HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Autoencoders for Object Classification and Position Estimation in Humanoid Soccer Robots

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science (M. Sc.)

eingereicht von:	Max Paul Patzelt
geboren am:	12.01.1991
geboren in:	Frankfurt am Main
Gutachter/innen:	Prof. Dr. Verena Hafner Prof. Dr.-Ing. Peter Eisert

eingereicht am: ……………… verteidigt am: ………………

Abstract. In this thesis, we explore the practical applications of autoencoders in robotics through a case study involving the Humboldt University's Adaptive Systems research group and their *"Berlin United"* robotic soccer team. Our exploration focuses on the applicability of autoencoders for joint object classification and feature regression using the SoftBank Nao humanoid robot platform, which is prominently used in the RoboCup Soccer Standard Platform League (SPL). Given the extensive set of unlabeled training data accumulated by the Berlin United team throughout their participation in SPL, we propose a hybrid learning framework that combines unsupervised and supervised learning techniques to enhance the perception and decision-making capabilities of these robots. The proposed framework aims to address challenges encountered by humanoid robots acting in dynamic and visually variable environments, by improving the robustness, precision, and real-time inference capabilities of the robot's visual perception systems. Specifically, we will investigate the efficacy of using autoencoder models as the basis for soccer ball recognition and localization.

Contents

1 Introduction

In this thesis, we explore the practical applications of autoencoders in robotics through a case study involving a long-running project at the Humboldt Universities Adaptive Systems research group, the *"Berlin United"* robotic soccer team. We investigate the applicability of autoencoders for joint object classification and feature regression using the SoftBank Robotics Nao humanoid robot platform [1], a robot prominently used in the RoboCup SPL [2] robotic soccer competition. The Berlin United team has participated in the SPL since its first iteration in 2008[3] and has since collected a large set of unlabeled training data suitable for exploring unsupervised learning approaches.

In the domain of humanoid robotics, the perception and interpretation of the environment are critical for autonomous operation and decision-making. Traditionally, the training of machine learning models in these domains is performed using supervised learning, a technique that requires extensive labeled datasets. As the labeling of data is inherently time-consuming, it can become infeasible for smaller groups such as student organizations.

To play soccer well, robots must perform various multimodal tasks, including but not limited to planning and executing locomotion, detecting and avoiding falls and collisions, identifying objects and landmarks, performing self localization, building and updating internal abstracted representations of their environment, and performing team communication. Recognizing and detecting the location of the soccer ball is a crucial component that influences most of these tasks, so achieving a precise and robust ball detection mechanism is of immense value in RoboCup. In this thesis, we explore the efficacy of training an autoencoder model on a vast amount of unlabeled training data and subsequently fine-tuning the model for soccer ball detection.

1.1 RoboCup Soccer

The RoboCup initiative was founded in 1993, with the first RoboCup Event held in 1997 [4]. RoboCup promotes research on robotics and artificial intelligence through competitions, most prominently the RoboCup Soccer competition. The self-proclaimed goal of RoboCup Soccer states that

> *By the middle of the 21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of FIFA, against the winner of the most recent World Cup.* [5]

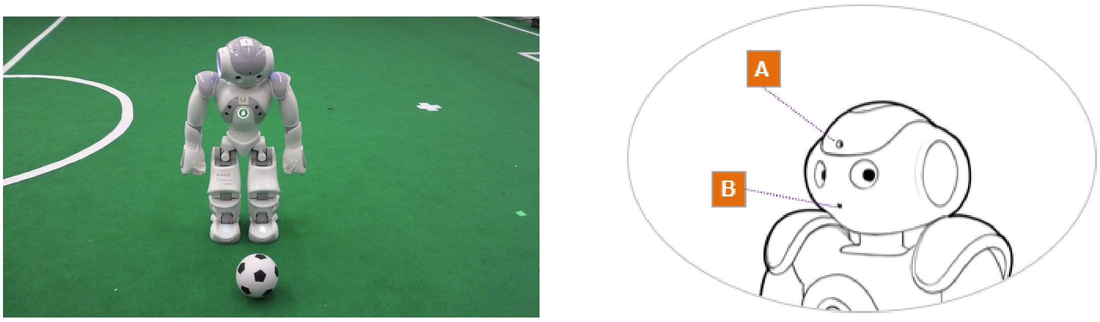
This vision serves as a stretch goal, incentivizing the development of novel technologies and algorithms through a playful yet appealing vision.

The RoboCup Soccer competition is composed of multiple leagues, each with a particular set of rules, objectives and participating teams. In the RoboCup SPL, all teams work with the same model of humanoid robot, specifically the Nao model by SoftBank Robotics (formerly Aldebaran Robotics) [6]. This sets the focus of the SPL on systems design and the development of holistic frameworks that enable the robots to act autonomously in a standardized environment, where hardware variations are eliminated to focus purely on software innovation. Each robot operates autonomously and must therefore plan and perform actions in real-time and on-device. Each year, the competition's difficulty increases as more complex challenges and rules are introduced, challenging participants to continually evolve and improve their software systems. Communication between robots is regulated, with a limited number of messages allowed to be exchanged between robots of the same team wirelessly and through sound.

The Humboldt University has a long-standing tradition of participating in RoboCup, starting with the *Humboldt Heroes* competing in RoboCup from 1999 onwards and later the *Aibo Team Humboldt*, which emerged from the previous team in 2001 [3]. Until 2007, the Sony AIBO four-legged robot was the only model allowed to play in the aptly named Sony Four-Legged League. With the introduction of the Nao robot, the league was renamed to Standard Platform League and the HU team followed suit, rebranding as *Nao Team Humboldt* or *NaoTH* for short. In 2011, a joined *Berlin United* team was formed together with the *FUmanoids* team from Berlin. This collaboration lasted until 2017 when the FUmanoids disbanded. Since then, *NaoTH* remains as the sole member of team *Berlin United* and still competes under this name.

RoboCup Soccer, especially the SPL, provides a well-suited environment for research in robotics and artificial intelligence. By requiring teams to use the same hardware, the SPL removes variability in the physical capabilities of robots and focuses development efforts on effective and efficient software solutions for complex tasks, such as motion planning, perception, real-time decision-making, and multi-agent cooperation. The limited processing capabilities of the Nao robot restrict the usage of complex existing software components, which necessitates a good understanding of the presented challenges and requires specific and optimized solutions capable of overcoming them. Although the robot soccer framework provides a constrained environment and well-specified game mechanics and rules to follow, having multiple agents dynamically interacting with their environment and working toward a common goal remains a vastly complex task. Even in the well-defined bounds of RoboCup Soccer, every game will present unique environmental conditions and no two sensors or motors ever behave completely identical. Within a match, the dynamic decision-making processes of various agents inevitably lead to novel situations that have never been observed or simulated before. Sensors produce noisy results and require intermittent re-calibration, and any downstream consumers need to be able to handle the inherent uncertainty of their measurements. By providing different research environments and focus points for robotic capabilities in multiple disciplines, the RoboCup framework fosters incremental innovations in different aspects of robotics and artificial intelligence research, making a complex problem more approachable by breaking it down into smaller, more manageable challenges.

In this work, we hope to contribute to robotics in the spirit of RoboCup by breaking down a complex challenge into manageable sub-problems that we can better understand and design optimized solutions for.



(a) (b)

Figure 1: *(a): A Nao robot on a soccer pitch with a RoboCup regulation soccer ball. The graphic is sourced from the official RoboCup SPL rulebook [8]. (b) Camera placement in the Nao v6 robot. A denotes the top camera and B the bottom camera. The graphic is sourced from the Nao v6 datasheet [9].*

1.2 Nao Robot

The Nao is a humanoid robot developed by SoftBank Robotics. Currently, the Berlin United team uses the most recent generation of Nao robots, the version 6 model. The Nao v6 is a bipedal humanoid robot, standing 57.4 cm tall and weighing 5.48 kg [7]. It has 25 Degrees of Freedom (DoF), with 2 DoF in the head, 5 DoF in each of the two arms and legs, 1 DoF per Hand and 1 DoF in the pelvis.

The Nao v6 robot is equipped with two identical video cameras, located in the forehead and mouth area. At a resolution of 640x48 pixels, each camera can record 30 frames per second. The cameras have a diagonal Field of View (FoV) of 60.9° (56.3° horizontal FoV, 43.7° vertical FoV) [7].

The Nao v6 also includes other sensors such as touch sensors, an inertial measurement unit (IMU), force-sensitive resistors in the feet, four omnidirectional microphones and a sonar for distance measurement. It is equipped with an Intel Atom E3845 quad-core CPU (1.91 GHz).

Figure 1 shows a Nao robot with a RoboCup regulation soccer ball and highlights the camera locations in the head unit.

1.3 Problem Statement

The objective of this thesis is to develop and evaluate an autoencoder-based learning framework for soccer ball detection in the context of humanoid robot soccer. The framework uses a large volume of unlabeled image data collected from robot logs during RoboCup matches and training sessions to learn robust feature representations in an unsupervised training paradigm. We evaluate the efficacy of using autoencoders as a feature extractors for two critical downstream tasks, soccer ball classification and position estimation. The effectiveness of this approach will be compared against the current classification and regression models used by the Berlin United team. In this thesis, we investigate the potential of autoencoder-based feature extraction to reduce reliance on labeled data and enhance the adaptability and robustness of our robotic systems to new environments.

1.4 Research Objectives

The goal of this research is to develop a general and robust learning framework that uses existing unlabeled data to enhance the perception and decision-making capabilities of the robots in the Berlin United RoboCup Soccer Team. To keep the research and development of this framework aligned with our overarching objective, we establish the following evaluation criteria.

Generalizability This thesis develops a flexible and adaptable model training and inference framework that can be broadly applied across diverse robotics contexts. The framework incorporates modular components that can be adapted to different environments and tasks, providing a solid foundation for future research. We focus exclusively on image-based data and do not incorporate other sensory or robot-specific information to ensure that the training and inference processes are adaptable to other computer vision tasks.

Robustness In robotics, sensory environments are highly variable. This is particularly true for visual perception, where stark differences in brightness, contrast, color, and noise occur frequently due to changes in the environment and sensor calibration. Furthermore, a robot in motion often records blurred images due to the limitations of its cameras. The proposed framework accounts for this variability to ensure that the resulting models are robust against changing environmental conditions.

Precision Precision is a metric used to evaluate classification performance and measures the proportion of true positive predictions out of all positive predictions made by the predictive model. In the context of soccer-playing robots, achieving high precision is crucial because doing so minimizes the occurrence of false positives, i.e., predictions in which a classifier incorrectly identifies an object as a soccer ball. Such errors cause a robot to react unnecessarily to irrelevant objects, resulting in wasted computing resources, faulty internal modeling, and gameplay disruptions. Where applicable, we focus on developing precise models that minimize false-positive predictions. We aim to develop classification models that achieve a precision score of at least 95%, preferably higher.

Real-time Inference When robots interact with their environment, they must quickly respond to sensory inputs to detect objects, avoid obstacles, and adjust their movements. This real-time behavior is a hallmark of embedded systems. In this thesis, the upper bound for on-device inference is given by the Nao robot, which is equipped with two cameras running at a maximum of 30 frames per second each. This results in a total of 60 images to process per second, necessitating a maximum inference time of 16 ms per camera image, assuming that we could spend all compute resources exclusively on ball detection. However, since the robot requires additional computing resources for other cognition- and motion-related tasks, we define an optimistic target inference time of 8 ms to process all potential ball candidates.

Reproducibility To ensure that the experiments and analyses we conduct can be reproduced and expanded upon by the Berlin United team in the future, we use a combination of code versioning, containerization techniques and the teams *MLFlow* machine learning operations and metrics server instance to log experimental metrics and artifacts. Where applicable, contributions are integrated into the appropriate Berlin United software repositories.

1.5 Contributions of this Thesis

As part of this thesis, we develop, train and evaluate various autoencoder architectures and investigate their efficacy as shared feature extractors for ball classification and ball position estimation networks. We perform a thorough analysis of the foundational components of autoencoders and find an optimized baseline architecture for future research. We establish an evaluation framework that goes beyond the analysis of performance metrics by visualizing qualitative aspects of trained models to better understand their decision-making process. We develop a containerized model training workflow that enables isolated, hardware-independent, and reproducible experimentation. To ensure we operate on high-quality training data, we annotated, adjusted or confirmed the ground truth labels of over 200,000 data samples. We develop novel tools to aid in the annotation and model selection process. Our contributions are integrated into the existing Berlin United software repository and infrastructure.

1.6 Thesis Outline

Following this introduction, where we outlined the problem statement and research objectives of the thesis, we review computer vision and machine learning fundamentals relating to neural networks and autoencoders, as well as relevant research and literature, in Section 2. In Section 3 we present our methodology and introduce the datasets we will be working with, before conducting a series of experiments investigating various design aspects of autoencoders to find an optimized baseline architecture for further experimentation. We then implement and evaluate multiple autoencoder based machine learning models for soccer ball classification and position estimation. In Section 4, we analyze the experimental results, benchmark inference performance on the Nao robot and visualize qualitative aspects of the developed models to make an informed model selection decision. Finally, Section 5 concludes the thesis with a summary of the key research findings, a discussion of limitations, and suggestions for future work.

2 Literature Review

In this section, we provide an overview of key concepts and prior work relevant to our research of using autoencoders as the basis for ball detection in RoboCup Soccer. We begin with Section 2.1, reviewing the fundamental building blocks of artificial neural networks and laying a foundation for understanding more complex model architectures that will be presented in this thesis. In Section 2.2, we examine the Convolutional Neural Network (CNN) architecture and its image processing capabilities, as they will provide the basis for our work with autoencoders. This is followed by an overview of various autoencoder architectures, including convolutional autoencoders, denoising autoencoders and variational autoencoders in Section 2.3, examining their potential as feature extractors for ball classification and position estimation models. Finally, we survey how other leading teams perform ball detection in RoboCup and what we can learn from their approaches in Section 2.4. This section serves as the theoretical foundation for the experiments on autoencoder based feature extraction for soccer ball detection in Section 3.

2.1 Machine Learning with Artificial Neural Networks

Let us briefly review the building blocks of Artificial Neural Networks (ANNs) so we can better understand how autoencoders are constructed and trained. In this thesis, we assume some familiarity with the concept of Neural Networks on the readers' part. To summarize, we define Machine Learning as a field of artificial intelligence research that focuses on developing algorithms capable of identifying patterns and learning relationships within datasets without the need for explicitly programmed instructions. In a supervised learning paradigm, such algorithms are trained on labeled data, meaning for each input datum, there exists a distinct output datum, and a mapping from input to output is learned. In the unsupervised learning paradigm, there exist no such labels, and the algorithm learns to find patterns and structures in the unlabeled input data. Feed Forward Neural Networks are a class of machine learning models that are inspired by the biological brain, where a multitude of artificial neurons, together with their structure and weighted connections, define a concrete model instance [10]. The network learns by iteratively updating the connection weights based on the processed data samples and an objective function that determines the quality of its predictions. Through optimization techniques, the network weights are adjusted to minimize an objective function error term. Machine learning models can be trained to perform various tasks including classification, regression, feature extraction and pattern recognition.

Neuron The smallest unit of composition in ANNs is the artificial neuron, also known as Perceptron [11]. While the artificial neuron is inspired by the biological neuron, it is highly abstracted in form and function. Neural Networks composed of artificial neurons therefore also differ significantly from their biological counterparts. Artificial Neurons compute an output value that depends on one or more weighted inputs, a bias value that represents an inherent excitation level, and a typically nonlinear activation function. Formally, this can be expressed as

$$o = \phi\left(\sum_{i=1}^{n} w_i^\top x_i + b\right) \tag{1}$$

where **o** is the output, ϕ is the activation function, **w** is the weight vector, **x** is the input vector, and **b** is the bias term. A typical Neural Network is composed of multiple layers, each of which is in turn composed of one or more artificial neurons.

Activation Function Nonlinear activation functions are essential for a model's ability to learn and represent complex patterns and relationships in the input and its intermediate representations. Without nonlinear activation functions, a network of artificial neurons as described above would be equivalent to a series of linear transformations, which could be reduced into a single matrix multiplication. Commonly used nonlinear activation functions include the sigmoid function, the hyperbolic tangent function, the Rectified Linear Unit (ReLU) activation function and the Leaky ReLU activation function. Figure 2 shows, how each of the activation functions transforms a real valued scalar x.

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

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3}$$

$$\text{ReLU}(x) = \max(0, x) \tag{4}$$

$$\text{LeakyReLU}(x) = \max(\alpha x, x) \tag{5}$$

Layer A layer is a composite made up of one or more artificial neurons. Typically, neural networks are organized to have one input layer, one or more hidden layers and one output layer. By organizing artificial neurons in layers, we can abstractly mimic structures naturally found in the biological brain that facilitate hierarchical feature extraction and integration. The neuron activations can be computed efficiently using matrix multiplications in each layer. Information is propagated forward through the network, leading to the designation of such networks as Feed Forward Neural Networks. Figure 3 shows a simple fully connected network architecture with two neurons in the input layer, three neurons in the hidden layer, and one neuron in the output layer. When a network consists exclusively of fully connected layers and has at least one hidden layer, it is referred to as a Multi-Layer Perceptron (MLP).

Let us quickly show, how the forward pass of the network depicted in Figure 3 is

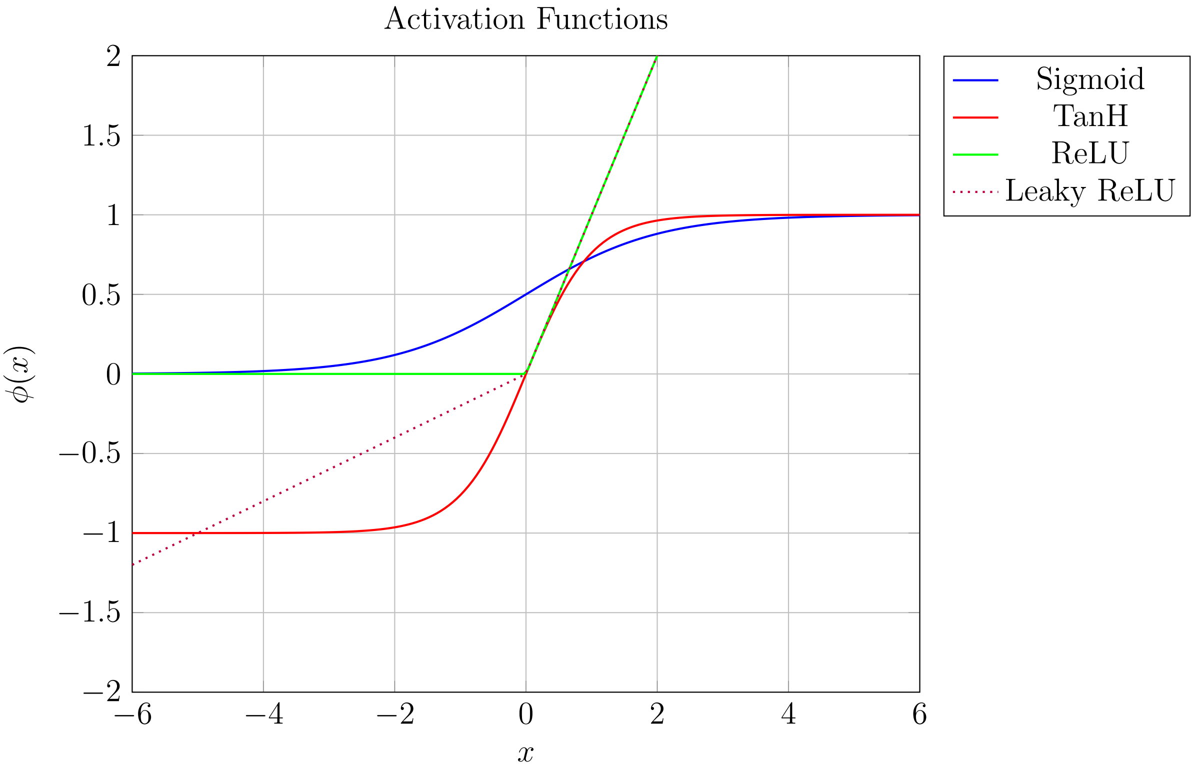


Figure 2: *A plot of four common activation functions applied to a real valued input x: Sigmoid, TanH, ReLU, and Leaky ReLU. The sigmoid function (blue) maps x to a range between 0 and 1, the TanH function (red) maps x to a range between -1 and 1, the ReLU function (green) outputs x if x is positive and 0 otherwise, and the Leaky ReLU function (purple, dotted) outputs x if x is positive and $0.2x$ otherwise. The scaling factor of the Leaky ReLU function is a configurable parameter.*

computed. We define

$\mathbf{x}$ as the input vector,
$\mathbf{W}^{(1)}$ as the weight matrix between the input and hidden layers,
$\mathbf{b}^{(1)}$ as the bias vector for the hidden layer,
$\mathbf{h}$ as the hidden layer neuron vector
$\mathbf{W}^{(2)}$ as the weight matrix between the hidden and output layers,
$\mathbf{b}^{(2)}$ as the bias vector for the output layer,
ϕ as the activation function

Now we can calculate the activation of the hidden layer neurons as follows:

$$\mathbf{h} = \phi(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) = \phi\left(\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) \tag{6}$$

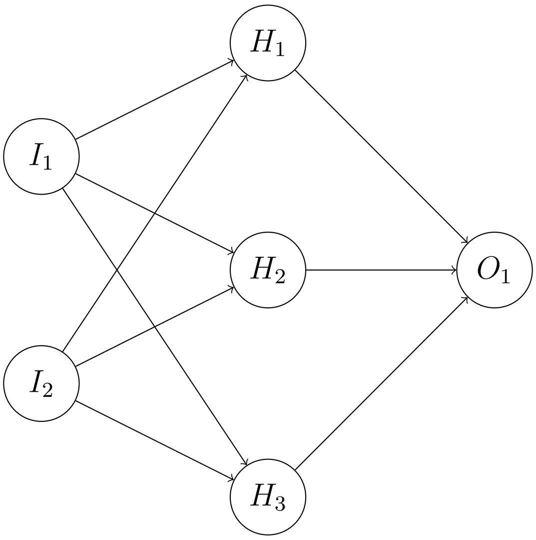


Figure 3: *A simple Feed Forward Neural Network architecture. Each input neuron I is connected to every hidden neuron H. The hidden neurons H are all connected to the output neuron O.*

The activation of the output neuron is calculated accordingly as

$$y = \phi(\mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}) = \phi\left(\begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} + b_o\right) \tag{7}$$

Loss Function A loss function, or objective function, is used to quantify the prediction error of a Neural Network after a successful forward pass. The obtained error value is then used to adjust the weights of the network's neural connections in a way that minimizes the prediction error in subsequent forward passes. Depending on the task to be learned, a suitable loss function has to be chosen. For classification tasks, Cross-Entropy loss is a common choice, while regression tasks often employ distance losses such as the Mean Squared Error (MSE). If we define n as the number of samples, y as the true value and $\hat{y}$ as the predicted value, the Binary Cross-Entropy (BCE) and MSE losses are computed as follows:

$$\mathcal{L}_{BCE}(y, \hat{y}) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{8}$$

$$\mathcal{L}_{MSE}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{9}$$

In Section 2.3.1 we examine the applicability of the aforementioned loss functions to autoencoder training in more detail.

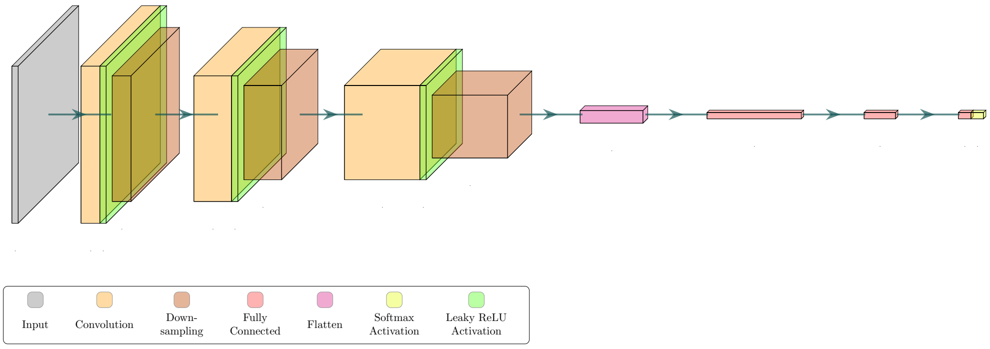


Figure 4: *A simple Convolutional Neural Network architecture with three convolutional blocks, each consisting of a convolutional layer, a non-linear activation function and a downsampling layer. The convolutional blocks are followed by fully connected layers. The output layer has a softmax activation function, indicating that this example architecture might be used to learn a classification task.*

Backpropagation & Optimization After performing a forward pass through a Neural Network and obtaining a loss value, the backpropagation algorithm [12] can be used to efficiently compute the gradients of the loss with respect to each network weight. We start with partial derivatives of the loss with respect to each weight in the output layer, then we propagate the error backwards through the network by applying the chain rule to compute the gradients in the previous layers. Optimization refers to the process of updating the weights using the gradients computed with the backpropagation algorithm to minimize the error in subsequent forward passes. The gradients are typically computed on a subset of the available training data and scaled by a learning rate factor α. This optimization technique is known as Mini-Batch Gradient Descent (MBGD) [13].

The MBGD update rule can be expressed as:

$$w_{t+1} = w_t - \alpha \frac{1}{m} \sum_{i=1}^{m} \nabla_{w_t} \mathcal{L}(y_i, \hat{y}_i) \tag{10}$$

where w_t represents the weights at iteration t, α is the learning rate, m is the batch size, and $\nabla_{w_t} L(y_i, \hat{y}_i)$ is the gradient of the loss function $\mathcal{L}$ with respect to the weights w_t, computed for the i-th sample in the batch. Since we are computing an average gradient over the batch, the batch size becomes a tunable hyperparameter at model training time.

2.2 Convolutional Neural Networks

CNNs are hierarchically structured Feed Forward Neural Networks capable of efficiently processing image-based input data [14]. In 1998, Yann LeCun introduced the LeNet-5 CNN architecture [15], laying the groundwork for a series of advancements in neural network based computer vision, such as AlexNet [16], Inception [17], ResNet [18], the YOLO [19] architecture family. CNNs have since become a staple of machine learning in the image processing domain. CNNs drastically reduce the number of connections between neurons compared to vanilla Feed Forward Neural Networks using fully connected layers by convolving input tensors with one or more kernel matrices in each convolutional layer. This enables the practical training of deeper network architectures with fewer parameters. The output of a convolution layer is a series of feature maps which serve as the input features to subsequent layers. By stacking multiple convolutional layers, a network learns to represent increasingly complex hierarchical feature combinations. The task-specific optimal weight values of each kernel matrix are approximated during model training using an objective function and optimization algorithm. Inputs are commonly represented as tensors of shape $(HEIGHT_{input}, WIDTH_{input}, CHANNELS_{input})$.

