University of Amsterdam
Faculty of Science
The Netherlands

Dutch Nao Team

Technical Report 2024

Students:
Harold Ruiter
Gijs de Jong
Macha Meijer
Marina Orozco González
Mark Honkoop
Julia Blaauboer
Rick van der Veen
Fyor Klein Gunnewiek
Morris de Haan
Fiona Nagelhout
Joost Weerheim
Stephan Visser
Juell Sprott
John Yao

Supervisor:
Arnoud Visser

December 30, 2024

Contents

1 Introduction

The Dutch Nao Team (DNT) is a RoboCup team founded in 2010 [1], that competes in the Standard Platform League (SPL). The team was formed after the move from AIBO to NAO robots in the SPL starting from 2008 onwards. Before its founding, the team had been active as the Dutch AIBO Team since 2003 [2].

This document serves as an overview to the changes made this year within the Dutch Nao Team, on an organizational and technical level. Chapter 2 discusses the team structure and our approach to working as a student team. Chapters 3 through 8 focus on the Dutch Nao Team software. Chapter 9 lists the activities we organized and participated in this last year. Chapter 10 highlights our planned goals for the coming year. Chapter 11 lists the contributions of all team members that contributed to this report. Finally, Chapter 12 concludes the report with a small summary.

2 Team Structure

To make active progress as a team, it is required to have people involved in technical and organizational tasks. In this section, we will describe the different layers of the team structure and the updates that have been implemented with respect to previous years [3] [4] [5].

This year, a completely new organizational structure that aims to achieve a more clearly divided and transparent organization of the team was introduced. A scheme of this structure can be found in Figure 1 and a description of each role can be found in the following subsections.

2.1 Board

The board is the legal entity behind the Dutch Nao Team. The board is part of the Dutch Nao Team Foundation. The board is primarily tasked with the monetary and administrative duties related to the team. The treasurer is responsible for creating an overview of the yearly budget and handling all money related issues. The secretary is responsible for communication within the team and with external parties. The president is more of a generalist helping out where needed and leads the board meetings. The president is also responsible for the legality of the actions of the board. This year, Lasse van Iterson acted as president of the board. Dário Xavier Catarrinho acted as Secretary of the board until April, and took on the role of vice-chair from April on. Joost Weerheim took over the role of secretary. Until April, Derck Prinzhorn was vice-chair of the board and Jurgen de Heus was

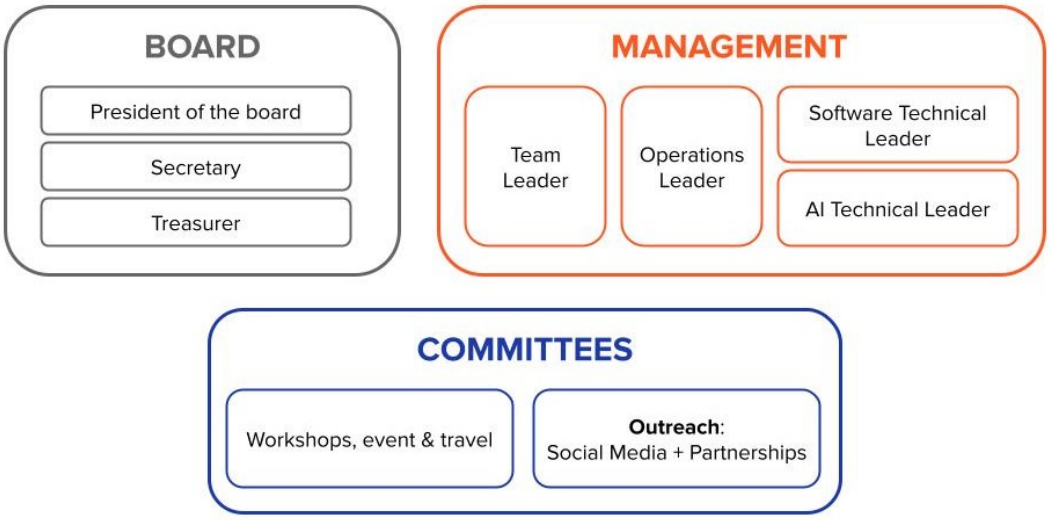


Figure 1: Overview of the new team structure.

the treasurer of the board. From April on, Stephan Visser took on the role as treasurer.

2.2 Management

From a high-level perspective, the management team is in charge of setting the goals the team plans to achieve within the year, drawing a roadmap to achieve it, checking each member's contribution and making sure that the team's activities, recruiting of new members, and future existence are secured. The management does weekly meetings discussing these tasks and other important topics that might arise.

As all team members are putting in time on a voluntary basis and the number of members has grown significantly in the last years, we found that having only a team lead managing most of these tasks was an excessively large task. We therefore decided to delegate these responsibilities to multiple people.

Within management, we have four different roles: the team leader, the operations leader, and two technical leaders for each of the technical subteams: AI and software.

The team leader is still the main point of contact in many situations and should be knowledgeable about everything that is going on in the team. The task of the team leader is to provide the team with the general strategic vision for the year and to ensure that the roadmap items are being achieved. They also do the

recruitment interviews together with the tech team leader that is most aligned with a potential new member's interest. In the year 2023-2024, Jakob Kaiser and Ross Geurts acted as team leaders. From September, Harold Ruiter took over the role of team leader.

The operations leader is a new role that focuses on operational and administrative duties. They make sure we are signed up for events on time and that accommodation and insurance are handled. Apart from that, their task is to check on the committees so that they are functioning properly. The role of operations lead was introduced in September. From September onwards, Marina Orozco Gonzáles acted as operations lead.

The technical team leads are the people with in-depth knowledge on either the software or artificial intelligence (AI) domain. They handle their respective weekly tech team meetings and support people if they need help with more specific questions. Together with the team lead, they develop the yearly roadmap. This year, Gijs de Jong acted as software lead and Macha Meijer acted as AI lead.

While each of these roles has their specific main tasks, the weekly management meetings ensure that they are all updated on each other's progress. This solves the problem of the work being too much for any single person to handle and also allows the management members to check in and help out if the need arises.

2.2.1 Roadmap

The yearly roadmap is scheduled in 3 big sprints, towards the Robotics Open Hamburg Workshop (RoHOW), RoboCup German Open and RoboCup respectively. After each of these events, the roadmap is evaluated by the management team on progress and adjusted accordingly. In order to avoid slowing down development in between events, additional practice matches are scheduled in between, acting as soft sprint goals.

2.3 Tech teams

Technical development in the team takes place in roughly two areas: software development and AI projects. Therefore, for the technical development, there are two subteams. Everyone in the team is part of either the software or the AI team. In the subteams, everyone is assigned one or multiple projects to work on. The projects are determined by both the interest of the team member and the roadmap which is defined by the management. To collaborate, help each other, and brainstorm, both subteams participate in weekly meetings. In these meetings, progress and potential issues regarding everyone's project are discussed, as well as

ways to continue the projects. The technical teams are led by the technical leads, who are also the direct point of contact of the subteam members. Collaboration between subteams, which is needed when, for example, AI projects need to be incorporated in the framework, goes through the subteam leads.

2.4 Committees

Like any other organization, the Dutch Nao Team requires external exposure and resources to fund its different outlays of the team. To address this, the team was organized into three different committees: Workshops and Events, Social Media, and Partnerships. However, due to the limited number of team members, the Social Media and Partnerships committees are combined under the Outreach committee. This year, Fiona Nagelhout acted as committee lead for Workshops and Events, and Gijs de Jong acted as committee lead for Outreach.

The partnerships committee is in charge of finding, approaching and sustaining contact with sponsors that could finance the team. This includes designing proposals templates for potential sponsors, developing outreach strategies, managing the mail contact of the team, and ensuring the terms of sponsorship agreements are adhered to and fulfilled.

The other committees are closely connected to the Partnerships committee, as our primary offerings to sponsors involve event organization and social media exposure. In collaboration with the Partnerships committee, the Workshops and Events committee not only organizes sponsor-related events, but also prepares presentations, workshops, and activities tailored for educational institutions and visitors, ranging from primary school students to members of the university community.

The Social Media committee ensures the creation of audiovisual content showcasing the team and its progress, keeping our social media active and the website up-to-date. This requires maintaining an appealing and cohesive feed that effectively captures the essence of our team for external audiences. Currently, we are on Instagram, LinkedIn and Youtube, and our goal is to post at least once per month, covering all competitions and trips we attend.

3 Framework

This section describes our framework `yggdrasil`, and the tooling surrounding it. Since 2022, we have been developing our robotics code in the Rust programming language. We started off building on the HULKs framework [6], but in 2023 we started creating our own framework from scratch, in order to deepen the under-

standing of the NAO within the team and improve on what we believe to be a shortcoming of the currently available frameworks. Concretely, we put a heavy emphasis on creating a fully parallelizable design that future proofs the team for future iterations of the NAO, as it is likely have more processing power and physical cores available. A current trend within the SPL is to have each system running at a different frequency assigned to a dedicated thread. With `yggdrasil`, systems are (unless specified) independent of threads and instead executed in parallel as tasks on the application's different thread pools. This gives us two major benefits:

- Because systems are no longer tied to specific threads, we can simply schedule more tasks on threads that are not executing anything, allowing for more complete utilization of the full processing power of the NAO.
- When a new model of the NAO with more available cores/threads releases, we can simply scale up the size of the thread pool to fully utilize the extra processing power.

3.1 Moving over to Bevy

Last year, we developed our own dependency injection and task scheduling library called Tyr. The internal design and outwards-facing API of Tyr were heavily inspired by the design of the Bevy game engine [7]. It might not seem obvious at first, but the requirements of creating video games are very similar to those of robot programming. Firstly, there is the need for high performance. Both games and the NAO are soft real-time systems, in the sense that a game always should to render to the screen within a certain time (usually around 30 or 60 Hz), and the NAO should read the sensor state and send a control message to the LoLA socket (82 Hz). This means that the underlying framework should allow us to easily write high performance code when needed. Secondly, it is important that many different interacting systems work together, but are still modularized such that they can be effortlessly inspected, modified, or even disabled without breaking the whole application. In the context of the Dutch Nao Team, this quality is especially good to have, as it allows us to more easily compare old versus new versions of modules and therefore iterate on code faster.

This year, we dropped Tyr entirely in favor of using Bevy. The reasoning is as follows:

- We found that there were a lot of features and capabilities getting added to Bevy, many of which we had to manually port over, costing us development time in a relatively complex part of the codebase. Now, we can enjoy new features with every Bevy release cycle, which is almost every three months.

- The Bevy ecosystem is large and rapidly growing, which means there are several open-source plugins and libraries built by other people we can use.
- While we initially chose to build our own library in order to have more control over our API surface and dependencies, we found ourselves converging more and more to the ones provided by Bevy. We found that we could get a sufficiently lean environment by disabling a lot of Bevy components like the renderer and asset capabilities.
- We now gain a fully-featured Entity Component System (ECS). The ECS model is an example of data-oriented design and strongly encourages clean, decoupled systems by forcing the programmer to break up app data and logic into its core components, making parallelism easier. It also helps to make your code faster by optimizing memory access patterns as it internally uses a Structure of Arrays to store data. While Tyr already introduced the ECS concept of systems, internally data was stored as singletons and it did not feature the entity/component part of ECS.

Since the design of Tyr was already heavily based on Bevy, a lot of the code was effortlessly ported over (see Figure 2.). Much of the remaining work is left in converting the existing modules to be more idiomatic, using the full capabilities of the ECS model.

```rust
// Simple example in Tyr
#[system]
fn count(counter: &mut Counter) -> Result<()> {
    println!("{counter:?}");
    *counter += 1;
    Ok(())
}

fn main() {
    App::new()
        .init_resource::<Counter>()?
        .add_system(count)
        .run()
}
```

```rust
// Simple example in Bevy
fn count(mut counter: ResMut<Counter>) {
    println!("{counter:?}");
    *counter += 1;
}

fn main() {
    App::new()
        .add_plugins(MinimalPlugins)
        .init_resource::<Counter>()?
        .add_systems(Update, count)
        .run()
}
```

Figure 2: A simple example that prints and increments a counter in both Tyr and Bevy.

