

Preface

This thesis was made as a concluding workpiece to my master's degree in information technologies at Ghent University. With this thesis, I aspire to demonstrate my analytical and independent problem-solving skills in combination with a critical research mindset. I choose this subject as it really intrigued me to be part of a research project led by the Royal Military Academy (RMA) of Belgium. Before I started my studies at Ghent University, I even considered pursuing an education in polytechnic engineering at the RMA. Autonomous systems such as drones and innovative applications of artificial intelligence have always been a great interest of mine. In addition, I saw the subject as a way to expand my knowledge in deep learning as this was a relatively new concept for me at the start of this thesis. Luckily, I had the privilege to be guided by a very experienced and helpful support team which provided me with knowledge and many insights in the field.

In the first place, I would like to thank ir. Tien-Thanh Nguyen. During the course of thesis, he remained a close contact and really pushed me to achieve the best results. Without his motivation and close follow-up on my progress, I would have not achieved the current level of success. In the second place I would like to thank Prof. dr. ir. Hiep Luong and Prof. dr. ir. Jan Aelterman. Both provided me with consistent, patient advice and guidance throughout this research process. I also want to thank the other members of my committee, Dr. ir. Benoit Pairet and Charles Hamesse, who helped made this research possible. Thank you for all your unwavering support. As a final note, I would like to mention Jose Portilla. While not knowing him personally, Portilla is a renowned teacher on all subjects related to data sciences. He has – based on his statistics – satisfied close to three million students on the online learning platform Udemy. I bought and followed an online course of his on deep learning with PyTorch (Portilla, 2019) which really helped me develop a profound understanding of all deep learning concepts I needed.

All source code and documentation is available in a public repository on GitHub and can be accessed with the following link: <https://github.com/lancedw/Thesis>

Abstract

The thesis proposes a method for ship motion prediction which is optimized to guide the landing of autonomous drones on unstable surface vessels. With this prediction, the drone can anticipate movements of the landing platform to reduce the landing impact and ensure the safety of both the drone and its surroundings. The target vessel is equipped with an onboard inertial measurement unit (IMU) motion sensor and a front-facing camera to facilitate the predictions. Based on the six degrees of object motion freedom, pitch and roll were chosen as primary prediction targets as they are most influential towards the stability of the landing platform. In future work, this could be expanded to include other parameters such as heave. Five deep learning neural network architectures were designed as prediction models in order to find an optimal architecture for the problem. Four out of the five models were capable of predicting multi-value sequential outputs with varying accuracy. It was found that the use of Long-Short Term Memory (LSTM) networks improved prediction accuracy. Additionally, the use of wave images resulted in the overall highest accuracy. However, performance decreases when the images contain obscurities. Models processing images as input also suffered from high latency. Based on empirical results, an Encoder-Decoder LSTM network architecture is proposed as the most optimal architecture. This model is capable of making one-minute-long predictions for pitch and roll with consistent accuracy. Tests resulted in an average prediction error of around two degrees and a latency of 155ms. The model uses previous pitch and roll measurements to make predictions which also omits its dependency on image clarity. All tests were performed in a simulated environment.

Table of contents

Preface.....	1
Abstract.....	3
Table of contents.....	4
List of Figures.....	7
List of Tables.....	8
List of Abbreviations.....	9
1 Introduction.....	10
1.1 Problem definition.....	10
1.2 Objectives.....	11
1.3 Thesis outline.....	11
2 Background and literature review.....	12
2.1 Motion states in six degrees of freedom.....	12
2.2 Ocean wave induced ship motion.....	13
2.3 Ship motion prediction methods.....	14
2.3.1 Dynamic linear modelling.....	14
2.3.2 Deep learning.....	15
2.3.3 Hybrid models.....	16
2.4 Artificial neural networks.....	16
2.4.1 Activation functions.....	17
2.4.2 Auto-encoders.....	18
2.4.3 RNN and LSTM layers.....	18
2.4.4 Convolutional Neural Networks.....	19
2.5 Tools.....	21
2.6 Expected performance.....	21
2.7 Summary.....	23
3 Data collection and preprocessing.....	24
3.1 Simulated data.....	24
3.1.1 Simulation parameters.....	25
3.1.2 Data augmentation.....	25
3.2 Real data.....	26
3.2.1 Sensors onboard the ASV.....	26
3.2.2 Sensory data challenges.....	27
3.3 Data pre-processing.....	27
3.3.1 Data cleaning.....	27

3.3.2	Data reduction	28
3.3.3	Data normalization.....	28
3.4	Data analysis.....	29
3.4.1	Statistical properties	29
3.4.2	Correlation and interactions.....	30
3.5	Data loading	31
3.5.1	Sequence creation	31
3.5.2	Train-test splitting.....	32
3.5.3	PyTorch batching.....	32
3.6	Summary	33
4	Model designs.....	34
4.1	Single-step model	34
4.2	Multi-step models	35
4.2.1	Encoder-Decoder LSTM.....	35
4.2.2	Sequential CNN.....	37
4.2.3	CNN LSTM single input.....	38
4.2.4	CNN LSTM dual input.....	39
4.3	Summary	40
5	Testing environment.....	41
5.1	Gradient descent algorithms.....	41
5.1.1	SGD Adam optimizer	41
5.2	Hyperparameters	42
5.3	Performance metrics.....	43
5.3.1	Loss function	43
5.3.2	Inference time	44
5.4	Summary	45
6	Results	46
6.1	Hyperparameter optimization.....	46
6.1.1	Number of epochs	46
6.1.2	Learning rate.....	47
6.1.3	LSTM hidden size	47
6.1.4	Activation functions	48
6.2	Single-step model	49
6.3	Multi-step models	51
6.3.1	Encoder-Decoder LSTM.....	51
6.3.2	Sequential CNN.....	54
6.3.3	CNN LSTM ensemble models.....	55

6.4	Augmented data	57
6.5	Inference time.....	57
6.6	Summary	58
7	Conclusion	59
7.1	Future work.....	59
	References.....	61
	Appendix A. Individual predictions and LPF	65

List of Figures

Figure 1: Pollux P902 landing target vessel with autonomous helicopter-type drone ("MarLand," 2019).....	10
Figure 2: Six degrees of freedom in ship motion (de Masi et al., 2011).....	12
Figure 3: Heave (m) in function of time (s) (Ham et al., 2017).....	13
Figure 4: Neural network structure (left) and a linear neuron (right).....	17
Figure 5: Neurons in a recurrent neural network with additional feed-back connections (Portilla, 2019)	18
Figure 6: LSTM-cell components (left) and their mathematical notations (right) (Portilla, 2019).....	19
Figure 7: Convolution over grayscale image with 3x3 kernel (Portilla, 2019).....	20
Figure 8: Max pooling with 2x2 kernel (Portilla, 2019)	20
Figure 9: Generated images of incoming waves (chronologically ordered left to right, top to bottom).....	24
Figure 10: Augmented images.....	26
Figure 11: ZED-mini IMU stereo camera (Stereolabs, Paris, France).....	26
Figure 12: Scatterplot of pitch and roll.....	27
Figure 13: Maximum range of rolling motion for ships	28
Figure 14: Simulated Pitch and Roll distributions.....	30
Figure 15: Correlation matrix for pitch and roll.....	31
Figure 16: Sequence creation with moving input and output sequence.....	32
Figure 17: Single-step model architecture variants: single output (left), dual output (right)	35
Figure 18: Encoder-Decoder LSTM architecture.....	36
Figure 19: Sequential CNN neural network architecture.....	37
Figure 20: CNN LSTM single input architecture	38
Figure 21: CNN LSTM dual input architecture.....	40
Figure 22: Inference time over 10.000 prediction with GPU warm-up effect.....	44
Figure 23: Training losses for different numbers of epochs.....	46
Figure 24: Training loss for different learning rates.....	47
Figure 25: Training loss for different LSTM hidden sizes.....	48
Figure 26: Predicted (orange) vs. real (blue) values for pitch (top) and roll (bottom)	50
Figure 27: MSE loss during training of the single-step models.....	51
Figure 28: Encoder-Decoder LSTM training and validation losses for different IO-ratios.....	52
Figure 29: LPF measurements for 10/60 (top) and 120/60 (bottom) IO-ratio with reference line at 3°	53
Figure 30: LPF measurements for 60/120 (top) and 120/120 (bottom) IO-ratio with reference line at 3°	54
Figure 31: Training and validation loss for different IO-ratios of the sequential CNN model.....	54
Figure 32: Training and validation loss for CNN LSTM ensemble models.....	55
Figure 33: LPF for CNN LSTM at 10/120 IO-ratio for single (top) and dual (bottom) input on grayscale images	56
Figure 34: CNN LSTM dual input prediction (red) vs. real (blue) on augmented frames.....	57
Figure 35: CNN LSTM single input prediction (red) vs. real (blue) on augmented frames.....	57
Figure A-1: Encoder-Decoder LSTM - 60/60.....	65
Figure A-2: Sequential CNN - 10/60.....	66
Figure A-3: CNN LSTM single input (colored) - 10/60	67
Figure A-4: CNN LSTM dual input (colored) - 10/60.....	68
Figure A-5: CNN LSTM dual input (colored) - 10/120.....	69

List of Tables

Table 1: Overview of different activation functions.....	17
Table 2: Kaminskyi's results for different models (Kaminskyi, 2019a).....	21
Table 3: Prediction error at different future time-steps of Kaminskyi's best model (Kaminskyi, 2019a).....	22
Table 4: Statistical information for pitch and roll in simulated dataset	29
Table 5: Architecture naming conventions and their explanations	34
Table 6: Parameter table for single-step models	34
Table 7: Parameter table for Encoder-Decoder LSTM	36
Table 8: Parameter table for sequential CNN model	38
Table 9: Parameter table for CNN LSTM single input and dual output (red)	39
Table 10: Average pitch and roll errors for different activation function configurations	49
Table 11: Average pitch and roll errors for single-step LSTM variants at 10/1 and 50/1 IO-ratios.....	49
Table 12: Average pitch and roll errors for the Encoder-Decoder LSTM per IO-ratio.....	52
Table 13: Average pitch and roll errors for CNN LSTM single [img] and dual [img-PR] input models.....	56
Table 14: Inference time, PR error and number of trainable parameters for each model's optimal configuration.....	58

List of Abbreviations

AI:	Artificial Intelligence
ANN:	Artificial Neural Network
ASV:	Autonomous Surface Vessel
CNN:	Convolutional Neural Network
CPU:	Central Processing Unit
FC:	Fully Connected
GPU:	Graphics Processing Unit
Img:	Images
IMU:	Inertial Measurement Unit
IO:	Input-Output
LSTM:	Long Short-Term Memory
NN:	Neural Network
PoV:	Point of View
PR:	Pitch and Roll
RAM:	Random Access Memory
ReLU:	Rectified Linear Unit
RMA:	Royal Military Academy
RNN:	Recurrent Neural network
Tanh:	Hyperbolic Tangent
VTOL:	Vertical Take Off and Landing

1 Introduction

In the last few years, the world has seen an exponential increase in technological advancements. This evolution sparked a new influence of autonomous systems controlled by artificial intelligence (AI). Each of these systems being designed with their own unique characteristics and optimized for the desired task. Increasingly more of these systems are being deployed as a direct or indirect replacement for tasks humans could do, but also, for tasks too complex for humans to accomplish. And because these autonomous systems are optimized for specific jobs, they are often more accurate and accomplish them faster than humans.

Autonomous systems are especially useful in military operations. They can take over the role of a human in dangerous environments such as an active warzone and can therefore eliminate the endangerment of someone's life. On the other hand, they can also be used as a complimentary asset, providing support and aid in logistics. An increasing amount of these autonomous assets such as drones, surface vessels, tanks and reconnaissance vehicles are being deployed around the world for various objectives. However, with this increasing amount of autonomous assets, there is need for proper communication between them, to allow them to work together and be aware of the state of each other when they need to interact (de Cubber, 2019).

"Interoperability is the key that acts as the glue among the different units within the team, enabling efficient multi-robot cooperation." ("MarSur," 2019)



Figure 1: Pollux P902 landing target vessel with autonomous helicopter-type drone ("MarLand," 2019)

1.1 Problem definition

The Robotics & Autonomous Systems lab of the Belgian Royal Military Academy is currently working on two autonomous vehicles in two separate projects named MarSur and MarLand. Project MarSur is developing framework for autonomous systems to easily interact with each other. More concrete, they are developing a heterogeneous interoperability and collaboration framework which is seamlessly interoperable with existing infrastructure. This framework will, among others, be used to facilitate the communication and interaction of autonomous surface vessels (ASV) and other unmanned aerial systems such as drones ("MarSur," 2019). Project MarLand focuses on research in one of these interactions, namely AI-assisted vertical take-off and landing (VTOL). The aim of the MarLand project is to provide a proof-of-concept solution and practical implementation for a helicopter-type drone with the capability to land autonomously on the Belgian Navy

vessels ("MarLand," 2020). The capability for these unmanned aerial drones to automatically take off and land on vessels in all kinds of environmental conditions remains a bottleneck for widespread deployment. Landing a relatively small aerial vehicle - that is inherently very receptive to wind gusts - on the pitching and rolling deck of a moving ship is a very difficult control problem that requires the consideration of the kinematics and dynamics of both the unmanned aerial vehicle and the ship. For a smooth landing to be possible, the target vessel (Figure 1) must be capable of determining its state in a three-dimensional space and predict its movement in the ocean. This way, the drone can anticipate the movement of the vessel and avoid collision. In addition, the predictions can be used to find a window of optimal landing opportunity in which the vessel remains in a relatively stable state. In conclusion, this thesis aspires to provide a state-of-the-art solution for ship motion prediction to serve as a landing guidance system for drones.

1.2 Objectives

The goal of this thesis is to research and develop a method to reliably predict the motion of a surface vessel using the data captured from onboard sensors. This method should be both accurate and have a low latency to allow for real-time deployment on minimal hardware. The prediction models can use current and past measurements of the state of the vessel captured by an Inertial Measurement Unit (IMU) sensor in combination with images of incoming waves taken from a stabilized camera pointing to the front of the vessel. These measurements are available in real-time. As an output, the model should provide the motion of the vessel in the form of a new sequence of motion states. The format of the prediction needs to be optimized towards facilitating VTOL on the vessel. The duration for these predicted sequences should be at minimum thirty seconds to provide the drone with ample time to prepare for a take-off or landing procedure. Bigger drones will need more time for this procedure so the predicted sequence duration should be maximized within the model's capabilities. The model will be deployed in real-time scenarios and will make predictions in real-time based on the continuous data stream from the onboard sensors. Therefore, it should be lightweight and not require substantial amounts of computational power. This means that the latency of the different proposed methods should also be considered when comparing different models. In conclusion, both prediction accuracy and latency should be optimized.

1.3 Thesis outline

The contents of this thesis are divided over the seven chapters. In this first chapter, a general introduction was given of the problem and the objectives. In the second chapter, the most important and insightful related studies are discussed. An overview is given of the state-of-the-art in motion prediction and commonly used methods for ship motion prediction are compared. All background information needed to fully understand the contents of this paper, is provided here as well. In the third chapter, all data related subjects are discussed, namely data collection, processing and analysis. In the fourth chapter, different solutions are proposed in the form of deep learning neural network architectures. The fifth chapter discusses how different models are going to be compared and evaluated. The results are discussed in chapter six followed by a concluding discussion in chapter seven. References and an appendix with complementary documents are added at the end of the paper.

2 Background and literature review

A lot of research has been performed in the field of motion prediction. It is a topic that has many different applications and is applied to solve or aid in a broad spectrum of problems. For example, with the upcoming trend of autonomous vehicles and self-driving cars, motion prediction is implemented to avoid collisions and provide a safe experience for the passengers, the vehicles themselves and their surroundings (Ren et al., 2021). Additionally, motion prediction also has applications in for example human motion prediction for robot cooperation (Tang et al., 2018) and ground motion prediction to anticipate seismic activity (Dhanya & Raghukanth, 2018). However, due to its complex and mostly non-deterministic nature, motion prediction remains a very difficult problem to solve and ship motion prediction is no exception. Before looking at some viable solutions however, a general understanding is required on how ship motion is described and how ocean wave dynamics induce interact with ships. The following two sections gently introduce some basic principles followed by three sections explaining different ship motion prediction methods.

2.1 Motion states in six degrees of freedom

The motion of a ship or any rigid object is described in six degrees of freedom, which can be divided into two categories: translational and rotational motion. To identify each movement type, three reference axes are introduced in a three-dimensional space. These reference axes run through the center of mass of the ship and are oriented as follows:

- Vertical Z-axis runs vertically through the vessel
- Transverse Y-axis runs horizontally across the vessel
- Longitudinal X-axis runs horizontally through the length of the ship

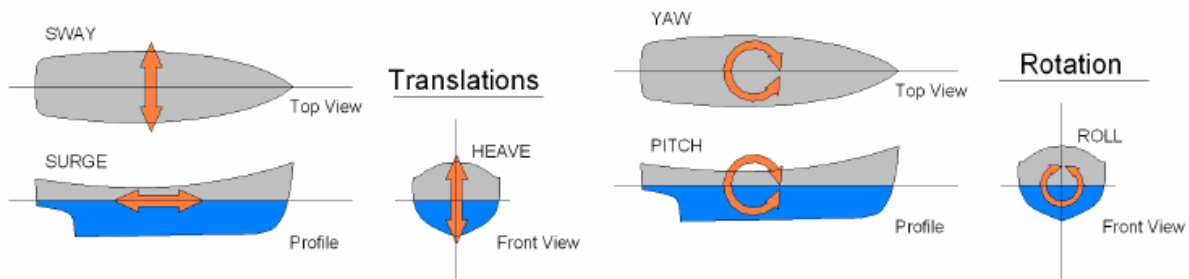


Figure 2: Six degrees of freedom in ship motion (de Masi et al., 2011)

Each class of motion, translation or rotation, among each of the three axes has a different effect on the movement of the vessel (Figure 2). The translational movements are expressed in linear units such as meters and are identified as followed:

- **Sway:** side to side movement along the transverse Y-axis
- **Surge:** forward and backwards movement along the longitudinal X-axis
- **Heave:** upward and downward movement along the vertical Z-axis

The rotational movements are expressed in angular units and are identified as followed:

- **Yaw:** rotational movement around the vertical Z-axis
- **Pitch:** rotational movement around the transverse Y-axis
- **Roll:** rotational movement around the longitudinal X-axis

To predict the motion of a ship, one must differentiate between these possible motions. Together they form the complete three-dimensional orientation of a ship. But not all of them need to be predicted. Surge and yaw are induced by the amount of thrust and the rudder position (steering the ship) which are controlled by the navigation systems. They will not change very drastically during the landing or take-off of the drone since this behavior would directly impede our main goal of

providing a smooth landing. On the other hand, the sway of a ship, also referred to as drift, is primarily caused by sideways winds or currents in the water and will have minor impact on the stability of the. In conclusion, surge, sway and yaw movements are slow and/or fully controllable by navigation systems and have minor impact on the stability of the vessel. A GPS-tracker broadcasting the position of the ship should be sufficient to anticipate these movements.

This leaves three remaining state parameters: roll, pitch and heave which have impact on the VTOL conditions. These three movements have one thing in common: they are all directly caused by the waves in the ocean and are very hard to control. Different methods exist to dampen these movements such as bilge keels and antiroll tanks. However, most of them are either infeasible, ineffective or do not provide the required stabilization on smaller vessels (Perez & Blanke, 2017). Predicting these movements instead of trying to dampen them, can be an alternative partial solution. Although using them together, will most likely yield the best performance. Pitch, roll and heave can be divided in two categories based on the effect they have on the landing and take-off of the drone. Pitch and roll are responsible for the stability of the landing surface and heave is responsible for the impact on the drone when landing.

To provide a stable and level landing zone for the drone, the pitch and roll of the vessel should remain constant and as close to zero as possible. The drone can adjust its pitch and roll to align itself with the landing platform, but only within a small operable VTOL window. Because the vessel can pitch and roll more than this window, landing can only take place if the ship remains stable for a certain duration. This is where the predictions play a crucial part. The drone can determine when to initiate a landing procedure if it detects a suitable window in the predictions. This way, it can avoid collision due to momentous changes in pitch or roll by anticipating the movement of the ship. The drone only comes in for landing when desired conditions are found within the predictions.

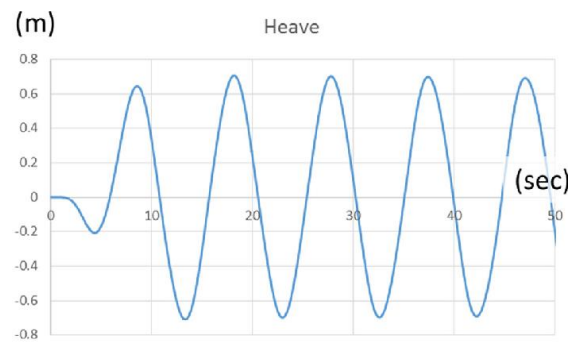


Figure 3: Heave (m) in function of time (s) (Ham et al., 2017)

To minimize the impact on the drone during the landing, the heave ideally needs to remain constant or slowly decrease. This means that the vessel is either not moving up or down, or it is slowly moving downwards following the motion of the descending drone. The heave of a floating vessel follows a regular wave-like function with a period equal to the ocean waves. The window of take-off/landing opportunity can be defined as the points where the vessel transitions from upwards to downwards motion, or vice versa, and thus has an acceleration of zero. Although conditions are not as strict as with pitch and roll. Lightweight drones have an extensive range of motion for heave during VTOL. They can change their altitude with low response time based on real-time readings from the IMU. Therefore, heave predictions are less important than pitch and roll. The prediction of heave only starts to become important when heavier drones with higher inertia are used that have slower reaction times.

2.2 Ocean wave induced ship motion

Natural ship motions are primarily caused by ocean waves which are little deterministic. Ocean waves are the result of an accumulation of several types of waves including capillary waves generated from atmospheric pressure, wind waves, and planetary waves (Silva, 2015). The product of this accumulation of waves is the stochastic nature of sea motion, which is

often described using a wave energy density spectrum (Abujoub, 2019). According to Perez T., the motion of a ship can be decomposed into three separate motion inducing forces. The superposition or accumulation of these three forces results in the magnitude of the motion.