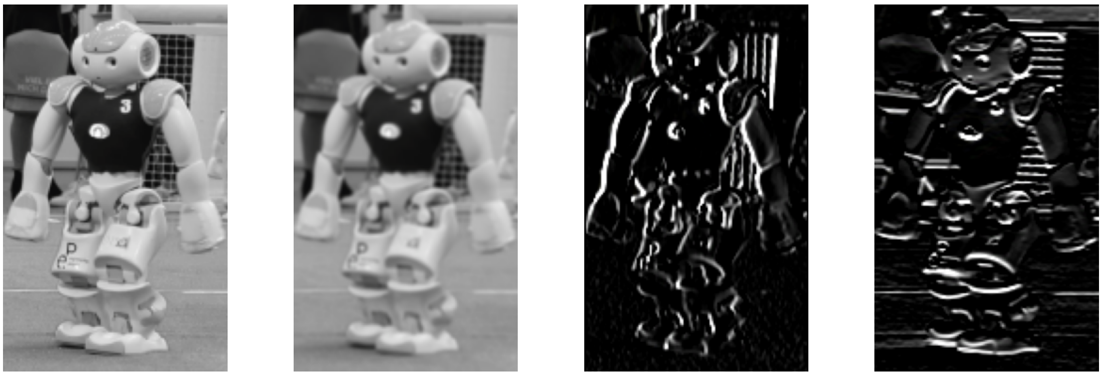
Kernels The kernel, or convolutional matrix, is the basic building block of CNNs and is typically structured as a square matrix of integer or floating-point values. A kernel can be used as an operand in a convolution operation, which produces a scalar value when applied to a region of the input with the same shape as the kernel. The convolution operation in image processing is defined as

$$g(x,y) = \omega * f(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \omega(i,j) f(x-i, y-j) \tag{11}$$

where $g(x,y)$ is the output value at position (x,y) after applying the convolution operation, ω is the kernel, $f(x,y)$ is the input image at position (x,y), and a and b are the half-width and half-height of the kernel respectively. The convolution operation produces a weighted sum of an input pixel and its neighbors, with the kernel determining the weights to be applied. For grayscale images that contain only one intensity channel and thus have the shape $(\text{input height}, \text{input width}, 1)$, a kernel with a single channel is sufficient for performing the convolution operation. For multichannel inputs, such as RGB or YUV color space images with a shape of $(HEIGHT_{input}, WIDTH_{input}, 3)$, a kernel needs to have the same number of channels, also called kernel depth, to produce a single value when convolved with a region of the input. Kernels have long been used in image processing because they can produce various useful image transformations based on the weights that are used. Figure 5 illustrates the effects of convolving different simple kernels with a grayscale image.

Convolutional Layers A convolutional layer applies one or more kernels to an input and produces one feature map output per kernel. By sliding a kernel over regions of the input, an output feature map is created, consisting of values produced by the convolution operation between the kernel and the input regions. Figure 6 illustrates a convolution operation between a single channel input and a 3×3 kernel. The resulting output will have a reduced height and width, which can be prevented by adding padding values to the edges of the input, where the number of padding values to add depends on the kernel shape. The output shape is also dependent on the stride with which one slides the kernel over the input.

Let



Gauss Kernel Sobel X Kernel Sobel Y Kernel

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} \quad \begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix} \quad \begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{bmatrix}$$

Figure 5: *Effects of different kernel matrices when convolved with an image. From left to right: the Gauss kernel used for blurring, Sobel X kernel highlighting horizontal edges, and Sobel Y kernel highlighting vertical edges.*

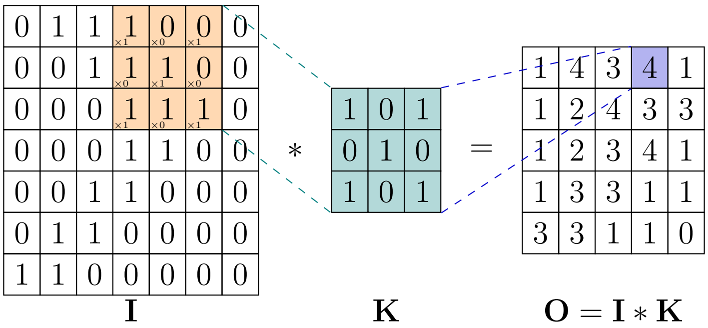
- W_{in} and H_{in} be the width and height of the input
- W_{out} and H_{out} be the width and height of the output
- K_w and K_h be the width and height of a kernel
- S be the stride
- P be the padding
- F be the number of Kernels used

The output shape of a convolutional layer can be calculated as

$$W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} - K_w + 2P}{S} \right\rfloor + 1 \tag{12}$$

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} - K_h + 2P}{S} \right\rfloor + 1 \tag{13}$$

The output channels C_{out} will be equal to the number of filters F applied in the convolutional layer. Therefore, the final output shape will be $(H_{\text{out}}, W_{\text{out}}, C_{\text{out}})$. By stacking multiple convolutional layers in a CNN, the region of influence from the original input that affects the neuron activations in subsequent layers increases. This region of influence is called the receptive field. The size of the receptive field at a given convolutional layer is a useful metric for determining how much of the input is contributing to the



$$\begin{array}{ccccccc}0&1&1&1&0&0&0\\0&0&1&1&1&0&0\\0&0&0&1&1&1&0\\0&0&0&1&1&0&0\\0&0&1&1&0&0&0\\0&1&1&0&0&0&0\\1&1&0&0&0&0&0\end{array} * \begin{array}{ccc}1&0&1\\0&1&0\\1&0&1\end{array} = \begin{array}{ccccc}1&4&3&4&1\\1&2&4&3&3\\1&2&3&4&1\\1&3&3&1&1\\3&3&1&1&0\end{array}$$

Figure 6: *Illustration of the convolution operation. By sliding kernel K over the input I and performing a convolution operation at every position, output feature map O is created. Adapted from [21].*

feature maps produced by the layer. Research suggests, that the *effective receptive field* differs from the theoretical receptive field, with central pixels contributing more to a neurons activation in later layers than pixels from the input perimeter [20]. The receptive field R_n at layer n is calculated as

$$R_n = R_{n-1} + (K_n - 1) \prod_{i=1}^{n-1} S_i \tag{14}$$

where R_{n-1} is the receptive field of the previous layer, K_n is the kernel size of the current layer and S_i is the stride of the $i-th$ layer. The initial receptive field R_0 is 1, since each pixel in the input image can be considered to be its own receptive field. This calculation does not account for dilated convolutions.

Pooling Layers In addition to convolutional layers, CNNs often incorporate pooling layers. A pooling layer serves multiple purposes, with the two main goals being dimensionality reduction and feature abstraction. In a pooling layer, a custom sized window is being slid over each feature map, computing a single value at each position of the window and producing an output feature map with reduced width and height dimensions. The most commonly used operation in a pooling layer is max pooling, where only the maximum value of the window at a certain location in the input feature map is preserved. By reducing the shape of feature maps, pooling increases model efficiency, reducing the number of parameters in subsequent layers and therefore enables deeper model architectures. Each pooling layer also increases the receptive field of the following layers without adding extra model parameters. While max pooling layers lead to some information loss, they reduce the impact of noise in the input and provide small amounts of translation invariance by retaining only the highest values in a feature map and disregarding smaller details.

Output Layer When designing CNNs for classification and regression tasks, a typical architecture will stack multiple convolutional layers and pooling layers, followed by one or more fully connected, also known as dense, hidden layers, and finish with a dense output layer, where the number of output neurons depends on the number of predicted target features. While the weights of all layers are adjusted during the optimization steps at training time, one can regard the convolutional blocks as feature extractors and the dense layers as performing a task-specific integration of the extracted features. It is common to use a squashing non-linear activation function in the output layer that transforms the predicted targets to their expected value range. For multiclass classification tasks, the number of output neurons typically corresponds to the number of possible target classes and the softmax activation function is used to convert the vector of logits (raw real numbers) into a probability distribution of normalized positive values that sum to one. Each output can now be considered as the probability of the input belonging to the class of the corresponding index. For binary classification, one can either use two output neurons with a softmax activation or a single output neuron with the sigmoid activation. For regression tasks, the number of output neurons typically corresponds to the number of predicted targets and the activation function depends on the valid value range (e.g. hyperbolic tangent activation for targets in the range $[-1, 1]$).

The softmax activation function is calculated as

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{15}$$

for a real valued target vector $\mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$.

2.3 Overview of Autoencoders

Autoencoders (AEs) are models capable of learning lower-dimensional data representations in an unsupervised manner. A typical autoencoder consists of two distinct components, an encoder, that compresses inputs x into a lower dimensional latent representation z, and a decoder, which reconstructs the original inputs from the latent representations as $\hat{x}$. The autoencoder loss function quantifies the quality of the reconstruction and during training, the gradients flow through both the decoder and encoder components of the autoencoder. Autoencoders are commonly used for tasks such as compression, denoising, feature extraction and as a backbone for classification or regression models. Bourlard and Kamp were the first to describe an autoencoder like neural network in 1988 [22] which they called *"AutoAssociator"*, and demonstrated that such a network can be trained without using nonlinear activations to perform dimensionality reduction. Baldi and Hornik made a connection between autoencoders and Principal Component Analysis (PCA), showing in their work from 1989 [23] that autoencoders using simple linear transformations without nonlinear activation functions and Squared Error loss essentially perform PCA. Two years later, in 1991, Kramer expanded on the concept and introduced nonlinear sigmoid activation functions in both the encoder and decoder networks [24], thereby creating the first iteration of what is now commonly referred to as an autoencoder.

Conceptually, an autoencoder can be expressed as a sequence of two functions that

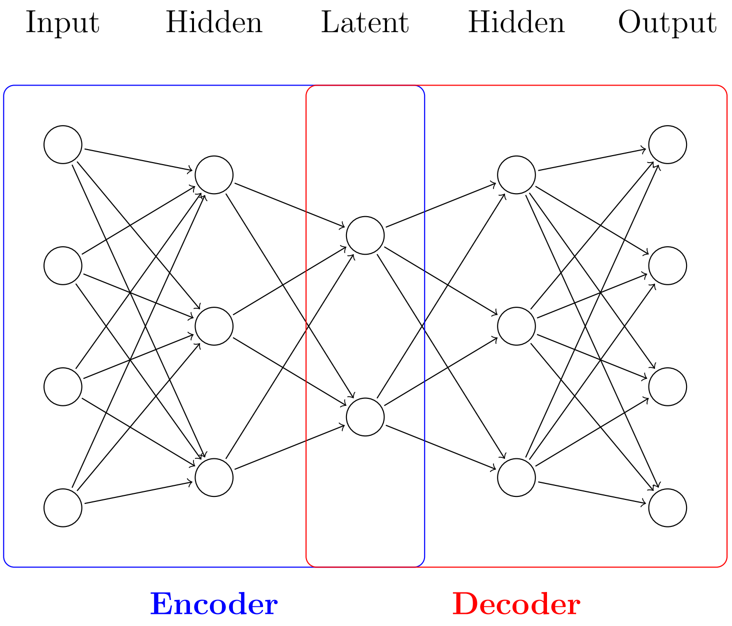


Figure 7: *A simplified autoencoder feedforward neural network. The encoder (blue) compresses the input into a lower dimensional latent representation (middle). The decoder (red) reconstructs the original input from the latent representation.*

map a data point from the input space to a latent space and back:

$$\begin{aligned} & x \xrightarrow{f} z \xrightarrow{g} \hat{x} \\ & z = f(x) \\ & \hat{x} = g(z) = g(f(x)) \end{aligned}$$

where x is the input, z is the compressed latent representation, $\hat{x}$ is the reconstructed output, $f(x)$ represents the encoder mapping x to a latent representation and $g(z)$ represents the decoder mapping the latent representation z back to the input/output space. In this thesis, we focus on autoencoders implemented as feedforward neural networks. A simplified autoencoder architecture is depicted in Figure 7. By using deeper neural networks with nonlinear activations, we can train autoencoders to learn complex patterns and nonlinear dependencies in the input data to achieve a higher rate of compression in the latent space.

Encoder The encoder model transforms inputs into latent representations. Since the latent space is typically designed to have fewer dimensions than the input space, the encoder learns to extract salient features and disregard redundant information in samples from the input space. The number of hidden layers in the encoder and the dimensionality of the latent space are important hyperparameters of the encoder model. After the full autoencoder training is completed, the encoder component can be re-used and adapted as a feature extractor for tasks like classification, regression and clustering. While it is possible to use the encoder directly for downstream tasks, it can be beneficial to fine tune the weights of lower layers in the encoder on a task-specific dataset after the autoencoder training has been completed. Since we want two downstream models to share one autoencoder based feature extractor, we will not be fine-tuning any autoencoder layers after the initial unsupervised training.

Latent Space Representation The latent space, also called the bottleneck or the code layer, represents the compressed outputs generated by the encoder model. Encoded data points are commonly called latent representations, latent variables or latent vectors. Traditionally, the latent space is designed to have fewer dimensions than the task specific input space, however exceptions to this custom exist in the form of sparse autoencoders. When the latent space has fewer dimensions than the input space, the corresponding autoencoder model is said to be *under-complete*. In this work, we focus exclusively on under-complete autoencoders. By constraining the latent space dimensionality, the encoder model is incentivized to learn meaningful disentangled features and discouraged to learn the trivial identity mapping between input space and latent space. The latent space representations can also be used to effectively visualize relations within a dataset, for example by showing samples belonging to a distinct class being grouped closely in the lower dimensional space [25].

Decoder The decoder model transforms latent representations back to data points in the original input space, therefore expanding the dimensionality of compressed samples in the case of under-complete autoencoders. The quality of the reconstruction is a common criterion in autoencoder loss calculation, affecting both the encoder and decoder model weights. When the original input values come from a well-defined range, e.g. pixel values in the range of $[0, 1]$ for normalized images, then a squashing nonlinearity (e.g. sigmoid) is often applied as the activation function in the last layer of a decoder. Vincent et al. recommend refraining from using a squashing nonlinearity in the decoder when using the MSE loss for reconstructing inputs under the Gaussian noise assumption [26].

2.3.1 Loss Function

The choice of loss function for an autoencoder model depends heavily on the type of data used for training and inference. Since the loss is used to quantify the similarity between the original input sample x fed to the encoder and the reconstruction $\hat{x}$ produced by the decoder, it is colloquially referred to as the reconstruction loss. Let us briefly review some of the most commonly used loss functions in autoencoder architectures that operate on image inputs.

Mean Squared Error The MSE loss function is a common choice for regression tasks involving real valued target vectors and pixel-based image data. If we assume that the target values y are produced by some true function of the input $f(x)$ with added Gaussian noise with zero mean and constant variance, then minimizing the MSE maximizes the likelihood.

$$\hat{y}_i = f(x_i) + \varepsilon_i, \text{ where } \varepsilon_i \sim \mathcal{N}(0, \sigma^2) \tag{16}$$

$$\log L(\hat{y}|y, \sigma^2) = n \log \left(\frac{1}{\sqrt{2\pi\sigma^2}} \right) - \frac{1}{2\sigma^2} ||y_i - \hat{y}_i||^2 \tag{17}$$

$$\mathcal{L}_{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{18}$$

We can see that ignoring the constant terms that do not involve x in Equation (17) leads us to the MSE loss defined in Equation (17). Here one can notice an important characteristic of the MSE loss function when working with image data, namely that by assuming a Gaussian target distribution for pixels, models fitted with the MSE loss will tend to produce blurry outputs by predicting pixel values close to the mean. We evaluate if this effect is observable when training on our dataset in the experimental section of this thesis.

Binary Crossentropy The BCE loss function is typically used when working with Bernoulli distributed data, i.e. target values that are either 0 or 1. However, we can extend the use of the BCE loss to working with grayscale images where pixel values are bounded to the range $[0, 1]$. In this case, the pixel values can be regarded as intensities or probabilities. The BCE performs Maximum Likelihood Estimation (MLE) for Bernoulli distributed data:

$$\log L(\hat{y}|y) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}) \tag{19}$$

$$\mathcal{L}_{BCE} = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)] \tag{20}$$

We shall investigate if the BCE loss produces notably different reconstructions compared to the MSE in our experimental setup.

Structural Similarity Index Structural Similarity Index Measure (SSIM) is a measure that accounts for perceived similarities in luminance, contrast, and structure between multiple regions of two images [27]. It is designed to better match human perception by taking local structural information into account. In contrast to MSE and BCE, it calculates local statistics within a sliding window and does not compare individual pixels. As such, the SSIM is less susceptible to small changes in individual pixels and minor translations and rotations, as long as they preserve structure. The SSIM between two regions of an image x and y is computed as

$$\mathrm{SSIM}(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{21}$$

where

- μ denotes the pixel mean
- σ^2 denotes the variance
- σ_{xy} denotes the covariance of x and y
- L denotes the dynamic range of pixel values
- $c_1 = (k_1L)^2$, $c_1 = (k_1L)^2$
- $k_1 = 0.01$ and $k_2 = 0.03$ per default

When using SSIM as a loss metric, we adjust the score such that a lower score represents a better match.

$$\mathcal{L}_{SSIM} = 1 - \mathrm{SSIM}(x,y) \tag{22}$$

The size of the sliding window is an important hyperparameter and depends on the overall image size. Due to the sliding window and statistical calculations, SSIM is more computationally expensive than the previously introduced loss functions. SSIM is well suited as a loss function for grayscale images, and thus we evaluate its efficacy for autoencoder training alongside MSE and BCE.

Focal Loss Focal Loss [28] is an extension of Crossentropy loss that can address class imbalances via the α parameter and give more weight to hard-to-classify samples via the γ parameter. We use it as a loss function for the ball classification models, not as a reconstruction loss in autoencoder training. It was originally proposed to improve the training of one-shot object detector architectures and has been successfully used in soccer ball patch detection by team B-Human with adjustments [29].

$$\mathcal{L}_{FC_\alpha}(y,\hat{y}) = -\frac{1}{n}\sum_{i=1}^{n}[\alpha y_i(1-\hat{y}_i)^\gamma \log(\hat{y}_i) + (1-\alpha)(1-y_i)\hat{y}_i^\gamma \log(1-\hat{y}_i)] \tag{23}$$

Intersection over Union The Intersection over Union (IoU) is a measure of overlap between two bounding boxes or segmentation masks. It is commonly used as a metric for object detection models, and when used as a loss function, one minimizes $1 - \mathrm{IoU}$. We use this loss when training ball position regression models. The IoU calculates the area of the intersection divided by the area of the union of the two regions. The IoU ranges from 1 (perfect overlap) to 0 (no overlap). When the IoU is zero, it becomes non-differentiable during model training, which can lead to unstable initial training periods. Therefore, an IoU based loss often includes additions such as the distance between bounding box centers. The vanilla IoU loss is given by

$$\mathcal{L}_{\text{IoU}}(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{24}$$

Maximum Likelihood Estimation The BCE and MSE loss functions we introduced in Section 2.1 can be derived from MLE [30]. To find the parameters θ of a model that maximize the likelihood of observing training data x, we can minimize the negative log-likelihood $-\log p(x|\theta)$. The logarithm of the likelihood is used due to the convenience of products becoming sums, as well as numerical stability when dealing with small probabilities. Since the log-likelihood is monotonically related to likelihood, minimizing the negative log-likelihood leads to a maximization of the likelihood. The maximum likelihood estimate for the model parameters θ is given by

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{n} \log p(x_i; \theta) \tag{25}$$

where n is the number of training samples x and x_i is the i-th training sample.

2.3.2 Convolutional Autoencoders

Convolutional Autoencoders (CAEs) extend the concept of autoencoders to the image processing domain. By using a CNN based encoder, image-like inputs, i.e. tensors of shape $(height, width, channels)$, are compressed to a latent representation through a series of convolutional layers. During training, the encoder learns a hierarchical set of kernels, which extract the most salient features from inputs. The decoder, tasked with reconstructing the original inputs from their latent representations, performs transposed convolutions. In a typical CAE architecture, the encoder's layers generate feature maps with progressively decreasing spatial dimensions and increasing channel depth, while the decoder reverses this process, producing feature maps with expanding spatial dimensions and reduced channel depth. If the encoder includes downsampling operations such as strided convolutions or pooling layers, then the decoder can perform a corresponding upsampling either through strided transposed convolutions or upsampling layers that perform interpolative resizing.

It is important to note that both convolution and pooling operations performed in the encoder cannot be truly inverted, only approximated, in the decoder, due to loss of information in non-unique mappings. The transposed convolution is not the inverse operation of a convolution, instead it attempts to reconstruct a plausible input that could have led to the given output of a convolutional layer. It uses zero-padding at the input borders and inserts zeroes between the elements of the input feature map, then performs a convolution with learnable filter weights. Masci et al. advocate for the usage of max pooling layers in CAEs, showing that the learned filter weights become more plausible and generalized compared to models not using max pooling layers in their experiments [31]. Figure 8 shows a simplified convolutional autoencoder architecture.

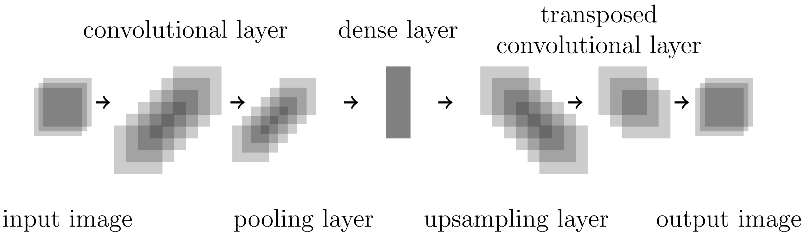


Figure 8: *Simplified architecture of a convolutional autoencoder. The encoder consists of a convolutional layer and a pooling layer. A dense layer represents the latent space. The decoder mirrors the encoder structure with an upsampling layer and a transposed convolutional layer to reconstruct the input. Arrows indicate the flow of information through the network.*

A common application of CAEs is using the weights of a trained encoder as the initial weights for a CNN with the same architecture [16][32][33].

In this work, we examine the convolutional autoencoder and variations of the convolutional autoencoder architecture in more detail. Due to its ability to effectively work with image-based inputs, the model architecture is well suited for experimentation on the unlabeled image data collected by the Berlin United RoboCup team.

2.3.3 Denoising Convolutional Autoencoders

The Denoising Convolutional Autoencoder (DCAE) is a simple, yet effective extension of the autoencoder architecture [26][33][34]. A denoising autoencoder is tasked with learning to reconstruct a clean version of a *corrupted* input, which ideally results in a more robust and generalized feature extraction performed by the encoder. Common forms of image corruption techniques include:

- Additive *Gaussian noise*: For each input sample x, add a noise vector $n_i \sim \mathcal{N}(0, \sigma^2)$ drawn from a zero mean normal distribution
- *Masking Noise*: For a fraction of the elements of an input sample x, set the value of the element to 0
- *Salt-and-Pepper Noise*: For a fraction of the elements of an input sample x, randomly set the value of the element to its minimum or maximum value

The Gaussian noise model is an intuitive choice when working with images recorded by the Nao robot, since camera noise is typically modelled as additive Gaussian noise resulting from thermal and electromagnetic radiation in the camera sensor and its surrounding circuitry. Salt-and-Pepper noise can also occur in digital images, often stemming from analog-to-digital conversion errors. Masking noise has been a staple in past work on denoising autoencoders [26], partly due to its intuitive effects during model training. By setting some elements of the input samples to 0, they are effectively *missing*, and the model hast to infer the missing values based on surrounding features.

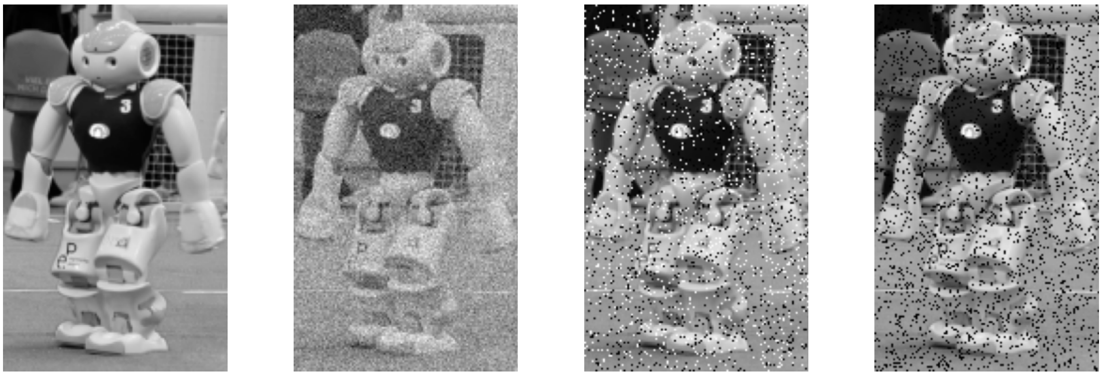


Figure 9: *Effects of corruption techniques on an image. From left to right: uncorrupted grayscale image, image with added Gaussian noise ($\sigma = 0.1$), image with added salt-and-pepper noise ($P(noise) = 0.1$) and image with added masking noise ($P(noise) = 0.1$).*

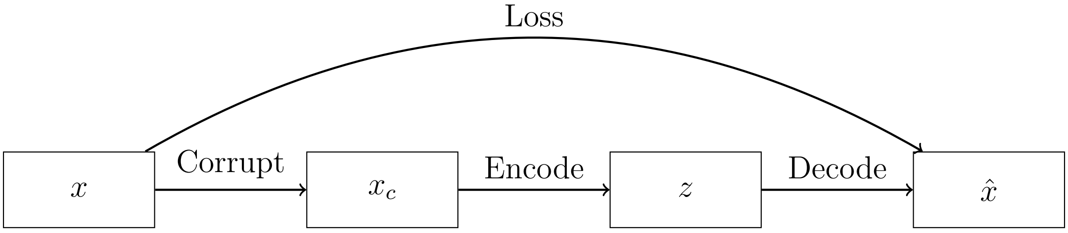


Figure 10: *Simplified graph representation of a Denoising Autoencoder architecture. The network is tasked with reconstructing an uncorrupted sample $\hat{x}$ from a latent representation z of a corrupted sample x_c. The loss function compares the reconstruction with the original, uncorrupted, input x.*

2.3.4 Convolutional Variational Autoencoders

The Convolutional Variational Autoencoder (CVAE) is a generative model, extending the concept of Convolutional Autoencoders to the probabilistic domain of variational Bayes [35]. In contrast to the deterministic autoencoder, which will always map the same input to the same latent representation, the CVAE instead encodes samples as distributions in the latent space. The encoder performs the opposite operation, decoding the original sample from the latent space according to a distribution. The distribution is typically modeled to be a multivariate Gaussian to allow for the joint training of encoder and decoder, enabling gradients to flow through both networks by employing the reparametrization trick, which will be explained shortly.