3.2 Hardware abstraction

Last year, we developed `nidhogg`, a layer built on top of LoLA. As `nidhogg` is an abstraction layer over LoLA, the library has full control over the data structures used for storing robot data. Therefore, the data structures in `nidhogg` are designed to be intuitive to use and change, making it simpler for developers to create and refine robot behaviors without unnecessary hurdles.

Another benefit of building `nidhogg` as an abstraction layer for LoLA is that it allows the usage of different back-ends for `nidhogg`. This makes it possible for `yggdrasil` to run on different platforms, including simulation environments. The ability to run `yggdrasil` in simulations is significant as it makes the application of various machine learning techniques possible, while also providing a safe environment to run tests without risking real hardware.

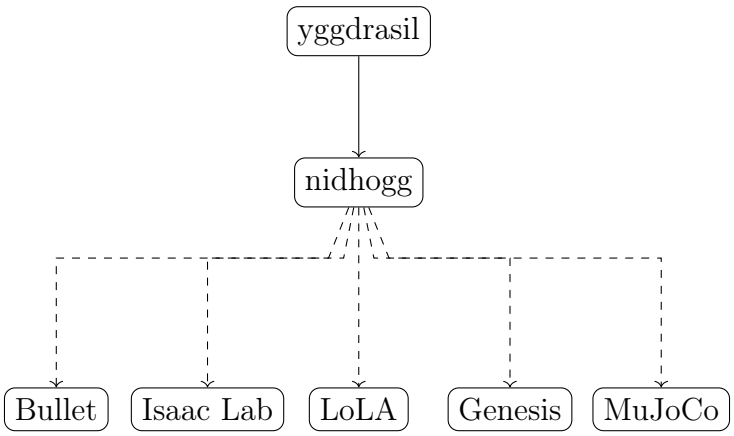


Figure 3: The nidhogg abstraction layer can have multiple backends which are transparent to the user.

This year, we started work on expanding the scope of `nidhogg`, so that it can be used as not just a layer wrapping LoLA, but more of the entire hardware environment. This change is made in order to run larger parts of the framework in a local environment (see subsection 3.3 for more info on the local feature).

3.3 Deployment

`sindri` is a development tool designed to enhance the efficiency of the development process. It is engineered to enable us to perform critical tasks using just a single command. For instance, deploying and running the `yggdrasil` binary, along with uploading any requisite assets directly to the designated robot can be achieved simply through the command `sindri run <robot_id>`. It is also equipped with a network scanning capability (`sindri scan`), to identify online robots within an IPv4 address range and many other useful utilities.

One of the main additions to `sindri` this year was the local feature, allowing us to simulate the framework on local hardware instead of having to use a real NAO. As of now, this does not make use of the changes in `nidhogg` and is mostly used for

testing changes in the vision algorithms, but we plan to introduce more feedback by integrating a 3D simulation of the real-world environment.

As this is the first year we started playing in the SPL using our own framework, we additionally wrote functionality for many-robot deployment and network configuration through the `sindri showtime` command.

3.4 Visualization

To streamline development and make debugging issues easier, we heavily rely on Rerun [8] for real-time monitoring of the robot's vision and control.

Rerun is a versatile tool for visualizing multi-modal data over time, making it perfect for debugging our robotics code. For example, it helped us find significant issues with our ball proposal algorithm. Using its SDK, we log real-time data streams - including images, line segments, bounding boxes, and joint transforms — and send them over TCP to a Rerun viewer on the developer's machine. Additionally, recordings can be written directly to an external flash drive, which we utilize during matches. We review these recordings post-match for in-depth analysis, aiding in performance improvements for subsequent games. These match recordings are also used to collect more data for our datasets using the Dataframe API[1].

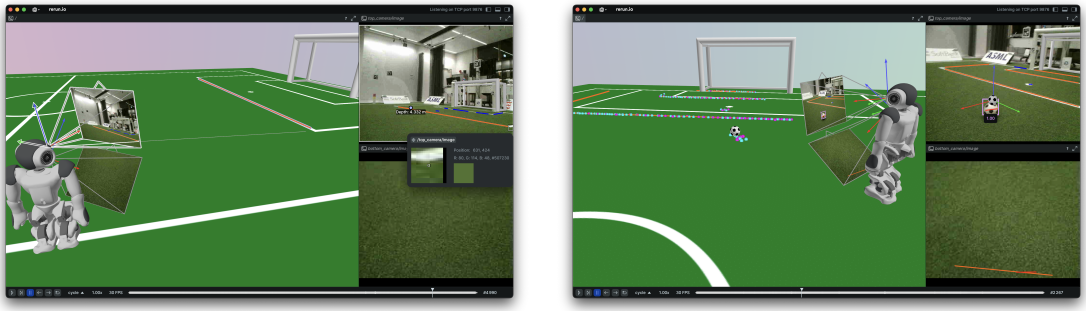


Figure 4: Rerun visualization of the yggdrasil framework

We implemented additional tooling surrounding Rerun to improve our workflow. For example, we implemented a control panel to selectively enable and disable data streams to save bandwidth where possible.

[1] `https://rerun.io/docs/howto/get-data-out`

3.5 Networking

In the last year, we developed the framework for our robot-to-robot communication during a match. As the networking environment is not part of our responsibility, we did not need to consider issues such as latency and packet drop. Instead, we assume that such conditions are perfect (negligible latency, no packet drops), which allows for a simplified design. One of the core design goals was to provide an ergonomic API that abstracts away the constraints laid out by the SPL rules. Not only does this aid modularity by keeping the networking logic close together, but it allows multiple producers and consumers that are unaware of each other to efficiently work together.

The constraints placed upon our networking can be summarized as three separate, but intertwined aspects:

1. There is a team-wide **message budget** over the entire game, amounting to 1200 packets or 1 per second on average. Messages only count towards this limit during the ready, set, and playing states.
2. There is a maximum **packet size** of 128 bytes, but messages are allowed to be shorter (which means we do not need to pad out messages and encode the payload length inside the message).
3. All messages must be passed via **UDP broadcast** over a single port assigned to the team. This allows the game controller to keep track of the message budget, which they in turn expose in the data packets it sends to the robots during gameplay.

These limitations require us to cleverly utilize the bandwidth available to us to share the most important information to improve our gameplay. There is definitely more information we would like to communicate than is possible, so we must prioritize some messages over others. Because the budget is expressed both in size and number, we could potentially waste a significant amount of our potential bandwidth if we do not fully utilize this space. At the same time, information can quickly become outdated, which means that keeping it around in a buffer waiting to be filled up might end up wasting it after all.

3.5.1 Deadlines

To satisfy these requirements, our networking stack consists of an inbound and outbound buffer that packs/unpacks multiple messages into/from a single packet. Crucially, the outbound buffer not only keeps track of the content of the messages, but is also supplied with a *deadline* for when the message is to be sent out. This deadline can be expressed through the API relative to the current time or at an

absolute moment in time. This way, producers can flag messages as needed to be sent out within a specific time frame in a single API call, and the networking stack uses this information to make the final decision on when the packet is actually sent out. Additionally, we expose a method for updating an existing message inside the buffer if it is still around, so messages with longer deadlines do not necessarily become stale. This is useful for information that is of lower priority but still benefits from being as up-to-date as possible by the time it is sent out. When the buffer is full enough, it packs as much of these messages in a single packet and then sends it out. The messages are packed ordered by their deadline, although smaller messages take precedence over larger ones if the larger one no longer fits in the remaining space.

We recognize, however, that a significant portion of our framework does not need such specific control over when a message is sent out. Instead, the general requirement is to send out as much as possible so that we have a near-zero budget remaining at the end of the game. This is why messages, unless specified otherwise, are given an automatic deadline that is calculated to be a sustainable rate at which to send them out. This deadline is not necessarily constant, but can vary over the game as long as it ultimately ends at the same number of packets sent over the entire game.

Going into more detail about when packets are exactly sent out, there are four variables relevant to this decision:

1. The **late threshold**, which controls the minimal interval between packets *even if any of the pending messages have reached their deadline.*
2. The **early threshold**, which controls the minimal interval between packets when none of the messages have yet reached their deadline.
3. The **automatic deadline**, which controls the deadline messages default to when not specified otherwise, and should typically be between the early and late thresholds.
4. The allowed **dead space**, which controls how much we are allowed to underfill the packet if we send it out early.

This design allows for global control of the rate at which messages are sent out, but has the tolerance to adaptively send messages slightly faster or slower. The late deadline is necessary to prevent messages that set their own deadline to use up to much of the bandwidth too early into the game The early deadline is not as necessary to keep the networking functional, but does ensure packets don't linger in the buffer when we are underutilizing our bandwidth. This incentivizes producers of low-priority messages to set a higher deadline so it gets sent out at a time when

higher-priority messages don't need to.

3.5.2 Encoding

Because we pack multiple messages into a single packet, we need to reconstruct the message boundaries, as well as the types. As there is a *maximal* rather than a *required* packet size, we can reconstruct the total number of messages by scanning the lengths of the individual ones. In order to be more space-efficient, we require that the length of the message can be determined by the decoder itself without an external length tag. This is done because most of the messages that we want to communicate do not contain any variably sized fields. As we cannot know what messages the buffer contains beforehand, we are need to prefix an enum tag that identifies what type of message this is. Because we have a centralized project, we use Rust enums and procedural macros to implement the encoding and decoding of these tags and the messages themselves. This ensures a unique tag for every possible message format without any conflicts at compile-time, at the cost of requiring every module that wishes to add a new message format to add it to one central enum definition rather than to their own separate modules. Also, as we control the deployment as well, we do not need any protocol negotiation or versioning to be implemented, as we ensure all robots run the same code.

Although we have a significant bandwidth constraint, the networking layer does not implement any transparent compression of messages when they are packed into a packet. The reason for this is two-fold, first of all we expect the sent over data to be of reasonably high entropy, and we require the message producers to be conscious of the amount of space their message takes up when encoded. Secondly, for data that could benefit from being compressed, we expect that in these cases a compression scheme specialized towards that specific message to be more space-efficient than a compression scheme generalized over all message types, even if that means compression will have to be done on the message-level rather than on the packet-level. To do this, producers can swap out the automatically derived encoding function for their custom implementation, keeping the framework modular.

3.6 Kinematics

By leveraging Rust's type safety, we have refactored our handling of spatial coordinates to keep track of the coordinate frame in which geometric objects are defined at compile-time. Not only does this enchance code readability and clear up ambiguity in naming, but it also exposes this semantic information to the compiler in a way that it can be introspected and transformed syntactically to provide powerful

abstractions. At the core, it tracks what space an object belongs to by wrapping it together with a zero-sized `PhantomData<S>` field. The generic parameter type `S` only serves as a marker and is never actually constructed, but only used to interact with the type system. This wrapper is expressed as `InSpace<T, S>` in our framework, with convenience type aliases for commonly used types such as `Vector3<S>` or `Point2<S>`. To provide further validation, we mark the space type with some trait bounds used as marker traits. These trait bounds are `Space`, which marks the type as a space marker, and `SpaceOver<T>`, which marks the type as expressible in that space. We use this latter bound to ensure that an n-D type cannot be marked with an m-D space (where $n \neq m$) at the cost of slightly longer trait bounds in function definitions generic over spaces. Additionally, we have another wrapper type `BetweenSpaces<T, S1, S2>` that marks a transform as mapping between these two spaces (with type aliases such as `Isometry3<S1, S2>`).

