INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Lucas Steuernagel

OPPONENT MODELING FOR ROBOCUP SMALL SIZE LEAGUE ROBOTS

Final Paper
2021

Course of Computer Engineering

CDU 681.3:16

Lucas Steuernagel

OPPONENT MODELING FOR ROBOCUP SMALL SIZE LEAGUE ROBOTS

Advisor

Prof. Dr. Marcos Ricardo Omena de Albuquerque Máximo (ITA)

COMPUTER ENGINEERING

São José dos Campos
Instituto Tecnológico de Aeronáutica

2021

Cataloging-in Publication Data
Documentation and Information Division

Steuernagel, Lucas
Opponent Modeling for RoboCup Small Size League Robots / Lucas Steuernagel.
São José dos Campos, 2021.
63f.

Final paper (Undergraduation study) – Course of Computer Engineering– Instituto Tecnológico de Aeronáutica, 2021. Advisor: Prof. Dr. Marcos Ricardo Omena de Albuquerque Máximo.

1. Dinâmica de robôs. 2. Aprendizagem (inteligência artificial). 3. Controle de robôs. 4. Robótica. 5. Controle. 6. Computação. I. Instituto Tecnológico de Aeronáutica. II.Opponent Modeling for RoboCup Small Size League Robots.

BIBLIOGRAPHIC REFERENCE

STEUERNAGEL, Lucas. **Opponent Modeling for RoboCup Small Size League Robots**. 2021. 63f. Final paper (Undergraduation study) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

Lucas Steuernagel
Rua Aracaju, 489
89221-637 – Joinville–SC

OPPONENT MODELING FOR ROBOCUP SMALL SIZE LEAGUE ROBOTS

This publication was accepted like Final Work of Undergraduation Study

Lucas Steuernagel
Author

Marcos Ricardo Omena de Albuquerque Máximo (ITA)
Advisor

Prof. Dr. Marcos Ricardo Omena de Albuquerque Máximo
Course Coordinator of Computer Engineering

São José dos Campos: November 24, 2021.

To those who have supported me through tough times: my family and my closest friends.

Acknowledgments

I would like to thank professor Maximo who has been guiding this project since the beginning and ITAndroids, the university robotics team, which provided the opportunity of research. I also appreciate professor Ana Carolina Lorena's and Elton Sbruzzi's availability for watching my presentation.

"A man is always a teller of tales,
he lives surrounded by his stories and the stories of others,
he sees everything that happens to him though them;
and he tries to live his life as if he were recounting it."
— Jean-Paul Sartre

Resumo

A RoboCup é uma comunidade científica internacional com foco em robôs inteligentes. O *Small Size League* (SSL) é uma das categorias de futebol RoboCup, cujos robôs são cilíndricos com no máximo 18 cm de diâmetro e 15 cm de altura. A ITAndroids - o time de robótica do Instituto Tecnológico de Aeronáutica - participa ativamente nessa categoria. Jogos do SSL são muito competitivos, e os times com a melhor estratégia possuem clara vatangem durante o jogo. Um passo essencial para atingir-se uma tomada de decisão boa é saber qual a trajetória futura do robô oponente. Além disso, uma predição fidedigna ajuda o time a planejar melhor sua trajetória para evitar colisões e fazer com que os robôs chegem ao seu destino mais rápido. Com o conjunto de dados público dos logs de jogo da RoboCup SSL, podem-se extrair as posições dos robôs, suas oritentações e a posição da bala a cada *frame* da visão. Utilizou-se um *smoother* de Kalman para remover o ruído dos dados e interpolar medições faltantes. O *smoother* também calcula a velocidade de cada objeto. Propõem-se duas redes neurais de arquitetura *encoder-decoder* para resolver o problem de predição: uma rede com um único *encoder*, que consome somente dados sobre os movimentos passados do robôs, e uma rede com *encoder* duplo, que emprega dados de ambos o robô e a bola. A entrada da rede para o robô é uma sequencia temporal de sua posição, velocidade e orientação. Outrossim, a entrada para a bola é um sequência temporal de sua posição e velocidade. Cada rede prevê uma série temporal de velocidades futuras para o robô. Faz-se integração para se obter a posição a partir das velocidades. Encontrou-se que a rede de *encoder* duplo funciona melhor para uma janela de passado de 30 passos de tempo (equivalente a 0,5 segundos) para prever 15 (0,25 segundos) passos de tempo futuros. Ela atinge resultados 54,5% menores para o erro de deslocamento médio se comparada com o preditor de Kalman. A rede de *encoder* único funciona melhor para prever 30 passos de tempo futuros (0,5 segundos) usando uma janela de passdo de 60 (1 segundo) e atinge resultados de erro de deslocamento médio 41,6% menores do que os do preditor de Kalman. Estes resultados são excepcionals, posto que o preditor de Kalman é o principal método usado para predição de trajetórias de robôs no contexto da RoboCup SSL. As arquiteturas apresentadas possuem potencial de fazer a tomada de decisão e o planjemanto de trajetória do time melhor e mais eficiente.

Abstract

The RoboCup is an international scientific community focused on intelligent robots. The small size league (SSL) is a mainstream category of the RoboCup soccer games, whose robots must fit in a cylinder of 18 cm diameter and 15 cm height. The ITAndroids – Aeronautics Institute of Technology robotics team – actively participates in such a category. SSL games are very competitive, and teams with the best game strategy have clear advantage during games. An essential step to achieve a good decision making is knowing what the future trajectory of opponent's robots is. Furthermore, a trustworthy prediction helps the team plan its trajectory to better avoid collisions and make the robots reach their destination faster. From the publicly available dataset of RoboCup SSL games' logs, we can extract the robots' position and orientation and the ball's position at each vision frame. We employ a Kalman smoother to remove noise from data and interpolate missing measurements. The smoother also calculates the speed for each object. We proposed two encoder-decoder network architectures for solving the forecasting problem: a single encoder network, which consumes only data from the robot's past motion, and a double encoder network, which employs data from both the robot and the ball. The network's input for the robot is a time series of its position, velocity and orientation. Likewise, the input for the ball is a time series of its position and velocity. Each network predicts a time series of future velocities for the robot. We perform integration to obtain the position from the speeds. We found out that the double encoder network performs better for a look back window of 30 time steps (equivalent to 0.5 seconds) to predict 15 (0.25 seconds) future time steps. It achieves 54.5% lower scores for Average Displacement error in comparison to the Kalman predictor. The single encoder network works better for predicting 30 future time steps (0.5 seconds) using a look back window of 60 (1 second) and achieves Average Displacement Error scores 41.6% lower than those of the Kalman predictor. These are remarkable results, as the Kalman predictor is the main method used for trajectory prediction in the context of RoboCup SSL. Both networks excelled at predicting the robot's trajectory and outperformed the Kalman predictor. The presented architectures have the potential to make our decision making and trajectory planning better and more efficient.

List of Figures

List of Tables

Contents

1 Introduction

RoboCup is an international scientific community focused on intelligent robots. It aims at developing robots and artificial intelligence to beat the best human soccer player team until 2050 (KITANO *et al.*, 1997). RoboCup has many robotic categories, but ITAndroids – the Aeronautics Institute of Technology robotics group – only participates in soccer related ones.

Given the general idea of our robotics group and the main competition the team is focused on, we are going to explain in the following sections the context of our work, our motivation and goals, as well as the related work.

1.1 Context

One of the mainstream RoboCup soccer categories is the Small Size League (SSL), whose robots are cylindric with 18 cm diameter and 15 cm height (ROBOCUP, 2021). A mock up of ITAndroids SSL is shown in Figure 1.1. Teams consist of eleven robots in division A and six robots in division B.

The main research areas for this league are control theory, navigation and opponent modeling. As robots move very quickly during the game, there has been efforts in optimizing the control of omnidirectional robots (BALABAN *et al.*, 2018) and in improving ball interception (MAKAROV *et al.*, 2019). For the same reason, there are many studies to make navigation more efficient. One remarkable paper proposes using a behavior tree for better path finding (ABIYEV *et al.*, 2014).

Lastly, as the games are increasingly competitive, many teams are trying to predict the opponent"s behavior and trajectory to leverage improved response for attacks. In a 2016 paper, researches try to classify possible behaviors for SSL robots by clustering them in an online fashion (ADACHI *et al.*, 2019a).

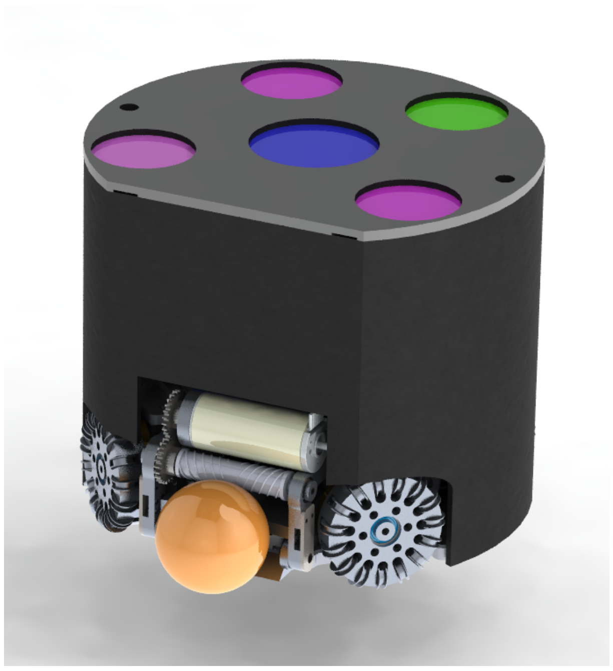


FIGURE 1.1 – Mock up of ITAndroids SSL robot.

1.2 Motivation

SSL teams play against each other in a soccer game. The games are increasingly dynamic as robots are omnidirectional, can quickly change direction and move fast. In such a context, teams have constant pressure to improve their strategies and predict their opponents' behavior and trajectory.

Presently, our team uses a Kalman filter to estimate the opponent's position. We also leverage it to predict the future position of the opponent's robots. Such forecast is necessary for planning the team's behavior and movement so that we can leverage the predicted movements to improve our own.

The Kalman predictor, however, has been a poor approximation of a robot's next move, for it does not consider many factors that influence the game. A robot decides its position based upon not only kinematics factors, but also the position of the ball and other robots and the current state of the game.

A precise trajectory prediction is useful for better decision making and for improving our motion planning. Teams that better predict the opponent's motion can leverage this to plan their trajectory, avoid collisions and reach the destination faster.

In such situation, we believe that an approach using machine learning might provide us better predictions. We are currently interested in investigating sequence-to-sequence models, as the robots' positions and velocities compose a time series, in which past values may influence future ones. We have seen tremendous advances in research for tackling time series problems, such as Attention (VASWANI *et al.*, 2017), MomentumRNN (NGUYEN *et al.*, 2020) and Bayesian Neural Networks (JOSPIN *et al.*, 2020).

1.3 Goals

In face of the recently developments of machine learning to deal with time series and improvements in algorithms that can exploit dependencies of past actions to predict future ones, we are planning to predict SSL opponents trajectories, including position and velocity, so that our robots can better evaluate how to plan their decisions and motion. In addition, we are going to compare the results with those from our Kalman predictor to evaluate the applicability of our proposed technique.

1.4 Related Works

In the RoboCup context, there has already been attempts to predict opponents' behavior. Adachi *et al.* (2017) and Adachi *et al.* (2019b) propose a classification for all possible actions of SSL robots during a game. They use their classes to cluster robots' movements into groups and identify their behavior, using similarities between robots' trajectories. Likewise, in Erdogan and Veloso (2018), researchers define a "behavior" as a trajectory of the robot during the period a team has the control of the ball. They then cluster behaviors according to their similarity. During a game, the system is supposed to identify which trajectory pattern a robot is executing among the classified ones. The authors cite, however, that, although their algorithm could effectively detect the rival's robot behavior, it did not do so fast enough for the team to adapt to it.

In the broad literature, there has been efforts to employ deep reinforcement learning to model agents (ALBRECHT; STONE, 2018). In He *et al.* (2016), the authors present a model that jointly learns the policy and behavior of opponents, and in Raileanu *et al.* (2018), the agents use their own policy to predict other agents' action. Furthermore, Papoudakis and Albrecht (2020) uses variational autoencoders to bypass the assumption of having access to opponents' observations during training and execution. It is worth citing that in cases when multiple agents are seen as a unique entity, it may be useful to model the behavior of all robots at once (ALBRECHT; STONE, 2018). This may be the case of SSL soccer, in which a single agent controls every robot.

All the aforementioned works have tried to predict an opponent's behavior. Our work aims at predicting opponents' trajectories (velocities and positions) by abstracting robots behavior in a time series forecasting neural network. The contribution of our work is adapting time series forecasting to learn robots' behavior and predict trajectories, aiming at an efficiency superior to a Kalman filter, which does not model behavior patterns.

There has been multiple works regarding trajectory prediction of different types of agents. Park *et al.* (2018) introduces an encoder-decoder architecture to forecast the trajectory of multiple vehicles. They utilize a occupancy grip map, whose cells should contain a maximum of one road car, to reduce the prediction into a classification problem. Although the system showed promising results for the proposed experiments, the Robocup SSL field is too big to be divided into cells that contain a single robot, rendering an occupancy grid map too computationally complex.

In addition, Capobianco *et al.* (2021) presents an encoder-decoder architecture that contains an attention module to aggregate information from the past and the future for vessel trajectory prediction. They also analyze the influence of consuming the vessel's destination to improve prediction.

Regarding multi-agent prediction, Ivanovic and Pavone (2019) proposed using a graph to account for the influence of neighboring pedestrians in the trajectory of the target agent, whose path we want to predict, in an attempt to obtain a better forecast. The model has been updated in Salzmann *et al.* (2020) to include the influence of many types of agents in the trajectory of each other, providing a more accurate prediction for a multi-agent scenario.

Researchers have also tried to incorporate acquired knowledge from other artificial intelligence fields into trajectory prediction with outstanding results. Giuliari *et al.* (2021) have embraced the use of transformers, commonly present in the natural language processing field. Similarly, Gupta *et al.* (2018) use a generative adversarial network for training a trajectory forecasting model. The model tries to trick an adversarial network that the predicted result is indeed a real trajectory.

Our work mingles the aforementioned techniques and ideas into the RoboCup SSL world. We could leverage an encoder-decoder architecture using attention to successfully predict the trajectory of robots, without the burden of an occupancy grid map.

1.5 Organization of this work

This chapter introduced our research subjects and gave a perspective of related works. The rest of this paper is organized as follows.

1. Chapter 2 provides the theoretical background for many terms and topics we use throughout the text and reports the mainstream trends in our research area.

2. Chapter 3 explains our data source and what types of information we can extract

from it.

3. Chapter 4 describes our method for leveraging the dataset to build a forecasting model.
4. Chapter 5 presents the results of our experiments and discusses them.
5. Chapter 6 closes our work, presenting our conclusions and future work.

2 Deep learning for time series

In this section, we describe the mainstream techniques in deep learning for time series forecasting. We begin introducing machine learning and deep learning. Then, we explore the specific techniques to apply deep learning in time series.

2.1 Machine learning

Machine learning is an optimization technique to find best functions for predicting the outcome of an event described by several inputs. Its origin dates back to 1950s when it was first featured in the multilayer perceptron neural network. "It is just a mathematical function mapping some set of input values to output values" (GOODFELLOW *et al.*, 2016). The basic formula of a neuron is given by the following:

$$f(\boldsymbol{x}) = g\left(\sum_{i=0}^{n} x_i a_i + b\right), \tag{2.1}$$

where $\boldsymbol{x}$ is a vector of inputs to a neuron, a_i are the neuron's weights and b is a bias factor. g is an activation function, which normalizes the outputs of the neuron, so that they stay between a predefined range. In addition, the activation function usually inserts non-linear operations to a neural network.

Common activation functions include the linear activation $g(x) = x$, the hyperbolic tangent $g(x) = \tanh(x)$ and the rectified linear unit (ReLu), defined as:

$$g(x) = \begin{cases} x & x \geq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

One problem in real world applications of artificial intelligence is that many factors influence a single set of inputs to a problem. Learning algorithms must be able to generalize many representations for the same input to be successful. Deep learning techniques have evolved from the shallow neural networks in a quest to solve the problem of representation. "Deep learning enables the computer to build complex concepts out of simpler ones" (GOODFELLOW *et al.*, 2016).

On the scope of this work, we are inquiring into the effectiveness of deep learning for time series to model the future behavior (position and velocity) of small size league robots. In the following sections, we are going to introduce the subject and present many techniques to work with time series and deep learning.