- **First-order wave-induced forces.** This force is oscillatory. This is commonly modelled as a time series disturbance obtained by combining the wave spectrum with the vessel's Motion Response Amplitude Operators (Motion RAO), which are transfer functions that map the wave elevation or wave slope into force and motion.
- **Slowly varying disturbance forces.** This force is produced by current, wind and second-order wave effects such as wave mean-drift.
- **Control-induced forces.** This is the force induced by the control system, which is usually designed to counteract only the effect of the slowly varying disturbances.

Each one of these forces has profound and proven mathematical foundation which is well documented in Perez T.'s lecture paper (Perez & Fossen, 2005). Besides these forces, the wave encounter frequency spectrum also plays a role in modelling the motion of a ship. According to Dr. Q. Judge, the motions of a vessel can be seen as a three-part black box structure. The input are the ocean waves and their induced forces. The black box is the combination of the ship's dynamics like its inertia, natural frequency and physical form and the output are the motions of the ship in oceanic waves (Q. Judge, 2019).

In conclusion, the motion of a vessel in ocean waves is a complex interplay between the dynamics of the ocean and the dynamics of the ship. Due to extensive research in these fields, all of the above-mentioned concepts have been supported with mathematical foundations. These foundations have allowed researchers to accurately model ship dynamics in physics simulations (Ran et al., 2021). These models with the ever-evolving technological possibilities by the likes of computer vision and artificial intelligence, present a variety of methods that can be considered when trying to predict the motion of a ship.

2.3 Ship motion prediction methods

Ship motion prediction is incredibly useful for several naval operations such as aircraft landing, cargo transfer, off-loading of small boats, and artillery trajectory prediction. With this wide variety of applications and its long history of research, numerous approaches have been developed to solve the problem. However, most of the approaches found in published research studies can be divided in just three categories. They either use a dynamic model based on the above-mentioned principles, artificial neural networks or a hybrid of the first two. In the following sections, each method is discussed.

2.3.1 Dynamic linear modelling

Dynamic linear models or state space models are a set of equations mapping inputs to outputs of a given system based all parameters that affect the model state. It is an approach that relies on the mathematical foundations of ship and wave dynamics. A common method that is used is **minor component analysis (MCA)** (Luo et al., 1997). MCA has similar mathematics as the Principal Component Analysis, except that MCA utilizes the eigenvectors corresponding to the minor components. The viability of this method is proven by Zhao. He proposed an algorithm using MCA that was able to predict a twenty second sequence from 800 input datapoints with high and consistent accuracy (Zhao et al., 2004). Zhao used a dataset provided by a software simulation from JJMA inc. The data (surge, sway, heave, pitch, roll, yaw) was collected at 8Hz and down sampled to 2Hz. This frequency combined with the simulated data is very similar to the simulated data that will be used for this research (Chapter 3). In his work he compared the method to a neural network, vector autoregression (VAR) (Stock & Watson, 2001) and a Wiener filter (Chen et al., 2006). The conclusion was that MCA outperformed all other compared methods and was also suited for real-time implementation. It had the lowest latency based on 500 predictions and the fastest training time. However, only a simple three-layered linear regression neural network was tested. Other neural network architectures exist which are better suited for time-series predictions. Additionally, 400 seconds were needed to predict only 20 seconds, this is a very high input-output ratio. Finally, but most importantly, the data of the simulation did not show any form of noise. To compensate this, Zhao tested the models with varying levels of introduced

zero-mean Gaussian random noise. This caused the MCA method to quickly lose accuracy with a tenfold decrease in performance at 20% introduced noise. The percentage refers to the peak amplitude percentage or standard deviation of the introduced Gaussian noise.

Another commonly used method is **Kalman filtering**, also known as Linear Quadratic Estimation (LQE) (Kalman, 1960). Initially developed in 1960 and proven effective and reliable by its implementation in the Apollo project (Grewal & Andrews, 2010), Rudolf E. Kalman received the National Medal of Science for Engineering for his research. In theory, the Kalman filter is an algorithm that uses a series of measurements observed over time, including statistical noise and other inaccuracies, and produces estimates of unknown variables. The Kalman filter produces an estimate of the state of the system as a weighted average of the system's predicted state and the new measurement. These estimates tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe (Chen et al., 2006). In the research study of Fossen and Fossen, an exogenous Kalman filter (Johansen & Fossen, 2017) is used for ship trajectory and position estimation based on multiple sensory inputs (Fossen & Fossen, 2018). Another research was done by Peng where Kalman filters are used for estimating the dynamic ship motion states (Peng et al., 2019).

These studies show that the Kalman filter can be a reliable method for estimation problems. However, due to the complex interrelation between ocean waves and ship motions, setting up a dynamic model for the Kalman filter can be quite challenging. It is mostly used for theoretical models and is hard to apply to a real-world scenario where not all parameters are known. The issue of not knowing all parameters to build an accurate dynamic model caused the need for an alternative. As proposed by Zhong-yi Z., one of the alternatives could be to estimate these parameters (Zhong-yi, 2012). However, the complexity of these dynamic models, still remains. For this reason, dynamic modelling was not selected for the purpose of this thesis.

2.3.2 Deep learning

Another alternative in trying to determine these parameters was found in artificial intelligence and more specifically, deep learning. The idea is that instead of trying to figure out all necessary parameters to build a dynamic model, a computer is trained to build its own model which provides the mapping between in- and outputs. The computer receives a large set of inputs and their corresponding outputs and learns the relation between the two. This completely eliminates the need to know or estimate all parameters for a dynamic model and drastically reduces the complexity of the problem, which is one of the main reasons why this method was chosen.

The idea of enabling computers to train themselves to solve a problem, dates back to 1958 when the US Navy made a first attempt. However, due to the inability of these early neural networks to learn simple linear decision boundaries like the XOR-function, researchers quickly lost interest (Minsky & Papert, 1969). Small improvements were made in the following decades that slowly expanded the capabilities of neural networks. Some notable advancements were support for non-linear decision boundaries with multiple layers and the ability to train these multi-layered networks by back-propagating errors (Rumelhart et al., 1986). Nevertheless, they remained inferior to classical methods like Support Vector Machines (SVM) (Boser et al., 1992). It was only around 2010, that they really became popular when deep neural networks started to outperform all other approaches in computer vision tasks. This breakthrough was possible due to the increase in computing power the availability of large datasets. Ever since, deep neural networks quickly evolved beyond computer vision tasks and have been widely adopted for a plethora of different applications. One of these applications is time-series forecasting problems like ship motion prediction.

Extensive research has been performed in search of optimal deep neural network architectures for time-series prediction and image feature extraction. This presents reliable options today when building a network for ship motion prediction based on images and sensor data. In most cases, Long Short-Term Memory (LSTM) networks are used because they excel in time-series forecasting. In one the of the reviewed studies, a multiscale attention-based LSTM network is proposed to predict ship motion based as an improvement on regular LSTM networks (Zhang et al., 2021). The attention mechanism boosts the sensitivity of the system by paying more attention to significant signals and suppress interference of noise. It resulted in better performance than other popular methods. In another study, an L1 regularized extreme learning machine

is used instead of an LSTM for single-step predictions (predicting only one future value). This resulted in very low near-zero degree roll prediction errors (Guan et al., 2018). Lastly, in the research of Rashid M., an ensemble model was proposed combining a Convolutional Neural Network (CNN) with an LSTM and with a Gated Recurrent Unit (GRU) (Rashid et al., 2021). The CNN processes two images of incoming waves while the LSTM/GRU processes a sequence of pitch and roll values. Both systems would make a prediction from which the average is taken as result. But once again, this study only provided a solution for single value prediction instead of sequences.

While providing good solutions for ship motion prediction, all above mentioned research fails to meet the requirements for this thesis. They either only predict one future datapoint or they don't use images. Only one publication was found by **Nazar-Mykola Kaminskyi** where both images and sensor data are used to predict a sequence of pitch and roll (Kaminskyi, 2019a). Kaminskyi explored different neural networks that can predict the motion of a vessel based on pitch, roll and incoming wave images. The different model designs used a combination of CNN, LSTM and linear layers with the ones using images showing the best performance. He created a simulated dataset for his research that was used for this thesis as well and provided research and results that are directly comparable. However, his research lacked a comprehensive evaluation of the used models and did not include latency testing. Both will be addressed in this thesis to form a more concise and robust solution that also fully complies with our needs.

In conclusion, deep neural networks provide a state-of-the art solution for ship motion prediction. The fact that they require little knowledge of the underlying physics makes them very accessible. This is clearly visible in the current lay of the land as a majority of found studies on ship motion prediction proposed some form of neural networks. However, due to the virtually unlimited possibilities for designing a neural network architecture, finding the optimal one is not unambiguous. This thesis will explore different architectures in order to determine an optimal configuration for ship motion prediction.

2.3.3 Hybrid models

Hybrid models are models that combine dynamic modelling and deep learning to achieve better performance. In one paper, a neural net is used to correct the prediction made by a dynamic model (Wei et al., 2022). In another study, a hybrid model is applied for ship trajectory prediction based on current, waves and wind (Skulstad et al., 2021). These studies show that hybrid models can be highly effective in different ship motion prediction applications. However, this approach was not chosen due to the complexity of both the dynamic model and its integration with deep learning concepts.

2.4 Artificial neural networks

Neural networks, also referred to as artificial neural networks (ANN) are the building blocks of deep learning problems. They are composed of algorithms that permit software to train itself to perform tasks by exposing a layered neural network to vast amounts of data. Neural networks are structures inspired by the human brain and more specifically the neurons within and how they pass signals from one to another. However, similarities end beyond their connected structure.

A neural network is built in different layers which consist of multiple neurons. These neurons are connected to other neurons in adjacent layers and pass data forward over their connections. The input of one neuron is the output of all the preceding neurons it is connected to. Each neuron applies a weight and adds a bias to all its inputs and passes the aggregated resulting value forward. This way data is fed forward through the network and updated in every neuron. At the end of the network, the resulting value is compared to a ground truth value. Each neuron updates its weight and bias via backpropagation to improve its predictions as the model learns.

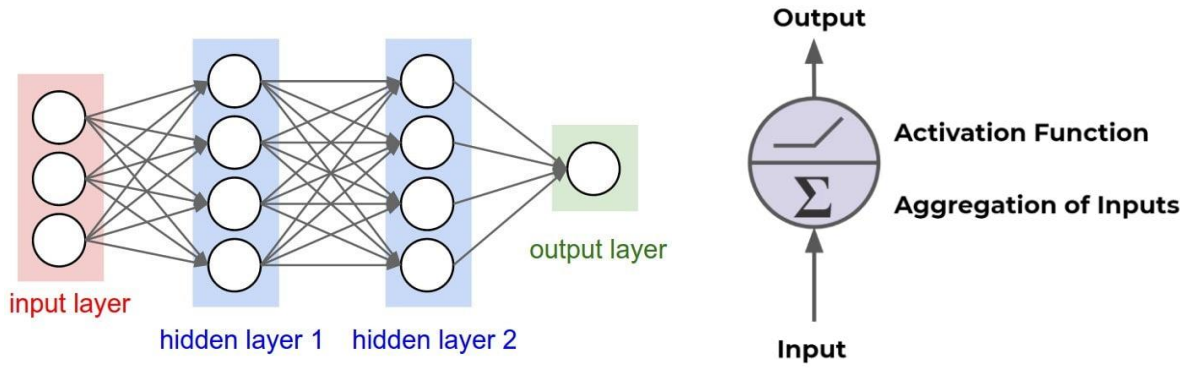


Figure 4: Neural network structure (left) and a linear neuron (right)

There are three main parts in a neural network, the input layer, output layer and hidden layers. The input layer has the same number of neurons as the features in the input data. The output layer size equals the number of features one wants to predict. For example, when pitch and roll are consumed and predicted, there are two input neurons and two output neurons.

Different architectures exist for different applications, each with their strengths and weaknesses. In this thesis, ordered time-series data is used which contains both numeric data as well as images. Because of this, different architectures are utilized that are designed to perform best with these types of data. Since no single model architecture exists that can handle all data well, different architectures will be combined to form ensemble models. In the following sections, all components and network architectures that play a role in this research are explained.

2.4.1 Activation functions

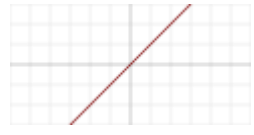


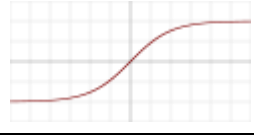
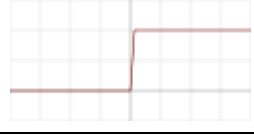
Linear	$f(a) = a$	
Rectified linear unit (ReLU)	$f(a) = \max(0, a)$	
Sigmoid	$\sigma(a) = \frac{1}{1 + e^{-a}}$	
Hyperbolic tangent	$\tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$	
Binary step	$f(a) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$	

Table 1: Overview of different activation functions

In every neuron the weighted sum of all input is taken, and a bias is added. This aggregation results in a new scalar value that is passed through an activation function before moving to the next layer. The activation function, also referred to as the transfer function, applies a linear or nonlinear transformation to this scalar to limit its output to a certain range. This is especially useful for classification problems where an activation function such as the sigmoid function can be used to

limit the output of a neuron to a range of zero to one. The output can then be interpreted as the predicted probability of the input belonging to a specific class. For linear regression problems where a numerical value needs to be predicted like pitch or roll, a linear activation is used. In Table 1, some common activation functions are shown together with their graphs. The sigmoid function $\sigma(a)$ and hyperbolic tangent $\tanh(a)$ are both used in LSTM networks while fold functions are used by pooling layers in CNN networks. Fold functions are special activation functions that perform aggregation over the results to take the mean, minimum or maximum.

In conclusion, different activation functions are used for different use cases. To predict numeric values such as pitch and roll, a linear or tanh (when input is normalized to $[-1, 1]$) activation can be used in the very last layer. However, hidden layers can use different activations to limit the output of their neurons. This way the network can decide whether each hidden neuron's output is important and should be activated. The choice of the activation function can heavily affect a model's performance (Sharma et al., 2020). As a future reference, ReLU and the hyperbolic tangent (tanh) will both be used in some proposed models in Chapter 4.

2.4.2 Auto-encoders

Auto-encoders are a type of neural networks that are designed to efficiently copy its input to its output. More specifically, the input gets encoded into a compressed representation, and then decoded or reconstructed based on this encoding. There are two main parts that make up an auto-encoder: an encoder and a decoder. An auto-encoder is characterized by two key features: the number of neurons in the input is the same as the output and the hidden layers serve as bottleneck. This bottleneck forces the model to learn only the most prominent features of its input that are needed to reconstruct it as accurate as possible (Lopez Pinaya et al., 2020).

The encoder part of an auto-encoder is capable of creating a sparse representation of the input that holds as much information as possible. This property can be used to train an auto-encoder on the images and use the encoder part as a pretrained feature extractor for the images. This concept is applied in the work of Kaminskyi. The auto-encoder was trained once on the simulated images and could afterwards be used without the need of retraining. However, in this research, the encoder-decoder configuration is used in a more liberal approach where two neural networks work together. One is used to encode the input and a second one decodes this encoding and makes new predictions based on the encoding. This form of auto-encoders is commonly referred to as variational auto-encoders, where encodings are decoded into new outputs instead of reconstructing the input (Kingma & Welling, 2019).

2.4.3 RNN and LSTM layers

Recurrent Neural Networks, RNN are special neural networks designed to work with sequential data (Sherstinsky, 2018). Sequential data is data where the order is important, for example time series data, sentences, audio etc. Ship motion data like pitch and roll fall within the time-series data category. The same goes for video footage or consecutive images where frames should be processed in a specific order. To learn the chronological relation between consecutive datapoints, RNNs are introduced. They are optimized network structures designed for sequential data processing.

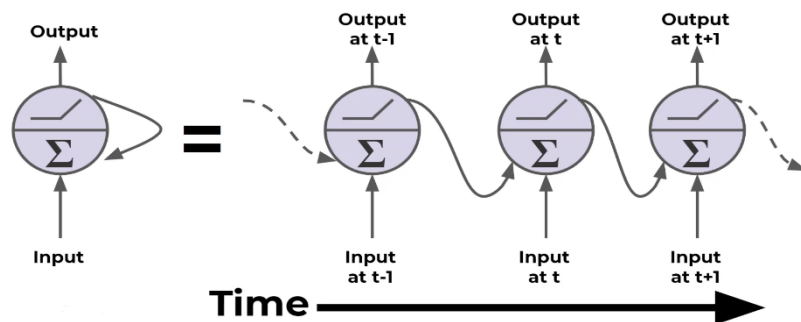


Figure 5: Neurons in a recurrent neural network with additional feed-back connections (Portilla, 2019)

To efficiently learn from ordered data, an RNN neuron passes its output forward to the next layer, but also back into an adjacent neuron in the same layer. Each neuron then uses both feed-forward and feed-back connections to compute a new result. This allows an RNN to progressively process an ordered sequence and maintain a memory of what has been processed. However, as illustrated in Figure 5, RNN network neurons just pass the previous output back into the network and thus only “remember” the short-term history in the sequence. This causes the model to “forget” its long-term history or in other words: the model loses perception on the general trend of the sequence.

LSTM networks provide a solution to this disadvantage by not only using a short-term memory but also saving a long-term memory state (Hochreiter & Schmidhuber, 1997). Each neuron of an LSTM network receives three inputs: the input data (x_t), the previous output (short-term memory, h_{t-1}) and a long-term memory state (c_{t-1}). The short-term and long-term memory are respectively called the hidden state and the cell state. Within an LSTM neuron or cell, four components are used to update these states and produce new hidden and cell states and a new output. These four components are gates, each with their own function: input, forget, update or output. The gates use activation functions to decide whether to let information through - “update” - or block information - “forget”. This way, the LSTM-cells gradually learn what information should be kept for long-term memory and which information can be discarded.

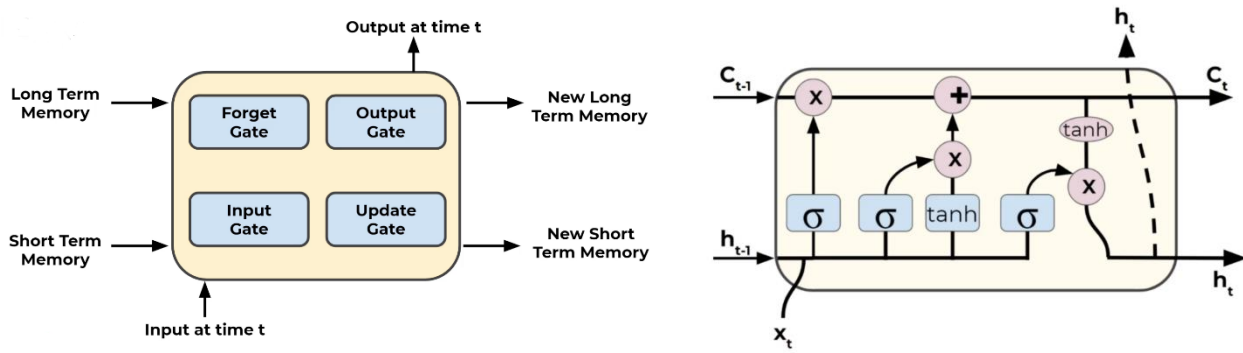


Figure 6: LSTM-cell components (left) and their mathematical notations (right) (Portilla, 2019)

Because of their optimized design for sequential data, LSTM networks will be the main building blocks for the deep learning models proposed in this thesis. They should be able to learn the trend of the ship's motion and make accurate predictions based on this trend. However, LSTM architectures are not the only viable option for sequence-to-sequence prediction problems. Gated Recurrent Units (GRU) and Peephole LSTMs could also be a reliable option; however, these are not used in this thesis and could be alternatives for future work.

2.4.4 Convolutional Neural Networks

Besides numeric pitch and roll values, there are also images available to aid with the prediction process. However, LSTM networks and standard linear networks are not very efficient when working with image data. Because of the way an image is represented, see below, the number of parameters in these networks ramp up very quickly, even with small images. Therefore, a second architecture is introduced: Convolutional Neural Networks, CNN. These networks are highly effective when dealing with images because they can learn multidimensional features (Wu, 2017).

Images are represented as three-dimensional matrices where every element in the matrix represents the intensity of a channel for a specific pixel. The width and height of this matrix are equal to image and the depth is the number of color channels in the image. When this data structure would be processed by a linear layer, the matrix would have to be flattened into a one-dimensional vector, causing substantial loss of information. CNN networks solve this problem by extracting features from the images with two-dimensional convolution kernels that don't require flattening.

A convolution kernel is a square, two-dimensional $n \times n$ array. Each element within this kernel contains a weight value. During a convolution, the kernel is moved over the matrix representation of the image and applies each of its weights to the corresponding pixel value as illustrated in Figure 7. The result is the weighted sum of those pixels placed at the center of the current kernel position in a new matrix. Because the kernel is two-dimensional just like the image, no information is lost, unlike when flattening the image. The kernel is moved across the whole image while repeating the same process. After one convolution, the result is a filtered image with more accentuated features. In typical fashion, multiple convolution layers are connected to allow each consecutive layer to extract more precise features such as wheels, eyes, windows etc. During the training process, the network learns the best values for each weight in the kernel to extract the most prominent features for the task. In the case of incoming wave images, the network is expected to extract the curve of the waves in the images. Because the wave images are colored, they contain three channels (red, green and blue) and thus three kernels are used – one for each channel. If images are loaded in grayscale, only one color channel is present.

