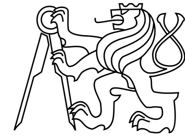
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	Detection of landing platform for drones
Student:	Jan Rudolf
Supervisor:	Mgr. RNDr. Petr Št pán, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of winter semester 2018/19

Instructions

Study convolution neural networks for object detection in a camera image.
Design a structure of a neural network to detect landing patterns from a drone's camera.
Multi Robot System Group at the Department of Cybernetics FEE CTU has a detection algorithm based on standard computer vision techniques.
Use this detection algorithm and manual labels to create a dataset specifying the center position of the landing pattern in the image.
Train the neural network on the created dataset, test the effect of the neural network structure on the output quality with respect to processing of data in real time.
Compare the neural network algorithm with the current detection algorithm and document the results.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

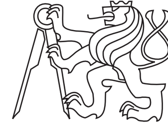
prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague March 2, 2017

Czech Technical University in Prague

Faculty of Information Technology

Department of theoretical computer science



Bachelor's thesis

Detection of landing platform for drones

Jan Rudolf

Supervisor: Mgr. RNDr. Petr Štěpán, Ph.D.

16th May 2017

Acknowledgements

In the first place, I would like to thank my supervisor Mgr. RNDr. Petr Štěpán, Ph.D. for his guidance, friendly and helpful approach. I would also like to thank MetaCentrum VO for providing the computing resources. Finally, I would like to thank my family for their support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 16th May 2017 ....

Czech Technical University in Prague
Faculty of Information Technology
© 2017 Jan Rudolf. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Rudolf, Jan. *Detection of landing platform for drones.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Práce se zabývá použitím konvolučních neuronových sítí pro detekci objektu v obrazu. Od vytvoření datasetu na základě dat z kamery drona s využitím detekčního algoritmu používající standardní metody počítačového vidění, po návrh konvoluční neuronové sítě trénované pro detekci přistávací plochy. Práce dává úvod do strojového učení, neuronových sítí a jejich praktického použití s programovacím jakykem Python.

Klíčová slova neuronové sítě, konvoluční neuronové sítě, strojové učení, detekce objektu, Python

Abstract

The thesis is about using convolutional neural networks for visual object detection. The work guides from making a dataset from a drone's camera using a provided detection algorithm based on standard computer vision methods, to design the convolutional neural network capable of detecting

drone's landing platform. The thesis introduces to machine learning, neural networks and their practical usage in Python.

Keywords neural networks, convolutional neural networks, machine learning, object detection, Python

Contents

Introduction **1**

1 Machine learning **3**
1.1 Supervised machine learning 4
1.2 Concept of learning 4
1.3 Optimization 5
1.4 Underfitting and overfitting 8
1.5 Summary 10

2 Artificial neural networks **13**
2.1 Basics of a biological neuron 13
2.2 Mathematical model 15
2.3 Multilayer feed-forward neural networks 17
2.4 Convolutional neural networks 26
2.5 Prevention of overfitting 32

3 Solution **33**
3.1 Making the dataset 33
3.2 Implementation 34
3.3 Network architectures 34
3.4 Measurements 39
3.5 Discussion 39

Conclusion **47**

Bibliography **49**

A Acronyms **51**

B Contents of enclosed USB **53**

List of Figures

1.1 We fit three models to this example training set. The training data was generated synthetically, by randomly sampling values and choosing deterministically by evaluating a quadratic function. (Left) A linear function fit to the data suffers from underfitting — it cannot capture the curvature that is present in the data. (Center) A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significiant amount of overfitting or underfitting. (Right) A polynomial of degree 9 fit to the data suffers from overfitting. The solution passes through all of the training points exactly, but we have not been lucky enough for it to extract the correct structure. It now has a deep valley in between two training points that does not appear in the true underlying function. It also increases sharply on the left side of the data, while the true function decreases in this area [4]. 10

1.2 Typical relationship between capacity and error. Training and test error behave differently. At the left end of the graph, training error and generalization error are both high. This is the underfitting regime. As we increase capacity, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error, and we enter the overfitting regime, where capacity is too large, above the optimal capacity [4]. 11

2.1 An illustration of a biological neuron [6]. 14

2.2 An illustration of the simplified mathematical model of a biological neuron [6]. 15

2.3 An example of a feed-forward neural network with 3 layers [6]. . 16
2.4 An example of a multilayer feed-forward neural network with an input layer with nodes 1 and 2. A hidden layer with nodes 3 and 4. Finally, an output layer with units 5 and 6. 18
2.5 The Sigmoid activation function. 21
2.6 The Tanh activation function. 21
2.7 The Rectified Linear Unit (ReLU). 22
2.8 The Leaky Rectified Linear Unit (Leaky ReLU) with $\alpha = 0.2$. . 22
2.9 An illustration of the standard convolutional neural network [6]. 28
2.10 Output of the filter [5]. 29
2.11 The filter convolves with a stride 1 [5]. 30
2.12 A convolutional layer has several filters [5]. 30

List of Tables

3.1 Architecture 1 measurement on the training set. 40
3.2 Architecture 1 measurement on the validation set. 40
3.3 Architecture 1 measurement on the test set. 41
3.4 Architecture 2 measurement on the training set. 41
3.5 Architecture 2 measurement on the validation set. 42
3.6 Architecture 2 measurement on the test set. 42
3.7 Architecture 3 measurement on the training set. 43
3.8 Architecture 3 measurement on the validation set. 43
3.9 Architecture 3 measurement on the test set. 44
3.10 Comparison between architecture 1 and the reference program. . 44
3.11 Comparison between architecture 2 and the reference program. . 45
3.12 Comparison between architecture 3 and the reference program. . 45

Introduction

Neural networks are computational models that are among us a very long time. They have seasons when they are very popular, and seasons when they are not so much. They are very popular now, because of the results that are often the state-of-the-art, thanks to quick hardware and a lot of data. Neural networks are used for example in visual recognition or self-driving cars.

The goal of my thesis is to study convolutional neural networks for object detection in a camera image. Design a neural network to detect landing patterns from a drone's camera. Use a current detection algorithm provided by the supervisor and video data from the camera to create a dataset for the neural network. Train the neural network on the generated dataset. Compare the prediction by the neural network and the current detection algorithm and document the results.

Chapter 1 presents the basic ideas of machine learning. Chapter 2 studies multilayer and convolutional neural networks. Chapter 3 describes a creation of the dataset, proposes neural networks for the goal of detecting the landing platform and compare the prediction with the current algorithm.

Chapter 1

Machine learning

According one of the leaders of machine learning, Andrew Ng, the first self-learning program created Arthur Samuel in 1959 [8]. Samuel created a program playing checkers capable playing againts human players with a decent success. In order to make his program better, he let it played thousand of times againts itself. Samuel stated, that machine learning is a field, which gives computers the ability to learn without being explicitly programmed.

Kevin P. Murphy in his book [10] defines machine learning as a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty (such as planning how to collect more data).

Tom M. Mitchell provides often mentioned definition: 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. In the context of checkers, E would be playing for thousand of times, T the game checkers and E the probability of winning.

This chapter is based on [11], [12], [6], [5].

1.1 Supervised machine learning

Supervised machine learning, often called learning with a teacher, is a paradigm of machine learning, where we have available a dataset with input-output pairs $\{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$ of N data, where each y_i some process or system generates with an unknown function $y = f(x)$. We are trying to find a function h that approximates the true function f. Each x_i and y_i can be a scalar, a vector etc. The function h is called a hypothesis. Learning is a search through the space of possible hypotheses, $\mathcal{H}$, for one that will perform well, even on new examples beyond the training set.

When the output y is one of a finite set of values (such as sunny, cloudy or rainy), the learning problem is called classification, and is called Boolean or binary classification if there are only two values. When y is a number (such as tomorrow's temperature), the learning task is called regression.

1.2 Concept of learning

Assume, that our hypothesis space $\mathcal{H}$ is a set of all polynomials with real coefficients and the max degree of two:

$$\mathcal{H} = \{\theta_2 x_2^2 + \theta_1 x_1 + \theta_0 | \theta_2, \theta_1, \theta_0 \in \mathbb{R}\} \tag{1.1}$$

The hypothesis function h with inputs x_2, x_1 and parameters θ_2, θ_1 and θ_0 is:

$$h_{\theta_2,\theta_1,\theta_0}(x_2, x_1) = \theta_2 x_2^2 + \theta_1 x_1 + \theta_0 \tag{1.2}$$

The usual notation is $\mathbf{x}^T = (x_2, x_1)$ vector for inputs and $\theta^T = (\theta_2, \theta_1, \theta_0)$vector for parameters, therefore $h_\theta(\mathbf{x})$:

$$h_\theta(\mathbf{x}) = \theta_2 x_2^2 + \theta_1 x_1 + \theta_0 \tag{1.3}$$

Suppose, we choose values for coefficients θ_2, θ_1 and θ_0. We would like to know, for each $(\mathbf{x_i}, \mathbf{y_i})$ from our dataset $\{(\mathbf{x_1}, \mathbf{y_1}), (\mathbf{x_2}, \mathbf{y_2}), ..., (\mathbf{x_N}, \mathbf{y_N})\}$, how good our hypothesis is and measure the error between output from the dataset $\mathbf{y_i}$ and the prediction made by our hypothesis $h_\theta(x)$. For this purpose, we define a loss function L (or sometimes also referred to as the cost function or the objective):

$$L : \mathbb{R}^m \times \mathbb{R}^m \mapsto \mathbb{R} \tag{1.4}$$

where $m \in \mathbb{N}$ is a dimension of the output vector. The most common and convenient is squared loss:

$$L(\mathbf{y_i}, h_\theta(\mathbf{x_i})) = (\mathbf{y_i} - h_\theta(\mathbf{x_i}))^2 \tag{1.5}$$

We seek for the hypothesis h_θ with parameters θ in the hypothesis space $\mathcal{H}$, that minimize the expected loss over our dataset:

$$h_\theta = \arg\min_\theta E(L(\mathbf{y}, h_\theta(\mathbf{x}))) \tag{1.6}$$

It turns out, for choice of the squared loss function (MSE - mean squared error), learning is an optimization problem of minimization the avereage squared error:

$$h_\theta = \arg\min_\theta \frac{1}{N} \sum_{i=1}^{N} (y_i - h_\theta(x_i))^2 \tag{1.7}$$

We choose different types of functions instead of polynomials here. For example K-Nearest Neighbor, Support Vector Machines or Neural Networks.

1.3 Optimization

Mathematical optimization (also called mathematical optimisation, mathematical programming) concers about finding the minimum (or the maximum) of real-valued function $f : X \mapsto \mathbb{R}$ on some set X.

There are three main categories:

- If the set X is finite, we talk about combinatorial optimization.
- If the set X is composed of real vectors, we talk about continous optimization.
- If the set X contains real functions, we talk about calculus of variations.

In the previous chapter, we discovered, that learning is a process of finding the right real-valued parameters (or real-valued vector of parameters) for the hypothesis, that minimize the loss function over our dataset. Because the hypothesis is often a complex non-linear function, we use iterative optimization algorithms for finding these parameters.