2.2 Time series forecasting

Time series forecasting predict future values of a target y for a given input x at a time t. In the simplest form, the problem can be modeled as:

$$\hat{y}_{t+1} = f\left(y_{t-k:t}, \boldsymbol{x}_{t-k:t}, \boldsymbol{s}\right). \tag{2.3}$$

In (2.3), $\hat{y}_{t+1}$ is the model forecast, $y_{t-k:t}$ and $\boldsymbol{x}_{t-k:t}$ are observations of the target and exogenous inputs over a look-back window k (LIM; ZOHREN, 2021). f is the prediction function and $\boldsymbol{s}$ is a variable that carries metadata about the input observations (e.g. sensor position).

In many applications, it is better to predict estimates at many points in the future. This way, we can better optimize the outcome of our future actions. To do so, we adapt the definition in (2.3) to the following:

$$\hat{y}_{t+\tau} = f\left(y_{t-k:t}, \boldsymbol{x}_{t-k:t}, \boldsymbol{s}, \boldsymbol{u}_{t-k:t}, \tau\right), \tag{2.4}$$

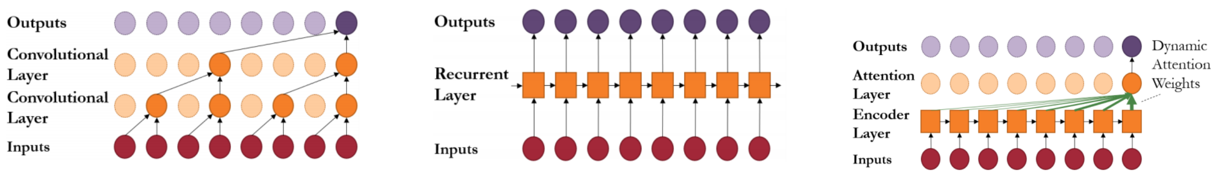
where $\tau \in \{1, ..., \tau_{max}\}$ is a forecast horizon, and $\boldsymbol{u}_t$ are known future inputs across the entire horizon (e.g. date, time stamp, day of the week).

Deep neural networks use a series of non-linear layers to encode intermediate feature representations. In the context of time series, they compile relevant historical information into a latent variable (LIM; ZOHREN, 2021). The final forecast is produced using the latent variable $\boldsymbol{z}$, the encoder h_{enc} and the decoder functions h_{dec}. Equations (2.5) and (2.6) better illustrate this reasoning. This architecture is known as encoder-decoder.

$$f\left(y_{t-k:t}, \boldsymbol{x}_{t-k:t}, \boldsymbol{s}\right) = h_{dec}(\boldsymbol{z}), \tag{2.5}$$

$$\boldsymbol{z} = h_{enc}\left(y_{t-k:t}, \boldsymbol{x}_{t-k:t}, \boldsymbol{s}\right). \tag{2.6}$$

The latent variable $\boldsymbol{z}$ is a fixed length tensor. This allows us to have no constraints



(a) CNN model. (b) RNN model. (c) Attention model.

FIGURE 2.1 – Mainstream building blocks for time series forecasting with deep learning. Extracted from Lim and Zohren (2021).

to the size of the input and the output sequence when we employ an encoder-decoder architecture (SUTSKEVER *et al.*, 2014). It means that the same model works for multiple lengths of inputs and outputs. In addition, such architecture facilitates training a single end-to-end model directly on the sequences (SUTSKEVER *et al.*, 2014).

In the following subsections, we describe Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN) and Attention. They are the mainstream building blocks for time series forecasting with deep neural networks. Figure 2.1 illustrates them.

2.2.1 Convolutional Neural Networks

CNNs are traditionally used in computer vision problems, because they are efficient at extracting relationships that are invariant across spatial dimensions (GOODFELLOW *et al.*, 2016; KRIZHEVSKY *et al.*, 2017).

To predict time series, researchers utilize convolutional filters adapted to account only past information for forecasting. This technique is known as casual convolution (LIM; ZOHREN, 2021). It takes the form described below for a feature at a hidden layer l:

$$\boldsymbol{h}_t^{l+1} = g\left(\left(\boldsymbol{W} * \boldsymbol{h}\right)(l,t)\right), \tag{2.7}$$

$$\left(\boldsymbol{W} * \boldsymbol{h}\right)(l,t) = \sum_{\tau=0}^{k} \boldsymbol{W}(l,\tau)\boldsymbol{h}_{t-\tau}^{l}, \tag{2.8}$$

where $\boldsymbol{h}_t^l$ represents an intermediate state at layer l and time t, $*$ is the convolution operator, $\boldsymbol{W}$ is a constant weight tensor at layer l, and g is an activation function.

In the time series context, the CNN's extracted relationships that are invariant across spatial dimensions represent time-invariant patterns. In addition, CNNs are restricted to use only the inputs in its defined lookback window to make forecasts, so they must be carefully tuned for the specific needs of an application (LIM; ZOHREN, 2021).

Casual convolutional layers, however, are computationally complex when long term time dependencies are significant, for the number of parameters grows too quickly with the number of inputs. Modern architectures solve this problem with dilated convolutional layers (OORD *et al.*, 2016; BAI *et al.*, 2018), which modify (2.8) to the following:

$$(\boldsymbol{W} * \boldsymbol{h})(l, t, d_l) = \sum_{\tau=0}^{\lfloor k/d_l \rfloor} \boldsymbol{W}(l, \tau) \boldsymbol{h}_{t-\tau}^{l}, \tag{2.9}$$

where d_l is the dilation rate of an specific layer. Dilated convolutions down sample information and reduce data resolution to incorporate information from distant past (LIM; ZOHREN, 2021). The higher the dilation rate, the greater the amount of information we can aggregate from earlier layers so that we can use distant history in a efficient fashion. This is the way Deep Mind's WaveNet architecture works (OORD *et al.*, 2016). Its usage of dilated convolution layers is shown in Figure 2.1a.

2.2.2 Recurrent Neural Networks

Recurrent Neural Networks have been used in sequence modeling with strong results for a variety of tasks (GOODFELLOW *et al.*, 2016). There are many RNN-based architectures that have been developed for temporal forecasting applications because RNN cells contain an internal memory state that acts as a summary of past information (LIM; ZOHREN, 2021). That memory is recursively updated at each time step, as illustrated in Figure 2.1b and in the following equation:

$$\boldsymbol{h}_t = \nu\left(\boldsymbol{h}_{t-1}, y_t, \boldsymbol{x}_t, \boldsymbol{s}\right), \tag{2.10}$$

where $\boldsymbol{h}_t$ is the hidden internal state of the network and ν is the memory update function. The main difference between CNNs and RNNs is that the latter does not require the specification of a lookback window (LIM; ZOHREN, 2021), allowing us to define that for every prediction.

2.2.2.1 Long Short Term Memory

As the lookback window can be infinite, RNNs may struggle to learn long range dependencies from data sets (BENGIO *et al.*, 1994; LIM; ZOHREN, 2021). Long Short Term Memory RNNs have been developed to address this issue (HOCHREITER; SCHMIDHUBER, 1997). They achieve so by using a cell state $\boldsymbol{c}_t$, which stores long term information, transformed with a series of gates as the following:

$$\text{Input gate: } \boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_{i_1} \boldsymbol{h}_{t-1} + \boldsymbol{W}_{i_2} y_t + \boldsymbol{W}_{i_3} \boldsymbol{x}_t + \boldsymbol{W}_{i_4} \boldsymbol{s} + \boldsymbol{b}_i\right), \tag{2.11}$$

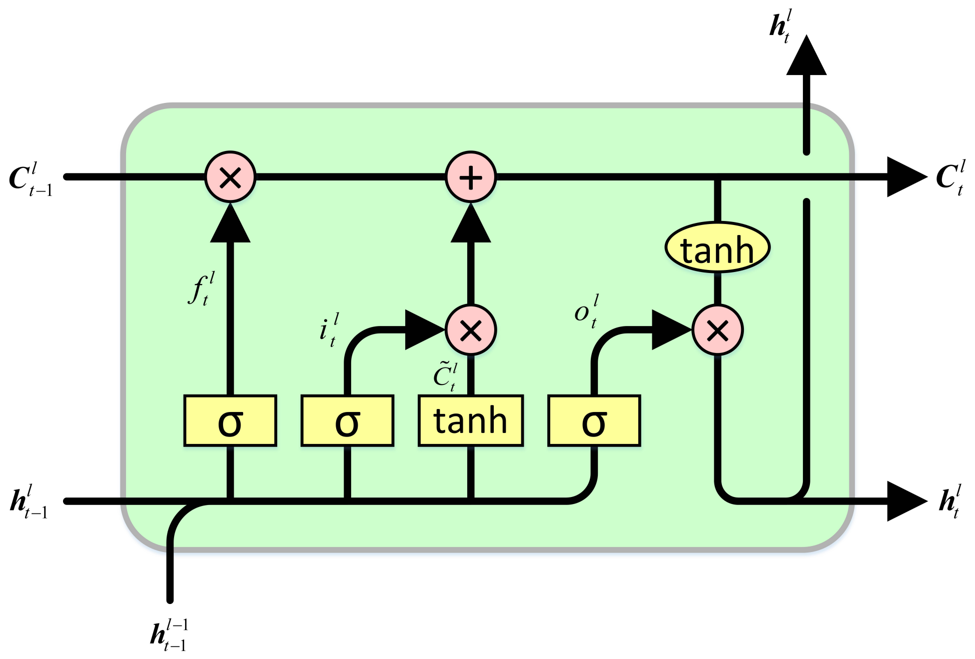


FIGURE 2.2 – Simplified representation of a LSTM cell. Notice that $\boldsymbol{s}$ and y are not shown for the sake of simplicity[1].

[1] Extracted from `https://stackoverflow.com/questions/50488427/what-is-the-architecture-behind-the-keras-lstm-cell`.

$$\text{Output gate: } \boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_{o_1}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{o_2}y_t + \boldsymbol{W}_{o_3}\boldsymbol{x}_t + \boldsymbol{W}_{o_4}\boldsymbol{s} + \boldsymbol{b}_o\right), \tag{2.12}$$

$$\text{Forget gate: } \boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_{f_1}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{f_2}y_t + \boldsymbol{W}_{f_3}\boldsymbol{x}_t + \boldsymbol{W}_{f_4}\boldsymbol{s} + \boldsymbol{b}_f\right), \tag{2.13}$$

where $\boldsymbol{W}$ is the linear weights of the network, $\boldsymbol{b}$, the biases, $\boldsymbol{h}_{t-1}$, the LSTM's hidden state and σ the sigmoid activation function. The gates in (2.11), (2.12) and (2.13) modify the LSTM's hidden and cell states as shown in (2.15) and (2.14) where $\odot$ is the element-wise product (LIM; ZOHREN, 2021). The basic structure of a LSTM cell is depicted in Figure 2.2. We have simplified the representation for the sake of simplicity, as $\boldsymbol{s}$ and y are optional parameters that depend upon the problem we are analyzing.

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tanh(\boldsymbol{W}_{c_1}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{c_2}y_t + \boldsymbol{W}_{c_3}\boldsymbol{x}_t + \boldsymbol{W}_{c_4}\boldsymbol{s} + \boldsymbol{b}_c), \tag{2.14}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t), \tag{2.15}$$

Equations (2.14) and (2.15) represent the output of a single LSTM cell. We can concatenate LSTM cells to build an encoder. In this case, the outputs $\boldsymbol{c}_t$ and $\boldsymbol{h}_t$ are fed into the next cell as $\boldsymbol{c}_{t-1}$ and $\boldsymbol{h}_{t-1}$. The complete latent state variable $\boldsymbol{z}$ is defined as a simple function of the encoder's hidden states (GOODFELLOW *et al.*, 2016).

2.2.2.2 Bidirectional LSTM

In many applications, we are interested in an output that depends on the whole input sequence. For instance, in speech recognition, the correct interpretation of a phoneme depend on past and future sounds (GOODFELLOW *et al.*, 2016). Bidirectional RNNs address this need. They combine an RNN structure that moves forward through time with another RNN that moves backward through time.

The output at each time unit for a bidirectional LSTM consists of the hidden state of both the forward and backward LSTM. They can benefit from a relevant summary information of the past (from the forward network) and from relevant information of the future (from the backward network) (GOODFELLOW *et al.*, 2016).

In special, the Bidirectional LSTM structure has shown strong performance in summarization tasks for both trajectory prediction (IVANOVIC; PAVONE, 2019) and natural language processing (BRITZ *et al.*, 2017).

2.2.3 Attention

Attention mechanisms brought improvements in long term dependency learning. Attention layers allow the network to focus on many time steps in the past, even if they are far away from the lookback windows. To achieve so, attention employs dynamically generated weights to aggregate temporal features (LIM; ZOHREN, 2021), as illustrated in Figure 2.1c. The mechanism is a key-value lookup based on a query (GRAVES *et al.*, 2014):

$$\boldsymbol{c}_t = \sum_{\tau=0}^{k} \alpha(\boldsymbol{\kappa}_t, \boldsymbol{q}_\tau)\boldsymbol{v}_{t-\tau}, \tag{2.16}$$

where the key $\boldsymbol{\kappa}_t$, query $\boldsymbol{q}_\tau$ and value $\boldsymbol{v}_{t-\tau}$ are features produced at different steps by earlier levels of the network. $\alpha(\boldsymbol{\kappa}_t, \boldsymbol{q}_\tau)$ is the attention weight for $t - \tau$ and generated at t. Lastly, $\boldsymbol{c}_t$ is the context output vector of the attention layer.

Networks with attention can directly attend to significant events that occur. This is a benefit in comparison with other methodologies, because it allows networks to adapt to seasonal variation of data. In retail forecasting, for instance, attention can predict the positive impact of holidays and promotional advertising on sales (LIM; ZOHREN, 2021). In addition, attention networks are also able to learn regime-specifics temporal patterns, as shown in Lim *et al.* (2020).

2.2.4 Additive Attention

The additive attention is often used as an aggregator for the encoder and decoder networks (CAPOBIANCO *et al.*, 2021). It achieved the best performance in a recent exploration of sequence-to-sequence modeling architectures in natural language processing (BRITZ *et al.*, 2017).

We can define the additive attention with the following equations (BAHDANAU *et al.*, 2015):

$$e_{t\tau} = \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_1}\boldsymbol{\kappa}_t + \boldsymbol{W_2}\boldsymbol{q}_\tau\right), \tag{2.17}$$

where $\boldsymbol{\kappa}_t$ is the attention key, $\boldsymbol{q}_\tau$ is the attention query and $e_{t\tau}$ represents the attention scores for the "quality of spatio-temporal relation" (CAPOBIANCO *et al.*, 2021) between the key at time t and the query at τ. Lastly, $\boldsymbol{W_1}$, $\boldsymbol{W_2}$ and $\boldsymbol{v}_a$ are trainable parameters.

We define the attention weights $\alpha_{t\tau}$ as the following:

$$\alpha_{t\tau} = \frac{\exp(e_{t\tau})}{\sum_{\tau'}^{T} \exp(e_{t\tau'})}, \tag{2.18}$$

where T is the size of the query sequence.

The context vector $\boldsymbol{c}_t$, which is the main output of the additive attention layer, is defined as:

$$\boldsymbol{c}_t = \sum_{\tau} \alpha_{t\tau}\boldsymbol{q}_\tau, \tag{2.19}$$

2.3 Probabilistic models

When forecasting time series, sometimes the predicted values do not carry all the information we need. In many cases, we can apply field knowledge to deep neural networks in order to improve our predictions (LIM; ZOHREN, 2021).

We can adapt Gaussian processes and linear state space models to be used in conjunction with neural networks. One remarkable example of such an adaptation are the Bayesian Neural networks (JOSPIN *et al.*, 2020).

In the context of time series forecasting, instead of modifying forecasting equations, probabilistic models use neural networks to generate parameters for predictive distributions (LIM; ZOHREN, 2021). Deep State Space Models (RANGAPURAM *et al.*, 2018), for example, perform inference using Kalman filter equations and encode time-varying parameters for linear state space models.

2.4 Adam Optimizer

Adam is an algorithm for “first-order gradient-based optimization of stochastic objective functions” (KINGMA; BA, 2015). It is the state-of-the-art method for optimizing the complex functions like those of neural networks.

Adam is an extension to stochastic gradient descent and is a successful replacement for such procedure to update the weights of a neural network. According to Kingma and Ba (2015), it has many benefits for non-convex optimization problems, such as little memory usage, well-suited for problems that are large in terms of data and parameters and appropriate for problems with noisy gradients.

The use of Adam requires us to set the following parameters: α, β_1, β_2 and ϵ. They are briefly described in the following itemization (KINGMA; BA, 2015).

- α is the learning rate, the rate at which we move our gradients.
- β_1 is the exponential decay rate for the first moment estimate.
- β_2 is the exponential decay rate for the second moment estimate.
- ϵ serves to prevent division by zero.