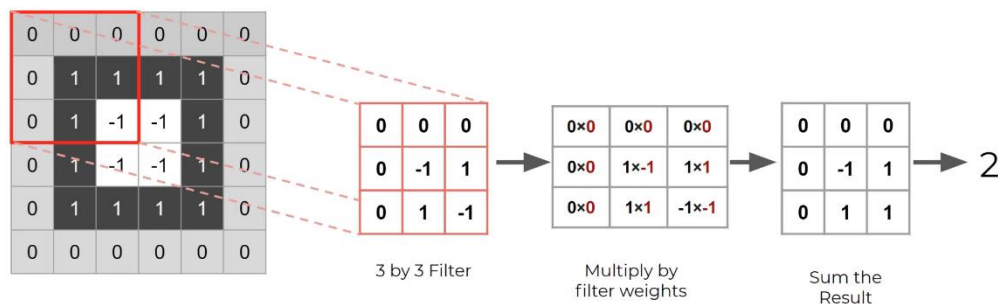


Figure 7: Convolution over grayscale image with 3x3 kernel (Portilla, 2019)

Two important parameters are used when working with convolutional layers: the kernel size and its stride distance. The first one defines the dimensions for the kernel while the stride determines how many steps the kernel moves forward before calculating a new filter value. Additionally, padding can be added to the border of the image to prevent data loss. This allows the filtered image to remain the same size as the input after a convolution. Otherwise, each convolution would remove a layer of pixels around the border of the image. Padding values are most common all white or all black pixel values.

A convolutional neural network still has a large number of parameters despite being more efficient than linear networks. When dealing with colored images and a lot of filters, tens to hundreds, a method is needed to reduce the number of parameters. A pooling layer samples the filter size down by applying a function to different subsets of data from the filter. The process is similar to a convolution where a $n \times n$ kernel is moved across the filter. However, the kernel has no weights. Instead, it computes the maximum of the values in the filter on a $n \times n$ basis. This method is called **Max Pooling**. For example, if a 2×2 kernel is used, four data points of the filter will be sampled down to one (Figure 8). If the kernel is moved forward two places on each iteration, this configuration leads to a 75% reduction in data. Another method is called Average Pooling where the average is taken from the kernel-sized subset.

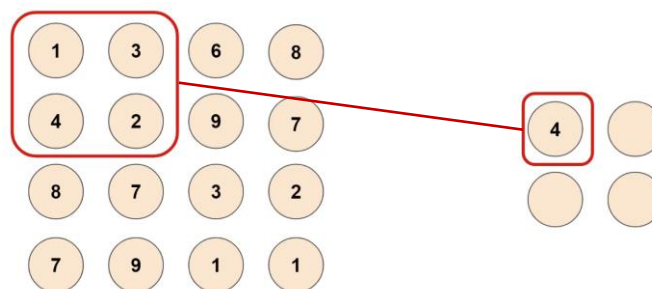


Figure 8: Max pooling with 2x2 kernel (Portilla, 2019)

2.5 Tools

To develop the deep learning neural networks, Python 3.8 was used. Python is a high-level, general-purpose programming language that is in - most cases - most suited for machine learning applications. To facilitate the development of the deep learning models, different packages were used. The most important of which are briefly discussed below.

Pandas, Seaborn and NumPy were used to process and manipulate data. NumPy provides functionality to perform mathematical functions on large datasets with multiple dimensions. Pandas provides the functionality to visualize and order the data in DataFrame objects. Seaborn was mainly used to perform data analysis on pitch and roll distributions and correlations. To plot training and test results, Matplotlib was used.

For the implementation of the deep learning models, PyTorch was used. PyTorch is an open-source framework for machine learning applications that allows fast, straightforward development. PyTorch was used together with the cudatoolkit extension. This enabled the models to be trained and tested on a dedicated graphics processing unit (GPU) to increase computing performance.

Blender was used to generate a dataset from an ocean wave simulation. Blender is an open-source three-dimensional creation suite that can be used for many purposes. In this case, it was used to simulate incoming waves on a vessel. The data itself will be discussed more in detail in Chapter 3.

All code was written in Jupyter Notebook for its ease of use and simple debugging. Whenever parts of the code in the notebooks were tested thoroughly, they were extracted to standalone Python files to be accessed via import statements.

2.6 Expected performance

To evaluate the performance of each model and their viability as possible solution based on the requirements discussed in the objectives, a minimum criteria needs to be defined. These criteria are a set of predefined baselines and references to which the model can be compared. They were determined based on related work, primarily on the research by Kaminskyi because he used the same dataset, and desired expectations. Note that the criteria defined in this section are defined for the simulation environment in which higher performance is expected.

Model	Pitch at 10 th second [denormalized RMSE]	Roll at 10 th second [denormalized RMSE]
LSTM enc dec PR	30.99°	29.44°
CNN stack FC	28.97°	28.94°
CNN stack FC PR	21.18°	22.56°
CNN FC PR	11.36°	11.28°
CNN LSTM image-enc PR-enc dec	3.56°	2.83°
CNN LSTM enc dec images PR	2.89°	2.70°
CNN LSTM enc dec images	3.42°	2.32°
CNN LSTM dec images PR	4.22°	4.16°

Table 2: Kaminskyi's results for different models with worst (red) and best (green) model highlighted (Kaminskyi, 2019a)

In Kaminskyi's research, various accuracies were measured between different model designs. The LSTM model that processed only pitch and roll as input performed worst, followed by the CNN models that only used images. The best performance was achieved by CNN LSTM ensemble models using both pitch, roll and images. The errors of each model are shown in Table 2, in which the worst and best performing model is highlighted in resp. red and green. These errors were calculated as the average difference between prediction and real values at the tenth predicted second. Models were tested

in a configuration where ten seconds are used to predict twelve (resp. 20 and 24 frames at 2Hz). For clarity, the model names in his research are based on the inputs they use and which architectures they are composed of. For example, *PR* means that pitch and roll are used as input, *stack* means that a stack of images was used as input and *FC* means that a fully connected linear layer is used. More information on the model design and naming can be found in his report.

From these results, a window of expected error margins can be derived. The green and red model serve as a reference benchmark to evaluate the models proposed in this research. The red model is a baseline of lowest expected performance. The green model is a reference to the accuracy that should be aimed for. In blue, the absolute lowest error value is highlighted, which was measured on a roll prediction. This value is a reference for what accuracies are possible. However, the model that resulted in this low error is suboptimal compared to the green one due to its high pitch error. Both pitch and roll are equally important when estimating the state of the vessel for a **drone landing**. Therefore, two similar and low prediction errors are more optimal than two low yet disproportional errors.

Below in Table 3, errors are shown of the best performing model after its parameters were optimized with the HyperBand algorithm (Li et al., 2016). The model was trained to predict a thirty second sequence at a 1Hz sample rate to conserve time and memory. Results show the accuracy decreases when along the prediction, with roll errors suffering more than pitch errors.

Parameter	10 th second	15 th second	30 th second
Pitch	2.89°	2.92°	3.06°
Roll	2.71°	3.75°	4.92°

Table 3: Prediction error at different future time-steps of Kaminskyi's best model (Kaminskyi, 2019a)

Based on these results, a general idea was formed about expected performances on the simulated dataset and criteria were set accordingly. Firstly, the **average error** on the time-series predictions at the 10th second should be no more than three degrees and no more than five degrees at the 30th predicted second. These values are chosen in assumption that the accuracy of any model decreases as it predicts further in the future. The errors should be calculated as the average over all predictions on a large dataset with unseen data (data that was not used for training). Secondly, all models will be compared to a **zero-predictor**. This is a hypothetical model that predicts zero for each point in the output. Models performing worse than the zero-predictor did not learn at all and can be classified as random generators. The zero-predictor serves as an un-biased reference for both new and old models. Finally, the prediction latency or **inference time** of the model needs to be as low as possible. The inference time should be measured as the average over ten thousand predictions with a batch size of one. The batch size is chosen at one as the model will not be able to predict more than one sequence at one time when predicting on live data-streams. There was no research found comparing similar neural network designs compared to ours. But some popular state-of-the-art classification neural networks like ResNet-18 (He et al., 2015) and GoogLeNet (Szegedy et al., 2014) show inference times between 30 and 150 milliseconds on a *single image* forward pass (Canziani et al., 2016). These values can serve as a reference, although it is expected that inference time will be higher due to multiple images needing to be processed to make a prediction. The testing on latency should ideally be expanded towards evaluating the model's hardware usage because resources will be limited when being deployed.

Whenever a prediction sequence is calculated, it will be analyzed by an algorithm that will search the sequence for a window of landing/take-off opportunity. During this window, roll and pitch – and additionally heave – should remain close to constant over a duration of continuous time. The requirements for this window will vary based the properties of the drone. To compensate for this, the parameters of this window should be easily changeable to the needs of the drone at hand. These are just some starting principles as this algorithm falls out of the scope of this thesis.

2.7 Summary

In this chapter, the state-of-the-art in motion prediction was discussed and different methods were introduced to predict wave-induced ship motions. The conclusion was made that for many years, ship motions were predicted based on linear modelling of the wave and ship dynamics. However, due to the non-linear non-deterministic nature of ocean waves and their interaction with ship motions, setting up an agile and accurate model is not trivial. For this reason, deep learning neural networks were chosen as an alternative solution. It is hypothesized that this modern approach can learn the complexity of dynamic models solely based on examples. Their accuracy and latency will be evaluated in a simulated environment to determine an optimal architecture. Results from related research and predefined expectations will be used to evaluate performance. In addition, all concepts of neural networks were introduced that are necessary to fully understand the contents of this paper. Primarily LSTM and CNN networks will be used in the design of proposed neural network architectures due to their efficiency in resp. capturing trends in sequential data and extracting features from images – which are both important given the dataset that will be used.

3 Data collection and preprocessing

The first step of any machine learning operation is collecting data. For this thesis, a simulated dataset was used as a substitute for the real data. This poses some challenges as the simulation data is captured in an ideal scenario and contains less motion parameters – only pitch and roll – compared to what will be available from the onboard sensors. This can cause the models to perform differently and have slightly different architectures due to the different input and output features they are limited to. In the following sections, the simulation dataset will be discussed in more detail. In addition, a brief overview is given on the real data collection process for comparison. The different parameters available in a dataset such as pitch and roll will be referred to as the *features* of the dataset.

3.1 Simulated data

During development, a simulated dataset was used from an ocean wave simulation made in Blender (Kaminskyi, 2019b). This dataset was made by Nazar-Mykola Kaminskyi and is publicly available through GitHub. For this simulation, a standard model of a vessel was used which floats on the simulated sea surface and moves along with the waves. Models trained on this data will be biased to this vessel's characteristics. A large, heavy vessel will behave different compared to a small, lightweight vessel. Therefore, it is expected that models will need to be retrained and re-evaluated on data captured from their specific target vessel, regardless of whether real or simulated data is used. The difference in performance of retrained and non-retrained models remains a topic for future work.

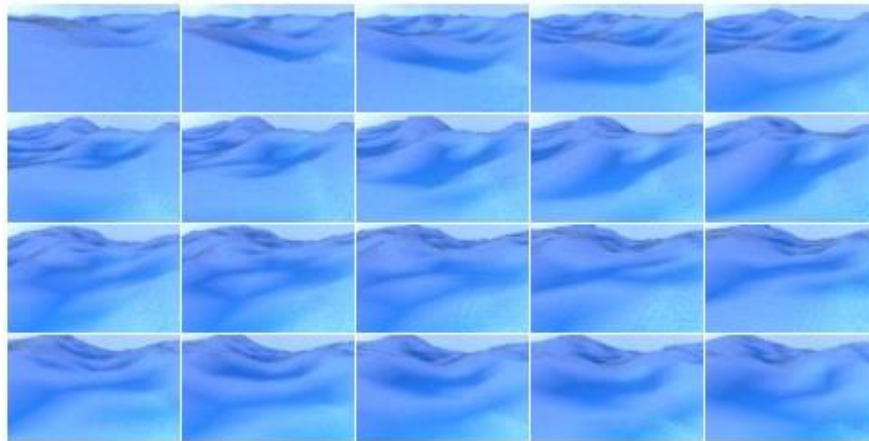


Figure 9: Generated images of incoming waves (chronologically ordered left to right, top to bottom)

The data captured from the simulation is from a perfect scenario, meaning that there are no obstructions or other objects in the images, the lighting is always the same and the camera is perfectly stabilized on the moving vessel. These are conditions that are unobtainable in reality. This can cause models that are trained on the simulation data to perform very well in simulation yet fail to meet desired expectations in a real-world scenario. To minimize this effect and close the gap between simulation and reality, data can be augmented to include more variation, or adjustments can be made to the simulation to make it more realistic such as including passing vessels in the frame. Finally, since the data is generated by a computer, it could be more predictable than naturally occurring measurements. A simulation can be very realistic, but behind the scenes there is still just an algorithm with parameters that a neural network might be able to learn very quickly.

The simulated dataset contains images of incoming waves (Figure 9) together with measurements of the ship's position at the time the image was captured. The data was generated in 540 different episodes, each episode containing 400 frames of data. Each frame contains one image together with the state of the vessel at the time of the image in the form of a pitch- and roll-value tuple. The frames were captured at two frames per second to minimize data overlap in consecutive

images. In its totality, this dataset contains 216.000 frames, which translates to thirty hours of simulated data. All episodes are structured equally. Images are named 0 to 399 and pitch and roll data points are saved in a separate *JSON* file. The pitch and roll couples are numbered 0 to 399 as well to easily identify which image corresponds to which tuple.

With this simulated data we only have access to pitch and roll as input data. This means that during testing in the simulation environment, heave cannot be predicted. However, with the neural networks that will be proposed later, this should not be a big issue for two main reasons. Firstly, heave follows a somewhat predictable pattern in regular conditions as illustrated by Figure 3. Because of this, it was assumed not necessary to regenerate the full dataset for just one extra output. It was also assumed that if the model performed well on pitch and roll, it should consequently perform well on heave. Secondly, adding extra input and output features to a neural network architecture, is not an expensive operation. It is highly likely that all models will have to be retrained anyways when switching from the simulation environment to a real-world environment to adapt to the new vessels' characteristics and the additional data available from the ZED-mini.

3.1.1 Simulation parameters

To simulate this data, different parameters were used to define the simulation environment. The effects of these parameters on the simulation were not tested as the used dataset was already generated with fixed parameters. However, assumptions can be made on how they would affect the simulation. The most important parameters will be discussed briefly.

As mentioned above, the images were taken at two frames per second by a virtual camera pointing to the front of the vessel. The position of the camera on the simulated vessel was set with the following parameters:

- Height: *5 meters*
- Rotation around x-axis: *76 degrees (slightly tilted downwards)*

The images were captured in color at a low resolution to decrease the memory requirements to load the dataset. The following resolution parameters were used:

- Height: *54 pixels*
- Width: *96 pixels*

3.1.2 Data augmentation

When a vessel is operating in open seas, the images of the incoming waves will generally be free of obstructions. However, when a vessel is close to land, other objects may be in front of the ship like other vessels and/or shorelines. To better evaluate performance in these scenarios, a small subset of the simulated images were edited and added to an augmented dataset. Using an image editing software, ink blots were manually drawn on the images to resemble external objects in front of the ship. The shape, size and colors of these blots were randomly chosen. In some cases, they are unrealistically large, but this was done to simulate a worst-case scenario so that performance can only improve in reality. Some of the augmented images are shown in Figure 10. To measure the effect on performance, models were trained on the normal images and predictions on the augmented sequence and corresponding normal sequence were compared.

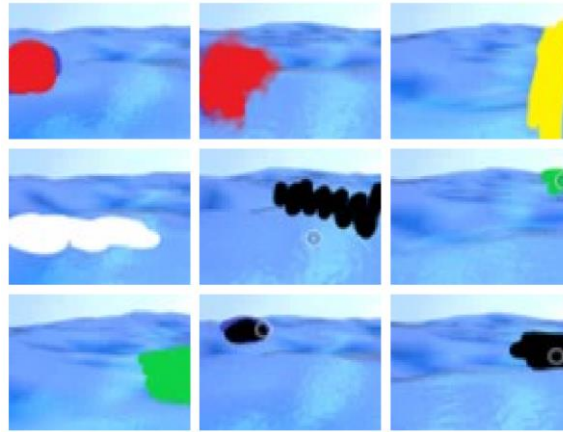


Figure 10: Augmented images

3.2 Real data

Eventually, after evaluation on the simulation set, some prediction models will be selected for deployment in a real environment. This means that they will need to transition from simulation data to real data. To provide the real data, the target vessel is equipped with multiple sensors. However, due to the nature of sensory data, using real data inherently comes with some challenges. In the following two sections, the onboard sensor is discussed, and some challenges are mentioned.

3.2.1 Sensors onboard the ASV

The target vessel will be equipped with a ZED-mini stereo IMU camera (Figure 11) (Stereolabs, Paris, France). This is a multipurpose sensor that can capture video from its two cameras and positional measurements from its Inertial Measurement Unit (IMU). The combination of these two sensors allows the ZED-mini to accurately describe the state of the sensor and its surroundings. The IMU has two built-in motion sensors: an accelerometer and a gyroscope. These provide a real-time data stream at 800Hz of the movement of the sensor in six degrees of freedom.

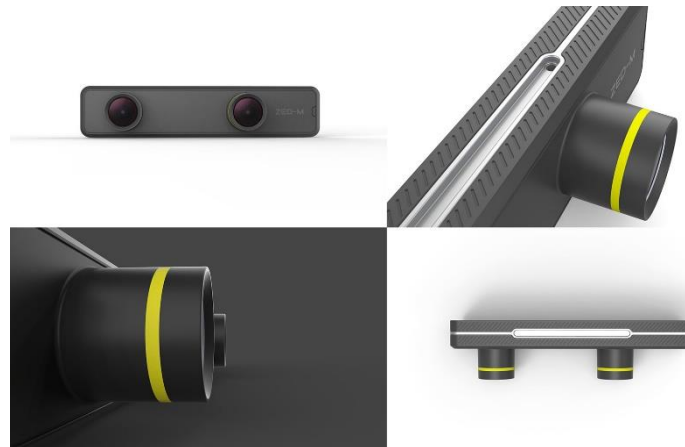


Figure 11: ZED-mini IMU stereo camera (Stereolabs, Paris, France)

The two forward facing lenses on the ZED-mini provide stereo video and images. This can be used to map the objects in front of the ZED-mini in a three-dimensional space. A stereo image is a combination of two separate images that are captured from two slightly offset point-of-views (PoV) like the two lenses on the ZED-mini. These two PoVs imitate the left and right human eye and create a perception of depth when the two images are fused together to create one stereo

image. This process is called stereoscopy. In computer vision, these two images can be compared to each other to extract three-dimensional information such as object distance from two dimensional images.

3.2.2 Sensory data challenges

The real data will be captured and fed into the model at real-time. This means that the data stream will have to be cleaned and filtered. As mentioned above, numeric motion parameters such as pitch and roll, are captured by the ZED-mini at 800Hz. This means that every second, 800 data points are measured. At this frequency, consecutive measured values contain a lot of overlapping data where minor change is happening between values. Therefore, the data stream will have to be reduced to minimize this data overlap and avoid overloading the prediction model with unnecessary and noisy data. Similarly, the video framerate from the stereo camera should be reduced to minimize data overlap between consecutive images. The video footage will also have to be compressed to reduce the memory and computational requirements needed to process a sequence of images.

(From this point in the thesis and onward, only the simulation data is used due to unavailability of real data)

3.3 Data pre-processing

Before data can be loaded into the model, it needs to be processed. This pre-processing of data is necessary to enhance to performance of the models and ensure that the model operates correctly. Due to a simulation dataset being used, less pre-processing is required compared to real data since the data is collected in a predefined format.

3.3.1 Data cleaning

The first step of data pre-processing is data cleaning. In this step, data points are checked for inaccuracies, anomalies and corrupted values. Odd data points are corrected or removed, and unnecessary features are dropped from the dataset.

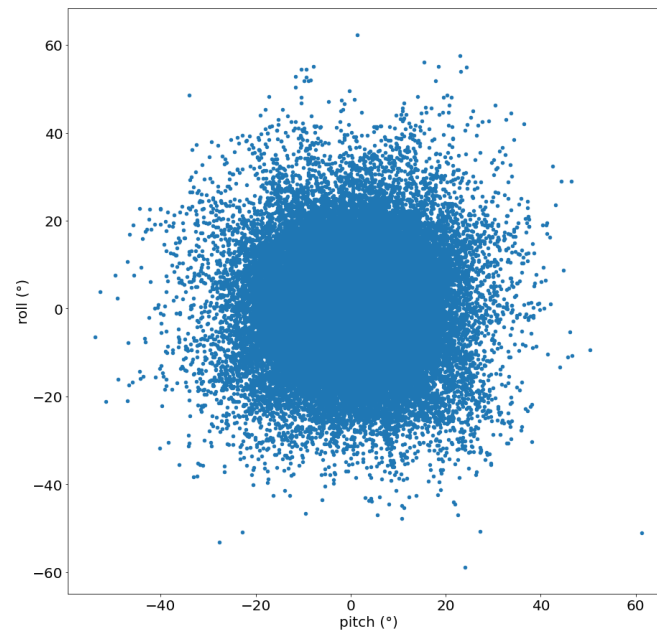


Figure 12: Scatterplot of pitch and roll

To clean the simulated data, all episodes were evaluated to contain 400 images and 400 pitch and roll couples. If this was not the case, there might have occurred an error during simulation at which point the validity of the sequence is infringed. Next, all pitch and roll values were audited to not contain any odd values such as un-numerical values or extreme outliers. This was done with `Pandas.isnull().values.any()` and `Pandas.DataFrame.plot.scatter()`. The scatter plot (Figure 12) shows the

two-dimensional layout of pitch and roll. The data is mainly concentrated in the middle with few outliers and no sign of skewing. Further analysis of the distributions will be conducted in 3.4 data analysis.

When real data is used, comprehensive cleaning will be required to filter out noise from the sensors and excessive outliers from inaccurate readings.

3.3.2 Data reduction

Because the data was generated by a simulation, values and images are captured in a desired format. This means that little to no data reduction is necessary. All images were captured with equal dimensions that are small enough to eliminate the need for additional compression or cropping but large enough to keep an adequate level of detail. The small dimensions also allow for lower computational requirements. The data was simulated at two frames per seconds (fps) or 2Hz in assumption that this is a good trade-off between minimizing overlapping data in images without introducing aliasing on the pitch and roll signals. The frequency of the simulated data can be reduced to a lower fps by dropping a certain number of frames between each loaded datapoint, however this was performed in this thesis to preserve maximum accuracy.

