Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Balltracking for Robocup Soccer using Deep Neural Networks

Bachelor thesis
im Arbeitsbereich Knowledge Technology, WTM
Dr. Cornelius Weber

Department Informatik
MIN-Fakultät
Universität Hamburg

vorgelegt von
Daniel Speck
am
27.06.2016

Gutachter: Dr. Cornelius Weber
Pablo Barros

Daniel Speck
Matrikelnummer: 632 13 17
Ottersbekallee 10a
20255 Hamburg

Abstract

The latest change of rules for the RoboCup Soccer in the humanoid league allows the ball's surface to consist of up to 50% of any color or pattern, whereas at least 50% have to be white. This makes ball localization an even more important and challenging task, because multi-color balls have changing color histograms and patterns, depending on the ball's current orientation and movement. To handle this newly introduced difficulty this thesis proposes a neural architecture – a convolutional neural network (CNN) – to localize the ball in various scenes. CNNs are capable of learning invariances in images and are used in several image recognition tasks. In this thesis CNNs are designed to locate a ball by training two output layers, one for the x- and one for the y-coordinates. The CNNs get trained with normal distributions fitted around the ball's center. This makes it possible for the network to not just locate the ball's position but also to provide an estimation of the noise. The whole image is processed by the architecture in full size, no sliding-window approach is used. Afterwards the CNN's output gets filtered by a recurrent neural network (RNN), which additionally tries to predict future positions of the ball.

Keywords: robocup, convolutional neural network, deep learning, tensorflow, ball detection, ball localization, noise, filtering, recurrent neural network

Zusammenfassung

Aufgrund der jngsten Regelnderung der RoboCup Soccer humanoid league muss die Oberflche eines Balles nur noch zu 50% wei sein, whrend die restlichen 50% der Oberflche jegliche Farbe oder Muster aufweisen drfen. Deshalb ist die Balllokalisierung wichtiger, aber auch herausfordernder denn je. Mehrfarbige Blle besitzen, in Abhngigkeit von ihrer Orientierung und Bewegung, sich stndig ndernde Farbhistogramme und -muster. Diese neu entstandene Komplexitt motivierte diese Arbeit, in der eine neuronale Architektur in Form eines gefalteten neuronalen Netzes (Convolutional Neural Network, CNN) zur Balllokalisierung vorgestellt. Gefaltete Neuronale Netze sind in der Lage Invarianten zu lernen und werden deshalb vor allem fr Bilderkennungsaufgaben genutzt. In dieser Arbeit werden gefaltete neuronale Netze genutzt, um den Ball zu lokalisieren, indem zwei verschiedene Ausgabeschichten trainiert werden, welche die x- und y-Koordinaten des Balles modellieren. Um diese Koordinaten wird jeweils eine Normalverteilung gelegt. Infolgedessen ist das Netz nicht nur in der Lage den Ball zu lokalisieren, sondern auch das Rauschen im aktuellen Prozess abzuschtzen. Die Architektur verarbeitet des gesamte Bild, eine "sliding-window" Methode wird nicht genutzt. Anschlieend wird die Ausgabe des CNNs von einem rekurrenten neuronalen Netz (RNN) gefiltert und geglttet. Zeitgleich versucht das RNN zuknftige Ballpositionen abzuschtzen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

RoboCup consists of various leagues such as: RoboCup@Work, -@Home, -Rescue and -Soccer. The @Work league concentrates on the development and competition of robots that are intended to cooperate with humans to perform difficult tasks in the industrial field. As the name suggests @Home focuses on robots assisting humans in their own homes. RoboCup-Rescue league introduces robots that are developed to respond in cases of emergencies and catastrophes. The aim is to maximize security for rescue personnel and the victims chance of survival.
RoboCup Soccer was the first league and contains the most subleagues. In the championship two teams of robots compete with each other in a game of soccer. The RoboCup Soccer humanoid kid-size league is the intended field of application for this thesis and the developed ball localization and conceptional ball prediction. In the aforementioned league the rules state that the robots have to look humanoid, measure 40-90cm and operate fully autonomous. They are however allowed to communicate with each over through wifi. The goal for the RoboCup humanoid soccer league is to beat the reigning champion of soccer by 2050, or at least being competitive. This goal is challenging and opens several research questions, because the robots have to look, walk, kick and plan like humans to compete, which would be a huge step for robots.
Today robots are mostly present in manufactures, managing very simple, specific tasks. One example is the factory of BMW in Spartanburg, Germany. They use robots for pressing on door seals on cars. Despite the fact, that such robots can heavily increase a factory's efficiency and reduce repetitious work of the staff, their tasks are straightforward and always the same or at least very similar. These robots adhere closely to a hard coded algorithm for finishing one step at a time. Therefore they can be used at an assembly line. However, in the real world most tasks include complex problems that require dynamic handling. Thus, these straightforward architectures cannot be used for complex, real world scenarios. Considering the stationary robots in an assembly line for example: equipping them with a movement system would grant them mobility and more possibilities, but it would also require them to navigate to different places at certain times. In this scenario their working tasks are the same but since they have to navigate now, they have to avoid obstacles, plan routes, and so forth. Architectures that are able to fulfill

these and comparable tasks need to react intelligently to their environment, because of the vast amount of possibilities in real world scenarios. Even navigation tasks are highly complex due to possible obstacles that change over time and vary in shapes, colors, et cetera. Neural solutions are an interesting step forward, since they can be trained and learn invariants rather then follow a static sequence of steps like standard algorithms.
In RoboCup humanoid soccer the already mentioned standard computer vision algorithms often utilize color and edge information for ball tracking [14, 5, 25, 10], because such algorithms are rather easy to implement and do not require lots of test data. One common solutions in RoboCup is to search for shapes, examine whether they are round or not, and try to find the center if the shape is similar to a ball's shape [5]. Most of the standard algorithms are computationally cheap and deliver usable results. However, since there is no intelligent decision whatsoever, these algorithms detect false positives quite often. It is hard to determine whether a ball was found or something that just appears to have a similar shape or colors. Moreover, due to the rise in complexity of the tasks of RoboCup [29], motivation for new solutions is growing. This thesis is motivated by the recent ball specification changes: the ball's color in RoboCup humanoid soccer was completely orange until 2014, but from 2015 onwards the specifications have changed to a ball with at least 50% white color leaving the rest of the ball open for any color combinations [29]. These rule changes led to the risen complexity and heavily affected the results of several standard algorithms that have been used. The new ball specifications introduce color distributions that change in dependency of the camera's direction, additionally they are dependent on the ball's orientation. The contrasting and changing appearance of the new ball especially differs if the ball is moving, Figure 1.1 shows these scenarios.
In sum the hypothesis of this thesis is that a neural architecture should outperform standard algorithms currently in use and produce reasonable results. It is expected that a neural architecture is able to learn ball invariants, in order to get a better rate of true positives and reduce misclassification. The idea is that a deep neural architecture should be able to learn the mentioned invariants, if supplied with a sufficiently large dataset. Additionally the neural architecture should work on camera output, without heavy preprocessing, to leave as much raw information as possible. This strategy not only covers more variation, but also a more distinctive feature set. Deep networks enable to learn visual features for comparable tasks [23]. This thesis presents a convolutional neural network (CNN) to localize the ball in real world scenes, which outputs a distribution instead of single coordinates. This information can be utilized for determining the noise in the input signal. One additional hypothesis is that utilizing such distributions should lead to better results for training the CNN, because the amount of possible solutions delivering a low training error is reduced. Afterwards the CNN's output gets processed by a recurrent neural network (RNN) to reduce the noise and predict the next, subsequent output. The thesis presents a concept of this RNN.

Figure 1.1: One of the balls currently used in Robocup. The white color of the ball is almost identical compared to the color of the goal posts and the endlines. In the center image one can see the overemphasized red channel; this happens occasionally with our robot's camera and the lighting in our laboratory. The left and center images show non-moving balls with a distance of 0.5 meters to the robot. The right image shows a moving ball with a distance of 1.0 meter to the robot.

1.1 Research Topic & Objectives

Our working hypothesis is that neural architectures perform better than previous, non-adaptive algorithms, like standard vision algorithms. A neural solution should be more stable and robust when supplied with enough training data (against different ball orientations, speeds, trajectories, color distributions, ...) for ball detection as well as tracking. The current most stable solution of our team distinguishes between finding the ball in a picture and actually *tracking* it. The tracking is mainly done via linear regression and weighted means ignoring any information from the ball detection. Creating a neural architecture being able of ball detection and tracking delivers a better performance by learning invariants from the supplied training data.
The neural architecture should be able to reliably detect the ball in the robot's vision. Therefore a high rate of true positives is one of the main goals. Additionally the solution must keep track of the ball with a low noise on it's predicted trajectory. Temporal information changing over time should be considered in the tracking/filtering stage.

Chapter 2

Ball Localization & Tracking

2.1 Related Work

2.1.1 Localization

CNNs have often and successfully been used for object classification tasks [20, 13, 31, 22] while in the last years also several CNNs for object localization have been proposed and showed good results [27, 6, 24, 7, 30]. However, e.g. sliding-window approaches are not optimal for ball localization in RoboCup soccer. Due to the large change in size of the features representing the ball when it is moving away or towards a robot the ball would cover several segments of a sliding-window solution quite often. Therefore we decided to let our architecture always classify the full image and manipulate the output, not the input, by feeding a probability distribution over the width and height of the image as the teaching signal. Apart from that, deep learning architectures like convolutional neural networks seem to be novel in the RoboCup humanoid league for object localization tasks.

2.1.2 Tracking

Even simple convolutional neural networks with just a few layers can achieve reasonable results and be used to develop a robust representation for visual tracking. Zhang et al. have developed a CNN where a k-means algorithm is applied for getting normalized patches of a target region. The filters are applied to subsequent frames for developing feature maps that react to similarities between the frames [33]. Another method for tracking and also noise filtering is a neural Kalman filter. Szirtes et al. created a neural architecture similar to a Kalman filter, which aims for predicting dynamical systems [32]. Trained with Hebbian learning rules their network is able to calculate a recursive prediction error.

2.2 RoboCup Ball Localization

While neural solutions are not very common in RoboCup ball localization, there are some innovative strategies to localize a ball. For example, a color based segmentation algorithm that classifies & separates objects by its color with a minimized dependence to illumination was developed by Bandlow et. al. [1]. Recognizing objects by colors is dependent on the lighting and many other variables. To eliminate this dependence on colors, parametric curves are fitted to shapes inside a frame. This procedure is called *Contracting Curve Density* and does not rely on trained color data. Instead, it gathers statistics about image information on-line and separates neighboring regions iteratively fitting curves around the shapes [11].

Chapter 3

Basics

3.1 Neural Networks

For approximating certain functions in machine learning disciplines artificial neural networks are used frequently. Modeling a learning process by optimizing a gradient descent in the very basic case, they can adapt to invariants in training data. While standard algorithms often fail at tasks that require some kind of intelligent behavior, neural networks can succeed due to their dynamical character. One easy example for this is recognizing handwritten digits. Since every person has its own style of writing, handwritten digits vary in size, shape, color and over variables. For standard algorithms it is hard to determine which features are really important, in order to characterize a digit. Therefore most standard algorithms would fail in classifying digits. Neural approaches however can identify patterns in the training data, invariants that many samples share. By doing so, the network adapts to certain characteristics of digits and is therefore able to classify handwritten digits, even when the network is supplied input data that it has never seen before. Neural networks need a sufficiently large database of training samples, otherwise the network may run into overfitting.
Figure 3.1 illustrates the standard architecture for fully-connected neural networks (also called multi-layer perceptrons). The connections between two layers can be vectorized by using weight matrices to model these connections. Hence, each layer's activation can be described as a dot product between the output of the preceding layer and the layer's weight matrix, transformed by an activation function. With this procedure all computations can be vectorized efficiently. Let n_j be the amount of neurons in layer j. The dimension of a weight matrix W_j is two, with n_{j-1} rows and n_j columns.

3.1.1 Supervised Learning

Training a model can be realized with different strategies. In this thesis we solely used "supervised learning", which is supposed infer a model's parameters by learning from a training data set. Each sample in the training data set consists of some input data vector and a corresponding label. A sample's label is a data vector

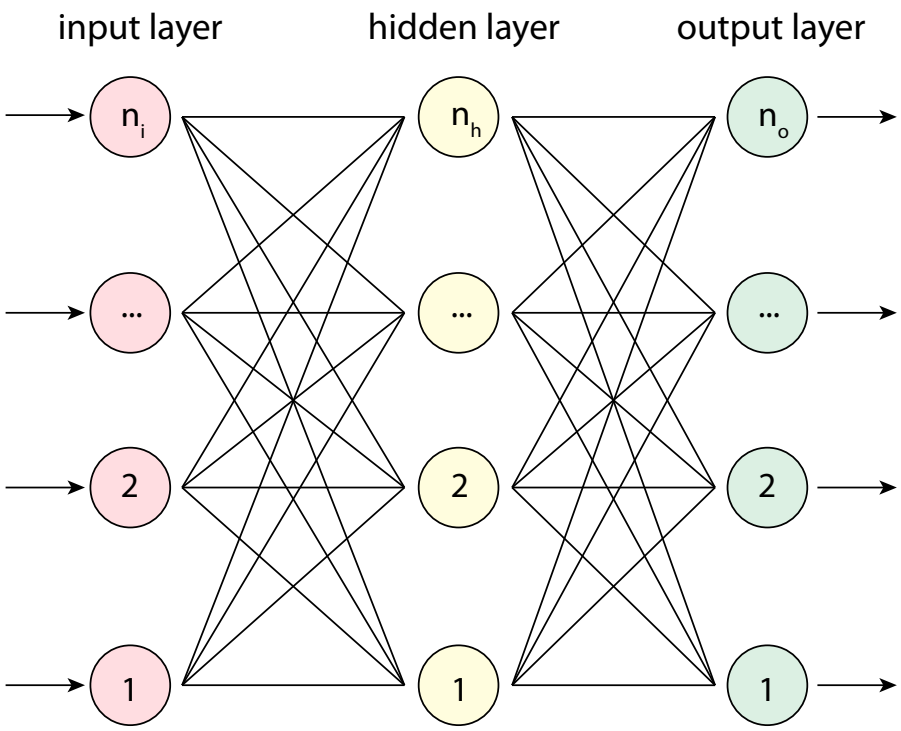

Figure 3.1: Figure of an example for an artifical neural network. The network has three layers, an input layer, a *hidden layer*, and an *output layer*. Each layer generates an output vector of a dimension equal to the size of the layer (n_i, n_h, respectively n_o number of neurons)

.

that contains "perfect", precalculated data for the model's output. This output is used to determine the error for one training sample by applying a cost function, e.g. a squared-error, a cross entropy-error et cetera. By adjusting the model's parameters to minimize the error the model is able to learn from the training data set.