3 RoboCup SSL DataSet

During an SSL game, cameras installed above the field track the position of the robots and the ball. A software provided by the organizing committee processes the images and extracts data about the field and the objects inside it. In addition, a virtual referee manages the game and transmits data to the SSL server. All the aforementioned data composes the log of a game.

All SSL games are recorded and their data is publicly available at the RoboCup's website[1]. We can find log records available from 2016 to 2019, as games in 2020 and 2021 have been canceled due to the pandemics. Considering the four years of data available, we have logs from more than 200 games. Logs are encoded using Google's Protobuf library and are divided into two major types: referee packets and wrapper packets.

Referee packets contain data about the game's running time, the game's stage (penalty, extra time, half time, etc.), time left to finish the play, referee's commands (halt, stop, start, etc.), including a command counter and a timestamp. In addition, the referee packet includes information about teams: their name, color, yellow and red card counters, time out and which robot is the goalie.

The wrapper packet contains two nested classes: the `GeometryData` and the `DetectionFrame`. The `GeometryData` contains data about the field (lines, dimensions, boundaries) and about the camera calibration (focal length, principal point, etc.).

The `DetectionFrame` is the most important class for us. It contains the position and orientation of the robots and their position in the field's coordinate system. It also carries the robots' identification and height. Furthermore, the frame carries the ball's position, and its area. The robots' detection data is split in a vector for each team and all detections transmitted have an associated confidence.

In the next sections, we will explain how we conducted an analysis of the data set.

[1]`https://ssl.robocup.org/game-logs/`

3.1 Data cleansing

The data we received from the SSL server is not completely trustworthy because some detection packets are missing and others are repeated. Packets may be missing due to network delays or a faulty camera and repeated detection may happen if an object appears in the field of view of multiple cameras.

The detection rate is 60 Hz. It means that we are supposed to receive the next frame after 0.0166 seconds from the last one. We will consider the third decimal digit as an uncertainty because the time it takes for a computer to capture another image from the field is not deterministic.

After analyzing the data set, we found out the time range for two frames to be consecutive. If the difference between the time in seconds of capture between two frames is in the range $[0.01, 0.022]$, we consider them consecutive. When the difference is less than 0.01, we discard the newest frame. The aforementioned range stems from an analysis of the data set, by checking

However, whenever the difference is greater than 0.022, we might create a problem, because our time series would not have a constant time difference between elements. To solve this, we insert fake data into our stream, using intervals of 1/60 s, until the time difference between the last fake item and the detected one stays within our defined range.

To distinguish between fake data and real data, we created a new attribute for the detection packet: a boolean mask. It contains true for each real element and false for created ones. We leverage this mask to smooth the time series we are interested in analyzing. We employed the algorithm described in Barratt and Boyd (2020) to optimize the parameters of a Kalman smoother, which substitutes the fake points for a reliable interpolation that considers the objects' dynamics. In the next subsection, we are going to explain how we configured our Kalman smoother to perform such a task.

3.1.1 Kalman Smoother

Considering a linear system with dynamics

$$x_{t+1} = Ax_t + \omega_t, \tag{3.1}$$

with the following sensor measurements:

$$y_t = Cx_t + v_t, \tag{3.2}$$

where x_t is the state, ω_t is the process noise, y_t the sensor measurement, A is the state dynamics matrix and C is the output matrix (BARRATT; BOYD, 2020).

The Kalman filter is a method of estimating the current state of a dynamical system, given the observations so far. A Kalman smoother leverages the concepts behind the filter to refine estimates of previous states, in the light of later observations.

The optimization algorithm described in Barratt and Boyd (2020) fits the Kalman smoother parameters to a noisy sequence from a dynamic system as the one described above that may have missing measurements. Let W and V be the process noise and the sensor noise covariance matrices, respectively. The set of parameters θ for the Kalman smoother is the following, which we can optimize using the Boyd's algorithm.

$$\theta = \left(A, W^{-\frac{1}{2}}, C, V^{-\frac{1}{2}}\right). \tag{3.3}$$

Moreover, the algorithm's implementation also accepts a boolean vector, which serves as a mask to signal that a measurement at time t is unreliable and should be exchanged by a Kalman smoother prediction.

After running the optimization for a sequence of data, we can use the optimized parameters θ to smooth all our data set, as it contains a constant pattern of noise and unreliable measurements.

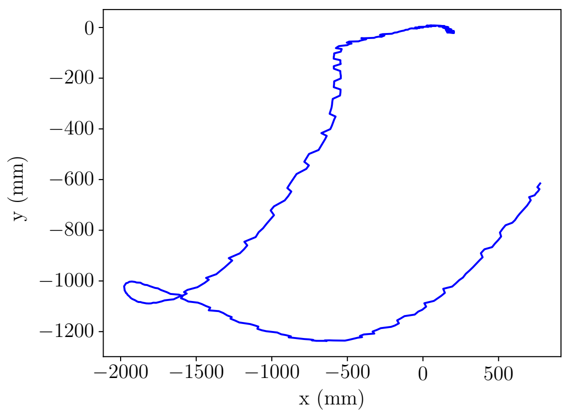
3.2 Robots' data

In this section, we are going to explain our treatment of robots' data so that our deep learning algorithm receives reliable data for training. For each robot we collected data of its position, velocity and orientation.

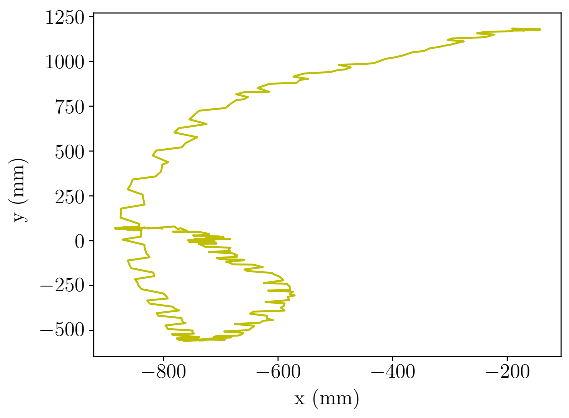
3.2.1 Robots' trajectory and velocity

From the `DetectionFrame` packet we can extract the position of each robot in the field at each frame. From the extracted positions, we can plot them in a graph to analyze how noisy the data is. We captured the positions of each robot between a play and a stop signal from the game controller (virtual referee). We will reference henceforth such time period as play time. This way, we avoid tracking robots team members displace during time out. In Figure 3.1, we plot two trajectories from robots of different teams during the same play time. These figures contain raw trajectories without any preprocessing.

From both images, we clearly notice that the data is very noisy. The SSL Vision (the Robocup program that processes cameras' images) transform coordinates in pixels to



(a) Trajectory from a blue team's robot during play time.



(b) Trajectory from a yellow team's robot during play time.

FIGURE 3.1 – Trajectories from two robots during game play.

coordinates in the field's system. This process introduces quantization noise: the robot's position oscillates between two adjacent pixels, as highlighted in Figure 3.2.

To bypass the quantization noise, we utilized the algorithm described in Section 3.1.1 to optimize the parameters for a Kalman smoother (KALMAN, 1960). We set the learning rate to 1×10^{-2} and the regularization parameter to 1×10^{-10}. After running the optimizer for 25 iterations in a single trajectory, we smoothed all the trajectories from our data set. The smoothed trajectories are depicted in Figure 3.3.

One concern about smoothing or filtering data is the introduction of delay, because it will decrease the effectiveness of our prediction algorithm. To assure we have reliable data, we plotted the x coordinate against the time frame in Figure 3.4. It shows no delay in comparison to the raw trajectory.

The advantage of using a Kalman to smooth data is that although our observations contain only the robot's x and y position, we can configure our state dynamics to track the speed in the x axis and in the y axis. After doing so, the kalman smoother returns the velocity of the robot, in addition to its position. The speed we obtained for both x and y for the trajectory depicted in Figure 3.3a is shown in Figure 3.5.

As described in Section 3.1, we interpolated all missing points from our measurements so that the interval between to data frames is about 0.0166 s. To streamline the development of a prediction algorithm, we chose to track the robot's speed in $mm/frame$ instead of mm/s.

3.2.2 Robots' heading

The robots' heading also has detection noise. It does not suffer from the quantization noise we detailed in Section 3.2.1, however the noise introduces a new problem for this case.

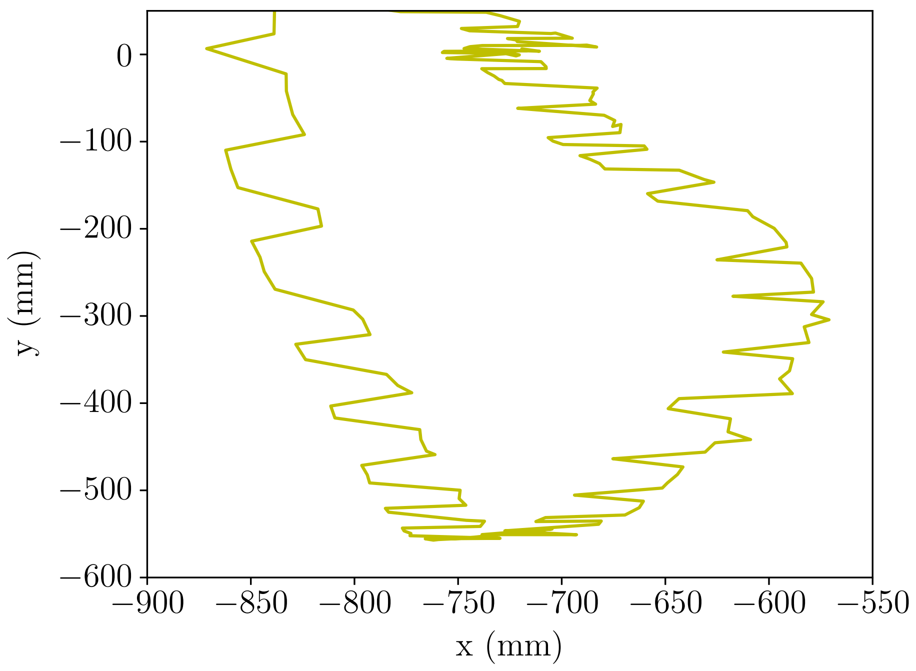
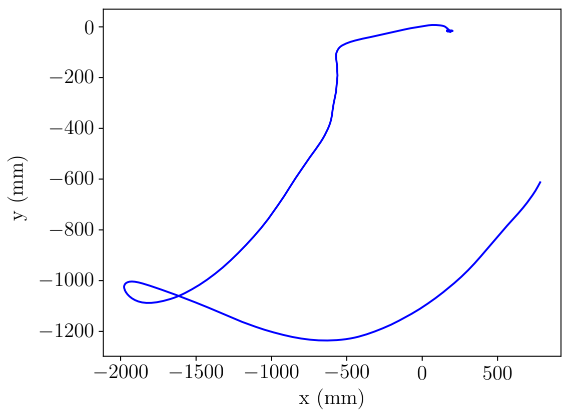
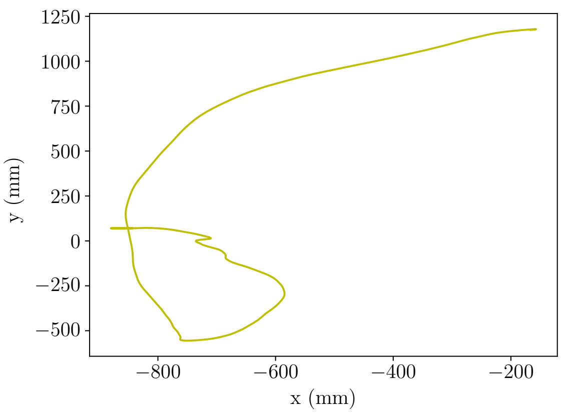


FIGURE 3.2 – Quantization noise highlighted from Figure 3.1b.





(a) Trajectory from the blue team's robot shown in Figure 3.1a after that we employed the Kalman smoother.

(b) Trajectory from yellow the team's robot shown in Figure 3.1b after that we employed the Kalman smoother.

FIGURE 3.3 – Trajectories from two robots during game play after that we smoothed them.

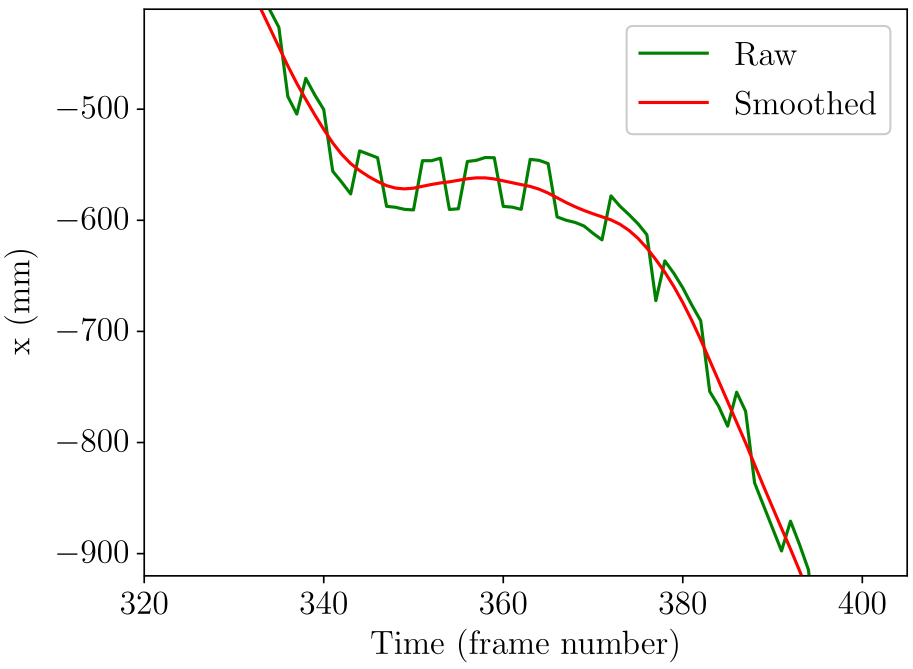
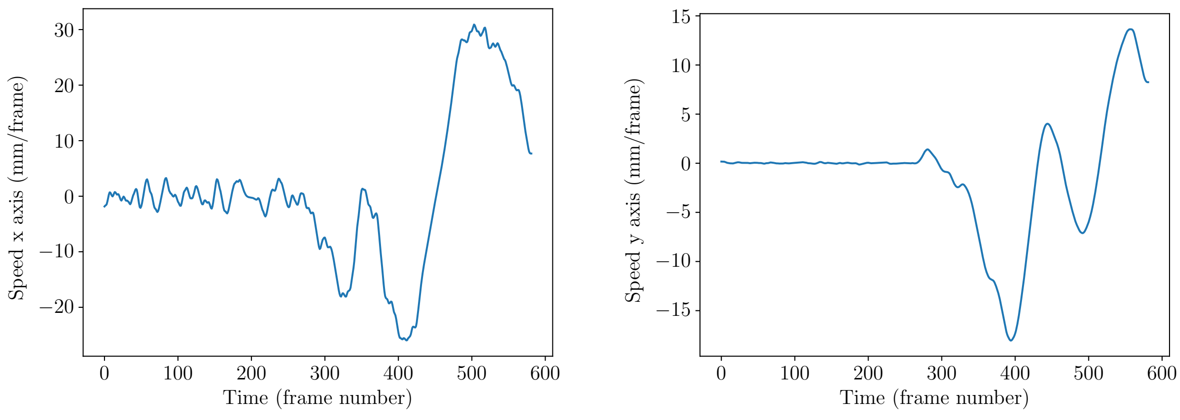


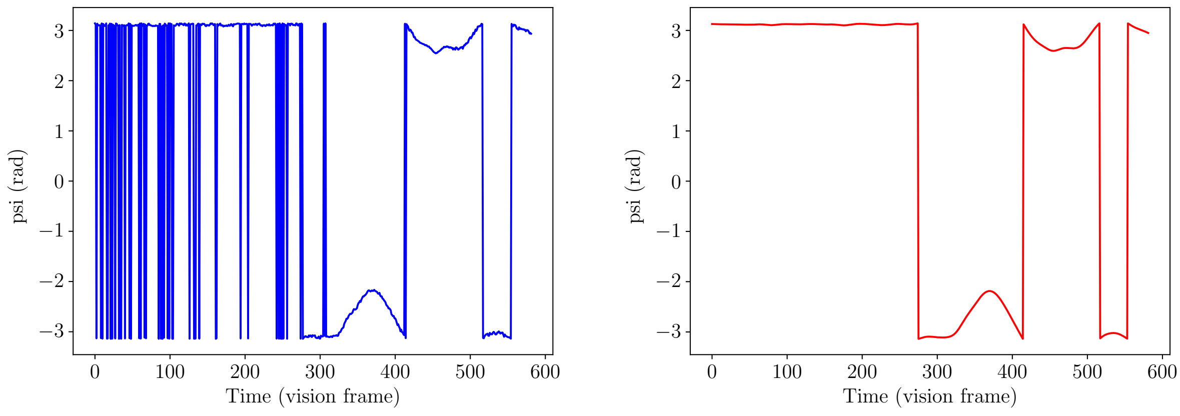
FIGURE 3.4 – Comparison of delay and advance in data from our filters.