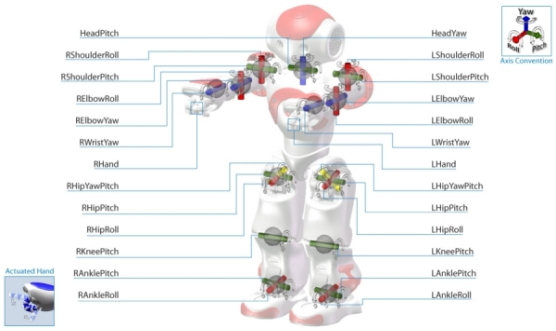


Figure 5: Joints and actuators of the NAO robot

The benefits of this approach are mostly reaped by code that interfaces with our kinematics implementation. Previously, our forward kinematics struct contained an isometry from every joint to a single point on the robot referred to as "robot space". These transforms were pre-processed when constructing the kinematics chain from an array of joint positions. Because all isometries transformed into the same space, transforming to and from every other space was achieved through a relatively simple chain of a transformation followed by an inverse transformation. However, this design stored every isometry in a separate field, without a way for the compiler to introspect what field maps from which space into which other space. This prevented us from using Rust's generic features to avoid code repetition. Although we could have stored all the isometries into a single homogeneous array indexed by an enum, which would have had a negligible cost at run-time and compile-time, we opted for the more expressive system described above. With this design, every field has a different type (even though the wrappers are ultimately transparent and do not affect the memory layout, which allows for the casting of

references to and from the inner type). Adding an elementary trait implementation for accessing every concrete space then allows us to build more generic abstractions on top. Because of Rust's limitations on trait bounds, most notably the lack of negative trait bounds and specialization, creating a generic function to transform every possible type to and from every possible space necessitated a couple of convoluted workarounds.

Ultimately, we settled on using procedural macros to generate the implementation of a `Transform` trait, which is implemented manually for transformation primitives such as matrices and isometries, but automatically for bundles of transformations. It does this by inferring what spaces a field maps between based on the last two generic parameters of its type. As the procedural macro lacks access to a lot of context, there is no way to access this type of information directly. Unfortunately, as a consequence, we cannot create nested bundles or type names without the use of additional attribute annotations. Using the available information, it generates a concrete implementation of a transformation to and from every space, generic over the inner geometric object. To accomplish this, it builds a graph and uses it to find the shortest path of transformations between any two spaces. The only constraint is that the input and output type must match at each point along the way, as, otherwise, there could be multiple paths which would then require a concrete implementation for every inner type as well. This results in a lot of code generation, but the simplicity of the generated code does not require much additional work on the part of the compiler. Ultimately, our setup provides a sufficient basis for building complex abstractions without the need for excessive additional code duplication. Practically, it allows us to transform individual points or vectors along a chain. However, by passing a type that implements `Transform` itself through the chain, we end up with a direct transform.

4 Sensing

Apart from the cameras, the NAO has several sensors. In our framework, the output of the inertial measurement unit (IMU) and the microphone are very important. In this chapter, we explain how we use the IMU in our orientation filter, and how we use the audio input to do whistle detection.

4.1 Orientation filter

To keep track of its orientation, the NAO is equipped with an inertial measurement unit (IMU) with a gyroscope and accelerometer. The readings from these sensors are provided by LoLA, alongside an *angles* property which contains the estimated

roll and *pitch* of the robot.

This leaves us without the very important *yaw* of the robot. Moreover, the *roll* and *pitch* values are noisy and lag behind a few frames at best. Additionally, LoLA alternates updating the values for the gyroscope and accelerometer respectively [9]. This means that instead of the usual 12 millisecond delay between measurements, there is a 24 millisecond delay.

To obtain a consistent and trustworthy estimate of the robot's orientation, we perform sensor fusion of the gyroscope and accelerometer measurements using the Versatile Quaternion-based Filter (VQF) [10]. Unlike traditional techniques like Madgwick [11] or VAC [12], VQF represents the current orientation estimate as a concatenation of multiple quaternions. This enables it to estimate separate biases and uncertainties for each component, making it more robust over time.

The algorithm begins with *strapdown integration*, which uses the angular velocity measurements from the gyroscope to calculate the change in orientation over time. By itself, this technique will lead to drift over time due to small biases in the gyroscope measurements. To address this, the VQF estimates the gyroscope bias by comparing the estimated gravity vector derived from accelerometer measurements with the one inferred from the integrated orientation. Any difference is assumed to be caused by gyroscope bias, which the algorithm then corrects for.

The authors provide a Python and C++ implementation[2] of the VQF. However, since no Rust implementation is available, we implement a pure Rust version and make it publicly available here: `https://github.com/oxkitsune/vqf`.

4.2 Whistle detection

Another key area that has seen [13] significant improvement during the past year was automated whistle detection. Whistle detection is the task of recognizing when the referee blows their whistle, which is important to correctly judge the current game situation. Failing this task can have counterproductive consequences, hence it is paramount to deploy an accurate whistle detection method.

The development consisted of two steps. Firstly, programming a function for converting a raw waveform (i.e., an energy-time graph) recording from the NAO microphone to a spectrogram (i.e., an energy-frequency graph). The reason is that for this task, spectrograms are a highly useful audio representation, as frequency information is necessary for distinguishing a whistle from a non-whistle sound with identical energy levels. Secondly, developing a whistle detection model that classifies a spectrogram with a binary label, where 0 denotes that no whistle was

[2] `https://github.com/dlaidig/vqf`

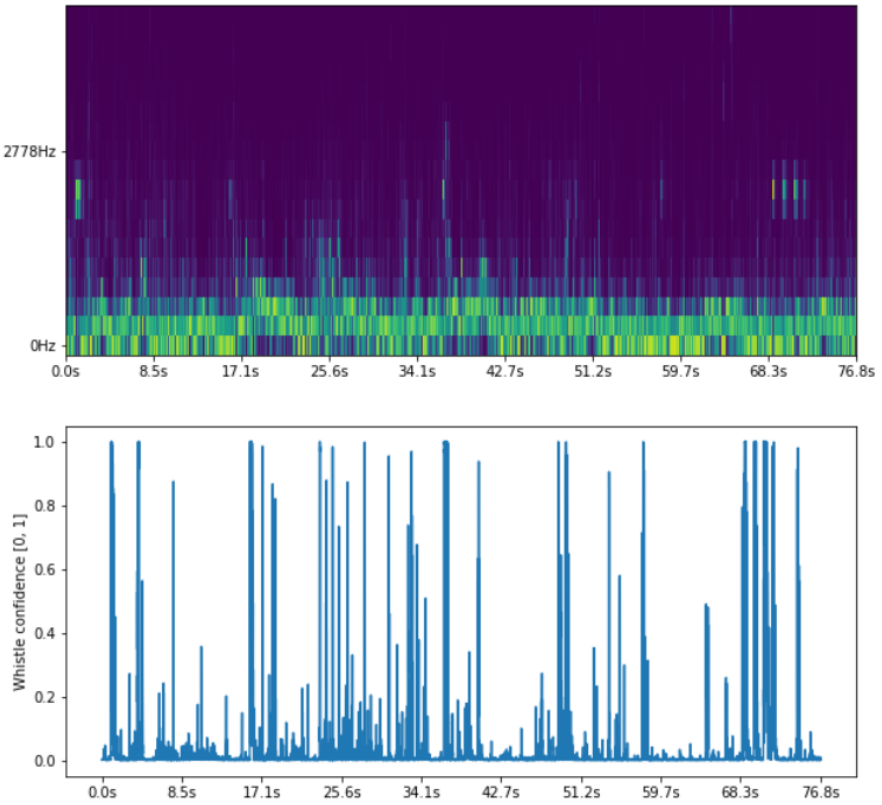


Figure 6: A spectrogram constructed from an audio segment of a part of a RoboCup match with corresponding output of the whistle detection model.

detected and 1 that a whistle was detected. See Figure 6 for an example of a spectrogram along with the output of the final detection model.

The first step, converting a raw waveform to a spectrogram, is typically done using the Fourier transform (FT) [14]. However, a spectrogram does not account for frequencies changing over time, even though our goal is to classify at which moments in time a whistle sound is present and at which ones not for an audio sequence. Hence, we need an adapted version of the FT: the short time Fourier transform (STFT) [15], which returns an array of spectrograms by repeatedly computing the FT for a sliding window over time. Because at the time of development no suitable Rust implementation of the STFT was available, we implemented it ourselves using the FT provided by the RustFFT[3] crate.

For the second step, developing the whistle detection method, we compared two different methods:

[3] `https://github.com/ejmahler/RustFFT`

- An *algorithmic* approach described by a member of the HULKs team[4], which serves as a baseline. As its precise inner workings have been expanded upon before by the HULKs team, we shall only give a brief overview. Principally, their method takes an audio spectrogram and searches the high frequencies between 2kHz and 4kHz for high energy zones using a handcrafted algorithm. If the mean energy in such a zone exceeds a predefined threshold, the method classifies a whistle. The reason for only searching high frequency bandwidths is that whistles typically are high pitched, which means searching low bandwidths is futile and could even be harmful for classification. To improve accuracy, their method also localizes the place of origin of the sound using multiple NAO robots, but that aspect was not considered in our implementation.
- A *machine learning* approach leveraging a feedforward neural network (FNN). Analog to the previous method, this method only considers the high 2kHz to 4kHz bandwidth. However, instead of using a handcrafted detection mechanism, we train a FNN to classify whistles. The previous iteration of our whistle detection model trained on the entire frequency spectrum, but we found it was overfitting on lower frequencies to such a degree it became nearly unusable. For training, we used the dataset provided by [16].

  Another key factor we changed with respect to our previous iteration is that we use our Rust implementation of the SFTF in both the framework and to preprocess the training data. Previously, we used the Pytorch implementation in the training code and modeled our Rust implementation after the Pytorch version. However, the implementations were not precisely equal, which adversely affected training.

Model	Accuracy
Algorithmic baseline	0.93
FNN	**0.99**

Table 1: Whistle detection accuracy results

After evaluating the models on the test dataset, we find that the FNN approach achieves a near perfect accuracy and substantially outperforms the algorithmic baseline (see Table 1). Lastly, we performed a practical test by simulating an exceedingly noisy match environment and sporadically blowing a whistle during the course of roughly 15 minutes. We found that the model was able to correctly identify all whistles.

[4] `https://github.com/ykonda/Masterthesis`

5 Vision

A large part of the information about its surroundings the robot receives is through its two cameras. Last year, we worked on various projects surrounding vision to extract and use the camera information. In this chapter we will explain more about projects on projection, calibration, line detection, field boundary detection and object detection we did.

5.1 Projection

In order to build a 3D understanding of world using the monocular vision of the NAO, we need to go from coordinates in 2D on the image plane to coordinates in the 3D world.

The NAO has two cameras, one of which faces towards the ground (see Figure 7). This renders it impossible to use stereo vision to estimate the depth of a 2D position in the image. Instead, we use an intersection with a known plane: The ground plane which is always at $z = 0$. This lets us project all pixels we have to a position on the ground. We found the resulting positions to be reasonably good.

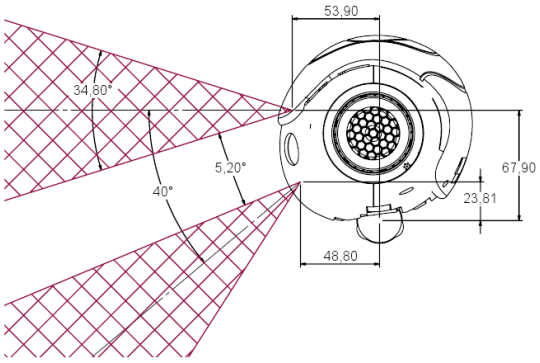


Figure 7: The positions of the two cameras in the NAO's head [17].

To project a point to the ground, we first normalize the pixel coordinates using the intrinsic matrix K:

$$\begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix} = K^{-1} \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

This gives us the ray direction in the camera coordinate system. We then find the

intersection of this ray with the ground plane given the extrinsic matrix, by first transforming the ray to world space:

$$d_{\text{world}} = R \cdot \begin{bmatrix} x_c \\ y_c \\ 1 \end{bmatrix}$$

In this equation Z_c is the depth value of the ray

5.2 Camera calibration

Accurate projection from the camera's view onto the ground plane requires both intrinsic and extrinsic camera parameters.

5.2.1 Intrinsics

The intrinsic parameters describe the internal properties of the camera, and how it maps 3D world points onto the 2D image plane. The intrinsic matrix in the pinhole camera model consists of:

- Focal lengths (f_x, f_y), which represent the scaling factors for the x- and y-axis of the image.
- Principal point (c_x, c_y), the coordinates of the image center (typically the optical center) in the image plane.
- Skew (s), represents the non-orthogonality between the x- and y-pixel axes. For most modern cameras this is close to 0.