3.1.2 Feedforward Activation

The activation a of a single neuron i in layer j is defined by:

$$a_{j,i} = \sum_{k=1}^{n_{j-1}} \left(w_{j,i,k} * h_{j-1,k}\right) + b_{j,i} \; . \tag{3.1}$$

where $w_{j,i,k}$ is the weight of the connection between the k-th neuron of the preceding layer $j-1$ and the i-th neuron of layer j. $h_{j-1,k}$ is the output of the k-th neuron of layer $j-1$, while $b_{j,i}$ is the bias of the i-th neuron in layer j. This leads to the following vectorization for one feedforward activation:

$$a_j = h_{j-1} \times W_j + b_j \; . \tag{3.2}$$

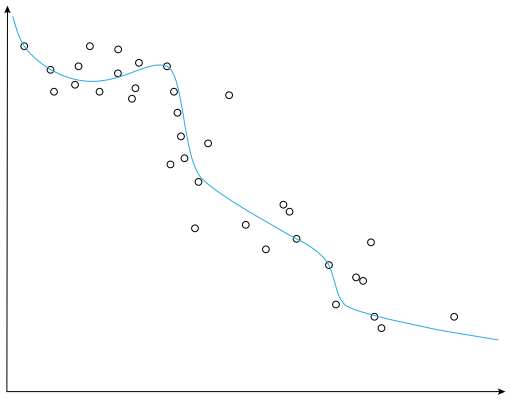

Figure 3.2: Figure of an example for regression. The model calculated a function (blue line) that fits to the dataset with low overfitting.

.

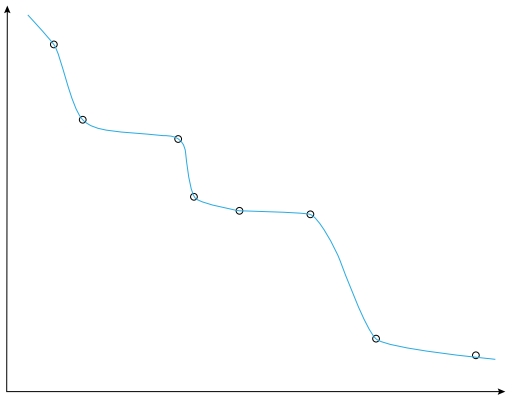

Figure 3.3: Another Figure of an example of a regression problem. Here the model clearly overfits (blue line) to the training samples, therefore the generalization of the problem domain will be low, thus the model will not have learned invariants.

.

where a_j is a n_j-dimensional vector covering all activations for every neuron of layer j. W_j is two-dimensional weight matrix with n_{j-1} rows and n_j columns. The output h_j of layer j is a n_j-dimensional vector defined by:

$$h_j = \tau(a_j) \; . \tag{3.3}$$

where τ is an *activation function* (also called *transfer function*). This is a non-linear function, e.g. the hyperbolic tangent, which is needed to model non-linearities.

3.1.3 Overfitting

In machine learning many models are supposed to learn a generalization of a problem's domain by processing training data adjust its parameters according to some error evaluation. While it is important for the model to learn from the training data, the goal is to extract invariants from the training data that belong to the problem's domain. If the model is only trained to get a high prediction accuracy on the training data, the process would have no purpose, because the model's parameters would be "overfitted" then. This means the model lacks of generalization and just performs well on a specific training data set. This behavior for a regression problem is illustrated in Figure 3.3. Another model, where the training data is processed while keeping the overfitting low to maximize the generalization of the model, can be seen in Figure 3.2. One simple technique to prevent overfitting is to simply stop the training process once the test error starts to increase again. With each training step (*iteration*) both, the error of the training data as well as the error of the test data should decrease. An illustration of this can be seen in

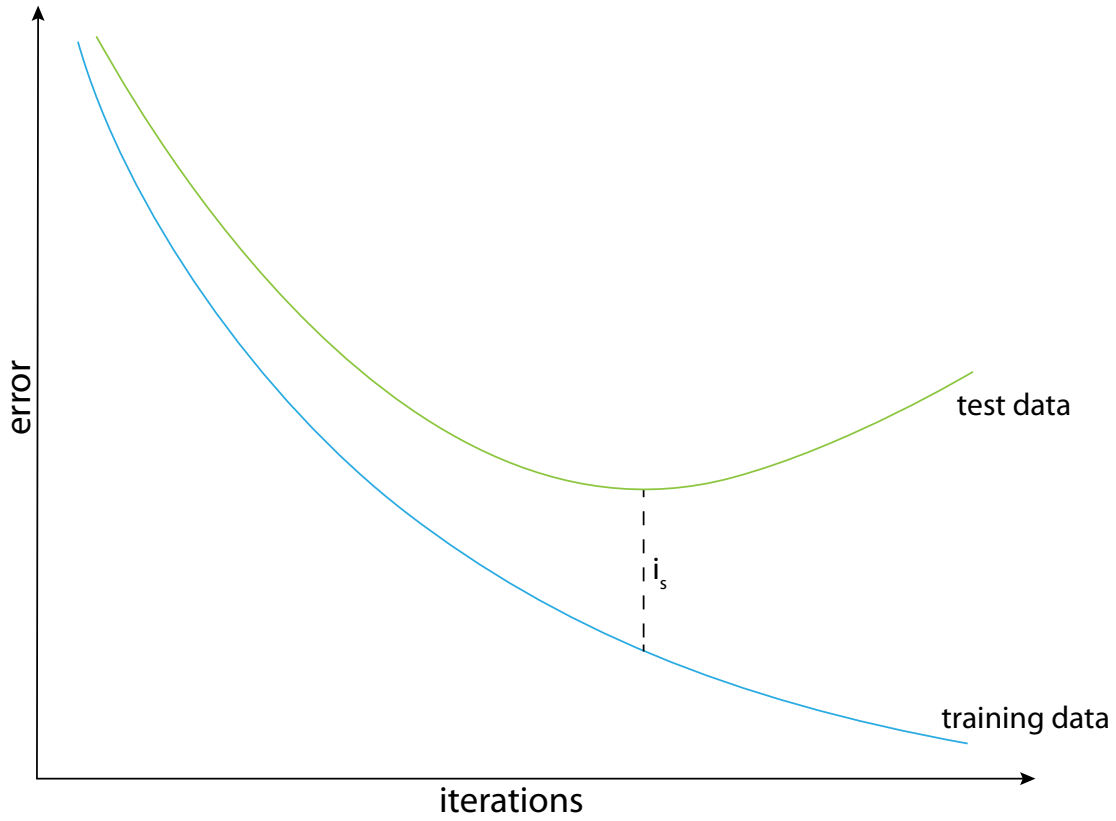

Figure 3.4: This Figure illustrates a simplified example of when to stop training in order to prevent overfitting. The test data error decreases until i_s, thus the training should be stopped and the model's parameters saved at iteration i_s. All training iterations after i_s just decrease the training error, but increase the test error.

Figure 3.4. At iteration i_s the previous training step, iteration $i_s - 1$, was the last one to decrease the test error. All iterations after i_s increase the test error, which lowers the model's robustness. Therefore i_s is the best choice for stopping the training process and saving the model's parameters.

3.1.4 Gradient Descent Optimization

Backpropagation is a popular method used to train neural networks for decades [12]. It is used in combination with an optimization algorithm, like gradient descent. In this thesis all neural networks are trained using backpropagation. The gradient is a generalization of the normal derivative of functions with several variables. If all partial derivations of some function $f(x_1, \ldots, x_n)$ exist and are real-valued, the gradient is a vector that contains all of these partial derivations as components. The gradient is often symbolized with ∇ (nabla operator). Hence,

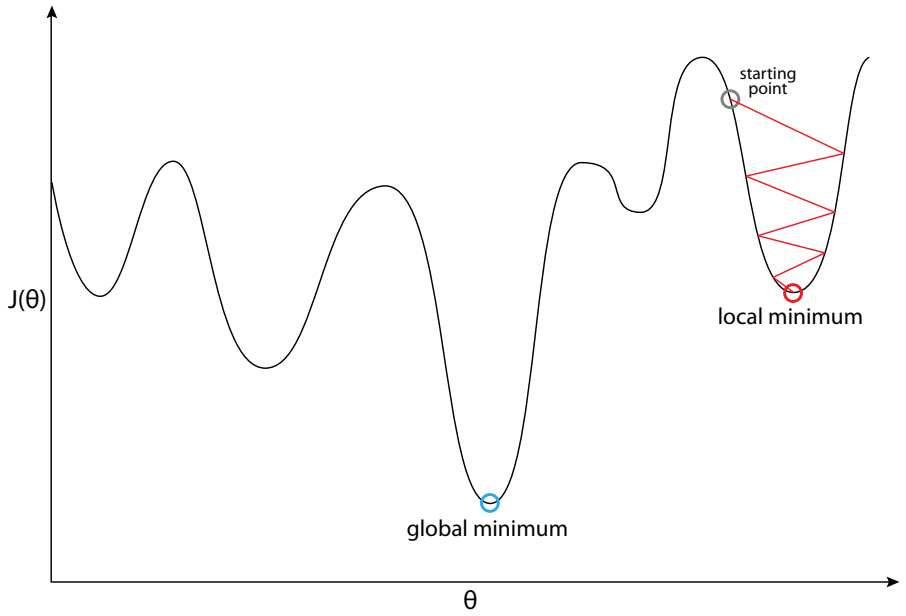

Figure 3.5: This Figure illustrates the gradient descent algorithm in a two-dimensional space. With a high learning rate (α), the algorithm makes big steps, which may result in "jumping" back and forth, like the red line demonstrates. The learning rate and the starting point affect the chance of falling into a local minimum.

the gradient (*grad*) of some function $f(x_1, \ldots, x_n)$ is defined by:

$$grad(f) = \nabla f = \frac{\partial f}{\partial x_1}\vec{e_1} + \cdots + \frac{\partial f}{\partial x_n}\vec{e_n} = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)^T . \tag{3.4}$$

where $\frac{\partial f}{\partial x_i}$ are the partial derivatives and $\vec{e_i}$ the unit vectors for $i = 1, \ldots, n$. The T in the vector of equation 3.4 symbolizes that the vector is transposed.
In terms of neural networks the model's current error is calculated by a *cost function* (in some literature also referred to as *(total) error function*). Let E be that cost function. In order to reduce the error, the model's parameters need to be adjusted, which are are the weights $w_{j,i,k}$ (see equation 3.1). Backpropagation starts at the output layer and propagates backwards through the network. Therefore one feedforward activation (see section 3.1.2) has to be processed at first. Afterwards the error at the output layer is determined. Considering a *mean squared error* (MSE), this is simply:

$$E_{\text{output}} = \sum_i \frac{1}{2}\left(\hat{t} - h_{j,i}\right)^2 . \tag{3.5}$$

where $\hat{t}$ is the target value (teaching signal), $h_{j,i}$ the output of neuron i in layer j. Obviously j has to be the index of the output layer here. The goal of the backpropagation is to minimize this error by adjusting the weights. Since the gradient (vector) points in the direction of the greatest increase of the function values, due to the partial derivatives, the negative gradient points in the direction

of greatest decrease. With the negative gradient we compute the delta (Δ), which is used to update the weights $w_{j,i,k}$. For the *output layer*, this procedure is defined by:

$$\Delta w_{j,i,k} = -\alpha * \frac{\partial E}{\partial w_{j,i,k}} \overset{chainrule}{=} -\alpha * \frac{\partial E}{\partial h_{j,i}} * \frac{\partial h_{j,i}}{\partial a_{j,i}} * \frac{\partial a_{j,i}}{\partial w_{j,i,k}} \ . \tag{3.6}$$

where α is the *learning rate*, a scalar which simply scales the step-size, E is the cost function, $h_{j,i}$ the output of neuron i in layer j, and $a_{j,i}$ the activation of neuron i in layer j. To calculate the delta for the hidden layer, the equation has to be changed to address the fact that each hidden neuron's output influences the error for every succeeding layer's neurons. Thus, the delta for weights $w_{j,i,k}$ of *hidden layers* are calculated by:

$$\Delta w_{j,i,k} = -\alpha * \frac{\partial E}{\partial w_{j,i,k}} \overset{chainrule}{=} -\alpha * \underbrace{\frac{\partial E}{\partial h_{j,i}}}_{\sum_{n=1}^{N_{j+1}} \frac{\partial E_n}{\partial h_{j,i}}} * \frac{\partial h_{j,i}}{\partial a_{j,i}} * \frac{\partial a_{j,i}}{\partial w_{j,i,k}} \ . \tag{3.7}$$

where N_{j+1} is the number of neurons in the succeeding layer, E_n the error for neuron n of layer $j+1$, and $h_{j,i}$ the output of neuron i of layer j. With the use of equation 3.6 and 3.7 the weights $w_{j,i,k}$ can be updated:

$$w_{j,i,k}^{\text{new}} = w_{j,i,k} + \Delta w_{j,i,k} \tag{3.8}$$

where $w_{j,i,k}^{\text{new}}$ is the new weight. While the mean squared error is easy to compute, other cost functions might supply better results. In this thesis the *cross entropy error* is used, which is defined by:

$$E_{\text{output}} = -\frac{1}{N} \sum_{i=1}^{N} \Big(\hat{t} \log(h_{j,i}) + (1 - \hat{t}) \log(1 - h_{j,i}) \Big) \ . \tag{3.9}$$

where N is the number of neurons in the layer, $\hat{t}$ the target value (teaching signal), and $h_{j,i}$ the output of neuron i in layer j. Again, j has to be the index of the output layer.