Determining the optimal sample frequency for pitch and roll depends on an accuracy trade-off and environmental parameters. Higher sample frequencies increase accuracy but introduce more overlapping data in images. Additionally, it may introduce *landing window noise*. This refers to minor changes that have negligible impact on the landing window but may negatively affect the learning process as the model tries to learn this noise. Lower sample frequencies decrease the amount of processing needed but decrease signal detail and may even introduce aliasing. The inertia of the vessel at hand plays a significant role as well. Large, heavy vessels move a lot slower compared to smaller, lightweight vessels. Because of this, the latter will require a higher data sampling frequency to compensate for its higher mobility. The period of the waves may also play a part in determining the optimal sampling frequency. However, according to Stewart H. Robert, ocean waves remain within frequency of approximately 12 to 24 waves per minute – resp. 0.2Hz and 0.4Hz (Stewart, 2008). Using this information, it was concluded that 2Hz is adequate for even the highest wave periods.

3.3.3 Data normalization

During the aggregation of data in a neural network, input features need to be normalized. When one feature contains values in a range of $[0, 100]$ and another feature contains values in a range of $[0, 5]$, the first feature will have a much larger impact on the result. For this reason, all data should be rescaled to one same domain.

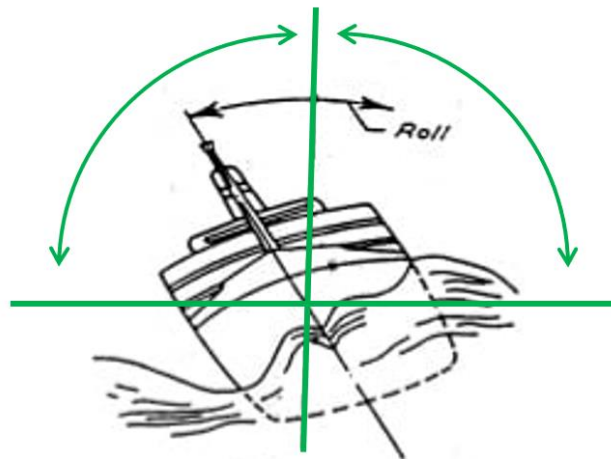


Figure 13: Maximum range of rolling motion for ships

Pitch and roll are normalized with min-max feature scaling. This method rescales values relative to their absolute maximum and minimum within a $[-1, 1]$ range. Since pitch and roll define the tilting of a ship, the absolute theoretical minimum and maximum were chosen to be -90° and 90° (Equation 2.1). If a ship pitches or rolls beyond these values, it

will capsize (Figure 13). A normalization domain of $[-1, 1]$ was chosen due to its equal symmetry around zero compared to the pitch and roll values in their theoretical range on motion.

$$x_{normalized} = \frac{x_{original} - (-90)}{90 - (-90)} \cdot 2 - 1 = \frac{2 \cdot (x_{original} + 90)}{180} - 1 \quad (2.1)$$

Images need to be normalized as well. Each image is three-dimensional matrix containing three values – red, green and blue channel - for each pixel in the image across the height and width. This means that for every pixel, three values need to be normalized. The three channels indicate the amount of red, green and blue in each pixel. Each value is an 8-bit integer with a minimum of zero and a maximum of 255. To normalize them, a similar min-max feature scaling function was used as for the pitch and roll values (Equation 2.2). Values are normalized to the same $[-1, 1]$ domain.

$$p_{normalized} = \frac{p_{original} - 0}{255 - 0} \cdot 2 - 1 = \frac{2 \cdot p_{original}}{255} - 1 \quad (2.2)$$

3.4 Data analysis

The simulation data was briefly analyzed to have a better understanding of the results. Very in-depth analysis of the input data is not necessary for deep learning applications. Deep learning is a form of unsupervised learning, where the model itself determines the importance of each feature. In comparison, supervised learning problems require thorough data analysis to find out which feature(s) have most impact on the desired output feature(s) and should be included in the models' input. The goal of the data analysis in this section is, in the first place, to better comprehend how pitch and roll behave so a more profound interpretation of results can be achieved. And in the second place, to discover potential interactions or correlation between pitch and roll.

3.4.1 Statistical properties

First off, basic statistical information was extracted from all pitch and roll data points from all episodes. With the built-in function of Pandas `pd.DataFrame().describe()`, this is easily achieved. The results are shown in Table 4.

PROPERTY	PITCH	ROLL
count	216.000	216.000
mean	0,0678°	0,303°
standard deviation	6,611°	7,022°
minimum	-53,721°	-58,846°
25%	-2,761°	-2,572°
50%	0,0262°	0,230°
75%	2,876°	3,187°
maximum	61,328°	62,217°

Table 4: Statistical information for pitch and roll in simulated dataset

The count refers to the amount of datapoints analyzed, which is equal to the 540 episodes each containing 400 frames. The quartile percentages at 25% and 75% show a similar interval for both pitch and roll. Additionally, pitch and roll respectively are centered around approximately zero degrees. This is concluded based on their nearly identical mean and median, meaning that 50% of their values are lower than zero and 50% are higher. This is expected behavior of a normal floating vessel that naturally wants to remain upright. Lastly, it was observed that roll had a slightly higher standard deviation, which indicates that its values generally deviate further from the mean than those of pitch. From these properties, it can be concluded that pitch and roll both behave similarly on a numerical basis across all fields. However, this does not necessarily mean that they physically behave similar. The minimum and maximum values give a good estimate of the interval in which the predicted values should remain. They also provide a means to normalize error values relative to these boundaries. When all data occurs in a $[-60^\circ, 60^\circ]$ interval, an error of three degrees is a 5% error relative to the absolute maximum which is really good. If all data lies in a $[-10^\circ, 10^\circ]$ interval, an error of three degrees is a 30% relative error which is very high. However, these minimum and maximum don't tell the full story.

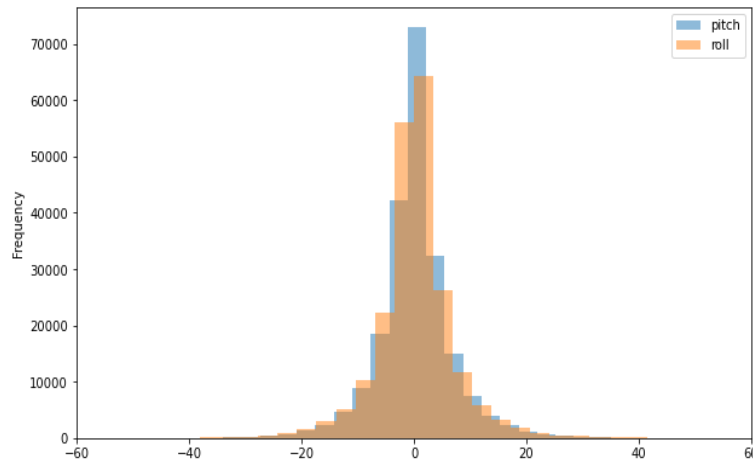


Figure 14: Simulated Pitch and Roll distributions

The distributions of pitch and roll are shown in Figure 14. Both of them are remarkably similar and show similar characteristics to a normal Gaussian distribution. Intuitively, it can be assumed that the majority of the datapoints are located in a $[-20^\circ, 20^\circ]$ interval. Compared to the properties of normal distribution, this assumption is justified when the 65%-95%-99.7% rule is applied. This rule determines that given a mean μ and a standard deviation σ of a normally distributed dataset, respectively 65%, 95% and 99.7% of all datapoints lie within a $\mu \pm \sigma$, $\mu \pm 2\sigma$ and $\mu \pm 3\sigma$ interval. If this rule is applied with the mean and standard deviation from Table 4, the 99.7% interval can be calculated and are as follows:

- Pitch: $[-19,74^\circ; 19,86^\circ]$
- Roll: $[-20,70^\circ; 21,30^\circ]$

These intervals confirm that nearly all data points – 99.7% – lie within the above intervals. Because of this, errors should be taken relative to these interval boundaries rather than the maxima and minima of the full dataset. The 99.7% intervals represent almost all data whereas the minima and maxima could be two extreme outliers.

3.4.2 Correlation and interactions

Secondly, the correlation of pitch and roll was reviewed to assess the influence of roll on the prediction of pitch and vice versa. This was done to decide if having both motion parameters as input is necessary for the predictions on one or the other feature individually.

A commonly used method for analyzing correlation is a correlation matrix. This is a symmetric matrix where the number of rows and columns equals the number of features. Each cell in the matrix contains a value equal to the correlation of the features of the corresponding row and column. These values range from -1 to 1 where -1 indicates a strong negative correlation, 1 a strong positive correlation and 0 indicates that there is no correlation. From the matrix in Figure 15, pitch and roll have a correlation close to zero, which indicates little to no correlation.

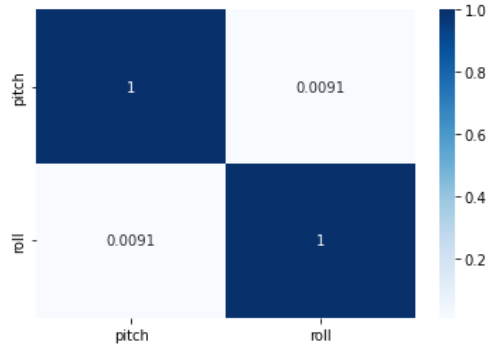


Figure 15: Correlation matrix for pitch and roll

With this result, it can be assumed that one feature has negligible contribution to the prediction of the other feature. Providing additional features besides the predicted feature can therefore be seen as overhead for the model. However, this assumption is only based on numerical values from a simulation containing only pitch and roll. According to this research (Nayfeh et al., 2012), pitch and roll are coupled nonlinearly when their frequencies are in a ratio of two to one. On top of this, more correlated features could be introduced with the additional input features from the ZED-mini sensor.

With this information, it was concluded that standalone feature correlation was not enough evidence to rule out other possible physical interactions. Because of this, all models were designed to ingest all available features instead of dropping some due to low correlation with the predicted target features. In the case of the simulation data, this meant that models were trained and evaluated to predict both pitch and roll from an input sequence also containing both pitch and roll.

3.5 Data loading

Data loading refers to the process of creating input-output sequences, splitting the data and creating batches in the correct format for network training. For each model, the inputs and outputs will be sequences with varying lengths. The following sections describe step-by-step how data loading was performed. First, sequences from coupled inputs and outputs are generated. Next, all generated sequences are split into two subsets for training and testing the models. Finally, the data is wrapped in PyTorch DataSet and DataLoader objects which convert the data into Tensors and create batches.

3.5.1 Sequence creation

The simulated data was generated in different episodes. Each episode contains 400 frames that chronologically follow each other as previously shown in 3.1, Figure 9. However, the episodes themselves do not follow each other chronologically. Different episodes are generated in different simulation sessions and can therefore not be seen as continuous. Because of this, the data must be sequenced correctly to maintain chronological order and not include data from different episodes. In short, all frames in the input and output sequence, should always be from the same episode.

For this reason, a dedicated function was designed to create sequences from this non-continuous dataset. One sequence consists of two parts, an input sequence and an output sequence. The input sequence is the data used to predict the output sequence. The length of the input and output sequence can be passed to the function as parameters to easily create different sequence configurations. This is necessary as the length of the input and output sequences will have a large effect on the performance of the model and will be thoroughly examined. In addition, the function also accepts parameters to define the input and output features when different datasets are being used that contain more features.

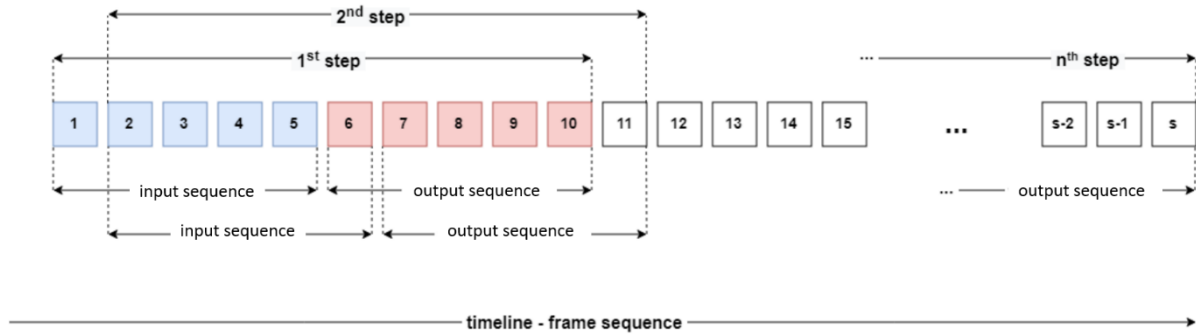


Figure 16: Sequence creation with moving input and output sequence

In Figure 16, the sequence creation process is illustrated on an episode with the number of frames being equal to s and the input and output sequences having both a length of five. In the first step, the function takes the first five elements which form the input sequence used to predict the next five elements: the output sequence. These two sequences are then coupled together and added to a list with all sequences. In the second step, the starting point for the sequences is moved to the next element and the same procedure is repeated, extracting and coupling the input with the output. This process is repeated until the end of the episode is reached. At this point, a new episode is loaded on which the cycle is repeated. The result is a list of all possible input-output sequences that only contain consecutive datapoints from within a same episode.

3.5.2 Train-test splitting

After the sequencing is done, the list of all coupled input-output (IO) sequences is split up in two subsets: one for training the model and one for testing the trained model's performance. This splitting is necessary to properly measure how well a model can generalize what it learned to new data.

When a deep learning neural network is trained on a dataset, it will optimize its performance as much as possible on this training dataset. If all available data is used for the training dataset, the model will have consumed all IO-sequences at least once, leaving no unseen data for testing. Therefore, in order to evaluate how well the model generalizes to new data, a second dataset is needed. This test dataset contains data that the model has not seen before and is not optimized towards, unlike the training data. As a result, if the model generates predictions on this second dataset, its true performance on new data can be evaluated. After this evaluation, the training parameters can be adjusted, and the model can be retrained to further optimize performance.

To achieve this split, elements from the list of coupled IO-sequences are randomly selected and assigned to one of the two subsets with an 80%-20% distribution between training and testing. This randomization has positive effect on the model's performance because it mitigates possible trends in the initial dataset. For example, due to pure coincidence it is possible that the first ten IO-sequences are from calm seas and the next five are from rough seas. If this dataset is split using normal forward iteration over the sequences - adding the first ten IO-sequences to training and the next five to testing - the model will perform very poorly as it never had the chance to learn the rougher sea's trends. Thus, the data is split with random selection. Additionally, since sequential data is used, consequent IO-sequences will have some data overlap. Figure 16 shows this more clearly: the sequences from the 1st and 2nd step are very similar. When random selection is used, the effect of this data overlap is minimized and IO-sequences in one batch contain the maximum amount of unrelated and differentiated data.

3.5.3 PyTorch batching

Finally, after the pre-processed data is sequenced and split in two subsets, IO-sequences are divided in batches and prepared in the specific format for the neural network. Since all neural networks were implemented with PyTorch, it was most convenient to load the data with PyTorch objects. The two different datasets - training and testing - are each

wrapped inside a PyTorch DataSet object. This class stores the IO-sequences in an iterable and indexable data structure that can be used by a PyTorch DataModule. The DataModule stores the two DataSets and makes them accessible via multiple DataLoader objects, one for each DataSet. Each DataLoader divides its DataSet in small batches for batch processing. The size of the batches is an important hyperparameter and determines how much data the model will consume before updating its internal parameters. Batch size and its effects will be discussed more in detail in section 4.2.4.

While only two datasets were used during development, one for training and one for testing, three DataLoader objects were created. One for the training dataset, and two for the test dataset. The latter two used the same dataset but one of them used a batch size of one (test DataLoader) while the other used a batch size equal to the training DataLoader (validation DataLoader). By doing so, the test DataLoader could be used to evaluate the performance of the models on a sequence per sequence basis instead of only having access to batches of multiple sequences. This resembles the real-time environment where predictions are made as data is measured, one at a time. Additionally, the validation DataLoader could be used to measure generalization and overfitting during training.

3.6 Summary

This chapter discussed all subjects related to data collection and processing. The properties of the simulated data and real data were introduced. Both images and ship motion data were collected. However, due to the real data being unavailable for a majority of the project's duration, there was insufficient time to work with real data and only the simulation data was used in this thesis. Some challenges were discussed when transitioning from simulated data to real data. These will have to be further assessed in future work. Besides this, it was concluded that 99.7% of all pitch and roll datapoints lie within an approximate window of $[-20^\circ, 20^\circ]$. This is important for profound interpretation of the results. In addition, there was no correlation found between pitch and roll. However, studies have shown that pitch and roll can interact with each other and therefore it was concluded that despite the low numerical correlation, pitch and roll should always be used together as input data. Lastly, the data loading process was discussed. Data is first sequenced into corresponding input-output pairs. These sequences are then split into two separate datasets for training and testing. Finally, a PyTorch DataSet and DataLoader object are used to divide the data of these different datasets in batches and feed it to the models in the right format.

4 Model designs

In this chapter, all model architectures and parameters will be discussed. An iterative process was used where each consecutive model increases in complexity in order to potentially increase performance. As discussed in the introduction, ensemble models combining different neural network architectures will be used to obtain optimal performance on the available data – images and numerical data. To visualize the model designs and input-output dimensions, illustrations were made and are added for each model. In addition, a table is provided with the specific parameters of each model. To identify different models based on their architecture, the following naming conventions are introduced. This nomenclature will also be used in the parameter tables and on the architecture illustrations.

Notation	Explanation
P – R – PR	Resp. Pitch – Roll – Pitch AND Roll
IMG	Notes that images are used as input
LSTM	An LSTM architecture was used
CNN	An CNN architecture was used
FC	Fully connected linear layer
Single-/Multi-step	Prediction of a single time-step / sequence of time-steps
N	Number of frames in the input sequence
M	Number of frames in the predicted output sequence

Table 5: Architecture naming conventions and their explanations

4.1 Single-step model

The first network that was created was a single-step stacked LSTM architecture. This model uses a sequence of pitch and roll inputs and predicts one future time-step ($M = 1$) - hence 'single-step' - of either pitch, roll or both. This model was designed to get familiar with the workflow of creating, training and testing neural networks with PyTorch. For simplicity, image data was omitted to allow for a simple model structure with relatively few parameters that can be trained quickly. Two variants of this model were trained, both having identical architectures with the only difference being that one variant can predict pitch *and* roll at the same time while the other can only predict *either* pitch or roll based on what it was trained for. This is reflected by the number of neurons in the output layer shown in Table 6, where out^* is equal to one or two neurons for resp. the single (P or R) or the dual output (PR) variant.

Input	Sequence of PR at ($2 \times N$)					
Name	Layers	Number of layers	Hidden size	Dropout	Input	Output
LSTM	LSTM	2	128	0.2	$2 \times N$	2×128
Regressor	Linear				1×128	$1 \times out^*$
Output	Single value for either P, R or PR					

Table 6: Parameter table for single-step models

The single-step model consists of an **LSTM architecture** followed by a linear fully connected regressor layer. The general idea behind this design is that the LSTM computes a hidden vector on the sequence which represents the short-term memory of the LSTM. This hidden vector is then used to calculate the predicted output(s). To compute the hidden vector, a two-layered or **stacked** LSTM is used. In this configuration, the output of the first LSTM is passed to the inputs of a second LSTM. This configuration was chosen based on the research of Cui Z. in which the performance of both single layered and stacked LSTMs were compared in a similar forecasting context (Cui et al., 2020). The conclusion was that the stacked LSTM performed better than its single layered counterpart. Both LSTM layers use 128 hidden LSTM cells (neurons) and have a

dropout of 0.2. Dropout is a function that randomly disables certain neurons in the network. The value for dropout represents the chance of any given neuron to be disabled. Using dropout helps to reduce overfitting during training and thus helps improve stability and performance (Srivastava et al., 2014). Dropout is applied between the two LSTM layers. The concept of overfitting is discussed later on in 5.1. The last layer is a linear fully connected layer, which aggregates all the hidden features into a single output feature. Notice that the hidden output of a layered LSTM is a three-dimensional tensor with the following shape: (number of layers, the batch size, number of hidden neurons). It is built by appending the two-dimensional hidden vectors of each layer one after the other in chronological order, numbered by their index. For the aggregation in the linear layer, only the hidden state from the last layer is used, hence the notation *hidden[-1]*. The single-step stacked LSTM model architecture is illustrated in Figure 17 with the single output variant on the left.

Due to its simplicity, this model also served as a testing ground for some important recurring parameters such as the size of the hidden layer and learning rate as well as for different configurations of activation functions. The model was trained with different values for these parameters to form a conclusion on what values worked well. From the results of these preliminary tests, later model designs could use similar hidden sizes and learning rates in assumption that they will behave similarly. This way, more efforts could be put in testing other parameters such as input and output sequence lengths. A more detailed discussion on these parameters is provided in 5.1.

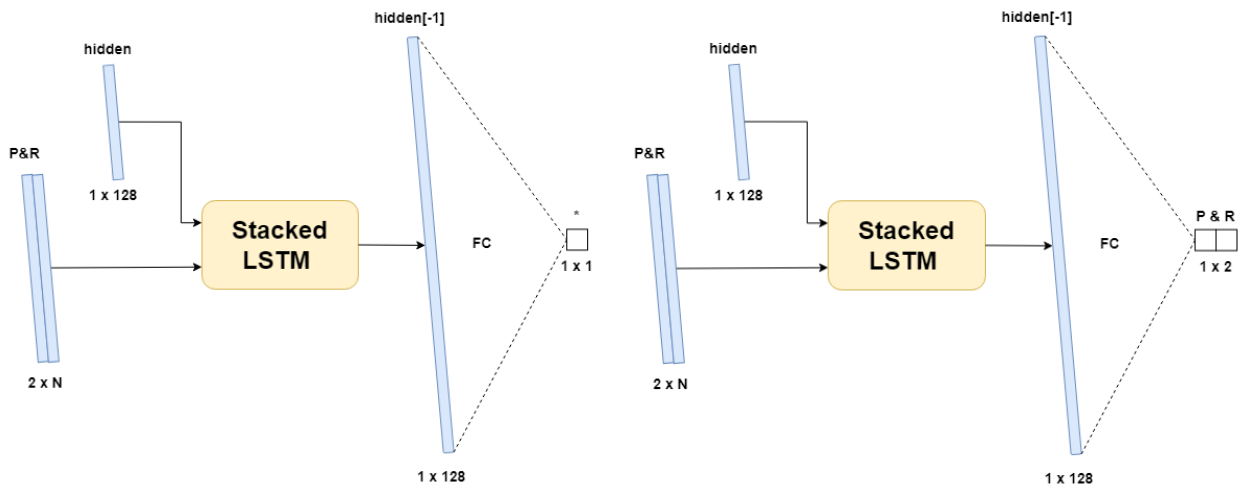


Figure 17: Single-step model architecture variants: single output (left), dual output (right)

4.2 Multi-step models

The next step in development was to design models capable of predicting a multi-step sequence of pitch and roll. In addition, images are also introduced as additional input. The combination of these two expansions results in more complex and more capable model architectures. In the next sections, four new multi-step model architectures are introduced.