We calibrate the focal lengths and the optical center, using the technique described in [18], which uses images of a known checkerboard pattern to estimate the intrinsic parameters of the camera. The skew parameter is ignored, as we found it does not noticeably impact the projection.

5.2.2 Extrinsics

Extrinsic parameters define the position and orientation of the camera's coordinate frame relative to another coordinate frame. These parameters form a rigid-body transformation, represented by a 3×3 rotation matrix R and a 3×1 translation vector T.

To transform points from the camera's coordinate space to the world space, the extrinsic matrix is updated continuously based on the robot's kinematics and ori-

entation (subsection 4.1). Since the robot's kinematic chain is defined only up to the neck, points in the camera's coordinate space are first transformed to the neck's coordinate frame. Although this transformation is generally constant, slight variations between individual robots make calibration necessary. Specifically, we calibrate the *roll*, *pitch* and *yaw* values of the extrinsic rotation matrix. The translation is assumed to be constant, small physical shifts in the camera's position over time are considered negligible compared to errors introduced by the IMU's slow update rate (subsection 4.1).

This calibration is done after the intrinsics are calibrated. We first place the NAO in the center of the field, facing towards one of the goals to ensure a controlled and repeatable environment. Then we use iteratively adjust the *roll*, *pitch* and *yaw* values until the projection lines up with known real world points.

5.3 Color calibration

Most of our crucial object detectors, such as ball or line detection subsection 5.5 assume correctly classified colors in order to work well. As our current color detection system works using thresholding, it is important that the color samples we get are therefore similar between robots.

However, varying lighting conditions significantly affect the perceived values of green, black, and white, leading to frequent misclassifications in scan lines and thereby compromising performance. These challenges arise not only when transitioning to different playing fields, but also within the same field as the position of the sun changes. In Figure 8, we can appreciate how different our understanding of colors can be.

Figure 8: Three scenarios for color calibration under varying lighting conditions. From left to right: (1) Homogeneous lighting with consistent color values, (2) High variance in color values due to uneven lighting, and (3) A challenging case where colors (green, black, and white) have similar pixel values, making detection difficult.

In order to mitigate this problem, we introduce a color calibration function designed to adjust thresholds based on changing lighting conditions. This function is intended to be run whenever lighting changes occur, ideally before each match, to ensure the use of accurate thresholds.

The calibration process begins by capturing pictures from different positions and perspectives of the field to be as generalizable as possible. The images should include the white and black cardboard sheets, as illustrated in Figure 8. Markers provide large and reliable reference areas for known colors.

After that, we leverage an interface to manually select quadrangles in the images that correspond to green, black, or white areas. For black and white, the quadrangles are selected from the cardboards, while for green, a representative area within the field is chosen.

Once these regions are defined, the function converts the images to the YHS color space and generates histograms showing the distributions of the YHS values for each selected color region, as shown in Figure 9.

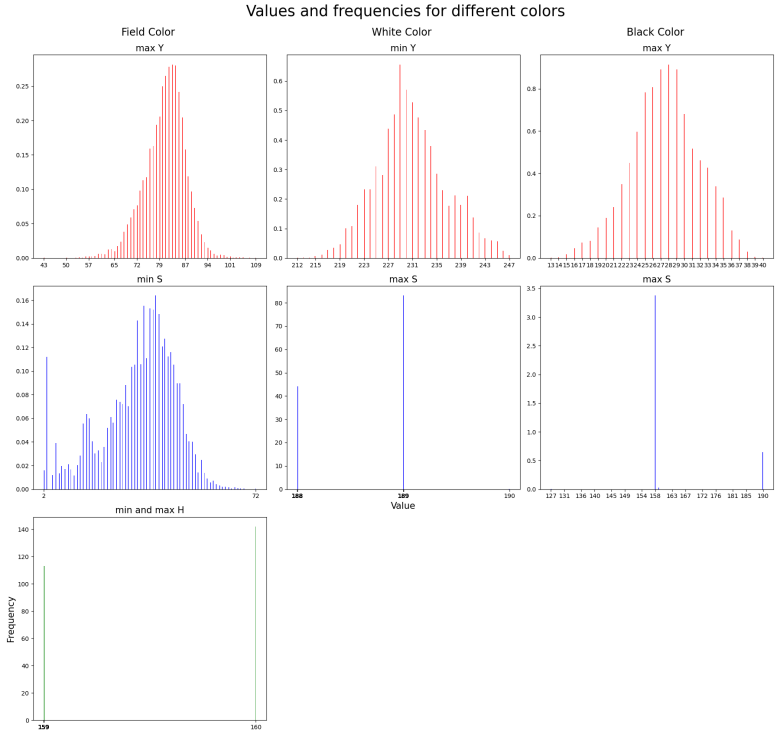


Figure 9: Example of the histogram analysis for choosing the YHS thresholds. Each plot gives the distribution/bounds for a different threshold value.

Visual analysis of these histograms is preferred over automatic calculations using statistical measures such as mean and standard deviation. We have this preference because the distributions are highly asymmetric and prone to outliers. In addition, experience-based insights can guide manual adjustments to enhance performance beyond what purely statistical methods can achieve.

5.4 Scan lines

Scan lines serve as an optimization step to build a representation of the image in which we can quickly find points of interest such as balls and lines. The scan lines reduce the number of columns and rows that are looked at by subsampling them in a way that still preserves most of the salient features.

5.4.1 Scan grid

In order to define the sampling frequency of our scan lines, we take a similar approach to B-Human, creating a *scan grid* that uses the camera parameters and horizon point in order to determine the width of field lines at each point in the image. From there, we can calculate the intervals between horizontal lines we need to sample in order to be able to theoretically detect all the lines in the image. For vertical scan lines, we use a static interval.

5.4.2 Scan line regions

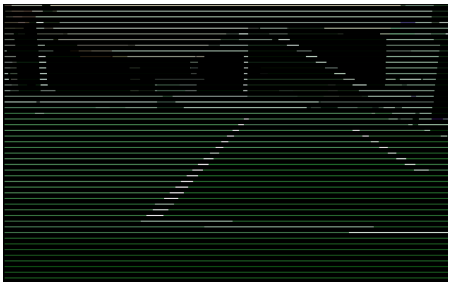
As we loop over the samples provided by the scan grid, we split each scan line into *scan line regions*. A scan line region represents a part of a scan line in which adjacent samples that have the same color.

For each row/column in the horizontal/vertical case respectively, we compare the luminance with the average luminance of all pixels in the line segment before (if there is one). As long as the difference is below a certain threshold, we append the pixels to the line segment. However, if the difference is above that threshold, we split the previous line segment, where the luminance difference between adjacent pixels in the line segment is the largest.

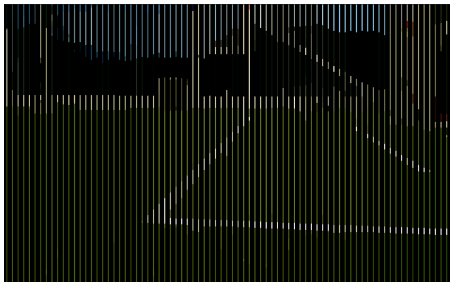
5.4.3 Color classification

Once we have our scan line regions, approximate colors are first converted to the YHS2 color space [19]. Finally, they are thresholded using a set of predetermined parameters. If two scan line regions are classified to be the same color, they are merged.

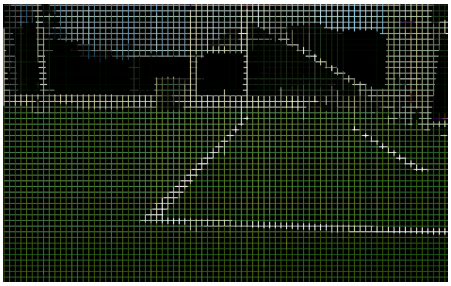
An example of the detected scan lines can be seen in Figure 10.



(a) The horizontal scan lines



(b) The vertical scan lines



(c) The horizontal and vertical scan lines combined

(d) The original image

Figure 10: Examples of horizontal and vertical scan lines

5.5 Line detection

For line detection, we consider the center of each white scan region as a sample called a line spot. We take the set of vertical and horizontal line spots and project them to the field. We then perform several iterations of RANSAC to fit lines between these points, which will serve as field line candidates. If adjacent line spots within a fitted line are too far apart, we split the line candidate into two.

A common occurrence is that multiple candidates lie on the same actual field line, so once we have all the line candidates, we perform an additional candidate merging step similar to the approach used by B-Human [19]. That is, if the candidates are part of the same line, the following assumptions should hold:

1. The points on the line segment connecting two candidates should be colored white.
2. Points one line thickness away in the directions along the normal of the line segment (i.e., points slightly outside of the candidate line) should be colored green.

Using these assumptions, the merging step works by taking all candidates that are close in angle and drawing a set of samples in between them. We test the ratio at

which samples in a connecting line segment are brighter and less saturated than the samples plus a small offset along the normal. If this ratio is high enough, we merge the lines.

Finally, we perform an extra rejection step for lines that do not fit our quality criteria. The current criteria are:

- A line segment must contain a certain minimum of inlier line spots.
- A line segment must be longer than a minimum length.
- A line segment must be shorter than a maximum length.

Figure 11: An example of lines and inlier line spots found by the line detector. A merge test is also performed between the outer lines of the goal area and the penalty area.

5.6 Field boundary detection

Some vision modules should only consider parts of the image containing the soccer field, since detecting lines or balls outside of the field would waste computational resources.

To address this, we implemented a neural network-based field segmentation model, based on the work of [20]. Our approach follows the original method by predicting the y-position of the field boundary at 40 evenly spaced points across the image (See magenta points in Figure 12). By fitting a line through these predicted points, we create a boundary where all pixels below the line are classified as part of the field.

The architecture described in [20] follows a two-stage design based on [21]. The backbone consists of 3-4 inception V3 blocks, which handle feature extraction.

These features then pass through a single-layer convolutional bottleneck that outputs the predicted boundary point positions.

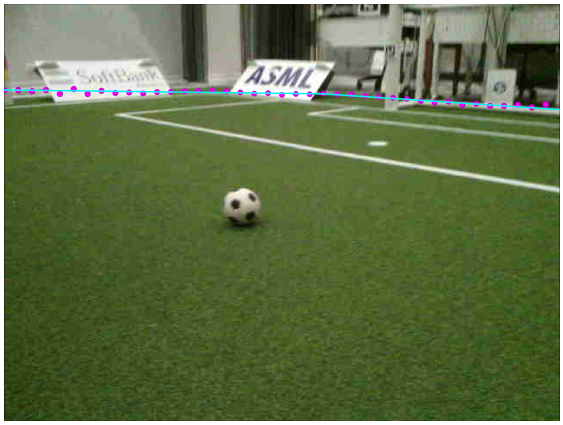


Figure 12: Predicted field boundary points (magenta) and the final boundary fit (aqua).

Our architecture is very similar, but we switch out the convolutional layers with depth wise separable convolutions and use the Sigmoid-weighted Linear Unit (SiLU) [22] as non-linearity instead of the traditional ReLU. The original EfficientNet architecture shows that this activation results in better performance [23] as it does not immediately discard negative inputs which makes training more stable.

We are working on incorporating an EfficientNetV2-inspired backbone [24]. Specifically, we use the "Fused MBConv" block, which combines a depthwise separable convolution [25] with an SE-Block [26]. This results in a fast parameter efficient operation.

5.7 Ball detection

In order to quickly find a ball within an image, we split up the detection process in two steps: a proposal and classification step. In the proposal step, patches that potentially contain a ball are identified, and in the classification step for each patch it is determined if it actually contains a ball. Both steps have been subject to scrutiny and improvement over the course of the past year.

5.7.1 Proposals

In the first step, ball proposals are generated. Because of the limitations that the NAO hardware poses, the major point of focus when developing the model was time efficiency. With that in mind, we focused on: (a) designing an efficient system

with low runtime, (b) minimizing the number of proposals to reduce the runtime of the classifier step, all the while not losing out on accuracy.