The CVAE architecture necessitates the addition of a new term to the loss function, which ensures the learned parameters of the latent distribution model the assumed prior distribution [36]. By regularizing the latent space in this way, CVAEs learn to map similar inputs to similar distributions in the latent space. Furthermore, a trained CVAE can be used as a generative model to create novel and plausible image samples. Additionally,

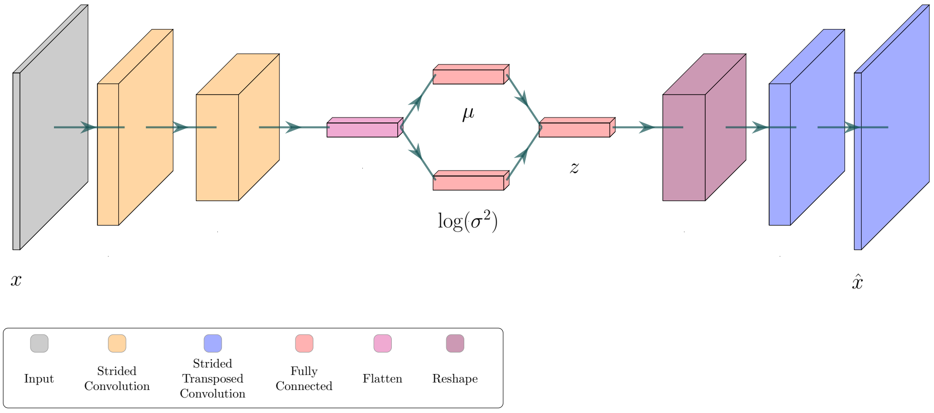


Figure 11: *A simplified CVAE architecture. The strided convolutions (orange) in the encoder increase the number of channels while down scaling the width and height in the resulting feature maps. The latent representation z is sampled using the learned mean and log variance weights, implemented as fully connected layers (magenta). The decoder performs strided transposed convolutions, thereby increasing the width and height and decreasing the number of channels in their resulting feature maps. The reconstruction $\hat{x}$ has the same dimensions as input x. The height and depth of the blocks in the figure represent the width and height of the feature maps they output, while the width of the blocks represent the number of channels.*

a trained CVAE can be used to cluster samples. Figure 11 depicts a simplified CVAE architecture.

During CVAE training, we want to find both good model parameters θ that maximize the likelihood $p(x|\theta)$, and the posterior distribution over latent variables $p(z|x, \theta)$.

Evidence Lower Bound When training a CVAE, we need to approximate the posterior distribution over latent variables z given x, so MLE as described in Section 2.3.1 alone is not sufficient. To find the marginal likelihood $p(x|\theta) = \int p_\theta(x|z)p(z)dz$, we need to integrate over all possible values of z, which is typically high dimensional and therefore intractable. To make the marginal likelihood estimation feasible for training CVAEs, we use the encoder model to approximate the true posterior distribution $p_\theta(z|x)$ as $q_\phi(z|x)$, where ϕ are the trainable parameters of the encoder. Now we can define a lower bound on the marginal likelihood, the Evidence Lower Bound (ELBO), to be optimized during training:

$$log(p(x|\theta)) = E_{q_\phi}(z|x)[log(p(x|\theta))] \tag{26}$$

$$log(p(x|\theta)) \geq E_{q_\phi}(z|x)[log(p_\theta(x, z)) - log(q_\phi(z|x))] \tag{27}$$

$$ELBO(\theta, \phi) = E_{q_\phi}(z|x)[log(p_\theta(x|z)) + log(p(z)) - log(q_\phi(z|x))] \tag{28}$$

$$ELBO(\theta, \phi) = E_{q_\phi}(z|x)[log(p_\theta(x|z))] - KL(q_\phi(z|x)||p(z)) \tag{29}$$

From Equation (29), we can identify all relevant parts of the CVAE Neural Network architecture:

- $q_\phi(z|x)$ is the posterior, represented by the encoder model
- $p_\theta(x|z)$ is the likelihood, represented by the decoder model
- ϕ are the learnable encoder parameters
- θ are the learnable decoder parameters
- $E_{q_\phi}(z|x)[log(p_\theta(x|z))]$ is the expected log-likelihood of the data given the latent variables, also called the reconstruction term
- $KL(q_\phi(z|x)||p(z))$ is the Kullback-Leibler Divergence between the approximated posterior and the prior of the latent variables z

Based on the the ELBO we derive the loss function for model training as

$$\mathcal{L}_{CVAE}(x, \hat{x}, z) = \mathcal{L}_{MSE}(x, \hat{x}) + KL(z, \mathcal{N}(0, \sigma^2)) \tag{30}$$

The Kullback-Leibler Divergence is a distance measure that quantifies the similarity of two distributions. For the typical case of the prior distribution for z being the multivariate Gaussian, the Kullback-Leibler Divergence can be computed in the loss function as

$$KL(z, \mathcal{N}) = -\frac{1}{2}\sum_{i}^{n}(1 + log(\sigma_{zi}^2) - \mu_{zi}^2 - e^{log(\sigma_{zi}^2)}) \tag{31}$$

The loss function matches nicely with the intuition we formulated for the CVAE, with the encoder learning to compress the input samples into a meaningful latent representation, which is now being regularized to follow a prior in the form of a Gaussian, and the decoder learning to reconstruct the original input samples from the latent representation. In practice, the KL loss term is often weighted by a factor β to prevent the network from mode seeking behavior or posterior collapse. Mode seeking behavior occurs when the approximate posterior $q_\phi(z|x)$ concentrates on a single mode of the true posterior $p(z|x)$, resulting in less diverse reconstructions and a suboptimal latent representation. Posterior collapse describes a scenario in which the output of the decoder network becomes mostly independent of the latent representation z and tends toward reconstructing basic patterns that do not represent any individual sample from x well. A posterior collapse occurs when the influence of input samples x on the latent representation z becomes weak or noisy, and the learned parameters μ and σ^2 tend towards constant values, regardless of the input. Both phenomena can be combated by decreasing the contribution of the KL term during model training, especially at the beginning of the optimization process. When using the KL loss in conjunction with the MSE loss, which takes on rather low error values, especially when the input data is scaled to a range between 0 and 1, it is sensible to downscale the KL loss contributions by multiple factors of 10 to prevent posterior collapse.

Reparametrization Trick If we define the latent space to be n-dimensional, the encoder outputs two n-dimensional vectors, a mean vector and a log variance vector, which parametrize the prior distribution. To get a single n-dimensional vector z that encodes a data point x, we sample from the distribution using the reparametrization trick

$$z = \mu + \sigma \odot \varepsilon \tag{32}$$

Where ε is a random vector sampled from a standard normal distribution $\mathcal{N}(0, I)$, $\odot$ denotes element-wise multiplication and σ is the element-wise square root of the variance. The reparametrization trick allows gradients to flow through the entire model during training, which makes it a necessity for gradient-based optimization techniques such as backpropagation [37].

2.4 Ball Detection in RoboCup

Detecting a soccer ball and accurately estimating its position in the field is an essential task for soccer playing robots in the RoboCup SPL. By assessing how other successful teams tackle this challenge, we can collect insights that inform the design choices of our autoencoder architectures. Since all teams in the SPL work on the same Nao robot platform and therefore face similar constraints on runtime and performance, we can gather useful information on how to design, train and run models on the Nao efficiently.

B-Human The B-Human team from Bremen has a proven track record of successful performance in the RoboCup and German Open soccer competitions, and with a recent winning result in RoboCup 2024, inspires many other teams who use their code base as a starting point for their development efforts. B-Human uses a CAE network that encodes ball patch images and decodes a segmentation mask for the ball area in the reconstruction [38]. This approach requires annotation of the autoencoder training data for the ball class with ground truth segmentation masks. After training the autoencoder, the decoder model is disregarded and the encoder forms the basis for a classification and a ball center and radius regression model, both using the latent representations produced by the encoder as inputs. For the upper Nao camera, they use a scanline based hypothesis generation to identify potential ball patches by looking for ball sized, high contrast areas surrounded by green pixels representing the soccer pitch. For the bottom camera, they use a neural network inspired by the ResNet [18] architecture to generate region proposals for potential ball patches. Patches that were classified to be soccer balls are finally processed by a regression model that estimates the ball center and radius. Here the latent representation produced by the encoder is reused as an input to the regression model.

HTWK Leipzig Team HTWK Leipzig, the RoboCup 2024 vice champion team, uses a two-step approach, identifying hypothesis regions for potential ball candidates which are subsequently classified by a CNN [39]. To generate ball hypothesis for the upper camera, they first compute an integral image and then evaluate blocks of the estimated ball size at the current position. By comparing the blue-difference channel values (Cb channel in the YCbCr color space) inside and outside the proposed region, they find patches with high values for black and white pixels inside the region and green pixels outside the region. The resulting candidate patches are classified by a small CNN with only two convolutional layers. For the bottom camera, they use a downsampled 40x30 pixel version of the YUV422 camera image to generate a single best candidate location coordinate prediction using an InceptionV2 [40] inspired neural network, which is subsequently classified by the aforementioned small CNN. In their report, the HTWK team does not mention the use of a dedicated regression model that estimates the ball center and radius, which suggests that they use the positional information of the hypothesis patches to estimate the ball location.

HULKS The HULKS team from Hamburg implements an interesting version of the two-step ball detection process. To generate ball hypothesis, they create grayscale patches from an image grid, excluding all regions that have been determined to be field lines or soccer pitch by other modules [41]. The resulting candidates are processed by two distinct CNNs, with the first network being optimized for recall and rejecting clear negative samples, and the second network being optimized for precision to accurately identify samples from the positive ball class. If both networks predict a patch to belong to the ball class, then a third regression model is used to estimate the ball center and radius. In contrast to the B-Human approach, no intermediate features are reused between the networks.

Nao Devils The Dortmund Nao Devils Team developed a CNN architecture optimized for inference speed on the Nao robot platform, by combining depthwise separable convolutions with the concept of early exiting [42]. The early exit approach enables the network to reject many samples after the first convolutional block, thereby skipping all subsequent calculations in the following layers. They also specifically choose network parameters like the number of filters in each convolutional layer to take advantage of the available Single Instruction Multiple Data (SIMD) instructions of the Nao robot's Central Processing Unit (CPU). The Nao Devils use a scanline based approach similar to B-Human for ball candidate hypothesis generation on both cameras.

Berlin United The Humboldt University Berlin United Team uses the same broad two-step approach employed by the previously mentioned teams. We use a heuristic to find promising ball candidate patches based on the estimated ball size and brightness variations in the parts of the camera image we previously determined to be inside the soccer pitch [3]. We know that all soccer balls are black and white and are of a fixed size, so we can use the robot's camera matrix to estimate the size of a soccer ball at any given point in the image. To narrow down the possible candidates, we evaluate square bounding box patches with a side length of the estimated ball radius and compute a difference in brightness of pixels inside the estimated ball area and pixels outside the estimated ball area (using the Y-channel of the YUV candidate patch). We expect actual ball patches to have a higher average pixel brightness inside the ball area than outside of it. We then find the candidate patch with the maximum brightness difference in its neighborhood, discarding overlapping patches with a lower score. Figure 12 shows a recorded game log being replayed in our *RobotControl* debug software, with ball patch candidates overlaid as colored rectangles. The resulting ball candidate patches are classified by a CNN classifier.

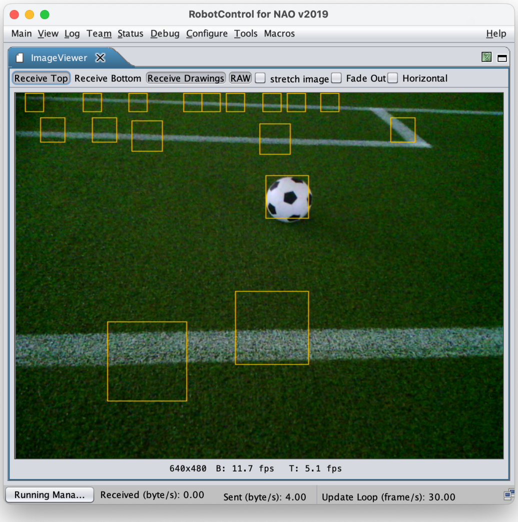


Figure 12: *Visualization of ball hypothesis generation during gameplay. The image shows a recorded image log from the top camera being replayed in the RobotControl debug software, with ball candidates overlaid as yellow rectangles.*

Until the German Open Event in April 2024, we used a single CNN model to predict the ball class confidence and the estimated ball center and radius in one forward pass. Since April, we have used a dedicated CNN to classify ball candidate patches and a second CNN to estimate the ball center and radius for patches that have been classified as a ball. In this work, we examine the efficacy of using an autoencoder to extract features that can be shared by a dedicated classification network and a position estimation regression network.

3 Experiments

The experiments presented in this section systematically evaluate and optimize the design aspects of autoencoders and work toward an optimized autoencoder based feature extraction architecture for soccer ball detection and localization in RoboCup Soccer. Our goal is to develop an autoencoder-based approach that can efficiently and accurately process ball candidate patches in the resource-constrained environment of the Nao humanoid robot. We conduct experiments at three levels of increasing specificity. We begin by examining the fundamental building blocks of convolutional autoencoders in Section 3.5, including loss functions, latent space dimensionality, pooling and downsampling strategies, upsampling techniques, batch normalization, and regularization methods. These experiments provide insights into the most effective basic components for our specific task of encoding low-resolution grayscale ball candidate patches. Building on the insights we gained, we compare different autoencoder architectures, including standard Convolutional Autoencoders, Denoising Convolutional Autoencoders and Convolutional Variational Autoencoders in Section 3.6. This comparison is intended to identify the most suitable overall architecture for our ball detection task. Finally, we evaluate the best-performing autoencoder architectures by using them as feature extractors for downstream tasks. Specifically, we assess their effectiveness in ball patch classification in Section 3.7 and ball position estimation within patches in Section 3.8. The following sections detail our experimental setup, data acquisition and evaluation methodology, before moving on to the specific experiments and their results.

3.1 Methodology

This section outlines the general approach and procedures used across all experiments in this chapter. We employed a systematic approach to evaluate various aspects of autoencoder design and architecture for the specific task of using autoencoders as feature extractors for soccer ball detection and localization in RoboCup Soccer.

Data Collection and Preprocessing We use log data collected from Nao robots during RoboCup matches and practice sessions. An existing data processing pipeline was extended and used to generate ball candidate patches from raw images. All patches underwent consistent preprocessing, including grayscale conversion, resizing to a width and height of 16×16 pixels, pixel value scaling, and samplewise brightness normalization. The data acquisition process and preprocessing steps are explained in more detail in Section 3.2.2.

Experimental Design We conduct a series of experiments at three levels of increasing specificity:

1. Evaluation of fundamental autoencoder components (loss functions, latent space dimensionality, downsampling and upsampling techniques, batch normalization and regularization)

2. Comparison of different autoencoder architectures (CAE, DCAE, CVAE and Denoising Convolutional Variational Autoencoder (DCVAE))
3. Assessment of practical utility of trained autoencoders as feature extractors for ball classification and position estimation models

When evaluating the autoencoder components in the first series of experiments, we use a baseline autoencoder architecture that remains fixed across all experiments, except for one design aspect that is subject to change according to each experiment.

For experiments at levels one and two, we use a subset of all available image data composed of a total of 10 games and practice sessions. Statistics are computed from 10×10 cross-validation trials, as described in Section 3.3. The most promising architectures emerging from the second round of experiments are trained on a complete dataset of available image patches. In the final set of experiments, we create novel datasets for classification and ball position estimation. The datasets are described in greater detail in Section 3.2.2.

Model Training All models are implemented using the Tensorflow/Keras [43] machine learning framework. All autoencoder models are implemented as custom *Keras* model classes. By default, we use the Adam [44] optimizer with a learning rate of 0.0005 and a batch size of 32 samples per batch. Training data is split into training and validation subsets, with 85% of data used for training and 15% used for validation. Early stopping is employed to prevent overfitting by monitoring the validation loss [45]. The learning rate is reduced dynamically by monitoring plateaus in the training loss. The loss function, number of training epochs, and patience hyperparameter for early stopping and learning rate reduction are based on the experiment and dataset used and are provided where applicable. Training metrics and artifacts are logged to an *MLFlow* metrics and operations server instance running in the Berlin United Kubernetes cluster for posterity. This enables us to reconstruct and analyze past experiments and store trained model instances for posterity.

Evaluation Metrics To evaluate autoencoder reconstruction ability, we primarily use the Mean Squared Error (MSE), but other loss functions are evaluated as well, which will be indicated in the description of the corresponding experiments.

For binary classification tasks, we focus on metrics related to the positive class, in our case the soccer ball, since it is one of the most important objects to detect in robotic soccer. The following metrics are calculated and compared across classifiers:

- Precision $= \frac{TP}{TP+FP}$
- Recall $= \frac{TP}{TP+FN}$
- F1-Score $= 2 \cdot \frac{\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}$

Where TP are true positive predictions, FP are false positives, FN are false negatives and TN are true negatives. For the binary ball classification task, we design all models to use two output neurons with a softmax activation function. This allows us to easily adapt our experiments for multi-class tasks in future research. After model training is completed, we can adjust the thresholds that a prediction score needs to exceed in order to classify an instance as positive. This allows us to fine-tune the model's performance and balance between precision and recall.

Computational Resources Experiments were conducted on the Humboldt Universities compute resources, using servers with two Xeon 6254 or two Xeon 6354 CPUs with 72 hardware threads each and clock speeds of 3,1GHz and 3,6GHz respectively. Additionally, we used an Nvidia RTX A4000 Graphics Processing Unit (GPU) with 16 GB GDDR6 memory from the Berlin United team to run containerized workloads. By balancing training workloads across multiple host machines and employing parallelization techniques, we were able to run cross-validated experiments involving multiple candidate architectures within the span of one and three days, depending on the dataset and number of epochs of training per cross-validation fold.

3.2 Data Acquisition

During gameplay, every active robot collects log files, which are retrieved and stored after the match or experiment has ended. Our robots record three distinct log files, a game log containing game state, robot state and behavior information, a sensor log containing recorded sensor values and an image log containing a subset of the raw YUV422 images recorded by the top and bottom cameras. At the time of writing, we log images every two seconds. Currently, a JPEG based asynchronous logger with greater temporal resolution is being developed. We store the unmodified log data and make it available for research and development by other RoboCup teams and researchers at the Berlin United team homepage. After the log data of an event or experiment has been collected, we use a data processing pipeline developed by Schlotter [46] to enrich, examine and annotate the data for analysis and use in model training. As part of this work, we made several contributions to the existing data processing suite, as well as developing novel tools, which we describe in more detail in Section 3.4. As part of an ongoing data labeling effort, we use the annotation software tool *LabelStudio* to enhance image data extracted from game logs by adding bounding box annotation for the object classes *ball*, *robot* and *penalty mark*. While the stored image data and annotations from the *LabelStudio* annotation tool can be used for a multitude of different tasks, we focus on the generation of ball candidate patches to be used for autoencoder training. Figure 13 shows a simplified schematic of the processing pipeline. The data processing pipeline for ball patch generation can be broken down into four steps:

1. Combine game log and image log. Enrich the raw images with additional information such as the camera matrix data, camera ID and log file name.

2. Store metadata in a database and images in object storage. Enable easy filtering and composition when creating datasets. Images are transformed from YUV422 to RGB and stored as PNG files for easy storage and viewing. Metadata such as camera matrix information is saved in the PNG file header.

3. Annotate images in Label Studio. Uses a pretrained YOLOv8[47] model to add bounding box information for robots, penalty marks and soccer balls to be verified and adjusted manually.

4. Generate ball candidate patches. Run the ball hypothesis algorithm on the stored images (converted back to the YUV color space), optionally using the annotation information from the *Label Studio* software tool to add additional information to the generated patches stored in the file header.

While it is possible to use the annotation information from the labeling tool to add class labels to the generated ball candidate patches, manual annotation and verification is not needed for autoencoder training, where the image patches themselves are the only data required to train a model. For classification and ball position estimation tasks, we use the annotated bounding box information both to determine the class of a patch (ball or not a ball) and to infer the ball center coordinates and radii of positive samples.

3.2.1 Data Preprocessing

We perform simple data preprocessing on all image patch data before model training. The preprocessing includes resizing to a common patch size, scaling the pixel values to a range suited for gradient-based optimization, as well as samplewise brightness normalization and ensuring the usage of consistent data types. The following preprocessing steps are performed on all images in the experiments described in this chapter:

- Ensure we are working with grayscale images, specifically the Y-Channel of the original YUV color space image. This step is performed once when generating the ball patch candidates.

- Resize the patch to shape *(16, 16, 1)*. During a game, the patches used for ball detection are resized by subsampling the full image. Since the generated ball patch candidates vary in size, we resize all patches to a common shape using nearest neighbor interpolation. This is also performed during ball patch candidate generation.

- Scale the pixel values in the patch from range $[0, 255]$ to range $[0, 1]$ for numerical stability at the optimization step during model training.

- Perform a samplewise brightness normalization by subtracting the total mean brightness value from each individual pixel of a given image patch. This effectively zero centers the dataset and helps convergence during optimization. Preliminary

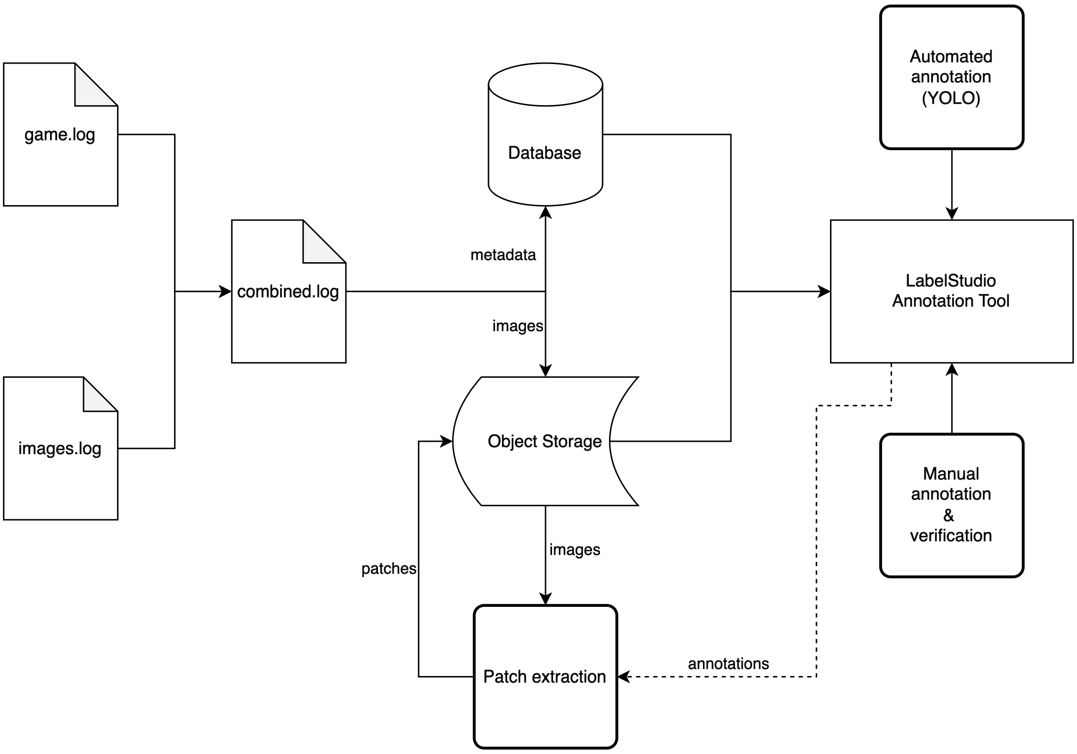


Figure 13: *Simplified schematic of the Berlin United data processing pipeline for ball candidate patch generation. Log data from a single robot is combined, the metadata is stored in a database, and images are persisted in object storage. The annotation tool Label Studio is used to add automated bounding box predictions for the classes robot, penalty mark and ball, which can be adjusted and verified manually. Ball candidate patches can be generated by running the ball hypothesis generation algorithm on the stored images. Optionally, the bounding box annotation data can be incorporated during ball patch generation.*

work on the ball classification CNN networks has shown that brightness normalization increases model performance in every case, regardless of model architecture. This step can be performed efficiently on the robot at inference time, and by using a samplewise normalization instead of computing the mean of the entire dataset, we do not have to track and implement this statistic when deploying different models on the robot.

- Explicitly set the data type to a 32-bit floating point (using the machine learning library *Tensorflow* used for model training) to ensure we use consistent data representations during training and inference.

It is important to apply the same preprocessing transformations during inference to obtain accurate results. Both rescaling and patchwise brightness normalization can be performed easily when constructing the input tensors on the robot. Based on the findings in [46], which suggest that using bigger patches and additional color channels do not

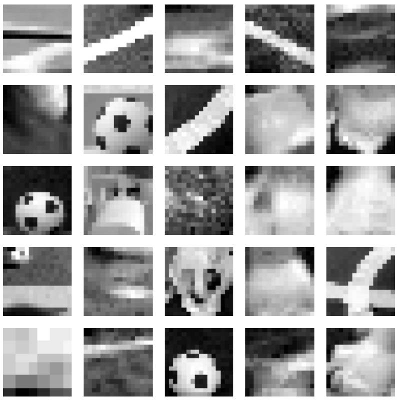


Figure 14: *A random selection of 25 ball candidate image patches used for unsupervised autoencoder training. Image patches have a width and height of 16 pixels and one luminance channel (grayscale).*

increase the performance of our ball classifier model, we consider only grayscale patches with a width and height of 16 pixels to avoid a combinatorial explosion of possible model architecture and dataset pairings.

3.2.2 Datasets

Using the data processing pipeline described in Section 3.2, we can easily create datasets for autoencoder training based on the log data collected by the Berlin United team thus far. In Section 3.5 we examine various components of autoencoders and their effectiveness in our problem domain. Thus, we require a representative sample over various events, lighting conditions and camera calibration settings to make meaningful comparisons. In Section 3.6 we train the most promising candidate architectures for later use as feature extractors and therefore should use all the available data. When training models for ball classification and ball position estimation, we require labeled datasets and therefore manual annotation effort. Figure 14 shows a small representative sample of the candidate patches we use in model training. We consciously refrain from using any historic handcrafted datasets that include image patches recorded by Nao robots version 5, which are not in use in the Berlin United team any longer.

Dataset for Autoencoder Architecture Comparison The dataset for autoencoder architecture comparison consists of ten distinct game halves and experiments collected between the years 2024 and 2021. All data comes from Nao version 6 robots, the most recent iteration of Nao robots available and currently playing in our team. It features images from various venues with distinct lighting conditions and camera calibrations. This dataset is used to compare different autoencoder architecture configurations using 10×10 cross-validation, meaning we perform ten rounds of 10-fold cross-validation, where in each fold we train with a different combination of nine of the available events and evaluate on the remaining subset. By grouping the available data by event instead of combining all

Event	Date	Home Team	Opponent	Samples
Lab Test	17.05.2024	Berlin United	Berlin United	6,748
Lab Test	10.05.2024	Berlin United	Berlin United	5,871
German Open	19.04.2024	Berlin United	Dutch Nao Team	7,085
German Open	19.04.2024	Berlin United	Bembelbots	9,603
German Open	18.04.2024	Berlin United	HTWK	9,358
RoHow	03.12.2023	Berlin United	Unknown	9,681
CCC Camp	17.08.2023	Berlin United	Berlin United	8,308
German Open Remote	09.05.2021	Berlin United	NaoDevils	2,519
German Open Remote	08.05.2021	Berlin United	Bembelbots	1,650
German Open Remote	08.05.2021	Berlin United	R-ZWEI-KICKERS	2,520
			Total Samples	63,343

Table 1: *Dataset for autoencoder architecture selection comprised of 10 distinct events with sample counts. Rows where both home team and opponent are denoted as Berlin United indicate test games where our robots play against themselves or an empty field.*

data and taking random splits, we get a better estimate of model performance on unseen data, since none of the patches used in training, or similar variations thereof, are used in the evaluation step. Table 1 depicts the events comprising the dataset and the number of individual and total samples.