4.2.1 Encoder-Decoder LSTM

The first model designed for multi-step predictions was an expanded version of the single-step model. The two-layered LSTM was replaced with **two single-layered LSTMs** in an encoder-decoder configuration. This design was inspired by a similar model of Kaminskyi, and by a model designed by Brownlee for power usage time series forecasting (Brownlee, 2018). This architecture was chosen to be implemented first because of its relatively simple design and because of its low expected performance. It performed worst out of all tested models in Kaminskyi's research, therefore it should be a good starting point to iteratively improve upon.

When tasked with multi-step prediction, the idea is that the first LSTM encodes a latent vector together with a hidden short-term memory vector which are then decoded by the second LSTM to generate an output sequence. With this model, it was possible to evaluate how well a neural network could predict pitch and roll without having the images of incoming waves. If this model would perform similarly to one that uses images as an additional input, the conclusion could be made that, if necessary, the images could be omitted. In that case, this model would provide a more lightweight solution, requiring only numeric data.

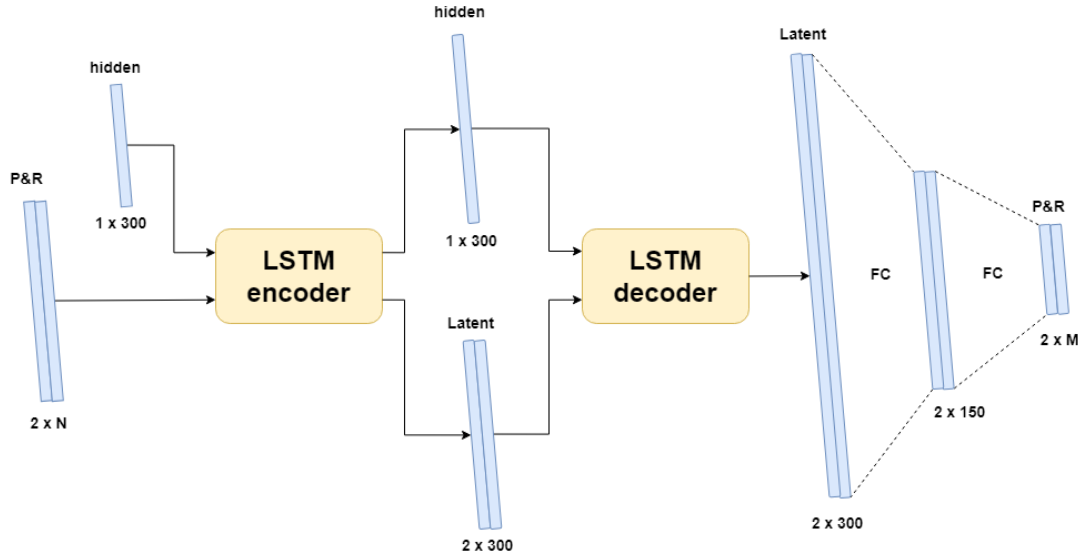


Figure 18: Encoder-Decoder LSTM architecture

Figure 18 shows the architecture of the implemented model. As mentioned above, two LSTMs are used together where the first one acts as an encoder followed by a second decoder LSTM. The first LSTM receives the input sequence of pitch and roll values and has a hidden size of 300. When the data is passed through this network, an encoded latent vector is output together with the hidden state of this first LSTM layer. The second LSTM layer uses this hidden and latent vector and decodes it into a new latent vector. Finally, the latent vector from the decoder is passed through two linear layers that aggregate the vectors into a sequence for pitch and roll.

Input	Sequence of PR at ($2 \times N$)					
Name	Layers	Number of layers	Hidden size	Dropout	Input	Output
LSTM component						
Encoder	LSTM	1	300	-	$2 \times N$	2×300
Decoder	LSTM	1	300	-	2×300	2×300
Linear component						
FC 2	Linear				2×300	2×150
FC 2	Linear + tanh				2×150	$2 \times M$
Output	Sequence of PR at ($2 \times M$)					

Table 7: Parameter table for Encoder-Decoder LSTM

The parameters for the Encoder-Decoder LSTM network are shown in Table 7. The LSTM hidden vector sizes are increased to 300. This increase was based on test results from the single-step model, showing better performance when a larger than 128 hidden size is used. The specific value of 300 was chosen equal to Kaminskyi's model in order to get a more equal comparison between both designs. Lastly, a second linear layer was added to the end of the model to allow for more depth when aggregating the hidden vector down to the predicted output. The predicted output is activated by a tanh to limit the models output to the normalized domains for pitch and roll, i.e., $[-1, 1]$.

4.2.2 Sequential CNN

After the single- and multi-step LSTM oriented models – that only use pitch and roll inputs – convolutional neural networks (CNN) were introduced. With their introduction, images could also be efficiently processed and used to make predictions. Three models were designed using CNN architectures with the first one being a sequential **CNN architecture**. On one hand, this simple model was created to familiarize with the new CNN network architecture, its parameters and the image data loading process. On the other hand, this model was created to evaluate performances when **no LSTM module** is used. Without an LSTM module, performance of this model is expected to be lower than other models. But it might still be better than previous LSTM models that only use numeric data. Additionally, LSTM networks have a high inference time due to their complexity (Mealey & Taha, 2018). Because of its absence in this model, lower inference time is expected which might make up for possible loss in performance.

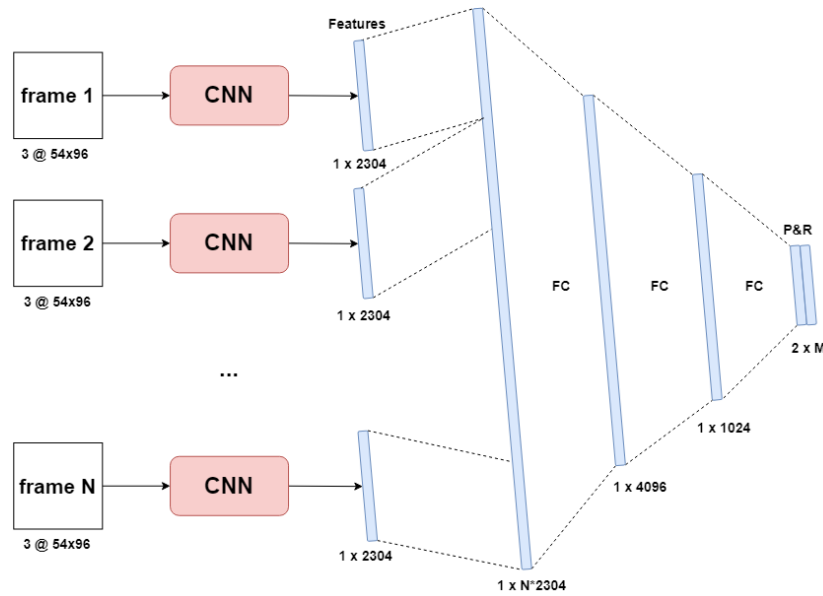


Figure 19: Sequential CNN neural network architecture

Each image in the input sequence is processed individually by a CNN module. The resulting N outputs are then flattened into a one-dimensional vector which describe the extracted features of the image. These feature vectors are then concatenated into one single large vector which serves as the input layer for the first linear layer. Because the feature vectors are appended in order, the sequence order is maintained. Three consecutive linear layers then aggregate this larger vector into smaller vectors and eventually into an output sequence of pitch and roll. The architecture for this model is illustrated above in Figure 19, parameters of each component are found in Table 8.

The CNN component consists of three consecutive convolutional layers that are used to extract features from the images. The first convolutional layer uses a 5x5 kernel while the next two layers use a 3x3 kernel. The channels follow a doubling pattern where each layer doubles the number of channels of the previous layer. Each layer is followed by a rectified linear unit activation and max pooling. This was done following the conventions of CNN network structures and to reduce parameters. The kernel sizes were based on the CNN auto-encoder architecture from Kaminskyi's work as it showed good feature extraction performance and was used in his best model. The size of the linear layers follow a decreasing trend to slowly aggregate the large vector into predicted outputs, which are activated by a tanh. The tanh was especially necessary for this model as it would otherwise predict extreme values on the first couple of training batches. This is normal behavior as the model is still learning. However, the tanh helps to better visualize training loss without affecting performance.

Input	Sequence of N images at ($N \times 3 \times 54 \times 96$)						
CNN component							
Name	Layers	Out channels	Kernel Size	Stride	Padding	Input	Output
Conv 1	Convolution	8	5x5	1	2	3 x 54 x 96	8 x 54 x 96
	ReLu	-	-	-	-		
Pooling 1	Max Pooling	1	2x2	2	1	8 x 54 x 96	8 x 27 x 48
Conv 2	Convolution	16	3x3	1	1	8 x 27 x 48	16 x 27 x 48
	ReLu	-	-	-	-		
Pooling 2	Max Pooling	1	2x2	2	1	16 x 27 x 48	16 x 13 x 24
Conv 3	Convolution	32	3x3	1	1	16 x 13 x 24	32 x 13 x 24
	ReLu	-	-	-	-		
Pooling 3	Max Pooling	1	2x2	2	1	32 x 13 x 24	32 x 6 x 12
Linear component							
FC 1	Linear					1 x N*2304	1 x 4096
FC 2	Linear					1 x 4096	1 x 1024
FC 3	Linear + tanh					1 x 1024	1 x M*2
Output	2 x M						

Table 8: Parameter table for sequential CNN model

4.2.3 CNN LSTM single input

Now that the two main proposed neural network architectures – CNN and LSTM – were both individually implemented in different models, an **ensemble model** was made composed of both architectures. Many of the design concepts of the previous sequential CNN model were kept the same for this model. The consecutive input images are processed individually by a **CNN** into N feature vectors. These different feature vectors are then concatenated into a large single vector. However, this vector is then processed by an **LSTM** encoder instead of a linear layer. Since the sequence of the images is kept intact during the concatenation as previously mentioned, the LSTM can use its long- and short-term memory properties and should achieve higher performance than linear layers.

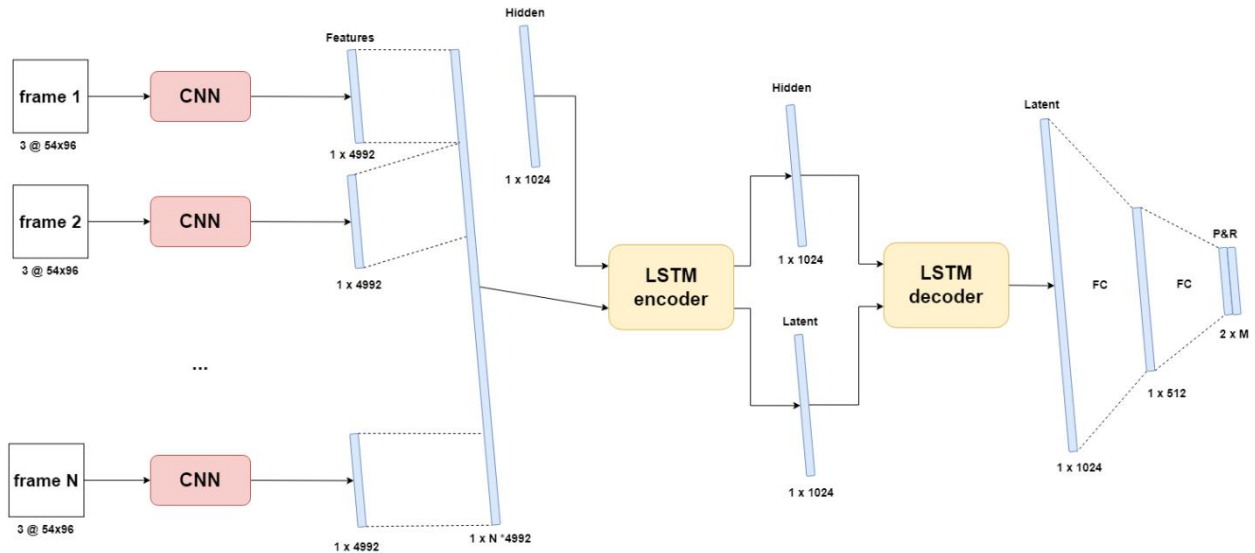


Figure 20: CNN LSTM single input architecture

The first LSTM encodes the large feature vector into a latent and hidden vector. These vectors are then passed to a second decoder LSTM that computes a new latent vector which is then aggregated into a sequence of pitch and roll predictions by two linear layers. Note the **single input** in the model's name to indicate that **only images are used as input** for this model.

A second model was designed as well that allows for dual input (images and PR). This way, the performance of all three input configurations could be measured: pitch and roll only, images only and both input types.

The parameters for the single output CNN LSTM model are shown in Table 9. The CNN component is identical to the one from previous model. However, the last convolutional, ReLU and Max Pooling layers were removed. This was done in order to reduce the complexity and the number of parameters in the model, and thus provide a more lightweight solution. The LSTM component is set up equal to the Encoder-Decoder LSTM architecture from 4.2.1, except for the increased hidden size to compensate for the much larger input vector.

Input	Sequence of N images at ($N \times 3 \times 54 \times 96$)						
CNN component							
Name	Layers	Out channels	Kernel Size	Stride	Padding	Input	Output
Conv 1	Convolution	8	5x5	1	2	3 x 54 x 96	8 x 54 x 96
	ReLu	-	-	-	-		
Pooling 1	Max Pooling	1	2x2	2	1	8 x 54 x 96	8 x 27 x 48
Conv 2	Convolution	16	3x3	1	1	8 x 27 x 48	16 x 27 x 48
	ReLu	-	-	-	-		
Pooling 2	Max Pooling	1	2x2	2	1	16 x 27 x 48	16 x 13 x 24
LSTM component							
Name	Layers	Number of layers	Hidden size	Dropout	Input	Output	
Encoder	LSTM	1	1024	-	1 x N x 4992 (+2)	1 x 1024	
Decoder	LSTM	1	1024	-	1 x 1024	1 x 1024	
Linear components							
FC 1	Linear					1 x 1024	1 x 512
FC 2	Linear + tanh					1 x 512	1 x M*2
Output	2 x M						

Table 9: Parameter table for CNN LSTM single input and dual output (red)

4.2.4 CNN LSTM dual input

The final model is a variation on the previous model which allows it to use both **numeric data and images (PR-IMG) as input** – hence “dual input”. It was built based on the architecture of the previous CNN LSTM single input model. The main difference is that pitch and roll values for each frame – a frame being an image and its corresponding PR values – are appended at the end of each image's feature vector before they are concatenated into one large vector. This way, both the image and PR sequence order are preserved for the LSTM. Correct normalization is of utmost importance with this model as one type of data may overshadow the other if they are not normalized to the same window.

Expectations for this model are high as it has all available data at its disposal to make as accurate predictions as possible. However, this model requires the most amount of processing, so inference time will probably be higher than other models. This is mainly due to the process of building the feature vectors and appending the pitch and roll tuples to them. This is an expensive operation in terms of computing resources and might cause this model's inference time to be too high. The architecture is illustrated in Figure 21. Since the model's only difference is found in the way it builds the image feature vectors, the parameters of the CNN and LSTM module are unchanged and Table 9 still applies. The only difference is the size of the feature vector, which is increased by two for each frame. This change is marked in red. Additionally, the values of the image feature vector are activated with a tanh. This is done to rescale them back into the domain of the PR-values that are appended. If this was not the case, the values of the appended PR-pairs might be overshadowed by the potentially larger values in the feature vector.

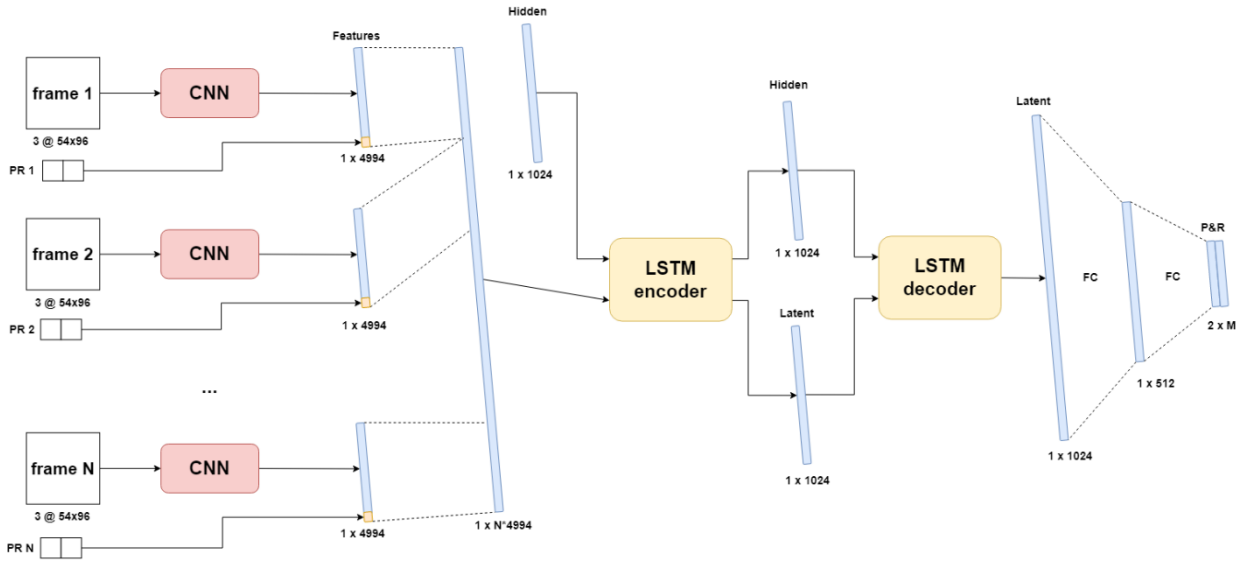


Figure 21: CNN LSTM dual input architecture

4.3 Summary

In this chapter, all proposed models and their parameters were discussed. An iterative process was followed where each consecutive model increases in complexity and capabilities to process different datatypes. First, a single-step model was implemented to acquire experience with PyTorch and deep learning. This model is limited to predicting only one future time-step. Two variants were trained, one for predicting either pitch or roll and one to predict both. Due to its simplicity and low parameter count, this single-step model also served as a platform to preliminary test different configurations of recurring parameters such as learning rate and hidden size, to find optimal values for future models. Next, four multi-step models were implemented. These are models that are able to predict ordered multi-value sequences of pitch and roll. Each model was designed with a different approach based on what data is used as input and which architectures are used. One model only uses pitch and roll as input, two models only use images, and one uses both. Both CNN and LSTM architectures are used either separately or in combination with each other in ensemble models. This variety in model architecture and input types allows for distinctive conclusions on which architecture works best and what factors contribute most to optimal performance.

5 Testing environment

To fairly evaluate each model's performance and match it to the criteria, a testing environment is needed. This environment determines what parameters will be used to train the models and which metrics will be used to measure and compare performance. In the following sections, all components that make up the testing environment are discussed.

5.1 Gradient descent algorithms

In supervised deep learning problems, each neuron in the network updates its optimal weight and bias by comparing the predicted output with the ground truth. The model receives input-output sequences and tries to learn the relation between the two. A metric is used to compare predicted output and ground truth. This metric – commonly referred to as the cost or loss function – is chosen based on the nature of the problem. Some examples are the mean square error (MSE) for regression problems or cross-entropy loss for classification problems (Wang et al., 2022a). The goal is to minimize this error during training. Each iteration, the model updates its parameters to move, *descent* in the direction of this minimum. The direction in which the minimum lies, is found by calculating the derivatives of the loss function for each neuron. However, since the impact of both weight and bias needs to be considered, both partial derivatives are calculated. This is done for all weights and biases. The result is a gradient that can be thought of as a multi-dimensional hyperplane with multiple local minima and one global minimum. To solve for the gradient, the model iterates through all data and computes the gradient in the current configuration of weights and biases. It then utilizes this gradient to find the slope of the cost function and the direction the weights and biases should move towards. Once this direction is known, the parameters of each neuron are updated accordingly starting at the last layer and moving towards the first layer. This process is called backpropagation and refers to the way the model propagates the errors backwards over its layers (Rumelhart E. et al., 1986).

Different implementations exist for gradient descent with three of them being the most prominent. The main difference between them is how the gradient is calculated. The first is batch gradient descent. This algorithm uses all training data to calculate the gradient before taking a step. This means that the model is updated once per training iteration. The second one is mini-batch gradient descent. Here, the algorithm uses only a subset – a mini batch – of all training data. This allows the model to be updated more throughout training, however some mini batches might not always provide the most optimal gradient. The last one is **stochastic gradient descent (SGD)**. With this implementation, instead of calculating the actual gradient based on all data, an estimate thereof is used (Ruder, 2016). This estimate is calculated through stochastic approximation based on a random subset of the data. Different variants and optimizations exist for each method. For this research, SGD was used with the Adam optimizer.

5.1.1 SGD Adam optimizer

As discussed above, different algorithms exist to calculate the gradient, namely batch, mini-batch and stochastic gradient descent. These algorithms are used to calculate the weights and biases for the neurons in the network by iteratively taking steps in the direction of the minimum on the gradient. The size of this step is determined by the learning rate which is fixed for the standard implementation of the above three methods. However, variants or optimizers exist to dynamically change the value of the learning rate to optimize the training process. In general, the learning rate should start high and decrease as the model gets closer to the local minimum on the gradient. This way, the model will converge quickly without overshooting or diverging from its optimal state.