(a) Speed in the x coordinate.

(b) Speed in the y coordinate.

FIGURE 3.5 – Speed in the x and y coordinates extracted from the blue team's robot, whose trajectory is shown in Figure 3.3a.



(a) Raw readings of a robot's heading ψ.

(b) Sequence of ψ values after applying the smoother.

FIGURE 3.6 – Comparison between a raw sequence of headings and a smoothed one.

We measure the heading in radians within the range $[0, 2\pi)$ and when the robot's heading is close to 0 radians we see a tremendous amount of discontinuity in the measurements as it keeps changing from 0 to the surroundings of 2π.

To overcome this problem, we calculated the sine and cosine of the heading and smoothed the data using a Kalman smoother, whose parameters we optimized using an optimization algorithm (BARRATT; BOYD, 2020) described in Subsection 3.1.1. We configured it with a regularization parameter of 1×10^{-10} and a learning rate of 1×10^{-4}. After running the algorithm for 25 iterations in a single sequence of headings, we had the parameters to smooth all sequences of sine and cosine we obtained from our data set. From the smoothed sine and cosine, we calculate the arc tangent to obtain the heading in radians again.

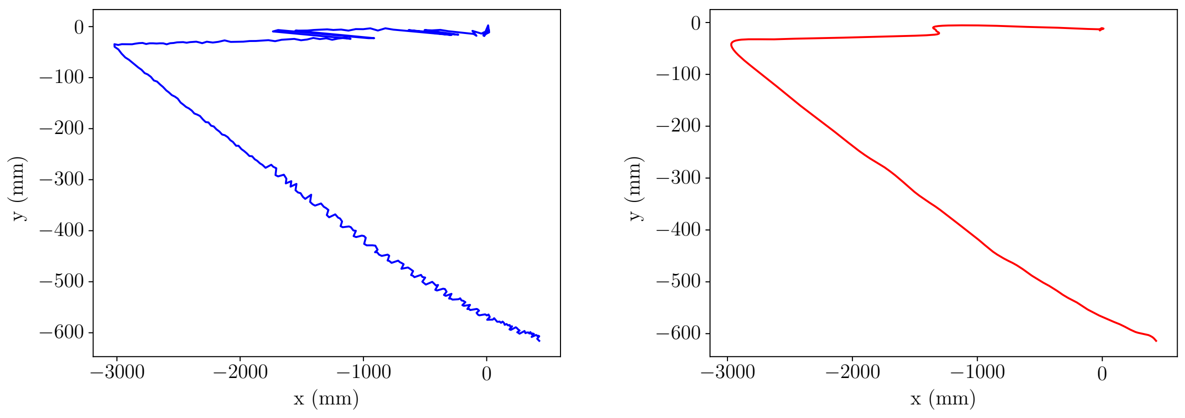
In Figure 3.6 we compare a raw trajectory with a smoothed one. We notice a clear decrease of noise and jumps between 0 and 2π.

3.3 Ball's data

The ball's data we collected from the log contains only the ball's position at each vision frame. The level of noise from the ball's data is higher than that of the robot's data. Besides having the quantization noise, the ball is smaller than a robot and much more difficult to detect, which introduces white noise to the measurements.

We employed again the algorithm described in Subsection 3.1.1 to optimize the parameters for smoothing our data set. We set a learning rate of 1×10^{-2} and a regularization parameter of 1×10^{-10}. We ran the algorithm for 25 iterations.

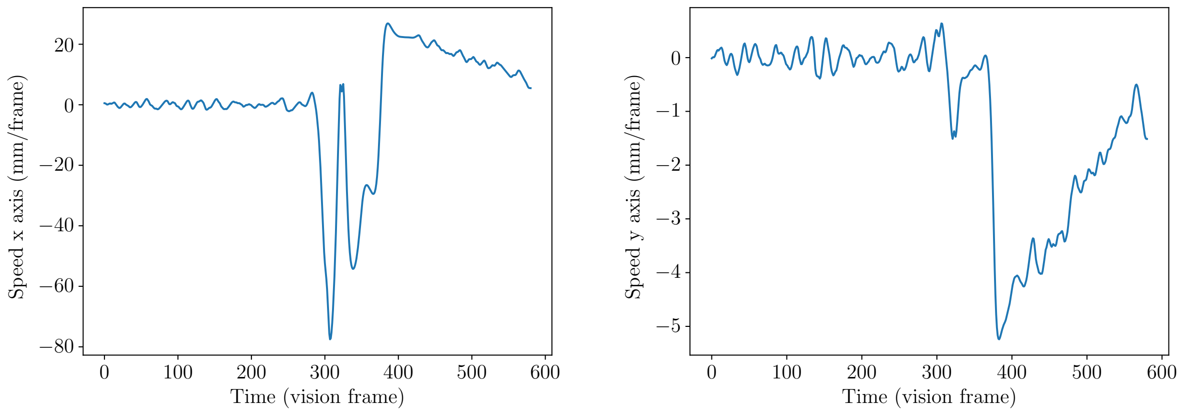
The Kalman smoother we employed serves to both interpolate unreliable points and



(a) Raw readings of the ball's position.

(b) Smoothed ball trajectory.

FIGURE 3.7 – Comparison between a raw sequence of ball positions and a smoothed one.



(a) Ball speed in the x axis.

(b) Ball speed in the y axis.

FIGURE 3.8 – Ball speed extracted from the Kalman smoother.

remove noise from the data. We configured the Kalman smoother state to calculate the ball's velocity given the measurements of position. In Figure 3.7 we show a comparison between the raw ball's trajectory and the smoothed one. The Kalman smoother is efficient for removing noise from the measurements. In addition, Figure 3.8 shows the calculated speed over time for the trajectories depicted in Figure 3.7.

4 Metodology

In this section, we explain how we built and tested a neural network to predict the future trajectory of robots. We describe in details the architecture, the hyperparameters, the inputs and outputs. Moreover, we explain how we assessed the network performance.

4.1 Neural network

The trajectory prediction problem has been explored in several papers (IVANOVIC; PAVONE, 2019; CAPOBIANCO *et al.*, 2021). The apparent advantage of employing a neural network instead of a Kalman predictor is that we can use metadata about the surroundings of the agent to predict its movements. For instance, in Capobianco *et al.* (2021), authors use the vessel's destination coordinates in the network's decoder to improve prediction. Likewise, in Ivanovic and Pavone (2019) researches model all the neighboring agents to predict the future position of a single target. We believe that this additional information provides more context for a neural network to surpass the performance of a traditional Kalman predictor.

We chose an encoder-decoder architecture for our network because it has shown outstanding performance for sequence to sequence neural networks (SUTSKEVER *et al.*, 2014). It is also the main architecture for mainstream trajectory prediction models such as Trajectron and Trajectron++ (IVANOVIC; PAVONE, 2019; SALZMANN *et al.*, 2020). Our neural network has been implemented using Tensorflow and Keras.

4.1.1 Architecture

An overview of the architecture we developed is shown in Figure 4.1. It has an encoder which summarizes information (BRITZ *et al.*, 2017) from the past into a fixed size tensor h_e and a decoder that predicts the future trajectory from the context vector z.

The decoder state initializer transforms LSTM hidden and cell states into initialization states for the decoder's LSTM cells. We also employed an additive attention mechanism to score the relationship between a summarization of the past and a prediction of the future (CAPOBIANCO *et al.*, 2021). The attention mechanism has also shown the best performance for such task, in comparison to other techniques (CAPOBIANCO *et al.*, 2021).

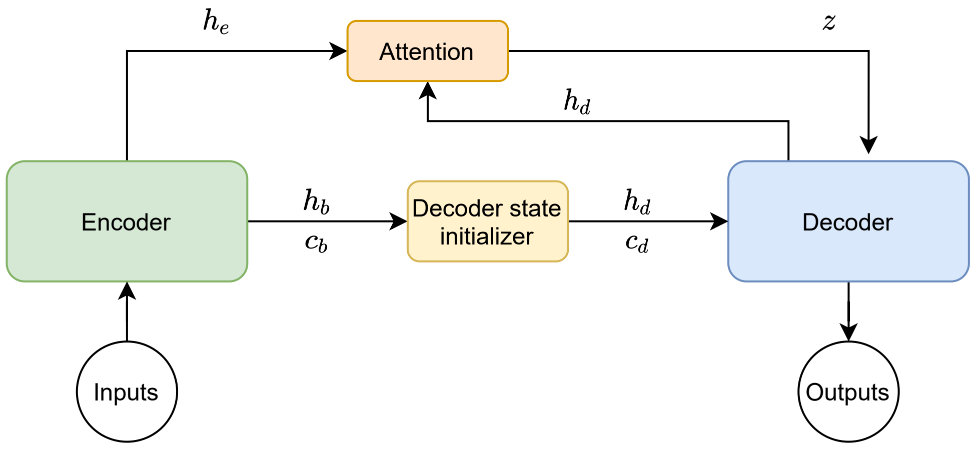


FIGURE 4.1 – Overview for the neural network we developed.

In the next subsections, we discuss the inputs and outputs of our model and the architecture of each component.

4.1.1.1 Inputs and outputs

To perform a time series prediction, we must define the size n of our look back window and the amount of time steps m in the future we aim at predicting.

Considering the data we captured from the robots and the pre-processing we made, following what we discussed in Section 3.2, we can define a measurement P_t at time t as the following:

$$P_t = \begin{bmatrix} x_t \\ y_y \\ \dot{x}_t \\ \dot{y}_t \\ \psi_t \end{bmatrix} \in \mathbb{R}^5, \tag{4.1}$$

where x_t and y_t represent the position of the robot at time t, $\dot{x}_t$ and $\dot{y}_t$ its velocity and ψ_t its heading at time t.

If we want to use a look back window of size n to predict m time steps in the future from t, the input is going to be the following:

$$X_r = \begin{bmatrix} P_{t-n-1} & \dots & P_{t-1} & P_t, \end{bmatrix} \in \mathbb{R}^{n \times 5}. \tag{4.2}$$

The neural network outputs a sequence of speeds the robot is going to have in the future m time steps. Let Q be the following:

$$Q_i = \begin{bmatrix} \dot{x}_i \\ \dot{y}_i \end{bmatrix} \in \mathbb{R}^2, \tag{4.3}$$

so the neural network prediction of m time steps ahead of t consists of the following:

$$Y = \begin{bmatrix} Q_{t+1} & Q_{t+2} & ... & Q_{t+m}, \end{bmatrix} \in \mathbb{R}^{m\times 2} \tag{4.4}$$

As the velocity we use is measured in millimeters per frame, we can use (4.5) and (4.6) to obtain the robot's future positions from the predicted velocity.

$$x_i = x_{i-1} + \dot{x}_{i-1}, \tag{4.5}$$

$$y_i = y_{i-1} + \dot{y}_{i-1}. \tag{4.6}$$

4.1.1.2 Encoder

Figure 4.2 depicts a simplified version of the model's encoder. As we have not defined the look back window n yet, the diagram shows an input of four time steps in the past.

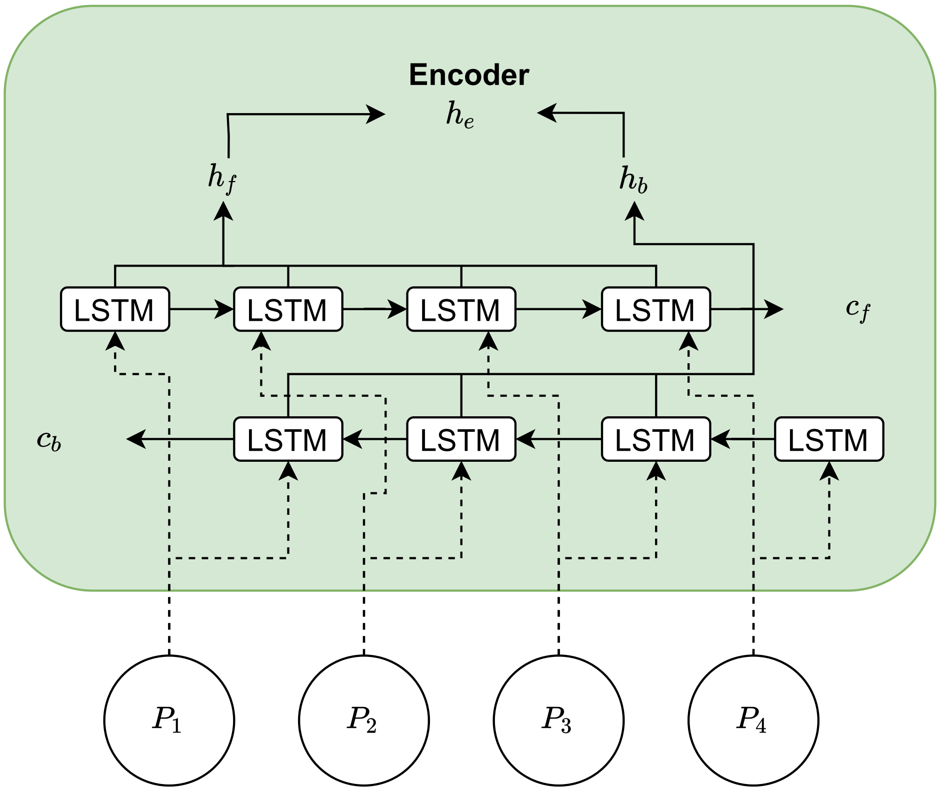


FIGURE 4.2 – Simplified overview of the model's encoder.

We use a BiLSTM architecture for the encoder, as it has shown great performance in summarization tasks (BRITZ *et al.*, 2017). The number of LSTM units (cells) for both the forward network and the backward network equals the size of the look back window.

In Figure 4.2, $\boldsymbol{c}_b \in \mathbb{R}^n$ and $\boldsymbol{c}_f \in \mathbb{R}^n$ represent the cell states for the backward and the forward network, respectively. We call $\boldsymbol{h}_b \in \mathbb{R}^n$ and $\boldsymbol{h}_f \in \mathbb{R}^n$ the hidden states of the backward and the forward networks, respectively. They are defined as the concatenation of the hidden state for each LSTM cell.

The encoded representation for the given input is $\boldsymbol{h}_e$, which is defined as follows:

$$\boldsymbol{h}_e = \begin{bmatrix} \boldsymbol{h}_f \\ \boldsymbol{h}_b \end{bmatrix} \in \mathbb{R}^{n \times 2} \tag{4.7}$$

4.1.1.3 State initializer

As demonstrated in Bahdanau *et al.* (2015), we apply a transformation to initialize our decoder. Our approach, however is slightly different than the one proposed in Bahdanau *et al.* (2015). We apply the following equations to create the initial states $\boldsymbol{h}_d \in \mathbb{R}^n$ (hidden state) and $\boldsymbol{c}_d \in \mathbb{R}^n$ (cell state) for our decoder.

$$\boldsymbol{h}_d = \tanh\left(\boldsymbol{W_1}\boldsymbol{h}_b + \boldsymbol{b}_1\right), \tag{4.8}$$

$$\boldsymbol{c}_d = \tanh\left(\boldsymbol{W_2}\boldsymbol{c}_b + \boldsymbol{b}_2\right), \tag{4.9}$$

where $\boldsymbol{W_1}$, $\boldsymbol{W_2}$, $\boldsymbol{b}_1$, and $\boldsymbol{b}_2$ are trainable parameters.

4.1.1.4 Attention aggregator

We use the additive attention mechanism as an aggregator for our encoder-decoder architecture. It is explained in details in Subsection 2.2.4. The inputs for our aggregator are $\boldsymbol{h}_e$ as the key and $\boldsymbol{h}_d$ for the query. It outputs a context vector $\boldsymbol{z} \in \mathbb{R}^n$.

We calculate the context vector $\boldsymbol{z}$ for each time step prediction, based on the decoder state $\boldsymbol{h}_d$ for the last prediction. We will refer henceforth to $\boldsymbol{z}$ as $\boldsymbol{z}_i$ as the context vector for the time step i.

The attention mechanism serves to score each context vector $\boldsymbol{z}_i$ with respect to the decoder's state $\boldsymbol{h}_d$ at time step i, while it is generating the output trajectory. It means we are "assigning a probability to each context of being attended by the decoder" (CAPOBIANCO *et al.*, 2021).

4.1.1.5 Decoder

We depict our decoder in Figure 4.3. The decoder has n LSTM cells, which are initialized with the output of the state initializer module (Subsubsection 4.1.1.3). The initialization states are depicted in the figure as $\boldsymbol{h}_{init}$ and $\boldsymbol{c}_{init}$.