3.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) have connections between units that form a directed cycle. The recurrent cycle is achieved by copying previous output data of a layer and feed this data back as input to the corresponding layer in addition to the regular feed-forward input at the next/subsequent time step. This procedure sets up an internal state that allows the recurrent neural network to consider/recognize temporal patterns in input data. Thus, a RNN can find invariants in time, which a normal feed-forward neural network can not learn. Therefore RNNs are a reasonable choice for any task that includes sequences or other input data that may contain patterns over time. An example of such a network can be seen in Figure 3.6.

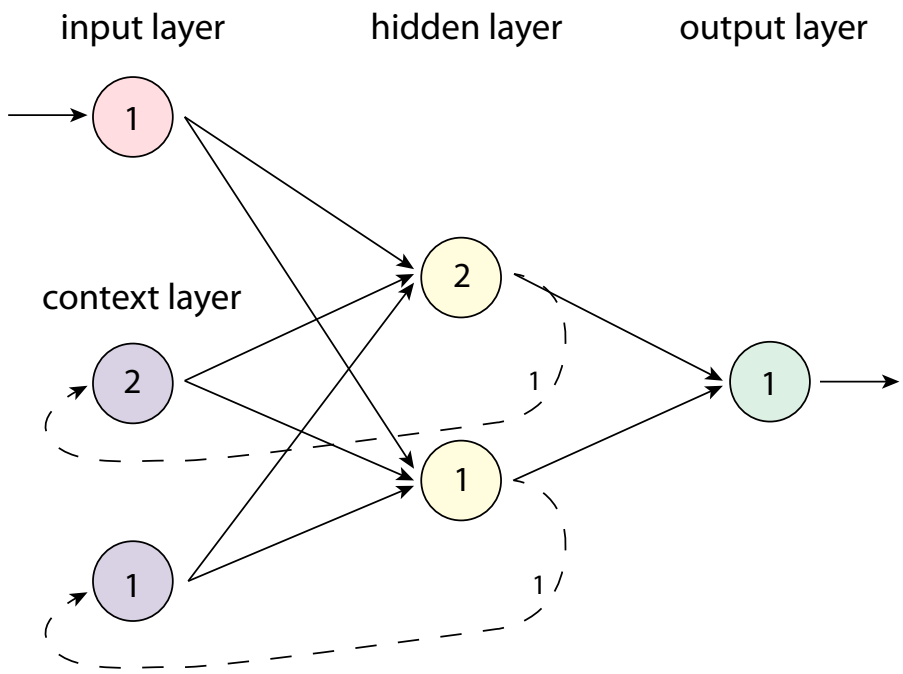

Figure 3.6: This Figure shows a (simple) recurrent neural network (also referred to as *Elman network*). The input layer has one neuron, the hidden layer two neurons, and the output layer one neuron. The recurrence is realized with the context layer, which always has the same size as the hidden layer it is applied on. To feed the context layer the output of the hidden layer is propagated as input to the context layer (with a weight of one, ensuring to really "copy" the data). This is illustrated by the dashed lines in the Figure.

3.3 Convolutional Neural Networks

Standard neural networks with just one hidden layer struggle in learning a vast variety of invariants, therefore they are limited at object detection tasks. The high resolution of real world images covers a high amount of features, but in order to properly extract invariants out of such images, more complex models are needed. Additionally fully-connected networks quickly consume too much memory, because of their exponential growth in connections. Being able to detect and distinguish objects in images is a challenging task. With the rise of GPU-computing deeper models became popular, because the computational power is now sufficiently large to train such models in hours or days. One example of a network with tens of layers, thousands of neurons, and millions of connections is Google's large scale video classification [16]. Complex structures such as this enable the possibility of computing complex problems that occur with tasks like image classification in real world images. To accomplish those tasks, deep neural networks are used in combination with GPU-optimized code that is considerably faster than CPU-optimized code and therefore allows larger, more accurate architectures [4]. For deep learning purposes (classic/fully-connected) multilayer perceptrons consume a decent amount of

resources for proper training, when they are designed to solve the involved tasks. This is because the amount of neurons – specifically the weights – increases rapidly with the network's size. For example, an MLP with four layers, an input layer with 25 neurons, two hidden layers with 10 and 5 neurons, and an output layer with 10 neurons for classifying images with a size of 5x5 pixels into 10 different classes would consist of $25*10+10*5+5*10 = 350$ weights/connections. In comparison to real world images these 5x5 images are tiny, even scaling this example up to images with a dimension of 50x50 it'll be nowhere near real world images but would already result in an architecture of 2500 (input), 1000 (hidden), 500 (hidden), 10 (output) neurons[1] and have $2500*1000+1000*500+500*10 = 3,005,000$ weights/connections. The human brain contains about 150 trillion ($1.5*10^{14}$) synapses [28]. Furthermore, as features in images capturing real world scenes are distributed in specific patterns (they cover spatially local correlation, such as shapes), it is not necessary to have every pixel's information being processed by one neuron. In fact, for most cases the results would be better, if the pixel's information was preprocessed, for instance by edge detection filters. A fully-connected layer of neurons is not an optimal solution for this task.
Convolutional neural networks (CNNs) are inspired by biology. Rather than connecting each pixels information directly with a neuron to process its information the CNN filters the information in the first layers [20]. This procedure is resembling the processes happening when a biological eye receives stimuli. The receptive field[2] has a great amount of photoreceptor cells[3] collecting information and passing the received information on to distinctly fewer retinal ganglion cells[4]. This process maps some features and lowers the input dimensionality as well as distinguishes the information to separate "channels" that are then transfered to the corresponding neurons, in order to process features such as color, motion, shapes and more separately [18].

3.3.1 Convolution

Figure 3.7 shows an application example of a convolutional layer. The image on the left side is hreyscaled and shows an anmimal, the one on the right side is the same image after being filtered with the kernel matrix in equation 3.10.

$$K_M = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} . \tag{3.10}$$

The kernel matrix K_M (see equation 3.10) was applied to each pixel in the left picture separately resulting in the information of the respective pixels in the right

[1] Assuming, for example, the task is classifying real hand-written digits
[2] http://en.wikipedia.org/wiki/Receptive_field
[3] http://en.wikipedia.org/wiki/Photoreceptor_cell
[4] http://en.wikipedia.org/wiki/Retinal_ganglion_cell

Figure 3.7: Left side: original image (greyscaled), right side: edge-detection kernel processed image. Original image by Michael Plotke, 28th of January, 2013. Open creative commons license. `http://upload.wikimedia.org/wikipedia/commons/5/50/Vd-Orig.png` and `http://upload.wikimedia.org/wikipedia/commons/6/6d/Vd-Edge3.png`

picture. With a wide variety of different kernels several different features can be extracted from an image. To extract as much information as possible most CNNs use an assortment of various kernels. The following formula is used to calculate the processed pixels:

$$I_{out}(x,y) = \sum_{a=1}^{3}\sum_{b=1}^{3} I_{in}(x+a-c_x, y+b-c_y) * K_M(a,b) \ . \tag{3.11}$$

$$I_{in} = \begin{bmatrix} 46 & 42 & 50 \\ 44 & 65 & 56 \\ 41 & 52 & 58 \end{bmatrix} \ . \tag{3.12}$$

$$I_{out}(2,2) = 131 \ .$$

where c_x represents the X-coordinate whereas c_y stands for the Y-coordinate of the input image. The filter is used for edge detection thus highlighting the edges and omitting everything else in the processed picture as a result (e.g. right picture of Figure 3.7). An example for a combination of a possible in- and output of the image in figure 3.7 is shown in Equation (3.12). The input I_{in} are 3x3 pixels of the original picture. Applying kernel filter K_M of equation (1) to the input image generates the output I_out for the center pixel (x=2, y=2). Repeating this procedure to all pixels of the input image creates the output image step by step.

3.3.2 Pooling

CNNs utilize a concept of non-linear down-sampling called pooling. While there are different approaches for pooling, the most used one is max-pooling. Pooling

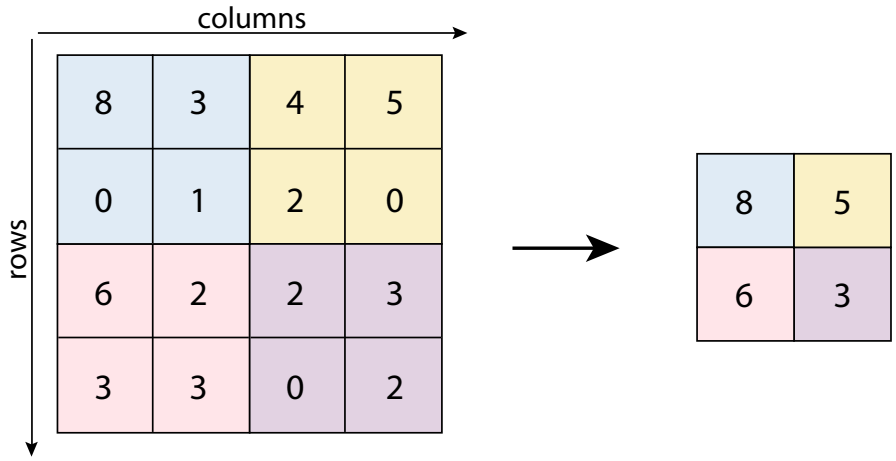

Figure 3.8: This Figure shows the most common form of max-pooling. The matrix on the left is a simplified example (small slice) of an image, where 2×2 max-pooling is applied on with a stride of two. This generates the output matrix on the right.

divides the input picture into bordering squares. The output for each of the squares is determined by finding the maximum value of the respective group, when max pooling is applied. Average pooling e.g. simply computes the arithmetic mean for each square. The intent of max pooling is to detect a high activation for a feature and it's approximate position. The usual size of these squares is 2×2 (referred to as *filter size* or simply $n \times n$ *pooling*) with a stride of two. The stride controls the "overlapping" of the squares. For example, a stride of one would cause the squares to only "move" one pixel at a time over the image, thus the squares would overlap most. With a stride of two the squares are always shifted two pixels spatially over the input matrix, causing them to not overlap when 2×2 max pooling is used (see Figure 3.8). Since pooling processes depth layers individually, the down-sampling only affects width and height but not depth, therefore only resizing the spatial dimension. A pooling layer gradually reduces the dimension of a network and therefore the computation time. Another side effect is that by the spatially down-sampled dimension less neurons are needed to classify this information later on, this leads to reduced overfitting, since less parameters have to be "tuned". It is also common practice to add a pooling between consecutive convolutional layers, focusing the attention of the network.

3.4 Regularizations

3.4.1 Activation Functions

While there are many possible different activation functions, I used only four of the most common ones: rectified linear units (ReLU) [26], rectified linear units with an upper bound of 6 (ReLU6) [19], hyperbolic tangent, and soft-sign activation [8].

A ReLU activation is defined by:

$$h(x) = max(0, x) \ . \tag{3.13}$$

where x represents a feature vector (the input for one complete layer) and h is the activation transformed by the activation function, calculating the output of one layer. ReLU6 activation is defined by:

$$h(x) = min\big(max(0, x), 6\big) \ . \tag{3.14}$$

The upper bound of 6 in ReLU6 activation preserves the network of having too high activations since every activation runs into a hard limit and saturation. Additionally the precision of floating point numbers is higher around 0 and gets lower with values much higher than 0. Hyperbolic tangent ($tanh$) is defined by:

$$h(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \ . \tag{3.15}$$

where e is the Euler's number, $sinh$ the hyperbolic sine, and $cosh$ the hyperbolic cosine. The soft-sign activation is defined by:

$$h(x) = \frac{x}{|x| + 1} \ . \tag{3.16}$$

The soft-sign activation function's shape is comparable to $tanh(x)$ but it saturates slower. This makes the soft-sign activation less dependent on weight initialization, which improves the forward activation as well as the backward learning [8]. A graphical illustration of the different activation functions can be seen in figure 3.9.