Adam is an optimization algorithm for stochastic gradient descent (SGD). While normal stochastic gradient descent maintains a single learning rate for all weight updates which does not change during training, Adam applies an adaptive learning rate that is computed for each network based on estimates for the first and second moments of the gradient. It combines the advantages of two other optimization techniques: AdaGrad and RMSProp. Instead of adapting the parameter

learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance) (Brownlee, 2017). According to (Kingma & Ba, 2014), Adam is computationally efficient, requires little memory and is invariant to diagonal rescale of the gradients.

The choice of the optimizer is based on which gradient descent algorithm will be used. Within PyTorch, implementation of an algorithm is very straightforward and therefore, the best optimizer was chosen based on the current state-of-the-art. Adam is currently one of the best optimizers and a very popular option. The method is really efficient when working with problems involving a lot of data or parameters (Doshi Sanket, 2019). For these reasons, Adam was used as optimizer for all models.

5.2 Hyperparameters

An important part to any deep learning problem is defining the hyperparameters and finetuning them for optimal performance. A hyperparameter is a parameter that can be manually set by the programmer. They affect the behavior of the model during training and thus have a direct influence on how well the model will perform afterwards. They are different from the internal parameters of the network such as a neuron's weights and bias, whose values are derived during the training process. Many different hyperparameters exist, each having their own function but not all of them need to be used in every model. Below, each hyperparameter that was used in this research is discussed.

- **Train-test ratio** is used to control how much of the original data is allocated to each subset of the main dataset as discussed in 3.5.2. The proportion of each subset is relative to all available data points, i.e., IO-sequences. For this research, an **80% - 20%** split was used for respectively the train and test dataset.
- **Learning rate** refers to the rate that an algorithm converges to a solution. For a deep learning neural network, it determines the size of each step during the gradient descent. High learning rates cause the model to converge very quickly but come with the risk of overshooting the minimum. On the other hand, low rates may cause the model to not reach convergence at all by the end of training. An initial learning rate of **0.001*** was used for the Adam optimizer.
- **Batch size** determines how many datapoints, IO-sequences are processed at once during training. A large batch size increases memory usage but also increases the number of predictions considered to calculate the gradient (Shen Kevin, 2018). Higher batch size also means that the total amount of batches is lower and thus a higher learning rate should be applied to assure that the minimum is reached within these fewer backpropagations. A small batch size has the opposite effects. A batch size of **64** was used for all models. This is a relatively low batch size, but it was chosen because of memory limitations when working with images.
- **Number of epochs** or the number of training iterations, determines how many times the model can process all training data. During one epoch, all batches of the training data are processed once to update internal parameters. When the model is trained for more epochs, it has more chances to find the best internal parameters. However, training the network for too many cycles can lead to overfitting where the network starts to learn the detail and noise in the training data to such an extent that it starts to negatively impact the performance on new data. By using an unseen validation dataset, overfitting can be measured. When the prediction errors on validation data start to increase while those of the training data are still decreasing, the model is overfitting and training should be stopped. All models were trained for **50*** epochs.
- **Data frequency** defines at which frequency data is fed as input and at which frequency data is predicted by the model. The simulation data was generated at **2Hz** or 2 frames per second as a good balance between preserving signal accuracy without having consecutive frames containing too similar data. With up sampling being impossible, the assumption was made that down sampling to a lower frequency was unnecessary as this would reduce the detail in the predictions and could introduce aliasing effects.
- **Input sequence length** sets the number of frames used as input to make a prediction upon. This hyperparameter had no fixed value and was **based on the model**.

- **Output sequence length** sets the number of datapoints the model needs to predict. An output length of **60 frames** at two frames per second was chosen as the minimum. This way, performance of each model could be measured over a 30 second prediction window as discussed in the objectives and criteria. However, as stated in the objectives, the prediction duration should be extended as much as possible.

The last two hyperparameters, input and output sequence length, are expected to have high impact on the performance of the model. They will be tested the most rigorously to find the best configuration for each model. Ideally, only a short input sequence is needed to make an accurate prediction over a long sequence length. Due to its importance and for readability, the ratio of input to output sequence length will be defined as the **IO-ratio** (input/output-ratio). The notation of the IO ratio will always be in non-simplified, non-fractional form, i.e., a 30/60 IO-ratio will never be written as $\frac{1}{2}$ or 0.5 to avoid confusion.

As a final note on the hyperparameters, finding the optimal value for each hyperparameter is called hyperparameter optimization. Different algorithms exist to perform this optimization (Claesen & de Moor, 2015). However, they are very time-consuming operations, especially when many different models are used. Each model would need optimizing if they were to be fairly compared which would be inefficient. An alternative approach was used in this thesis where only some parameters were manually tested to find optimal values. These parameters were tested on the simple single-step model. Once an optimal value was found, it was then used for all models in assumption that they would behave similarly. This way all models could be compared equally with roughly estimated optimal hyperparameters. The **empirically found values** are marked with an asterisk (*). Complete testing with results will be discussed in the next chapter.

5.3 Performance metrics

5.3.1 Loss function

To compare the ground truth with the predictions made by the model, a metric is used to calculate the error, i.e., the loss function. Many distinct functions exist to define this loss, each with their own appliances. For continuous data or linear regression, the **Mean Squared Error (MSE)** is a popular option (Wang et al., 2022b). The MSE of two sets of points of n elements, i.e., number of frames is defined as the average of the sum of the squared errors, where the error corresponds to the difference between ground truth Y_i and prediction \hat{Y}_i .

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2 \quad (5.1)$$

During training, the MSE was used as loss function for gradient descent. It calculates the average error over the whole predicted sequence. For example, if sixty values are predicted, the MSE will return one value: the average error over these sixty points. MSE will also be used to compare different models' accuracies to each other. However, this is quite a general comparison because some interesting information is lost when the difference between two sequences is averaged into one value. For a more detailed comparison and evaluation of the predictions, an extra metric was introduced: **Loss Per Frame (LPF)**. The LPF of two sequences of length n is defined as a sequence of n errors where each error is the difference between the two sequences at the corresponding index. Returning to the first example, if sixty values are predicted, the LPF will return an array of sixty elements containing the error between each pair of predicted and real values individually. When this metric is averaged over all predictions made on an unseen test dataset, the result can be used to analyse how the error changes on average throughout the predicted sequence. Additionally, if the **root of the MSE** is taken – the **RMSE** – the errors are transformed back to the units of the original data and can be de-normalised into their original domain. Using this principle, all MSE values can be translated back into their corresponding pitch and roll angles in degrees – which provides a more sensible metric for comparison than MSE.

5.3.2 Inference time

In cases where a deep learning model needs to be deployed in real world scenarios to make predictions on new data, the speed at which it can make those predictions can be of significant importance. For example, self-driving cars with limited computational resources need fast, low-latency models. This can be a challenging metric to fully optimize when high demand architectures are used such as LSTM networks (Kouris et al., 2019). The target vessel will similarly to the self-driving cars have a limited number of computational resources. Because of this, each model will be subjected to a measurement of its inference time – the time it takes to make one forward pass prediction on one IO-sequence. Besides having a low error across predictions, having a low inference time is equally important. In order to measure true inference time, all outside influences should be eliminated. When executing on a GPU, factors like GPU warm-up, asynchronous execution and system hardware specification all need to be considered. As discussed in 2.6, inference time will be measured as the average prediction time over ten thousand non-batched IO-sequences.

GPU warm-up is a process in which the GPU is initializing. Modern GPUs have multiple power states to reduce their power usage. When the GPU is not used, it will go into a power saving state and potentially completely turn off. In lower power state, the GPU shuts down different pieces of hardware, including memory subsystems, internal subsystems, or even compute cores and caches. A program attempting to interact with the GPU will cause the driver to load and initialize the GPU. It is this driver load that can influence the measurement of inference time. According to (Geifman, 2020), programs that cause the GPU to initialize can be subject to up to three seconds of latency due to the scrubbing behavior of the bit error correcting code (ECC). Memory scrubbing is the process of reading data from each memory location, correcting faulty bits with an ECC, and writing the data back to its original place. Below in Figure 22 the effects of GPU warm-up are clearly visible. A big spike in execution time is measured on the predictions when the GPU is still initializing. When the GPU is initialized, the timings decrease and remain constant. It is important that this initialization period is not included in the inference time calculation because it is inaccurate and does not reflect a production environment where the model is continuously making real-time predictions on an initialized GPU. To compensate for GPU warm-up, a broad estimate of one thousand predictions are executed before time measurements start.

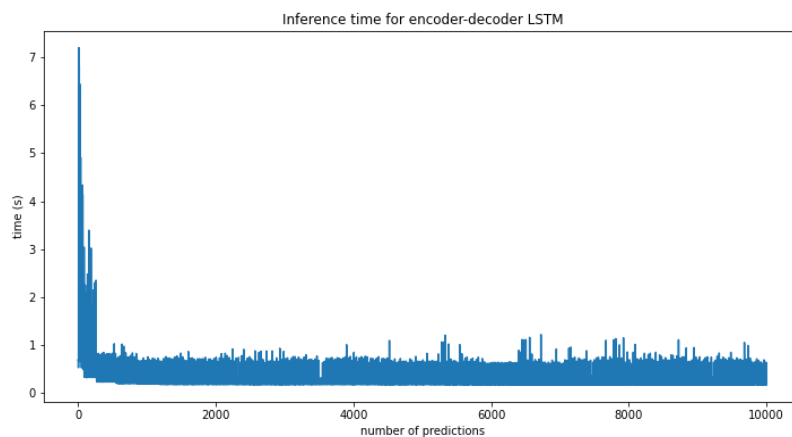


Figure 22: Inference time over 10,000 prediction with GPU warm-up effect

Asynchronous execution is mechanism that occurs in multi-threaded or multi-device programming. Different tasks are scheduled to different execution units which execute them asynchronously. This can cause a block of code to be executed before the previous one is finished and therefore speeds up execution. However, if a neural network is executed on the GPU – which is asynchronous by default – and inference time is measured on the CPU, the timer will stop before the model is done with its execution. This needs to be taken into consideration when certain popular Python libraries are used like *time* for example, which is executed on the CPU. PyTorch provides a very similar version of the popular *time* library designed for measuring and synchronizing events on a cuda-enabled device.

```

def inference_time(model, dummy_input, repetitions=10000):
    device = torch.device("cuda")
    model.to(device)
    dummy_input.to(device)

    # INIT LOGGERS
    starter = torch.cuda.Event(enable_timing=True)
    ender = torch.cuda.Event(enable_timing=True)
    timings = np.zeros((repetitions, 1))

    # GPU WARM-UP
    for _ in range(1000):
        _ = model(dummy_input)

    # MEASURE PERFORMANCE
    with torch.no_grad():
        for rep in range(repetitions):
            starter.record()
            _ = model(dummy_input)
            ender.record()
            # WAIT FOR GPU SYNC
            torch.cuda.synchronize()
            curr_time = starter.elapsed_time(ender)
            timings[rep] = curr_time

    mean_syn = np.sum(timings) / repetitions
    std_syn = np.std(timings)

    return timings, mean_syn

```

Listing 1: Function definition for inference time measurement

Lastly, the **system hardware specifications** on which the measurements were recorded are mentioned for reference. All measurements were taken on a workstation with the following specifications:

- CPU: Intel Core i7 10700k at 3.8GHz base clock and 4.6GHz boost clock
- GPU: Nvidia GeForce RTX 3080 with 8704 cuda cores at 1800MHz boost clock
- RAM: 3200MHz

5.4 Summary

In this chapter, the testing environment was discussed. This environment encompasses all relevant parameters and methods which have direct influence on the performance and evaluation of the models. By fixing most of these variables and methods across all models, the effect of specific individual parameters can be accurately measured, and models can be compared fair and equally. In the first section, the training optimizer was discussed. To optimize the gradient descent process during training, Adam was used. Adam is an optimizer for the stochastic gradient descent algorithm that has high performance compared to other optimizers. It dynamically changes the learning rate to achieve faster and better convergence. In the second section all hyperparameters are discussed in combination with their assigned value. Some of these hyperparameters were arbitrarily chosen, others were determined based on experimentation on the single-step model. Lastly, the error metrics were discussed. To measure performance, MSE loss, LPF and inference time are used. The first two are functions to calculate and visualize errors and their trend across a predicted sequence. The inference time is used to evaluate the latency that occurs between input and prediction. GPU warm-up, asynchronous execution and system hardware specification all play a role when measuring inference time.

6 Results

In this chapter, an extensive overview is given of all tests that were performed and their results. In first section, the results from the hyperparameter optimization are discussed. Afterward, each model will be discussed in detail, followed by a section on augmented data performance and inference time results. In the last section, a final overview and comparison is provided of the best performing models. To analyze the accuracy of the models, mainly the MSE, LPF and graphs showing the real vs. predicted values were used. As a baseline, each model is also compared to the *zero-predictor* as discussed in the criteria.

6.1 Hyperparameter optimization

Three hyperparameters were optimized: **LSTM hidden layer size**, **number of epochs** and **learning rate**. It was hypothesized that these three parameters would have the most influence on the performance of the models and thus a simple method was applied to optimize them. Three values for each parameter were tested: one low, one high and one in-between value. With the results of these three tests, the optimal value is chosen based on the trend they form. The resulting value is then used for all other models in assumption that they will show a similar behavior. All tests were executed on the dual output variant of the single-step model due to its low complexity and thus fast training time.

6.1.1 Number of epochs

The number of epochs determines how many times the model can process the full training dataset to optimize its internal parameters. Low training epochs allow for fast training but might lead to the model not reaching its optimum. High epochs cause longer training times but give the model more time to converge. To define the optimal value, the test model was trained for 8, 25 and 50 epochs and the training loss graphs were compared (Figure 23).

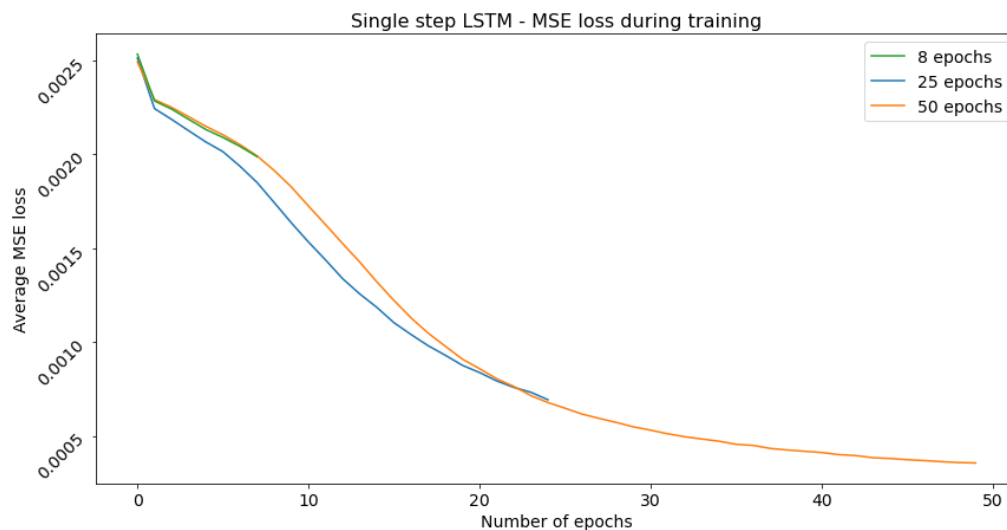


Figure 23: Training losses for different numbers of epochs

During training, MSE loss should rapidly decrease (converge) on the first couple of epochs and ease out to a point where it reaches its minimum. At that point, the model has learned the mapping between input and output to the best of its abilities and receives no more benefits from additional epochs. It is clear that the model needs at least 50 epochs to properly train itself and achieve the lowest MSE. Even more, the model is still marginally decreasing during the final epochs indicating that it hasn't reached optimal convergence yet. Because of this, an even higher number of epochs could possibly lead to slightly better results. Nevertheless, the number of epochs was chosen to be 50 for all future training in assumption that

this was a good trade-off between optimal performance and minimal training time. Especially when training the more complex models.

6.1.2 Learning rate

Learning rate was the second parameter that was optimized. Learning is a very hard to conceptualize hyperparameter. It is defined as the size of the step that is taken during one iteration of moving down the gradient towards the minimum. During research of this parameter's value in other deep learning applications, most cases used a learning rate of 0.01, 0.001 or 0.0001. Learning rate is very much dependent on the complexity of the problem. Simple problems may get away with high learning rates resulting in very fast conversion, but on complex problems such as this one, a smaller learning rate is recommended (Wilson & Martinez, 2001). The three above mentioned values were tested in order to find the optimal value. Once again, the training loss graph is used to discuss the results (Figure 24). It is important to note that these tested values were used as initial learning rates for the Adam optimizer. As discussed in previous chapter, Adam will dynamically change these learning rates during training. However this does not mean that the initial value is not important.

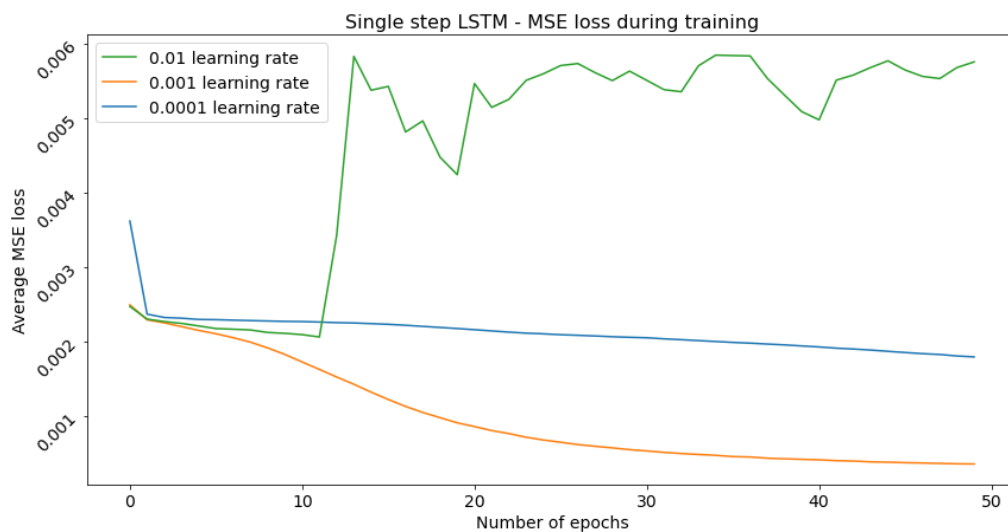


Figure 24: Training loss for different learning rates

The model shows interesting but not unexpected behavior. Based on the mathematical theory for learning rate, a learning rate that is too high will result in very drastic changes to the internal parameters, causing divergent behavior. This is clearly what is happening with the model on the green line of with 0.01 learning rate. The model is basically bouncing around on the gradient without really converging to the minimum at all. The opposite is visible on the blue line. In this case the learning rate is smaller than ideal as the model is converging very slowly compared to the orange line. Due to the smaller steps, the model needs more iterations to reach the minimum. If the current rate is projected forward, convergence will require somewhere between five to ten times more epochs. The model on the orange line and the model on the blue line will probably reach similar performance in the end. However, the orange one will be much more efficient which is why a learning rate of 0.001 was selected as the value for all future models.

6.1.3 LSTM hidden size

After determining the learning rate and number of epochs, the hidden size for LSTM modules was assessed. Judging based on the nature of our problem – sequence prediction – and the efficiency of the LSTM module for this type of problem, it was stated that the hidden size of the LSTM modules would be the most important parameter to optimize. The size of linear layers, and parameters of the CNN modules were assumed to be less impactful on performance. In addition, the parameters for the CNN were already tested and proven effective in Kaminskyi's research. Initially, a hidden size of 128

was chosen for the single-step LSTM model. To find the direction of the optimal hidden size, a lower 32 and higher 256 hidden size were tested. The results are shown in Figure 25.

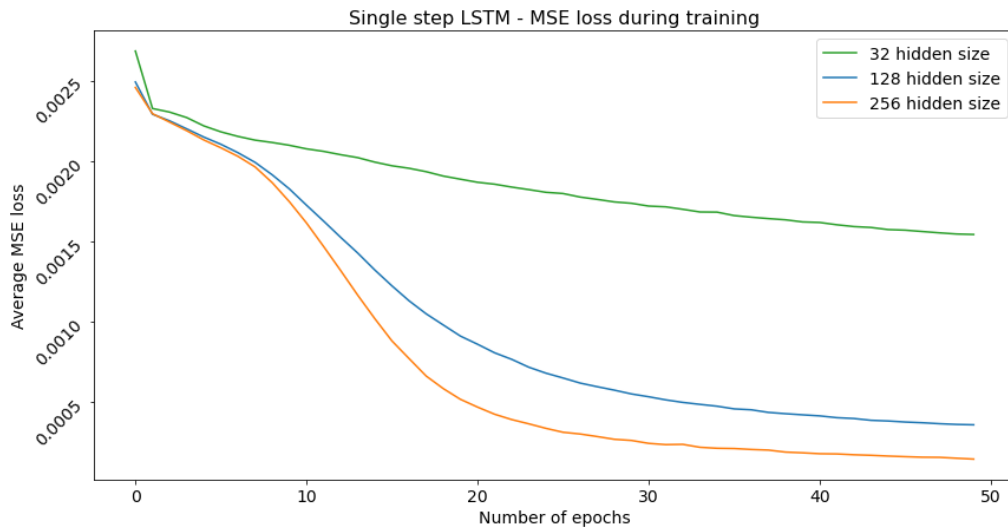


Figure 25: Training loss for different LSTM hidden sizes

Once again, the model shows expected behavior. Generally speaking, when increasing size of hidden layers, the network is able to learn a more powerful function between input and output. In the case of LSTM networks where memory cells are used as hidden nodes, increasing their number allows the model to better remember long term dependencies simply because there are more parameters to represent them. This is visible on the graph where the model converges to the lowest MSE with the highest number of hidden nodes. From this trend, the conclusion was made that larger hidden sizes are beneficial. For the Encoder-Decoder LSTM, the hidden size was increased to 300 following this trend. For the CNN LSTM ensemble models, the hidden size was increased even further to remain in proportion with the much larger input vector.