To reduce runtime, proposals are induced not from the camera image, but from the generated scan lines, which results in a significantly smaller search space. To find potential ball locations from scan lines, we make the following assumptions:

(1) The ball is white and black. This assumption is made in order to separate potential ball candidates from field patches.

(2) The ball has green grass on either the left or right side. This assumption is made in order to separate ball candidates from field lines.

(3) The ball is round. This assumption is made in order to filter out candidates that do not have enough vertical extension such as field lines.

(4) Beneath the ball, there is a patch of green grass. This assumption is made in order to filter out candidates made from patches of other robots in the image.

We search through the scan lines, and flag each area where the assumptions hold as a potential ball location.

More specifically, the procedure is as follows:

1. Begin by iterating through all horizontal scan lines and checking if assumption (1) and (2) are held. This check is trivial to perform, as each scan line has a color attribute that denotes if the line is black/white, green or another color. If the check is passed, we continue to the next step.

2. Then, identify no more than two points on the scan line of interest that might be the center of the ball. The next steps of the procedure are separately conducted for these points.

3. Next, assumption (3) is checked given a potential ball center in image space. Because the camera matrix and NAO joint positions and orientations are known, the visual radius of the ball can be estimated using solely the ball center. With that, we compute what percentage of the circle around the ball center with the computed radius is white/black by iterating over all scan lines. If that value exceeds a certain threshold (determined by a hyperparameter), there is a circular, ball-colored patch around the potential ball center. That means assumption (3) is held and the next step in the procedure should be performed.

4. Subsequently, given a potential ball center and radius, assumption (4) is validated by checking if the percentage of green color in the patch below the

ball exceeds a certain threshold (determined by a hyperparameter). If so, the ultimate assumption is held, and the proposal is stored.

5. Finally, after all proposals have been gathered, non-maximum suppression [27] is applied to the proposals to remove overlapping instances.

Upon visually testing the model (e.g., see Figure 13), we notice that the proposals are generally sensible and expected given our assumptions, while still employing a time-wise efficient model. Furthermore, we note that the number of proposals is kept low, also achieving the goal of not overloading the ball classifier.

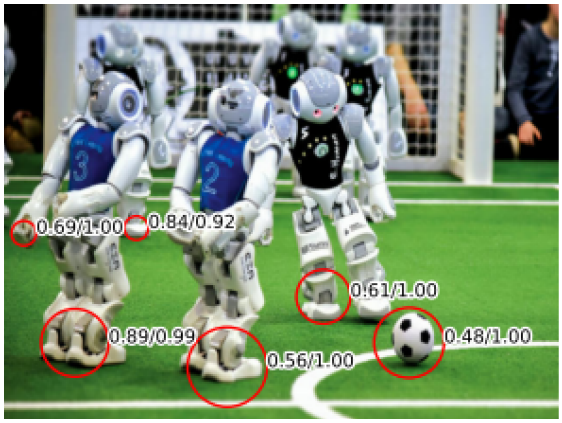


Figure 13: Ball proposals on an example image. Each proposal has two numerical values, that respectively denote to what degree (on a scale from 0 to 1) assumption (3) and (4) are held.

5.7.2 Classification

After ball proposals are determined by the algorithm described in subsubsection 5.7.1, a ball classification model is used to determine whether the proposals are actually balls. In the past year, this model was developed and improved.

Dataset

To develop the classifier, first a dataset of patches was gathered based on full-size images with annotations. To create the dataset, the NAO lower camera ball detection dataset of B-Human, together with the SPL Object Detection dataset V2 created by RoboEireann [28] was used as a basis. The patch dataset consists of 32x32 grayscale patches, which are sampled from full-sized images collected in different matches. Positive patches are created by taking (part of) a ball annotation and resizing it to 32x32 using nearest interpolation. Negative patches are extracted by using both random sampling from the areas without ball annotations, and

specifically taking samples from areas with high contrast. This was done because most false positives occured on patches with a lot of black and white. After evaluating performance of the developed model, samples from specific areas of the field, for example the penalty mark, were added, since it was observed that the model was not able to classify these samples correctly. Examples of the patches used for training the classifier, before converting them to grayscale and resizing, are shown in Figure 14.

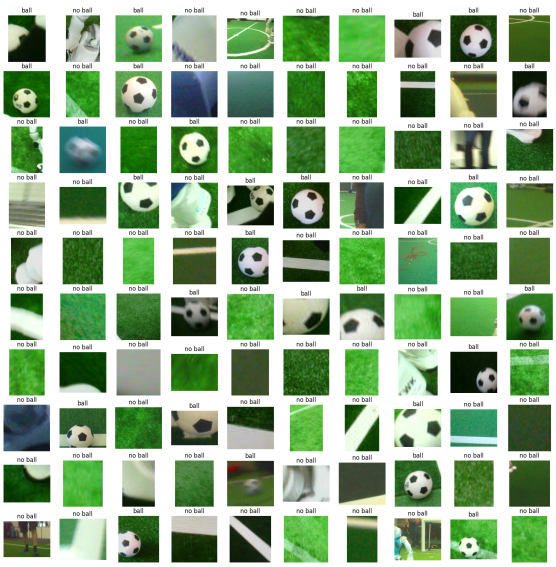


Figure 14: Examples of the patches used for training the ball classifier. The patches are converted to grayscale and resized.

Classification model

To classify the ball proposals, a ball classification model was developed. During development, the goal was to make the model as light as possible, so that we are able to classify as much proposals as possible in the cycle time available, while maintaining almost perfect performance. The ball classifier that was developed uses 939 parameters and has an inference time of around 350μs. The model has an accuracy, recall, precision and F1-score of above 99%.

The model architecture is shown in Figure 15. It makes heavy use of the Depthwise Separable Convolution [25] for inference speed. Additionally it uses a Squeeze-Excite (SE) block [26] between the two spatial reduction layers in order to learn the dependencies between channels. This allows the model to focus more on the relevant channels, and achieves a computationally efficient channel attention mechanism. This block is important for the model's ability to generalize, achieving

$0.3 \pm 0.2\%$ higher F1-score compared to models without the SE

Figure 15: TurboBallClassifier Network Architecture

5.8 Robot detection

When playing matches, it is important for robots to be able to detect other robots, both teammates and opponents. If a robot is able to detect other robots, this information can be used for multiple things. First of all, it can be used for path planning and object avoidance. Furthermore, the information could be used to help localize other robots. Lastly, accurate robot detections can be used to help filter out irrelevant ball proposals. Since robots have black and white components, similar to the ball, it can be hard to distinguish ball patches from robot patches. It was observed that in the ball proposal phase, there were a lot of proposals on other robots. Because of this, many proposals had to be processed, resulting in a longer cycle time. Therefore, a robot detection model was developed, with as the main goal to filter out all proposals that were on a detected robot.

Dataset

To train the robot detection model, first a dataset was collected. The initial dataset was based on the object detection dataset from RoboEireann [28]. However, the dataset was quite small, and a substantial percentage of the images was of low quality. Therefore, we decided to record and annotate our own dataset. This dataset consists of multiple videos, recorded in the Intelligent Robotics Lab and at the RoboCup 2024. The videos were manually annotated by members of the team, using an annotation pipeline based on CVAT[5]. In total, a dataset consisting of 26,295 samples was created. Together with a future version of our detection model, this dataset will be made public.

Detection model

The detection model developed is a single-shot detector (SSD) with a custom backbone. This means that first, feature extraction is done using the backbone. After this, bounding boxes are fitted on the features by the SSD, the boxes are refined, and subsequently classified. Since the images from the robot are in YUV color space, and converting them takes up valuable time, the model was specifically designed to be able to work with YUV images.

[5] `https://www.cvat.ai/`

Figure 16: Examples of robot detections in YUV space

The custom backbone consists of three convolution blocks with a kernel size of 3, all followed by a 2D maxpool operation with a 2x2 kernel. The convolution blocks consist of one convolution layer, followed by a ReLU activation function. The first convolution layer maps the number of channels from 3 to 32, the second has 32 layers as input and output, and the last doubles the number of channels to 64. After this, the SSD runs a network for classification and for regression on the bounding boxes.

The inference time of the model is $19ms \pm 8$, which is fast enough to run on the robot in real-time. The model uses 49,412 parameters. The mAP of the model on the test set is 31.1%, which was good enough in practice. Examples of robot detections are shown in Figure 16.

While the developed robot detection model provides a basis for robot detection, several improvements are still possible. First of all, it was observed that the model performance differs based on the color of the robot jersey. When the robot jersey is darker, the model is unable to accurately classify the robot. Furthermore, when the light is very bright, the YUV image gets noisy, leading to worse predictions. Lastly, the certainty scores of the detections of the model are quite low, making it hard to filter which detections to use. At the moment, a project to improve the robot detection model using state-of-the-art techniques is in progress [29] [30].

6 Motion

6.1 Walking engine

This year we continued work on the walking engine. The original version described in the 2023 tech report [31], which was based on the 2014 walk described in [32], has been kept largely the same this year.

However, we are in the process of significantly improving the performance and

consistency of the gait generator. A big part of the 2025 vision for the Dutch Nao Team walking engine is enhancing it with learning based methods which will be discussed in more detail in subsection 6.2 [33].

This started with a refactor of the existing gait generator in order to provide Python bindings. During this refactor we also implemented some quality of life features to make it easier to implement behavior using the walking engine.

6.2 3D reinforcement learning

This year we focused on building out the required infrastructure for motion reinforcement learning (RL). While high-level behavioral policies, as described in subsection 7.3, can be trained in simplified environments where the robot's dynamics are less important, low-level policies such as motion control require a significantly more complex environment. For such tasks, the accuracy of the robot's dynamics in simulation is important. Even small differences in the robot's dynamics, often referred to as "the reality gap" ([34], [35] and [36]), can cause a significant drop in performance when the policy is used on the physical robot.

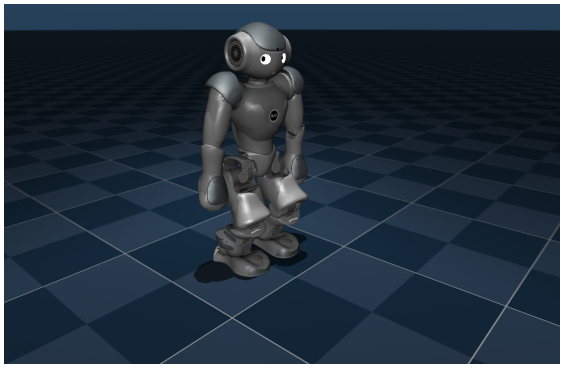


Figure 17: The NAO model in MuJoCo

To address this, we focused on reducing this gap by using the MuJoCo [37] physics engine to model the NAO robot (see Figure 17). MuJoCo is a general purpose physics engine that provides us with all the tools we need to accurately simulate the NAO robot, such as a gyroscope, accelerometer and pressure sensors.

6.2.1 Gait Modulation

Due to the limited computational resources available, we plan to enhance existing systems using these learned policies. The first component we want to enhance is the walking engine, specifically the gait generator. Previous work [38] has shown

that enhancing an existing gait generator using a policy learned in simulation leads to a more generic and stable gait. [33] showed that this technique is generalizable to different robots, and we are currently working on bringing this to the NAO.

6.3 Keyframe Motion Engine

During competitive matches, it is crucial that our robots are able to execute motions and make decisions dynamically, to react to a changing environment. However, certain movements, such as a standup motion after a fall, require the NAO to execute motions with high precision to ensure safety and success. While the overall robot behavior can be dynamic and adaptive, these motions need to be executed with exact control over each joint's position and timing.

The keyframe motion engine is designed to execute such precise movements consistently and safely. By using keyframes which dictate a specific position and stiffness for each joint which the robot should reach at defined points in time, the robot is able to perform complex motions the same way every time.

6.3.1 Composition

Composing a motion involves defining submotions, movements, and conditions that guide the robot's actions. A motion is essentially a sequence of keyframes, each representing the desired joint positions at a given time. However, to introduce the reusability of specific movements and to introduce more complex behaviors during motions, these motions have been structured hierarchically.