1.3.1 Gradient descent

Algorithm 1 Gradient descent

Require: $\alpha \in \mathbb{R}$ (stepsize)
Require: $\theta_0 \in \mathbb{R}^n$ (initial parameters vector)
  $k \leftarrow 0$
  repeat
    $k \leftarrow k + 1$
    $\theta^{(k)} = \theta^{(k-1)} + \alpha\Delta\theta^{(k-1)}$
  until stopping criterion is satisfied
  return $\theta^{(k)}$ (resulting parameters vector)

Let's have a random real-valued parameters vector $\theta_0 \in \mathbb{R}^n$. The loss function represents a $n + 1$ dimensional surface. As we would intuitively do in 3 dimensional space, to find the lowest point, we follow the steepest descent path, we can get in every step. We iteratively compute, from the initial parameters vector θ_0, vector $\theta_1, \theta_2, ...$, until some $\theta^* \in \mathbb{R}^n$, for which is the evaulation of the loss function sufficiently small or we stop by any other stopping criterion.

We follow this recurrent procedure to generete a sequence $\theta^{(k)}, k = 0, 1, 2, ...$:

$$\theta^{(k+1)} = \theta^{(k)} + \alpha^{(k)}\Delta\theta^{(k)} \tag{1.8}$$

where $\alpha^{(k)} \in \mathbb{R}$ is called stepsize and $\Delta\theta^{(k)} = -\nabla L(\theta^{(k)}) \in \mathbb{R}^n$ is the gradient of the loss function evaulated in $\theta^{(k)}$. Step size can be a constant or change gradually. The gradient is a multi-variable generalization of the derivative:

$$\nabla L(\theta) = (\frac{\partial L(\theta)}{\partial \theta_1}, \frac{\partial L(\theta)}{\partial \theta_2}, ..., \frac{\partial L(\theta)}{\partial \theta_n}) \tag{1.9}$$

whose components are n partial derivative of the loss function. The loss function has to be differentiable with respect to parameters θ. The gradient points in the direction of the greatest rate of increase of the function, therefore we use the minus to get the opposite direction.

Let's use the hypotheses from the previous chapter to demonstrate a concrete application. Let be the learning rate $\alpha \in \mathbb{R}$ constant and $\theta =$

$(\theta_2, \theta_1, \theta_0) \in \mathbb{R}^3$ random parameters vector. The partial derivatives are with respect to every parameter:

$$\frac{\partial L(\theta)}{\partial \theta_2} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y_i} - h_\theta(\mathbf{x_i}))x_2^2 \tag{1.10}$$

$$\frac{\partial L(\theta)}{\partial \theta_1} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y_i} - h_\theta(\mathbf{x_i}))x_1 \tag{1.11}$$

$$\frac{\partial L(\theta)}{\partial \theta_0} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y_i} - h_\theta(\mathbf{x_i})) \tag{1.12}$$

The rewritten procedure 1.8:

$$\theta_2^{(k+1)} = \theta_2^k + (-\alpha)\frac{\partial L(\theta)}{\partial \theta_2}$$

$$\theta_1^{(k+1)} = \theta_1^k + (-\alpha)\frac{\partial L(\theta)}{\partial \theta_1}$$

$$\theta_0^{(k+1)} = \theta_0^k + (-\alpha)\frac{\partial L(\theta)}{\partial \theta_0}$$

The gradient descent algorithm is also called a batch gradient descent algoritm by machine learning researchers, because the algorithm has to process the whole dataset until the next iteration occurs.

1.3.2 Stochastic gradient descent

Algorithm 2 Stochastic gradient descent

Require: $\alpha \in \mathbb{R}$ (stepsize)
Require: $\theta_0 \in \mathbb{R}^n$ (initial parameters vector)
  $k \leftarrow 0$
  randomly shuffle dataset
  repeat
    for $m \leftarrow 1, 2, ..., N$ **do**
      $k \leftarrow k + 1$
      $\theta^{(k)} = \theta^{(k-1)} + \alpha\Delta\theta^{(k-1)}$
    end for
  until stopping criterion is satisfied
  return $\theta^{(k)}$ (resulting parameters vector)

Stochastic gradient descent or SGD is an extension of the gradient descent algorithm. We saw in equations 1.10, 1.11 and 1.12 that the gradient descent algorithm have to go through the whole dataset to compute $\Delta\theta^{(k-1)}$. This can be computationally expensive for datasets with $N = 3 * 10^8$ elements.

The main difference is the loss function, for which we compute a gradient, is that it does not sum over the whole dataset:

$$L(\mathbf{y_i}, h_\theta(\mathbf{x_i})) = (\mathbf{y_i} - h_\theta(\mathbf{x_i}))^2 \tag{1.13}$$

and the partial derivatives end up looking:

$$\frac{\partial L(\mathbf{y_i}, h_{\theta_i}(\mathbf{x_i}))}{\partial \theta_i} = 2(\mathbf{y_i} - h_\theta(\mathbf{x_i}))x_i \tag{1.14}$$

We are making progress more quickly, looking on just one input-output pair from the dataset.

1.4 Underfitting and overfitting

The central challenge in machine learning is that we must perform well on new, previously unseen inputs — not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called generalization.

Typically, when training a machine learning model, we have access to a training set, we can compute some error measure on the training set called the training error, and we reduce this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the generalization error, also called the test error, to be low as well.

The generalization error is defined as the expected value of the error on a new input. Here the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice.

We typically estimate the generalization error of a machine learning model by measuring its performance on a test set of examples that were collected separately from the training set.

The factors determining how well a machine learning algorithm will perform are its ability to:

- Make the training error small
- Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: underfitting and overfitting. Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large.

We can control whether a model is more likely to overfit or underfit by altering its capacity. Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set.

1.4.1 Hyperparameters and validation sets

Most machine learning algorithms have several settings that we can use to control the behavior of the learning algorithm. These settings are called hyperparameters. The values of hyperparameters are not adapted by the learning algorithm itself.

The setting must be a hyperparameter because it is not appropriate to learn that hyperparameter on the training set. This applies to all hyperparameters that control model capacity. If learned on the training set, such hyperparameters would always choose the maximum possible model capacity, resulting in overfitting. For example, we can always fit the training set better with a higher degree polynomial than we could with a lower degree polynomial.

To solve this problem, we need a validation set of examples that the training algorithm does not observe. The subset of data used to guide the selection of hyperparameters is called the validation set. Typically, one uses about 80% of the training data for training and 20% for validation. Since the validation set is used to train the hyperparameters, the validation set error will underestimate the generalization error, though typically by a

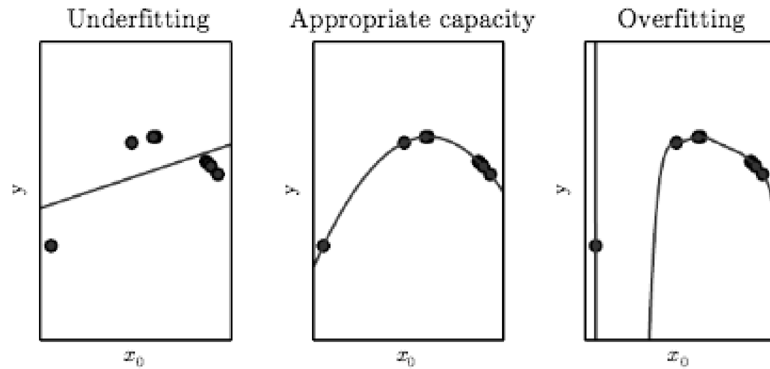


Figure 1.1: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling values and choosing deterministically by evaluating a quadratic function. (Left) A linear function fit to the data suffers from underfitting — it cannot capture the curvature that is present in the data. (Center) A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significiant amount of overfitting or underfitting. (Right) A polynomial of degree 9 fit to the data suffers from overfitting. The solution passes through all of the training points exactly, but we have not been lucky enough for it to extract the correct structure. It now has a deep valley in between two training points that does not appear in the true underlying function. It also increases sharply on the left side of the data, while the true function decreases in this area [4].

smaller amount than the training error. After all hyperparameter optimization is complete, the generalization error may be estimated using the test set.

1.5 Summary

The dataset is usually divided into three distinct subsets:

- A training set serves for learning of the parameters θ described above. This part is called training.
- A validation set is used for determining the best hyperparameters of the hypothesis, these are fixed during training, but can affect the

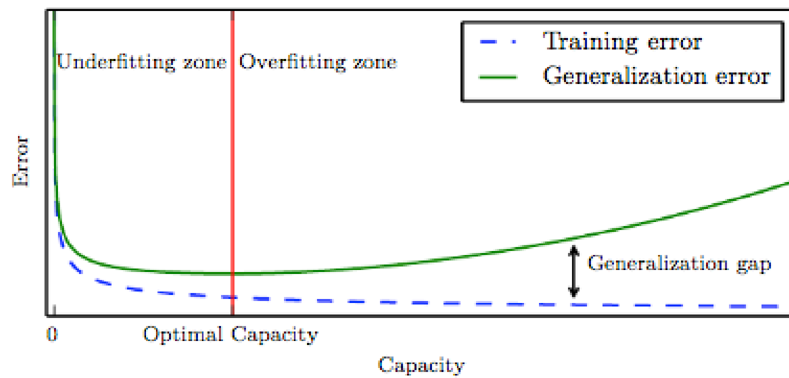


Figure 1.2: Typical relationship between capacity and error. Training and test error behave differently. At the left end of the graph, training error and generalization error are both high. This is the underfitting regime. As we increase capacity, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error, and we enter the overfitting regime, where capacity is too large, above the optimal capacity [4].

final performance, and depends on the hypothesis (for example size of neural network). This part is called tunning.

- The third is a test set. The test set measures a generalization of the hypothesis, how good the hypothesis performs with unseen data.

We can see also dataset divided only to the training and test set. Typical size of the validation and the test set is 20% of the dataset.

The overview concept in summary:

- The dataset contains input-output vector pairs. We divide the dataset into the training set, the validation set and the test set.
- Form a hypothesis, that could describe the relationship between the inputs and the outputs from the dataset.
- Find parameters, that minimize the hypothesis's mean squared error on the training set.
- Tune hyperparameters on the validation set for the best parameters from the previous step. Find such hyperparameters, that minimize the hypothesis's mean squared error on the validation set.

- Test the generalization of the hypothesis on the test set.

Chapter 2

Artificial neural networks

Artificial neural networks are computational models used in computer science trying to model biological neural networks and the central nervous system. They are part of artificial intelligence called statistical machine learning. Other names for artificial neural network include connectionism, parallel distributed processing, neural computations, adaptive networks or collective computation.

From a computational viewpoint, it is a method of representing functions using networks of simple arithmetic computing elements, and methods for learning such representations from examples. These networks represent functions in much the same way that circuits consisting of simple logic gates represent Boolean functions [2].

From a biological viewpoint, it's a mathematical model for the operation of the brain. The simple arithmetic computing elements correspond to neurons - the cells that perform information processing in the brain - and the network as a whole corresponds to a collection of interconnected neurons. For this reason, the networks are called neural networks [2].