6.1.4 Activation functions

As a final general-purpose optimization, the effect of activation functions was assessed on the single-step model. The single-step model has three logical points in its architecture where activation functions can be placed: one to activate the input before the LSTM module **(1)**, one after the LSTM **(2)** and one after the linear layer **(3)** which activates the predicted output. Since the model trains with normalized data between $[-1, 1]$ and is expected to return values in this domain, a tanh is the only possible activation usable on the output layer. The activation on the input should consider this domain as well. A ReLU here would be ineffective as it nullifies all datapoints lower than zero. Due to the angular units of pitch and roll where negative and positive values are equally important, adding a ReLU here leads to a large reduction of information in the input. This leaves the second option as the most interesting point for an activation function.

Three configurations were tested on the single-step model: one with a ReLU at (2), one with a tanh at (3) and one with a ReLU at (2) and a tanh at (3). The ReLU activation is used in almost every model of Kaminskyi after an LSTM module. However, as discussed in the previous paragraph, the use of the ReLU is expected to negatively impact the results. The goal of this experiment was thus to mainly evaluate the impact of a ReLU activation on the output of the LSTM module. A secondary objective was to evaluate the impact of a tanh activation on the output.

The results are shown in Table 10. Each configuration was trained for 50 epochs with a 0.001 learning rate. The test resulted in similar performance for each configuration. The introduction of the ReLU activation functions resulted in no measured benefit, rather a slight deterioration. From this experiment, expectations were confirmed, and it was concluded that ReLU activation functions were not beneficial to the performance. The tanh on the other hand, showed no immediate impact on performance. However, it was later on proven beneficial due to its ability to transform extremely high predictions into -1 or 1. This solved the issue of training loss graphs turning into one high peak at epoch one followed by a flat line due to the minor differences in MSE losses being out-scaled by the single large peak value.

Configuration	Average Pitch error [denormalized RMSE]	Average Roll error [denormalized RMSE]
Barebones model	2.18°	2.06°
ReLU (2)	2.36°	2.20°
Tanh (3)	2.19°	2.04°
ReLU (2) + Tanh (3)	2.26°	2.15°

Table 10: Average pitch and roll errors for different activation function configurations

6.2 Single-step model

In this section, the single-step model will be analyzed on prediction accuracy. Three different variants were trained: one single-output variant for each pitch and roll and one dual-output variant for simultaneous pitch and roll predictions. These models should perform very well as they only need to predict one value. The three models were trained with an input sequence length of both fifty and ten PR-values to identify their optimal IO-ratio. Below in Table 11, the results are shown for each model in each configuration. Note that error values in the table are calculated as the denormalized RMSE averages over all predictions made on the unseen test data set.

Variant	IO-ratio	Average Pitch error [denormalized RMSE]	Average Roll error [denormalized RMSE]
Pitch	50/1	1.85°	/
Roll	50/1	/	1.91°
Dual	50/1	2.17°	2.03°
Pitch	10/1	2.28	/
Roll	10/1	/	2.25°
Dual	10/1	2.34°	2.09°
Zero prediction		6.43°	7.30°

Table 11: Average pitch and roll errors for single-step LSTM variants at 10/1 and 50/1 IO-ratios

On the first three grey rows, the results are shown for the models that were trained with an input of fifty frames. Overall, the performance is very good. The dual-output variant performs slightly worse compared to the single output ones shown on row one and two. This could be due to the network only having to optimize to one parameter. The bottom row in red shows the average error for the hypothetical zero-predictor model. As expected, these values are significantly worse than the trained models, meaning that the models have indeed captured the underlying trend in the simulation data and are making logical predictions. The same variants were also tested with a lower number of frames in the input sequence. This was done to find out whether or not it is necessary to provide fifty frames for just a single prediction. With an input sequence length of ten, the single-step models still perform very well, even though the model only has a fifth of the input of the previous configuration. Only a marginal decrease in performance is shown compared to the 50/1 IO-ratio models.

A recurring trend was noticed among all these results. Roll prediction errors are overall lower than pitch prediction errors. Kaminskyi made the same observation and according to his results shown in 2.6, this behavior returned for almost all models. The difference could be explained by roll having a somewhat more predictable behavior than pitch. For example, pitch could inherently have a quirkiest behavior which causes it to have a lot more micro changes that are hard to learn and consistently predict. In contrast, the results from the zero predictor are the opposite, the roll error is higher than the pitch error. Based on the data analysis which showed that pitch and roll have similar distributions, this difference means that

roll in general has a higher deviation from zero than pitch. Yet, even with this wider spread, roll must have a more predictable trend.

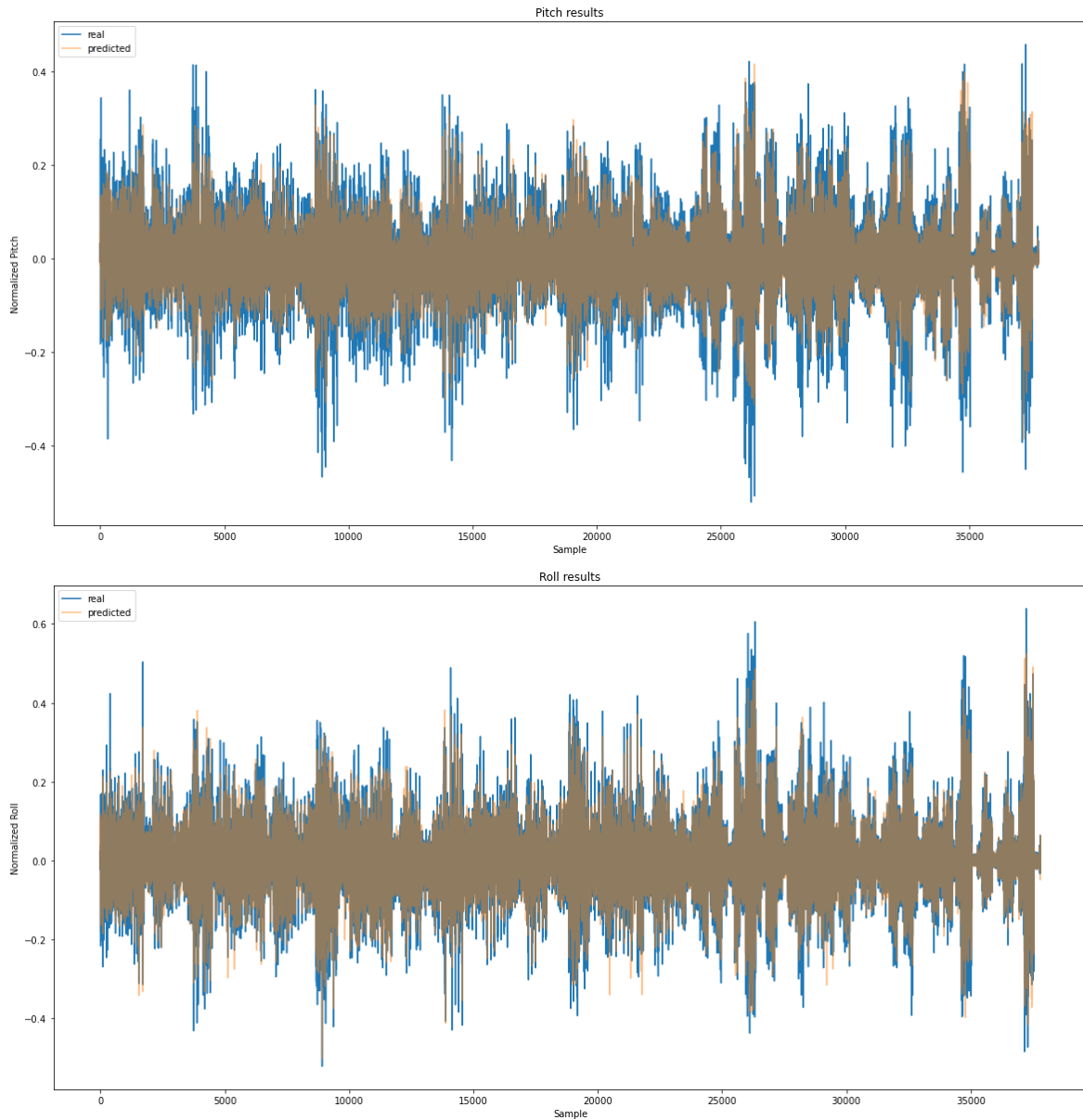


Figure 26: Predicted (orange) vs. real (blue) values for pitch (top) and roll (bottom)

To further analyze the results from the single-step model, only the dual output variant will be considered. The table above provides a general idea of prediction accuracy; however, some information is lost when averaging results. In Figure 26 the prediction results are shown for pitch (top) and roll (bottom) across the whole test dataset. The predicted values are shown in orange on top the real values which are shown in blue. The dark sand-colored area represents overlapping values between predictions and real values, while orange or blue areas mean that predictions respectively exceed or under-run the real values. For readability, the dark sand-colored area will be referred to as the *dessert*. The top graph for pitch shows a constant blue area around its dessert. Judging by the shape of the blue outline and the shape of the dessert, the predicted pitch values seem to follow the trend well but consistently fall short of the real values by a small margin. When comparing this to the roll graph, the effect is much less noticeable which indicates that roll is being predicted with greater accuracy. This confirms the hypothesis made above - that was solely based on the RMSE averages of Table 11 - that pitch must be more random and thus harder to predict.

Lastly, the training generalization of the three single-step models is analyzed for a 10/1 IO-ratio configuration. This is done by plotting the average MSE loss per epoch of the models for both the training and validation dataset. This is shown in Figure 24. Each full line represents the training dataset loss while the dotted line represents the validation dataset loss. Generally speaking, the solid training curve should be lower than the dotted validation loss curve because the model always aims to optimize towards the training data and does not “know” the validation data.

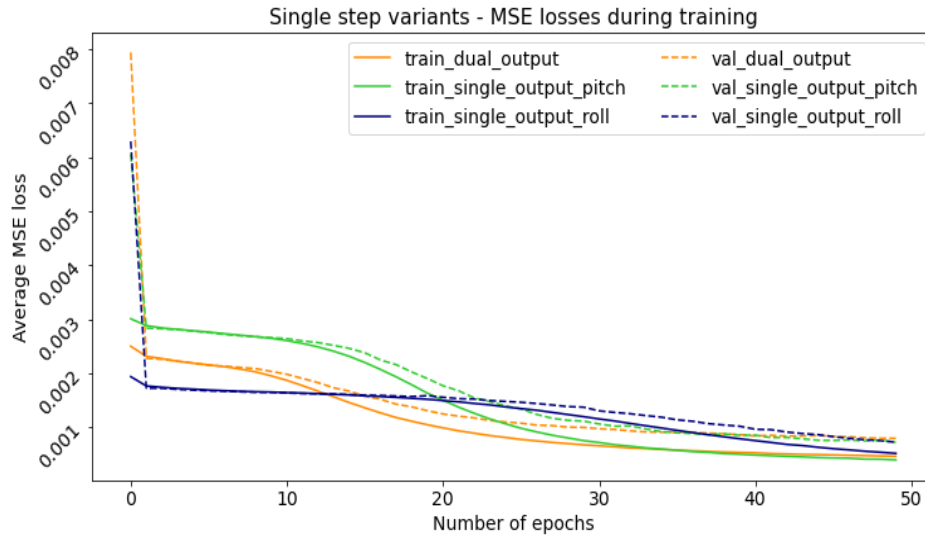


Figure 27: MSE loss during training of the single-step models

With this representation, convergence rate and overfitting can be visualized and assessed. The first of which is the rate at which the losses decrease. When the loss stops decreasing, the model has reached a local minimum on the gradient. The roll prediction model has by far the slowest convergence out of the three, followed by the dual-output model. This means that it needs the most time to find its optimal internal state. Eventually, all models converge to a similar average error. However, the roll prediction model's losses are still marginally decreasing during the final epochs, indicating that it might benefit from a few more training epochs. The same conclusion as above can be made here once again: roll has the most accurate predictions. Its validation loss (blue dotted line) is lowest in class on the last epoch even though its training loss (blue solid line) is the highest. Based on these graphs, little to no overfitting has happened during training. Overfitting is happening when the validation loss starts to increase again after reaching a certain point while training loss keeps decreasing.

In conclusion, even with roll having more spread-out values, it is still easier to predict than pitch. All models performed really well and produced results that can be later used as an additional benchmark. This model and its variants prove that pitch and roll can be predicted with **average error of approximately 2°**. However, they are not capable of sequence predictions.

6.3 Multi-step models

6.3.1 Encoder-Decoder LSTM

The Encoder-Decoder LSTM model is expected to perform the worst out of all proposed models according to Kaminskyi's research. However, based on the results below, it becomes clear that this is not the case, and this model actually performs very well. The Encoder-Decoder LSTM was extensively trained for a multitude of different input sequence lengths and a fixed length output of 60. This was done to determine the optimal IO ratio for accurate predictions without creating too much overhead of unnecessary long input sequences. Based on the empirical results from the single-step models, the number of epochs for all training was fixed at 50. The training results are shown in Figure 28 for the first six evaluated IO-

ratios. The same layout as above is used where each color represents a different configuration, and the training and validation losses are respectively solid and dotted lines.

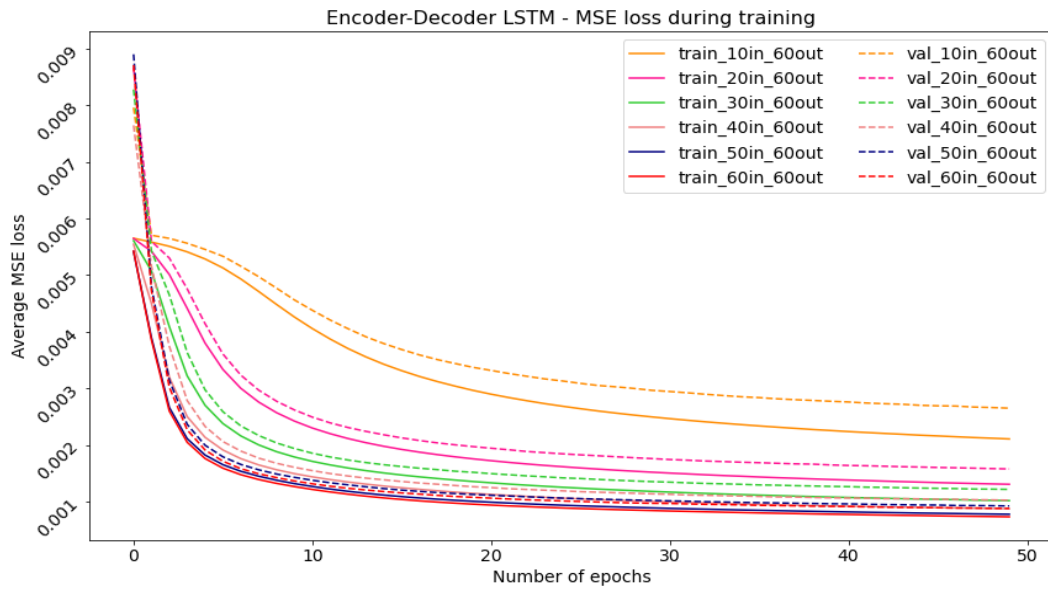


Figure 28: Encoder-Decoder LSTM training and validation losses for different IO-ratios

A general trend is present among the configurations: the loss and convergence get better the more input frames are used. Intuitively, this behavior is to be expected. The 10/60 IO-ratio (orange) performs the very worst, resulting in the highest MSE loss and slow convergence. As the input sequence length increases however, so does convergence. Overall, no overfitting was observed for any of the configurations. Additionally, fifty epochs proved to be a good amount for this model as the higher IO-ratios reach near optimal convergence during the final epochs with their loss remaining almost constant.

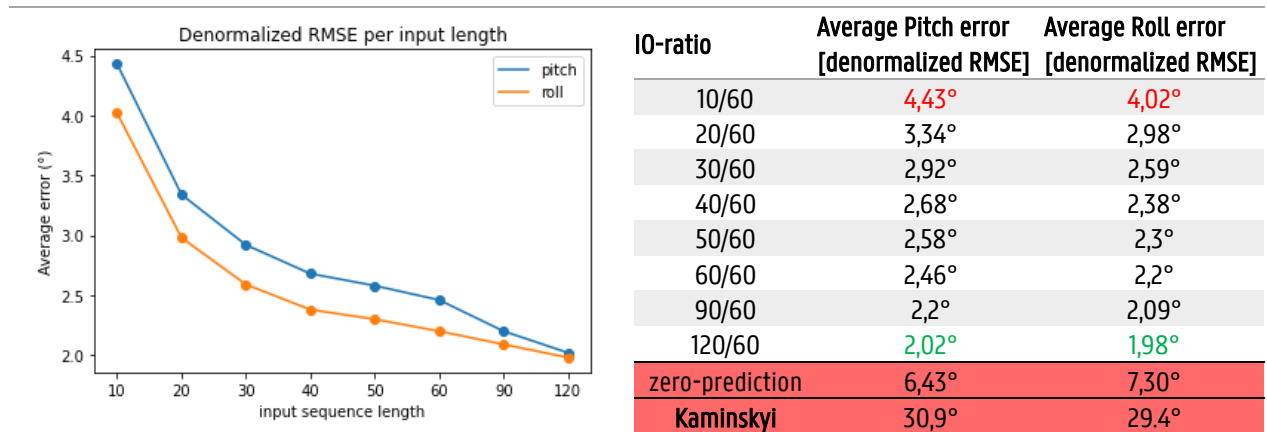


Table 12: Average pitch and roll errors for the Encoder-Decoder LSTM per IO-ratio

In order to further evaluate the optimal input sequence length for a thirty second prediction, two more IO-ratios were tested: 90/60 and 120/60. Table 12 shows the average error for pitch and roll for each IO-ratio in combination with a visual representation. These errors are denormalized RMSE averages of all predictions made on the test dataset. Once again, the zero-predictor results are appended for comparison. The trend found in the single-step model's results is noticed once again: pitch prediction errors are consistently higher than those of roll. This further affirms that predicting pitch is harder than predicting roll. Although it is observed that this effect decreases as the input sequence length increase. The error margin for both pitch and roll decrease with an exponential rate as the input sequence length linearly increases. Note that

on the graph, the last three points appear to be linearly decreasing, however this is due to their x-labels being disproportionate compared to the first six. From these results, it was concluded that the Encoder-Decoder LSTM could predict a thirty second sequence with an **average error of 2°** for both pitch and roll at a **120/60 IO-ratio**. Lower ratios like 90/60 or 60/60 retain high accuracy as well, however, they are not able to predict pitch with equal accuracy as roll.

These results are **very contradictory to Kaminskyi's results**. Kaminskyi tested a similar model for 50 epochs at a 20/24 IO-ratio. His network resulted in an average error of approximately 30° for both pitch and roll on the 10th predicted second (20th frame at 2Hz). In our case, even the worst IO-ratio configuration for this model does not reach an error of this magnitude at 30th predicted second (Figure 29), let alone at the 10th second. Kaminskyi's result is also far worse than even zero-predictor. This is an extreme difference because a similar model trained on the same data should - in theory - perform similar. Based on the distribution of the data discussed in 3.4.2, a 30° error is almost impossible when 99.7% of all datapoints lies within a [-20°, 20°] interval. Therefore, it was concluded that Kaminskyi had made an error during the calculations or de-normalization of his results, or that the model had a too low learning rate (0.0001) which caused it to not have converged by the end of training.

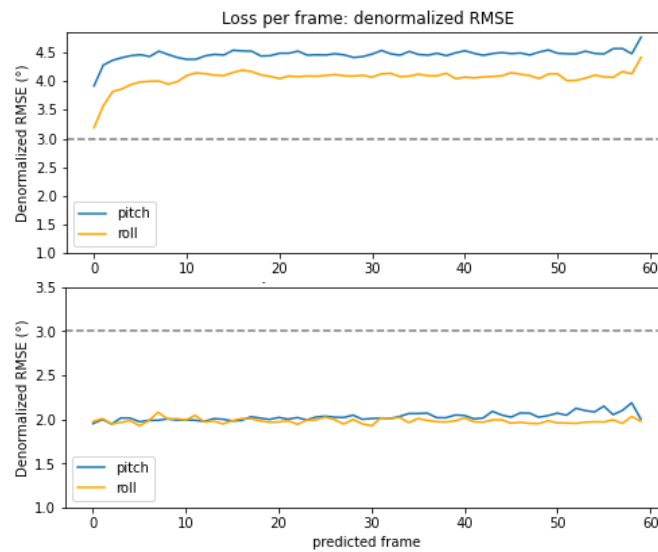


Figure 29: LPF measurements for 10/60 (top) and 120/60 (bottom) IO-ratio with reference line at 3°

For each of the different configurations, the average loss per frame was calculated to observe error evolution over the predictions. In Figure 26, the two graphs are shown for resp. the worst and best IO-ratio. In the 10/60 LPF plot (top), the difference in pitch and roll prediction difficulty is very apparent. All LPF plots for this model resulted in a blue pitch line that is higher in its entirety than the orange roll line. Besides this, a remarkable and counterintuitive property of this model was observed. Only a minor increase in prediction error happens when predicting further into the future. It was expected that on average, the error would increase over the predicted sequence. However, the models predict with a close to **constant error** across the whole sequence and the magnitude of the error is mainly affected by the IO-ratio. This behavior is similar to Zhao's results, where a consistent error was found as well: prediction errors at the 5th second and 20th second had equal magnitudes (Zhao et al., 2004).

As a final test for this model, a higher **output sequence length of 120** was tested to evaluate if errors would start to increase over a longer predicted sequence. At two frames per second or 2 Hz, this model predicts a one minute window. A prediction of this duration provides a lot of agility to find the optimal window for take-off/landing. In assumption that the ideal input sequence length for this model is independent of the output length, 60/120 and 120/120 IO-ratios were chosen based on previous results. The loss per frame measurements for these configurations are shown in Figure 30. In both cases an increase in pitch error is observed after the 60th frame whilst roll remains relatively constant. With an **average PR error of 2,52°** and **2,22°** for resp. 60/120 and 120/120, errors are similar to the 60/60 and 90/60 models, meaning that the increase in prediction length did not lead to a substantial decrease in accuracy. In conclusion, the Encoder-Decoder LSTM was found

to be a very reliable and accurate prediction model that outclassed Kaminskyi's best model *and* best overall prediction. It was able to predict up to one minute for both pitch and roll with equal, consistent accuracy.