Dataset for Autoencoder Training We prepare a dataset for full autoencoder training that contains all the image data from Nao v6 logs available at the time of writing. Again, we use the game, experiment, or training session name to group all images from an event so we can perform a leave-k-out cross-validation scheme to better estimate the expected reconstruction performance on unseen data. To increase the number of samples during model training, we apply one of five possible image augmentations with a 50% chance. The possible augmentations are a horizontal flip, a vertical flip and a clockwise rotation by 90°, 180° or 270°. Augmentations are applied to the images in a batch dynamically, meaning that over the course of multiple epochs, we present the model both with unaltered samples and with different augmentations of each sample. Augmentations related to image corruption are discussed in Section 3.6 because we only apply those when working with denoising model architectures. The full dataset consists of 33 distinct events totaling 207,014 samples (not factoring in augmentations).

Dataset for Ball Patch Classification We use a subset of all available data for ball patch classification, relying on the annotation information added through the *LabelStudio* labeling tool. In previous experiments, we only considered bounding box annotations that had been manually verified and corrected by members of the Berlin United team from within the *LabelStudio* software. To increase the number of positive samples from the ball class we can use to train classification models, we ran the YOLOv8 object detector on all image logs before exporting ball candidate patches and considered all patches that

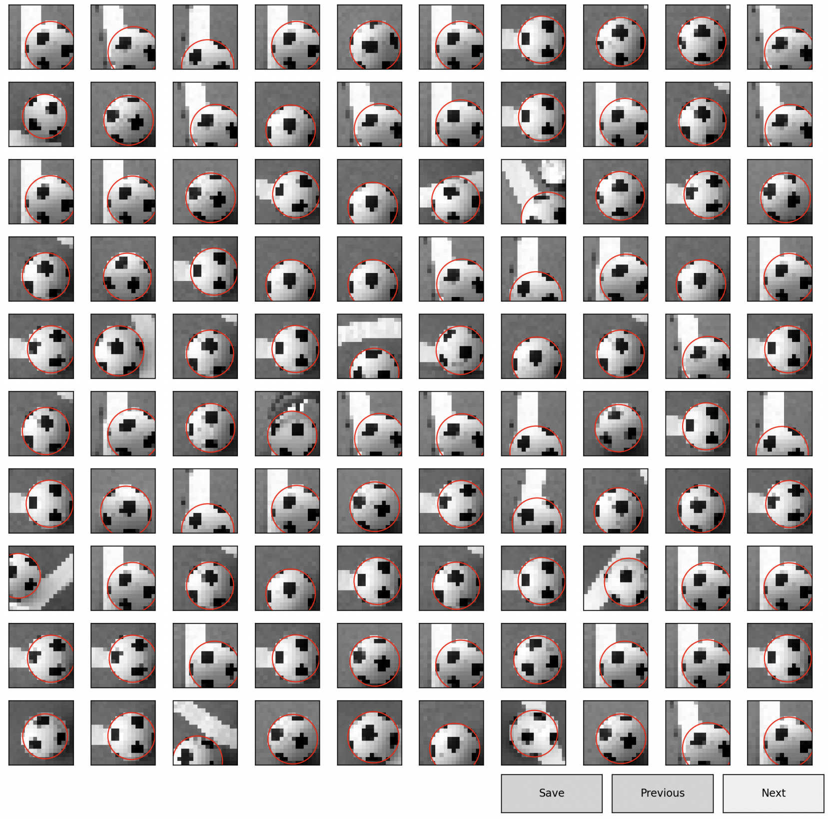


Figure 15: *Position estimation annotation tool developed using Python and Matplotlib. Ball patches are displayed in an interactive grid, with an overlaid red circle determined by their corresponding ground truth target values. Ball center and radius target values are adjustable for every patch using a Mouse and Keyboard.*

intersect with a ball class bounding box to belong to the ball class. Next, we used the calculated intersection of the ball bounding box and the ball candidate patch to filter out patches containing less than 40% of a balls total area. Since the object detection model also predicts bounding boxes for Nao robots, we can further exclude any patches that contain robots partially or fully covering balls. Finally, the author manually inspected all remaining 189,164 image patches, correcting any false positive and false negative ball class labels. This leaves us with a dataset of 31 games, experiments, and practice sessions that we can use to perform cross-validated leave-k-out experiments. The dataset contains 7,876 ball patches and 181,288 non ball patches. Figure 15 shows the tool we developed to adjust ball position estimation ground truth data.

Dataset for Ball Position Estimation The dataset for soccer ball position estimation is derived from the classification dataset. We use the annotated bounding box information to approximate the ball center x and y coordinates as the center of the bounding box and the ball radius as half the bounding box side length. Since bounding boxes are not always accurate, and we manually corrected many class labels of misclassified balls when constructing the dataset for classification, we created a software tool to help us verify and correct the target features of the ball samples. The tool is a Graphical User Interface (GUI) application written in Python and uses the *Matplotlib* visualization package to plot a paginated grid of ball patch images with overlaid circles derived from the target ball radius and center coordinates. Using the mouse and keyboard, one can quickly adjust the ball center and radius of each patch and save the adjusted dataset in the same format it was read in. The resulting dataset contains 7,876 ball patch images and their radii and ball center x and y coordinates as fractions of the square image width.

Benchmark Dataset We use a dataset provided by the B-Human team from Bremen to compare the classification and position estimation performance of the models developed in this work with our currently used classification model. Naturally, we have used all available data in the past for training, which would have made a meaningful comparison rather difficult without acquiring and labeling novel data. We expect the patches from this dataset to differ from our ball candidate patches, since the B-Human team uses a different hypothesis generation algorithm to find possible ball candidates. Therefore, we expect worse performance of our models than what we would find in our cross-validated experiments. Nevertheless, we can still use this dataset as a benchmark to compare relative performance differences between our current classifier and position estimator CNN models and the autoencoder based models we develop in this work. The dataset consists of 26,265 ball patches and 113,244 non ball patches. The target values contain both the class label and the ball radius and center x and y coordinates, making it an ideal candidate for our benchmark purposes. Since the B-Human patches have a width and height of 32 pixels, we apply the same resizing and preprocessing procedure as we do when generating our training data (Section 3.2.1.

3.3 K-Fold Cross-Validation

K-Fold cross-validation is a resampling method used to estimate model performance on unseen data samples. The complete dataset is divided into k parts, also called folds, then model training is performed a total of k times with $k-1$ parts used as training data and 1 part used for evaluation. By resampling and training multiple times, we obtain a better estimation of expected model performance compared to using a single train/test split [48]. After every evaluation, the model is discarded and built anew for the next trial. The K-Fold cross validation procedure can be repeated N times to perform NxKCross-Validation.

For our autoencoder architecture comparison experiments, we use the dataset presented in Section 3.2.2 and use the ten events as folds. Keeping the events grouped and not mixing samples from events to draw randomized subsets gives a better estimate on expected performance on data from future games and experiments, since lighting conditions, camera noise and even factors like motion blur resulting from adjustments in robot movement can differ a lot between distinct games. We perform a 10×10 cross validation, resulting in 100 distinct trials per experiment. We further split each training part into a training and

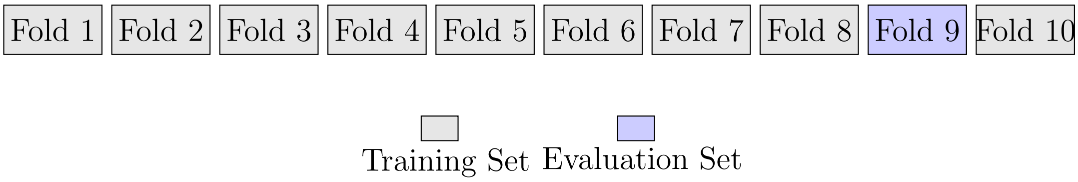


Figure 16: *Illustration of the 10-fold cross-validation procedure. The dataset is divided into 10 equal parts. In each of the 10 total iterations, nine parts (shown in gray) are used for training, while the remaining one part (shown in blue) is used for evaluation.*

validation subset, where the validation subset is used for reducing the learning rate and early stopping to prevent overfitting. The training and validation splits remain consistent until a 10-Fold cross-validation run is completed, then they are re-computed for the next run.

3.4 Tools & Software Contributions

In preparation for the work in this thesis and the RoboCup 2024 World Cup Event in Eindhoven (Netherlands), we contributed new features to the existing infrastructure and data processing software suite, as well as developing novel software tools. Noteworthy contributions include:

- Development of novel generic dataset creation scripts for autoencoder, ball classification and ball position estimation model training. We integrate data from the data processing pipeline database, object storage and annotation software components to quickly create up-to-date datasets from ingested and annotated game logs. Image patches can be exported in customizable dimensions and in color (YUV422) or grayscale (Y-Channel) to enable future experiments with different requirements.

- Integration of bounding box information from the annotation tool *LabelStudio* when generating ball candidate hypothesis from recorded game logs. Ball candidate patches are checked for overlap with annotated bounding boxes to determine class labels automatically. We added support for all valid labels, including *ball*, *robot* and *penalty mark*.

- Enable export of ball hypothesis image patches from game logs without verified annotations, facilitating the creation of unlabeled datasets for autoencoder training.

- Development of a GPU enabled containerized model training workflow on the Berlin United compute infrastructure. This allowed us to perform an extensive grid search over CNN based ball classification model hyperparameters. In preparation for RoboCup 2024, we performed over 500 individual experiments to improve upon the existing ball classification model.

- Implementation of search capabilities for the *LabelStudio* annotation tool. We extended the Python Django framework backend with custom serializers and URL routes to enable search and filtering of game logs by event name, game name and date from within the *LabelStudio* software.

- Development of a novel labeling tool for ball position estimation target features based on ball hypothesis candidate patches. In addition to the existing annotation solution that operates on full camera images, we can now efficiently add and adjust radii and ball center coordinates for exported ball candidate image patches.
- Development of a novel Single Page Application (SPA) classifier model comparison tool. To aid the classifier model selection process after training hundreds of candidate architectures for the RoboCup 2024 event, we developed a web-based solution to compare and contrast promising models. Metrics and visualizations include a classification report, a confusion matrix, a plot of the Precision-Recall Curve with indicators for optimal precision thresholds and visualizations of false positive and false negative ball predictions as image grids displayed side by side for each model.

3.5 Autoencoder Design Aspects

Before comparing different autoencoder architectures and training paradigms, let us first examine the common building blocks used in Convolutional Autoencoders. Since the data we are working with to classify and detect soccer balls differs significantly from datasets commonly used to benchmark CNNs and autoencoders in both resolution and content, we evaluate various architectural components to determine their effectiveness in our specific context. The following section compares commonly used loss functions, batch normalization, the usage of pooling layers and regularization techniques as well as downsampling and upsampling strategies to establish a sensible foundation for subsequent experiments. In each of the experiments, we change one design aspect of the architecture while keeping the others fixed.

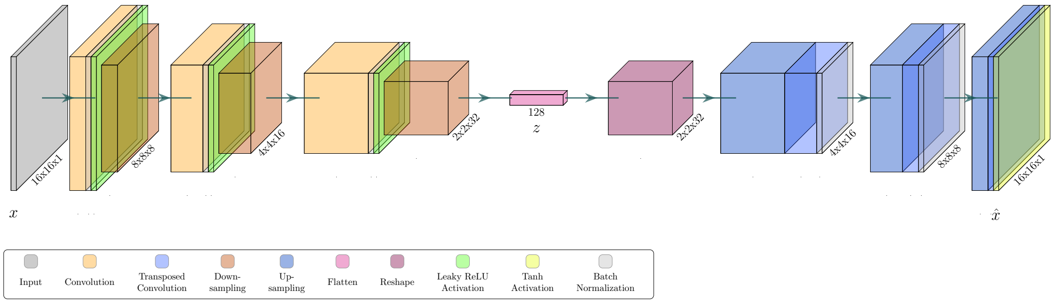


Figure 17: *Illustration of the Convolutional Autoencoder baseline architecture.*

We use a simple CAE architecture as a baseline model to serve as the foundation for relative comparisons, changing a single design aspect per experiment. Figure 17 depicts the baseline model we use in this run of experiments. The baseline architecture is composed of an encoder with three blocks each consisting of a convolutional layer (with padding, kernel size 3×3), a batch normalization layer, a leaky ReLU activation layer and a max pooling layer (window size and stride 2×2). The latent space feature vector has 128 components. The decoder network mirrors the encoder and uses nearest neighbor upsampling and transposed convolutional layers to reconstruct the input image of shape *(16×16×1)* from the 128-dimensional latent space representation. The decoder uses linear activations in all layers but the final transposed convolution, which uses the hyperbolic tangent to squash the output values into the valid range of $[-1, 1]$.

In each of the following experiments, we use the dataset described in Section 3.2.2 and perform 10×10 cross-validation as described in Section 3.3. Every cross validation fold is trained for 25 epochs with early stopping on nine training folds and evaluated on the held out evaluation fold.

3.5.1 Loss Function

In this experiment, we compare the effects of several loss functions on the quality of reconstructed images. To compute the gradients in the optimization step, we take the mean over all error terms in the mini-batch. The following loss functions are tested:

- MSE loss as described in Section 2.3.1, without any additional data transformations
- BCE loss as described in Section 2.3.1, scaling both the preprocessed original image and the reconstruction back to the range $[0, 1]$ when computing the loss
- SSIM loss as described in Section 2.3.1, scaling both the preprocessed original image and the reconstruction back to the range $[0, 1]$ when computing the loss
- A combination of MSE and the rescaled version of SSIM, with $\mathcal{L}_{combined} = \alpha\mathcal{L}_{MSE} + \mathcal{L}_{SSIM}$ and $\alpha = 50$ to bring both error terms to a similar value range

Since both the rescaling and the patchwise brightness normalization preprocessing steps transform the range of possible pixel values, we need to ensure the data provided in the loss term calculations fit the expected value range of the loss functions. For the adjusted BCE and SSIM losses, we use the preprocessed images with a pixel value range of $[-1, 1]$ as inputs to the encoder and rescale both the original preprocessed input and the reconstructed image to the range of $[0, 1]$ before computing the loss. Figure 18 depicts the loss value statistics of autoencoder architectures optimized using different loss functions. We used a 10×10 cross-validation training and evaluation procedure and collected the results of the loss scores computed on the evaluation fold at the end of each model training run, for a total of 100 evaluation loss scores per loss function.

We observe that the MSE, BCE and the combined MSE + SSIM loss functions achieve comparable reconstruction loss values in all trials. Comparing the MSE and BCE reconstruction loss subplots in Figure 18, we note that differences in model reconstruction performance are better discernible when using the MSE loss metric, which matches our intuition given that differences in reconstructed and original pixel values are squared and therefore lead to a non-linear increase in loss values for greater deviations between

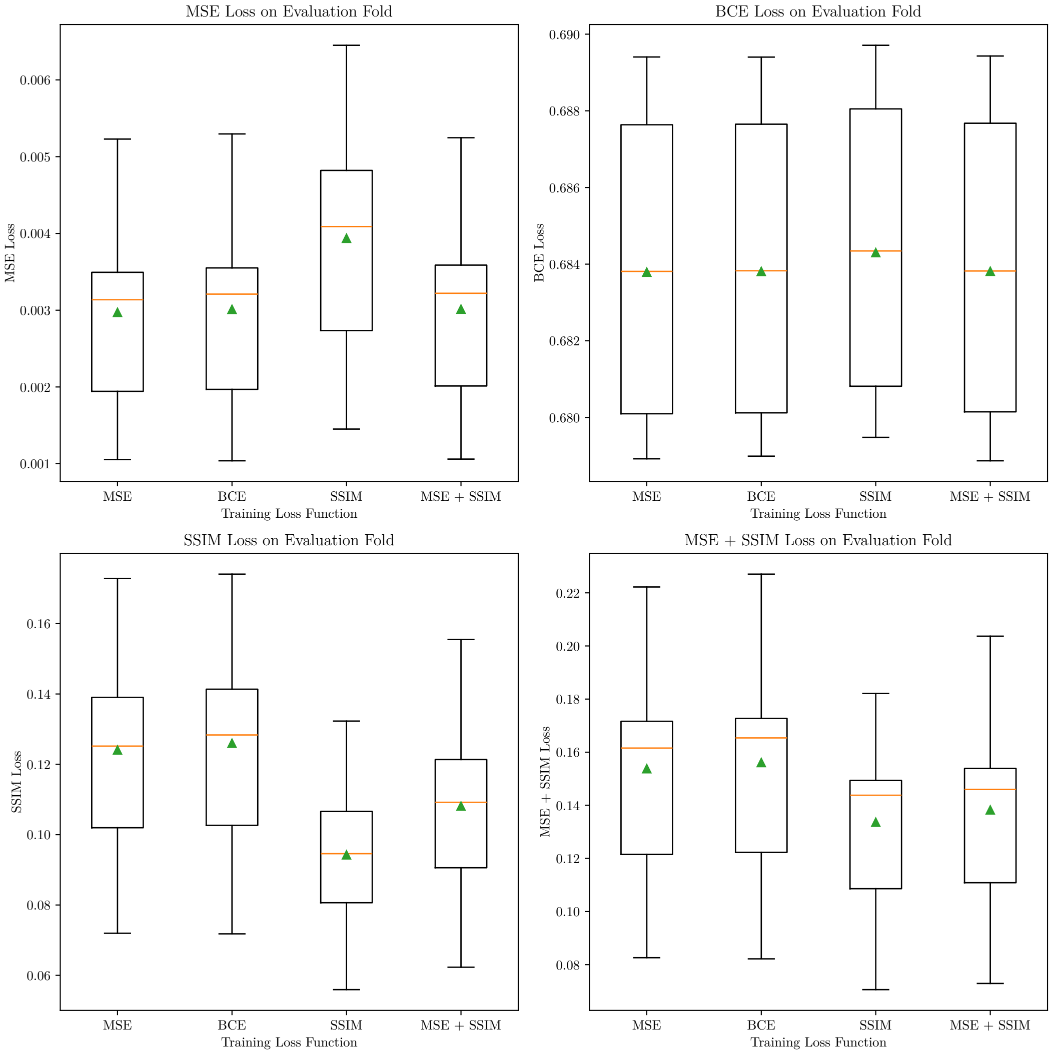


Figure 18: *Box plots of 10×10 cross-validation results of autoencoders trained with MSE, BCE, SSIM and a combination of MSE+SSIM loss functions. In each subplot, the y-axis represents the loss value computed on the evaluation sets over all trials. The x-axis denotes which loss function was minimized in model training. Each box shows the interquartile range, median (orange line), mean (green triangle) and variability of loss values over all trials. Whiskers extend to 1.5xIQR.*

reconstruction and original. Training the autoencoder on SSIM loss alone is not enough to achieve a comparable reduction in MSE loss. Using a combination of MSE and SSIM loss leads to good results in all four trials.

Figure 19 shows the difference in qualitative aspects of the reconstructed patches. Here we observe that models trained to minimize the MSE and BCE loss functions appear to produce moderately blurred reconstructions, while models trained to minimize the SSIM and MSE + SSIM loss have the capability to reconstruct sharper image features. Considering implementation and model training, MSE loss is both the fastest to

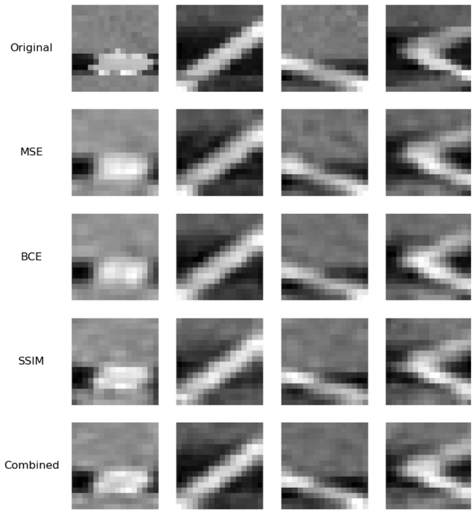


Figure 19: *Reconstructed patches from autoencoders trained to minimize different loss functions. From top to bottom, we see the original patches, MSE, BCE, SSIM and MSE+SSIM optimized autoencoder reconstructions.*

compute and easiest to use regardless of image preprocessing, as it requires no rescaling of reconstructions and targets. We therefore use the MSE loss for our subsequent experiments and will re-examine the combined MSE + SSIM loss in Section 3.6.

3.5.2 Latent Space Dimensionality

In this experiment, we compare the effects of different latent space dimensionalities on autoencoder reconstruction quality. The patches used as input to the encoder model have a width and height of 16 pixels each, with one grayscale color channel for a total of $16 \times 16 \times 1 = 256$ individual features. To build an under-complete autoencoder, the number of features in the latent space representation must therefore be less than 256. Given the baseline autoencoder architecture, the feature maps of the last convolutional layer after pooling have the shape $(2, 2, n_{filters})$, so we adjust the number of filters in the last convolutional layer to be $\frac{n_{latent}}{4}$ with the desired latent space dimensionality in each trial. We expect an increase in reconstruction quality as the number of features in the latent space grows, however there might be a point of diminishing returns, beyond which additional features provide minimal benefit at the cost of increased inference time. Figure 20 shows the effects of different latent space dimensionalities on the MSE evaluation loss metric achieved after 10×10 cross-validation. Using our default of 128 features in the latent space yields the best results and is used in further experiments.

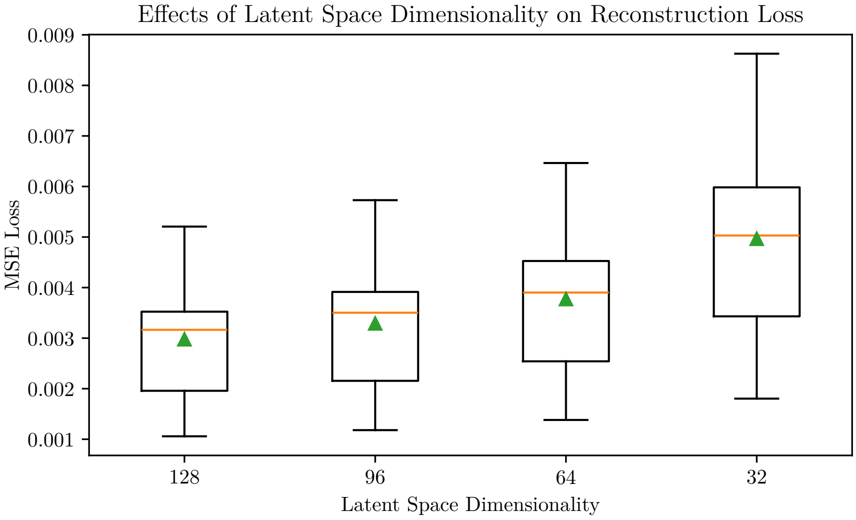


Figure 20: *Box plot of 10×10 cross-validation results of autoencoders with a latent space dimensionality of 128, 96, 64 and 32 respectively. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes which latent space dimensionality was used in training. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability of loss values achieved over all trials. Whiskers extend to 1.5xIQR.*

3.5.3 Pooling

This experiment evaluates the effect of using max pooling layers in the encoder and upsampling layers in the decoder on the reconstruction capabilities of autoencoders. We compare the models using max pooling layers in the encoder and nearest neighbor upsampling in the decoder against the same model architecture using strided convolutions in the encoder and strided transposed convolutions in the decoder. By changing the kernel stride in the convolutional layers from $(1, 1)$ to $(2, 2)$, we achieve the same resolution downsampling after each convolutional block as with using pooling layers. Pooling layers are commonly used in literature and by other teams, both for autoencoders and CNNs used to classify ball candidate patches. We investigate if the theoretical benefits of pooling, such as a supposed reduction of the influence of noisy samples, improved translation invariance and even regularization [31] can be observed when working with low resolution grayscale images. Figure 21 shows the outcome of the experiment. The results indicate that using strided convolutions in the encoder and strided transposed convolutions in the decoder leads to significantly lower reconstruction errors in our use case than using max pooling and nearest neighbor upsampling. Figure 22 depicts some example reconstructions of both models.

