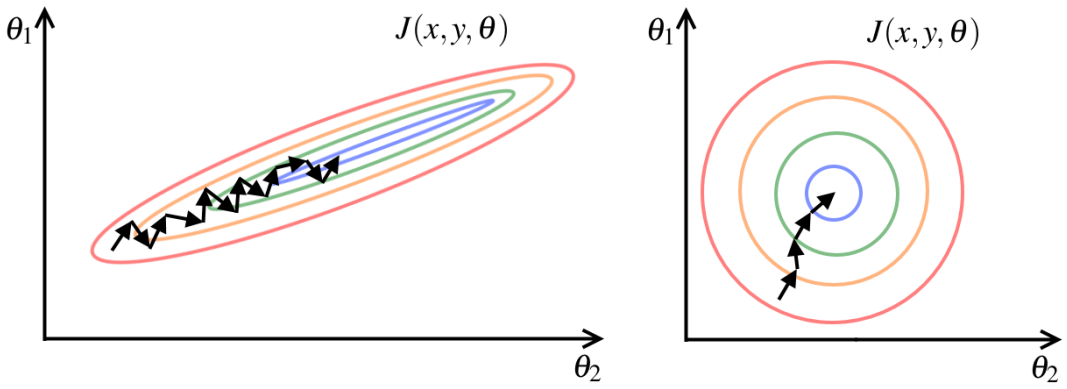

Figure 8. Effect of normalization on GD convergence

4.4 Regularization

Regularization is a common technique used to prevent overfitting and improve generalization of machine learning models. The core idea behind regularization is incorporating additional information about the desired solutions into the model. The most common example of such information is a preference for simpler models, which can be viewed as imposing Occam's razor on the solution. Another common example of the additional information is a preference for sparsity in some part of the model. From the Bayesian viewpoint regularization can be seen as a prior on the model's parameters θ.

Below two most common regularization methods, L2-norm penalty and Dropout regularization, are introduced.

4.4.1 L2-norm Penalty

L2-norm penalty is one of the oldest and most well-known regularization methods in machine learning. L2-norm penalty consists of adding a regularization term $\lambda||\theta||^2$ to the cost function $J(x,y,\theta)$:

$$L = J(x,y,\theta) + \lambda||\theta||^2. \tag{27}$$

Here λ is usually a small constant and $||\theta||^2$ is the squared L2 norm of the parameters θ, which is simply a sum of squares of each of the elements of θ. The newly added regularization term is differentiable, which allows using GD-based methods for training the models using L2-norm penalty.

This penalty can be seen as a preference for the values of θ obtained during training to be closer to zero, which usually results in the model capturing less noise in the data and therefore better generalization to the unseen data. However, very high values of λ can result in the regularization term dominating the cost, which often leads to degradation of model's performance.

4.4.2 Dropout Regularization

Dropout (Srivastava et al. 2014) is a technique widely used to regularize deep neural networks. The core idea behind dropout is adding multiplicative noise to the output of a hidden layer. Concretely, in the forward propagation stage each of the elements of a hidden layer is set to zero with probability p. Analogously to Equation 18, a hidden layer with dropout applied has the following form:

$$r^{(l)} = \text{Bernoulli}(p) \tag{28}$$

$$\widetilde{h^{(l)}} = h^{(l)} \odot r^{(l)} \tag{29}$$

$$h^{(l+1)} = g(W^{(l+1)}\widetilde{h^{(l)}} + b^{(l+1)}) \tag{30}$$

When dropout is used each neuron is forced to work with a randomly chosen sample of the neurons from the next layer, which results in a higher degree of redundancy in the NN. Additionally, dropout drives the neurons to learn more accurate features as other neurons that were correcting for their mistakes may be switched off. This makes the network more robust, often increases the accuracy and prevents overfitting.

5 AUTONOMOUS CONTROL OF A RC CAR WITH A CONVOLUTIONAL NEURAL NETWORK

This chapter presents the methodology for solving the problem of autonomous control of a remote controlled (RC) car. First the project setup and an overview of the solution are introduced, followed by the details of the solution steps.