2.1 Basics of a biological neuron

The neuron, or nerve cell, is the fundamental functional unit of all nervous system tissue, including the brain, whose principal function is the collection, processing, and dissemination of electrical signals. Each neuron cosists of a cell body, or soma, that contains a cell nucleus. Branching out from the cell body are a number of fibers called dendrites and a single long fiber called the axon. Dendrites branch into a bushy network around the cell, whereas

the axon stretches out for a long distance - usually about a centimeter, and as far as a meter in extreme cases. Eventually, the axon also branches into strands and substrands that connect to the dendrites and cell bodies of another neurons. The connecting junction is called a synapse. Each neuron forms synapses with anywhere from a dozen to a hundred thousand other neurons [2].

Signals are propagated from neuron to neuron by an electrochemical reaction. Chemical transmitter substances are released from the synapses and enter the dendrite, raising or lowering the electrical potential of the cell body. When the potential reaches a threshold, an electrical potential or action potential is sent down the axon. The pulse spreads out along the branches of the axon, eventually reaching synapses and releasing transmitters into the bodies of other cells. Synapses that increase the potential are called excitatory, and those that decrease it are called inhibitory. Connections exhibit plasticity - long-term changes in the strenght of connections in response to the pattern of stimulation. Neurons also form new connections with other neurons, and sometimes entire collections of neurons can migrate from one place to another. These mechanisms are thought to form the basis for learning in the brain [2].

Figure 2.1 depicts an illustration of a biological neuron. Much more detailed and realistic models have been developed, both for neurons and for larger systems in the brain, leading to the modern field of computational neuroscience [3].

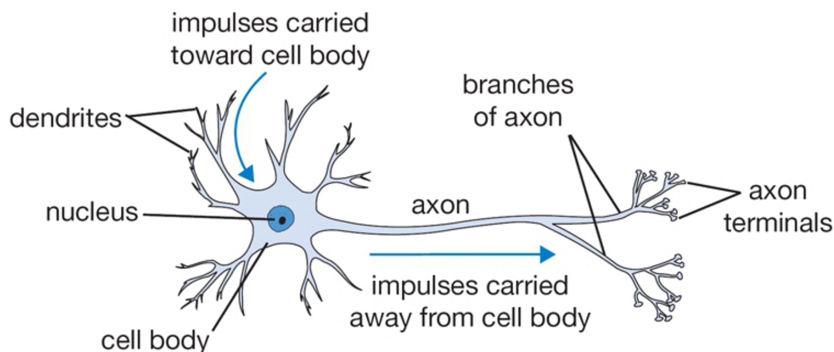


Figure 2.1: An illustration of a biological neuron [6].

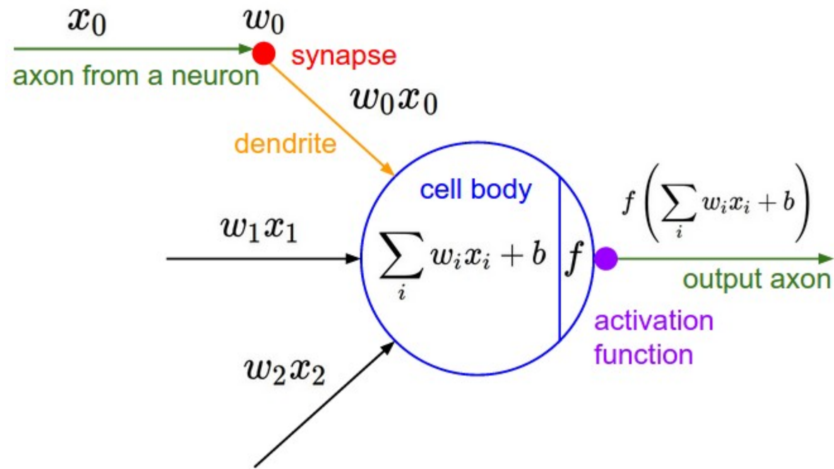


Figure 2.2: An illustration of the simplified mathematical model of a biological neuron [6].

2.2 Mathematical model

Figure 2.2 shows an illustration of the mathematical model of a neuron. Each neuron, also called a node or a unit, receives an input signal from its dendrites and computes a new activation level that it sends along each of its output links. Each input link represents a variable $x_1, x_2, ..., x_n$. Each input link has associated a real-valued weight $w_1, w_2, ..., w_n$. The threshold represents the variable b, another name used for the threshold is a bias, therefore the variable b. The computation is split into two components. First is a linear component in_i, called the input function, that computes the weighted sum of the unit's input values:

$$\text{in}_i = \sum_{j=1}^{n} w_j x_i + b_i \tag{2.1}$$

Second is a nonlinear component f, called the activation function, that transforms the weighted sum into the final values that serves as the unit's activation (output) value a_i:

$$a_i = f(\text{in}_i) = f(\sum_{j=1}^{n} w_j x_i + b_i) \tag{2.2}$$

Different models are obtained by using different mathematical functions for f. Commonly used activation functions are sigmoid, tanh, ReLU, Leaky

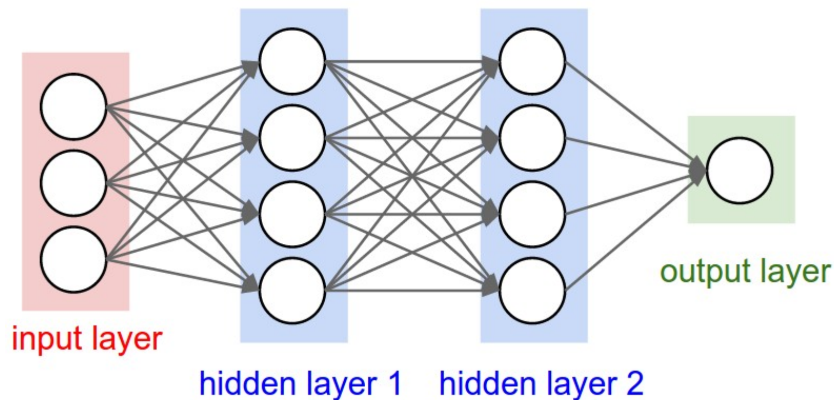


Figure 2.3: An example of a feed-forward neural network with 3 layers [6].

ReLU etc. [6].

The idea is that the synaptic strengths (the weights w_i) are learnable and control the strength of influence, excitory (positive weight) or inhibitory (negative weight), of one neuron on another.

2.2.1 Neural network structures

The most common network structure is acyclic or feed-forward network [2]. It's a directed acyclic graph, where nodes are structured into layers. Each node is linked only to units in the next layer. There are no links between units in the same layer, no links backwards to a previous layer.

Typical network consits of an input layer, an ouput layer and zero or more hidden layers. The name for networks with more than one hidden layer is a multilayer network or now very popular term deep network. The input/output layers can have one or more nodes, therefore the input and the output of the network can be a real-valued vector. Activations of the input layer are input data without aplications of the activation function.

Figure 2.8 shows an example of a feed-forward network with the input layer, 2 hidden layers and the output layer. Total number of layers of this network is 3.

With a fixed structure and fixed activation functions f, the functions representable by a feed-forward network are restricted to have a specific

parameterized structure. A feed-forward network has no internal state other then the weights themselves [2]. Because the activation functions f are non-linear, the whole network represents a complex nonlinear function [1].

If you think of the weights as parameters or coefficients of this function, then learning just becomes a process of tuning the parameters to fit the data in the training set [1]. Another used structures are cyclic or recurrent networks, where are the connections backward and between nodes in the same layer allowed.

2.3 Multilayer feed-forward neural networks

Multilayer feed-forward neural networks, also called feed-forward neural networks or deep feed-forward networks are neural networks with one or more than one hidden layers. The goal of a feed-forward network is to approximate some non-linear function. The non-linearity arises from a choice of a non-linear activation function g. A feed-forward network defines a mapping $\hat{\boldsymbol{y}} = h(\boldsymbol{x}; \boldsymbol{W})$ and learns the value of the parameters $\boldsymbol{W}$, called weights, that result in the best function approximation [4].

The advantage of adding layers is that it enlarges the space of hypotheses that the network can represent [Norvig]. For example, we might have a three function $f^{(1)}, f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $h(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)})(\boldsymbol{x}))$ hypotheses. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is called the second layer, and so on. The overall lenght of the chain gives the depth of the model. From this terminology arises the modern term deep learning.

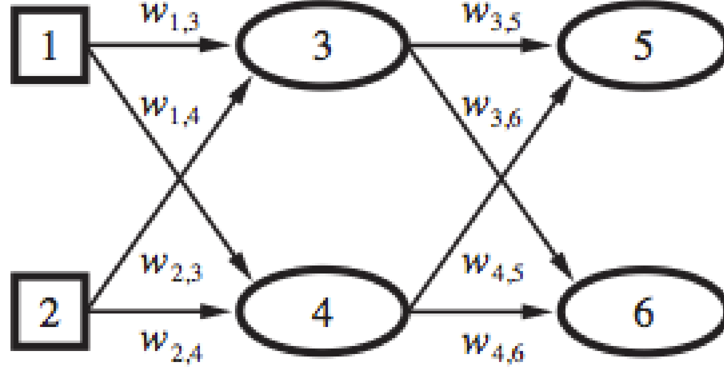


Figure 2.4: An example of a multilayer feed-forward neural network with an input layer with nodes 1 and 2. A hidden layer with nodes 3 and 4. Finally, an output layer with units 5 and 6.

For example consider a simple network in Figure 2.4, given an input vector $\boldsymbol{x} = (x_1, x_2)^T$, the activations of the input units are set to:

$$a_1 = x_1 \tag{2.3}$$

$$a_2 = x_2 \tag{2.4}$$

The hidden units 3 and 4 are given by equations:

$$\begin{aligned} a_3 &= f(w_{1,3}a_1 + w_{2,3}a_2) \\ &= f(w_{1,3}x_1 + w_{2,3}x_2) \end{aligned} \tag{2.5}$$

$$\begin{aligned} a_4 &= f(w_{1,4}a_1 + w_{2,4}a_2) \\ &= f(w_{1,4}x_1 + w_{2,4}x_2) \end{aligned} \tag{2.6}$$

The output units 5 and 6 are given by:

$$\begin{aligned} a_5 &= f(w_{3,5}a_3 + w_{4,5}a_4) = \\ &= f(w_{3,5}f(w_{1,3}a_1 + w_{2,3}a_2) + w_{4,5}f(w_{1,4}a_1 + w_{2,4}a_2)) = \\ &= f(w_{3,5}f(w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}f(w_{1,4}x_1 + w_{2,4}x_2)) \end{aligned} \tag{2.7}$$

$$\begin{aligned} a_6 &= f(w_{3,6}a_3 + w_{4,6}a_4) = \\ &= f(w_{3,6}f(w_{1,3}a_1 + w_{2,3}a_2) + w_{4,6}f(w_{1,4}a_1 + w_{2,4}a_2)) = \\ &= f(w_{3,5}f(w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}f(w_{1,4}x_1 + w_{2,4}x_2)) \end{aligned} \tag{2.8}$$

We can rewrite the equations above in a vector notation. The activation function f has to be a vector function applied elementwise on each member of a vector.