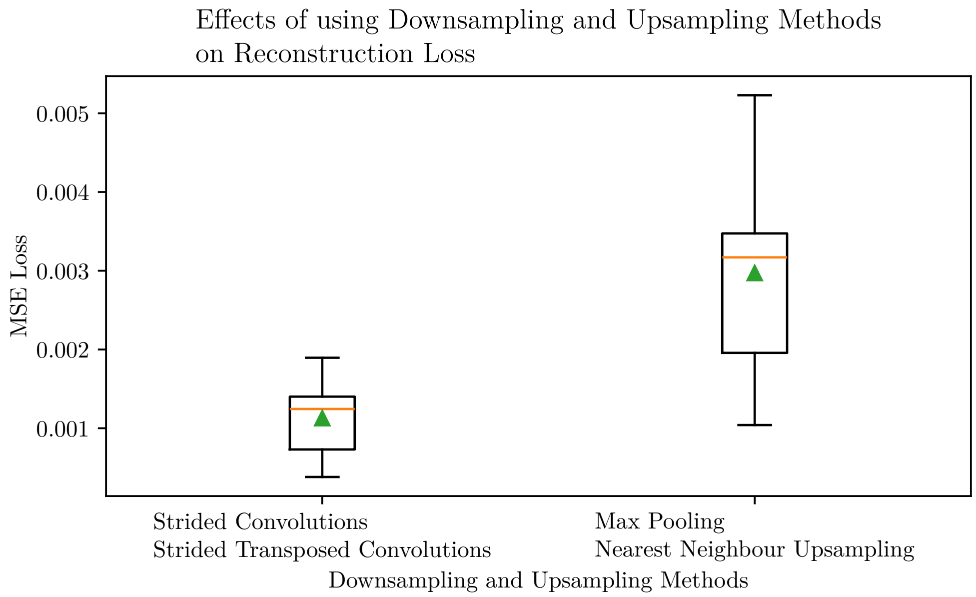


Figure 21: *Box plot of 10×10 cross-validation results of autoencoders trained with strided convolutions compared to using max pooling layers. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes whether pooling or strided convolutions were used during model training. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*

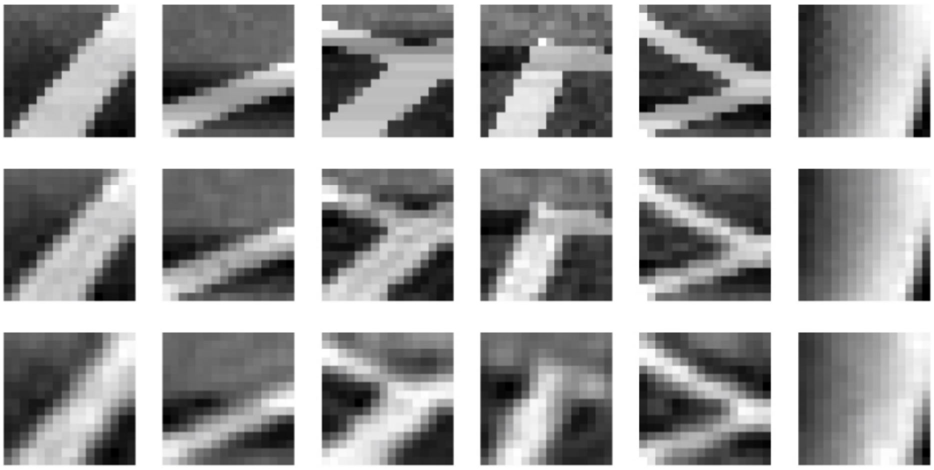


Figure 22: *A sample of CAE model reconstructions using different downsampling and upsampling strategies. The first row shows the original input images. The second row shows reconstruction from a model using strided convolutions in the encoder and strided transposed convolutions in the decoder. The third row shows reconstructions from a model using max pooling in the encoder and nearest neighbor upsampling in the decoder.*

3.5.4 Downsampling & Upsampling

As discussed in Section 2.3.2, there are several ways to shape the dimensions of feature maps in a convolutional autoencoder. In the encoder, we can reduce the width and height of a feature map by using convolutions without applying padding to the input, by using convolution strides greater than one or by using a pooling layer. In the decoder, we can increase the width and height of a feature map by using transposed convolutions without padding the input, by using transposed convolutions with a stride greater than one or by using an upsampling layer. Our chosen machine learning framework (TensorFlow/Keras) lacks a dedicated unpooling layer implementation, leading us to exclude this method from our analysis. In this experiment, we compare the effects of combining different methods of downsampling in the encoder and upsampling in the decoder on the reconstruction quality of patches generated by our baseline autoencoder architecture. In the encoder, we compare transposed convolutions with a stride of two in both width and height and a max pooling layer with the standard window size of two. In the decoder, we compare strided transposed convolutions with nearest neighbor upsampling and bilinear upsampling. The results of the experiment are depicted in Figure 23. We observe that using strided convolutions as a downsampling technique results in the lowest reconstruction errors, regardless of which upsampling technique is used in the decoder.

3.5.5 Batch Normalization

In this experiment, we test the efficacy of batch normalization layers for our use case. During training, a batch normalization layer normalizes the inputs to the following

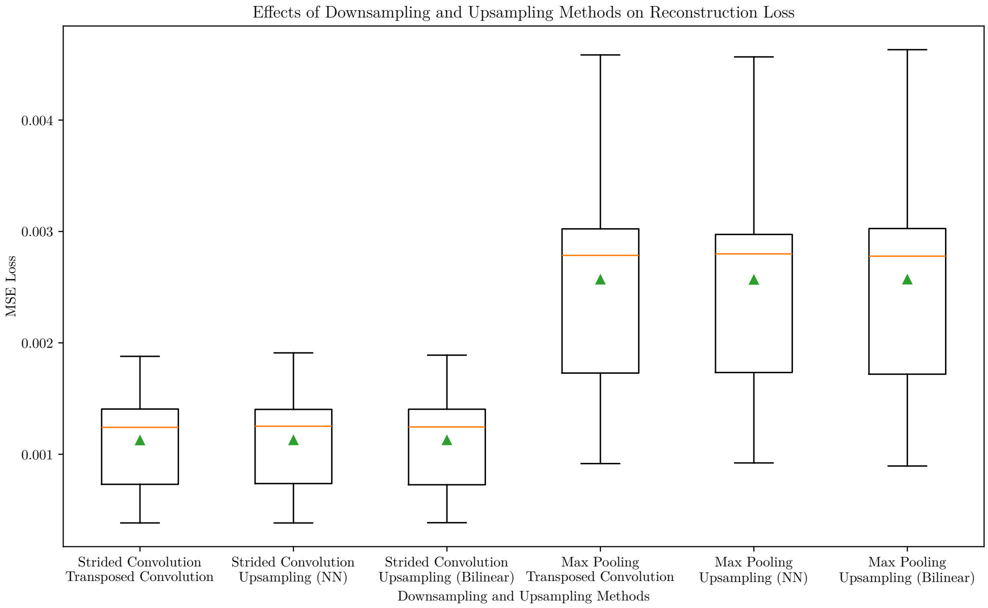


Figure 23: *Box plot of 10×10 cross-validation results of autoencoders trained with a combination of different downsampling and upsampling strategies. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes which downsampling and upsampling techniques were used in the encoder and decoder during model training. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*

layer by subtracting the mean and dividing by the standard deviation of the mini-batch, transforming the data distribution of the mini-batch to have a zero mean and unit variance. During inference, data is normalized using a moving average and moving standard deviation computed from the mini-batches the model has observed thus far. Batch normalization is used to speed up model training through faster convergence of the optimization process, though it is still disputed why this effect exactly occurs. Two popular explanations offered are a reduction of internal covariance shift in the mini-batch [49] and a smoothing of the objective function [50]. We evaluate the effects of using batch normalization in our baseline model architecture trained with a low batch size of 32 samples per batch to check if the theoretical benefits still manifest in our use case. The results depicted in Appendix Figure 1 indicate that we do not observe an improvement in convergence time or a reduction in the reconstruction error given our dataset and training hyperparameters. We will therefore exclude batch normalization from further autoencoder experiments.

3.5.6 Regularization

In this experiment, we evaluate different regularization techniques and their effects on the reconstruction quality of our baseline autoencoder architecture. In principle, regularization is applied to prevent models from overfitting on the training data at the detriment of their ability to generalize. Since we use a simple model and our data is both very low resolution and grayscale, it is worth investigating if any benefits resulting from regularization can be observed. We test both regularizing the values of the network weights, biases and activations, as well as a less traditional approach where we add a small dropout to the feature maps produced by the convolutional blocks. The second approach is inspired by the B-Human team, who employ a 20% dropout layer after each convolutional block. While they do not mention this regularization technique in their team reports, their published encoder model file reveals the usage of dropout layers when parsed and inspected. Additionally, we test a combination of both weight regularization and dropout. The results depicted in Appendix Figure 2 show that we observe no benefits in the achieved reconstruction loss scores from applying regularization, indicating that our baseline model is not overfitting on the training data. Since [31] suggests that using pooling layers in the encoder acts as a form of regularization, we also tested the effects of regularization on an autoencoder that uses strided convolutions in the encoder and strided transposed convolutions in the decoder, where we again observed no improvements in the final reconstruction performance when applying regularization. The results of this experiment are depicted in Appendix Figure 3.

3.5.7 Optimized Baseline Convolutional Autoencoder Architecture

Based on the results we have obtained thus far, we build a convolutional autoencoder that greedily combines the components and parametrizations that have worked well in previous experiments. We compare this new baseline with the previously used CNN models used for ball patch classification in the Berlin United team by re-implementing them as Convolutional Autoencoders. We do this by loading and inspecting the persisted model files to reveal their architecture, layer types and parameters. We then build a new encoder that contains only the convolutional and downsampling layers using the same parameters and discard the final fully connected layers. Preliminary tests have shown that better reconstructions are generated when we use custom decoders that mirror the encoders we generated based on the two CNNs compared to using the decoder from our baseline autoencoder architecture. The autoencoder architecture based on the current classifier model (in use since 2024) is depicted in Appendix Figure 7 and the autoencoder architecture based on the previously used classifier model (used 2019-2014) is depicted in Appendix Figure 8. The results of the architecture comparison are depicted in Figure 24.

We observe that our new baseline autoencoder architecture performs comparable, if not slightly better, to the encoder based on the current classifier model and significantly better than the encoder based on the old classifier model. However, our new baseline encoder model achieves this with 5888 trainable parameters, whereas the encoder based on the current classifier has 13792 trainable parameters. Since fewer trainable parameters

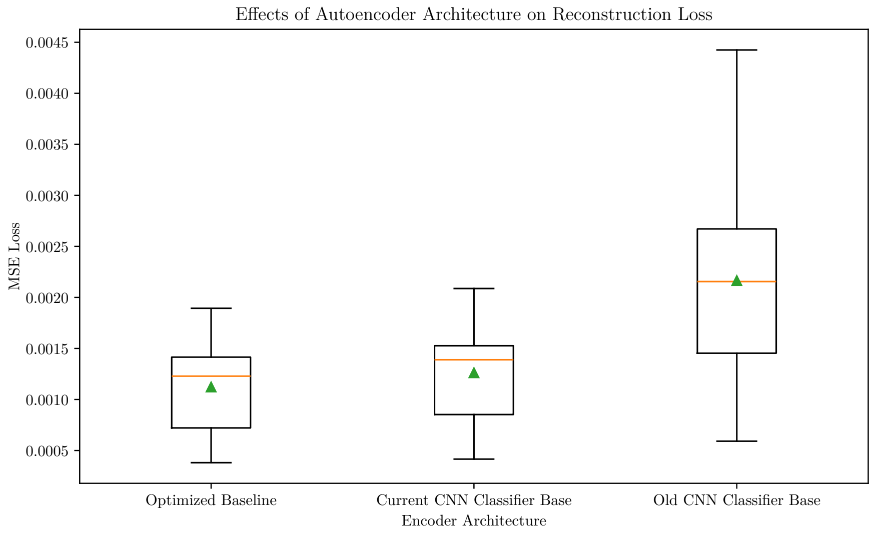


Figure 24: *Box plot of 10×10 cross-validation results of Convolutional Autoencoders trained with different encoder architectures. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes the encoder architecture, comparing the optimized baseline approach resulting from our experiments and the feature extraction architectures of the last two CNN classifiers used by the Berlin United team. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*

lead to faster training and, more importantly, faster inference, we select the optimized baseline autoencoder architecture as the basis for further experimentation. The optimized baseline CAE architecture is depicted in Figure 25. It is worth pointing out, that our optimized baseline has a similar architecture as the currently used CNN classifier, which also uses strided convolutions instead of max pooling layers. However, the CNN model architecture resulted from trial-and-error experimentation instead of rigorous testing, as was performed for the autoencoder based models in this work.

3.6 Autoencoder Architecture Evaluation

In this section, we build and train various Convolutional Autoencoder architectures to be used as feature extractors for a ball classifier model and a ball position estimation model. In addition to the optimized baseline CAE architecture we derived in Section 3.5, we examine the Denoising Convolutional Autoencoder, Convolutional Variational Autoencoder and the Denoising Convolutional Variational Autoencoder model architectures. These architectures differ both in the tasks optimized during training and their latent space properties. The variational models introduce a regularization of the latent space and therefore result in different encodings compared to the unregularized autoencoder models, which may lead to differences in performance in downstream tasks. The denoising models are tasked to reconstruct a less noisy output from an input corrupted with noise, and we will evaluate their efficacy in downstream tasks as well. For this run of experiments, we use the full dataset of ball candidate patches from a total of 33 games and experiments to train the various candidate architectures and apply augmentations to the training data splits as described in Section 3.2.2. Since we are expecting significant differences in the reconstructions of these models, we will not be performing an extensive cross-validation to compare reconstruction performance across model architectures. However, we use the smaller dataset comprised of 10 distinct events and experiments and perform 10×10 cross-validation to find suitable hyperparameters for the distinct autoencoder architectures. We will be training models using both MSE loss and with the combined MSE + SSIM loss.

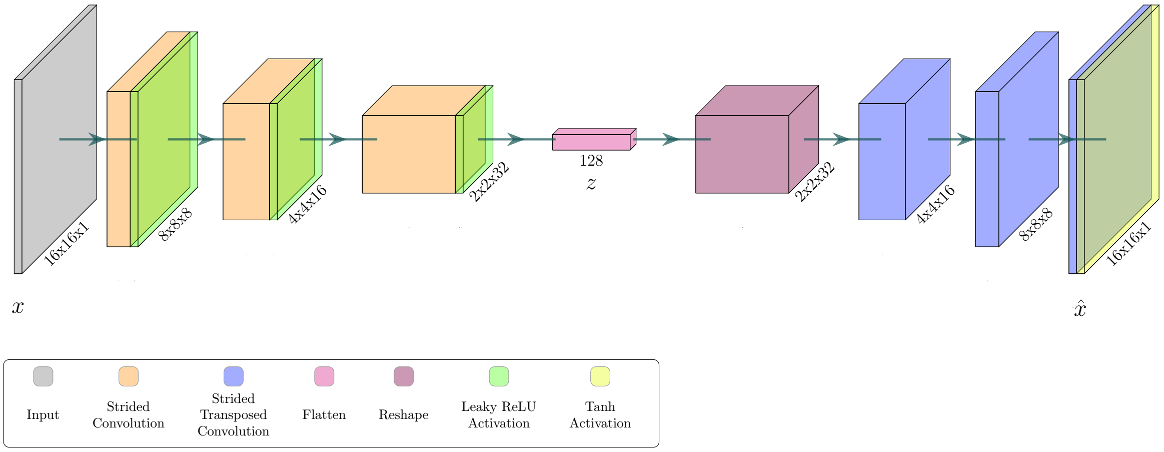


Figure 25: *Illustration of the optimized Convolutional Autoencoder architecture resulting from our experimentation.*

3.6.1 Denoising Convolutional Autoencoder

A DCAE shares the same architectural structure as a vanilla CAE, but differs in the task that is optimized during model training. Where the CAE reconstructs that original input image as closely as possible, the DCAE is presented with an input image that has been corrupted with one or more types of image noise and is tasked with reconstructing an uncorrupted image. In theory, optimizing for this task should increase the capability of a trained model to generalize and reduce its sensitivity to noise and less salient features when presented with unseen data samples at inference time. The latent representations generated by a Denoising Autoencoder may therefore be preferable to their vanilla counterparts for our downstream tasks of ball classification and ball position estimation. Additionally, we would be able to artificially increase the amount of available training data for these tasks, by creating noisy variations of existing samples. We will test this hypothesis in Section 3.7.

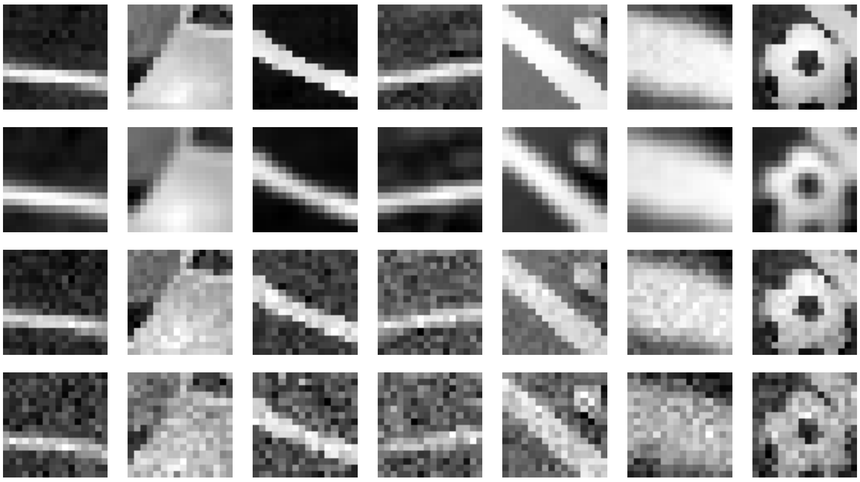


Figure 26: *Effects of smoothing and Gaussian image noise on ball candidate patches. The top row shows unmodified patches. The second row shows patches that have been blurred by convolution with a Gaussian kernel (3×3). The third and fourth rows show patches with added Gaussian noise, using a standard deviation of* 0.035 *and* 0.075 *respectively.*

Traditionally, one would use a dataset comprised of clean images to use as targets and apply a form of image corruption, such as Gaussian noise or masking noise, to the images to use as inputs to a denoising model. In our case, this approach poses a challenge, since our input data is very low resolution and already contains a significant amount of (Gaussian) camera noise that becomes apparent at these low resolutions. We ran a preliminary experiment to compare the effects of adding Gaussian noise and masking noise to our already noisy image samples, and found that the reconstruction loss would increase with corruptions added to the input images in every case. This can be seen in Appendix Figure 4. Therefore, we use a different approach and generate less noisy and blurred target images from our dataset by performing a convolution with a 3×3 Gaussian

Kernel. A sample of the resulting images is depicted in Figure 26.

To find a reasonable approximation for the variance in Gaussian noise our dataset exhibits in comparison with the smoothed targets we generated through the Gaussian blur, we perform a 10×10 cross-validated experiment. In each cross-validation run, we use 9 out of the 10 events as training data and the remaining event as test data. The training data is split further into a training and validation set. In all three splits, we use a Gaussian blur as described to generate the desired blurred target images. Gaussian noise is then dynamically added to the training data input samples, resulting in unique variations of the input samples in every epoch. We apply no additional noise to the validation and test input samples, as they are already noisy. The results of the experiment are depicted in Figure 27. We observe that the lowest MSE reconstructions error scores are achieved when noise is added to the training data sampled from a Gaussian normal distribution with $\sigma = 0.075$. We also observe that levels of image noise higher than that lead to an increase in reconstruction error. We train two models on the denoising tasks using Gaussian normal distributed image noise with $\sigma = 0.075$ for further experimentation as feature extractors, one trained to minimize the MSE loss and one trained to minimize the combined MSE + SSIM loss. The DCAE model architecture is identical to the optimized baseline CAE architecture depicted in Figure 25.

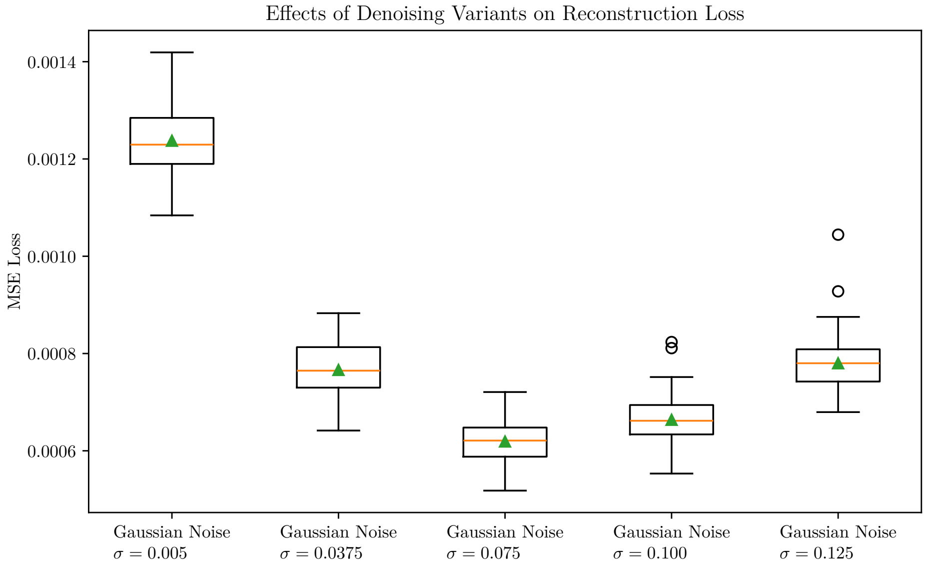


Figure 27: *Box plot of 10×10 cross-validation results comparing different levels of image corruption using Gaussian noise. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes the standard deviation of the Gaussian image noise added during training. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR. Circles indicate outliers.*

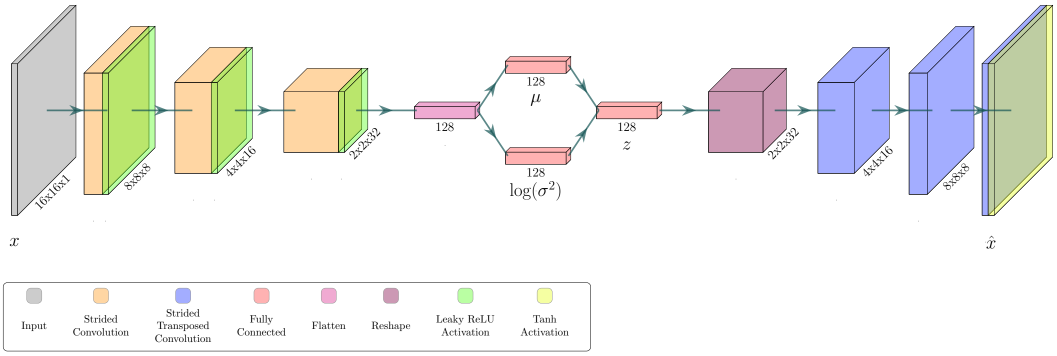


Figure 28: *Illustration of the optimized Convolutional Variational Autoencoder architecture used in experiments.*

3.6.2 Convolutional Variational Autoencoder

The CVAE is a generative model that encodes inputs as distributions in the latent space, which is regularized to follow a prior distribution, typically a multivariate Gaussian normal distribution (see Section 2.3.4). The goal of this regularization is to encode inputs in such a way, that similar inputs result in overlapping probability distributions in the latent space. In contrast, a vanilla autoencoder is free to encode similar inputs as vectors that are farther apart in the latent space. We hypothesize, that the regularized latent space improves the robustness of downstream tasks, such as ball classification, which we test in Section 3.7. We extend the CVAE loss function formulated in Equation (30) by a scaling factor α for the reconstruction term and a scaling factor β for the KL divergence term. The resulting loss function we use in model training therefore becomes:

$$\mathcal{L}_{CVAE} = \alpha\mathcal{L}_{MSE} + \beta\mathcal{L}_{KL} \tag{33}$$

Our task is to find a suitable value for β, such that the latent space regularization property remains while preventing posterior collapse and ensuring acceptable reconstruction quality. We have determined empirically that the reconstruction loss takes on values of magnitude 10^{-3} to 10^{-4} and that an unregularized CVAE with $\beta = 0$ leads to KL divergence terms of magnitude 10^3. We perform a 10×10 cross validated trial using the dataset comprised of 10 distinct events to find a suitable value for β that strikes a balance between reconstruction quality and latent space regularization. We use the optimized baseline CAE architecture and re-implement it as a CVAE (Figure 28). The results of the cross-validated experiment using MSE loss are depicted in Figure 29. We find promising tradeoffs between regularization and reconstruction performance for $\beta = 1 \times 10^{-4}$ and $\beta = 1 \times 10^{-5}$. We repeat the experiment with the same model architecture using the MSE + SSIM loss and select the models with $\beta = 1 \times 10^{-3}$ and $\beta = 1 \times 10^{-4}$ for further experimentation (Appendix Figure 5). We deliberately include models with a higher reconstruction loss value and higher latent space regularization to be evaluated as feature extractors for downstream tasks.

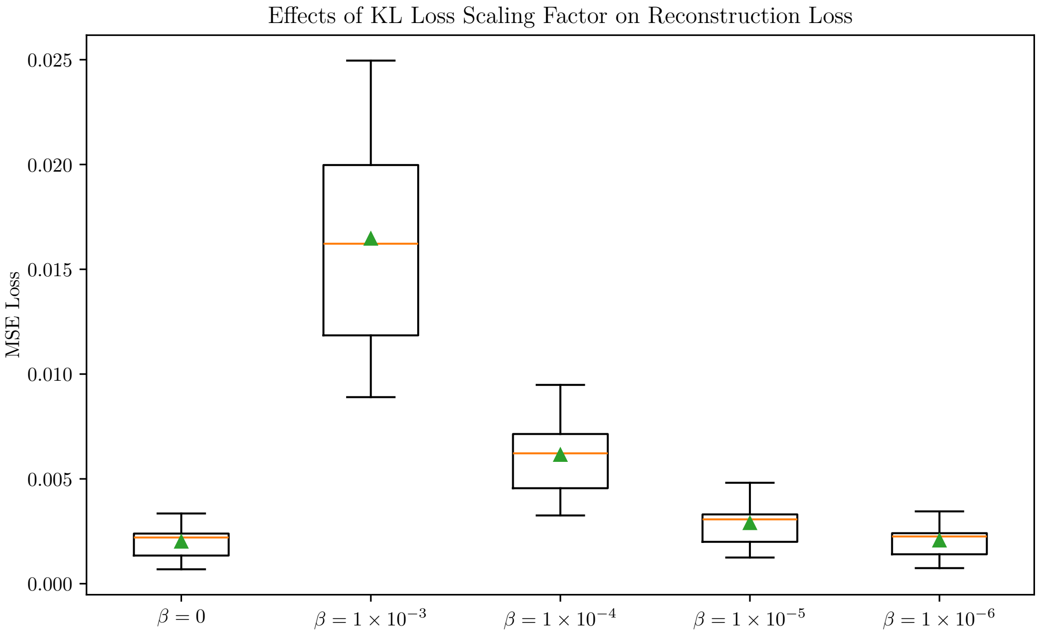


Figure 29: *Box plot of 10×10 cross-validation results comparing the effects of the KL loss scaling factor on reconstruction loss in Convolutional Variational Autoencoders. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes the KL loss scaling factor β. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*

3.6.3 Denoising Convolutional Variational Autoencoder

The final autoencoder architecture we investigate is the DCVAE. We use the CVAE architecture from the previous experiment (Figure 28) and task it with reconstructing a denoised output from an input image corrupted with Gaussian noise. We aim to combine the strengths of the CVAE, primarily the regularization of the latent space, and the denoising task, namely the ability of models to learn robust features and improve generalization. We perform a 10×10 cross-validation training run using the dataset comprised of 10 distinct events to find a suitable β scaling factor for the KL divergence term in the loss function (see Equation (33)). The results of the experiment using MSE loss are depicted in Figure 30. Again, we perform another cross-validated experiment using the combined MSE + SSIM loss as well, the results of which are shown in Appendix Figure 6. We select the models using $\beta = 1 \times 10^{-5}$ and $\beta = 1 \times 10^{-6}$ from the MSE training experiments and models with $\beta = 1 \times 10^{-3}$ and $\beta = 1 \times 10^{-4}$ from the combined loss training experiments as candidate architectures for further experimentation.