As stated in the introduction, a system that allows a remote controlled car autonomously follow a track on the floor made of sticky notes using the imagery from the car's built-in camera is developed.

In order to do this we trained a convolutional neural network (CNN) to map pix-

els from processed images taken from the single front-facing camera directly to steering and acceleration commands. This proved to be a powerful approach: without any feature engineering the system automatically learned relevant features, such as the borders of the track and the direction of movement in the room.

5.1 Previous and Related Work

The discussion of autonomous driving began as early as the 1920s, but it was not until the 1980s that the first self-sufficient autonomous vehicles appeared. Notable pioneers were CMU's Navlab 1 (Thorpe et al. 1988) and ALVINN (Pomerleau 1989) projects, as well as the European PROMETEUS project (Williams 1988).

The idea of using a neural network to predict the control commands is not new. For example, Pomerleau (1989) used a fully-connected neural network (one hidden layer with 29 hidden units) to predict steering commands for the vehicle in the ALVINN project. This neural network is tiny by the modern standards, and as time goes on the researchers of autonomous driving are able to use significantly more computational power to run their systems.

More recently, DARPA seeded a project named DAVE, or DARPA Autonomous Vehicle (Net-Scale Technologies 2004). The approach taken in this thesis is in many ways similar to the one described in DAVE: both use sub-scale RC cars as experimental vehicles and both use convolutional neural networks to predict the car control commands. Inspired by the DAVE project, the NVIDIA team trained a large CNN mapping images obtained from driving a real car to the steering commands (Bojarski et al. 2016). This thesis takes inspiration from both the approach taken by the Net-Scale Technologies team and the NVIDIA team.

5.2 Methodology

As the experimental vehicle we use the *Parrot Jumping Sumo* car with a built-in camera, shown in Figure 9. The camera's resolution is 480 $\times$ 640 pixels, and the frame rate is 15 frames per second. The car creates its own Wi-Fi hotspot which it uses to transmit the images and receive the control commands. We connect our PC to this hotspot and control the vehicle remotely. The car

runs *Robot Operating System* (ROS) locally, and communicates with the PC via *ARDroneSDK3*, the official Parrot SDK. For training the CNN we use Python with Theano (Theano Development Team 2016), a library that allows for efficient manipulation of expressions involving multidimensional arrays, features symbolic differentiation and transparent use of a GPU. We also use Lasagne (Dieleman et al. 2015), a high-level wrapper library for Theano to speed up the coding. The CNN is trained using a NVIDIA Titan X 2015 GPU.

Figure 9. The Parrot Jumping Sumo car (Parrot Development Team 2016)

More formally, the overall structure of the project is as follows:

1. Implementing the car control system;
2. Collecting video frames with corresponding control commands by manually driving the car around various tracks;
3. Training a CNN to predict control commands from the obtained video frames;
4. Evaluating the performance of the CNN controlling the car on the track.

The next sections describe each of the steps above in more detail.

5.3 RC Car Control System

In order to receive and record the images from the car and send back the corresponding control commands a RC car control system is needed. The car control system is implemented in Python, as it is then easy to use it together with the image recognition system developed for controlling the car. The implemented control system relies on Parrot's ARDroneSDK3 and Rossumo, a low level library for the Jumping Sumo car developed by Ramey (2016).

The tasks performed by the RC car control system include the following:

1. **Receiving the video**: receiving images from the car, optionally displaying them and buffering the last n of them. To receive the images ARDroneSDK3 and Rossumo are used, which together create a ROS communication channel for the images. The control system is subscribed to this channel and gets the images as they arrive. Each time a new image is received it is appended to a small buffer `imageQueue`, and is optionally displayed.

2. **Car control**: reading the joystick commands in real time and sending them to the car at the same rate at which the frames are received. For the simplicity of handling the joystick commands a second ROS channel is created. The control system is subscribed to this channel and gets the joystick commands as they arrive. In order to collect time-aligned pairs of [`image`, `joystick Command`], the frequency of receiving joystick commands is set to 15 Hz, same as the frequency at which the images are received. Joystick commands are buffered at `joystickQueue` and by default are sent to the car.