The output of the forward LSTM network is $\boldsymbol{h}_d$, which we built as the concatenation of all cells' hidden state. The $\boldsymbol{h}_d$ is then fed into a multilayer perceptron network (MLP) with linear activation to calculate Q_i, so the MLP has two neurons. We concatenate Q_{i-1} with $\boldsymbol{z}_{i-1}$ to serve as the input for the decoder.

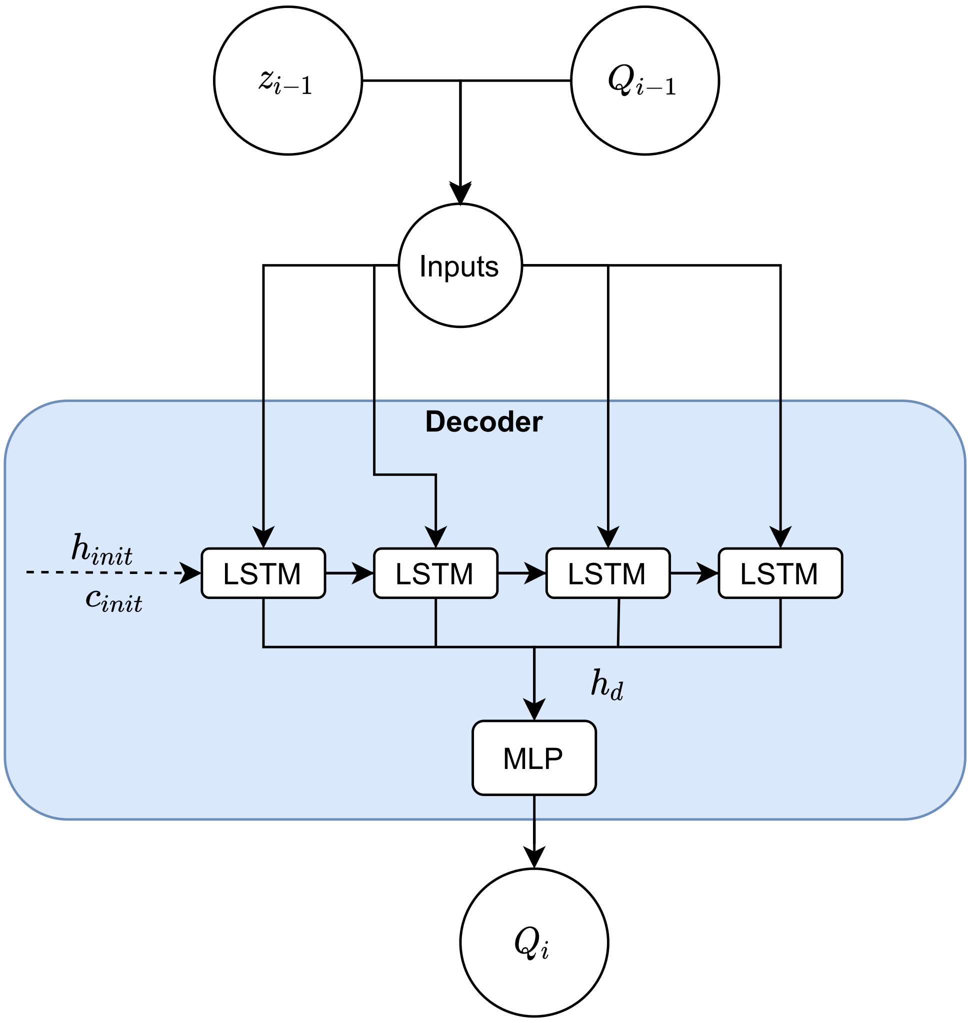


FIGURE 4.3 – Simplified overview of the model's decoder.

The decoder works in a recursive manner, as shown in Figure 4.5. The figure depicts the prediction of three time steps. To predict the first one, we feed the decoder with $\dot{x}_n$ and $\dot{y}_n$ from P_n. $\boldsymbol{z}_0$ represents the context vector obtained using the decoder's initialization states. For the time steps after the first one, we always utilize Q_{i-1} and $\boldsymbol{z}_{i-1}$ (calculated from $\boldsymbol{h}_d$ at time step i).

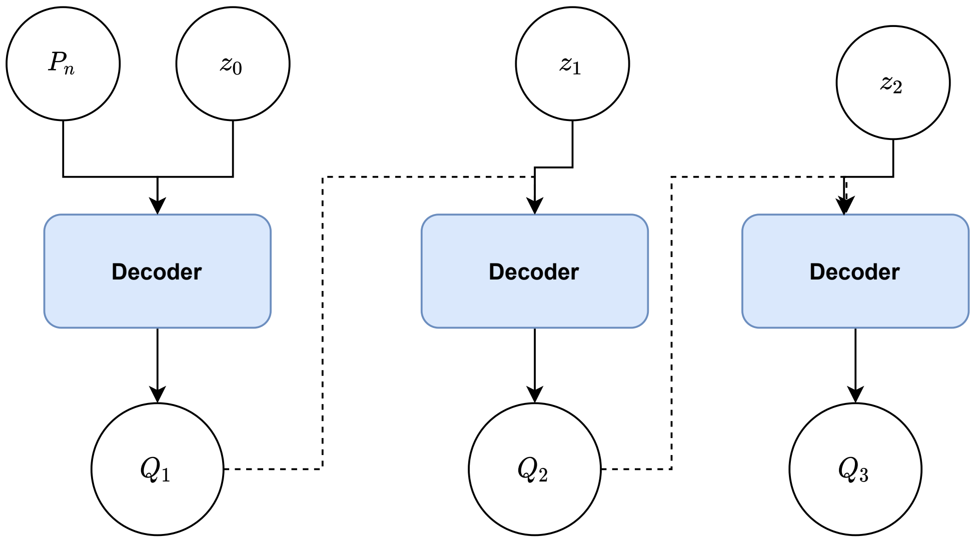


FIGURE 4.4 – Diagram of the decoder recursive prediction mechanism.

4.1.2 Using ball data

We sought to utilize the data we have about the ball during game time to improve our prediction algorithm. The ball influences the movement of the robots as they try to chase it. So we propose a second encoder-decoder architecture that accounts for the ball information as well. We define a look back window for the ball with the same size n as that of the robot. Then, a single measurement of the ball consists of the following:

$$B_t = \begin{bmatrix} x_t \\ y_t \\ \dot{x}_t \\ \dot{y}_t \end{bmatrix}, \tag{4.10}$$

where x_t and y_t are the ball's coordinates at time t. $\dot{x}_t$ and $\dot{y}_t$ are the ball's speed at the x and y axis, respectively. So a look back window for the ball is the following:

$$X_b = \begin{bmatrix} B_{t-n-1} & ... & B_{t-1} & B_t \end{bmatrix}, \tag{4.11}$$

To process the ball's data, we employ an encoder like the one described in Subsubsection 4.1.1.2, containing n LSTM cells. Equation (4.11) represents the input of this decoder. The encoder summarizes the ball information into a fixed length vector.

We synchronized the ball's data with the robot's data using the time of capture from the cameras, to avoid using sequences which are displaced in time to each other. However, for some cases we did not have enough measurements to generate a look back window of size n for the ball. The ball's encoder, then, receives as input a boolean mask as well, which informs the encoder what measurements it should attend to. Items marked as false are not considered for the encoding processes. The boolean mask has the following format:

$$X_m = \begin{bmatrix} w_{t-n-1} & ... & w_{t-1} & w_t \end{bmatrix}, \tag{4.12}$$

where w_t represents whether the measurement B_t should be considered for encoding and $w_t \in \{0, 1\}$.

The encoded information $\boldsymbol{h}_{eb}$ is then fed into an additive attention module, alongside the encoded information about the robot's past motion $\boldsymbol{h}_e$. Using the notation described in Subsection 2.2.4, $\boldsymbol{h}_{eb}$ is the query and $\boldsymbol{h}_e$ is the key. This module outputs a context vector $\boldsymbol{z}_b \in \mathbb{R}^n$. The additive attention has been in used in the research literature to merge data from more than two agents in trajectory prediction (IVANOVIC; PAVONE, 2019; SALZMANN *et al.*, 2020).

The attention aggregator serves to measure the quality of the relationship between the summarized information from the ball and the robot. We perform a linear transform to decrease the dimension of the context vector, as the following:

$$\boldsymbol{s}_b = \boldsymbol{W}\boldsymbol{z}_b + \boldsymbol{\omega}, \tag{4.13}$$

where $\boldsymbol{s}_b$ is the final processed ball information, and $\boldsymbol{W}$ and $\boldsymbol{\omega}$ are trainable parameters.

An overview of how we merged the ball's data into the main neural network pipeline is shown below.

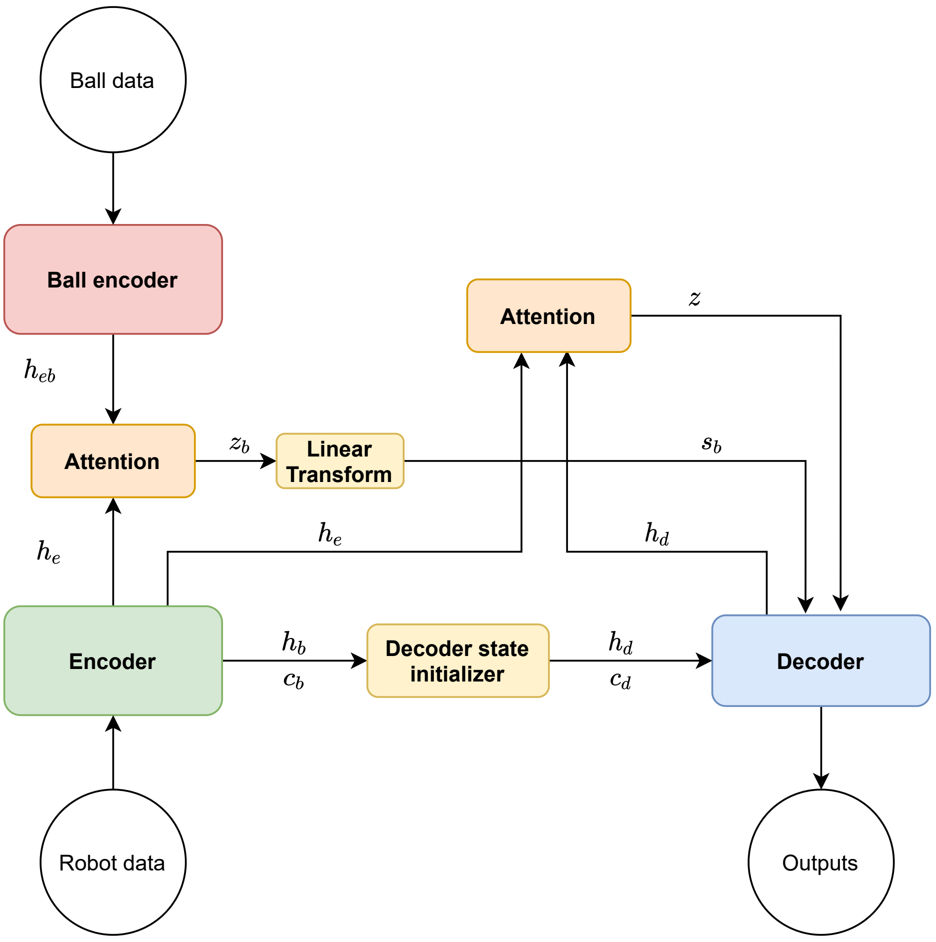


FIGURE 4.5 – Diagram of the whole network with an encoder for the ball.

The decoder for this new architecture is the same as the one described in Section 4.1.1.5, however the inputs are now the concatenation of Q_{i-1}, $\boldsymbol{z}_{i-1}$ and $\boldsymbol{s}_b$ to predict Q_i.

We opted for dimensional decrease in the size of $\boldsymbol{s}_b$, because we wanted to reduce its influence in the decoder. As the decoder's input is the concatenation of three variables, the greatest one will influence the results the most. Our tests have shown that s_b with 4/15 the size of $\boldsymbol{z}_b$ performs reasonably.

We will refer to this architecture henceforth as double-encoder architecture. The architecture in which we only pass the robots' data as inputs is the single-encoder architecture.

4.2 Dataset

We collected the logs from three RoboCup games. We used two of them to build our training set and one for our testing set. We also collected 10% of the samples from our training set to use as validation data. We randomized the data to avoid bias during the training. In terms of percentages the training set contains 78.03% of all data, the validation set has 8.67% and the testing set, 13.2%.

The exact quantity of elements we utilize for training and testing depends on the size of the look back window n and the number of time steps m we want to predict. However, the distribution of elements stay within the aforementioned percentages. To give an estimate about the exact number we have in our dataset, we can say that for $n = 30$ and $m = 15$, we have a total of $325,504$ elements. For $n = 60$ and $m = 30$, we have $306,739$. In Section 4.4 we explain the experiments for choosing the best n and m.

To create an element, we fragment each measurement array for a single robot (or the ball) into all possible subarrays of size $n + m$. Then, we break it into two arrays of size n and m. The former serves as input and the latter, as the output.

Moreover, we normalize the data to be fed into the neural network so that it has zero mean and unit variance. The outputs should be denormalized to acquire meaningful value.

4.3 Training procedure

We trained the neural network for 10 epochs with batches of 2048 elements. We employed Adam as the optimizer. For our tests we used the standard parameters for the optimizer (KINGMA; BA, 2015), namely $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1 \times 10^{-7}$.

When we were defining the parameters of our neural network (look back window size n and number of time steps prediction m), we used a constant learning rate of 0.001. After we found the best parameters for our purposes, we tuned the training procedure using a learning rate decay that follows the equation:

$$L_i = L_0 r^{\frac{s_i}{S}}, \tag{4.14}$$

where L_i is the actual learning rate at decay step i, L_0 is the initial learning rate, s_i is the current decay step, r is the decay rate and S is the number of decay steps. A new decay step happens whenever we call the optimizer (at the end of each batch). This is the implementation of Kera's `ExponentialDecay` function.

Our loss function $\mathbb{L}$ for training is the sum of the mean squared error and the mean absolute error. We multiply this sum by a factor of 100 to avoid working with small numbers for loss, which have shown to slow down training (PASCANU *et al.*, 2013). Our loss function is, then, described as follows:

$$\mathbb{L}(Y_{true}, Y_{pred}) = \frac{100}{K}\left[\sum_{k}^{K}\left(Y_{true}^{k} - Y_{pred}^{k}\right)^2 + \sum_{k}^{K}|Y_{true}^{k} - Y_{pred}^{k}|\right], \tag{4.15}$$

where K is the number of samples in our set, Y_{true} represent real values and Y_{pred} predicted ones. The superscript k indicates which sample we are analyzing.

4.4 Tests

In this section, we explain our metrics to evaluate the neural networks, how we determined the n and m parameters and how we conducted an experiment to compare our proposed architecture to a Kalman predictor and a fully-connected neural network.

4.4.1 Metrics

The properly assess the effectiveness of our algorithm, we chose the mean absolute error (MAE) and two other metrics that have a more practical meaning in the context of trajectory prediction: average distance (AVD) and average final displacement (AFD).

The mean absolute error is calculated as follows:

$$\text{MAE} = \sum_{k}^{K} |Y_{true}^{k} - Y_{pred}^{k}|, \tag{4.16}$$

where K is the number of samples in our set, Y_{true} represent real values and Y_{pred} predicted ones. The superscript k indicates which sample we are analyzing.

The average distance is the sum of the euclidean distance between a predicted position and a real one. Let the following be the euclidean distance:

$$d = \sqrt{(x_{true} - x_{pred})^2 + (y_{true} - y_{pred})^2}, \tag{4.17}$$

where x and y represent the object's position. The subscripts *true* and *pred* represent the real position and the predicted one, respectively.

Given the prediction containing m positions in future time steps, we sum the euclidean distance over the predicted positions:

$$d_{sum} = \sum_{i=1}^{m} d_i \tag{4.18}$$

where d_i is the euclidean distance calculated in (4.17) for a prediction at a future time step i.

In Figure 4.6, we illustrate a true trajectory and a predicted trajectory diverging from the last received measurement. The sum in (4.18) for the shown trajectory is the sum

$d_1 + d_2 + d_3$. The AVD is then the average between (4.18) for all samples.