3.4.2 Softmax

A softmax function normalizes data by making outliers less important in a data vector. Thus, it is frequently used for output layers of artificial neural networks. It is a special form of a logistic function that "squashes" the values of a n-dimensional vector into an interval of $(0, 1)$. Additionally these new values have a sum of exactly 1.0. For some feature vector $\vec{v}$ the vectorized version of softmax is defined by:

$$\text{softmax}(\vec{v}) = \frac{e^{\vec{v}}}{\sum_{i=1}^{I} e^{\vec{v}_i}} \ . \tag{3.17}$$

where $e^{\vec{v}}$ computes the exponential of every element in the vector and divides this value by the sum of all elements' exponentials. Figure 3.10 shows the benefit of softmax: noise and outliers are reduced, thus a neural network can concentrate on building up correct spikes. Furthermore the teaching signal for all networks in this thesis consists of a normal distribution, which ensures that the sum of such a training vector is smaller than one. Actually the integral of a normal distribution is always exactly one (see equation 4.2), but the teaching signals are vectors and therefore discrete, so that some values are neglected.

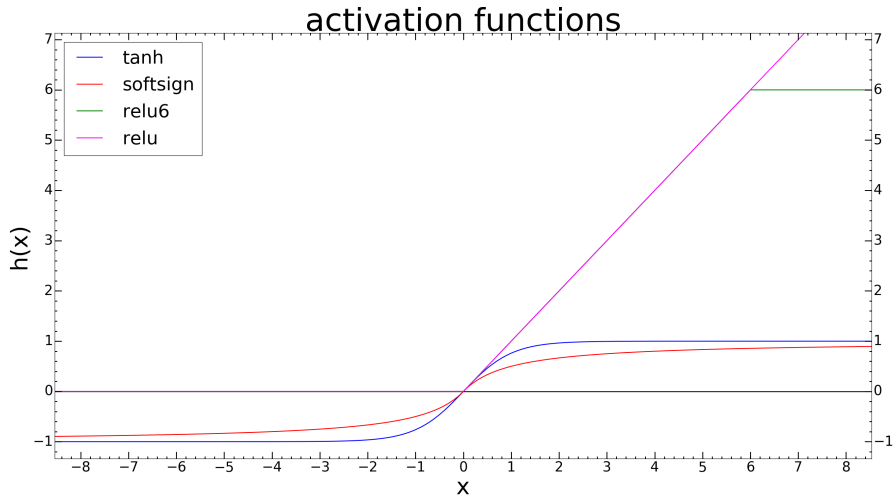

Figure 3.9: Different activation functions: tanh, soft-sign, ReLU, and ReLU6. The values of the ReLU and ReLU6 functions coincide with each other for $x <= 6$. While ReLU is increasing linearly for $x > 0$, ReLU6 is a constant function for $x > 6$ with $h(x) = 6$.

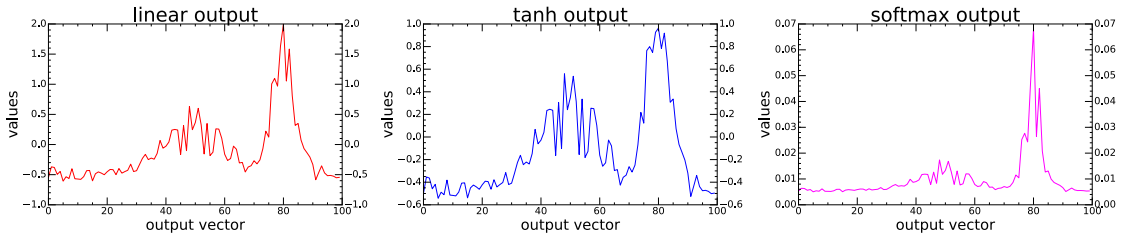

Figure 3.10: Comparison of linear, tanh, and softmax applied to a potential output vector with 100 entries. The linear output vector is negative for some entries, like the tanh. Despite the fact that entry 80 has the highest activation, the noisy output – roughly entries 40 to 60 – are messing up the results. Due to the logistic normalization in softmax this noise is reduced, leaving only one major activity bump at entry 80. Additionally the softmax output guarantees that all entries are in the interval $(0, 1)$.

3.4.3 Adam

The first experiments used standard gradient descent optimization as the learning algorithm, but for big data sets – especially when the model has lots of parameters – need a huge amount of training steps for the networks to converge. Additionally, the chance of falling into local minima is increased. This problem was solved by switching to a stochastic gradient descent algorithm: *Adam*. Adam is successfully proposed especially for deep learning [17]. It utilizes different techniques to improve converging compared to other (stochastic) gradient descent algorithms. One of these techniques is *momentum*, utilizing moving averages of the parameters, to speed up the convergence. To give an example: considering an optimization in a 3-dimensional space, where the objective function's minimum is an elongated

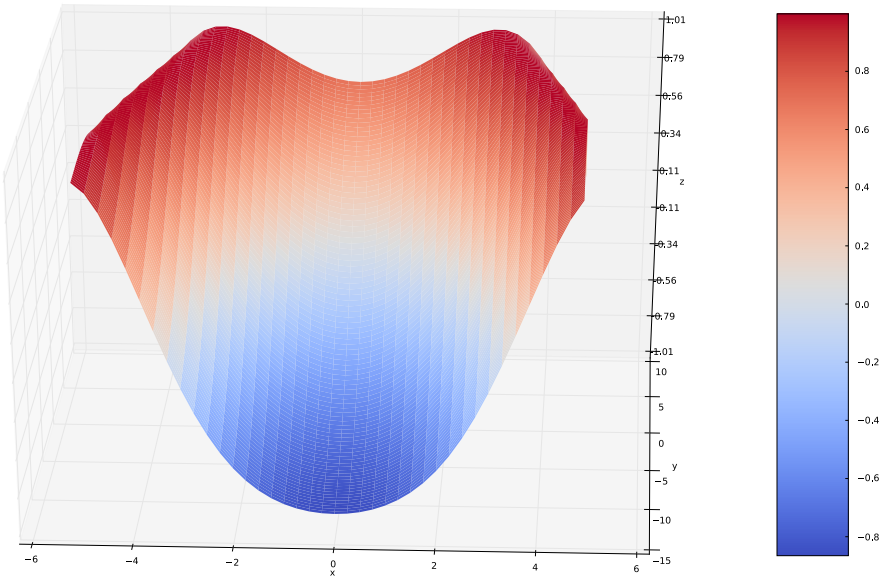

Figure 3.11: Example for an optimization problem. This 3-dimensional plot shows a valley, surrounded by local maxima to the left and right. Since the space of the plot is limited only a part of the valley is shown, the whole valley is significantly longer (y-axis). The minimum decreases stepwise in y-dimension.

valley, like in Figure 3.11. The minimum of the whole valley (y-dimension) decreases stepwise. If the starting point for a standard gradient descent algorithm is somewhere alongside the maxima ("walls" in Figure 3.11), the negative gradient would generate a vector pointing to the steepest descent, which would mostly be downwards the walls of the valley, since the walls decrease the function's value the most. If the learning rate is comparatively low, standard gradient descent would descend downwards the walls, then – alongside the valley's longitude (y-axis) – down to the valley's minimum. Hence, the algorithm would need many iterations. Otherwise, if the learning rate is high, standard gradient descent might overshoot the valley's bottom, thence ending up at the other wall of the valley. Due to the high learning rate, this behavior would repeat, alternating between the left and right wall. Again, the number of iterations would be high. This problem gets even more complex in higher dimensionalities. Utilizing momentum addresses this problem: in every iteration a part of the previous negative gradient is added for the next update. This strategy results in a momentum that quickly accelerates in directions that are similar for several iterations. Therefore, without altering the learning rate, Adam omits recurring small steps in the same direction. Hence, Adam would reach the valley considerably faster as well as redirecting faster to the valley's absolute minimum. Additionally, if the learning rate is too high, momentum reduces the chance of alternating between the walls. This is because "jumping" between the walls needs to greatly change the direction of the trajectory every time, while Adam's momentum benefits following the previous trajectory, damping rapid changes in direction [2].

Another technique Adam introduces is adaptively changing the learning rate for each parameter of the model. In deep learning especially the connections of neurons in the first layers can have small gradients, while the last layers may have high gradients. If parameters are updated by high amounts consecutively, Adam reduces the learning rate for these parameters in subsequent iterations. The same behavior applies, if some parameters are merely updated for several iterations, thus Adam would increase the learning rate for these parameters. This speeds up the learning for all parameters, while preventing to favor or neglect small groups of parameters. Adam's main down side is that the algorithm requires more computations for training, due to calculating the moving averages, variance, and the resulting, scaled gradient.

3.4.4 Xavier Initialization

One of the most ambitious parts for deep neural networks is the initialization of weights [8]. If the weights are initialized too high the signal spreads while being propagated through the network; if the weights are too small it concurs to zero or falls into a local minimum. The first scenario produces an activation of the output layer that will *not* give usable results. Rather than an applicable distribution with an activity bump, the activation is large for all neurons whilst the second scenario results in an output that is similar for almost all input signals – the delta between individually initialized outputs for different inputs is very small. Many networks showed a convergence, when their weights were initialized with a normal distribution $\mathcal{N}(\mu, \sigma^2)$, where $\mu = 0$ and $0.01 <= \sigma^2 <= 0.2$. This was consistently the case for many different parameter settings in the majority (about 60%) of experiments. Although, the results of the first experiments depended on heavy empirical testing, which was very time-consuming. The results were better by using *normalized initialization* [8] (also referred to as *Xavier initialization*), which gives a good approximation for initializing the weights. This approach gave good results in conjunction with soft-sign activation. Essentially, the weights W between layer j and layer $j+1$ are initialized with a uniform distribution (U) with upper and lower bounds defined by:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] . \tag{3.18}$$

where n_j is the amount of neurons in layer j and n_{j+1} the number of neurons in layer $j+1$. In addition to the distribution shown in equation 3.18, it is furthermore possible to approximate the variance of a normal distribution with normalized initialization by taking equation 3.18 as the variance for a normal distribution and set μ to 0. For ReLU activation this initialization also works, but has to be scaled up a bit, due to the ReLU activation's lower bound of 0.

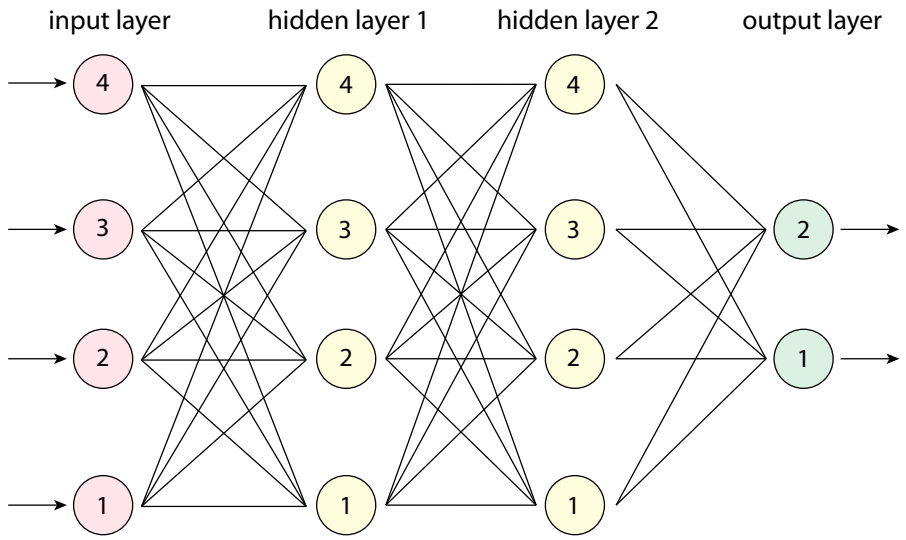

Figure 3.12: Figure of a standard, fully-connected neural network. The network has four input neurons (input layer), two hidden layers with 4 neurons each, and an output layer with two neurons. When trained with dropout, some neurons will get deactivated randomly in each training step.

3.4.5 Dropout

One technique to prevent a model from overfitting and therefore improve the test error is called *dropout* [13]. Basically the idea is to prevent the co-adapting of neurons to each other. Neurons can tend to "familiarize" to specific patterns in their parent layer's output rather than generalizing and considering the whole input vector. Dropout sets the output of randomly chosen neurons to zero with a fixed probability, often 0.5 [20]. This procedure can lower the distortion of a neuron's output, since the neuron cannot rely on the presence of a specific pattern / other neuron's output anymore, due to the random deactivation of dropout in every training step. This higher the chance of neuron's to focus on robust features, neglecting single attractive input neurons. Dropout has proven to be especially effective when applied to fully-connected layers [20].
Figure 3.12 shows a standard, fully-connected neural network architecture. When applying dropout while training, a fixed probability of neurons is deactivated. While dropout possibly could be applied to every layer or even single connections between layers, a common strategy is to only apply dropout on hidden layers. An example of applying dropout to the architecture of Figure 3.12 during training is shown in Figure 3.13. Note that dropout is only applied while training the model; for evaluating the test data set as well as running the network live later on dropout gets deactivated.