3. **Data collection:** recording an arbitrary sized array of images from `image_Queue` and their corresponding joystick commands from `joystickQueue`. Several such arrays are recorded and later used for training the CNN which predicts the joystick commands from the images. To record a large number of pairs of [`image`, `joystick Command`], the original 480 $\times$ 640 RGB images from `imageQueue` are converted to grayscale and hereafter downsampled to 120 $\times$ 160 resolution. This way each of the processed images takes 48 times less RAM than the original, allowing us to record arrays of

up to 20 thousand images at once.

4. **Autopilot**: the control system is implemented such that it is possible to seamlessly plug in an autonomous car control module. The autopilot implemented in this thesis uses a CNN for predicting the control commands, and there is a flexibility to use other autopilot modules too.

The data flow in the implemented system is shown in Figure 10. The next section describes the workings of the autopilot in more detail.

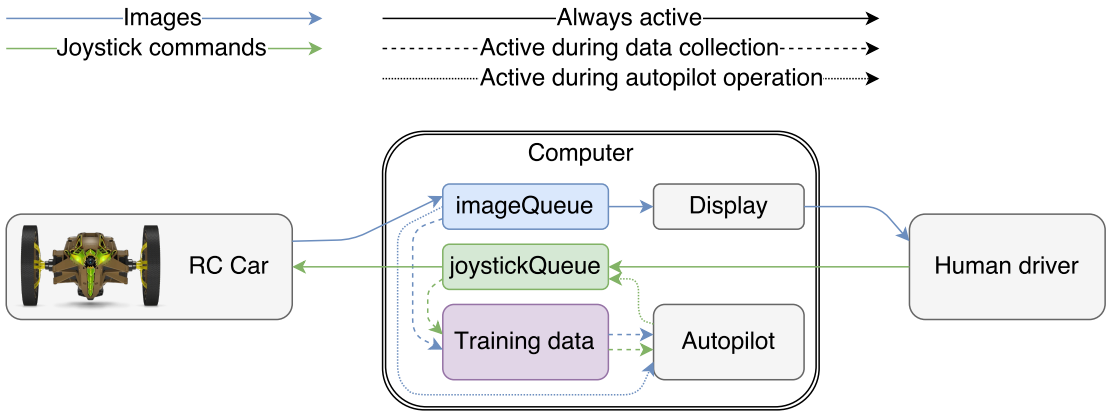

Figure 10. Data flow in the implemented car control system

5.3.1 Autopilot

The central part of the autopilot is a CNN predicting the car control commands from the real-time, 15 frames per second video stream. This implies that for a simple single-threaded program the time required to process a single frame must be on the order of 1/15s or 67ms in order to maintain small constant response delay. The image preprocessing and the forward pass of the CNN chosen as the central component of the command prediction system (CPS) fit into this time window.

One of the buttons on the joystick acts as an `autopilotFlag`: once this button is pressed the control system starts the CPS. The CPS reads the first image from the `imageQueue`, processes it and uses it to predict the corresponding car control command. If the `autopilotFlag` is on, the control system prioritizes sending the commands from the CPS to the car over the "no action" commands received from the joystick. However, if the command from the joystick is different from "no action", it is prioritized over the command predicted by CPS and the autopilot is switched off. This way one can correct the car's course without having to

manually turn off the autopilot. Additionally, the autopilot can be disabled by pressing the button that turned it on once more. Algorithm 4 demonstrates the autopilot operation with image and joystick buffering routines omitted for simplicity.

Algorithm 4. Autopilot

Require: `CPS`, the control prediction system object with a method `predict` which outputs a control `command` given an `image`. In the `CPS` implemented in this thesis `predict` is a feedforward computation of a CNN.