Equations 2.5 and 2.6 in the vector notation:

$$f\left(\begin{pmatrix} w_{1,3} & w_{2,3} \\ w_{1,4} & w_{2,4} \end{pmatrix}\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}\right) = \begin{pmatrix} f(w_{1,3}a_1 + w_{2,3}a_2) \\ f(w_{1,4}a_1 + w_{2,4}a_2) \end{pmatrix} = \begin{pmatrix} a_3 \\ a_4 \end{pmatrix} \tag{2.9}$$

Equations 2.7 and 2.8 in the vector notation:

$$f\left(\begin{pmatrix} w_{3,5} & w_{4,5} \\ w_{3,6} & w_{4,6} \end{pmatrix}\begin{pmatrix} a_3 \\ a_4 \end{pmatrix}\right) = \begin{pmatrix} f(w_{3,5}a_3 + w_{4,5}a_4) \\ f(w_{3,6}a_3 + w_{4,6}a_4) \end{pmatrix} = \begin{pmatrix} a_5 \\ a_6 \end{pmatrix} \tag{2.10}$$

Denote $\boldsymbol{W_2}$ a matrix of weights from the layer 1 to the layer 2, $\boldsymbol{W_3}$ a matrix of weights from the layer 2 to the layer 3 and $\boldsymbol{x}$ a vector of input features:

$$\boldsymbol{W_2} = \begin{pmatrix} w_{3,5} & w_{4,5} \\ w_{3,6} & w_{4,6} \end{pmatrix} \tag{2.11}$$

$$\boldsymbol{W_1} = \begin{pmatrix} w_{1,3} & w_{2,3} \\ w_{1,4} & w_{2,4} \end{pmatrix} \tag{2.12}$$

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \tag{2.13}$$

Finally, we can express a hypotheses from this neural network in the compact form:

$$h(\boldsymbol{x}; \boldsymbol{W}_2, \boldsymbol{W}_1) = f(\boldsymbol{W}_2 f(\boldsymbol{W}_1 \boldsymbol{x})) \tag{2.14}$$

We have the output expressed as a function of the inputs and the weights. As long as we can calculate the derivatives of such expression with respect to the weights, we can use the gradient descent loss-minimalization method to train the network.

2.3.1 Activation functions

Every activation function takes a single number and performs a certain mathematical function.

Sigmoid activation function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.15}$$

It takes a real-valued number x and returns a number between 0 and 1. Very large negative numbers become 0 and large positive numbers become 1. The sigmoid function has seen frequent use historically since it has a nice interpretation as the firing rate of a neuron. In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used.

The tanh non-linearity is shown on the image above on the right, the tanh activation function is defined as:

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{2.16}$$

It squashes a real-valued number to the range -1 and 1. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity.

The Rectified Linear Unit (ReLU) has become very popular in the last few years. It computes the function:

$$f(x) = \max(0, \text{x}) \tag{2.17}$$

There are several pros and cons to using the ReLUs:

- It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form [6].
- Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero [6].
- Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on [6].

In summary, ReLU is now the most recommended activation function.

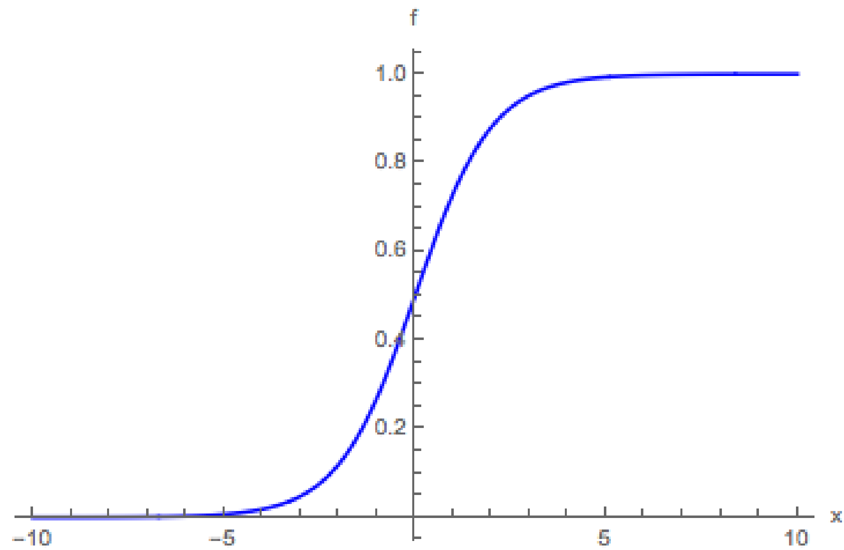


Figure 2.5: The Sigmoid activation function.

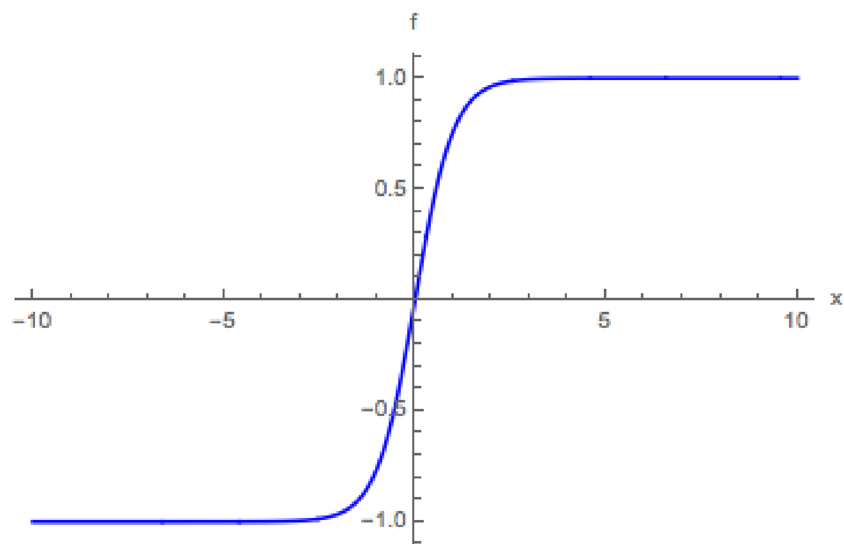


Figure 2.6: The Tanh activation function.

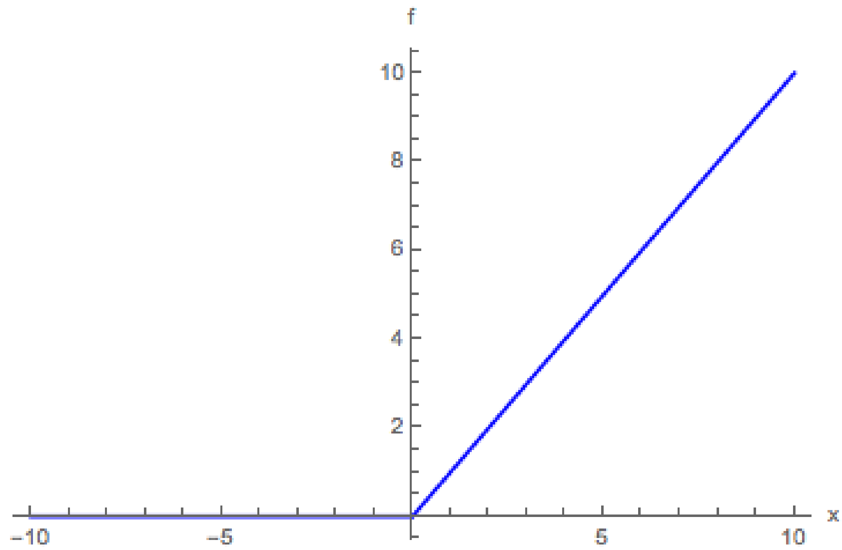


Figure 2.7: The Rectified Linear Unit (ReLU).

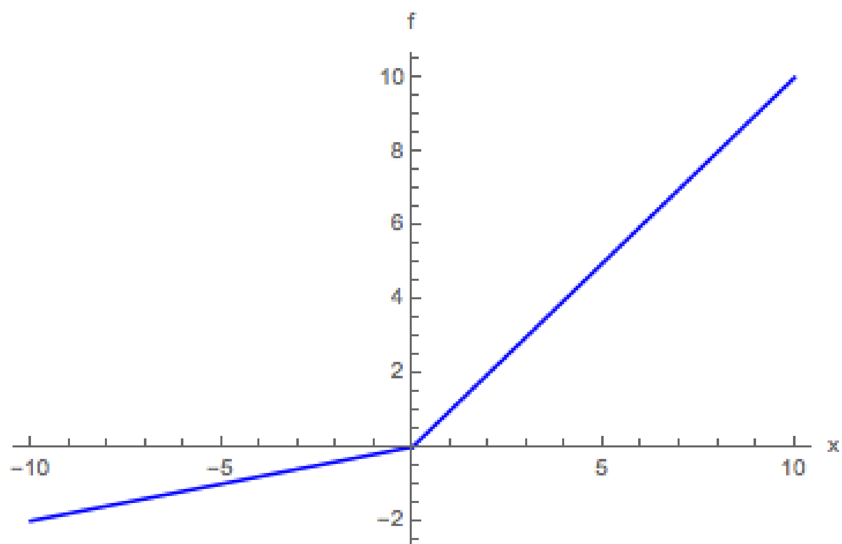


Figure 2.8: The Leaky Rectified Linear Unit (Leaky ReLU) with $\alpha = 0.2$.

2.3.2 The learning algorithm - Backpropagation

The backward propagation of errors, or backpropagation, is a standard method of training artificial neural networks and used in conjunction with an optimization method such as gradient descent [Wiki Backpropagation].

The algorithm works in two stages:

- Given an input vector $\boldsymbol{x}$, compute an output of a network $\boldsymbol{h}$. This stage is also called forward propagation.
- Compare an output from forward propagation with a desired target. Compute an error by a loss function and propagate the error back through a network to update weights.

The major complication comes from the addition of hidden layers to the network. Whereas the error $\boldsymbol{y} - \boldsymbol{h}$ at the output layer is clear, the error at the hidden layers seems mysterious because the training data do not say what value the hidden nodes should have. It turns out that we can back-propagate the error from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient.

At the output layer, we have multiple output units, so let Err_k be the k-th component of the error vector $\boldsymbol{y} - \boldsymbol{h}$ and $\Delta_k = \mathrm{Err}_k \times f'(\mathrm{in}_k)$ be the modified error, so that the update rule becomes:

$$w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times \Delta_k \qquad (2.18)$$

To update the connections between the input units and the hidden units, we need to define a quantity analogous to the error term for output nodes. Here is where we do the error back-propagation. The idea is that hidden node j is responsible for some fraction of the error Δ_k in each of the output nodes to which it connects. Thus, the Δ_k values are divided according to the strength of the connection between the hidden node and the output node and are propagated back to provide the Δ_j values for the hidden layer. The propagation rule for the Δ values is the following:

$$\Delta_j = f'(\mathrm{in}_j) \sum_k w_{j,k} \Delta_k \qquad (2.19)$$

Now the weight-update rule for the weights between the inputs and the hidden layer is essentially identical to the update rule for the output layer:

$$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta_j \tag{2.20}$$

The back-propagation process can be summarized as follows:

- Compute the Δ values for the output units, using the observed error.
- Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
  - Propagate the Δ values back to the previous layer.
  - Update the weights between the two layers.