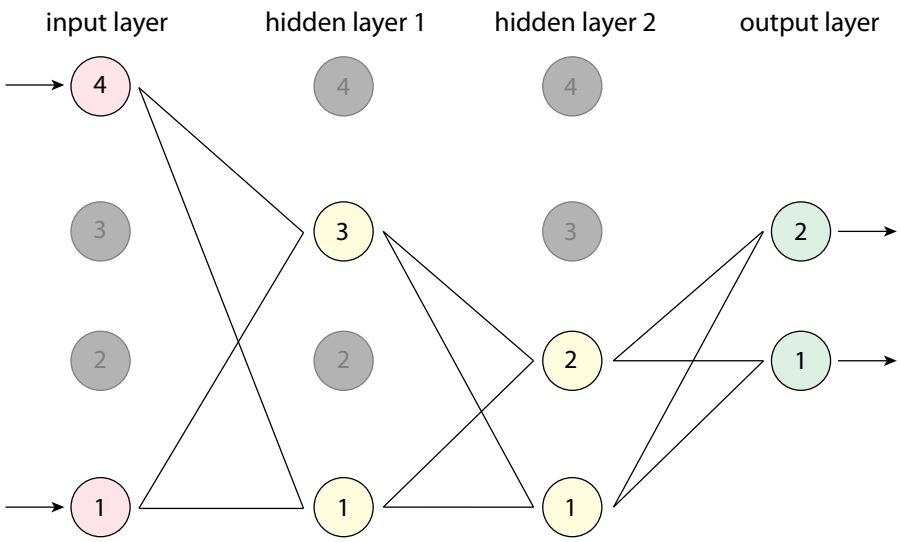

Figure 3.13: Example for the applying of dropout to a neural network while training the model. In each training iteration every neuron is deactivated with a certain, fixed probability.

3.4.6 Top-11 Error

To evaluate our models we used a top-11 error rate, which describes the accuracy of the model by measuring the top-11 activations. Basically this is calculating how many of the model's top-11 activations (which is the activity bump of the network) in one feed-forward step match the teaching signal's top-11 activations. A normal distribution has a symmetric shape, this guarantees that the network's top-11 activations are 5 pixels around the ball's real center (one top activation in the center, 5 to the left, 5 to the right). Hence, this technique simply counts how many of the network's top-11 activations are the same compared to the top-11 activations in the teaching signal. Afterwards an arithmetical means is calculated over every image's individual results, to address the whole test data set. This approach has a worst case of 10 pixels around the ball's center if – and only if – the ball and therefore the normal distribution is exactly at a corner.

Chapter 4

Approach

4.1 Probability Distribution

One-hot encoded vectors are frequently used for the labels of neural networks. They are vectors where every entry is set to zero, except for one value that describes the desired output, which is set to one. In image classification tasks every entry in an one-hot encoded output vector is considered a class that refers to specific object, e.g. different vehicles. After one feedforward activation each entry of the output vector is interpreted as a probability of that object. This is why softmax normalization is quite popular such models, because after softmax is applied to the output vector each entry's value can be interpreted as the probability of the corresponding class/object being presented in the input image.
Instead of one-hot encoded vectors, all models in this thesis use a less standard approach for the teaching signal, i.e. a probability distribution over the whole vector. A normal distribution gets fitted around the real coordinates of the ball's center. Since our input's dimension is $800 \times 600 \times 3$ (800 is the image width, 600

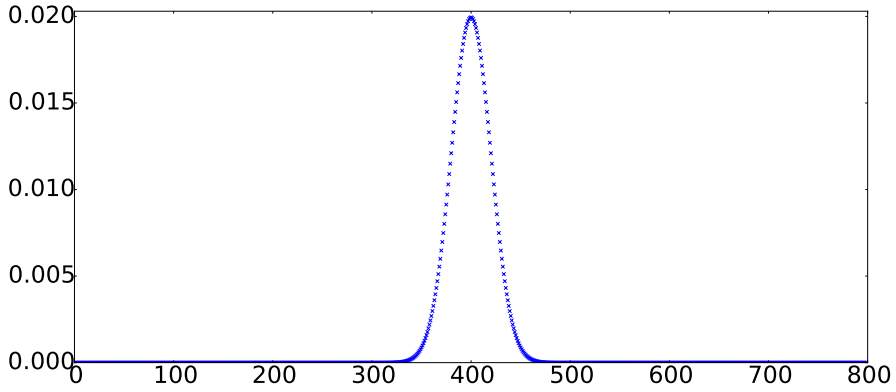

Figure 4.1: A sample normal distribution that is taken as the teaching signal for the x output layer of the network. In this hypothetical sample image the ball is located at $x = 400$, therefore the normal distribution has a μ of 400. The width σ is 20.

the image height, and 3 the RGB-colors), the output layer is a 800-dimensional vector for the x position and a 600-dimensional vector for the y position of the ball. Therefore the teaching signal consists of one 800-dimensional vector for the x output layer (width of image) and one 600-dimensional vector for the y output layer (height of image). A *normal distribution* (also called *Gaussian distribution*) is a continuous probability distribution, which has two parameters: μ (*mean* or *expectation*, sometimes *location*) and σ (*standard deviation*). μ shifts the distribution on the x-axis. The distributions maximum is always exactly at μ. σ varies the distribution's width. Thus, μ is set to the center of the ball, while σ was fixed to 20. Figure 4.1 shows an example of one teaching signal. A normal distribution's probability density is defined by:

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\sigma^2\pi}} * e^{-\frac{(x-\mu)^2}{2\sigma^2}} = \frac{1}{\sigma\sqrt{2\pi}} * e^{-\frac{(x-\mu)^2}{2\sigma^2}} \ . \tag{4.1}$$

where μ is the mean and σ the standard deviation. A normal distribution ensures that all possible values always add up to a sum of exactly one, no matter how the normal distribution was parameterized:

$$\forall\, \mu, \sigma \in \mathbb{R} : \int_{-\infty}^{\infty} \frac{1}{\sigma\sqrt{2\pi}} * e^{-\frac{(x-\mu)^2}{2\sigma^2}} \, \mathrm{d}x = 1 \ . \tag{4.2}$$

Using a normal distributions involves several benefits. The network converges faster with a probability distribution like the normal distributions, because a pixel-precise prediction (when an one-hot encoded vector is used) is harder to learn and has no substantial advantage in terms of the ball localization or tracking. In fact, a pixel-precise prediction would include some problems, like figuring out what predictions can be interpreted as noise. At a feedforward activation the shape of the network's output data can be taken as an observation of the current process's noise. While clean images produce well-formed bell shapes that have maxima of similar width and height, distorted images yield a lot more noise on the shape, more variation of the width and height of the maximum, and perhaps even several smaller bumps. Blurring, reflections, differences in lighting, and other visual distortions frequently caused such noise in the output. Therefore using a probability distribution as the teaching signal improved our results by enabling a faster convergence and supplying additional information about the noise in the input data. Additionally a high number of solutions that deliver a small training error increase the chance of the network to fall into local minima [21]. For that reason the chance of learning invariants and representing a generalization of the problem is increased. This also increases the chance of a low test error. The use of a probability distribution decreases the likeliness for the network to find a solution that only results in a low training error, since the amount of possible solutions for this is decreased. A maximum in the teaching signal – the normal distribution – consists of many classes that are activated with a certain shape, thus there are far less possible solutions with a low training error rate. Figure 4.1 illustrates an example for our teaching signal that gets fed to the network for training.

4.2 Localization Architecture

4.2.1 CNN Development

To keep an overview of all experiments they are divided in different series. Series 1 was the start for the localization architecture's CNN and a typical sliding-window approach: the input image gets divided in squares of the same size. Each of these squares get fed to the network individually, to classify whether a ball is present in the current shape. If a window detects a ball, the ball's position in the image is reconstructed by analyzing the corresponding window's position and it's output. This technique is widely approved [30, 33, 9], but involves some problems: when the ball is exactly at the border of two (or more) windows, the chance is high that neither of them will detect the ball. Additionally, each window's size is just a fraction of the whole image, therefore, when the ball approaches the camera, the ball could get significantly bigger than one window's size. Although there are some techniques that address these problems [27], the sliding-window approach was dropped in this thesis, since the standard sliding-window approach was unpromising for ball localization. Instead, a novel approach for RoboCup humanoid soccer was used by taking the full image as input and localizing the ball via probability distributions, like described in section 4.1.
Series 2 introduced training with probability distributions. This approach delivered promising results, but had one major problem: the output layer was one big 1400-dimensional vector. The first 800 entries modeled the x-position (image width) and the remaining 600 entries the y-position (image height). Throughout 13 different experiments different parameter tunings improved the results, but overall the localization was unstable. Mainly, the lack of robustness was a problem of overlapping distributions: when the balls x-coordinate was higher than 700, while the y-coordinate was less than 100, the two distributions formed one activation with two peaks. This behavior confused the network heavily.
For that reason series 3 was created, where the output layer was split into two different vectors, one 800-dimensional vector (x-coordinate, image width) and one 600-dimensional vector (y-coordinate, image height). This change led to better results and more robustness. Additionally, this approach decouples the dependency between movements on the x- and y-dimension, which is beneficial for the RNN. However, all five experiments of series 3 did not achieved higher test data accuracy than 50%, because (i) the training data set – covering less than 300 images – was too small, and (ii) only the output layers were split, not the higher level, fully-connected, hidden layers.
Series 4 introduced a bigger training data set, including more than 1.000 training images. In addition to that different activation functions, initialization values, and so forth were tested. This raised the performance above 60%. However, the localization performance still was not satisfying. It turned out that one problem was the single, combined hidden layer for both output layers. Nevertheless it turned out that for this architecture soft-sign activation showed the best results; tanh activation was dropped.

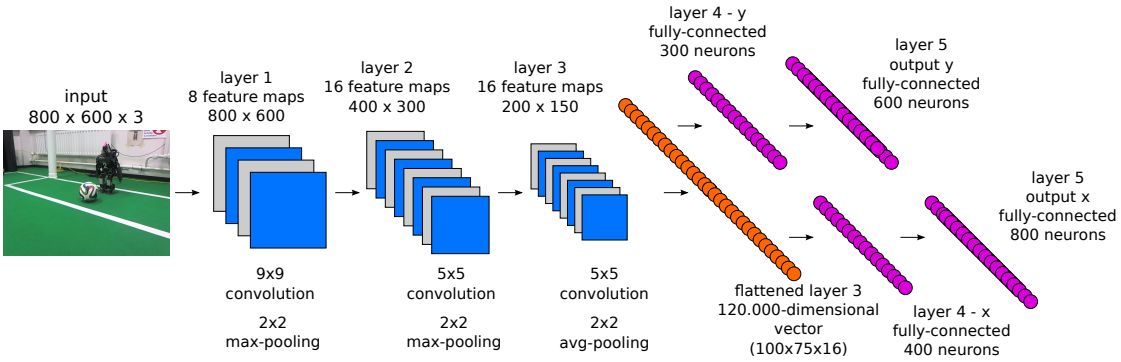

Figure 4.2: Graphical illustration of the architecture of model 1. This model showed the highest accuracy.

Series 5 was the step forward to completely separated output and fully-connected hidden layers for localizing the ball's x- and y-coordinates. From now on, every model's fully-connected hidden layers and output layers were similar to the final design illustrated in Figure 4.2. More tests with different convolutional and pooling setups were made, but still the performance did not go above 70% top-11 accuracy. It turned out that different initializations heavily affected the results. Many empirical tests led to the motivation for a better initialization technique.
In series 6 Xavier initialization (see section 3.4.4) was implemented. This technique raised the localization performance above 70% and was a major step forward. 18 different experiments in series 6 showed that three to five convolutional layers perform the best for this thesis' ball localization task. Additionally training the models on Hummel cluster's GPU nodes enabled the possibility of up to 200.000 training iterations, in order to compare different network designs in converging in less time.
After this experiments the network design was pretty final, in series 7 22 different experiments just evaluated different parameter settings, like number of feature maps, kernel filters, stride, pooling configuration, and so forth. Series 7 achieved 80+% top-11 accuracy. Series 8 did not introduce new architecture insights, it was just a refactoring to prepare the code for different environments, mainly making it easier to port it for the robots, live demos on laptops, etc. Series 9 covered all these features: benchmarks on our team's robots with the new, smaller architectures (800×600 image input size downscaled to 200×150) supplying faster running times, and live demos on a laptop with a camera plugged in to the USB-port. Additionally a whole directory of images could be fed to the network, resulting in a plot of the accuracy per image. Series 10 was the last series that resulted in the proposed architectures. Its main contribution was the average-pooling layer on the third convolutional layer, shown in Figure 4.2.

4.2.2 Proposed CNN Architecture

For building the localization architecture I've experimented empirically with many different setups. The most promising ones where CNNs with three to five convolutional layers, pooling applied on some or all of them and two fully-connected layers (including the output). The first experiments had only one single output layer (one 1400-dimensional vector, 800 for the image's width + 600 for the image's height), but this strategy was less successful, since two neighboring distributions confused the network, thus they look like an overlapping of functions. Therefore I decided to split the output into two different layers. The networks showed the best performance, then this strategy was also applied to the previous fully-connected layer. This led to the current setup all high accuracy networks have in common: three to five convolutional layers, the output of the last convolutional layer transformed into one big, flattened vector, which gets fed to two different, completely independent, fully-connected layers. The best performing network regarding the accuracy was model 1, while model 2 was the only one capable of running on my team's robot.