Motion: A *motion* is the primary structure that encapsulates all the actions to be performed, for which an example can be seen in Figure 18. It gives an abstract representation of what the robot will do during it's execution and includes the overall settings for the motion. The settings alter the motion execution by:

- Changing the interpolation type: Linear, Ease-In, Ease-Out, Ease-In-Out, or a custom bezier curve.
- Changing the exit routine the robot will execute after a successful execution.

```toml
# standup_back.toml
global_interpolation_type = "Linear"
exit_routine = "Standing"
motion_order = [
    "standup/on_back/ready_to_kick",
    "standup/on_back/kick_up",
    "standup/on_back/halfsit_to_halfcrouch",
    "standup/halfcrouch_to_sit",
    "standup/extend_legs",
]
```

Figure 18: An example of a motion file, used in the keyframe motion engine.

Submotions: Within each motion, there are one or more *submotions*. A submotion represents a smaller, modular part of the motion. It represents a specific part of the larger motion which can be reused by other motions. Submotions are defined with an order of movements to be executed and more motion settings specific to this submotion, as can be seen in the example in Figure 19. These settings include:

- Joint stiffness
- Torso angle bounds which dictate what angles the robots torso should remain between to be considered stable.
- Exit wait time to limit how long the robot will wait after a submotion to see if it is stable.
- Fail routine which will be executed if the current submotion fails (for example, when the robot exceeds it's torso limits).
- Entry conditions that the robot will have to fulfill to be able to enter the submotion. Otherwise, it will execute the fail routine of the last submotion.

Movements: At the core of each submotion are the *Movements*. A movement specifies the exact target joint positions, along with the duration it should take to reach those positions. Movements are the smallest components in the motion structure and are executed in sequence. The movements also contain an optional setting, which is able to override the interpolation type set by it's parent motion.

6.3.2 Execution

The following flowchart shown in Figure 20 illustrates the steps involved in executing a robot's motion. It covers the initialization, safety checks, motion execution, and transitions between submotions, ensuring smooth and safe operation throughout the process. Absent from this flowchart are additional safety measures such as checking the chest angle bounds every cycle, which may trigger a failroutine.

```
{
    "joint_stifness": 0.9,
    "torso_angle_bounds": [
        {
            "variable": "AngleY",
            "min": -0.6,
            "max": 0.6
        }
    ],
    "fail_routine":"Abort",
    "exit_wait_time": 1.5,
    "block_pickup": true,
    "entry_conditions": [
        {
            "variable": "AngleY",
            "min": -0.6,
            "max": 0.6
        }
    ],
    "keyframes": [
        {
            "duration": 0.5,
            "target_position": {
                "head_yaw": 0.0,
                "head_pitch": 0.3839724354387525,
                "left_shoulder_pitch": 2.0943951023931953,
                "left_shoulder_roll": -0.08726646259971647,
                "left_elbow_yaw": -1.48352986419518,
                "left_elbow_roll": -0.33161255787892263,
                // more joint values...
            }
        }
    ]
}
```

Figure 19: An example of a submotion file with a single keyframe, used in the keyframe motion engine.

7 Behavior

Once all sensor inputs have been processed into meaningful information, the behavior system's role is to translate this information into actions that the robot can execute. Our current implementation employs a traditional, deterministic approach, where actions are assigned based on the robot's state and the game context. However, we are exploring more advanced methods, like reinforcement learning (RL), which could have more potential in the future to provide more dynamic and adaptable behaviors.

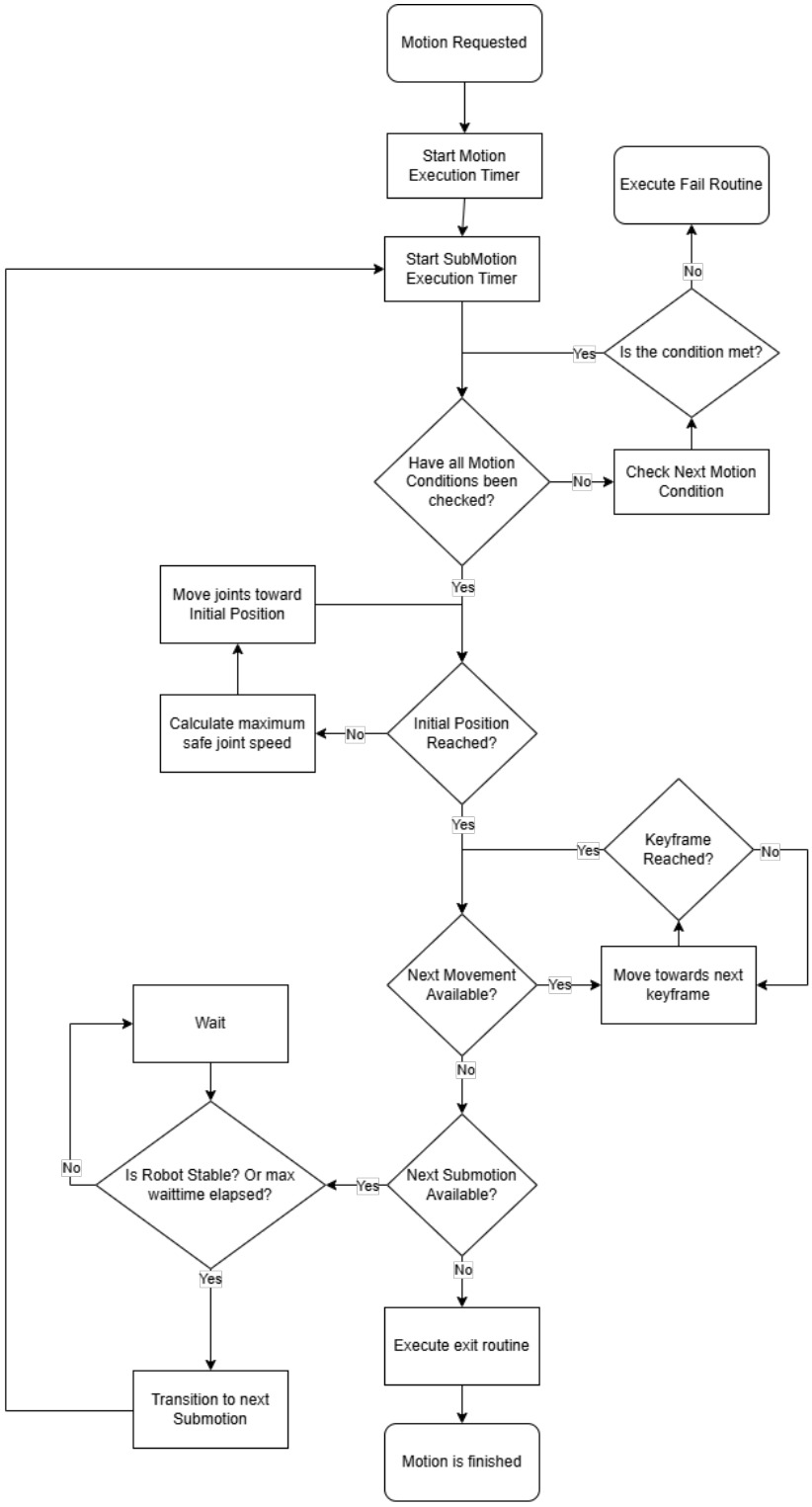


Figure 20: The flowchart for the execution of a motion, simplified.

7.1 Behavior Engine

The current implementation of the behavior engine prioritizes executing tasks determined by the PrimaryState, which, in play, is determined by the GameController's GameState. This ensures that the behavior always obeys the GameController. In the Playing state, the behavior engine employs a dynamic Role system. Each robot has a default role determined by its number and starting position on the field. As the game progresses, the robot's role can be changed, though, in its current implementation it is limited to becoming a Striker upon detecting a ball.

7.2 Behavior Simulation

With a framework as new as ours, developing behaviors with in-progress dependencies becomes a difficult task, which became especially apparent during the German Open 2024. To be able to develop behaviors in parallel with other systems, we created a behavior simulation that abstracted all detections. Using this method, we were able to develop more advanced behaviors during the RoboCup 2024, whilst new features were being implemented alongside it.

The simulation provides a simple top-down 2D overview of the field with an interface similar to the GameController. The simulation is done in Bevy, and the interface is implemented using egui[6]. The simulation allows for simulating a normal game's most common situations, like starting procedures and general play, whilst viewing information related to the behaviors and roles.

The recent move of yggdrasil to the Bevy framework will allow us to create a more directly integrated simulation. Our current plan is to expand the 2D simulation so that it allows for further development of behaviors with a high level of abstraction and so that it can be used in our RL pipeline. Tight integration with the framework will also make it possible to move parts of the simulation to 3D, in order to simulate more complex behaviors such as kicks with higher accuracy.

7.3 Reinforcement learning behavior

As part of our behavior engine, over the last year, work has been done on developing a framework that incorporates learned behaviors using reinforcement learning (RL). A framework based on stablebaselines3[7] was developed, together with a structure to create specific environments to train agents for specific behaviors. To train RL behaviors, a simulation based on the AbstractSim developed by WisTex United was used [39]. Several components, including an improved overview of the

[6] `https://github.com/emilk/egui`

[7] `https://stable-baselines3.readthedocs.io/en/master/index.html`

field and a visualizer of the robot vision, were added. An overview of the current simulator is shown in Figure 21.

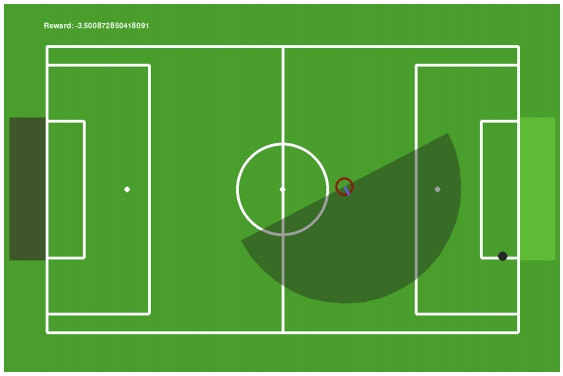


Figure 21: The current simulation used for behavior reinforcement learning.

All behaviors output a walking engine input, which is a value for the forward, left and turn of the requested step. This makes the integration of RL behaviors in the framework straightforward, since we can replace the initial step prediction by the trained RL policy. For this year, the focus was to develop single-agent behaviors. Several single-agent behaviors have been successfully developed and deployed on the robot, including a partially-observable walk-to-ball behavior, which lets the robot to search for the ball and once it is found, walk towards it. After developing single-agent behaviors, the developed framework was rewritten to be able to use the skrl[8] library, which supports multi-agent training and further customization of policy models. This allows for training higher-level behaviors using hierarchical RL or multi-agent learning.

8 Machine Learning Integration

Over the past years, machine learning (ML) has become increasingly important in the SPL league. This year, we have developed various ML models that we currently use in our framework. This chapter explains how we efficiently port these models to the framework, and how we develop the models in a unified style using DNT-ML.

8.1 ML in framework

After a machine learning (ML) model has been trained, the subsequent step is to port it to the framework so that it can run on the NAO robots. However, as most

[8] `https://skrl.readthedocs.io/en/latest/`

ML models are trained in Python, integrating a model into the Rust framework is not trivial. In order to make ML integration less time-consuming, we developed user-friendly functionalities that make it straightforward to load and execute ML models within the framework. As a result, new models can easily be shipped and tested.

8.1.1 Backend

Firstly, we need a backend that provides the core utilities necessary for deploying ML models around which we can build a user-friendly interface. The hard requirements for the backend are that it must be able to

- Load a model from stored weights once, and be able to be infer multiple times without reloading.
- Load input values and fetch output values after execution has finished with minimal latency.
- Support a wide range of easy to use operations, as we want to experiment on models with many different layer types.

As such, we compared three popular Rust solutions that fulfill these requirements: Tract[9], PyTorch bindings[10], and OpenVINO bindings[11]. We judged the performance of these solutions based on the inference times of some canonical computer vision ML models, where a lower value is more favorable. Benchmarks were conducted with Criterion[12].

Backend/Model	ResNet-18	MobileNetv2
OpenVINO	158ms	33ms
tch-rs (PyTorch bindings)	250ms	143ms
tract	9050ms	1805ms

Table 2: Backend performance comparison.