Require: `imageCh`, a ROS channel subscribed to the car's camera. Event `imageCh.receivedNew()` occurs as a new image is received to this channel. The latest image can be read with the method `imageCh.read()`.

Require: `joystickCh` a ROS channel subscribed to the joystick. The latest joystick command can be read with the method `joystickCh.read()`. The default state of the joystick axes corresponding to car movement is referred to as `defaultState`. Default state of a real car is then zero accelerator pedal pressure and the central position of the steering wheel.

Require: `imageQueue` and `joystickQueue`, queues where the received images and joystick commands are buffered.

```
autopilotFlag ← False
while system is on do
    upon event autopilotButtonPressed do
        autopilotFlag ← Not(autopilotFlag)
    upon event imageCh.receivedNew() do
        imageQueue.push(imageCh.read())
        commandQueue.push(joystickCh.read())
    if imageQueue is not empty then
        image ← imageQueue.pop()
        command ← joystickQueue.pop()
        if command ≠ defaultState then
            autopilotFlag ← False
        if autopilotFlag is True then
            command ← CPS.predictCommand(image)
        rcCar.sendCommand(command)
end while
```

5.4 Predicting the Control Commands from Images

This section describes the methodology for training the convolutional network, which is later used as the main component of the CPS. However, before training the CNN we must decide how much and which kinds of data to collect, and whether to use additional image preprocessing. After this we settle on the architecture of the network and train it.

5.4.1 Data Collection and Augmentation

The car's task is to follow a track on the floor which suggests that most of the training data, pairs of [image, joystick Command], has to be recorded from manually driving the car on the track. Additionally, the car may lose the track from its camera view – this would sometimes happen at sharp turns of the track. In this case the reasonable courses of action could be:

- Stop the car, stop the autopilot and require human intervention to start following the track again.
- Stop the car and slowly rotate in place until the track is in the field of camera's view, and continue following the track.

In this thesis we follow the latter approach. To achieve the desired behaviour, in addition to the regular driving data we collect several sets of pairs of [image, joystick Command] from situations where the car returns to the track after losing sight of it for some period of time.

As driving the car around the track is a fairly tedious process (and a rather expensive one for a real car), we have a preference for being data-efficient: collecting only as much data as is needed to perform the task well. To improve the data-efficiency the collected data is augmented with images mirrored horizontally. The corresponding steering command is also "mirrored" to encode steering with the same magnitude, but in the opposite direction.

We collect 42K pairs of [image, joystick Command] of the car driving around the track normally, and 8K pairs of the car returning back to track after losing it from sight and driving on the track afterwards. In total this amounts to 50K examples or roughly 1 hour of driving time. After the data augmentation the dataset size doubles to 100K samples.

5.4.2 Image Preprocessing

To further improve data-efficiency, the images are preprocessed such that the variance in the data distribution is reduced, making the data easier to model. The requirement for preprocessing is that it must be possible to successfully

control the car using the preprocessed images. The following preprocessing steps are introduced:

1. Cropping the upper 60 percent of the stored 120 $\times$ 160 images such that the new resolution becomes 48 $\times$ 160. This way most of the car's visual field is focused on the floor with the track, as opposed to less relevant features of the indoor space such as desks and upper parts of the chairs.

2. Downsampling the image fourfold, from 48 $\times$ 160 to 12 $\times$ 40. Even at such a small resolution it is easy to see the track and distinguish its finer features.

3. Normalizing both the image pixels and the control commands to have values in the range between 0 and 1. As we know the minimum and the maximum of possible values for both pixels and the control commands, we perform the min-max normalization (Equation 25).

The cropping and downsampling steps of the preprocessing procedure applied to one of the images are shown in Figure 11.