Model 1 is illustrated in Figure 4.2. It has only three convolutional layers, max-pooling is applied to the first two layers, while average-pooling is applied to the third and last convolutional layer. The aim was to get the best localization. The model was evaluated with and without dropout. If dropout was used, it was applied on every layer except the output layer with a dropout rate of 0.1 for training and no dropout for testing. Higher dropout rates always heavily increased the number of training steps needed to reach a high accuracy. Instead of dropping single connections, always whole neurons have been dropped out. The initial bias for the fully-connected layers was zero, while 0.01 was used for the convolutional layers, and -0.01 for the output layers. These setup was empirically tested and showed a marginally faster convergence for this architecture.

Model 2 – illustrated in Figure 4.3 – was the only one capable of running on the robots. The network's architecture is similar to model 1, but the input images are scaled down in size by 75%. Additionally no average pooling is applied on the third convolutional layer, the kernel filters for the convolution are smaller, and the fully-connected layers are also scaled down in size. This heavily boosts the computation time needed for one feed-forward activation as well as the time needed to train the model. While the full-size networks (800×600 input image size) consume 1 to 6 GB system memory in total, model 2 needs less than 800 MB. One feed-forward activation on a modern laptop (Intel Skylake i7 U-Series; mobile processor for low power consumption: 15W TDP) takes 0.026s with model 2, while model 1 needs $0.91s$. These value are arithmetical means, generated with the test data set. The small network was always about 25 to 40 times faster compared to the big nets (depending on their configuration).

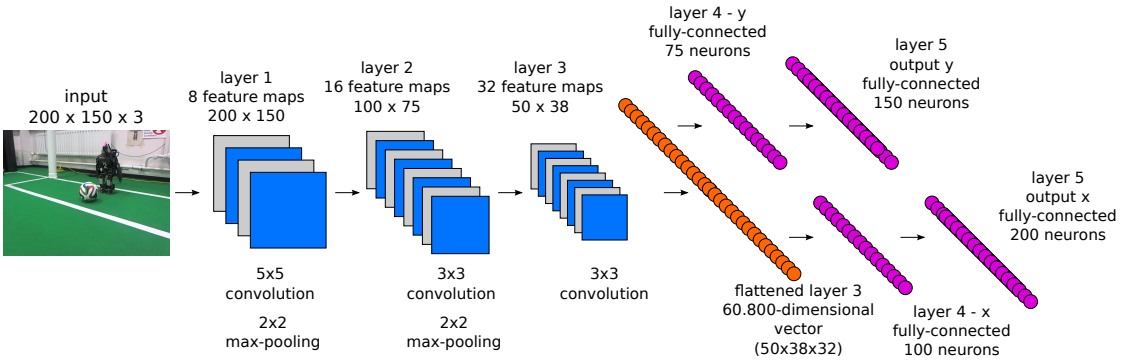

Figure 4.3: Graphical illustration of the architecture of model 2. This model was the only one capable of running on the robots. The image resolution was reduced by 75% to speed up the computations and consume less memory. Therefore the input size of the 800×600 images (original) is decreased to 200×150.

4.3 Tracking Architecture

Localizing the ball is a key feature for RoboCup humanoid soccer. However, in an actual game the robots also have to plan reasonable actions. Hence, some form of prediction is needed in order to develop good strategies. This prediction needs to take invariants in time into account and calculate possible outcomes for the future. In terms of ball localization the prediction of future outcomes needs the model to analyze the ball's trajectory, which leads to the need of ball tracking. Without predictions for future ball positions, the robot would consistently try to follow the ball's current position. This would be inefficient. Therefore it is a better choice to predict the ball's trajectory and plan one's movements accordingly. Especially for the goal keeper it is mandatory to know if a moving ball would score a goal in the future or not. In order to block the ball the goalkeeper has to decide whether to go left or right, when the ball is kicked. For addressing this tracking a concept of a recurrent neural network is proposed. Moreover the localization architecture's output always covers some noise. A RNN can filter this noise over time, which is a key benefit for accurately modeling the ball's position. To remove the dependency between x- and y-movements, there were two RNN's trained with the same network design. One for the x-movement with 800-dimensional input and output, one for the y-movement with 600-dimensional input and output.

4.3.1 RNN Development

Due to the fact that just a concept for the RNN is proposed, the RNN covered less experiments. At the beginning a RNN with topographic weight initialization was used, but the results were worse compared to a RNN with Xavier initialization. Soft-sign showed the smoothest output in terms of filtering, therefore the RNN – as shown in Figure 4.4 – was the final design found for the concept. The total amount of experiments was just two for the topographic RNNs and five for

the Xavier initialization ones. One problem was that the training data set did not include many sequences. Therefore a world model was built that simulates movements for x- and y-dimensions. The starting point and acceleration is initialized randomly. RNNs modeling the x-dimensions are trained with the full range of the 800-dimensional vector, while the world model does not use the full range for the y-dimension. That is due to the fact that movements from left to right often use the whole dimension of the image, but top to bottom movements mostly scale alongside the depth, rarely hitting the top or bottom of the input image. In fact, this is just the case when the robot's camera looks down or up by more than 25 degrees. Moreover the world model de-accelerates the movements with an exponential function, gathered by an exponential regression of real sequences. The movements in y-dimension are always modeled slower (due to the depth) by the world model script. After a RNN model is trained with the world model script, the progress is saved. This pre-trained model is then re-used to train it with the real world sequences, but for less iterations.

4.3.2 Proposed RNN Architecture Concept

For the architecture a simple recurrent neural network (see Figure 3.6) is used. Model RNN 05 performed best. In order to work on the whole CNN output it's input layer is 800-dimensional for the x-distributions and 600-dimensional for the y-distributions. The hidden layer has 100 neurons and the output is again 800 for the x-prediction respectively 600 for the y-prediction. Therefore the RNN predicts a whole distribution, rather single coordinates. This has several benefits, firstly the reduced chance of fall into local minima, but most importantly it does not only predict future distributions but also filter the distribution. Thus, noisy distributions get smoothed to produce accurate output. The network architecture of model RNN 05 for the x-distribution is shown in Figure 4.4. I use two different version of each RNN model, one for the x-distributions (800-dimensional input and output) and one for the y-distributions (600-dimensional input and output).

4.4 Technical Approach

Every model, evaluation, calculation, and so forth was implemented in Python[1], utilizing different libraries. Most of the plots were generated with Matplotlib[2], a Python plotting library with an interactive interface. Calculations – despite the models themselves – were made with NumPy[3], an efficient numerical computing library, developed for scientific computing. It supplies an interface to C-Arrays in order to keep computations fast. However, NumPy solely processes data using the CPU. Since the models' amount of parameters is very high, they require many computations, drawing a CPU inappropriate for training. Additionally, a high level

[1] `https://www.python.org/`
[2] `http://matplotlib.org/`
[3] `http://www.numpy.org/`

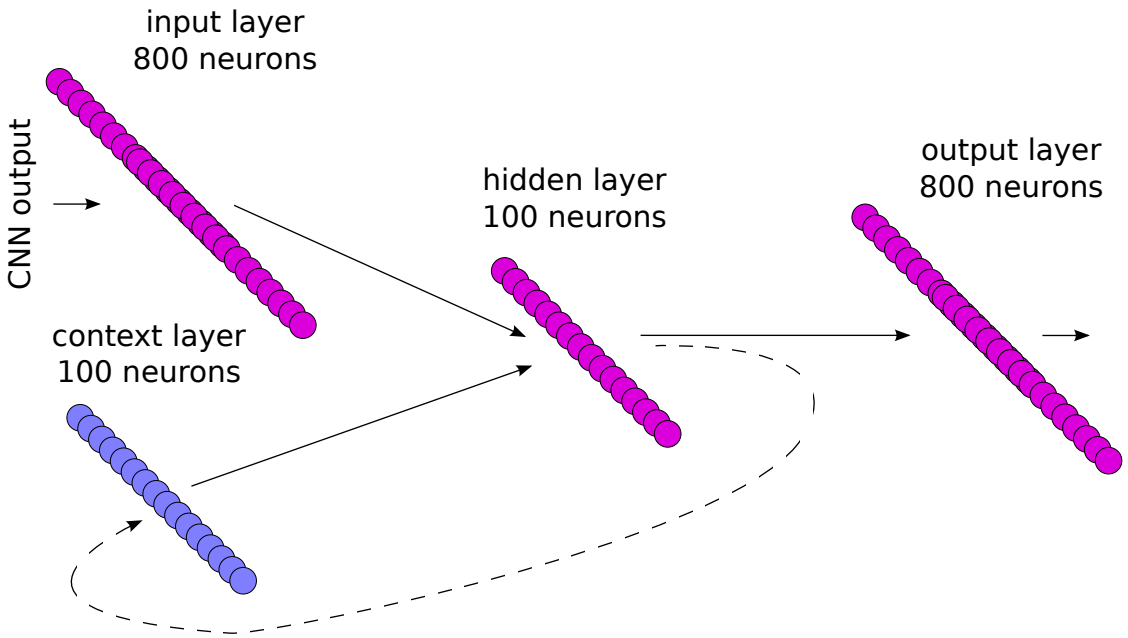

Figure 4.4: Figure of recurrent neural network model 05. This is the configuration for the x-distribution. The setup for the y-distribution is the same, except for the input and output layer size, which is 600 to fit for the y-distributions.

library increases the number of possible experiments, because the implementation time for each experiment is decreased. Therefore all models created in this thesis were implemented in TensorFlow[4], which is designed for numerical computations – especially for neural networks – and offers the possibility of training the models using a GPU. For saving large matrices or vectors, e.g. evaluation data for each experiment, h5py[5] was used.
The models were trained on the Hummel Cluster[6], a high performance computing (HPC) cluster of the University of Hamburg. This was mandatory, because training model 1 (see Figure 4.2) took one hour for less than 200 iterations on a standard desktop 4-Core CPU. Thus, a whole training would take days for single experiment. The Hummel cluster's CPU nodes consist of two 8-Core Intel Xeon CPUs, enabling to train the models with 32 threads in parallel. Unfortunately, due to the fact that CPU's have a comparably low rate of floating point operations per second (FLOPs), this was not fast enough, either. One node of the cluster needed one hour for less than 1.000 iterations during training. This led setting up the architecture for GPU-Computing. The Hummel cluster does also offer GPU nodes, which consist of a NVIDIA K80 dual-gpu graphics card, a professional GPU for scientific computing. Utilizing TensorFlow and NVIDIA's CUDA[7] as well as cuDNN[8] – NVIDIA's GPU-

[4] https://www.tensorflow.org/
[5] http://www.h5py.org/
[6] https://www.rrz.uni-hamburg.de/services/hpc/hummel-2015/hardware-konfiguration.html
[7] https://en.wikipedia.org/wiki/CUDA
[8] https://developer.nvidia.com/cudnn

Computing language and NVIDIA's library for deep neural networks – enabled the possibility to train the models on the GPU nodes. One GPU node on the Hummel cluster processed barely 20.000 iterations in one hour.

4.5 Robots

The new robots of our team are based on the Hambot platform [3]. This platform uses an ODROID-XU3 Lite[9], featuring 2GB system memory and a Samsung Exynos 5422 processor, which consists of one Cortex-A15 and one Cortex-A7 4-Core CPU. Since the CPUs have an ARMv7 instruction set, any software has to be compiled for ARMv7 to run on the robots. As of June 2016, TensorFlow did not provide any install packages for ARM CPUs. Hence, TensorFlow and Bazel (Google's build tool) had to be supplied with an ARM toolchain and compiled for ARMv7 manually.

[9] `http://www.hardkernel.com/main/products/prdt_info.php?g_code=G141351880955`

Chapter 5

Methodology

5.1 Data

CNNs need a large training data set [23]. All images of our data set have been recorded in our laboratory, which has a play field set up according to the RoboCup humanoid league rules but scaled down in size. The dataset contains 1.160 images; 80 are used as test images while 1.080 are used for training the network. The images portray various scenes, some are very clean with just the ball on the field, others cover a goalkeeper robot, the goal posts, a striker robot, and several arbitrary objects in the background (like tables, chairs, a heater, windows, a door). See Figure 5.1 for examples. Mostly the distance to the ball is between 0.5 and 5.0 meters. 400 images show a non-moving ball at various locations, while the rest are sequences of a moving ball. Roughly 20% of the training and test images contain a robot, a goal post or something similar right next to the ball and in almost 50 images the ball is partly covered, e.g. by the legs of a robot. The images are of dimension $800 \times 600 \times 3$ (width $\times$ height $\times$ RGB-channels) without *any* preprocessing. Hence, the dataset contains images with reflections, blurry images, images with overemphasized color channels, and so forth. Especially the red channel is intensified in some images, letting white walls appear pink.

5.2 Experiments

For evaluating the CNN architectures we randomly chose single frames of a non-moving ball or a few frames out of sequences as test images and measured the accuracy. In the sequences the ball is moving in various directions. We started with clean scenes with just a few objects and the ball, but our current dataset contains more images with robots, goal posts and miscellaneous objects (tables, chairs, windows, doors, heaters) in the background. The RNN concept architecture is tested only with our recorded sequences.

Figure 5.1: Illustration of some test images used for evaluation, taken from the robot's camera.