In Table 2, the average results for a single forward pass are shown for ResNet-18 [40] and MobileNetv2 [41]. As OpenVINO outperformed the other backends on both benchmarks by a large margin, we continued to use it as the backend for ML integration.

[9] `https://github.com/sonos/tract`
[10] `https://github.com/LaurentMazare/tch-rs`
[11] `https://github.com/intel/openvino-rs`
[12] `https://github.com/bheisler/criterion.rs`

8.1.2 Interface

Using OpenVINO, we designed an interface where a user only has to provide the file where the model (hyper)parameters are stored and input and output data types (to enable Rust's strong type inference) to create a so-called ML task. These tasks automatically handle checking of input and output sizes and count, their types and output strategy. Users can choose to infer on a separate thread and output to a Bevy resource or entities, or run the model in scope and block until completion.

```rust
/// The robot detection model, based on a VGG-like backbone
/// using SSD detection heads.
pub struct RobotDetectionModel;

impl MlModel for RobotDetectionModel {
    // we define the input as an array of bytes
    type Inputs = Vec<u8>;
    // we define an output as a multidimensional array of boxes
    // and an array of scores
    type Outputs = (MlArray<f32>, Vec<f32>);
    // we need to specify the path to our weights and
    // model definition
    const ONNX_PATH: &str = "models/robot_detection.onnx";
}

fn detect_robots(
    mut commands: Commands,
    mut model: ResMut<ModelExecutor<RobotDetectionModel>>,
    // other data we need ..
) {
    // resize image ..

    commands
        .infer_model(&mut model)
        .with_input(&resized_image)
        .create_resource()
        .spawn({
            // moving data to this thread ..

            // type checked post-processing step
            // (matches definition in MlModel::Outputs)
            move |(box_regression, scores)| {
               // setup some detection data ..

                // automatically gets pushed to a bevy resource
                Some(RobotDetectionData {
                    detected: detected_robots,
                    image: image.clone(),
                    result_cycle: cycle,
                })
            }
        });
}
```

Figure 22: A snippet from yggdrasil, doing inference on the robot detection model.

By using ML tasks, the ML integration workflow has been significantly simplified

and we can now very easily iterate on models in the framework. An example of ML tasks is shown in Figure 22.

8.2 DNT-ML

A crucial part of creating and integrating machine learning models in the framework is training models with varying (hyper)parameters, optimization methods and data augmentations. While straightforward, setting up the full pipeline for each of these models requires time. This time includes setting up the main training loop, choosing data transformations, setting up evaluation metrics, and creating the means to export the results and models obtained from the pipeline.
To streamline this process, we introduce the DNT-ML framework. Through this framework, we abstract away the training pipeline and simplify the pre-, mid- and post-training components that are typically part of a full machine learning (ML) model training pipeline. This is done in a modular fashion, such that any components can effortlessly be added or removed to the pipeline as desired. As such, users need only to define a ML model compatible with the PyTorch library, choose the modules required for their model pipeline, and optionally define custom modules if the existing module set does not fit the pipeline requirements.

8.2.1 Training pipeline

At the very minimum, the training pipeline requires the user to define a PyTorch model, a loss function for their model, an optimization algorithm, a dataloader (custom or otherwise), and the number of epochs required to run the pipeline. With these, the training pipeline is as follows:

1. Initialize pipeline with the given arguments and parameters.
2. At each epoch, iterate through the batches of training data.
3. Perform the basic training operations, including backpropagation and optimization step.
4. Perform a forward and loss calculation for the validation step on the validation data.

This essentially forms the skeleton and only provides the basic operations that any ML model would require to be trained. This simplicity, however, allows us to hook into any point of the timeline, and perform operations with the data obtained throughout the pipeline. For example, the basic pipeline has no evaluation metrics incorporated. However, we can hook into the point after which the train and

validation step were executed, and the data required to compute the metric will be passed on. Likewise, we can also hook in a point after each batch from a dataloader, or right before a train or validation step was performed. This allows for a highly modular training pipeline that can be adapted for any PyTorch model. Furthermore, through the use of dataloaders, we can perform any operations needed to transform or adjust the data in the loader as desired. These modifications can even be chained and thus, also allows for a modular data pre-processing pipeline.

8.2.2 Modules

The modules within the framework are divided into two sections. The first is a pipeline with *component callables*, while the second is made up of *callback functions*. During a train or validation step, the batch retrieved from a dataloader is passed through the timeline to apply transformations to the data. These transformations are applied sequentially and can be done in any form.
The framework is bundled with several basic callables, which can adjust the data structure of the data batch and filter out data based on any user conditions. At the moment, common `torchvision` transforms are also added to the list of built-in callables. Regardless, users can create new callables that can be used in the pipeline through the `nn.Module` module from `PyTorch` by defining a class as a child of this module, and defining a forward function that takes an input and any additional arguments, and returns an output.

The callback functions use predefined hooks in the training pipeline to execute code. Typically, callbacks receive the model, training parameters, configuration dictionaries. Depending on where in the pipeline the callback hooks into, they also receive model outputs, batch data and calculated loss. Like the pre-processing pipeline, the framework houses several pre-defined callbacks and allows users to create their own.

9 Workshops and events

During 2024 DNT has organized and attended multiple events with the workshops and events committee. Throughout the year we have helped on several different occasions where we arrange robot soccer match demonstrations, presentations and python workshops. These are mostly focused on educational purposes and creating an interest in our field of autonomous robotics. However, there are also demonstrations and talks with the purpose of gathering sponsors and creating publicity for DNT.

The DNT presentations include general information about DNT, some information

about the Nao robots we work with, and the Robocup and SPL rules and what we need to do in order to make the robots play soccer on their own. Depending on the group, their initial knowledge, and age we make the presentation longer or shorter, for example by adding more technical details about what we work on.
For our match demonstrations, we will have two teams of robots, either 2 vs 2 or 1 vs 1, depending on the number of robots we have available. We then play a 10 minute match where visitors can see the capabilities of our robots and learn how a SPL match looks and works.
We have also given two Python workshops, which involved creating simple assignments to teach the beginnings of Python to senior highschool students. The topics that are included in these workshops are variables, operators, if statements and loops. We aim to create fun and easy exercises that help these students understand the concept of programming, and enable them to make small python programs of their own.

9.1 List of activities

As a team, we give demonstrations, presentations, and workshops about what we do. The following subsections provide a list and descriptions of all the events, demonstrations, and workshops DNT attended and organised last year. In addition to these big events, we also give presentations and demonstrations to small groups of people, who, for example, get a tour around Lab42.

9.1.1 Sponsor Event RoboCup

This was an event we attended with the intention of showing our work to different companies to make them excited for RoboCup and find new RoboCup sponsors. Around 150 people from different companies attended. Different aspects of the RoboCup will be shown. The exposition is a trade fair setup in which different teams present/demonstrate themselves. The focus was mostly on telling them what we do at the RoboCup. We also brought two robots for small demonstrations.

9.1.2 Visit of Metis College - February

Forty highschool students of the Metis college who have been following a computer science course in school, will visit our lab. We have given them a succinct presentation that includes an introduction to the team and some background information on the assignments they will be making. We also gave a brief demo of 1 robot following the ball and scoring on the field. Showing its different abilities such as ball and field detection, the walking engine, being able to stand up after falling, and all the other functions we implemented. After this, we have a workshop planned

for them where they will do short programming assignments to learn the basics of python programming.

9.1.3 School Visit - March

Sixteen eighteen year old students paid a visit to Amsterdam Science park, of which the Robolab was a part as well. They were studying in their final year of computer science specialization (IIS Avogadro, Turin). We held a brief general presentation of DNT and a 2v2 soccer match demonstration.

9.1.4 Career Day - March

The Career day is an event for highschoolers who are choosing a study. It is organized by JetNet. The day is divided into rounds, within each round a group of students comes to our stand. We have 5 minutes for explanation and 20 minutes for interactive activities. The focus lies on what kind of studies the team members do and what kind of career they can expect if they follow our path.

9.1.5 Girls Day - April

Forty two girls from highschool VWO 1 come in groups of three after lunch to join a carousel of activities at science park, our robolab is one of these activities. The girls day is meant for girls that are interested in scientific subjects. They will be divided into three different groups. Each group will get a demo and a presentation of 25 minutes in total.

9.1.6 School Visit Belgium - May

On this day, a school from Belgium visited the lab. The group consisted of 30 students of approximately 17 years old. Their visit begun with a presentation and demo, after this they worked on the python programming workshop we provide.

9.1.7 24 uur Oost - September

The Dutch Nao Team participated in 24 uur Oost, an event organized by Amsterdam where there were several activities all in the eastern part of Amsterdam. For this event, we gave demonstrations and presentations about what we do as a team.

9.1.8 Weekend of Science - October

The "weekend van de wetenschap" or weekend of science is a day when Amsterdam Science park is open for everyone to visit and learn about the research that is done

there and the topics that the students at science park are learning about. There are multiple small activities, mainly focused at younger kids to let them learn about scientific topics in a fun way. On this day DNT showed them a demonstration by doing a 2v2 soccer match. Every hour there was a scheduled match that people could join to watch. Between the scheduled matches we stood outside the lab with one robot in NAOqi, for children to interact with.

9.1.9 UvA Open Campus Day - November

The open campus day is an event where senior highschoolers who want to choose a study at a university can come to the UvA to look around, talk to students and visit presentations about different study directions. DNT gave a scheduled robot soccer demonstration every 45 minutes, and in the meanwhile answered questions about DNT our study and the UvA.

9.1.10 Visit of French Students - November

On this day highschoolers from France got a tour and some workshops from FNWI at science park to learn more about what happens at universities. We were an activity for the group as well. The focus was on showing them what we work on in the team. We gave them a presentation about DNT and a demonstration in the form of a 2v2 match. Afterwards we did a small quiz.

9.1.11 Startup Village Visits - throughout the year

DNT has a partnership deal with Startup Village (SUV) at science park. This deal entails that they sponsor us with a certain amount of money and they can schedule visits to our lab. When they have a group or company visiting them, they can come to the Intelligent Robotics Lab and we will give them a talk about DNT and what we do, and show them a demo with our robots. These are mostly short 10-15 minute visits with a small group of ± 20 people. They happen more often throughout the year. Examples of groups we hosted are a Swedish delegation, a delegation from Krakow, master students from the NTNU Trondheim and groups from small companies.

9.2 SPL events

Throughout the year, several events for and by the Standard Platform League of the RoboCup are organized. In 2024, the Dutch Nao Team participated in three events: the German Open, the Robocup, and the RoHOW.

9.2.1 German Open

The German Open is an event from the RoboCup where a subset of SPL teams (and other leagues) gather to play matches against each other and spend their week programming and improving their performance in Germany. This year the German Open took place in Kassel. The German Open took place in April. For our team, the main goal of the GO was to be able play with yggdrasil for the first time. We ended last, but were able to play with our framework and made a lot of progress during the week.

9.2.2 RoboCup

The RoboCup is the main robotics competition where teams from all over the world come together to play competions. In July 2024, the RoboCup was hosted in the Netherlands in Eindhoven. The teams have spend one week where they got to work on their software and test them in the environmental conditions at the RoboCup. The first 2 days are primarily for improving and testing before the matches begin. After this the matches will be held until the final competition on the last day. During this week DNT has worked on several improvements, under which improving the ball and robot detection models and refining behaviors, as well as collecting data for future datasets. We placed fifth out of the six teams that participated in the Challenge Shield of the SPL league.

9.2.3 RoHOW

RoHOW, the Robotics Hamburg Open Workshop Event, is organized by SPL team HULKs and is held at the Technical University of Hamburg in November. There are seminars and workshops about Robotics within our league held by other teams, but there is also the possibility to request a certain seminar and/or give your own. As usual, the Dutch Nao Team participated. We also won the yearly SPL Quiz and have won ourselves the honor to prepare the one for next year's event.

10 Plans for 2025

In 2024, the foundation of `yggdrasil` has been continued and developed into a framework that allows us to play matches. For 2025, this means that the focus can shift from developing elements necessary to play matches to developing elements that enable us play better matches. In the first part of 2025, the focus will be working towards the German Open 2025 and RoboCup Brazil. In the second part of 2025, the performance of DNT on the RoboCup Brazil will be analyzed to determine a roadmap towards RoboCup 2026.