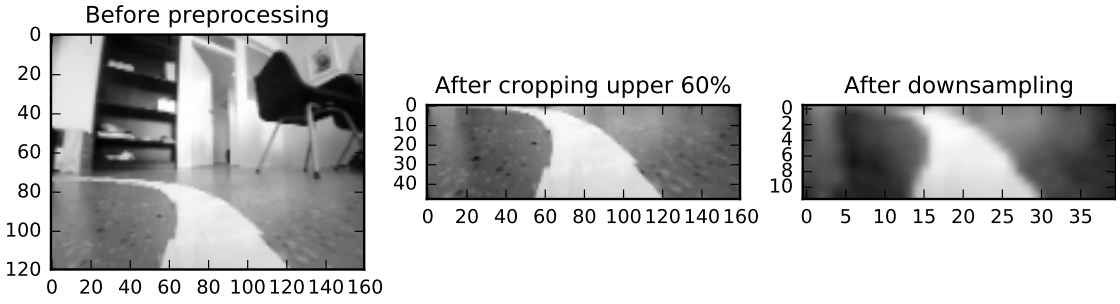

Figure 11. Preprocessing a stored image

To summarize, the data used for training the CNN is of the following form: preprocessed images $x^{(i)} \in \mathbb{R}^{12\times40}$ and their corresponding control commands $y^{(i)} \in \mathbb{R}^2$, where $i \in (1, ..., 100\text{K})$.

For training the network the 100K examples are randomly split into the training set containing 80K samples and the test set containing 20K samples. The training data is used to train the CNN while the test data is used to obtain an accurate evaluation of the network's performance.

5.4.3 Training the Convolutional Neural Network

The CNN used for predicting the control commands from images has the following architecture, from input to output. Each layer's output is the subsequent layer's input:

- `input` The input layer
- `conv1` A convolutional layer: 32 kernels with 3 $\times$ 3 kernel size
- `conv2` A convolutional layer: 32 kernels with 3 $\times$ 3 kernel size
- `pool1` A max-pooling layer with 2 $\times$ 2 kernel
- `conv3` A convolutional layer: 32 kernels with 3 $\times$ 3 kernel size
- `conv4` A convolutional layer: 32 kernels with 3 $\times$ 3 kernel size
- `conv5` A convolutional layer: 32 kernels with 3 $\times$ 3 kernel size
- `pool2` A max-pooling layer with 2 $\times$ 2 kernel and dropout ($p = 0.3$)
- `fc1` A fully-connected layer (128 units) and dropout ($p = 0.3$)
- `output` The output layer with 2 units

The schema of this architecture is shown in Figure 12.

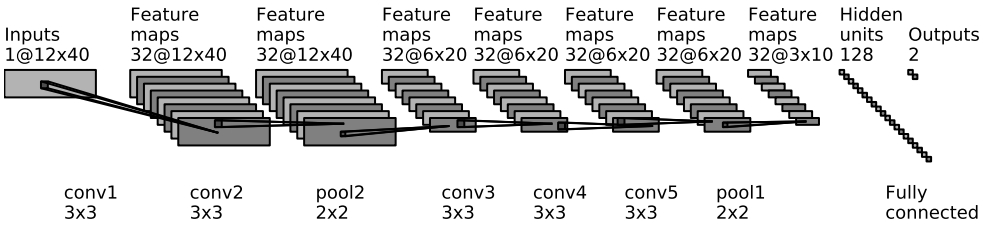

Figure 12. Architecture of the CNN used in the CPS

The RELU activation function is used for all the layers except the `output` layer. The `output` layer is a linear layer as the network is performing regression. Therefore, the `output` layer can be seen as performing multivariate linear regression on the output of the `fc1` layer.

The loss used to train the CNN is a sum of the mean squared error (Equation 5)

and the L2-norm penalty (Equation 27). The value of λ used for the L2-norm penalty is 3×10^{-4}. The L2-norm penalty and dropout (Equation 30) are used to regularize the network and counter the likely overfitting problem, which has a good chance of occurring given the relatively small dataset size.

The CNN is trained using the minibatch gradient descent algorithm with momentum (Section 4.1). The gradients of the loss w.r.t. the parameters are computed with the backpropagation algorithm (Section 4.2). The network is trained for 100 epochs, meaning that the weights are updated 100 times on each of the 80K training examples.