The squared loss function on a single example is defined as

$$\mathrm{L} = \frac{1}{2}(y_i - a_i)^2 \tag{2.21}$$

where the sum is over the nodes in the output layer. To obtain the gradient with respect to a specific weight $w_{j,i}$ in the output layer, we need only expand out the activation a_i as all other terms in the summation are unaffected by $w_{j,i}$:

$$\begin{aligned}
\frac{\partial \mathrm{L}}{\partial w_{j,i}} &= \frac{\partial}{\partial w_{j,i}} \frac{1}{2}(y_i - a_i)^2 \\
&= -(y_i - a_i)\frac{\partial a_i}{\partial w_{j,i}} \\
&= -(y_i - a_i)\frac{\partial f(\mathrm{in}_i)}{\partial w_{j,i}} \\
&= -(y_i - a_i) f'(\mathrm{in}_i)\frac{\partial \mathrm{in}_i}{\partial w_{j,i}} \\
&= -(y_i - a_i) f'(\mathrm{in}_i)\frac{\partial}{\partial w_{j,i}}\left(\sum_j w_{j,i} a_j\right) \\
&= -(y_i - a_i) f'(\mathrm{in}_i) a_j = -a_j \Delta_i
\end{aligned} \tag{2.22}$$

To obtain the gradient with respect to the $w_{k,j}$ weights connecting the input layer to the hidden layer, we have to keep th entire summation over

i because each output value a_i may be affected by changes in $w_{k,j}$. We also have to expand out the activations a_j:

$$
\begin{aligned}
\frac{\partial \mathrm{L}}{\partial w_{k,j}} &= -\sum_i (y_i - a_i)\frac{\partial a_i}{\partial w_{k,j}} \\
&= -\sum (y_i - a_i)\frac{\partial f(\mathrm{in}_i)}{\partial w_{k,j}} \\
&= -\sum (y_i - a_i) f'(\mathrm{in}_i)\frac{\partial \mathrm{in}_i}{\partial w_{k,j}} \\
&= -\sum \Delta_i \frac{\partial}{\partial w_{k,j}}\left(\sum_j w_{j,i} a_j\right) \\
&= -\sum \Delta_i w_{j,i}\frac{\partial a_j}{\partial w_{k,j}} \\
&= -\sum \Delta_i w_{j,i}\frac{\partial f(in_j)}{\partial w_{k,j}} \\
&= -\sum \Delta_i w_{j,i} f'(in_j)\frac{\partial in_j}{\partial w_{k,j}} \\
&= -\sum \Delta_i w_{j,i} f'(in_j)\frac{\partial}{\partial w_{k,j}}\left(\sum_j w_{k,j} a_k\right) \\
&= -\sum \Delta_i w_{j,i} f'(in_j) a_k = -a_k \Delta_j
\end{aligned}
\tag{2.23}
$$

where Δ_j is defined as before. The process can be continued for networks with more than one hidden layer.

The detailed pseudocode of the algorithm is shown in Algorithm 3.

Algorithm 3 Backpropagation algorithm with gradient descent

```
Require: network
Require: examples
  repeat
    for all weight w_{i,j} in network do
      w_{i,j} ← a small random number
    end for
    for all example (x, y) in examples do
      /* propagate the inputs forward to compute the outputs */
      for all node i in the input layer do
        a_i ← x_i
      end for
      for l ← 2 to L do
        for all node j in layer l do
          in_j ← Σ_i w_{i,j} a_i
          a_j ← g(in_j)
        end for
      end for
      /* propagate deltas backward from output layer to input layer */
      for all node j in the output layer do
        Δ[j] ← g'(in_j)(y_j − a_j)
      end for
      for l ← L − 1 to 1 do
        for all node i in layer l do
          Δ[i] ← g'(in_i) Σ_j w_{i,j} Δ[j]
        end for
      end for
      /* update every weight in network using deltas */
      for all weight w_{i,j} in network do
        w_{i,j} ← w_{i,j} + α a_i Δ[j]
      end for
    end for
  until stopping criterion is satisfied
  return network
```

2.4 Convolutional neural networks

Convolutional neural networks (CNNs) are inspired by the work of Kunihiko Fukushima from 1979, the neural network Neocognitron, the first neural network being able to recognize written text. Fukushima implemented an

observation made by neurophysiologists David H. Hubel and Torsten Wiesel in 1959 on the experiments with cats, where they were making research on visual sensory recognition processing. Hubel and Wiesel inserted a microelectrode into the primary visual cortex (the part of the brain responsible for visual processing) of an anesthetized cat and examined, how neurons react, while they were showing the cat different geometric objects and patterns. They discovered that exist neurons, which activate on a pattern with lines under a particular angle and place on the screen. Other neurons react on the similar object under a different angle and other on the same pattern no matter the location of the object on the screen. They found out that the visual cortex is hierarchical and visual information is at first detects simple patterns like edges etc. and then are recognized more complicated and abstract patterns by the combination of the simple ones.

Yann LeCun created the modern convolutional neural networks in 90's, using Backpropagation algorithm with Gradient descent for training. CNNs are becoming very popular around 2012 by the win in the ImageNet competition by the Geoffrey Hinton's team. They won by a big gap between the standard computer vision approaches and their CNN. They called their network Deep Convolutional Neural Networks because of using a lot of layers with 60 million parameters, and the modern term deep learning is now a synonym for neural networks with a big number of layers. Carefully chosen and trained CNNs are often state-of-the-art and used for image classification (what object is in a picture), localization (where is a location of the object), detection (detecting types and positions of multiple objects in the picture), and segmentation (classifying every pixel in the image). Complex systems using CNNs are self-driving cars or Google Deepmind's program playing Atari games.

In the following subsections, I describe a standard structure of CNNs and the function of convolutional and polling layers. They are based on [5], [6].

2.4.1 The structure of convolutional neural networks

CNNs are also feed-forward networks (there is no cycle) and represent a differentiable function. The difference is that they are assuming an image or grid-like topology as an input. For example, an RGB picture can be imagined as three matrices (one matrice for each color canal) with the number of columns and rows corresponding to the width and the height of

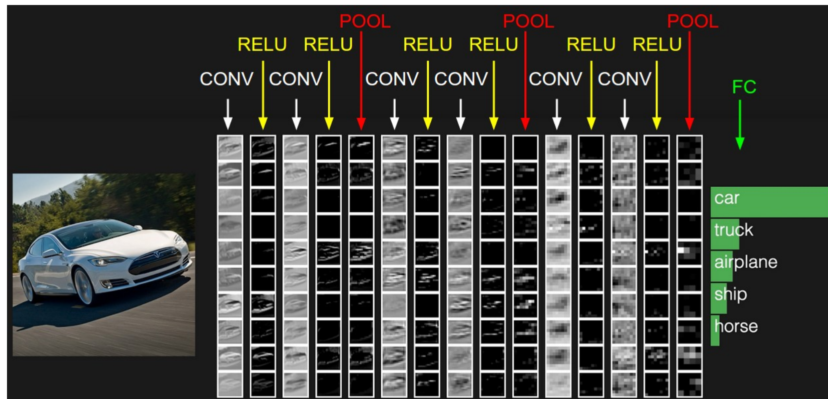


Figure 2.9: An illustration of the standard convolutional neural network [6].

the picture.

A CNN contains:

- a convolutional layer
- an activation layer (the most used is ReLU)
- a pooling layer
- a fully-connected layer (is in the context of CNNs a typical multilayer feed-forward neural network)

Figure 2.9 depicts a standard structure of CNNs. A pooling layer follows a convolutional layer, then follow an activation layer, or more convolutional and pooling layers can follow a couple of times. A fully-connected follows at the end.

The reason is:

- the convolutional layers closer to the input detects simple patterns
- the following convolutional layers identifies more complicated patterns
- the fully-connected layer is a function approximation between found patterns (features) to labels (for classification) or coordinates (for regression or localization)

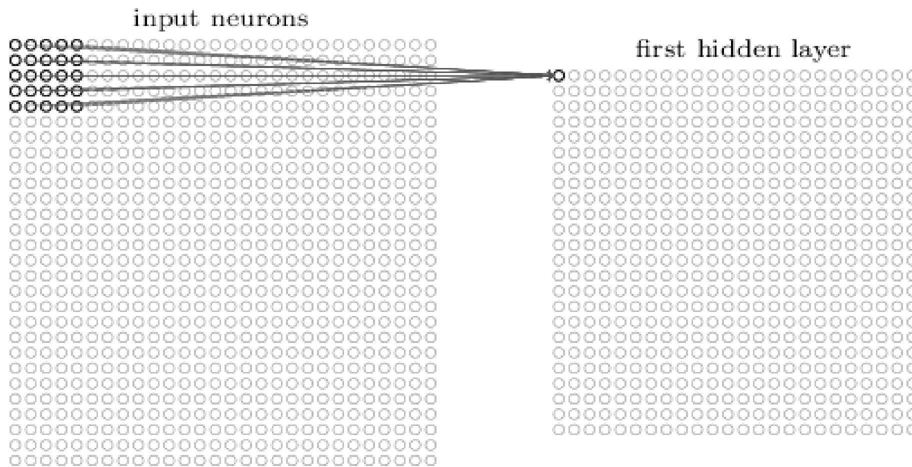


Figure 2.10: Output of the filter [5].

Because a fully-connected layer is a multilayer feed-forward neural network and an activation layer is an element-wise application of an activation function, that I have already described, the next two subsections describe convolutional and pooling layers.

2.4.2 Convolutional layers

Each convolutional layer has K filters (kernels) with a square size F. Figure 2.10 shows, how is a filter applied to the input and outputs a real number, according to:

$$f(b + \sum_{l=0}^{F-1} \sum_{m=0}^{F-1} w_{l,m} a_{j+l,k+m}) \tag{2.24}$$

A filter contains weights and a bias similar to neurons, the number of weights is equal to the size F. Every filter convolves through the input matrice and creates an output matrice (Figure **??**). The output of a filter is called a feature map because every filter detects some feature. Another hyperparameter is a stride S of the convolution. The stride affects the size of the output matrice. A convolutional layer generally outputs a feature map tensor.

The last hyperparameter is zero-padding P. It pads the input volume with zeros around the border. It allows us to spatial control the output.

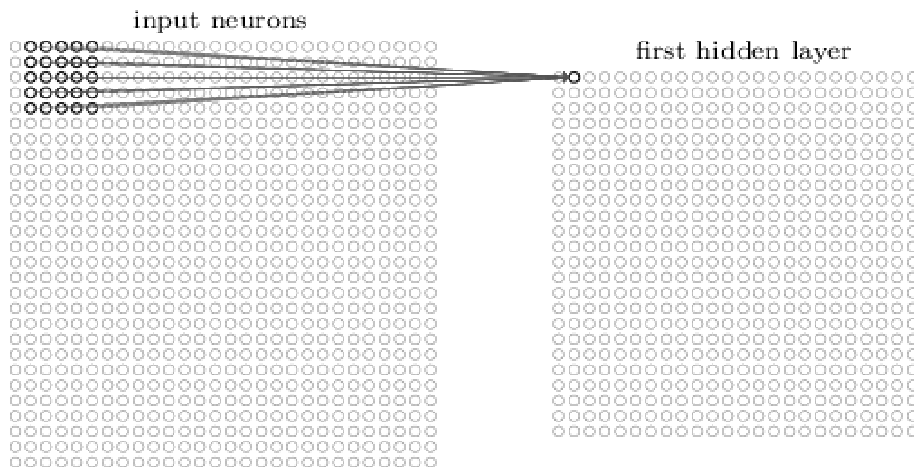


Figure 2.11: The filter convolves with a stride 1 [5].