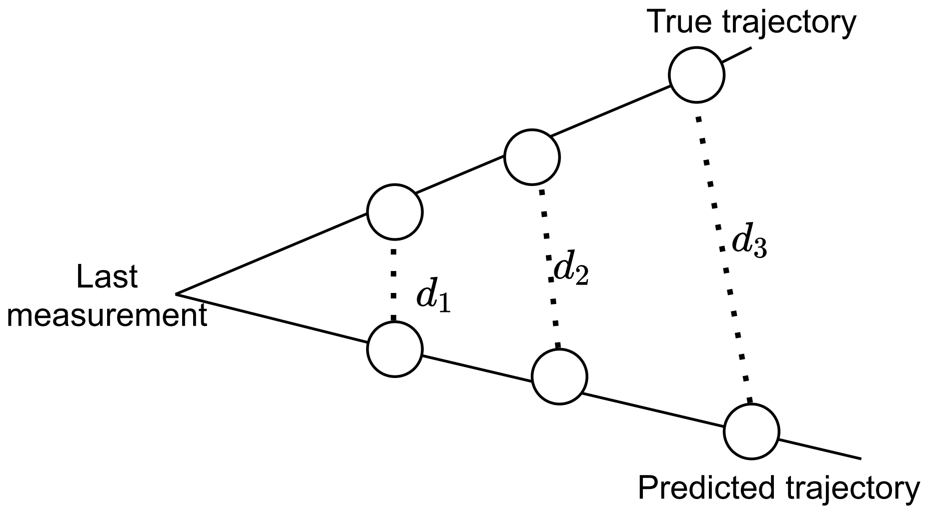


FIGURE 4.6 – Illustration to better visualize the AVD and AFD calculation.

The average final displacement is the average for all samples of the euclidean distance (4.17) calculated only for the last predicted time step m. Using the example in Figure 4.6, the final displacement is only the distance d_3.

The neural network output Y consists of a sequence of Q_i, which contains only speed. To calculate all the aforementioned metrics, we denormalize the output and convert all Q_i into positions, using (4.5) and (4.6).

The AVD and AFD are very meaningful ways to assess the effectiveness of our models. The AVD measures the accumulated displacement from the ground truth trajectory and AFD measures the displacement between the real last position and the predicted one. They provide us a concrete way to measure how bad we are predicting the trajectories. As our inputs are in millimeters, AVD and AFD also represents values in millimeters.

4.4.2 Look back window and prediction time steps

The frame rate of detection in the SSL server is 60 Hz. It means that we receive 60 frames per second. We decided to test three values of look back window n for our architecture: 30, 60 and 120, as they represent half a second, an entire second and two seconds.

Then we must chose the number of time steps m that favors our algorithm the most. We assume that n must be greater than m, otherwise we would not have enough information about the past to make an accurate prediction. We are deciding between $n = m$, $n = 2m$ and $2n = 3m$, after selecting the right n.

For this test we are not interested in exact numbers for the MAE, AVD and AFD, as we cannot directly compare the error between different values of m. The error for a prediction of more time steps will naturally have more error because we add more time steps.

We are evaluating, thus, the ratio between the MAE, AVD and AFD for the network prediction and for the Kalman prediction. This shows us how much each configuration outperforms the Kalman predictor. The ratio we are calculating is the following:

$$R_m = \frac{M_{nn}}{M_{kp}}, \tag{4.19}$$

where R_m is the ratio for a certain metric (either MAE, AVD or AFD), M_{nn} is the calculated metric for the neural network and M_{kp} is the metric for the Kalman predictor.

4.4.3 Analysis of the trained network

After selecting the right n and m, we trained the neural networks with a learning rate decay set up in the decay formula (4.14). This should improve the performance of our neural network.

We set up a Kalman predictor to compare with our neural network prediction. The Kalman predictor receives as input raw measurements from the robots' positions and updates its state matrices. The predictor has been initialized with the parameters we got from the optimizer described in Subsection 3.1.1. It first updates its state with the first n measurements for each trajectory. Then, after each update, we use its the state dynamics matrix to forecast m time steps in the future, given the last valid measurement. We calculate the MAE, AVD and ADF from the results using the smoothed trajectories as the ground truth. The comparison with a Kalman predictor is important because it is the main method SSL teams use to forecast trajectories.

To further improve our testing we created a basic multilayer perceptron network with four layers. Every layer has a ReLu activation function, except the last, which has no activation. The layers have 128, 1024, 128 and $2m$ neurons, respectively. The last layer predicts Q for each of the m future time steps.

We also analyze the influence of adding the ball data to the network comparing the performance of the single encoder network and the double encoder network.

To perform inference in the context of a Robocup game, the neural network must run in real time, so we also analyze the execution time of each proposed architecture, calculating it for 100 inferences and getting the average. We discarded the first inference time before starting to count the time, as the first involves loading dynamic libraries.

We have also done a qualitative analysis of the predicted trajectory, i.e. how well it predict sharp turns, how noisy the prediction is when looking at it and how trustworthy the results are.

5 Results and discussions

In this chapter, we will analyze the results of our tests and discuss them. We discuss the lookback window size n, number of prediction time steps m, the efficiency of adding ball data to the network and do an overall comparison between the presented architecture and a Kalman predictor.

5.1 Look back window and prediction time steps

At first we compared the ratio of MAE, AVD and AFD for the neural network and a Kalman filter. The ratio is the results obtained for the network divided by those of the Kalman predictor. We do this comparison for the following pairs of (n, m): $(30, 15)$, $(60, 30)$ and $(120, 60)$ to identify the best n for our neural network. All networks here have been trained with a learning rate of 0.001. We present results in Table 5.1.

Table 5.1 clearly shows that choosing $(n, m) = (120, 60)$ is not feasible for predicting the trajectory, as it performs worse than a Kalman predictor. Having found that, we analyzed the best relationship between n and m. Again, we are comparing the ratios of MAE, AVD and AFD, as the error is naturally larger for greater m. Table 5.2 shows the results for the ratios $n = m$, $n = 2m$ and $2n = 3m$ for n equals 30 and 60.

Table 5.2 shows that for $n = 30$, the ratio between n and m does not affect the results so much, however as the pair $(n, m) = (30, 15)$ most outperforms the Kalman predictor, we proceed to further analyze this pair. For $n = 60$, the ratio $n = 2m$ is clearly better than the others, so our analysis will consider only this ratio as well for $n = 60$.

TABLE 5.1 – Pairs (n, m) and their ratio of MAE, AVD and AFD with respect to the Kalman predictor.

Ratio	**(30, 15)**	**(60, 30)**	**(120, 60)**
MAE	0.542	0.726	1.150
AVD	0.542	0.725	1.148
AFD	0.507	0.817	1.252

TABLE 5.2 – Pairs (n, m) and their ratio of MAE, AVD and AFD with respect to the Kalman predictor.

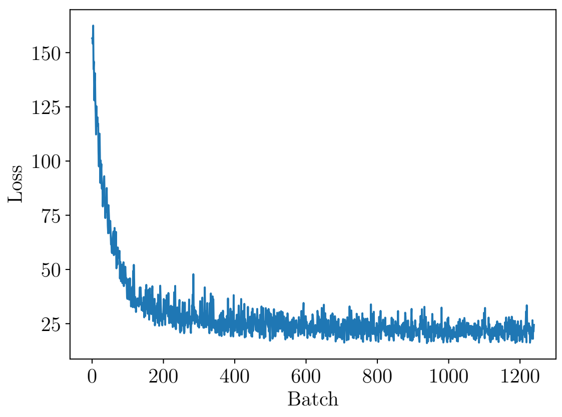
Ratio	**(30, 15)**	**(30, 20)**	**(30, 30)**	**(60, 30)**	**(60, 40)**	**(60, 60)**
MAE	0.542	0.548	0.562	0.726	0.856	0.924
AVD	0.542	0.550	0.561	0.725	0.851	0.923
AFD	0.507	0.574	0.655	0.866	0.941	0.981

5.2 Analysis of the trained networks

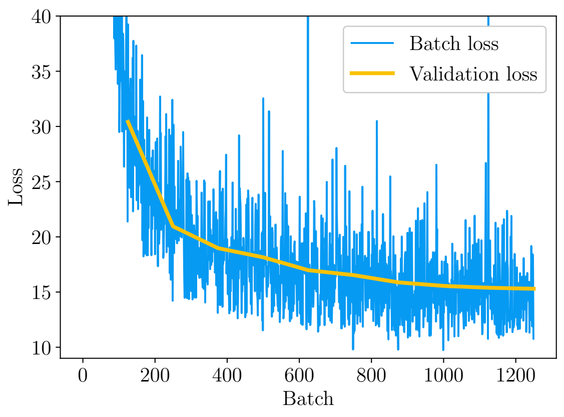
As soon as we had the pairs (n, m) that outperforms the most the Kalman filter, we decided to tune the training adding a learning rate decay, according to (4.14).

For the pair $(n, m) = (30, 15)$, the single encoder network yielded better training results with 1000 decay steps and 0.98 as the decay rate. For the double encoder network, the same configuration has been used. Considering $(n, m) = (60, 30)$, we used 1000 decay steps and 0.98 as the decay rate for the single encoder network and 10,000 decays steps and 0.99 as the decay rate for the double encoder network. The initial learning rate has been 0.001 for all cases.

To make sure the training indeed decreased the loss function, we show below the plots we got from the batch loss and the validation loss during training. They confirm the neural networks converged to a minimum. Figures 5.1, 5.2, 5.3 and 5.4 also shows us that we avoided overfitting because the last validation loss is close to the last obtained batch loss. It is worth to mention that, although the batch loss is calculated at the end of each batch, the validation loss is calculated at the end of the epoch, so we show in the plots the validation loss at the last batch of each epoch. For $n = 60$, there are 117 batches, meanwhile for $n = 30$, there are 125 batches.

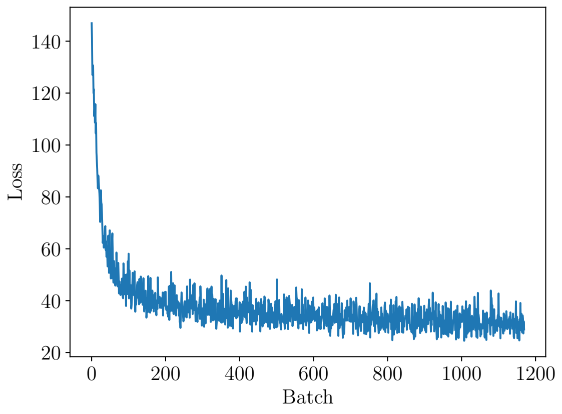


(a) Batch loss during training.

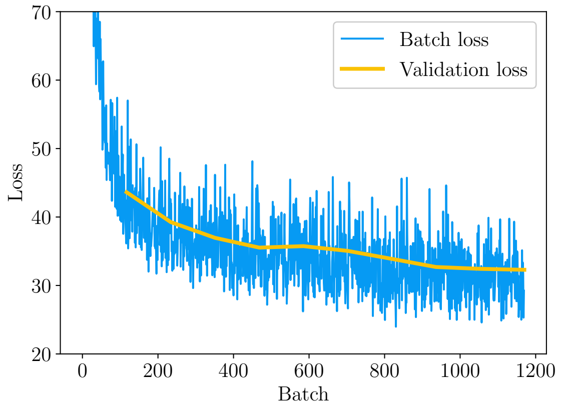


(b) Validation loss during training, plotted over the batch loss.

FIGURE 5.1 – Losses during training for the single encoder network using $(n, m) = (30, 15)$.

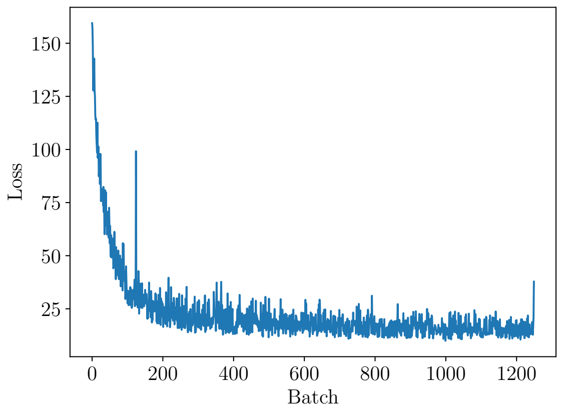


(a) Batch loss during training.

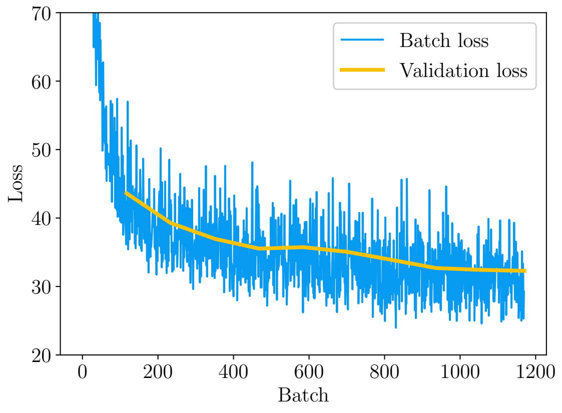


(b) Validation loss during training, plotted over the batch loss.

FIGURE 5.2 – Losses during training for the single encoder network using $(n, m) = (60, 30)$.

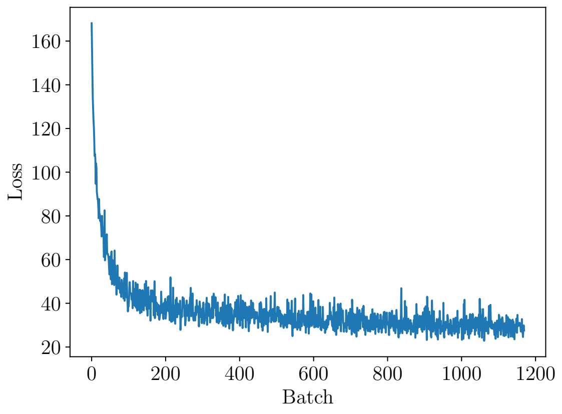


(a) Batch loss during training.

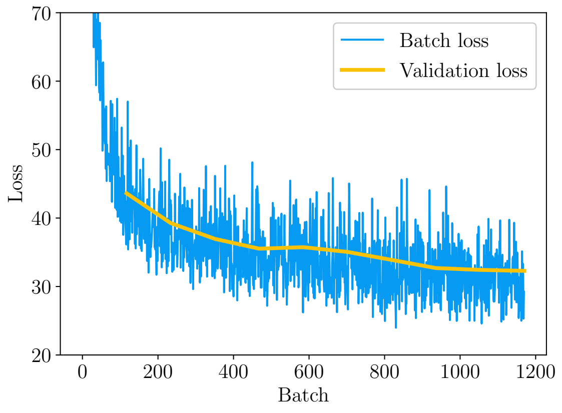


(b) Validation loss during training, plotted over the batch loss.

FIGURE 5.3 – Losses during training for the double encoder network using $(n, m) = (30, 15)$.



(a) Batch loss during training.



(b) Validation loss during training, plotted over the batch loss.

FIGURE 5.4 – Losses during training for the double encoder network using $(n, m) = (60, 30)$.

5.3 Comparison with other forecasting methods

We compared our proposed architecture with other standard forecasting techniques. The Kalman prediction is the standard method for predicting trajectories in the robotics field, so it is our guidance for benchmarks. As we explained in Subsection 4.4.3, we created a simple multilayer perceptron network to highlight that our method is indeed effective and necessary for a trustworthy trajectory forecast. We show the results we got of MAE, AVD and AFD in Tables 5.3 and 5.4.

TABLE 5.3 – Comparison of several between our architectures and other forecasting methods for $(n, m) = (30, 15)$.

Metric	**Single encoder**	**Double encoder**	**Kalman Predictor**	**MLP**
MAE	64.63	59.38	131.50	182.95
AVD	51.73	47.42	103.98	146.23
AFD	5.99	5.58	13.96	16.42

From Table 5.3, we notice that an MLP struggles to learn the right representation for forecasting trajectories. It reinforces that a specific neural network architecture is needed to predict future trajectories.

Both of our proposed models outperform the Kalman predictor. Compared to the Kalman predictor results, the single encoder network decreases the MAE and AVD error in 51% and the AFD in 57.1%. The double encoder network performs slightly better. It reduces MAE and AVD 54.5% with respect to those of the Kalman predictor and decreased the AFD in 60.1%.

TABLE 5.4 – Comparison of several between our architectures and other forecasting methods for $(n, m) = (60, 30)$.

Metric	**Single encoder**	**Double encoder**	**Kalman Predictor**	**MLP**
MAE	351.43	374.64	598.60	890.24
AVD	277.65	296.90	474.95	709.92
AFD	24.29	25.48	35.54	54.47

From Table 5.4, we notice again that a MLP network is not suitable for trajectory prediction as it cannot beat the performance of a Kalman predictor. For the $(n, m) = (60, 30)$, both architectures outperform the Kalman predictor, but not quite so well as in $(n, m) = (30, 15)$. The single encoder network decreases the MAE, AVD and AFD in 41.3%, 41.6% and 31.7% with respect to Kalman predictor. The double encoder network performs worse here. It reduces the MAE, AVD and AFD with respect to the Kalman predictor in 37.5%, 37.5% and 28.4%, respectively.