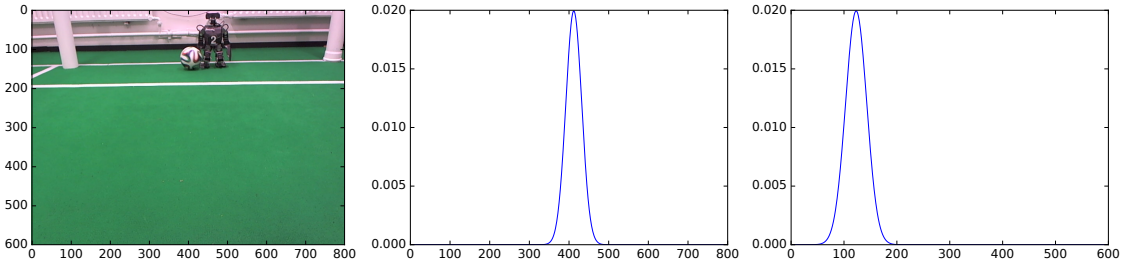

Figure 5.2: One test image including its teaching signals. Original test image (left side), x-axis probability distribution (center), and y-axis probability distribution (right side).

5.3 Results – CNN

Overall every network that converged successfully started to converge after about 15.000 to 30.000 training steps. Some of the easier, cleaner training images even started to converge after approximately 8.000 training steps. More complex images (partly covered balls, images with a high distance to the ball) always needed at least 15.000 training steps. Obviously the networks were focusing on easily detectable features first, while distinguishing between similar objects (e.g. the ball near the goal posts – both are mostly white) was reasonable tougher for the networks. Table 5.1 shows the results of the model performing best (model 10.01) and our small network, capable of running on the robots (modell 10.07) with soft-sign activation, which empirically proofed to the best activation function for our networks. The results are *only* evaluated on test images. Model 10.01 outperformed all other models. While soft-sign activation delivered the best results, ReLU6 was just off by some percent. ReLU activation often produced the worst accuracy.

Table 5.1: Results for full training.

network	top11 x peak	top11 y peak
model 1 soft-sign	**89%**	**86%**
model 2 soft-sign	53%	48%

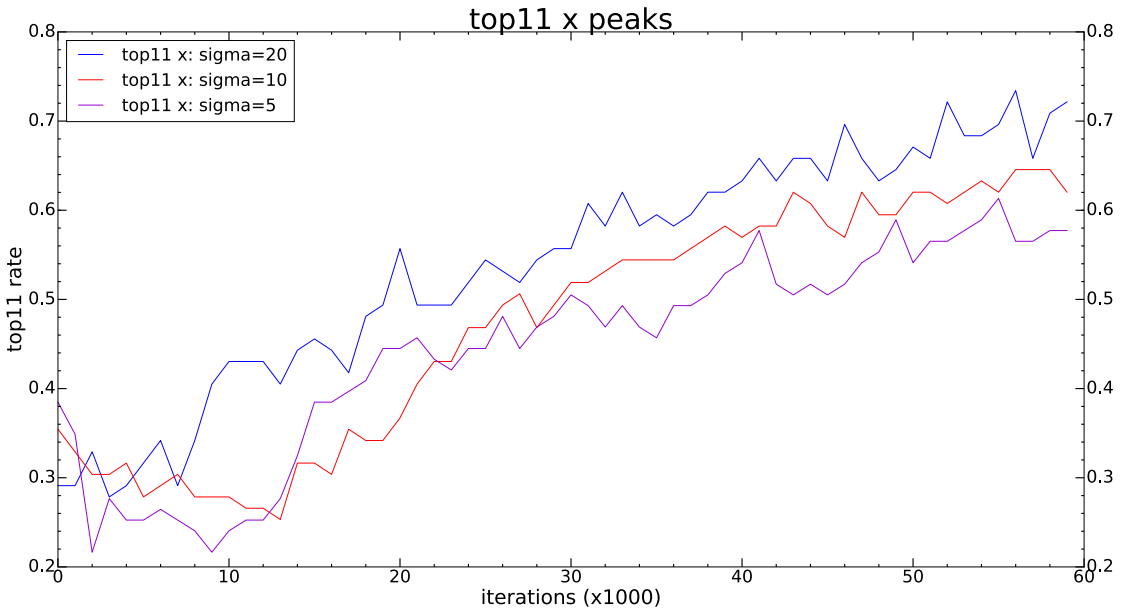

Figure 5.3: Top-11 test accuracy for model 1 with three different sigma settings; 20 is the default. Narrowed distributions results in lower test accuracy. The top-11 y-accuracy showed a similar behavior.

Hyperbolic tangent and sigmoid activation were only tested in the very first CNN experiments and were dropped due to unsatisfactory results.
The network's output is visualized by plotting the top-11 prediction as well as a heatmap on top of the test images. The heatmaps and the distribution plots, illustrated in Figure 5.4, show that the network is localizing the ball accurately. The outcome of smaller probability distributions was also tested: a decreased sigma (standard deviation) results in worsened top-11 test accuracy. However, the training error did not increase. The top-11 test accuracy for this scenario is shown in Figure 5.3

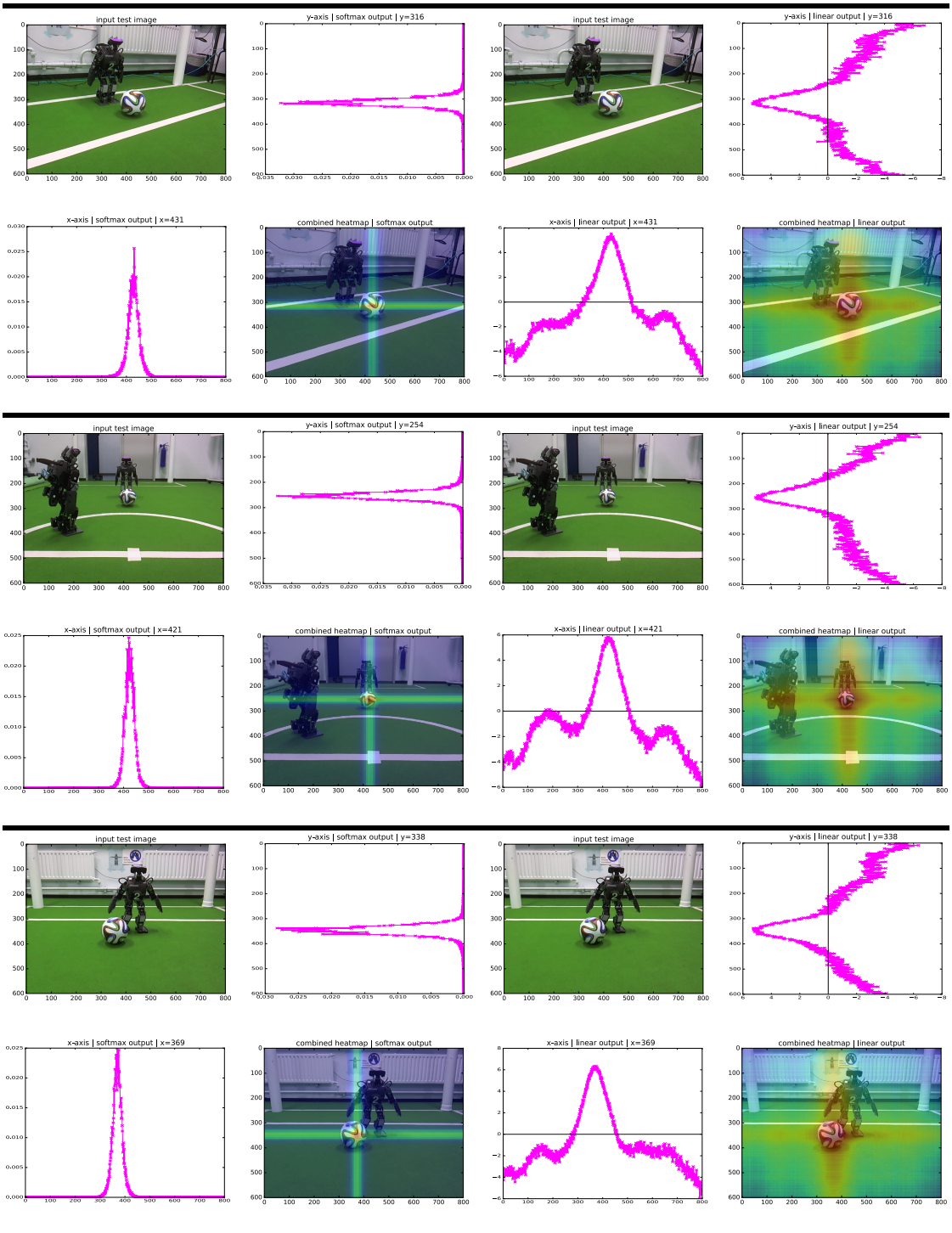

Figure 5.4: Output of model 1 for three different test images including the probability distributions and heatmaps for evaluation.

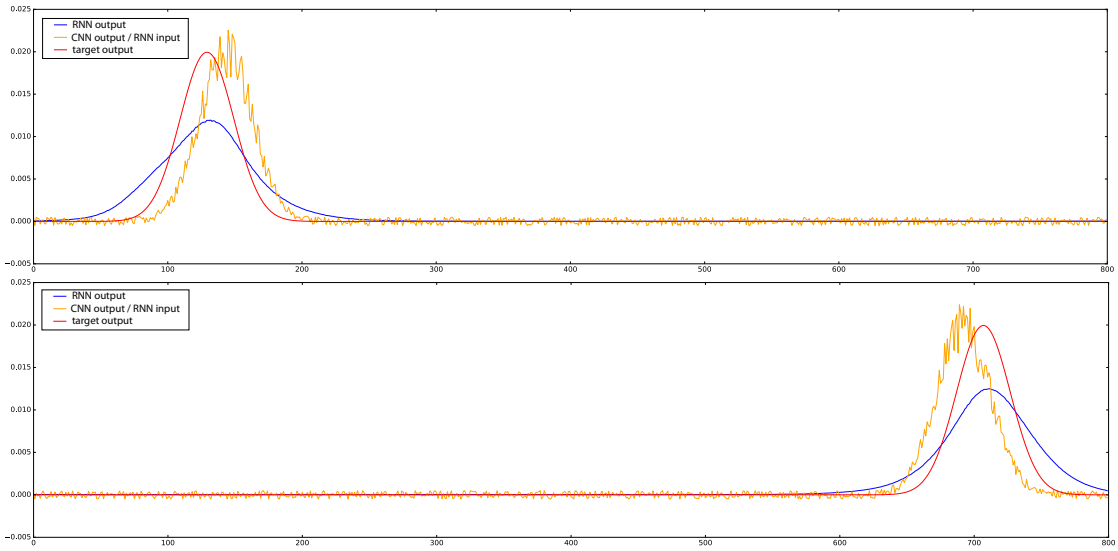

Figure 5.5: Output of the RNN for two *different*, unrelated test sequences, including a high noise CNN output. The blue line shows the RNN output, the yellow line the CNN output respectively RNN input, and the red line the target output (perfect probability distributions for the current test sequence's frame).

5.4 Results – RNN

The concept for the recurrent neural network had issues in predicting the ball's future position: building up the probability distributions at the right place, predicting the ball's future position, mostly took more than 10 subsequent frames and even after this period the prediction was often off by ±20 pixels. However, the results were reproducible and show that the RNN always converges towards an accurate prediction with some uncertainty. Additionally the RNN was able to remove the noise of the CNNs output even very unclean images. An example for a high noise CNN output is shown in Figure 5.5.

Chapter 6

Discussion

6.1 Localization Architecture Analysis

Model 1 (see Figure 4.2) delivered the best results of all models and was able to reliably detect the new ball, even in newly recorded scenes: in a demo video sequence[1] it detected the ball despite the fact, that the network's training images do not contain any persons, cardboard boxes, cupboards, and so on, which are shown in the video. With a top-11 error precision peak of 89% (x) respectively 86% (y) – as shown in table 5.1 – the model was able to accurately detect the ball in more than 90% of the test images (only less than 4% of the test images had noisy activity bumps where the top-11 activations were off by more than 10 pixels).
Albeit the fact that most models showed reasonable results, all networks trained with the full resolution (800×600) were too big to run on our robots (2GB system memory). Therefore the downscaled models were introduced. Model 2 was able to run on the robots with a reduced input dimensions of 200×150, while achieving usable results for short distances. The top-11 rates were above 60% for distances to the ball in a range of 1 to 2.5 meters. Whenever the distance to the ball is too short, the ball is considerably bigger than the kernel filters that look for characteristic features, hence they mostly miss to detect the ball. Similar outcomes apply for too high distances: at 4 meters and above the ball's size is less than 10 pixels in width/height, thus the ball's appearance is very blurry and most features are too small to trigger the network. However, the processing time was 25 to 35 times faster compared to the full size networks, which is a huge benefit for low resource robots.

6.2 Tracking Architecture Analysis

With the concept of the RNN an architecture is supplied that is able to smooth noisy distributions. Therefore even outputs with a higher discrepancy (like 10-20

[1] `http://data.bit-bots.de/videos/speck_bachelor_demo_video.avi`

pixels off) may be useful in a game, due to the smoothing and the recurrence of the RNN. Additionally filtering the approximated location of the ball is more important than a pixel-precise knowledge of the ball's position. However, this does not have to imply a tradeoff between an accurate knowledge of the position and a filtered output. In fact, filtering the CNN's output can improve the results by achieving a higher resolution. Moreover the RNN's output is stable and robust, due to its time invariant interpretation. When the CNN's output localizes one false positive, for example a goal post, in three subsequent frames the RNN's output would mostly stay at the original position. A pixel-precise information is even unnecessary for some scenarios, like the goal keeper. An approximated prediction to get an idea of the ball's trajectory is more important, since the ball has just to be blocked. Furthermore the robot's hardware is not able to kick or catch balls at pixel-precision levels. For that reason, distributions that point in the right direction are an efficient way for ball localization, especially when the discrepancy to the real coordinates is low enough.