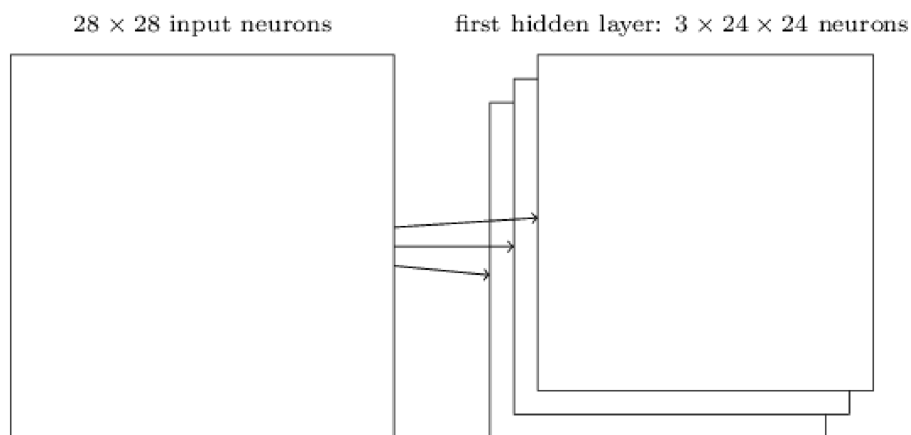


Figure 2.12: A convolutional layer has several filters [5].

For example, we would like the output to have the same size as the input.

In summary, the convolutional layer:

- accepts a volume of size $W_1 \times H_1 \times D_1$
- requires four hyperparameters:
  - number of filters K
  - their spatial extent F
  - the stride S

  - the amount of zero padding P

- produce a volume of size $W_2 \times H_2 \times D_2$ where:

  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$
  - $D_2 = K$

- $2FD_1$ weights per filter, for a total of $(2FD_1)K$ weights and K biases.

- in the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

2.4.3 Pooling layers

Pooling layers typically follow after convolutional layers. They reduce the spatial size to decrease the number of parameters and therefore prevent or control the overfitting. Pooling operation slide across the entire input grid in the same fashion as the convolutional layer, the typical size of the filter is 2×2 or 3×3. The most used operation is taking maximum from the filter at each position. So the max-pooling filter with size 2×2 takes four numbers and outputs the maximum.

The pooling layer:

- Accepts a volume $W_1 \times H_1 \times D_1$.

- Requires two hyperparameters:

  - their spatial extent F
  - the stride S

- Introduces zero parameters since it computes a fixed function of the input.

The pooling layer hasn't any parameters to train. Zero-padding is not used here. The standard settings are $F = 3, S = 2$ (also called overlapping pooling), and more commonly $F = 2, S = 2$.

2.5 Prevention of overfitting

I introduce here several the most used techniques for the prevention of overfitting. Namely L2 regularization, Dropout, Early stopping and Dataset augmentation.

2.5.1 L2 regularization

The most used prevention of overfitting is the L2 regularization. For every weight w_i, we add the term $1/2\lambda w_i^2$ to loss function. $\lambda \in \mathbb{R}$ is a hyperparameter specifying the regularization strength. The constant $1/2$ is used, because the derivative during the computation of the gradient, has the form λw_i. The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors.

2.5.2 Dropout

Dropout is a technique that assigns probability p to every neuron. The probability specifies how likely is going to be a neuron excluded from training in one pass. The p is another hyperparameter. Dropout is very often used since its introduction in 2012.

2.5.3 Early stopping

Early stopping prevents overfitting by stopping the training after some number of epochs without any progression, that means the loss function is no longer progressing towards the minimum. It's often used stopping criterion for optimization algorithms. Early stopping introduces a hyperparameter specifying a patience - the number of epochs, that we tolerate, not making any progression.

2.5.4 Dataset augmentation

The idea is to expand a dataset in the way, that we apply some transformations to an existing dataset. The goal is to get better generalisation. In the case of images, transformations could be for example rotation, making images darker or lighter etc.

Chapter 3

Solution

This chapter presents how I created a dataset. A proposal of several convolutional neural networks, their results of training, a comparison with the reference program provided by my supervisor and discussion.

3.1 Making the dataset

I used the reference program (the current detection algorithm) and video data from real flights to create the dataset. The reference program is written as a node for Robot Operating System (ROS), which is a framework or a platform for the development of robotic systems. ROS's software is made of several nodes communicating between each other with so called messages, that can be saved into Bag files. One can use Bag files to simulate a flight or any other simulation without the actual physical device. My superviser provided me Bag files containing the video data from a camera used to detect the landing platform.

The reference program's detection algorithm takes an image from a Bag file and returns (x, y) coordinates of the middle of the landing platform. I slightly altered the program to extract images from Bag files and their coordinates. I extracted 10 805 images that I stored as black and white images with 150px width and 95px height in the JPG format and the coordinates in the JSON format an array of couples. The detection algorithm sometimes missed the middle, so I hand-corrected the dataset. There are also images without the landing platform or corrupted images, I didn't delete them, but I added a sign to every coordinate - a zero if it is a bad image or a one for a good picture. I classified 9272 images as good and the rest as

not. Finally, I augmented the dataset from good pictures by making every image slightly lighter and darker - this resulted in 120 548 images.

3.2 Implementation

I chose the Python programming language for implementation, because of the Keras framework for neural networks. Keras is a popular neural networks framework using Theano or Tensorflow as a backend. It supports GPU training, that saves a lot of time. I used Cesnet's Metacentrum servers to train my models on GPUs.

3.3 Network architectures

I was inspired by popular architectures as the Oxford's VGG or the ImageNet network by G. Hinton and recommendations from literature. These neural networks are very good for classification of thousand of objects. The problem is, they were not created with an embedded device application in mind, that doesn't have modern GPUs. These networks have a lot of layers and millions of parameters - G. Hinton's network has 60 million parameters. I try to offer rather shallow convolutional neural networks with less learnable parameters than a third of the dataset's volume.

These neural networks are trying to find a mapping between an image and a coordinate, therefore they inputs an image and return two numbers. We adress this problem as regression with two output units. The recommended loss function for regression is the mean squared function, I also measure the mean absolute error. I used Adam optimization algorithm, that is also recommended and according to the research paper [7] has better results than Stochastic gradient descent.

One has to choose a lot of hyperparameters before training. It's not really possible to try all choices. I offer three different architectures that have a lot of hyperparameters chosen based on recommendations from sources [6] and these hyperparametes that I vary for every architecture:

- Dropout $D = 0.3, 0.5, 0.7$
- L2 regularization's $\lambda = 0.01, 1, 100$
- The number of FC layer neurons FCN $= 50, 100$

This results in different 54 models.

3.3.1 Architecture 1

1. convolutional
    - number of filters: 16
    - filter size: 6
    - stride: 3
    - L2 regularization: λ
    - activation: ReLU
2. max-pooling
    - pool size: 3
    - stride: 2
3. convolutional
    - number of filters: 16
    - filter size: 3
    - stride: 2
    - L2 regularization: λ
    - activation: ReLU
4. convolutional
    - number of filters: 16
    - filter size: 3
    - stride: 1
    - L2 regularization: λ
    - activation: ReLU
5. max-pooling
    - pool size: 3
    - stride: 2
6. dropout

- probability: D

7. fully-connected
   - number of neurons: FCN
   - L2 regularization: λ
8. dropout
   - probability: D
9. fully-connected
   - number of neurons: FCN
   - L2 regularization: λ
10. dropout
    - probability: D
11. fully-connected
    - number of neurons: 2

With 12286 parameters for $FCN = 50$ and 23186 parameters for $FCN = 100$.

3.3.2 Architecture 2

1. convolutional
   - number of filters: 16
   - filter size: 6
   - stride: 3
   - L2 regularization: λ
   - activation: ReLU
2. max-pooling
   - pool size: 3
   - stride: 2
3. convolutional

- number of filters: 16
- filter size: 6
- stride: 2
- L2 regularization: λ
- activation: ReLU

4. max-pooling
   - pool size: 3
   - stride: 2
5. dropout
   - probability: D
6. fully-connected
   - number of neurons: FCN
   - L2 regularization: λ
7. dropout
   - probability: D
8. fully-connected
   - number of neurons: FCN
   - L2 regularization: λ
9. dropout
   - probability: D
10. fully-connected
    - number of neurons: 2

With 14766 parameters for $FCN = 50$ and 30466 parameters for $FCN = 100$.

3.3.3 Architecture 3

1. convolutional
    - number of filters: 16
    - filter size: 6
    - stride: 3
    - L2 regularization: λ
    - activation: ReLU
2. max-pooling
    - pool size: 3
    - stride: 2
3. convolutional
    - number of filters: 16
    - filter size: 6
    - stride: 2
    - L2 regularization: λ
    - activation: ReLU
4. max-pooling
    - pool size: 3
    - stride: 2
5. dropout
    - probability: D
6. fully-connected
    - number of neurons: FCN
    - L2 regularization: λ
7. dropout
    - probability: D
8. fully-connected

- number of neurons: 2

With 12216 parameters for $FCN = 50$ and 20366 parameters for $FCN = 100$.

3.4 Measurements

Following tables contain measured loss (mean squared error + L2 regularization), mean squared error (MSE) and mean absolute error (MAE) on the train set, the validation set and the test set. Measurements were made when the weights were saved for the last time before early stopping, these weights are then used for prediction.

Tables 3.1, 3.2 and 3.3 show measurements made on architecture 1. Tables 3.4, 3.5 and 3.6 show measurements made on architecture 2 and tables 3.7, 3.8 and 3.9 show measurements made on architecture 3.

3.4.1 Comparison with the reference program

I used, for the comparison between trained models and the reference program, the dataset before augmentation. I have predictions for coordinations made by the reference program and the hand-corrected coordinations. So, I can compare MSE and MAE between the reference program and trained models.

Tables 3.10, 3.11 and 3.12 show the result of the comparison.

3.5 Discussion

Differences between errors of convolutional networks looks very similar, one tends to think that the generalization of a network is good. It's not really clear for unexperieced people if it is really the case. It looks like that 50 or 100 in fully-connected layer doesn't make a big difference. We can see worse results on models with big L2 regularization. However, we can see that a lot of models are doing better than the reference program. Having overfitting in mind, I would choose the model with the lowest number of parameters and best result for example Architecture 3 ($D = 0.3, \lambda = 0.01, FCN = 50$) or Architecture 1 ($D = 0.3, \lambda = 0.01, FCN = 50$).