We can infer from the results that the MLP network is unsuitable for trajectory forecasting for it performs worse than a Kalman predictor. Our proposed architecture indeed has great results and can be up to 54.1% better than a Kalman predictor for $(n, m) = (30, 15)$ in terms of average final distance (AVD) and up to 41.6% better for $(n, m) = (60, 30)$. It is worth mentioning that, even though the Kalman predictor shows such low scores, it is much more cheaper in terms of computational complexity.

Adding a new encoder to the architecture makes it perform slightly better for the look back window of 30. Indeed, it decreased the AVD in 8.3%. However, for a look back window of 60, the new encoder harms the results. The AVD increased 6.9%, showing us that the second encoder might not be so useful for us.

5.4 Inference time

We calculated the inference time for our networks as described in Section 4.4.3. We wanted to determine if it is possible to run the models in real time during a SSL game. Results are shown in Table 5.5. We used an Intel Core i7-7500U 2.70 GHz to measure inference time with Tensorflow compiled with vector instructions.

TABLE 5.5 – Inference time in milliseconds for our proposed architectures.

(n, m)	**Single encoder**	**Double encoder**
(30, 15)	28.2 ms	32.9 ms
(60, 30)	31.0 ms	37.0 ms

The inference times we show in Table 5.5 are the inference for a single robot. Nonetheless, if we predict trajectories in batches of 11 (maximum number of robots in each SSL team), the inference time does not change considerably, possibly because modern CPUs have SIMD instructions, which our deep learning framework utilizes.

Ideally, we would expect that the inference to consume less than 16.6 ms for it is the time between two SSL frames and it is the time we expect the system's decision making to run and send new commands to robots. The focus of this work is to elaborate a reliable prediction mechanism, so we were not interested in achieving better execution times now. In addition, there are many techniques known to decrease execution time, such as model quantization and pruning. Some specialized deep learning frameworks also help make inference faster.

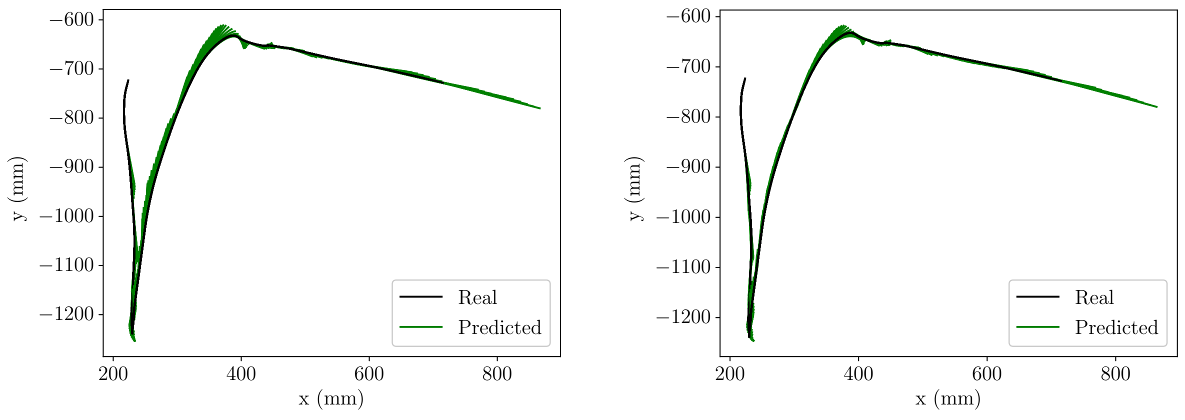
In comparing both proposed models, we can say that the double encode network has a greater inference time, but only a slightly better results compared to the single encoder. If we consider the look back window of 60, it is both slower and has worse results.

5.5 Qualitative analysis of results

SSL robots have a diameter of 18 cm. The mean average final displacement between the real trajectory and the predicted one is less than 6mm for a prediction of 15 future time steps. It is a remarkable result, considering the precision of our algorithm. Regarding the prediction of 60 future time steps, the AFD is less than 26 mm. The result is higher because, as we forecast more time steps into the future, the more our results tend to diverge. However, an error of 26 mm for an object of 18 cm is very precise.

We present in Figures 5.5, 5.6 and 5.7 a comparison between the single encoder network and the double encoder for a look back window of 30. Figure 5.5 shows the real trajectory in black and all the possible predictions in green, using only a 30 frame look back window. Overall, the prediction from both networks appear very similar, however we notice that Figure 5.5b has less green bumps and the turn at the top is closer to its real value.

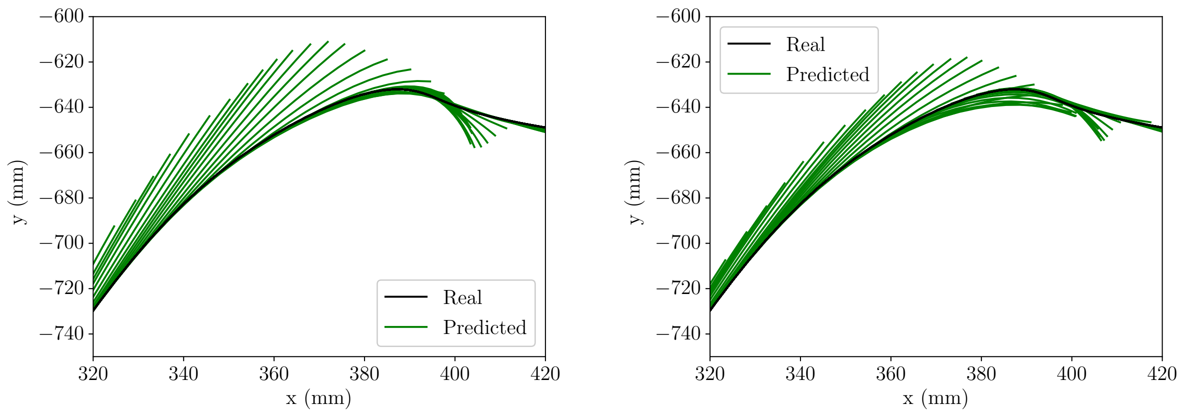
Indeed, from Figure 5.6, we notice that the double encoder network has predictions that are closer to the real trajectory. To compare only the predicted time step with its corresponding ground truth, we present Figure 5.7. It shows the input look back window in black, the predicted trajectory in green and the true trajectory in read. This figure confirms what we have seen in Table 5.3: the double encoder network has a better prediction for the case $(n, m) = (30, 15)$. It forecasts a trajectory close to the true one for the given fragment.



(a) Prediction for single encoder network.

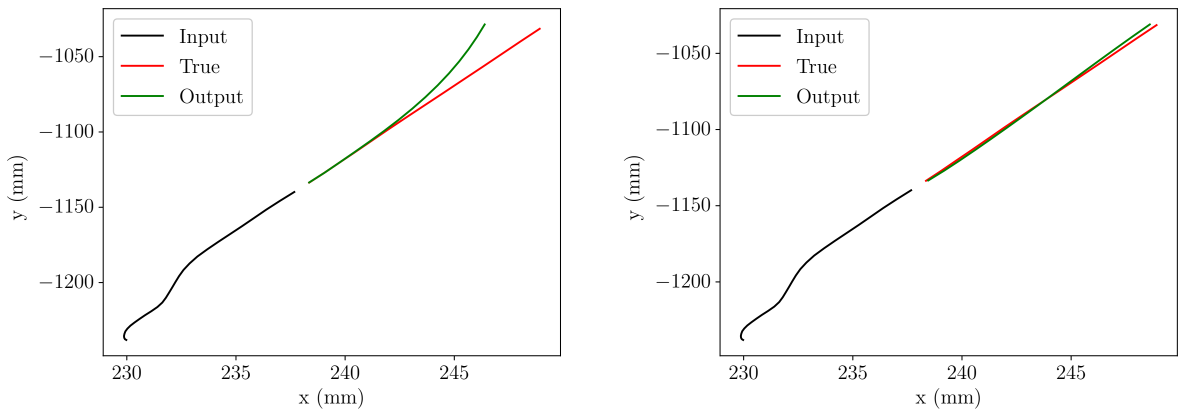
(b) Prediction for double encoder network.

FIGURE 5.5 – All predicted segments (in green) compared to the whole trajectory (black), using $(n, m) = (30, 15)$.



(a) Prediction for single encoder network. (b) Prediction for double encoder network.

FIGURE 5.6 – Up close in a turn, containing multiple predictions (in green), compared to the whole trajectory (black), using $(n, m) = (30, 15)$.

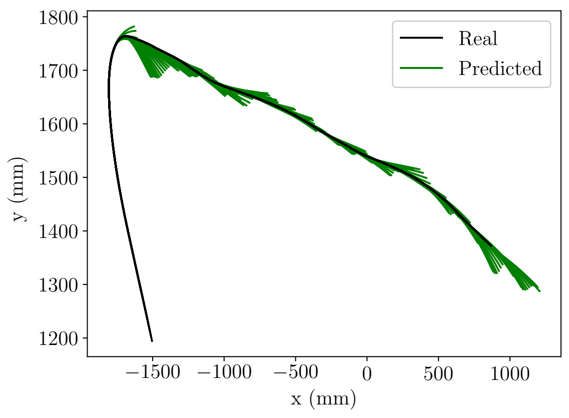


(a) Prediction for single encoder network. (b) Prediction for double encoder network.

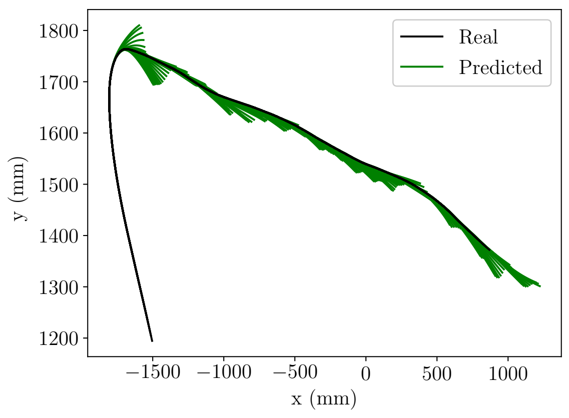
FIGURE 5.7 – Single predictions (in green), compared true trajectory (red), using $(n, m) = (30, 15)$.

Regarding $(n, m) = (60, 30)$, we can analyze Figures 5.8 and 5.9. Figure 5.8 shows the real trajectory in black and all the possible predictions in green. It is noticeable that the single encoder prediction has less green lines pointing outside the black route, if we compare the trajectories at $x > -1000$ mm. The figure confirms that our data from Table 5.4 is indeed accurate.

Figure 5.9 shows a look back window in black, a prediction in green and a true trajectory in red. We can see that the single encoder network better predicted the shape of the trajectory and its length, having a superior performance.

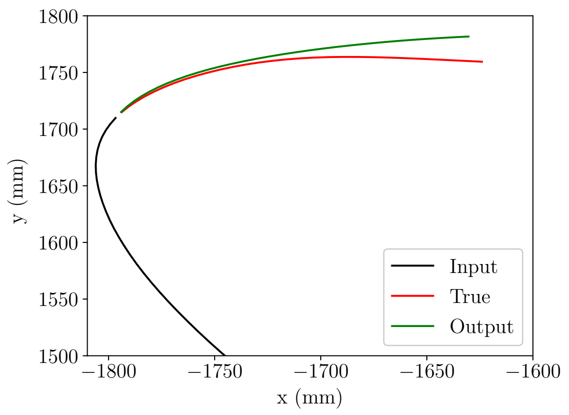


(a) Prediction for single encoder network.

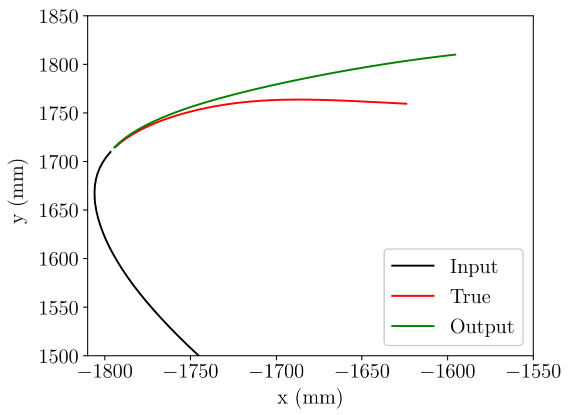


(b) Prediction for double encoder network.

FIGURE 5.8 – All predicted segments (in green) compared to the whole trajectory (black), using $(n, m) = (60, 30)$.



(a) Prediction for single encoder network.



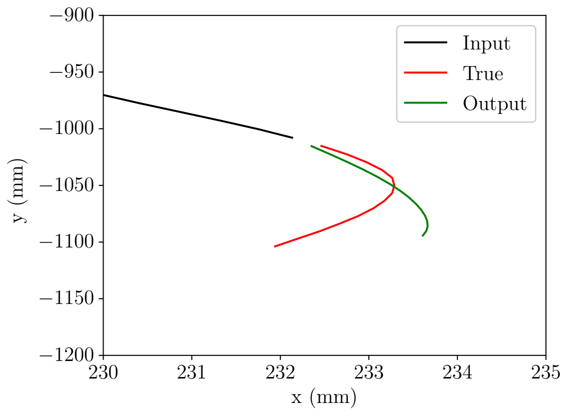
(b) Prediction for double encoder network.

FIGURE 5.9 – Single predictions (in green), compared true trajectory (red), using $(n, m) = (60, 30)$.

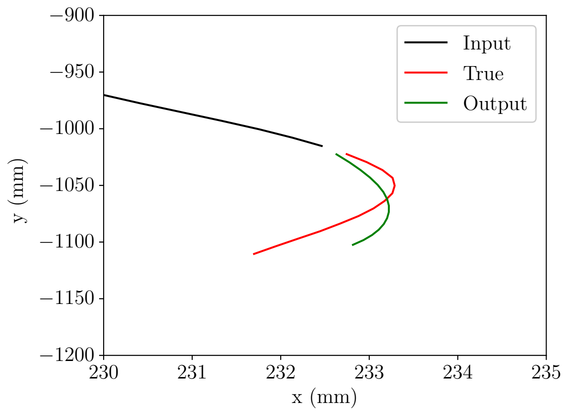
From Figures 5.7b and 5.9a we notice that the best network for each look back window size generalized the robots' dynamics with remarkable precision, as the predicted trajectories resemble the true ones. Indeed, the average displacement for the predicted trajectories is only 47.42 mm for the double encoder network with $(n, m) = (30, 15)$. Likewise, the AVD for the double encoder network with $(n, m) = (60, 30)$ is 277.65 mm. Considering that the scale of a robot is around 18 cm, an accumulated deviation of 4 cm is very small, and an accumulated deviation of 27.7 cm is slightly bigger than a robot.

Both architectures, however, struggle with sharp turns. During a sharp turn, there is not enough information in the past about what the robot might do, no matter how big is the look back window. As soon as the robot start turning, the neural network quickly correct its prediction to the right direction. Figure 5.10 shows a sequence of frames during a sharp turn. Figure 5.10a is the moment the turn has just started and Figure

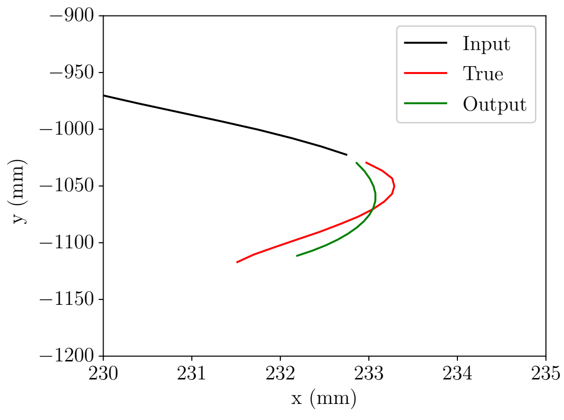
5.10d shows the turn almost complete. We can see that the neural network struggles at first, but predicts better with more past information.



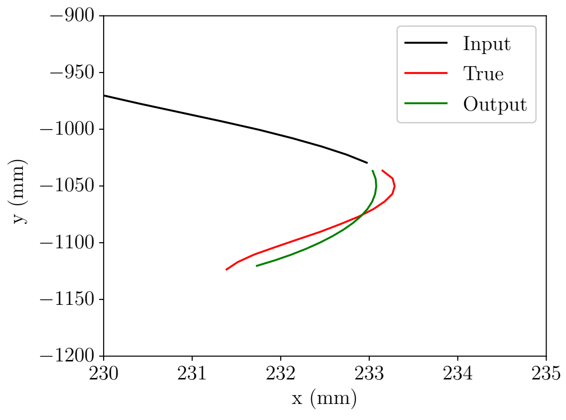
(a) First frame of a sharp turn.