6.3 Architecture Insights

The main task of the architecture is to localize the ball, nonetheless probability distributions – in contrast to single coordinates – supply an approximated idea about the uncertainty of the localization, by interpreting the distribution's shape. Clean, bell-shaped distributions cover low noise and achieve high accuracies, while noisy distributions lower the accuracy. In most cases, this also implies that the input image was noisy, blurry, had a bad illumination or contained an very unknown background that confuses the CNN. Like all deep learning architectures our models need a big training data set, otherwise the prediction results worsen a lot. Scenes showing a partly covered ball mostly led to mis-localizations. The assumption is that the training data set is too small for extracting enough features of partly covered balls, thus they are not localized reliably. Scenes with no ball falsely led the network to output some location, although the activity bumps in the linear output were always smaller compared to input images showing a ball, consequently implementing a threshold to the linear output is possible to address this issue. However, simply increasing the size of the training data set should also improve the results, since currently there are clearly not enough complex training samples and therefore invariants for such complex scenes.
Concerning the models' architecture soft-sign activation achieved the best results of all tested activation functions, therefore it was the best choice for the particular network setup in this thesis and the task of RoboCup humanoid soccer ball localization. The first experiments covered many tests in order to tune the network's parameters, especially the initialization of weights. This led to the idea of normalizing the initializations to get better results and faster convergence, which was realized with Xavier initialization and significantly improved the convergence of all networks [8]. CNNs with three to four convolutional layers supplied the highest accuracy, in combination with pooling. The kernel filters for the convolution

are trained, which improved the results compared to fixed kernel filters that were used in the first experiments. Furthermore trained kernel filters are biologically more plausible. The most successful pooling strategy was to use max-pooling twice and afterwards average-pooling. The max-pooling in the first layers extracted significant, characteristic features and localized these features. In contrast to this, average pooling in the first layers seems to always lose some spatial information. However, three max-pooling layers did not improve the results. The feature maps suggest that three max-pooling layers filter too strictly. An average pooling layer as the third and last pooling layer improved the results, as the average pooling layer just down-samples the dimensionality while mostly keeping the information of the previous layer. The feature maps look less strict and the reduced number of connections for the fully-connected layers – due to the reduced dimensionality achieved by the third pooling layer – enable faster convergence, because less parameters have to be adjusted. Another strategy used is dropout to reduce overfitting. Possibly because of the rather small training data set dropout worsened the results with high dropout rates. Best results were achieved with a dropout rate of 10%. Thus, co-adapting of neurons is presents and logically, since the fully-connected layers have a vast amount of neurons / connections. Dropout was used just for training, not for testing or live evaluation. For the training itself Adam proved itself superior to a standard gradient descent algorithm.

6.4 General Analysis

The main downside of the CNN is the fixed sigma rate, which adjusts the teaching signals width. This results in the CNN being only capable of detecting the ball's features in a specific range of size. Every time the ball is significantly larger or smaller the accuracy drops. In live testing sessions this behavior mainly existed for distances less than 40cm or more than 4m to the ball. Furthermore inhibitory feedback should stabilize the results and lead the less false positives, but it turned out that learning inhibitory feedback was complicated for the network and therefore not a main part of the thesis. While inhibitory feedback reduced false positives, like goal posts, it also increase false negatives, there the ball was clearly inside the image but the network activation was lower and more noisy.
The statement of Larochelle et al. that a high number of solutions delivering a small training error may increase the chance of the network converging into local minima supports the results shown in Fig 5.3. By narrowing the normal distributions (teaching signal) the test accuracy dropped, but the training errors were similar and did not drop significantly. Narrowing the normal distributions slowly converges the teaching signal towards a one-hot encoded vector, where more possibilities show up to lower the training error, but most of these possibilities cover local minima that prevent the network from converging towards better solutions that offer higher test accuracy [21].
According to the RNN the main disadvantage was the very slow adaption of the ball's movement. The RNN mostly needed more than 10 frames to build up a

correct prediction. Before the movement of the activity bump builds up, the RNN mostly placed the activity bump around the input signal. However, as already discussed, this could also be beneficial: when the CNN rapidly changes the ball's position for very few frames (due to false positives), the RNN would mainly remain the original position.
The architectures proposed by this thesis are adaptive and can be trained for a wide variety of balls easily. Moreover, when tested with a different (mostly white) ball – that the network was *not* trained with – the results were promising: the chance of a correct prediction was almost 50:50, although the CNNs output was highly noised and less accurate compared to the trained ball. Nonetheless this proves the idea of generalization of the network. Other balls that are less similar to the trained ball, which have a different surface including less white color, for example the old orange ball, were not recognized. Overall a comparison to our current team's solution is difficult, since this solutions aims for a pixel-precise localization and single coordinates. Furthermore it needs less computation time and is able to run on the robots. However, the robustness of model 1, especially with its generalization to similar balls, enables a localization that is more stable and accurate, if the CNN's output gets filtered by the RNN or a similar architecture.

Chapter 7

Conclusion

7.1 Conclusion

This thesis proposed a deep neural architecture, which is able to reliably locate the ball, although no pre-training was used for the CNN. Albeit the training data set is comparably small, even new, more complex scenes were processed reliably. Thus, the architecture – if supplied with enough training data – should outperforms standard vision algorithms. Different ball orientations, speeds/trajectories, color distributions, distances, and other characteristics are important key features that have to be covered by the training data. Due to the low computational power of our robots, especially when compared to modern computers, the size of the network is limited. For that reason the development of the proposed architectures was not only focused on achieving satisfactory accuracy, but the limited computational power of the robots was also kept in mind. Though considerably bigger, deeper neural networks with a different setup may achieve better results, the current setup achieves the results with just a few layers and not *too* many neurons – enabling at least current laptops to run the models live. With a training time that took no longer than 10 to 20 hours for a full training on GPU node of the Hummel cluster, the architectures learn comparably fast. Additionally, the thesis showed that the architectures can run on modern robots with just small changes. Conclusively, the networks are able to deliver a reasonable performance in locating *multi-color* balls with arbitrary color patterns, even when the ball is moving. Moreover, localization is not the only benefit. The CNN not only predicts the ball's location, but does so by producing a distribution over full dimension (width & height) of the image. The distributions give an idea about the noise, also. Figure 5.4 shows that for an accurate prediction the distributions on the output layer have a single bump with a spiky maximum. Noisy distributions can be utilized to interpret the noise of the current process. Additionally the x- and y-output bumps are of a similar shape (width, height). This behavior can be seen in the majority of the test images, when fed to the network. Therefore the hypothesis of the distributions can be accepted and suggested for similar tasks, since the results clearly show the benefit.

7.2 Future Work

The most significant benefit should be provided by varying the sigma rate of the normal distributions in dependency of the ball's current size in the input image. In order to learn a better representation of invariances and therefore create better architectures, a larger, more complex data set is needed, which has more images are where the ball is partly covered, as well as images with no ball at all. The training data set is simply too small: in some situations the endlines or goal posts (because of their white color and in some angles round-shaped appearance) shift the network's attention. The activity bumps moved in between the ball and these objects. This behavior most likely happens, because the majority of images does not include the goal posts or only small parts of them. Even mirroring the images of the training data improved our results. The top-11 accuracy of the test was increased by a few percent. Hence, recording a bigger training data set with more images that show varying contents should stabilize the results. Our current data set only has a few images that hide the ball e.g. between the legs of a robot. Addressing the problems of inhibitory feedback should further improve the performance.
Another goal is to filter the noise and to predict the ball's movement over several frames with a recurrent neural network. The current RNN is promising, but needs improvements: the lazy adaption of the RNN to the ball's current movement has to solved, preferably while keeping its robustness. One possible and interesting solution is implementing a neural Kalman filter with recurrent neural networks [15]. Szirtes et al. have shown that a neural architecture can learn predictable features along with the noise on specific features of the input information [32]. Potentially future work could connect both parts of the architecture even more: when the recurrent neural network outputs the next prediction of the ball's location, this information could be used to direct the attention of the convolutional neural network to the corresponding area in the image. Deep architectures can supply very good, robust results for those tasks [33].

Appendix A

Nomenclature

Appendix B

Additional Proofs

Appendix C

Complete Simulation Results

Bibliography

[1] T Bandlow, M Klupsch, R Hanek, and T Schmitt. Fast image segmentation, object recognition and localization in a robocup scenario. *RoboCup-99: Robot Soccer*, 1999.

[2] Yoshua Bengio. Practical Recommendations for Gradient-Based Training of Deep Architectures. 2012.

[3] Marc Bestmann, Bente Reichardt, and Florens Wasserfall. Hambot : An Open Source Robot for RoboCup Soccer. 2015.

[4] Dan Cirean, Ueli Meier, and Juergen Schmidhuber. Multi-column Deep Neural Networks for Image Classification. *International Conference of Pattern Recognition*, (February):3642–3649, 2012.

[5] Genevieve Coath and Phillip Musumeci. Adaptive arc fitting for ball detection in robocup. *Proceedings of APRS Workshop on Digital Image Analysing, Brisbane, Australia*, pages 63–68, 2003.

[6] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. Deep Neural Networks for Object Detection. *Advances in Neural Information Processing Systems*, pages 2553–2561, 2013.

[7] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. Scalable Object Detection Using Deep Neural Networks. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2147–2154, 2014.

[8] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Aistats*, 9:249–256, 2010.

[9] Pavel Golik, Patrick Doetsch, and Hermann Ney. Cross-entropy vs. Squared error training: A theoretical and experimental comparison. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, 2(2):1756–1760, 2013.

[10] R Hanek, T Schmitt, S Buck, and M Beetz. Towards robocup without color labeling. *RoboCup 2002: Robot Soccer World Cup VI*, pages 179–194, 2002.

[11] R Hanek, T Schmitt, S Buck, and M Beetz. Towards robocup without color labeling. *RoboCup 2002: Robot Soccer*, 2002.

[12] R. Hecht-Nielsen. Theory of the backpropagation neural network. In *International Joint Conference on Neural Networks*, pages 593–605 vol.1. IEEE, 1989.

[13] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv: 1207.0580*, pages 1–18, 2012.

[14] M Jamzad, B S Sadjad, V S Mirrokni, M Kazemi, H Chitsaz, A Heydarnoori, M T Hajiaghai, and E Chiniforooshan. RoboCup 2001: Robot Soccer World Cup V. chapter A Fast Vis, pages 71–80. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[15] Rudolph Emil Kalman. A New Approach to Linear Filtering and Prediction Problems. 1960.

[16] Andrej Karpathy and Thomas Leung. Large-scale Video Classification with Convolutional Neural Networks. *Proceedings of 2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[17] Diederik P. Kingma and Jimmy Lei Ba. Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations*, pages 1–13, 2015.

[18] Norbert Kr, Peter Janssen, Sinan Kalkan, Markus Lappe, Justus Piater, and Antonio J Rodr. Deep Hierarchies in the Primate Visual Cortex : What Can We Learn For Computer Vision ? *IEEE Transactions on Pattern Analysis and Machine Intelligence,*, 35(8):1847 – 1871, 2012.

[19] A Krizhevsky and G Hinton. Convolutional Deep Belief Networks on CIFAR-10. *Unpublished manuscript*, 2010.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, pages 1–9, 2012.

[21] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring Strategies for Training Deep Neural Networks. *The Journal of Machine Learning Research*, 10:1–40, 2009.

[22] Y LeCun, L Bottou, Y Bengio, and P Haffner. Gradient Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[23] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. Convolutional networks and applications in vision. *ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems*, pages 253–256, 2010.

[24] Jitendra Malik, Ross Girshick, Jeff Donahue, and Trevor Darrell. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. *2014 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 580–587, 2014.

[25] CL Murch and SK Chalup. Combining Edge Detection and Colour Segmentation in the Four-Legged League. *Australasian Conference on Robotics and Automation (ACRA'2004)*, 2004.

[26] V Nair and GE Hinton. Rectified linear units improve restricted boltzmann machines. *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

[27] Maxime Oquab. Is object localization for free? Weakly-supervised learning with convolutional neural networks. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 685–694, 2015.

[28] Bente Pakkenberg, Dorte Pelvig, Lisbeth Marner, Mads J. Bundgaard, Hans Jørgen G Gundersen, Jens R. Nyengaard, and Lisbeth Regeur. Aging and the human neocortex. *Experimental Gerontology*, 38(1-2):95–99, 2003.

[29] RoboCup-Team. RoboCup Soccer Humanoid League Rules and Setup. 2015.

[30] P Sermanet, D Eigen, and X Zhang. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[31] Christian Szegedy, Scott Reed, Pierre Sermanet, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. pages 1–12, 2014.

[32] Gábor Szirtes, Barnabás Póczos, and András Lrincz. Neural Kalman filter. *Neurocomputing*, 65-66:349–355, 2005.

[33] Kaihua Zhang, Qingshan Liu, Yi Wu, and Ming-Hsuan Yang. Robust Visual Tracking via Convolutional Networks. *CoRR*, abs/1501.0:1–18, 2015.

Erklärung der Urheberschaft

Ich versichere an Eides statt, dass ich die Bachelor thesis im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum Unterschrift

Erklärung zur Veröffentlichung

Ich erkläre mein Einverstndnis mit der Einstellung dieser Bachelor thesis in den Bestand der Bibliothek.

Ort, Datum Unterschrift