Model	Train set loss	Train set MSE	Train set MAE
$D = 0.3, \lambda = 0.01, FCN = 100$	44.9	40.15	4.49
$D = 0.3, \lambda = 0.01, FCN = 50$	51.4	46.8	4.86
$D = 0.3, \lambda = 100, FCN = 100$	231.4	169.3	9.1
$D = 0.3, \lambda = 100, FCN = 50$	220.64	161.86	8.80
$D = 0.3, \lambda = 1, FCN = 100$	81.5	58.9	5.3
$D = 0.3, \lambda = 1, FCN = 50$	83.18	62.02	5.48
$D = 0.5, \lambda = 0.01, FCN = 100$	71.3	65.9	5.9
$D = 0.5, \lambda = 0.01, FCN = 50$	65.48	60.9	5.4
$D = 0.5, \lambda = 100, FCN = 100$	250.02	184.34	9.54
$D = 0.5, \lambda = 100, FCN = 50$	471.25	375.45	14.97
$D = 0.5, \lambda = 1, FCN = 100$	118.19	91.94	6.88
$D = 0.5, \lambda = 1, FCN = 50$	117.20	93.45	6.83
$D = 0.7, \lambda = 0.01, FCN = 100$	91.40	86.07	6.56
$D = 0.7, \lambda = 0.01, FCN = 50$	99.99	95.04	6.81
$D = 0.7, \lambda = 100, FCN = 100$	290.23	240.97	11.13
$D = 0.7, \lambda = 100, FCN = 50$	497.39	410.63	15.80
$D = 0.7, \lambda = 1, FCN = 100$	152.78	124.97	8.07
$D = 0.7, \lambda = 1, FCN = 50$	186.51	159.39	9.23

Table 3.1: Architecture 1 measurement on the training set.

Model	Valid set MSE	Valid set MAE
$D = 0.3, \lambda = 0.01, FCN = 100$	15.6	2.45
$D = 0.3, \lambda = 0.01, FCN = 50$	15.6	2.37
$D = 0.3, \lambda = 100, FCN = 100$	157.6	8.27
$D = 0.3, \lambda = 100, FCN = 50$	149.93	8.24
$D = 0.3, \lambda = 1, FCN = 100$	26	3.02
$D = 0.3, \lambda = 1, FCN = 50$	27.9	3.05
$D = 0.5, \lambda = 0.01, FCN = 100$	23.6	3.06
$D = 0.5, \lambda = 0.01, FCN = 50$	21.8	2.84
$D = 0.5, \lambda = 100, FCN = 100$	173.03	8.97
$D = 0.5, \lambda = 100, FCN = 50$	325.22	13.6
$D = 0.5, \lambda = 1, FCN = 100$	42.82	4.14
$D = 0.5, \lambda = 1, FCN = 50$	41.72	3.95
$D = 0.7, \lambda = 0.01, FCN = 100$	33.44	3.74
$D = 0.7, \lambda = 0.01, FCN = 50$	38.56	4.3
$D = 0.7, \lambda = 100, FCN = 100$	247.94	11.21
$D = 0.7, \lambda = 100, FCN = 50$	327.30	13.88
$D = 0.7, \lambda = 1, FCN = 100$	56.07	4.83
$D = 0.7, \lambda = 1, FCN = 50$	81.39	6.26

Table 3.2: Architecture 1 measurement on the validation set.

Model	Test set MSE	Test set MAE
$D = 0.3, \lambda = 0.01, FCN = 100$	14.3	2.45
$D = 0.3, \lambda = 0.01, FCN = 50$	14.1	2.40
$D = 0.3, \lambda = 100, FCN = 100$	156.2	8.34
$D = 0.3, \lambda = 100, FCN = 50$	148.93	8.3
$D = 0.3, \lambda = 1, FCN = 100$	23.97	3.01
$D = 0.3, \lambda = 1, FCN = 50$	25.96	3.1
$D = 0.5, \lambda = 0.01, FCN = 100$	21.58	3.06
$D = 0.5, \lambda = 0.01, FCN = 50$	20.26	2.88
$D = 0.5, \lambda = 100, FCN = 100$	172.64	9.08
$D = 0.5, \lambda = 100, FCN = 50$	327.83	14.06
$D = 0.5, \lambda = 1, FCN = 100$	36.74	3.83
$D = 0.5, \lambda = 1, FCN = 50$	39.01	3.96
$D = 0.7, \lambda = 0.01, FCN = 100$	31.20	3.77
$D = 0.7, \lambda = 0.01, FCN = 50$	36.81	4.33
$D = 0.7, \lambda = 100, FCN = 100$	251.84	11.47
$D = 0.7, \lambda = 100, FCN = 50$	330.31	14.1
$D = 0.7, \lambda = 1, FCN = 100$	53.90	4.88
$D = 0.7, \lambda = 1, FCN = 50$	81.39	6.40

Table 3.3: Architecture 1 measurement on the test set.

Model	Train set loss	Train set MSE	Train set MAE
$D = 0.3, \lambda = 0.01, FCN = 100$	55.2	50.6	5.2
$D = 0.3, \lambda = 0.01, FCN = 50$	55.2	50.58	5.5
$D = 0.3, \lambda = 100, FCN = 100$	428.89	325.25	13.7
$D = 0.3, \lambda = 100, FCN = 50$	381.8	314.99	13.47
$D = 0.3, \lambda = 1, FCN = 100$	97.38	62.16	5.5
$D = 0.3, \lambda = 1, FCN = 50$	87.3	60.7	5.5
$D = 0.5, \lambda = 0.01, FCN = 100$	80.9	74.5	6.3
$D = 0.5, \lambda = 0.01, FCN = 50$	100.4	95.1	7.2
$D = 0.5, \lambda = 100, FCN = 100$	605.35	400.5	15.5
$D = 0.5, \lambda = 100, FCN = 50$	423.1	340.6	14.1
$D = 0.5, \lambda = 1, FCN = 100$	138.7	97.2	7.2
$D = 0.5, \lambda = 1, FCN = 50$	152.6	117.6	7.9
$D = 0.7, \lambda = 0.01, FCN = 100$	105.3	98.7	7.2
$D = 0.7, \lambda = 0.01, FCN = 50$	148.96	143.9	8.8
$D = 0.7, \lambda = 100, FCN = 100$	455.3	353.2	14.5
$D = 0.7, \lambda = 100, FCN = 50$	349	336	14
$D = 0.7, \lambda = 1, FCN = 100$	171.7	133.9	8.5
$D = 0.7, \lambda = 1, FCN = 50$	355.1	329.97	14.3

Table 3.4: Architecture 2 measurement on the training set.

Model	Valid set MSE	Valid set MAE
$D = 0.3, \lambda = 0.01, FCN = 100$	11.65	1.94
$D = 0.3, \lambda = 0.01, FCN = 50$	11.65	1.94
$D = 0.3, \lambda = 100, FCN = 100$	322.3	13.7
$D = 0.3, \lambda = 100, FCN = 50$	321.4	12.74
$D = 0.3, \lambda = 1, FCN = 100$	28.98	3.2
$D = 0.3, \lambda = 1, FCN = 50$	21.9	2.7
$D = 0.5, \lambda = 0.01, FCN = 100$	19	2.59
$D = 0.5, \lambda = 0.01, FCN = 50$	21.8	2.95
$D = 0.5, \lambda = 100, FCN = 100$	326.98	13.93
$D = 0.5, \lambda = 100, FCN = 50$	327.6	13.9
$D = 0.5, \lambda = 1, FCN = 100$	36.4	3.8
$D = 0.5, \lambda = 1, FCN = 50$	46.8	4.4
$D = 0.7, \lambda = 0.01, FCN = 100$	34.7	3.8
$D = 0.7, \lambda = 0.01, FCN = 50$	58.37	5.27
$D = 0.7, \lambda = 100, FCN = 100$	323.8	13.8
$D = 0.7, \lambda = 100, FCN = 50$	331.6	14
$D = 0.7, \lambda = 1, FCN = 100$	58.4	5.12
$D = 0.7, \lambda = 1, FCN = 50$	246.4	11.92

Table 3.5: Architecture 2 measurement on the validation set.

Model	Test set MSE	Test set MAE
$D = 0.3, \lambda = 0.01, FCN = 100$	9.95	1.93
$D = 0.3, \lambda = 0.01, FCN = 50$	9.97	1.94
$D = 0.3, \lambda = 100, FCN = 100$	324	13.94
$D = 0.3, \lambda = 100, FCN = 50$	323.3	13.94
$D = 0.3, \lambda = 1, FCN = 100$	26.7	3.2
$D = 0.3, \lambda = 1, FCN = 50$	20	2.74
$D = 0.5, \lambda = 0.01, FCN = 100$	16.6	2.57
$D = 0.5, \lambda = 0.01, FCN = 50$	19.7	2.96
$D = 0.5, \lambda = 100, FCN = 100$	329.3	14.1
$D = 0.5, \lambda = 100, FCN = 50$	330.2	14.1
$D = 0.5, \lambda = 1, FCN = 100$	34.2	3.8
$D = 0.5, \lambda = 1, FCN = 50$	44.7	4.5
$D = 0.7, \lambda = 0.01, FCN = 100$	32.6	3.8
$D = 0.7, \lambda = 0.01, FCN = 50$	56.43	5.3
$D = 0.7, \lambda = 100, FCN = 100$	326	14
$D = 0.7, \lambda = 100, FCN = 50$	334	14.2
$D = 0.7, \lambda = 1, FCN = 100$	56.9	5.2
$D = 0.7, \lambda = 1, FCN = 50$	245.7	11.99

Table 3.6: Architecture 2 measurement on the test set.

Model	Train set loss	Train set MSE	Train set MAE
$D = 0.3, \lambda = 0.01, FCN = 100$	40.08	35.65	4.25
$D = 0.3, \lambda = 0.01, FCN = 50$	53.39	49.06	5.07
$D = 0.3, \lambda = 100, FCN = 100$	565.04	363.16	14.68
$D = 0.3, \lambda = 100, FCN = 50$	609.06	382.89	15.15
$D = 0.3, \lambda = 1, FCN = 100$	79.06	49.93	4.93
$D = 0.3, \lambda = 1, FCN = 50$	84.85	57.97	5.35
$D = 0.5, \lambda = 0.01, FCN = 100$	60.21	55.35	5.36
$D = 0.5, \lambda = 0.01, FCN = 50$	100.76	95.94	7.20
$D = 0.5, \lambda = 100, FCN = 100$	385.09	327.87	13.83
$D = 0.5, \lambda = 100, FCN = 50$	384.90	327.76	13.84
$D = 0.5, \lambda = 1, FCN = 100$	119.46	85.45	6.70
$D = 0.5, \lambda = 1, FCN = 50$	119.96	89.68	6.76
$D = 0.7, \lambda = 0.01, FCN = 100$	125.86	119.74	8.08
$D = 0.7, \lambda = 0.01, FCN = 50$	152.24	147.46	8.98
$D = 0.7, \lambda = 100, FCN = 100$	368.01	339.34	14.14
$D = 0.7, \lambda = 100, FCN = 50$	331.55	325.39	13.78
$D = 0.7, \lambda = 1, FCN = 100$	158.45	123.03	8.14
$D = 0.7, \lambda = 1, FCN = 50$	176.05	145.31	8.68

Table 3.7: Architecture 3 measurement on the training set.