(b) Second frame of a sharp turn.



(c) Third frame of a sharp turn.



(d) Fourth frame of a sharp turn.

FIGURE 5.10 – Sequence of frames for a sharp turn, using the double encoder network with $(n, m) = (30, 15)$.

5.6 Trade off between architectures

From our previous analysis we can conclude that the double encoder network performs better for a look back window of 30, and the single encoder network works better for a look back of 60. Both have very similar inference times of about 32 ms, roughly two camera frames.

The single encoder network with $(n, m) = (60, 30)$ provides us with more information about the future, nonetheless its short term prediction is not as accurate as that of the double encoder for $(n, m) = (30, 15)$. If we calculate the MAE, AVD and AFD for the single encoder with look back window of 60, but only consider its first 15 predicted time step for the error metrics we got a MAE of 65.31, an AVD of 51.82 and an AFD of 6.34.

Using the double encoder with $(n, m) = (30, 15)$, thus, provides us better accuracy, while the single encoder with $(n, m) = (60, 30)$ gives us more future time steps.

6 Conclusions and Future Work

6.1 Conclusions

In this work, we introduced deep learning techniques for time series forecasting and applied them to predict the future trajectory of small size league robots during the play time. We proposed two architectures for solving our problem. The first one is an encoder-decoder network that receives a look back window of 60 times steps and predict accurately the next 30. The second one adds a new encoder to the aforementioned architecture to account for the ball's past information. It outperforms the single encoder network when we consider a look back window of 30 to forecast the next 15 time steps.

The neural network we developed excelled at predicting the robot's trajectory. Its predicted trajectory scores better than a Kalman predictor in average displacement and average final displacement. The double encoder network excels at predicting 15 time steps in the future, with great accuracy, while the single encoder provides less accuracy but 60 future time steps. The single encoder model achieves 54.5% lower scores for Average Displacement error in comparison to the Kalman predictor. The double encoder network decreased the Mean Absolute Error in 41.6%.

A simple video of the double encoder network forecasting a trajectory is available at `https://youtu.be/NfpZMYRrpgY`.

6.2 Future Work

The proposed models did not have an inference time low enough to be used in real time in RoboCup games, so we will work with model quantization techniques to decrease inference time for the architecture and we will look into the main inference engines capable of providing us faster inference. We aim at achieving an inference time smaller than 16.6 ms as this is the time between two SSL vision frames. We are also going to integrate the models in our SSL decision making code to find out whether more accuracy or more future time steps favors our robot the more.

Besides that, future work can be done to improve our network. We can leverage the graph architecture introduced in Salzmann *et al.* (2020) to model the whole opponent team and predict the trajectories of all robots. This is interesting because a single entity controls all the robots of a team in a SSL game. We can also analyze the efficiency of using Generative Adversarial Networks to train the model, as it has shown great results for forecasting the trajectory of pedestrians (GUPTA *et al.*, 2018). In addition, we can add new features to the prediction, like the kick, the dribble as they are important for the decision making of our team.

Bibliography

ABIYEV, R.; AKKAYA, N.; AYTAC, E.; IBRAHIM, D. Behaviour tree based control for efficient navigation of holonomic robots. **International Journal of Robotics and Automation**, v. 29, 01 2014.

ADACHI, Y.; ITO, M.; NARUSE, T. Classifying the strategies of an opponent team based on a sequence of actions in the robocup ssl. *In*: BEHNKE, S.; SHEH, R.; SARIEL, S.; LEE, D. D. (Ed.). **RoboCup 2016: Robot World Cup XX**. **Proceedings** [...]. Cham: Springer International Publishing, 2017. p. 109–120. ISBN 978-3-319-68792-6.

ADACHI, Y.; ITO, M.; NARUSE, T. Online strategy clustering based on action sequences in robocupsoccer small size league. **Robotics**, v. 8, n. 3, 2019. ISSN 2218-6581. Available at: https://www.mdpi.com/2218-6581/8/3/58.

ADACHI, Y.; ITO, M.; NARUSE, T. Online strategy clustering based on action sequences in robocupsoccer small size league. **Robotics**, v. 8, n. 3, 2019. ISSN 2218-6581. Available at: https://www.mdpi.com/2218-6581/8/3/58.

ALBRECHT, S. V.; STONE, P. Autonomous agents modelling other agents: A comprehensive survey and open problems. **Artificial Intelligence**, v. 258, p. 66–95, 2018. ISSN 0004-3702. Available at: https://www.sciencedirect.com/science/article/pii/S0004370218300249.

BAHDANAU, D.; CHO, K.; BENGIO, Y. Neural machine translation by jointly learning to align and translate. *In*: **ICLR 2015 : International Conference on Learning Representations 2015**. **Proceedings** [...]. San Diego, United States: ICLR, 2015.

BAI, S.; KOLTER, J. Z.; KOLTUN, V. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. **CoRR**, abs/1803.01271, 2018. Available at: http://arxiv.org/abs/1803.01271.

BALABAN, D.; FISCHER, A.; BISWAS, J. A real- time solver for time-optimal control of omnidirectional robots with bounded acceleration. *In*: **2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. **Proceedings** [...]. Madrid, Spain: IROS, 2018. p. 8027–8032.

BARRATT, S. T.; BOYD, S. P. Fitting a kalman smoother to data. *In*: **2020 American Control Conference (ACC)**. **Proceedings** [...]. Denver, United States: ACC, 2020. p. 1526–1531.

BENGIO, Y.; SIMARD, P.; FRASCONI, P. Learning long-term dependencies with gradient descent is difficult. **IEEE Transactions on Neural Networks**, v. 5, n. 2, p. 157–166, 1994.

BRITZ, D.; GOLDIE, A.; LUONG, M.-T.; LE, Q. Massive exploration of neural machine translation architectures. *In*: **Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing**. **Proceedings** [...]. Copenhagen, Denmark: Association for Computational Linguistics, 2017. p. 1442–1451. Available at: https://aclanthology.org/D17-1151.

CAPOBIANCO, S.; MILLEFIORI, L. M.; FORTI, N.; BRACA, P.; WILLETT, P. Deep learning methods for vessel trajectory prediction based on recurrent neural networks. **IEEE Transactions on Aerospace and Electronic Systems**, p. 1–1, 2021. ISSN 1557-9603.

ERDOGAN, C.; VELOSO, M. M. **Action Selection via Learning Behavior Patterns in Multi-Robot Domains**. Carnegie Mellon University, Jun 2018. Available at: https://kilthub.cmu.edu/articles/journal_contribution-/Action_Selection_via_Learning_Behavior_Patterns_in_Multi-Robot_Domains/6602957/1.

GIULIARI, F.; HASAN, I.; CRISTANI, M.; GALASSO, F. Transformer networks for trajectory forecasting. **2020 25th International Conference on Pattern Recognition (ICPR)**, p. 10335–10342, 2021.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. Cambidge, MA, USA: MIT Press, 2016. `http://www.deeplearningbook.org`.

GRAVES, A.; WAYNE, G.; DANIHELKA, I. Neural turing machines. **CoRR**, abs/1410.5401, 2014. Available at: http://arxiv.org/abs/1410.5401.

GUPTA, A.; JOHNSON, J.; FEI-FEI, L.; SAVARESE, S.; ALAHI, A. Social gan: Socially acceptable trajectories with generative adversarial networks. *In*: **2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition**. **Proceedings** [...]. Salt Lake City, United States: IEEE, 2018. p. 2255–2264. ISSN 2575-7075.

HE, H.; BOYD-GRABER, J.; KWOK, K.; DAUMÉ III, H. Opponent modeling in deep reinforcement learning. *In*: BALCAN, M. F.; WEINBERGER, K. Q. (Ed.). **Proceedings of The 33rd International Conference on Machine Learning**. **Proceedings** [...]. New York, New York, USA: PMLR, 2016. (Proceedings of Machine Learning Research, v. 48), p. 1804–1813. Available at: http://proceedings.mlr.press/v48/he16.html.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural computation**, v. 9, p. 1735–80, 12 1997.

IVANOVIC, B.; PAVONE, M. The trajectron: Probabilistic multi-agent trajectory modeling with dynamic spatiotemporal graphs. *In*: **2019 IEEE/CVF International Conference on Computer Vision (ICCV)**. **Proceedings** [...]. Seoul, Korea: IEEE, 2019. p. 2375–2384.

JOSPIN, L. V.; BUNTINE, W.; BOUSSAID, F.; LAGA, H.; BENNAMOUN, M. **Hands-on Bayesian Neural Networks – a Tutorial for Deep Learning Users**. 2020.

KALMAN, R. E. A New Approach to Linear Filtering and Prediction Problems. **Journal of Basic Engineering**, v. 82, n. 1, p. 35–45, 03 1960. ISSN 0021-9223. Available at: https://doi.org/10.1115/1.3662552.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. *In*: BENGIO, Y.; LECUN, Y. (Ed.). **3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings**. **Proceedings** [...]. San Diego, United States: ICLR, 2015. Available at: http://arxiv.org/abs/1412.6980.

KITANO, H.; ASADA, M.; KUNIYOSHI, Y.; NODA, I.; OSAWA, E.; MATSUBARA, H. Robocup: A challenge problem for ai. **AI Magazine**, v. 18, n. 1, p. 73, Mar. 1997. Available at: https://www.aaai.org/ojs/index.php/aimagazine/article/view/1276.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. **Commun. ACM**, Association for Computing Machinery, New York, NY, USA, v. 60, n. 6, p. 84–90, maio 2017. ISSN 0001-0782. Available at: https://doi.org/10.1145/3065386.

LIM, B.; ARIK, S. O.; LOEFF, N.; PFISTER, T. **Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting**. 2020.

LIM, B.; ZOHREN, S. Time-series forecasting with deep learning: a survey. **Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences**, The Royal Society, v. 379, n. 2194, p. 20200209, Feb 2021. ISSN 1471-2962. Available at: http://dx.doi.org/10.1098/rsta.2020.0209.

MAKAROV, P. A.; YIRTICI, T.; AKKAYA, N.; AYTAC, E.; SAY, G.; BURGE, G.; YILMAZ, B.; ABIYEV, R. H. A model-free algorithm of moving ball interception by holonomic robot using geometric approach. *In*: CHALUP, S.; NIEMUELLER, T.; SUTHAKORN, J.; WILLIAMS, M.-A. (Ed.). **RoboCup 2019: Robot World Cup XXIII**. **Proceedings** [...]. Cham: Springer International Publishing, 2019. p. 166–175. ISBN 978-3-030-35699-6.

NGUYEN, T. M.; BARANIUK, R. G.; BERTOZZI, A. L.; OSHER, S. J.; WANG, B. **MomentumRNN: Integrating Momentum into Recurrent Neural Networks**. 2020.

OORD, A. van den; DIELEMAN, S.; ZEN, H.; SIMONYAN, K.; VINYALS, O.; GRAVES, A.; KALCHBRENNER, N.; SENIOR, A. W.; KAVUKCUOGLU, K. Wavenet: A generative model for raw audio. *In*: **The 9th ISCA Speech Synthesis Workshop, Sunnyvale, CA, USA, 13-15 September 2016**. **Proceedings** [...]. ISCA, 2016. p. 125. Available at: http://www.isca-speech.org/archive/SSW_2016/abstracts-/ssw9_DS-4_van_den_Oord.html.

PAPOUDAKIS, G.; ALBRECHT, S. V. Variational autoencoders for opponent modeling in multi-agent systems. **CoRR**, abs/2001.10829, 2020. Available at: https://arxiv.org/abs/2001.10829.

PARK, S.; KIM, B.; KANG, C.; CHUNG, C.; CHOI, J. Sequence-to-sequence prediction of vehicle trajectory via lstm encoder-decoder architecture. *In*: **2018 IEEE Intelligent Vehicles Symposium, IV 2018**. **Proceedings** [...]. [*S.l.*]: Institute of Electrical and Electronics Engineers Inc., 2018. (IEEE Intelligent Vehicles Symposium, Proceedings), p. 1672–1678. Publisher Copyright: © 2018 IEEE.; null ; Conference date: 26-09-2018 Through 30-09-2018.

PASCANU, R.; MIKOLOV, T.; BENGIO, Y. On the difficulty of training recurrent neural networks. *In*: **Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28**. **Proceedings** [...]. [*S.l.*]: JMLR.org, 2013. (ICML'13), p. III–1310–III–1318.

RAILEANU, R.; DENTON, E.; SZLAM, A.; FERGUS, R. Modeling others using oneself in multi-agent reinforcement learning. *In*: DY, J.; KRAUSE, A. (Ed.). **Proceedings of the 35th International Conference on Machine Learning**. **Proceedings** [...]. PMLR, 2018. (Proceedings of Machine Learning Research, v. 80), p. 4257–4266. Available at: http://proceedings.mlr.press/v80/raileanu18a.html.

RANGAPURAM, S. S.; SEEGER, M. W.; GASTHAUS, J.; STELLA, L.; WANG, Y.; JANUSCHOWSKI, T. Deep state space models for time series forecasting. *In*: BENGIO, S.; WALLACH, H.; LAROCHELLE, H.; GRAUMAN, K.; CESA-BIANCHI, N.; GARNETT, R. (Ed.). **Advances in Neural Information Processing Systems**. **Proceedings** [...]. Curran Associates, Inc., 2018. v. 31. Available at: https://proceedings-.neurips.cc/paper/2018/file/5cf68969fb67aa6082363a6d4e6468e2-Paper.pdf.

ROBOCUP. **Rules of the RoboCup Small Size League**. 2021.

SALZMANN, T.; IVANOVIC, B.; CHAKRAVARTY, P.; PAVONE, M. Trajectron++: Dynamically-feasible trajectory forecasting with heterogeneous data. *In*: VEDALDI, A.; BISCHOF, H.; BROX, T.; FRAHM, J.-M. (Ed.). **Computer Vision – ECCV 2020**. **Proceedings** [...]. Cham: Springer International Publishing, 2020. p. 683–700. ISBN 978-3-030-58523-5.

SUTSKEVER, I.; VINYALS, O.; LE, Q. V. Sequence to sequence learning with neural networks. *In*: **Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2**. **Proceedings** [...]. Cambridge, MA, USA: MIT Press, 2014. (NIPS'14), p. 3104–3112.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, L.; POLOSUKHIN, I. **Attention Is All You Need**. 2017.

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO TC	2. DATA 08 de novembro de 2021	3. DOCUMENTO Nº DCTA/ITA/TC-048/2021	4. Nº DE PÁGINAS 63

5. TÍTULO E SUBTÍTULO:
Opponent Modeling for RoboCup Small Size League Robots

6. AUTOR(ES):
Lucas Steuernagel

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):
Instituto Tecnológico de Aeronáutica – ITA

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:
robotics; deep learning; robocup

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:
deep learning; robotics; robocup

10. APRESENTAÇÃO: **(X) Nacional** ( ) **Internacional**
Trabalho de Graduação, ITA, São José dos Campos, 2021. 63 páginas.

11. RESUMO:

The RoboCup is an international scientific community focused on intelligent robots. The small size league (SSL) is a mainstream category of the RoboCup soccer games, whose robots must fit in a cylinder of 18 cm diameter and 15 cm height. The ITAndroids – Aeronautics Institute of Technology robotics team – actively participates in such a category. SSL games are very competitive, and teams with the best game strategy have clear advantage during games. An essential step to achieve a good decision making is knowing what the future trajectory of opponent's robots is. Furthermore, a trustworthy prediction helps the team plan its trajectory to better avoid collisions and make the robots reach their destination faster. From the publicly available dataset of RoboCup SSL games' logs, we can extract the robots' position and orientation and the ball's position at each vision frame. We employ a Kalman smoother to remove noise from data and interpolate missing measurements. The smoother also calculates the speed for each object. We proposed two encoder-decoder network architectures for solving the forecasting problem: a single encoder network, which consumes only data from the robot's past motion, and a double encoder network, which employs data from both the robot and the ball. The network's input for the robot is a time series of its position, velocity and orientation. Likewise, the input for the ball is a time series of its position and velocity. Each network predicts a time series of future velocities for the robot. We perform integration to obtain the position from the speeds. We found out that the double encoder network performs better for a look back window of 30 time steps (equivalent to 0.5 seconds) to predict 15 (0.25 seconds) future time steps. It achieves 54.5% lower scores for Average Displacement error in comparison to the Kalman predictor. The single encoder network works better for predicting 30 future time steps (0.5 seconds) using a look back window of 60 (1 second) and achieves Average Displacement Error scores 41.6% lower than those of the Kalman predictor. These are remarkable results, as the Kalman predictor is the main method used for trajectory prediction in the context of RoboCup SSL. Both networks excelled at predicting the robot's trajectory and outperformed the Kalman predictor. The presented architectures have the potential to make our decision making and trajectory planning better and more efficient.