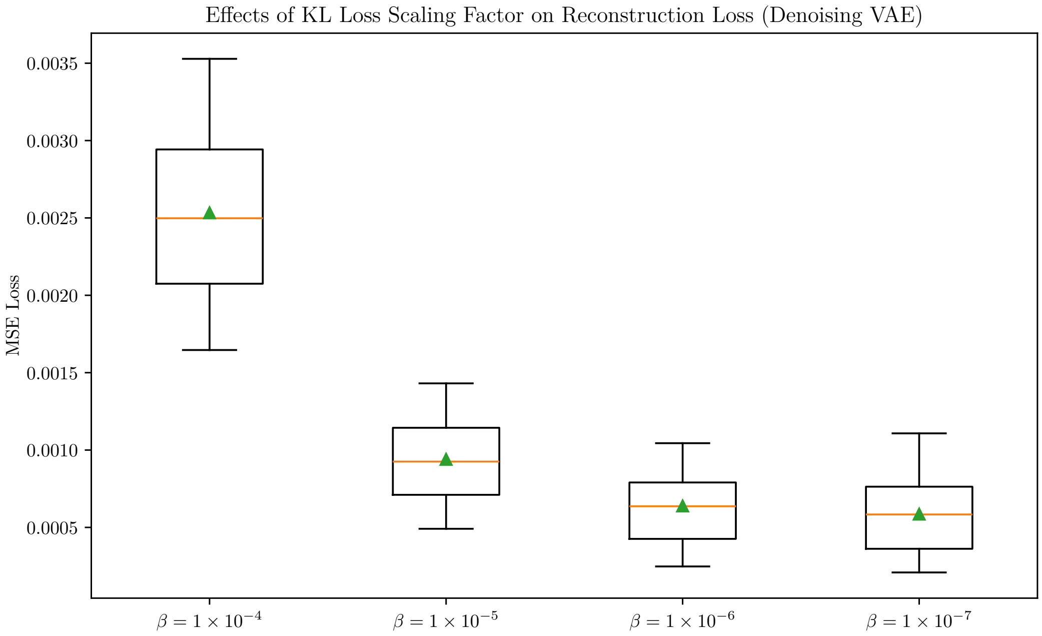


Figure 30: *Box plot of 10×10 cross-validation results comparing the effects of the KL loss scaling factor on reconstruction loss in Denoising Convolutional Variational Autoencoders. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes the KL loss scaling factor* β*. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*

3.7 Autoencoder based Soccer Ball Classification

By training various CAE architectures with different loss functions and reconstruction tasks in section 3.6, we created 12 distinct models on that can serve as potential feature extractors for a CNN ball classification model. In this experiment, we compare the predictive performance of classifier models based on the trained autoencoder models. We derive a classifier from a trained autoencoder by discarding the decoder network and freezing all layers in the encoder. Then we add multiple fully connected layers, with the last layer consisting of two output neurons with the softmax activation function applied. When using CVAEs as the basis, we avoid the expensive sampling step and select the fully connected mean feature vector as the last layer of the encoder. In addition to the classifier models based on autoencoders, we also evaluate a CNN with the same configuration of layers that is trained from the ground up on the classification task.

We train the models on a labeled dataset of all available ball candidate patches (Section 3.2.2), where the binary ground truth label of each sample (*ball* or *not ball*) has been verified manually by the author. Since our dataset is highly imbalanced, we use the α-*balanced Focal Loss* function for model training (Section 2.3.1) with $\alpha = 0.25$ and $\gamma = 2$, which yielded a good balance between the achieved precision and recall metrics in preliminary testing.

To artificially increase the number of samples available during model training time, we apply slight augmentations to the training data image patches. These augmentations do not change the characteristics of the image patches drastically and produce samples that could realistically be expected to be observed in future games. The augmentations include flipping the image columnwise with a chance of 50% and adding a small amount of Gaussian normal distributed image noise with $\sigma = 0.005$.

We employ a 10-fold cross-validation evaluation trial, comparing classification metrics with a focus on the positive (*ball*) class. In robot soccer, we are especially interested in a low number of false positive predictions, as erroneous ball percepts propagate through the internal modelling of the agents and can lead to detriments in behavior and action selection. Historically, we have observed various undesired robot behaviors when false positive ball detections accumulate, ranging from losing sight of the actual ball, to kicking the ground resulting in falling and even multiple robots converging on a false ball percept and pushing each other over. Therefore, we evaluate and compare classifier performance on the precision, recall and F1-Score metrics of the positive ball class (Section 3.1). Since we use a softmax activation function in the last dense layer of the classifier models, we can adjust the threshold value of prediction probabilities of the positive class to optimize for a higher precision. We determine the precision optimized threshold per fold by computing the mean prediction score over all ball samples.

Feature Extractor	Training Loss	Precision	Recall	F1-Score	Precision$_{T_{opt}}$	Recall$_{T_{opt}}$	F1-Score$_{T_{opt}}$
CAE	MSE	0.830±0.116	0.397±0.179	0.505±0.197	0.654±0.207	0.523±0.084	0.565±0.112
CAE	MSE+SSIM	0.887±0.055	0.561±0.175	0.671±0.149	0.862±0.147	0.549±0.055	0.668±0.085
DCAE	MSE	0.921±0.049	0.852±0.069	0.884±0.053	0.984±0.019	0.677±0.045	0.802±0.036
DCAE	MSE+SSIM	0.925±0.033	0.865±0.074	0.893±0.049	0.989±0.010	0.688±0.044	0.811±0.032
CVAE$_{\beta=1\times10^{-4}}$	MSE	0.934±0.038	0.839±0.084	0.882±0.059	0.990±0.014	0.669±0.084	0.796±0.064
CVAE$_{\beta=1\times10^{-5}}$	MSE	0.932±0.034	0.838±0.081	0.881±0.055	0.983±0.016	0.696±0.089	0.812±0.068
CVAE$_{\beta=1\times10^{-3}}$	MSE+SSIM	0.950±0.019	0.847±0.081	0.893±0.049	0.992±0.007	0.689±0.090	0.810±0.065
CVAE$_{\beta=1\times10^{-4}}$	MSE+SSIM	0.935±0.033	0.850±0.067	0.889±0.044	0.984±0.012	0.691±0.069	0.810±0.051
DCVAE$_{\beta=1\times10^{-5}}$	MSE	0.941±0.029	0.860±0.069	0.897±0.043	0.988±0.016	0.703±0.075	0.820±0.055
DCVAE$_{\beta=1\times10^{-6}}$	MSE	0.937±0.040	0.862±0.077	0.896±0.057	0.990±0.014	0.720±0.075	0.832±0.055
DCVAE$_{\beta=1\times10^{-3}}$	MSE+SSIM	0.942±0.036	0.870±0.064	0.904±0.047	0.988±0.018	0.675±0.076	0.800±0.059
DCVAE$_{\beta=1\times10^{-4}}$	MSE+SSIM	0.936±0.035	0.852±0.076	0.891±0.054	0.990±0.009	0.711±0.079	0.825±0.057
CNN (CAE-like)	Focal Loss	0.956±0.023	0.932±0.036	0.944±0.023	0.993±0.007	0.854±0.043	0.917±0.025

Table 2: *Classification performance metrics of classifier models using autoencoder based feature extractors. Results are computed from a 10-fold cross-validation trial. Metrics are computed both for a default prediction confidence threshold of* 0.5 *and for a higher threshold based on the mean ball confidence score. The last row shows the metrics computed for a CNN model that shares the feature extractor architecture of the autoencoder based models, but has not been initialized with weights from autoencoder training.*

Table 2 shows the results of the 10-fold cross-validated ball classification experiment. We report the precision, recall and F1-score metrics for both the default confidence threshold of $T = 0.5$ and for using the precision optimized threshold computed per evaluation fold. The reported metrics show the mean and standard deviation calculated over all folds. We can observe that the autoencoders trained on the denoising tasks and the CVAE models perform significantly better than the vanilla autoencoder models. We also observe the CNN model using the same model architecture as the autoencoder models achieving a better recall than the autoencoder based classifier models. Increasing the confidence threshold does indeed improve the precision for all architectures except the vanilla autoencoders, although not surprisingly at the cost of a lower recall. In Section 4 we take a closer look at the false negative predictions to investigate how we might improve the recall of the autoencoder based classifiers.

3.7.1 Evaluation on Benchmark Dataset

In the past, the Berlin United team has always used all labeled data available at the time for model training. This presents us with a challenge when we want to compare the autoencoder based ball classification models with the currently used CNN ball classifier model. Not only does a significant overlap of training data exist between the two approaches, but our limited number of labeled ball samples makes it difficult to justify holding out a significant amount of distinct event logs for testing. Therefore, we chose to evaluate our current CNN classifier and the autoencoder based classifier models presented in this work on a benchmark classification dataset provided by the Bremen B-Human team in 2019 (see Section 3.2.2). Since the B-Human team uses a different ball hypothesis algorithm and image patch processing routine, we do not expect to achieve a performance comparable to the cross-validated trials we performed using our data. However, we can still attempt to compare relative performance differences between our autoencoder based classifiers and the current CNN ball classifier.

Model Architecture	Training Loss	Precision	Recall	F1-Score	False Positive Rate
CAE	MSE	0.207	0.105	0.140	0.086
CAE	MSE+SSIM	0.187	0.973	0.314	0.495
DCAE	MSE	0.745	0.149	0.249	0.012
DCAE	MSE+SSIM	0.488	0.362	0.416	0.081
$\text{CVAE}_{\beta=1\times10^{-4}}$	MSE	0.760	0.180	0.291	0.013
$\text{CVAE}_{\beta=1\times10^{-5}}$	MSE	0.558	0.215	0.310	0.038
$\text{CVAE}_{\beta=1\times10^{-3}}$	MSE+SSIM	0.623	0.276	0.383	0.037
$\text{CVAE}_{\beta=1\times10^{-4}}$	MSE+SSIM	0.467	0.394	0.428	0.095
$\text{DCVAE}_{\beta=1\times10^{-5}}$	MSE	0.636	0.249	0.357	0.032
$\text{DCVAE}_{\beta=1\times10^{-6}}$	MSE	0.752	0.167	0.273	0.013
$\text{DCVAE}_{\beta=1\times10^{-3}}$	MSE+SSIM	0.790	0.195	0.313	0.012
$\text{DCVAE}_{\beta=1\times10^{-4}}$	MSE+SSIM	0.539	0.297	0.383	0.056
CNN (CAE-like)	Focal Loss	0.833	0.383	0.524	0.018
CNN Classifier (Current)	Focal Loss	0.220	0.995	0.360	0.450

Table 3: *Classification performance metrics on the B-Human benchmark dataset of classifier models using autoencoder based feature extractors. Results are computed from a 10-fold cross-validation trial. The trial included a CNN model that shares the feature extractor architecture of the autoencoder based models, but has not been initialized with weights from autoencoder training, as well as the current CNN classifier used by the Berlin United team.*

Table 3 compares the precision, recall, F1-score and false positive rate achieved by the autoencoder based models and the current CNN ball classifier model on the B-Human benchmark dataset. We also include the CNN model that uses the same architecture as the autoencoder feature extractors, but has not been initialized with weights from autoencoder training. We observe that our models significantly outperform the currently used ball classification model in terms of precision and false positive rate on this dataset. However, none of the evaluated models reach a level of quality that would result in acceptable in-game performance. It is apparent that the B-Human ball candidate patches significantly differ from our ball candidate patches. Note that in most cases, the CVAEs based models with a higher latent space regularization achieve a higher precision and lower recall than the CVAEs based models with less latent space regularization applied. As is the case in the 10-fold cross-validated ball classifier experiment, the best results are achieved by the CNN model that has not been initialized with weights from autoencoder training.

3.8 Autoencoder based Soccer Ball Position Estimation

In this experiment, we use the trained CAE models as feature extractors for a ball position estimation regressor. The target features predicted are the estimated ball radius and ball center x and y coordinates as fractions of image patch width and height. We rely on the bounding box annotation data added during data processing for ground truth target values. As part of this work, we verified and corrected all available ball position target features manually. Analogous to the autoencoder based ball classification experiment (Section 3.7), we create regression models from the trained autoencoders by discarding the decoders and freezing the weights of the encoder networks. For CVAEs, we use the mean feature vector as the last layer to avoid having to perform a sampling step. We add multiple fully connected layers with trainable parameters and use three output neurons with a sigmoid activation in the last layer, since our target values are ranged between 0 and 1.

We train the models on the regression dataset described in Section 3.2.2, which is composed exclusively of ball patch images and their target radii and center coordinates and does not include any negative class samples. We use the IoU loss as described in Section 2.3.1 and add an MSE error term, to get a differentiable error gradient for the case of non-overlapping bounding boxes. We can easily derive the necessary bounding box coordinates for the IOU loss from the available ball center and radius information by taking defining all bounding boxes to be squares with side length of two times the radius. The loss function used during training therefore becomes:

$$\mathcal{L}_{reg}(y, \hat{y}) = 1 - \mathrm{IoU}(y, \hat{y}) + \mathcal{L}_{MSE}(y, \hat{y}) \tag{34}$$

We perform a 10-fold cross-validation trial for each of the 12 candidate autoencoder architectures, as well as one regression model with the same model architecture trained without any prior weight initialization. We also perform a position estimation inference pass with the currently used CNN model, which has been trained on a completely disjunct handcrafted dataset created by a Berlin United team member in 2019. This dataset has been manually crafted from small experiments which are not included in our game log database, and includes image patches from the older Nao v5 robot models. We do not retrain this position estimation model, we simply compute the combined IoU+MSE loss for each test fold of the cross-validation dataset. Based on the poor predictive performance of all tested models on the B-Human benchmark data in the classification experiment, we forego a position estimation comparison using this dataset. The results depicted in Table 4 show that the CNN trained from scratch outperforms all autoencoder based regression models. We can again observe the autoencoders trained on the denoising task and the CVAE architectures achieving a lower average loss score than their vanilla counterparts. We also see an improved performance of the autoencoder based models in comparison to the currently used CNN model that has been trained on the handcrafted dataset containing data from older robot models.

Feature Extractor	Training Task	Training Loss	$1 - \text{IoU} + \text{MSE}$
CAE	Reconstruction	MSE	0.1157±0.0425
CAE	Reconstruction	MSE+SSIM	0.1134±0.0413
DCAE	Denoising	MSE	0.1011±0.0321
DCAE	Denoising	MSE+SSIM	0.0958±0.0268
$\text{CVAE}_{\beta=1\times10^{-4}}$	Reconstruction	MSE	0.1082±0.0388
$\text{CVAE}_{\beta=1\times10^{-5}}$	Reconstruction	MSE	0.1045±0.0364
$\text{CVAE}_{\beta=1\times10^{-3}}$	Reconstruction	MSE+SSIM	0.1057±0.0346
$\text{CVAE}_{\beta=1\times10^{-4}}$	Reconstruction	MSE+SSIM	0.1054±0.0386
$\text{DCVAE}_{\beta=1\times10^{-5}}$	Denoising	MSE	0.1087±0.0359
$\text{DCVAE}_{\beta=1\times10^{-6}}$	Denoising	MSE	0.1024±0.0351
$\text{DCVAE}_{\beta=1\times10^{-3}}$	Denoising	MSE+SSIM	0.1090±0.0371
$\text{DCVAE}_{\beta=1\times10^{-4}}$	Denoising	MSE+SSIM	0.1040±0.0354
CNN (CAE-like)	Position Estimation	IoU+MSE	0.073±0.0163
CNN Estimator (Current)	Position Estimation	MSE	0.1409±0.0220

Table 4: *Position estimation performance comparison of regression models using different autoencoder based feature extractors. Results are computed from a 10-fold cross-validation trial. The trial includes a CNN regression model that shares the feature extractor architecture of the autoencoder based models, but has not been initialized with weights from autoencoder training. We also include the performance metrics achieved by the currently used position estimation CNN model, which has been trained on a disjunct handcrafted dataset.*

4 Analysis of Results

In this section, we analyze characteristics of the trained autoencoder models to better understand how they work and guide us in selecting promising candidates for evaluation on the Nao robot platform. We inspect a lower dimensional representation of image patch embeddings to judge the structure of the models' latent space and determine the effects of latent space regularization and the denoising training task. Then we compare aspects of the learned convolutional kernels to gain insight into which visual features the network focuses on when making classification predictions. Finally, we benchmark the inference runtime performance on the Nao robot.

4.1 Visualization & Explainability

When evaluating machine learning models, it is often not possible to judge qualitative differences between models by examining performance metrics alone. Given multiple candidate models with similar performance, one would prefer to use a robust model that has learned to identify meaningful features and relationships in the training data. In Section 3, we trained various CAE architectures and formed the following hypothesis:

1. Training a DCAE to reconstruct denoised images from corrupted image samples leads to a more robust feature extraction compared to training on the reconstruction task alone
2. Regularization of the latent space performed by CVAEs causes similar inputs to be encoded as overlapping distributions
3. Robust feature extraction and a regularized latent space improves the performance of CAE based ball classification and position estimation models

We evaluate the first two hypotheses by investigating the latent space embeddings that our candidate architectures produce in a lower dimensional visual representation. We evaluate the third hypothesis by relating feature map activations in the convolutional layers of the autoencoder based classification models to target class predictions to better understand which image features are most informative for the classification task.

4.2 Latent Space Structuring

We use the Uniform Manifold Approximation and Projection (UMAP) [51] dimensionality reduction approach to visualize the latent space embeddings of the ball classification dataset in two dimensions. UMAP constructs a weighted graph representation of the data, then optimizes a low-dimensional embedding of the graph that preserves both local and global topological structure. It is well suited for visualizing large datasets, making it a sensible choice for our analysis.

Using the available target class information, we can distinguish between embeddings from ball images and non-ball images. Figure 31 depicts the two-dimensional representations of image embeddings from the classification dataset for each candidate autoencoder

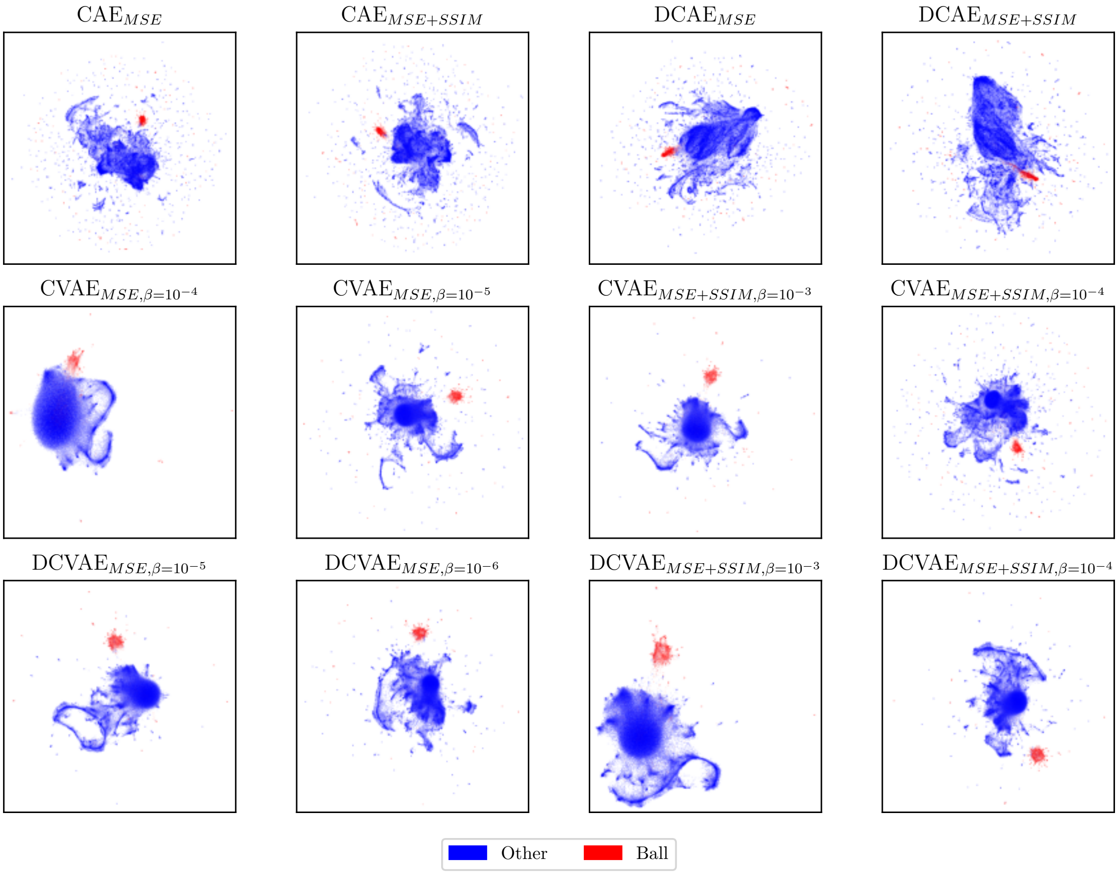


Figure 31: *Two-dimensional representation of autoencoder latent space embeddings created with the UMAP dimensionality reduction approach. Class information is encoded as color, with red samples belonging to the positive ball class and blue samples belonging to the negative class.*

model. We can clearly observe the regularizing effect on the latent space of the CVAE and DCVAE models. The effect is more pronounced in models with higher KL Divergence scaling factors β, resulting in two-dimensional representations characterized spherical clusters of embeddings. Stronger regularization also leads to a more concentrated distribution of embeddings in the latent space overall, which can be best observed by comparing the latent space visualizations of two otherwise identical variational models with differing β factors and by comparing the CAE/DCAE models with the regularized CVAE and DCVAE models. A qualitative difference in the latent space structure of denoising autoencoder models and autoencoder models trained on the reconstruction task cannot be observed in Figure 31.

4.2.1 CNN Classifier Activations

To better understand which input image features influence the autoencoder based classifier models prediction process, we use the GradCAM [52] approach to compute gradients of the models' target outputs with respect to their convolutional feature map activations. First, we remove the softmax activation from the final dense layer, then we compute the gradients of the target class output neuron logit with respect to the first and second convolutional layers activations. By performing a global average pooling of the gradients over the width and height dimensions of each channel in the output feature map of the convolutional layers, then taking a weighted combination of all averaged gradients across feature map channels and applying the activation function, we obtain a heatmap that shows which neurons have a positive influence on the target class prediction. This heatmap is then upsampled and overlaid onto the original image to highlight regions in the image that are most relevant to the models' prediction.

Figure 32 shows the combined heatmaps obtained from performing the GradCAM process for the first and second convolutional layers of the trained ball classification CNN models on eight randomly chosen sample images (4 ball samples and 4 non-ball samples). The heatmap uses a color map ranging from blue for low predictive importance to red for high predictive importance. In this figure, we can observe some interesting properties of the trained classifiers. Firs, we see that the two vanilla CAE models fail to identify some ball images as belonging to the positive class, indicated by a completely blue heatmap that shows no activations for the positive class. In this small sample, we even observe the CAE trained to minimize the combined MSE+SSIM loss to have a better Recall, matching with the results we obtained in Table 2. It is interesting to note that while the other models all show neuron activations for the positive class for ball patch images, there appears to be a difference in which image features they deem relevant to identify balls. Models like the CVAE$_{MSE+SSIM,\beta=10^{-4}}$ and DCVAE$_{MSE,\beta=10^{-5}}$ appear to focus on the soccer balls and their black spots, while others like the CVAE$_{MSE+SSIM,\beta=10^{-4}}$ appear to instead focus on the space around the balls. There are also models that seem to employ both strategies, depending on the ball patch, like the CNN trained from scratch or the autoencoder based DCVAE$_{MSE+SSIM,\beta=10^{-3}}$ model. In the authors' opinion, the DCVAE$_{MSE,\beta=10^{-5}}$ and DCVAE$_{MSE+SSIM,\beta=10^{-4}}$ models exhibit the most plausible activations. We are confident that these findings can lead us to confirm our third hypothesis, as we clearly observe in improvement both in performance metrics and qualitative model aspects by including a denoising criterion during autoencoder training and regularizing the latent space structure through CVAE models.

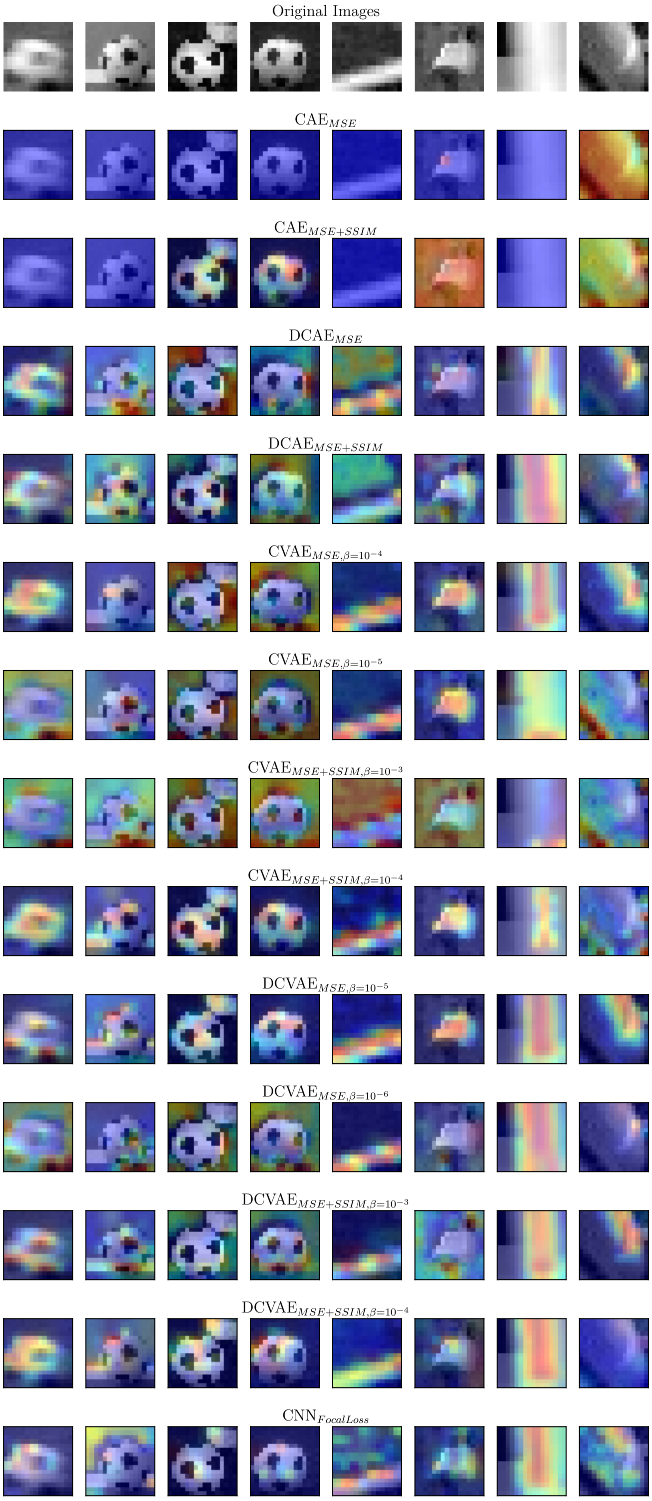


Figure 32: *Ball candidate patches with overlaid GradCAM heatmaps showing regions of convolutional neuron activation with respect to the predicted class.*

4.3 Improving Ball Classification Recall

Our autoencoder based ball classification experiments indicate that we can achieve a high level of prediction precision for the positive ball class by increasing the prediction score threshold. In doing so, we lose recall performance, as more true ball samples are erroneously classified to belong to the negative class (see Table 2). By analyzing true positive and false negative prediction samples, we can better understand which image characteristics influence a ball class prediction.