10.1 Software

For the software development roadmap in 2025, our primary focus will be on strengthening the core framework while simultaneously preparing for the integration of more RL-based behavior policies. Finally, we are planning to make our code open-source at the RoboCup German Open 2025.

A large part of our efforts will be spent on expanding the framework to build a strong foundation for the wide RL integration. Example tasks include expanding nidhogg so that simulators are as frictionless to use as real NAOs, or more code sharing between the framework and 2D behavior simulation. Another part of this effort is better visualization and evaluation of our gameplay. To support this initiative, we will be building out more advanced tooling with Rerun.

In the shorter term, we are planning to finalize the visual localization system. This will significantly improve our pose estimation, which is a hard requirement for success. The walking engine will be improved as well, with a complete rewrite in the works that includes in-walk kick motions. The goal is to improve stability and speed, while also making the code easier to use in simulation environments. These two tasks should allow us to score our first official goals against opponents using the yggdrasil framework at the German Open.

10.2 AI

For the projects around artificial intelligence, the main focus for 2025 will lie on reinforcement learning. In 2024, a large part of the needed object detection models was developed, which allows for more research projects and state-of-the-art development in 2025. For the Dutch Nao Team, a large part of the focus will go towards integrating reinforcement learning in most layers of the framework. We are planning to use reinforcement learning to replace a large part of the behavior engine, which allows to integrate tactics and teamplay, things that are now lacking in the behaviors. To do this, we want to use hierarchical RL to both make higher level decision policies and lower level policies that execute behaviors.

Furthermore, we will focus on doing motion with 3D reinforcement learning. This year, we developed an accurate 3D simulation of the NAO that can be used for training models. In 2025, we want to use this to replace and extend multiple motion-specific aspects of the framework. Specifically, we want to continue our research in enhancing the walking engine with RL. Furthermore, we want to start researching the possibilities for usage of RL in getup motions, damage prevention and balancing.

Lastly, we plan to continue improving the current object detection models. For this, we want to explore several possibilities, including the use of a shared feature backbone and moving to new state-of-the-art object detectors, such as YOLO11. Upcoming challenges such as the visual referee pose estimation requirements for upcoming matches will benefit greatly from these improvements.

10.3 Management, board and committees

For the management, the goal of 2025 will be to continue the way the team is organized as smoothly as possible. Since the members of the management will most likely leave the team halfway through 2025, the focus of the current management will be to enable a smooth transition to a new management, while also organizing the team in working towards RoboCup Brasil.

The board will continue to take care of the financial and legal aspects of the foundation and team.

10.4 Committees

In terms of committees, we plan to continue with the nice performance of the Workshops and events committee, that is currently one of the highlights of the team as it provides us with incomes and a nice exposure to the exterior.

If last year we put lots of efforts in the social media committee, successfully enhancing both the feed and overall presentation of our social media platforms, our primary focus for the coming year is to further develop the Partnerships Committee. While this committee had not been a major focus until recent months, where we started it to really push it forwards. We have high expectations for this committee and their new members.

11 Contributions

The following list summarizes the contributions of everyone that worked on the tech report, in alphabetical order:

- **Fiona Nagelhout** acted as lead of the workshop and events team. She was responsible for organizing all workshops, talks and presentations of the Dutch Nao Team. Furthermore, she helped organizing the participation of the DNT in the SPL events.

- **Fyor Klein Gunnewiek** developed the current behavior engine and behavior simulation.
- **Gijs de Jong** acted as software tech lead. Furthermore, he worked on the bevy framework, the walking engine and several AI vision projects.
- **Harold Ruiter** acted as team lead in the new management. Furthermore, he worked on tyr/the bevy framework, the ball proposals and localization.
- **John Yao** worked on NAO head interpolation. Furthermore, he has been actively involved in the outreach committee.
- **Joost Weerheim** acted as secretary of the board and built upon the work of Macha and Gijs for robot detection.
- **Juell Sprott** developed the DNT-ML framework.
- **Julia Blaauboer** developed the robot-to-robot communication framework, refactored the kinematics interface, and worked on obstacle avoidance and pathfinding.
- **Macha Meijer** acted as AI tech lead. Furthermore, she developed the behavior reinforcement learning framework and worked closely with Gijs on various detection models, including robot detection and ball classification.
- **Marina Orozco González** acted as operations lead in the new management. Furthermore, she contributed in the AI team on multiple projects, including color calibration, ball detection and field boundary detection.
- **Mark Honkoop** worked on the initial versions of the scan lines and line detection. Furthermore, he is our most treasured pull request reviewer.
- **Morris de Haan** developed the whistle detection model and improved the ball proposals, making them more efficient.
- **Rick van der Veen** developed the rerun control integration, greatly improving developer experience.
- **Stephan Visser** acted as treasurer of the board and developed the keyframe motion engine along with current motion composition.

12 Conclusion

In this report, the work of the Dutch Nao Team in 2024 was discussed. Details of all projects that were done over the past year on framework, sensing, vision, motion, behavior and machine learning integration were discussed, together with

possible future improvements. Furthermore, an overview of all past events was given. On the organizational side, an overview of the newly developed team and management structure was given, together with a clear definition of all tasks in the management and board. Lastly, plans for 2025 for software, AI, organization, and committees were discussed. Based on the developments this year, the goal is to be able to play good matches and score goals at the RoboCup Brasil.

References

[1] A. Visser, R. Iepsma, M. van Bellen, R. K. Gupta, and B. Khalesi, *Dutch nao team – team description paper – standard platform league – german open 2010*, Jan. 30, 2010.

[2] S. Oomes, P. Jonker, M. Poel, A. Visser, and M. Wiering, "The dutch aibo team 2004," Jul. 1, 2004.

[3] J. Kaiser, R. Geurts, H. Ruiter, G. de Jong, and M. O. Gonzalez, "Team description paper dutch nao team 2024," Tech. Rep., Feb. 13, 2024.

[4] W. Duivenvoorden, G. de Jong, H. Ruiter, *et al.*, "Team qualification document for robocup 2023," Tech. Rep., Feb. 13, 2023.

[5] W. Duivenvoorden, H. L. gezegd Deprez, T. Wiggers, *et al.*, "Team qualification document for robocup 2022 bangkok, thailand," Tech. Rep., Feb. 14, 2022.

[6] HULKs, *Hulk*, `https://github.com/HULKs/hulk`, Accessed: 2024-12-26, 2024.

[7] B. Contributors, *Bevy engine*, `https://github.com/bevyengine/bevy/releases/tag/v0.10.0`, version 0.15.0, Nov. 30, 2023.

[8] Rerun Development Team, *Rerun: A visualization sdk for multimodal data*, version 0.21.0, Available from `https://www.rerun.io/` and `https://github.com/rerun-io/rerun`, Online, 2024.

[9] J. Richter-Klug, "Visuelle odometrie in der robocup standard platform league," M.S. thesis, University of Bremen, 2018.

[10] D. Laidig and T. Seel, "Vqf: Highly accurate imu orientation estimation with bias estimation and magnetic disturbance rejection," *Information Fusion*, vol. 91, pp. 187–204, Mar. 2023, ISSN: 1566-2535. DOI: `10.1016/j.inffus.2022.10.014`. [Online]. Available: `http://dx.doi.org/10.1016/j.inffus.2022.10.014`.

[11] S. Madgwick *et al.*, "An efficient orientation filter for inertial and inertial/magnetic sensor arrays," *Report x-io and University of Bristol (UK)*, vol. 25, pp. 113–118, 2010.

[12] R. G. Valenti, I. Dryanovski, and J. Xiao, "Keeping a good attitude: A quaternion-based orientation filter for imus and margs," *Sensors*, vol. 15, no. 8, pp. 19 302–19 330, 2015.

[13] N. Backer and A. Visser, "Learning to recognize horn and whistle sounds for humanoid robots," Nov. 7, 2014.

[14] R. N. Bracewell, "The fourier transform," *Scientific American*, vol. 260, no. 6, pp. 86–95, 1989.

[15] M. Ashouri, F. F. Silva, and C. L. Bak, "Application of short-time fourier transform for harmonic-based protection of meshed vsc-mtdc grids," *The Journal of Engineering*, vol. 2019, no. 16, pp. 1439–1443, 2019.

[16] N. W. Backer, B. O. K. Intelligentie, and A. Visser, "Horn and whistle recognition techniques for nao robots," *Bachelor thesis, Universiteit van Amsterdam*, 2014.

[17] H. Aagela, V. Holmes, M. Dhimish, and D. Wilson, "Impact of video streaming quality on bandwidth in humanoid robot nao connected to the cloud," in *Proceedings of the Second International Conference on Internet of things, Data and Cloud Computing*, 2017, pp. 1–8.

[18] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000.

[19] T. Röfer, T. Laue, A. Baude, *et al.*, *B-Human team report and code release 2019*, Only available online: `http://www.b-human.de/downloads/publications/2019/CodeRelease2019.pdf`, 2019.

[20] A. Hasselbring and A. Baude, "Soccer field boundary detection using convolutional neural networks," in *Robot World Cup*, Springer, 2021, pp. 202–213.

[21] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[22] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural networks*, vol. 107, pp. 3–11, 2018.

[23] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, PMLR, 2019, pp. 6105–6114.

[24] M. Tan and Q. Le, "Efficientnetv2: Smaller models and faster training," in *International conference on machine learning*, PMLR, 2021, pp. 10 096–10 106.

[25] L. Sifre and S. Mallat, "Rigid-motion scattering for texture classification," *arXiv preprint arXiv:1403.1687*, 2014.

[26] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7132–7141.

[27] R. Girshick, "Fast r-cnn," *arXiv preprint arXiv:1504.08083*, 2015.

[28] Z. Yao, W. Douglas, S. O'Keeffe, and R. Villing, "Faster yolo-lite: Faster object detection on robot and edge devices," in *Robot World Cup*, Springer, 2021, pp. 226–237.

[29] Y. Peng, H. Li, P. Wu, Y. Zhang, X. Sun, and F. Wu, *D-fine: Redefine regression task in detrs as fine-grained distribution refinement*, 2024. arXiv: 2410.13842 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2410.13842.

[30] K. Han, Y. Wang, Q. Tian, J. Guo, C. Xu, and C. Xu, "Ghostnet: More features from cheap operations," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 1580–1589.

[31] G. de Jong, H. Ruiter, D. W. Prinzhorn, *et al.*, "Dutch nao team - technical report," Tech. Rep., Dec. 31, 2023.

[32] B. Hengst, "Runswift walk2014 report robocup standard platform league," *School of Computer Science and Eng., Univ. of New South Wales*, 2014.

[33] G. de Jong, L. Eshuijs, and A. Visser, "Learning to walk with a soft actor-critic approach," in *Proceedings of the 35th Benelux Conference on Artificial Intelligence (BNAIC 2023)*, 2023.

[34] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the reality gap: The use of simulation in evolutionary robotics," in *Advances in Artificial Life: Third European Conference on Artificial Life Granada, Spain, June 4–6, 1995 Proceedings 3*, Springer, 1995, pp. 704–720.

[35] K. Bousmalis, A. Irpan, P. Wohlhart, *et al.*, "Using simulation and domain adaptation to improve efficiency of deep robotic grasping," in *2018 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2018, pp. 4243–4250.

[36] F. K. Gunnewiek, *Quantifying the reality gap in abstracted pedestrian detection in simulated environments*, Bachelor's thesis, Jun. 2023. [Online]. Available: https://staff.fnwi.uva.nl/a.visser/education/bachelorINF/Bachelor_Thesis_Fyor_Klein_Gunnewiek.pdf.

[37] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[38] M. Rahme, I. Abraham, M. L. Elwin, and T. D. Murphey, "Dynamics and domain randomized gait modulation with bezier curves for sim-to-real legged locomotion," *arXiv preprint arXiv:2010.12070*, 2020.

[39] W. United, *Abstractsim*. [Online]. Available: https://wistex-united.github.io/docs/Code/AbstractSim/AbstractSim.html.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[41] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.