5.5 Evaluating the Performance of the Autonomous RC Car

Measures commonly used to evaluate the performance of autonomous vehicles are *number of accidents per 1M kilometers* and *autonomy*, the percentage of the driving time the car could be on autopilot. The latter measure is more applicable to evaluate the results of this thesis. Bojarski et al. (2016) measure the number of human *interventions* required for safe driving – moments when the human driver needs to take over the autopilot. They define autonomy as:

$$\text{autonomy} = (1 - \frac{(\text{number of interventions}) \times (\text{time per intervention})}{\text{elapsed time}}) \times 100 \qquad (31)$$

Unfortunately, due to time constraints we were not able to calculate the value of *autonomy* precisely. During several tests the Jumping Sumo car was able to autonomously follow the track for 30–120 seconds before requiring an intervention by a human driver. Following Bojarski et al. (2016) we assume 6s required per intervention. This suggests that the autonomy value of the RC car using the CNN to predict the control commands is in the range of 80–95 percent.

6 CONCLUSIONS AND FUTURE WORK

The practical goal of this thesis was to build a simple and reliable testbed for the evaluation of algorithms for autonomous vehicles and to implement a baseline car control algorithm. This was accomplished by developing an autopilot system on the Parrot Jumping Sumo RC car and implementing a convolutional neural network (CNN) that predicts the car control commands from the imagery from the car's camera. The CNN was trained on one hour of driving data and was

able to stay on the track for 30–120 seconds in the autopilot mode without requiring an interruption by a human driver. To understand the workings of CNNs, the theoretical part of this thesis explored the construction of CNNs from simple components, methodology for training the CNNs and several practical methods often used in CNNs, such as regularization and data normalization.

More work is needed to improve the robustness of the implemented system as well as to attain a precise evaluation of the system's robustness. These areas can be improved significantly without introducing large changes to the implemented system by collecting more driving data and dedicating additional time to evaluating the car's performance.

- Collecting more data from driving on a larger number of tracks and training the CNN on this data would likely result in a CNN with better generalization capabilities. This would entail the car's ability to accurately navigate a wider range of tracks in different light conditions. Minor changes in the image preprocessing and in the CNN architecture may be required. Additionally, the system can be taught to perform new tasks, such as stopping when encountering an obstacle on the track.

- Rigorous evaluation of the car's *autonomy* (Equation 31) following the methodology outlined in Section 5.5 would allow for clear communication of the existing results and would serve as an important metric for measuring the future progress. Another benefit of using such metric would be the ability to compare the results of this work to the results obtained by other research teams such as Bojarski et al. (2016).

The progress in these two areas is straightforward, with time being the main bottleneck. Given the time and resource constraints of the project, the improvements mentioned above are outside the scope of this Bachelor's thesis.

REFERENCES

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T. & Freitas, N.de . Learning to learn by gradient descent by gradient descent. In Advances in Neural Information Processing Systems, 3981–3989, 2016.

Ayodele, T. O. 2010. Types of machine learning algorithms.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J. et al. 2016. End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316.

Dieleman, S., Schlüter, J., Raffel, C., Olson, E., Sønderby, S. K., Nouri, D., Maturana, D., Thoma, M., Battenberg, E., Kelly, J., Fauw, J. D., Heilman, M., Almeida, D. M.de , McFee, B., Weideman, H., Takács, G., Rivaz, P.de , Crall, J., Sanders, G., Rasul, K., Liu, C., French, G. & Degrave, J. August 2015. Lasagne: First release. Python package. Available at: http://dx.doi.org/10.5281/zenodo.27878. [Accessed 17 March 2017].

Domingos, P. 2012. A few useful things to know about machine learning. Communications of the ACM, 55(10):78–87.

Dreyfus, S. 1962. The numerical solution of variational problems. Journal of Mathematical Analysis and Applications, 5(1):30–45.

Evans, R. & Gao, J. 2016. DeepMind AI Reduces Google Data Centre Cooling Bill by 40 percent. WWW document. Available at: https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/. [Accessed 29 January 2017].