In Figure 33 we can see a representative subset of image patches correctly classified as belonging to the ball class, and in Figure 34 we see a representative subset of false negative ball class predictions. We used the $\text{DCVAE}_{MSE+SSIM,\beta=10^{-4}}$ based classifier with an optimized threshold to generate these predictions, as it is one of the highest scoring models in terms of precision, and we confirmed a similar distribution of true positive and false negative ball predictions exhibited by the other autoencoder based models. We can observe that all true positive predictions are made for balls that are fully visible in the image patch, and in most cases positioned close to the image center. Every soccer ball has clearly visible contours and exhibits at least three black panels, with the majority of the samples showing four of their black panels. In contrast, the false negative ball predictions seem to occur predominantly for off-centered and partially visible soccer balls, including blurry ball patches and many patches with three visible black panels or less. When manually sighting and annotating the ball patches, we noticed that a majority of ball samples contained fully visible balls in the image patch.

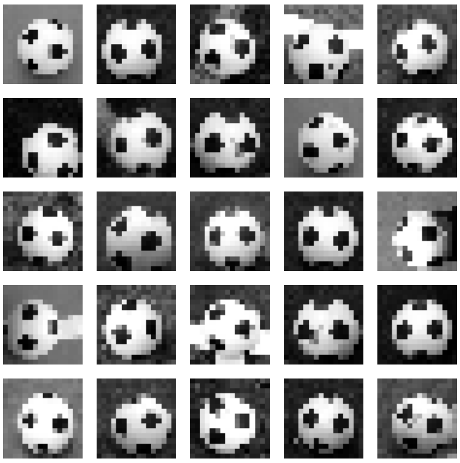


Figure 33: *Examples of true positive predictions from a CNN classifier model using a feature extractor based on a Denoising Variational Autoencoder.*

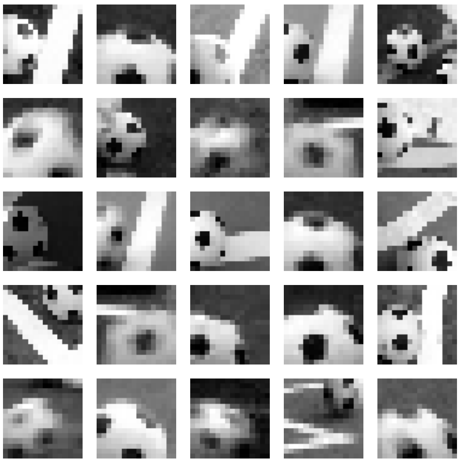


Figure 34: *Examples of false negative predictions from a CNN classifier model using a feature extractor based on a Denoising Variational Autoencoder*

We notice a similar pattern when investigating the ball position estimation target predictions, as depicted in Figure 35. Here we show the predicted ball center and radius as an overlaid red circle in the image patches. We use the DCVAE$_{MSE+SSIM,\beta=10^{-4}}$ autoencoder based regression model to make the predictions. Again, we observe good prediction performance for centered and largely visible ball patches and bad predictive performance for partially visible and off-centered ball patches in Figure 36. We notice that these observations support the research suggesting that the effective receptive fields of CNN models tend to manifest towards the center of images [20].

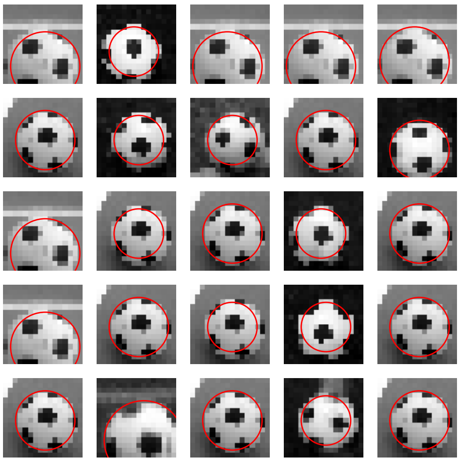


Figure 35: *Examples of accurate ball position estimation results from a CNN regression model using a feature extractor based on a Denoising Variational Autoencoder.*

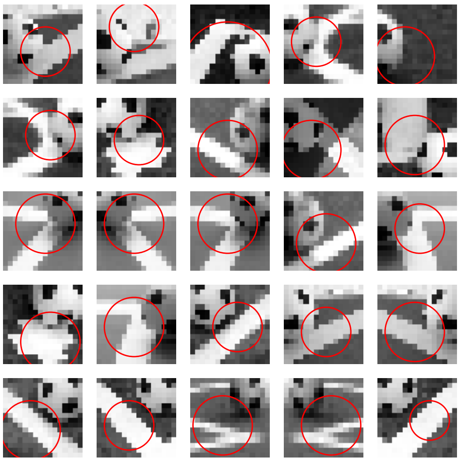


Figure 36: *Examples of inaccurate ball position estimation results from a CNN regression model using a feature extractor based on a Denoising Variational Autoencoder*

To improve both the classification and position estimation performance, we see two tractable approaches. First, we could improve the ball hypothesis algorithm to produce better, i.e. fully contained and centered, ball candidate patches. Second, we could generate more of the hard-to-classify samples by using the available bounding box information to crop partially visible balls from higher resolution image patches.

Additionally, we examined the prospect of using the generative capabilities of CVAE models to generate novel training data points by sampling from the latent space and have found that a random sampling from the latent space using the mean and standard deviation of feature embeddings across all ball samples leads to implausible reconstructions. We found that interpolating between ground truth ball patch embeddings produces more plausible reconstructions. Figure 37 shows a grid of generated interpolated ball patch reconstructions, with the four corners showing reconstructions from ball patch images and

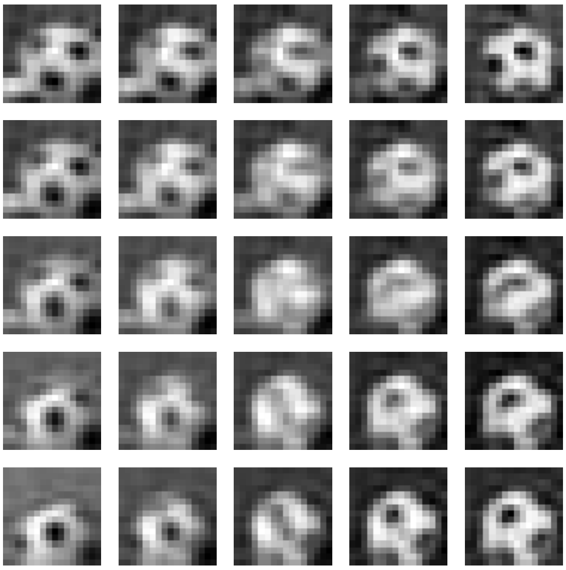


Figure 37: *Novel ball patch image reconstructions generated by interpolating between ground truth latent space embeddings. The four corners show reconstructions from ground truth embeddings, while the other grid cells display reconstructions from interpolating between the ground truth embeddings row and columnwise.*

all other grid cells showing interpolated results between these corner images. Note that the plausibility of the reconstructions diminishes towards the center of the grid, where samples are reconstructed from an interpolation of all four reference embeddings. We propose to use a conservative interpolation strategy to generate novel training samples of the ball class for future research.

4.4 Evaluation on Nao Robot

Once we have trained a new classification or position estimation model using the TensorFlow/Keras [43] machine learning framework, we need to compile it to use it for inference on our Nao robots. Currently, we use a custom compiler originally developed by the NaoDevils team [53] that we have extended over time to suit our evolving requirements. The devils-compiler takes a trained Keras model file in the *hdf5* file format as input and produces optimized C++ header and source files that can be included in our robot cognition framework. Some layer types, such as convolutional layers, can be optimized to use SIMD instructions with 128-bit registers to efficiently perform multiplication and addition operations on four 32-bit single precision floating-point numbers at a time. To enable the compilation of the autoencoder based feature extractors and classification and position estimation heads, we extended the compiler to support latent space encodings of shape *(1, 128)* as inputs and added an implementation of the sigmoid activation function for fully connected layers, which are required for the position estimation model.

4.4.1 Inference Time

With an adjusted compiler, we can now benchmark the inference time of our autoencoder based feature extractors, as well as the ball classification and position estimation models that use the latent space embeddings of the feature extractors as their inputs. First, let us quickly recap how the current CNN based ball detection process works. Recall that we use a heuristics based approach to generate ball candidate hypothesis patches. For all candidate patches, we perform a classification to determine whether the patch contains a ball or not. For patches that are classified as balls, we then perform the position estimation inference to find the ball center coordinates and radius. The number of ball candidate patches to generate and process is a configurable parameter and is currently set to 24 candidate patches. In the best-case scenario, there is only one ball candidate patch that contains an actual ball, which means we perform the classification step for all 24 candidates and the position estimation step for a single patch. In the worst case, all candidate patches supposedly contain a ball, and we need to perform both inference steps 24 times. In practice, the worst case can not truly occur in a regulation RoboCup game that uses a single soccer ball on the pitch. However, false-positive ball predictions may occur. The robot's internal ball model is capable of detecting and modelling more than one soccer ball, which might occur in practice and training scenarios.

The proposed autoencoder-based ball detection process is similar to the current ball detection approach, but uses the encoder model from a trained autoencoder architecture to create a latent space representation of all candidate patches. The latent space representation is then used as input to the classification and position estimation models. For vanilla autoencoders and autoencoders trained on the image denoising task, the feature extraction is rather efficient, as we only perform convolutional operations and compute activations, which can be optimized using SIMD vector operations. For CVAE based feature extraction, we incur a performance penalty since we need to additionally compute the mean latent space vector. This adds an extra dense layer after the flatten operation, which increases the computational complexity and memory requirements of the feature extraction process. Both the ball classification head and the position estimation head use multiple fully connected layers, making them more computationally expensive than the feature extraction step.

Model	Feature Extraction	Classification	Position Estimation	Best Case	Worst Case
CAE/DCAE	0.0224 ms	0.2908 ms	0.2914 ms	7.8082 ms	14.5104 ms
CVAE/DCVAE	0.1048 ms	0.2908 ms	0.2914 ms	9.7858 ms	16.4880 ms
CNN (Current)	-	0.3343 ms	0.4122 ms	8.4330 ms	17.9316 ms

Table 5: *Inference times measured on a Nao v6 robot. The columns Feature Extraction, Classification, and Position Estimation indicate average inference time for a single patch. The Best-Case scenario denotes the total average inference time for feature extraction and classification of 24 ball candidate patches, plus the position estimation inference for one ball patch image. The Worst-Case scenario denotes the total average inference time for feature extraction, classification, and position estimation of all 24 candidate patches.*

To benchmark the inference times of the current CNN based approach and the autoencoder based approach developed in this thesis, we developed a simple C++ test script which performs inference on random inputs for 5 seconds, computes the number of inference cycles completed within that time, and calculates the average time taken per input. This test also includes a small warm up phase of 10 inference cycles per model to stabilize performance measurements. Model inputs are randomized after each inference call. We compiled and ran the test on a Nao v6 robot to obtain the results depicted in Table 5. The biggest performance improvements materialize when there is a high number of patches that need processing by the position estimation model, since our position estimation model has fewer parameters than the current implementation, and is therefore faster. For the expected average case, where there might be up to one ball patch among all ball candidate patches, the vanilla autoencoder based approach is 8% faster than the current CNN based approach, while the CVAE feature extraction model is 16% slower. To achieve our goal of a total inference time of 8 ms to process all ball candidate patches using the CVAE based approach, we would need to decrease the number of generated candidate patches from 24 to 20.

4.4.2 Gameplay

Quantifying the impact of a new ball classification or position estimation model on gameplay performance is a notoriously hard problem. Many behaviors resulting from false-positive or false-negative ball class predictions might only be observed properly in a regulation game against an opposing team, and may not be noticeable in a practice session or during tests with a single robot. In the past, we have estimated model performance using a single practice game held out from the training data, to maximize the amount of high-quality samples available for training. Perhaps unsurprisingly, this resulted in us often over-estimating expected model performance before testing a particular model in a proper (test-) game. Even then, the assessment of model performance is highly subjective, as there is currently no established repeatable testing routine we perform to evaluate model inference on the robot. We believe that by establishing a cross-validated evaluation approach that keeps individual game halves and experiments separated across folds, we made a contribution to narrowing the gap between expected and observed model performance. However, it remains a challenge to meaningfully quantify the real-world performance of the models we created and trained in this work. We performed a basic deployment test and could confirm that our most promising CVAE based classification and regression models are indeed sufficiently precise to enable a robot to dribble a soccer ball over the pitch and score a goal. However, we feel that we are unable to make well-founded claims about an individual model's gameplay performance or draw dependable comparisons to the current classification and position estimation models. We acknowledge that the absence of a quantifiable and repeatable testing protocol for gameplay performance is a shortcoming that should be addressed in future research.

5 Conclusion

In this research, we presented an autoencoder-based model training framework for vision tasks in the context of robot soccer. We built upon an established data processing pipeline and extended it to enable the creation and curation of unlabeled image datasets. We added, confirmed and adjusted a total of over 200,000 annotations of image class labels and ball position parameters. We analyzed and evaluated the fundamental building blocks of Convolutional Autoencoder architectures to find an optimized model architecture for our use case.

Based on our findings, we trained a variety of autoencoder architectures using multiple loss functions and training tasks. We investigated the efficacy of using the trained autoencoder models as feature extractors for downstream classification and regression models and examined the qualitative aspects of these models to guide the model selection process. We believe we showed that our approach is a viable alternative to the currently used classification and ball position estimation strategy. We identified limitations and opportunities for future improvement, specifically concerning classification recall performance and position estimation regression of image patches with partially visible soccer balls. We measured acceptable inference times on a Nao v6 robot, with room for further improvements.

In Section 1 we formulated five research objectives to strive for over the course of this work. Let us re-examine these objectives and review the progress we made towards achieving them.

Generalized Training Framework We set out to develop an autoencoder based training framework for computer vision and machine learning tasks in the context of robot soccer, which is based on the game and image log data collected by the Berlin United team. We implemented a flexible dataset generation routine that integrates processed logs, bounding box annotation data and our ball hypothesis generation algorithm to create both labeled and unlabeled datasets of ball candidate image patches that can be used for a variety of tasks. We created isolated and repeatable experiments for training autoencoder architectures and using them as feature extractors for downstream classification and regression models. The workflows we developed can be easily adjusted and repurposed for other research scenarios.

Precision We showed that autoencoder based ball classification can be performed with sufficient precision, especially when adjusting prediction confidence thresholds. Without adjusted confidence thresholds, the DCAE, CVAE and DCVAE models achieve a mean precision score of around 90% in our cross-validated experiments. By adjusting confidence thresholds to the mean positive prediction confidence, we achieve a mean precision between 98% and 99% for all models except the vanilla CAE architectures. We proposed two possible approaches to further increase the recall of these models in future experiments.

Robustness We have shown that our autoencoder based approach outperforms the currently used CNN classifier model on the B-Human benchmark dataset in terms of precision and false positive rate. While this alone does not merit a claim of a more robust architecture over all, we believe that by incorporating an analysis of autoencoder latent space structure and visualizing the convolutional layers neuron activations we can make a contribution to improving the current classifier model selection process. Our autoencoder based position estimation models improve upon the currently used CNN model when evaluated on the available Nao v6 image log data.

Real-time Inference While our autoencoder based approach improves inference times in scenarios where many candidate patches are classified as balls, we were unable to reach our objective of an average case inference time to process all candidate patches in up to 8 ms using all examined autoencoder architectures. Using the most capable CVAE based feature extractors and downstream models, we achieve an average case inference time of 9.79 ms to process 24 ball candidate patches. This is still performant enough for smooth inference without risking frame loss. To further improve upon the measured inference times, we could decrease the maximum number of candidate patches to process.

Reproducibility By isolating experiments, containerizing model training code and storing training metrics and artifacts in our *MLFlow* machine learning operations server instance, we ensure reproducible experiments and access to historic results for independent analysis. Contributions to existing code and novel developments are integrated into the Berlin United team code base. Containerized workflows can be executed on a variety of hardware with minimal setup overhead. We acknowledge that tightening the gap between model training and evaluation performance remains an important subject for further research.

5.1 Limitations and Future Work

Our research has shown, that autoencoder based ball classification and position estimation is a viable approach for robotic visual perception in the RoboCup soccer challenge environment. By utilizing large amounts of unlabeled training data, we built precise and plausible feature extraction models that can be used for ball classification and position estimation. Since the data needed to train these downstream models requires manual, or at least semi-automated, annotation efforts, we expect the biggest potential improvements to come from additional high-quality, plausible and diverse training data. Our classification dataset is currently highly imbalanced with a bias towards negative class samples, with ball patching images making up only 4.3% of all available image patches.

We observed that the majority of trained models precisely classifies ball candidate images where the ball is largely visible and centered in the patch, while struggling with partially visible and off-centered balls. We found the same to be true for the ball position estimation regression models. In this work, we presented two possible strategies to combat this imbalance. We propose to use the bounding box information from our annotated high-resolution image data to deliberately create ball candidate patches with partially visible and off-centered balls through strategic cropping. Additionally, we showed that a trained Variational Autoencoder model can be used to generate novel and plausible samples by interpolating between ground truth ball sample latent space embeddings. We suggest that a performance improvement in both downstream tasks is to be expected by modifying the ball hypothesis generation to produce candidate patches that contain more of a ball's total area. We began an investigation into this hypothesis in the 2024 RoboCup world championship, where we modified the candidate generation algorithm by extending the borders of the proposed patches in both width and height. We believe the collective impressions we gathered as a team, although being highly subjective, warrant further research in this direction.

Having found a promising baseline architecture through extensive testing, we believe that past experiments with increased input image dimensions and the incorporation of additional color channel information could be re-evaluated using the new autoencoder based classification and position estimation approach.

Finally, we acknowledge that the lack of a dedicated on-device testing framework for trained ball classification and position estimation models leaves room for uncertainty when estimating expected model performance. We think this is a promising area for further research.

Appendix

References

[1] SoftBank Robotics America Inc. *NAO: Personal Robot Teaching Assistant — SoftBank Robotics America.* `https://us.softbankrobotics.com/nao` (visited on: visited on 07/30/2024).

[2] *RoboCup Standard Platform League.* `https://spl.robocup.org/` (visited on: visited on 07/30/2024).

[3] Heinrich Mellmann et al. *Berlin United - Nao Team Humboldt Team Report 2019 DRAFT v0.* 2019.

[4] Hiroaki Kitano et al. "RoboCup: The Robot World Cup Initiative". In: *Proceedings of the first international conference on Autonomous agents - AGENTS '97.* the first international conference. Marina del Rey, California, United States: ACM Press, 1997, pp. 340–347. ISBN: 978-0-89791-877-0. DOI: `10.1145/267658.267738`.

[5] *RoboCup A Brief History of RoboCup.* `https://www.robocup.org/a_brief_history_of_robocup` (visited on: visited on 07/29/2024).

[6] *Meet NAO — NAO 6 — Aldebaran.* `https://www.aldebaran.com/en/support/nao-6/1-meet-nao` (visited on: visited on 07/30/2024).

[7] *Specifications_NAO6.pdf.* `https://www.generationrobots.com/media/Specifications_NAO6.pdf` (visited on: visited on 09/06/2024).

[8] "RoboCup Standard Platform League (NAO) Rule Book". In: ().

[9] *Video cameras — Aldebaran 2.8.7.4 documentation.* `http://doc.aldebaran.com/2-8/family/nao_technical/video_naov6.html` (visited on: visited on 09/06/2024).

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* Adaptive computation and machine learning. Cambridge, Mass: The MIT press, 2016. 168-171. ISBN: 978-0-262-03561-3. DOI: `10.1007/s10710-017-9314-z`.

[11] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton.* Cornell Aeronautical Laboratory, 1957.

[12] Seppo Linnainmaa. "Taylor expansion of the accumulated rounding error". In: *BIT* 16.2 (June 1976), pp. 146–160. ISSN: 0006-3835, 1572-9125. DOI: `10.1007/BF01931367`.

[13] Yann A. LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade.* Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Vol. 7700. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48. ISBN: 978-3-642-35288-1. DOI: `10.1007/978-3-642-35289-8_3`.

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521.7553 (May 28, 2015), pp. 436–444. ISSN: 0028-0836, 1476-4687. DOI: `10.1038/nature14539`.

[15] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 00189219. DOI: `10.1109/5.726791`.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet classification with deep convolutional neural networks”. In: *Communications of the ACM* 60.6 (May 24, 2017), pp. 84–90. ISSN: 0001-0782, 1557-7317. DOI: `10.1145/3065386`.

[17] Christian Szegedy et al. “Going deeper with convolutions”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Boston, MA, USA: IEEE, June 2015, pp. 1–9. ISBN: 978-1-4673-6964-0. DOI: `10.1109/CVPR.2015.7298594`.

[18] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Las Vegas, NV, USA: IEEE, June 2016, pp. 770–778. ISBN: 978-1-4673-8851-1. DOI: `10.1109/CVPR.2016.90`.

[19] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Las Vegas, NV, USA: IEEE, June 2016, pp. 779–788. ISBN: 978-1-4673-8851-1. DOI: `10.1109/CVPR.2016.91`.

[20] Wenjie Luo et al. “Understanding the Effective Receptive Field in Deep Convolutional Neural Networks”. In: *Advances in neural information processing systems* 29 (2016). arXiv: `1701.04128[cs]`.

[21] *TikZ/2D Convolution · PetarV-/TikZ - MIT License.* GitHub. `https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution` (visited on: visited on 09/06/2024).

[22] H. Bourlard and Y. Kamp. “Auto-association by multilayer perceptrons and singular value decomposition”. In: *Biological Cybernetics* 59.4 (Sept. 1988), pp. 291–294. ISSN: 0340-1200, 1432-0770. DOI: `10.1007/BF00332918`.

[23] Pierre Baldi and Kurt Hornik. “Neural networks and principal component analysis: Learning from examples without local minima”. In: *Neural Networks* 2.1 (Jan. 1989), pp. 53–58. ISSN: 08936080. DOI: `10.1016/0893-6080(89)90014-2`.

[24] M. A. Kramer. “Autoassociative neural networks”. In: *Computers & Chemical Engineering.* Neutral network applications in chemical engineering 16.4 (Apr. 1, 1992), pp. 313–328. ISSN: 0098-1354. DOI: `10.1016/0098-1354(92)80051-A`.

[25] G. E. Hinton and R. R. Salakhutdinov. “Reducing the Dimensionality of Data with Neural Networks”. In: *Science* 313.5786 (July 28, 2006), pp. 504–507. ISSN: 0036-8075, 1095-9203. DOI: `10.1126/science.1127647`.

[26] Pascal Vincent et al. “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion”. In: *Journal of machine learning research* 11 (2010), pp. 3371–3408. ISSN: 1532-4435.

[27] Z. Wang et al. “Image Quality Assessment: From Error Visibility to Structural Similarity”. In: *IEEE Transactions on Image Processing* 13.4 (Apr. 2004), pp. 600–612. ISSN: 1057-7149. DOI: `10.1109/TIP.2003.819861`.

[28] Tsung-Yi Lin et al. “Focal Loss for Dense Object Detection”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.2 (Feb. 1, 2020), pp. 318–327. ISSN: 0162-8828, 2160-9292, 1939-3539. DOI: `10.1109/TPAMI.2018.2858826`.

[29] Bernd Poppinga and Tim Laue. “JET-Net: Real-Time Object Detection for Mobile Robots”. In: *RoboCup 2019: Robot World Cup XXIII.* Vol. 11531. Cham: Springer International Publishing, 2019, pp. 227–240. ISBN: 978-3-030-35698-9. DOI: `10.1007/978-3-030-35699-6_18`.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* Adaptive computation and machine learning. Cambridge, Mass: The MIT press, 2016. 131-135. ISBN: 978-0-262-03561-3. DOI: `10.1007/s10710-017-9314-z`.

[31] Jonathan Masci et al. “Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction”. In: *Artificial Neural Networks and Machine Learning – ICANN 2011.* Vol. 6791. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 52–59. ISBN: 978-3-642-21734-0. DOI: `10.1007/978-3-642-21735-7_7`.

[32] Marc’Aurelio Ranzato et al. “Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition”. In: *2007 IEEE Conference on Computer Vision and Pattern Recognition.* 2007 IEEE Conference on Computer Vision and Pattern Recognition. Minneapolis, MN, USA: IEEE, June 2007, pp. 1–8. ISBN: 978-1-4244-1179-5. DOI: `10.1109/CVPR.2007.383157`.

[33] Bo Du et al. “Stacked Convolutional Denoising Auto-Encoders for Feature Representation”. In: *IEEE Transactions on Cybernetics* 47.4 (Apr. 2017), pp. 1017–1027. ISSN: 2168-2267, 2168-2275. DOI: `10.1109/TCYB.2016.2536638`.

[34] Yihui Xiong and Renguang Zuo. “Robust Feature Extraction for Geochemical Anomaly Recognition Using a Stacked Convolutional Denoising Autoencoder”. In: *Mathematical Geosciences* 54.3 (Apr. 2022), pp. 623–644. ISSN: 1874-8961, 1874-8953. DOI: `10.1007/s11004-021-09935-z`.

[35] Diederik P. Kingma and Max Welling. “An Introduction to Variational Autoencoders”. In: *Foundations and Trends® in Machine Learning* 12.4 (2019), pp. 307–392. ISSN: 1935-8237, 1935-8245. DOI: `10.1561/2200000056`.

[36] Diederik P Kingma and Max Welling. *Auto-Encoding Variational Bayes.* Version Number: 11. 2013. DOI: `10.48550/ARXIV.1312.6114`.

[37] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. "Stochastic Backpropagation and Approximate Inference in Deep Generative Models". In: *International conference on machine learning.* PMLR, 2014, pp. 1278–1286. DOI: `10.48550/arXiv.1401.4082`.

[38] Thomas Röfer et al. "B-Human 2019 – Complex Team Play Under Natural Lighting Conditions". In: *RoboCup 2019: Robot World Cup XXIII.* Vol. 11531. Cham: Springer International Publishing, 2019, pp. 646–657. ISBN: 978-3-030-35699-6. DOI: `10.1007/978-3-030-35699-6_52`.

[39] Rico Tilgner et al. *Nao-Team HTWK Team Research Report 2019.* 2019. `https://robots.htwk-leipzig.de/fileadmin/portal/m_nao/Publikationen/HTWK_TRR_2019.pdf` (visited on: visited on 07/30/2024).

[40] Christian Szegedy et al. "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Las Vegas, NV, USA: IEEE, June 2016, pp. 2818–2826. ISBN: 978-1-4673-8851-1. DOI: `10.1109/CVPR.2016.308`.

[41] Andrea Essig et al. *HULKs Team Research Report 2021.* 2021. `https://hulks.de/_files/TRR_2021.pdf` (visited on: visited on 07/30/2024).

[42] Arne Moos. *Efficient Single Object Detection on Image Patches with Early Exit Enhanced High-Precision CNNs.* Sept. 7, 2023. arXiv: `2309.03530[cs]`. `http://arxiv.org/abs/2309.03530` (visited on: visited on 07/29/2024).