Model	Valid set MSE	Valid set MAE
$D = 0.3, \lambda = 0.01, FCN = 100$	9.3	1.7
$D = 0.3, \lambda = 0.01, FCN = 50$	16.85	2.69
$D = 0.3, \lambda = 100, FCN = 100$	327.07	13.99
$D = 0.3, \lambda = 100, FCN = 50$	333.00	14.17
$D = 0.3, \lambda = 1, FCN = 100$	21.69	2.76
$D = 0.3, \lambda = 1, FCN = 50$	20.69	2.60
$D = 0.5, \lambda = 0.01, FCN = 100$	17.94	2.60
$D = 0.5, \lambda = 0.01, FCN = 50$	25.49	3.37
$D = 0.5, \lambda = 100, FCN = 100$	329.56	14.00
$D = 0.5, \lambda = 100, FCN = 50$	327.83	13.96
$D = 0.5, \lambda = 1, FCN = 100$	26.97	26.98
$D = 0.5, \lambda = 1, FCN = 50$	36.82	3.86
$D = 0.7, \lambda = 0.01, FCN = 100$	35.93	3.97
$D = 0.7, \lambda = 0.01, FCN = 50$	56.61	5.22
$D = 0.7, \lambda = 100, FCN = 100$	329.17	13.99
$D = 0.7, \lambda = 100, FCN = 50$	328.14	13.97
$D = 0.7, \lambda = 1, FCN = 100$	53.42	4.88
$D = 0.7, \lambda = 1, FCN = 50$	75.61	5.84

Table 3.8: Architecture 3 measurement on the validation set.

Model	Test set MSE	Test set MAE
$D = 0.3, \lambda = 0.01, FCN = 100$	7.74	1.69
$D = 0.3, \lambda = 0.01, FCN = 50$	15.05	2.68
$D = 0.3, \lambda = 100, FCN = 100$	329.10	14.19
$D = 0.3, \lambda = 100, FCN = 50$	335.00	14.36
$D = 0.3, \lambda = 1, FCN = 100$	19.72	2.75
$D = 0.3, \lambda = 1, FCN = 50$	18.22	2.59
$D = 0.5, \lambda = 0.01, FCN = 100$	16.36	2.60
$D = 0.5, \lambda = 0.01, FCN = 50$	23.50	3.38
$D = 0.5, \lambda = 100, FCN = 100$	331.45	14.19
$D = 0.5, \lambda = 100, FCN = 50$	329.96	14.16
$D = 0.5, \lambda = 1, FCN = 100$	24.70	3.25
$D = 0.5, \lambda = 1, FCN = 50$	35.18	3.92
$D = 0.7, \lambda = 0.01, FCN = 100$	33.94	3.99
$D = 0.7, \lambda = 0.01, FCN = 50$	54.75	5.28
$D = 0.7, \lambda = 100, FCN = 100$	331.32	14.18
$D = 0.7, \lambda = 100, FCN = 50$	330.48	14.17
$D = 0.7, \lambda = 1, FCN = 100$	52.30	4.95
$D = 0.7, \lambda = 1, FCN = 50$	75.02	5.92

Table 3.9: Architecture 3 measurement on the test set.

Model	MSE	MAE
Current detection algorithm	612.89	9.97
$D = 0.3, \lambda = 0.01, FCN = 100$	8.3	1.6
$D = 0.3, \lambda = 0.01, FCN = 50$	10.8	1.86
$D = 0.3, \lambda = 100, FCN = 100$	317.46	13.67
$D = 0.3, \lambda = 100, FCN = 50$	317.69	13.64
$D = 0.3, \lambda = 1, FCN = 100$	18.35	2.37
$D = 0.3, \lambda = 1, FCN = 50$	20.81	2.53
$D = 0.5, \lambda = 0.01, FCN = 100$	14.52	2.27
$D = 0.5, \lambda = 0.01, FCN = 50$	22.57	2.99
$D = 0.5, \lambda = 100, FCN = 100$	316.57	13.57
$D = 0.5, \lambda = 100, FCN = 50$	315.82	13.57
$D = 0.5, \lambda = 1, FCN = 100$	26.07	3.08
$D = 0.5, \lambda = 1, FCN = 50$	34.70	3.50
$D = 0.7, \lambda = 0.01, FCN = 100$	34.49	3.83
$D = 0.7, \lambda = 0.01, FCN = 50$	55.06	5.03
$D = 0.7, \lambda = 100, FCN = 100$	316.47	13.57
$D = 0.7, \lambda = 100, FCN = 50$	313.82	13.57
$D = 0.7, \lambda = 1, FCN = 100$	51.65	4.7
$D = 0.7, \lambda = 1, FCN = 50$	75.6	5.7

Table 3.10: Comparison between architecture 1 and the reference program.

Model	MSE	MAE
Current detection algorithm	612.89	9.97
$D = 0.3, \lambda = 0.01, FCN = 100$	8.61	1.66
$D = 0.3, \lambda = 0.01, FCN = 50$	11.39	1.92
$D = 0.3, \lambda = 100, FCN = 100$	309.56	12.32
$D = 0.3, \lambda = 100, FCN = 50$	309.84	13.37
$D = 0.3, \lambda = 1, FCN = 100$	24.32	2.83
$D = 0.3, \lambda = 1, FCN = 50$	21.8	2.67
$D = 0.5, \lambda = 0.01, FCN = 100$	17.23	2.61
$D = 0.5, \lambda = 0.01, FCN = 50$	20.96	2.82
$D = 0.5, \lambda = 100, FCN = 100$	313.54	13.46
$D = 0.5, \lambda = 100, FCN = 50$	313.61	13.46
$D = 0.5, \lambda = 1, FCN = 100$	34.86	3.55
$D = 0.5, \lambda = 1, FCN = 50$	36.35	3.65
$D = 0.7, \lambda = 0.01, FCN = 100$	34.01	3.72
$D = 0.7, \lambda = 0.01, FCN = 50$	56.53	5.13
$D = 0.7, \lambda = 100, FCN = 100$	311.56	13.41
$D = 0.7, \lambda = 100, FCN = 50$	316.27	13.57
$D = 0.7, \lambda = 1, FCN = 100$	56.62	4.91
$D = 0.7, \lambda = 1, FCN = 50$	233.27	11.53

Table 3.11: Comparison between architecture 2 and the reference program.

Model	MSE	MAE
Current detection algorithm	612.89	9.97
$D = 0.3, \lambda = 0.01, FCN = 100$	11.86	2.09
$D = 0.3, \lambda = 0.01, FCN = 50$	14.99	2.33
$D = 0.3, \lambda = 100, FCN = 100$	148.09	8.00
$D = 0.3, \lambda = 100, FCN = 50$	143.33	8.08
$D = 0.3, \lambda = 1, FCN = 100$	26.44	2.99
$D = 0.3, \lambda = 1, FCN = 50$	28.64	3.03
$D = 0.5, \lambda = 0.01, FCN = 100$	18.51	2.62
$D = 0.5, \lambda = 0.01, FCN = 50$	20.47	2.73
$D = 0.5, \lambda = 100, FCN = 100$	165.2	8.75
$D = 0.5, \lambda = 100, FCN = 50$	312.77	13.44
$D = 0.5, \lambda = 1, FCN = 100$	37.94	3.66
$D = 0.5, \lambda = 1, FCN = 50$	31.92	3.53
$D = 0.7, \lambda = 0.01, FCN = 100$	43.72	4.01
$D = 0.7, \lambda = 0.01, FCN = 50$	37.72	4.17
$D = 0.7, \lambda = 100, FCN = 100$	228.35	10.39
$D = 0.7, \lambda = 100, FCN = 50$	314.31	13.46
$D = 0.7, \lambda = 1, FCN = 100$	57.76	4.82
$D = 0.7, \lambda = 1, FCN = 50$	77.91	5.95

Table 3.12: Comparison between architecture 3 and the reference program.

Conclusion

I introduced some basic concepts from machine learning. I explained multilayer and convolutional neural networks. I created a dataset from drone's camera using the current detection algorithm from my supervisor. I successfully designed convolution neural network that has better results than the current detection algorithm on the dataset.

Future work could be to understand neural networks better. Try to use these systems on a real drone to better verify its current capability.

Bibliography

[1] RUSSELL, Stuart J. and NORVIG, Peter. *Artificial Intelligence: A Modern Approach*. 1st ed. Prentice Hall, c1995. ISBN 0-13-103805-2.

[2] RUSSELL, Stuart J. and NORVIG, Peter. *Artificial Intelligence: A Modern Approach*. 2nd ed. Prentice Hall, c2002. ISBN 978-0137903955.

[3] RUSSELL, Stuart J. and NORVIG, Peter. *Artificial Intelligence: A Modern Approach*. 3rd ed. Pearson, c2009. ISBN 978-0136042594.

[4] GOODFELLOW, Ian et al. *Deep Learning*. MIT Press, c2016. ISBN 978-0262035613. URL: <http://www.deeplearningbook.com/>.

[5] NIELSEN, Michael A. *Neural Networks and Deep Learning* [online]. URL: <http://www.neuralnetworksanddeeplearning.com/>.

[6] KARPATHY, Andrej et al. *Convolutional Neural Networks for Visual Recognition* [online]. URL: <http://www.cs231n.stanford.edu/>.

[7] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. CoRR. abs/1412.6980. 2014. URL: <http://arxiv.org/abs/1412.6980>

[8] NG, Andrew. *Machine Learning by Stanford on iTunes* [online]. Available in: <https://itunes.apple.com/us/course/machine-learning/id495053006>.

[9] KRIESEL, David. *A brief introduction to Neural Networks* [online]. Available in: <http://www.dkriesel.com/_media/science/neuronalenetze-en-zeta2-2col-dkrieselcom.pdf>.

[10] Murphy, Kevin P. *Machine Learning: A Probabilistic Perspective (Adaptive Computation and Machine Learning series).* The MIT Press, 24.8.2012. 1104 s. ISBN-10 0262018020.

[11] HASTIE, Trevor, Robert TIBSHIRANI a J. H. FRIEDMAN. The elements of statistical learning: data mining, inference, and prediction. 2nd ed. New York: Springer, c2009. Springer series in statistics. ISBN 03-878-4857-6.

[12] WERNER, Tomáš. *Optimalizace* [online] URL: `https://cw.fel.cvut.cz/wiki/_media/courses/a4b33opt/opt.pdf`

Appendix A

Acronyms

ANN Artificial neural network

CNN Convolutional neural network

ROS Robot operating system

MSE Mean squared error

MAE Mean absolute error

FC Fully-connected layer

Appendix B

Contents of enclosed USB

```
readme.txt ........................ the file with USB contents description
├── datasets ....................... the directory with datasets
├── trained models ................. the directory with trained neural networks
├── src ............................ the directory of source codes
│   ├── script ..................... the directory with Python and Bash scripts
│   └── thesis ..................... the directory of LaTeX source codes of the thesis
└── text ........................... the thesis text directory
    └── thesis.pdf ................. the thesis text in PDF format
```