Ghahramani, Z. 2015. Parametric vs Nonparametric Models. Lecture notes. Available at: http://mlss.tuebingen.mpg.de/2015/slides/ghahramani/gp-neural-nets15.pdf. [Accessed 29 January 2017].

Glorot, X., Bordes, A. & Bengio, Y. Deep sparse rectifier neural networks. In Aistats, volume 15, 275, 2011.

Goodfellow, I., Bengio, Y. & Courville, A. 2016. Deep Learning. MIT Press. Available at: http://www.deeplearningbook.org. [Accessed 5 January 2017].

Hornik, K., Stinchcombe, M. & White, H. 1989. Multilayer feedforward networks are universal approximators. Neural networks, 2(5):359–366.

Kelley, H. J. 1960. Gradient theory of optimal flight paths. Ars Journal, 30(10): 947–954.

Kingma, D. & Ba, J. 2014. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P. 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324.

LeCun, Y., Bengio, Y. & Hinton, G. 2015. Deep learning. Nature, 521(7553): 436–444.

Linnainmaa, S. 1970. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 6–7.

Mitchell, T. 1997. Machine learning. Available at: https://books.google.com/books?id=EoYBngEACAAJ. [Accessed 13 December 2016].

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. 02 2015. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533. Available at: http://dx.doi.org/10.1038/nature14236. [Accessed 17 January 2017].

Murphy, K. P. 2012. Machine learning: a probabilistic perspective. MIT press.

Net-Scale Technologies. Autonomous off-road vehicle control using end-to-end learning. Technical report, 2004. Available at: http://net-scale.com/doc/net-scale-dave-report.pdf. [Accessed 17 March 2017].

Ng, A. 2013. CS229 Stanford Lecture notes. Lecture notes. Available at: http://cs229.stanford.edu/notes/cs229-notes1.pdf. [Accessed 23 March 2017].

Parrot Development Team. 2016. The Parrot Jumping Sumo car. WWW document. Available at: https://www.parrot.com/us/minidrones/parrot-jumping-sumo#parrot-jumping-sumo. [Accessed 29 March 2017].

Pomerleau, D. A. Alvinn, an autonomous land vehicle in a neural network. Technical report, Carnegie Mellon University, Computer Science Department, 1989.

Ramey, A. 2016. Rossumo: a wrapper of the ARDroneSDK3 sample "JumpingSumoPiloting.c" as a C++ lightweight class for ROS. Git Repository. Available at: https://github.com/arnaud-ramey/rossumo. [Accessed 17 March 2017].

Ruder, S. 2016. An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Driessche, G.van den , Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. & Hassabis, D. 01 2016. Mastering the game of go with deep neural networks and tree search. Nature, 529(7587):484–489.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. 2014. Dropout: a simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(1):1929–1958.

Sutherland, D. J. 2015. General assumptions about functions in machine learning. WWW document. Available at: http://stats.stackexchange.com/questions/154439/general-assumptions-about-functions-in-machine-learning. [Accessed 29 January 2017].

Theano Development Team. May 2016. Theano: A Python framework for fast computation of mathematical expressions. arXiv e-prints, abs/1605.02688. Available at: http://arxiv.org/abs/1605.02688. [Accessed 17 March 2017].

Thorpe, C., Hebert, M. H., Kanade, T. & Shafer, S. A. 1988. Vision and navigation for the carnegie-mellon navlab. IEEE Transactions on Pattern Analysis and Machine Intelligence, 10(3):362–373.

Williams, M. Prometheus-the european research programme for optimising the road transport system in europe. In Driver Information, IEE Colloquium on, 1–1. IET, 1988.

Wolpert, D. H. & Macready, W. G. 1997. No free lunch theorems for optimization. IEEE transactions on evolutionary computation, 1(1):67–82.

Zhu, X. & Goldberg, A. B. 2009. Introduction to semi-supervised learning. Synthesis lectures on artificial intelligence and machine learning, 3(1):1–130.