[43] TensorFlow Developers. *TensorFlow.* Version v2.15.0. July 11, 2024. DOI: `10.5281/ZENODO.4724125`.

[44] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* Version Number: 9. 2014. DOI: `10.48550/ARXIV.1412.6980`. `https://arxiv.org/abs/1412.6980` (visited on: visited on 08/22/2024).

[45] Lutz Prechelt. "Early Stopping - But When?" In: *Neural Networks: Tricks of the Trade.* Ed. by Genevieve B. Orr and Klaus-Robert Müller. Red. by Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen. Vol. 1524. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 55–69. ISBN: 978-3-540-65311-0. DOI: `10.1007/3-540-49430-8_3`.

[46] Stella Alice Schlotter. "Objekterkennung im RoboCup Kontext". Masters Thesis. 2022.

[47] Rejin Varghese and Sambath M. “YOLOv8: A Novel Object Detection Algorithm with Enhanced Performance and Robustness”. In: *2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS)*. 2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS). Chennai, India: IEEE, Apr. 18, 2024, pp. 1–6. DOI: `10.1109/ADICS58448.2024.10533619`.

[48] Nathalie Japkowicz and Mohak Shah. *Evaluating Learning Algorithms: A Classification Perspective*. 1st ed. Cambridge University Press, Jan. 17, 2011. ISBN: 978-0-511-92180-3. DOI: `10.1017/CBO9780511921803`.

[49] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *International conference on machine learning*. International conference on machine learning. pmlr, 2015, pp. 448–456. DOI: `10.48550/arXiv.1502.03167`.

[50] Shibani Santurkar et al. “How Does Batch Normalization Help Optimization?” In: *Advances in neural information processing systems* 31 (2018), pp. 2488–2498. DOI: `10.48550/arXiv.1805.11604`.

[51] Leland McInnes et al. “UMAP: Uniform Manifold Approximation and Projection”. In: *Journal of Open Source Software* 3.29 (Sept. 2, 2018), p. 861. ISSN: 2475-9066. DOI: `10.21105/joss.00861`.

[52] Ramprasaath R. Selvaraju et al. “Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization”. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017 IEEE International Conference on Computer Vision (ICCV). Venice: IEEE, Oct. 2017, pp. 618–626. ISBN: 978-1-5386-1032-9. DOI: `10.1109/ICCV.2017.74`.

[53] Oliver Urbann et al. “A C Code Generator for Fast Inference and Simple Deployment of Convolutional Neural Networks on Resource Constrained Systems”. In: *2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*. 2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS). Vancouver, BC, Canada: IEEE, Sept. 2020, pp. 1–7. ISBN: 978-1-72819-615-2. DOI: `10.1109/IEMTRONICS51293.2020.9216395`.

Acronyms

AE Autoencoder. 14

ANN Artificial Neural Network. 6

BCE Binary Cross-Entropy. 9, 17–19, 38–40

CAE Convolutional Autoencoder. 19, 20, 24, 28, 37, 43, 46–50, 52, 55, 57–59, 64, 66

CNN Convolutional Neural Network. xii, xiii, 6, 10–13, 19, 20, 24–26, 31, 35–37, 42, 45, 46, 52–56, 59, 61, 62, 64, 65, 67

CPU Central Processing Unit. 25, 29

CVAE Convolutional Variational Autoencoder. 21–23, 28, 50–52, 54–59, 62, 64–67

DCAE Denoising Convolutional Autoencoder. 20, 28, 48, 49, 57, 58, 64, 66

DCVAE Denoising Convolutional Variational Autoencoder. 28, 51, 58, 64, 66

DoF Degrees of Freedom. 3

ELBO Evidence Lower Bound. 22, 23

FoV Field of View. 3

GPU Graphics Processing Unit. 29

GUI Graphical User Interface. 35

IoU Intersection over Union. 18, 19, 55, 56

MBGD Mini-Batch Gradient Descent. 10

MLE Maximum Likelihood Estimation. 17, 19, 22

MLP Multi-Layer Perceptron. 7

MSE Mean Squared Error. viii–xi, 9, 16–19, 23, 28, 38–42, 44, 46, 47, 49–52, 55, 56, 59

PCA Principal Component Analysis. 14

ReLU Rectified Linear Unit. 7

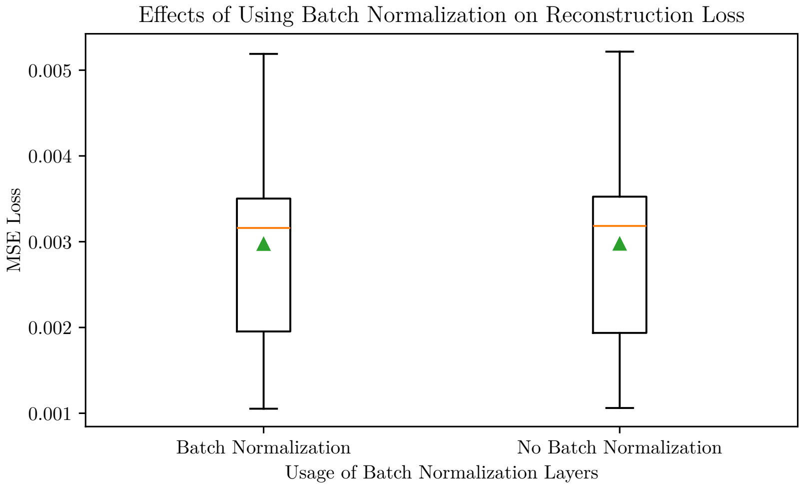
SIMD Single Instruction Multiple Data. 25, 63, 64

SPA Single Page Application. 37

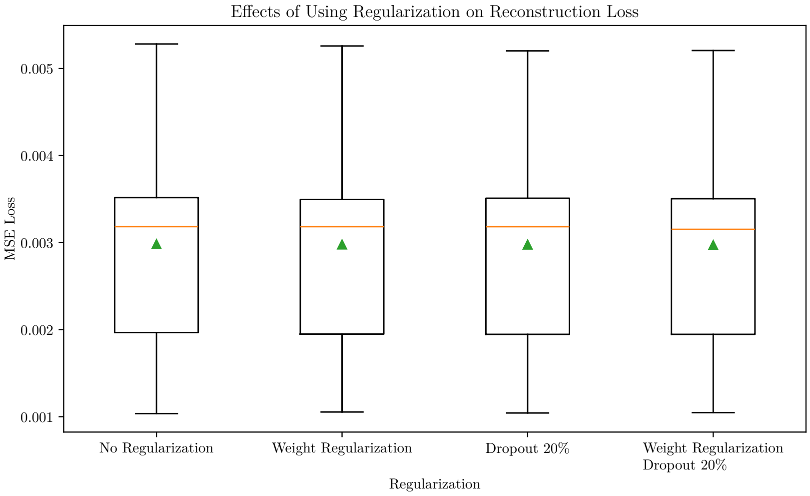
SPL Standard Platform League. 1, 2, 24

SSIM Structural Similarity Index Measure. xi, 17, 18, 38–40, 47, 49–51, 59

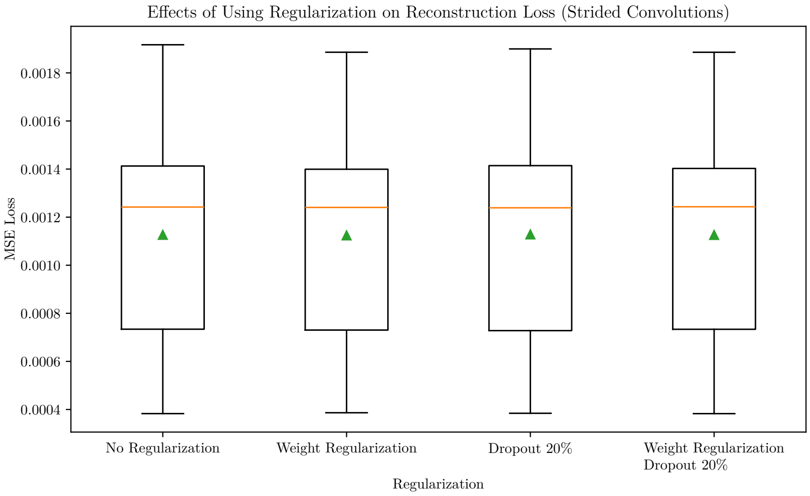
UMAP Uniform Manifold Approximation and Projection. 57, 58



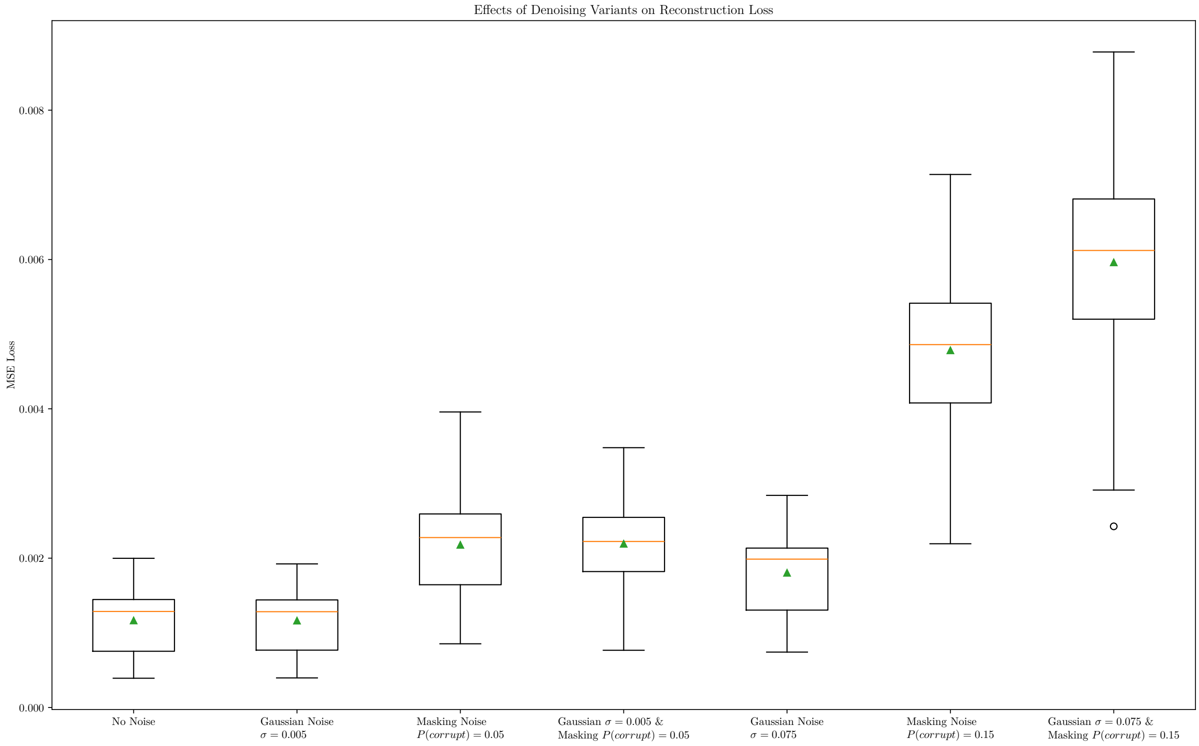
Appendix figure 1: *Box plot of 10×10 cross-validation results comparing the effects of batch normalization on reconstruction loss in Convolutional Autoencoders. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes whether batch normalization was used during training or not. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*



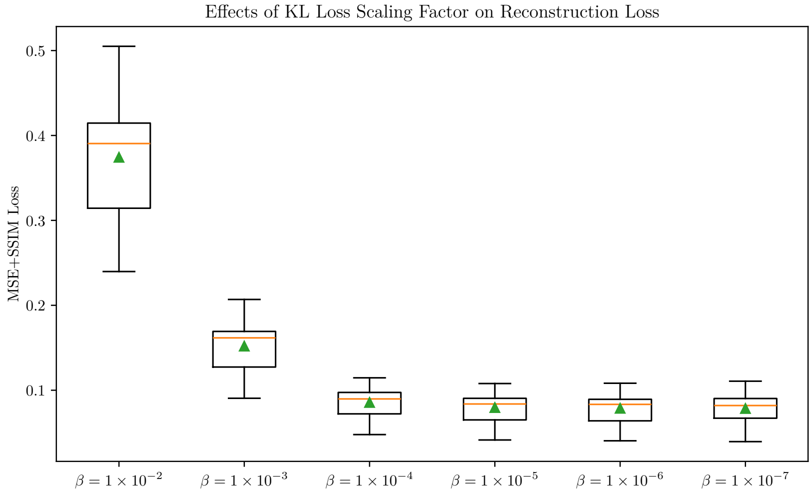
Appendix figure 2: *Box plot of 10×10 cross-validation results comparing the effects of regularization techniques on reconstruction loss in Convolutional Autoencoders using max-pooling. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes which regularization technique was used during training. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*



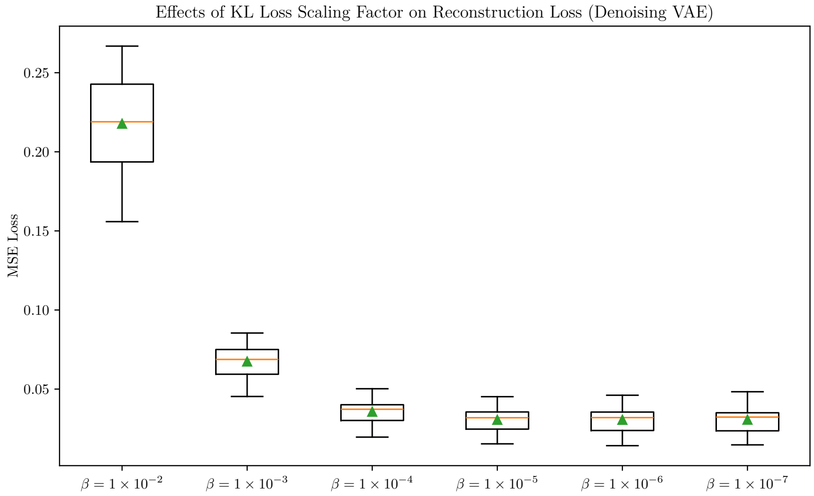
Appendix figure 3: *Box plot of 10×10 cross-validation results comparing the effects of regularization techniques on reconstruction loss in Convolutional Autoencoders using strided convolutions. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes which regularization technique was used during training. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*



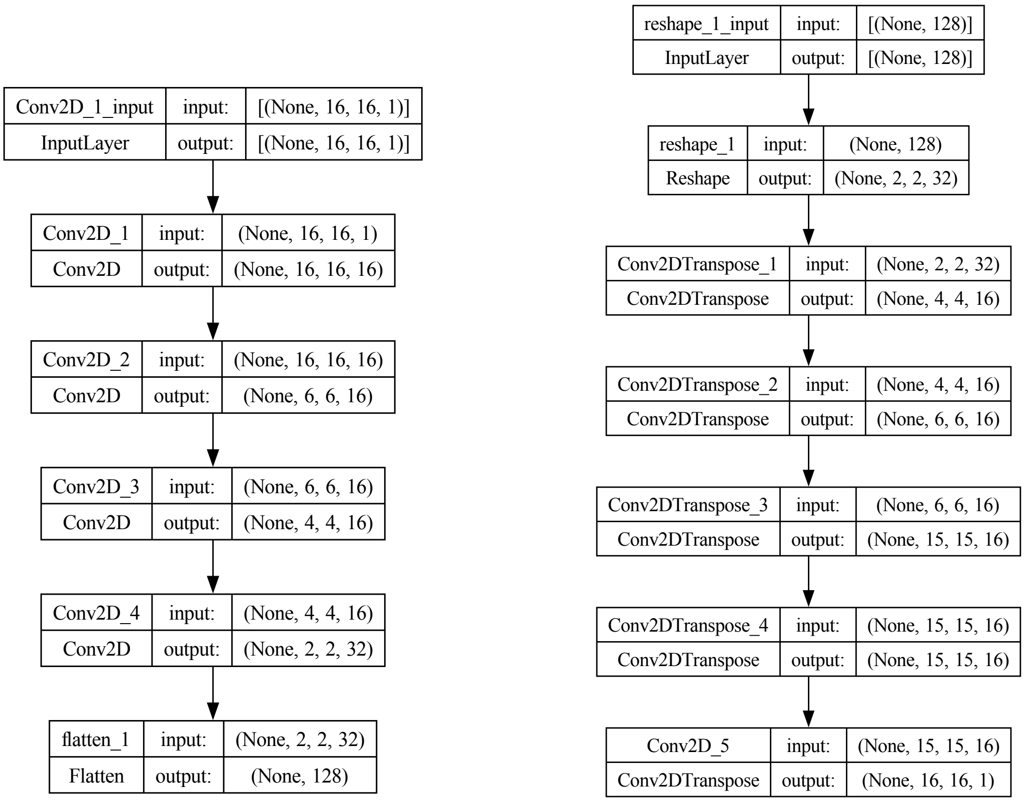
Appendix figure 4: *Box plot of 10×10 cross-validation results comparing the effects of image corruption techniques on reconstruction loss in Denoising Convolutional Autoencoders. The y-axis represents the MSE loss value computed on the evaluation sets over all trials. The x-axis denotes which type of image corruption was used on inputs during training. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*



Appendix figure 5: *Box plot of 10×10 cross-validation results comparing the effects of the KL loss scaling factor on reconstruction loss in Convolutional Variational Autoencoders. The y-axis represents the combined MSE+SSIM loss value computed on the evaluation sets over all trials. The x-axis denotes the KL loss scaling factor β used during training. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*



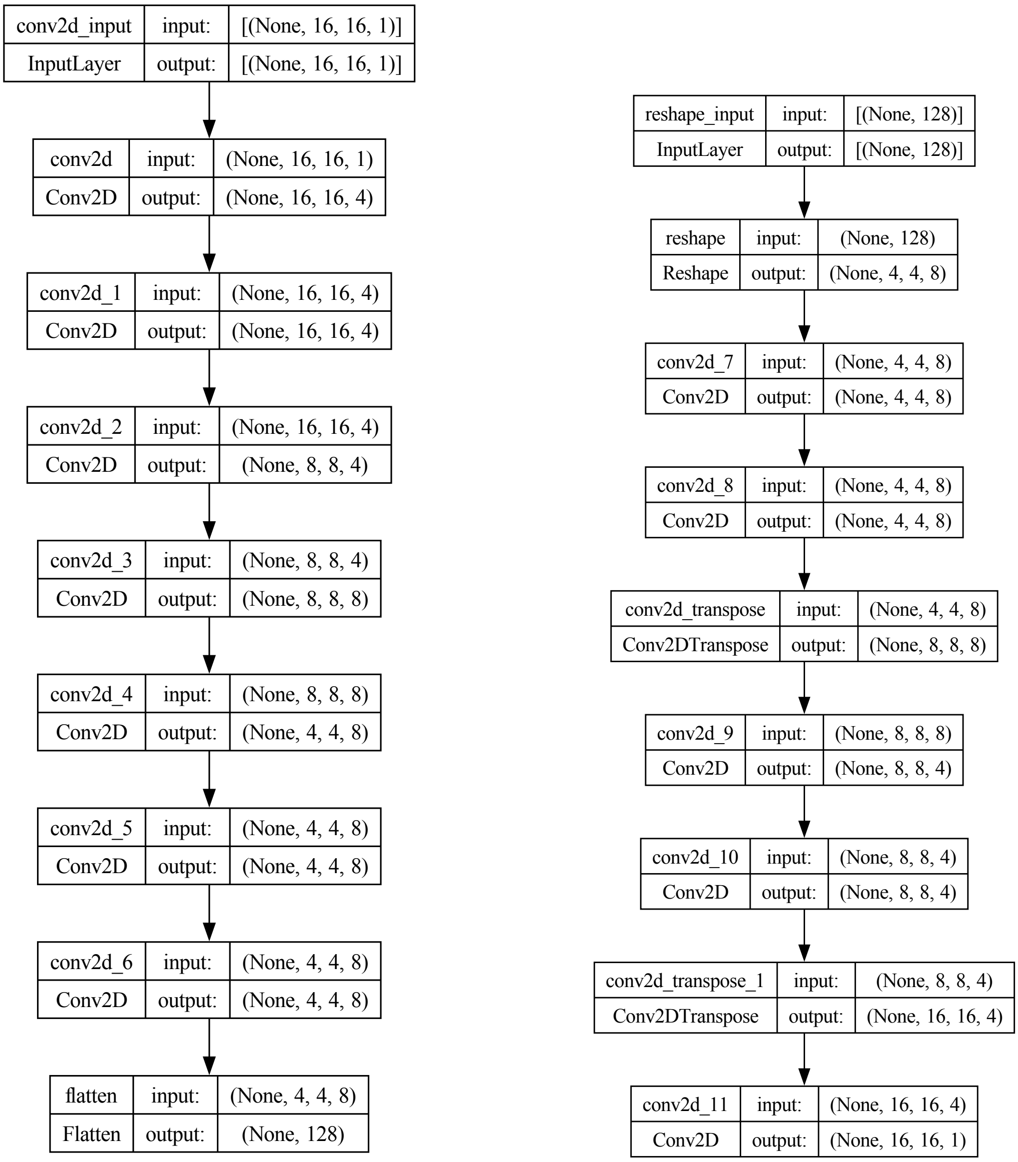
Appendix figure 6: *Box plot of 10×10 cross-validation results comparing the effects of the KL loss scaling factor on reconstruction loss in Denoising Convolutional Variational Autoencoders. The y-axis represents the combined MSE+SSIM loss value computed on the evaluation sets over all trials. The x-axis denotes the KL loss scaling factor β used during training. Boxes show the interquartile range, median (orange line), mean (green triangle) and variability. Whiskers extend to 1.5xIQR.*



(a) Encoder

(b) Decoder

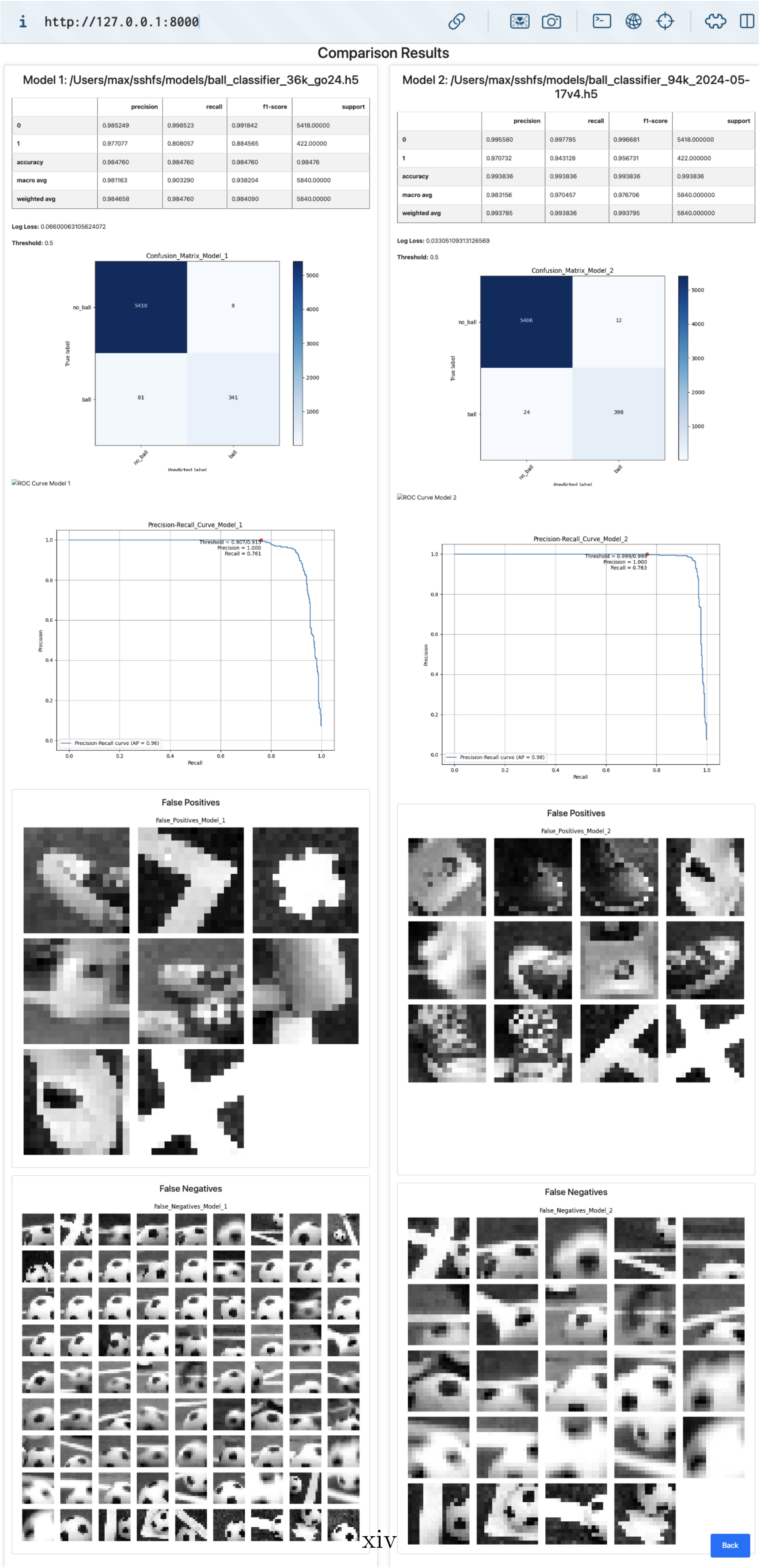
Appendix figure 7: *Autoencoder architecture based on the currently used CNN classification model of the Berlin United team (since 2024).*



(a) Encoder

(b) Decoder

Appendix figure 8: *Autoencoder architecture based on the previously used CNN classification model of the Berlin United team (2019-2024).*



http://127.0.0.1:8000

Comparison Results

Model 1: /Users/max/sshfs/models/ball_classifier_36k_go24.h5

	precision	recall	f1-score	support
0	0.985249	0.998523	0.991842	5418.00000
1	0.977077	0.808057	0.884565	422.00000
accuracy	0.984760	0.984760	0.984760	0.98476
macro avg	0.981163	0.903290	0.938204	5840.00000
weighted avg	0.984658	0.984760	0.984090	5840.00000

Log Loss: 0.06600063105624072

Threshold: 0.5

ROC Curve Model 1

False Positives

False Negatives

Model 2: /Users/max/sshfs/models/ball_classifier_94k_2024-05-17v4.h5

	precision	recall	f1-score	support
0	0.995580	0.997785	0.996681	5418.000000
1	0.970732	0.943128	0.956731	422.000000
accuracy	0.993836	0.993836	0.993836	0.993836
macro avg	0.983156	0.970457	0.976706	5840.000000
weighted avg	0.993785	0.993836	0.993795	5840.000000

Log Loss: 0.03305109313126569

Threshold: 0.5

ROC Curve Model 2

False Positives

False Negatives

Appendix figure 9: *Web-based classification model comparison tool we developed for classification model selection in preparation for RoboCup 2024.*

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den October 2, 2024 ....