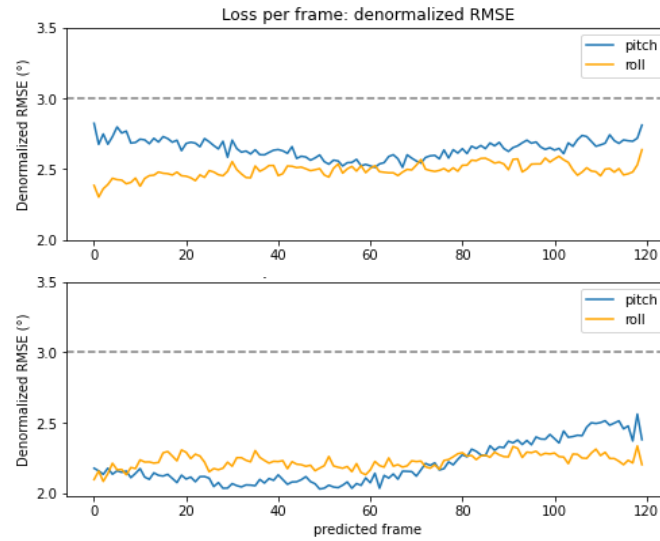


Figure 30: LPF measurements for 60/120 (top) and 120/120 (bottom) IO-ratio with reference line at 3°

6.3.2 Sequential CNN

The sequential CNN model is the only model to not use an LSTM architecture. For this reason, it was expected to have the highest difficulty learning the trend of the time series data causing it to perform worse than other models. However, the performance trade-off might be made up for by the expected lower inference time due to the absence of the LSTM module. The model was trained twice for two IO-ratios, each with only ten frames in the input sequence. The lower input sequence length was chosen in assumption that less images would be necessary to predict with the same level of accuracy as the Encoder-Decoder LSTM model. Additionally, the size of the concatenated image feature vector grows linearly with the number of input images. Processing more images in the input sequence quickly leads to high memory requirements which should be kept at a minimum as discussed in the objectives.

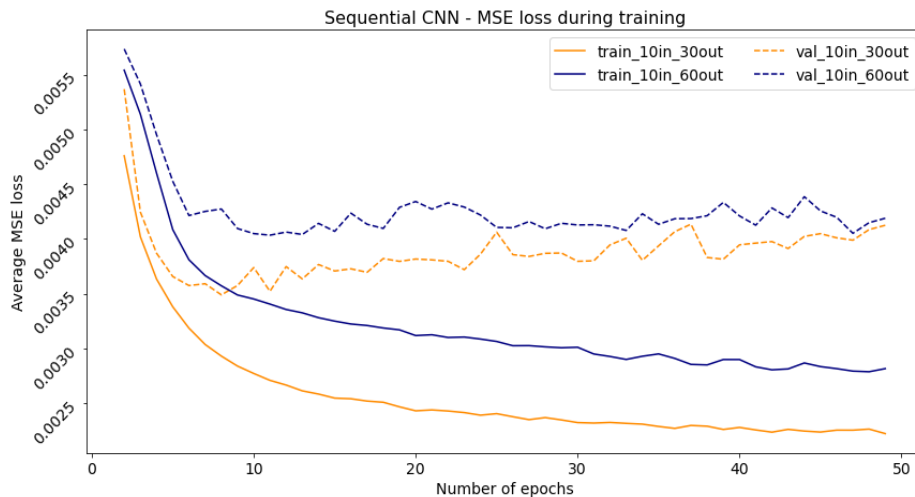


Figure 31: Training and validation loss for different IO-ratios of the sequential CNN model

In Figure 31, the training and validation losses are shown for a 10/60 (blue) and 10/30 (orange) IO-ratio. As hypothesized, the model indeed shows **poor convergence**. The 10/60 validation loss (dotted line) during training decreased as expected during the first ten epochs. However, from this point onward, validation loss did not continue to further decrease and instead followed a slight increasing trend. Training loss (solid line) on the other side, did proceed to further decrease across all training epochs. This indicates that the model suffered from slight overfitting after only a few epochs. Further testing

was done with a higher IO-ratio of 10/30 to assess whether this poor convergence originated from a suboptimal IO-ratio. However, even with a shorter prediction window, only marginal improvement in convergence was measured. The validation loss once again reaches a minimum in the first ten epochs after which overfitting starts to occur – even more than before – where validation loss increases while training loss keeps decreasing. In both configurations, the **average errors** of the sequential CNN on the test dataset for both pitch and roll ranged around **five degrees**.

After observing these poor results, training had been done on the CNN LSTM ensemble models. These in contrast, did show good performance with the same IO-ratios without significant increase in inference time compared to the sequential CNN. For these reasons, it was concluded that without an LSTM module, the sequential CNN could not achieve desired performance and was not further optimized nor evaluated.

6.3.3 CNN LSTM ensemble models

Compared to the sequential CNN model, the CNN LSTM ensemble models use an LSTM module to process the image feature vectors. Because of this, they are expected to perform best out of all models. Additionally, Kaminskyi's models with both a CNN and LSTM module performed best as well. However, due to their complexity, they are also expected to suffer from the highest inference timings. Both CNN LSTM ensemble models will be compared in this section as their architecture is only differentiated by the inputs they process, either only images (single input) or images and PR (dual input). Each model was trained and evaluated for a **10/60** and a **10/120 IO-ratio**. These ratios were chosen in order to compare prediction accuracy against the Encoder-Decoder LSTM. The low input sequence length was once again chosen in assumption that less images are needed compared to PR and to keep down memory requirements. The two models were also trained on **grayscale** images to lower computational demand and improve inference time.

Out of all proposed architectures, the ensemble models were the most resource demanding models to train, each taking around three hours to complete fifty epochs on a high performance cuda device. In Figure 32, the training and validation losses are shown for the **single (img)** and **dual (img-PR)** input model. Each configuration follows an almost identical path, even the grayscale ones. They converge rapidly during the first epochs after which the convergence rate decreases drastically. In the small window on the right, a more detailed view is provided of the last thirty epochs. All configurations of the CNN LSTM model reach a validation loss of around 0,0004 with only marginal difference between them. Overall, fifty epochs proved to be a good estimate for optimal training once again, for any configuration of the CNN LSTM ensemble models.

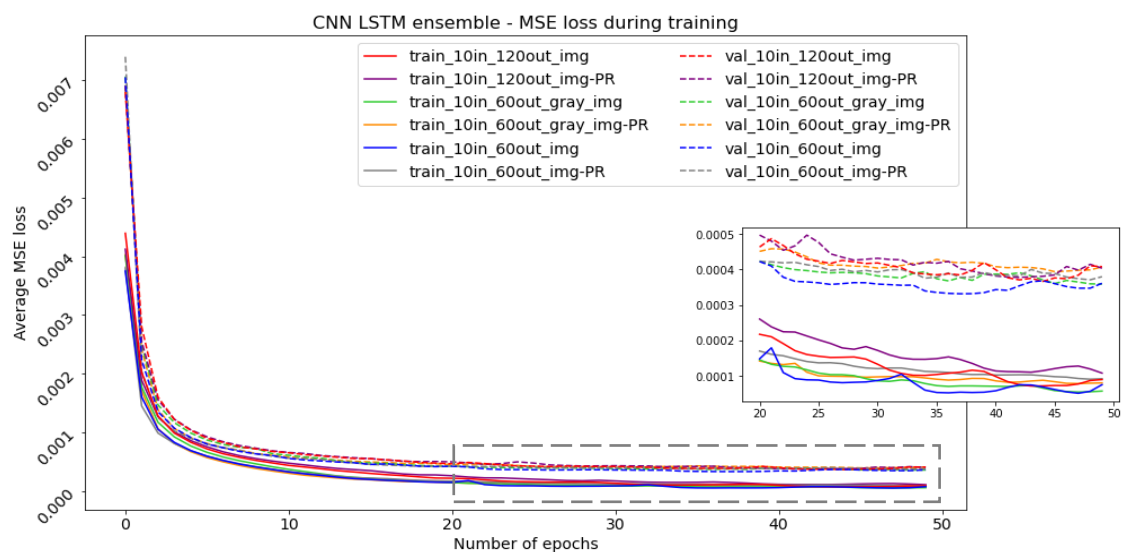


Figure 32: Training and validation loss for CNN LSTM ensemble models

In Table 13 the average denormalized errors are shown for both ensemble models. Four observations were made based on the results for the CNN LSTM ensemble model. **Firstly**, with the introduction of images, the previously observed recurring trend of pitch having higher errors than roll is not present anymore. This indicates that the introduction of images leads to more equally accurate predictions of PR. **Secondly**, no significant increase or decrease in accuracy occurs when grayscale images are processed. Based on this observation, it can be safely assumed that this will hold true for any IO-ratio and can therefore be used as a means to lower computational requirements without compromise. **Thirdly**, no benefit was found in using both images and PR as input. Prediction errors for both input type combination – img and img-PR – are within a margin of insignificance. As a result, it was concluded that the computationally expensive addition of PR served no benefit towards accuracy. **Lastly**, based on LPF measurements (Figure 33), each configuration showed the same prediction behavior as the Encoder-Decoder LSTM where prediction accuracy remains consistent across the whole sequence.

Input type(s)	IO-ratio	Average Pitch error [denormalized RMSE]	Average Roll error [denormalized RMSE]
img	10/60	1,74°	1,65°
img-PR	10/60	1,67°	1,76°
(gray) img	10/60	1,77°	1,66°
(gray) img-PR	10/60	1,86°	1,76°
img	10/120	1,81°	1,81°
img-PR	10/120	1,76°	1,78°

Table 13: Average pitch and roll errors for CNN LSTM single [img] and dual [img-PR] input models

In conclusion, the average prediction errors are near identical for each configuration of input types and IO-ratio. From the LPF metrics, it was observed that the accuracy remains consistent across the whole sequence. Compared to the Encoder-Decoder LSTM, which already outperformed Kaminskyi's best model and best result, the CNN LSTM ensemble models achieve even better performance by a significant margin (approx. 15%-25% PR error reduction).

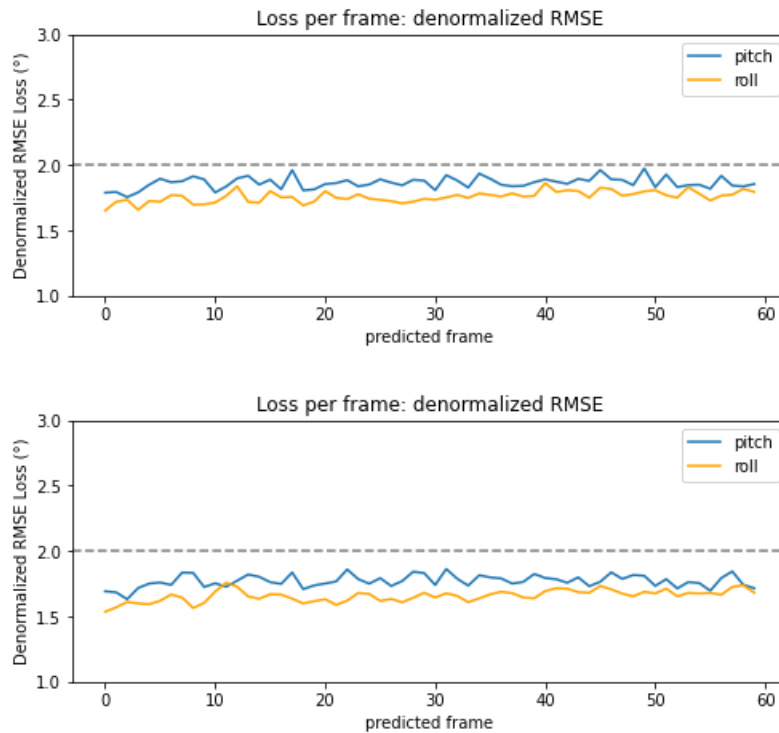


Figure 33: LPF for CNN LSTM at 10/120 IO-ratio for single (top) and dual (bottom) input on grayscale images

6.4 Augmented data

As discussed in 3.1.2, one sequence of augmented data was created by adding random ink blots to the images. These blots represent scenarios in which other objects are in the front of the vessel. Only the ensemble models were tested on this augmented sequence due to their excellent performance on images. It is expected that the dual input model might obtain better performance due to it not being solely dependent on images as input. To evaluate performance, only one heavily augmented sequence was created in assumption that results would be similar for other sequences and can only improve if a worst-case scenario is used.

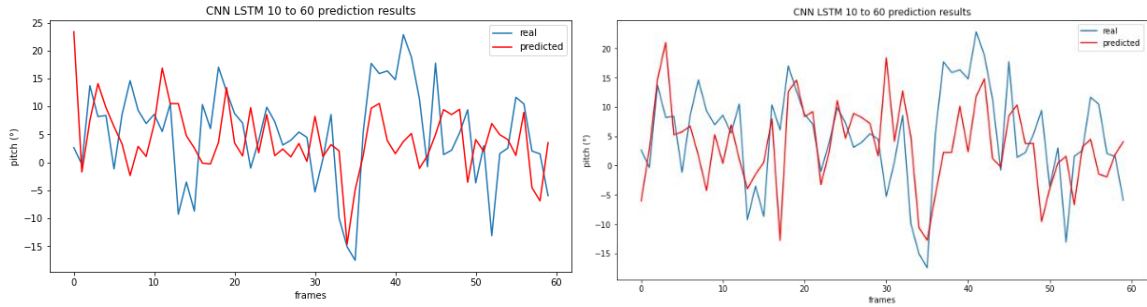


Figure 35: CNN LSTM *single* input prediction (red) vs. real (blue) on augmented frames: grayscale (left) and colored (right)

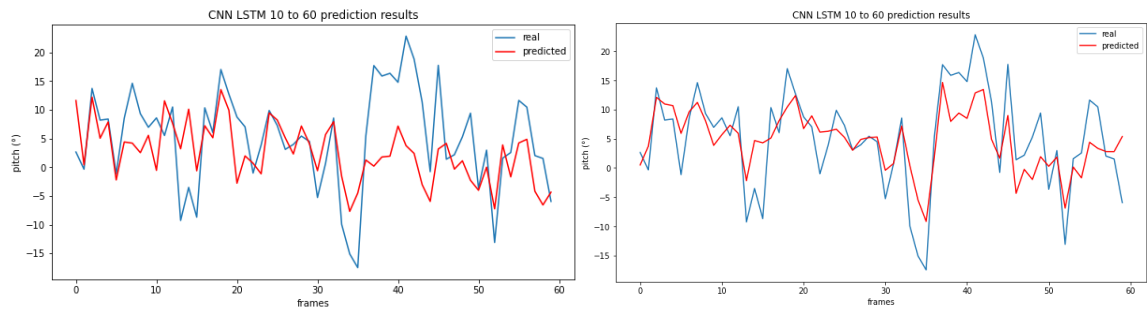


Figure 34: CNN LSTM *dual* input prediction (red) vs. real (blue) on augmented frames: grayscale (left) and colored (right)

In Figure 35 and Figure 34, results are shown for resp. the single (img) and dual (img-PR) input ensemble model. Pitch predictions are shown in red on top of the real values in blue for both grayscale (left) and colored (right) images. For reference, predictions of the ensemble model on the non-augmented sequence have almost perfect overlap with the real values. These are added in the Appendix. When visually comparing both models, the single input model (top) shows significantly more divergence from the blue line than the dual input model (bottom). As hypothesized, this could be caused by the lack of PR as input. Having PR as additional input, the dual input model is less dependent on the clarity of the images. Additionally, processing greyscale images tends to negatively impact peak prediction ability. In both cases, the peaks in the second half of the prediction window are more accurately predicted. Anticipating large movements is critical for safe landing of the drone and thus accurate peak prediction is of utmost importance. For this reason, it was concluded that the dual input model with colored images has the highest reliability and accuracy in obstruction dense environments.

6.5 Inference time

As discussed in the objectives, computational demand and inference time are equally important compared to prediction accuracy. In this section, all proposed models are compared in their optimal form. This optimal form is based on the results discussed in the previous sections: both prediction length and prediction accuracy should be optimized. In this fully optimized form, inference time or latency is used to make a final conclusion on which model is best overall. As discussed in 5.3.2, inference time is measured as the average over ten thousand predictions. The single-step model is not evaluated as it cannot predict sequences.

Table 14 provides a summary of the results from each multi-step model. Error values for each model are included in order to compare inference time and accuracy. These errors are calculated as the average of both pitch and roll over all predictions across all test dataset sequences. As expected, all image processing models have a drastically higher latency. The use of grayscale images improves latency by around 20%. However, this is not enough to make them viable options compared to the Encoder-Decoder LSTM* which is around twenty times faster and only marginally less accurate compared to the best performing image processing models.

Model name	IO-ratio	Inference time	Avg. PR error	Trainable parameters
Encoder-Decoder LSTM*	120/60	155ms	2°	1.211.010
Sequential CNN	10/30	4.203ms	5,79°	98.639.180
CNN LSTM [img]	10/120	4.042ms	1,73°	217.659.752
CNN LSTM [img] (gray)	10/120	3.545ms	1,71°	217.659.352
CNN LSTM [img-PR]	10/120	4.453ms	1,72°	217.741.672
CNN LSTM [img-PR] (gray)	10/120	3.797ms	1,81°	217.741.272

Table 14: Inference time, PR error and number of trainable parameters for each model's optimal configuration

As a final note, inference time measurements were taken on various IO-ratios for each model. This was done to evaluate whether this had effect on the inference time. However, no meaningful change in latency occurs when changing the IO-ratio. Image processing models were only trained for different output sequence lengths. It is expected that inference time of these models will change linearly with change in the number of input frames.

6.6 Summary

In this chapter, each model proposed in Chapter 4 was rigorously evaluated to form a final conclusion on one optimal architecture. In the first section, three hyperparameters were optimized: number of epochs, learning rate and LSTM hidden size. As an alternative to HyperBand optimization, three values were evaluated for each parameter in order to find a trend in performance and determine an optimal value. These three parameters were chosen as they were expected to have the highest impact on model performance. It was concluded that for the number of epochs and the learning rate resp. fifty and 0,001 were near optimal values. As for hidden size, no concrete number was chosen, however it was found that higher hidden sizes, resulted in better performance. Additionally, the effect of activation functions was assessed. In related work, ReLU activations are used to activate LSTM outputs. However, it was found that model performance decreases, although by a small margin, if ReLU activations are used. Tanh activations on the final output proved to be non-influential towards performance.

In the following sections, each model was evaluated individually based on prediction accuracy, convergence, error consistency and inference time. The Encoder-Decoder LSTM was able to accurately predict at least one minute of future pitch and roll values with consistent error of approximately 2°. The sequential CNN did not perform very well as it lacked an LSTM module. It showed poor convergence and suffered from overfitting. The CNN LSTM ensemble models on the contrary did perform exceedingly well, only needing five seconds to predict a one-minute window with consistent errors of around 1,7°. However, they had an extremely high inference time. As an attempt to lower inference time, they were also trained with grayscale images, but this only resulted in marginally lower inference time and even decreased accuracy on augmented data. In conclusion, the CNN LSTM ensemble models achieved most accurate predictions. However, they have high latency and quickly lose accuracy when objects are in frame of the camera. For these reasons, the **Encoder-Decoder LSTM** is proposed as final and optimal model architecture.

7 Conclusion

The objective of this thesis was to research and implement a method for ship motion prediction using an on-board IMU motion sensor and a front facing camera. The predictions should be optimally designed to aid in take-off and landing for drones on a vessel. The drone should be able to anticipate movements of the vessel based on these predictions to reduce landing impact and ensure safety of the drone and its surroundings. Finally, the method should be capable of real-time low latency predictions on a continuous data stream with minimal hardware requirements.

To define ship motion, six degrees of freedom were used: pitch, roll, yaw, heave, sway and surge. Due to the nature of the problem, pitch and roll were chosen as primary prediction targets as they are most influential towards ship stability and cannot be controlled. With these targets, ship motion was defined as a sequence of pitch and roll values. In order to predict these sequences, five deep learning neural network architectures were designed and evaluated in a simulated environment. Empirical results show that models with an LSTM module achieve higher accuracy and that it is possible to achieve low prediction error with any combination of the available input types – pitch and roll and/or images. However, including images in the input of the model resulted in an increase in latency. Additionally, the use of images introduces problems when visibility is low or when objects like other vessels are in frame. Based on these results, the Encoder-Decoder LSTM was selected as the optimal neural network architecture. It achieves high accuracy, low latency and has no dependency on image clarity.

The objective of the thesis was reached but not to the full extent. Ship motion prediction was achieved in an accurate and low latency manner. However, real-time predictions were not tested. In addition, due to the late availability of real data, only a simulated dataset was used. The proposed models show promising results, but further research is necessary to fully develop the system for deployment.

The key contributions of this thesis were:

- Analysis of the simulated dataset for better interpretation of results
- Design and implementation of five deep neural network architectures for ship motion prediction on a simulated dataset of pitch and roll signals and images
- Optimization of the following hyperparameters: number of epochs, learning rate, LSTM hidden size, input sequence length and output sequence length
- Thorough evaluation of performance of each model architecture based on convergence and prediction accuracy, consistency and latency
- Experimentation with augmented data to close the gap between simulated and real environments

7.1 Future work

For future work, the main goal is to transition to real data. In the first place, a very generous dataset would have to be collected on the vessel of deployment. In order for the model to generalize well to the vessel's behavior, substantial amounts of data of many different sea conditions are needed. This data will have to be cleaned and sampled to an optimal rate. Additionally, more motion parameters could be added to the predicted features to further aid in impact reduction and anticipation. In this thesis, only pitch and roll were predicted. However, heave – the upward and downwards motion along a vertical axis – also plays a key role in smoothening the drone landing. Other metrics such as linear and angular acceleration could also prove to be useful for better predictions. All these metrics are captured by the on-board IMU sensor.

Besides the transition to real, more optimizations can be done on the model architectures. The models using both a CNN and LSTM module achieved overall highest prediction accuracy. However, they suffered from high inference times which made them less appealing. Further optimizing could be performed to lower their computational demand and decrease their latency. For example, the slow concatenation of the PR values and the image feature vectors could be replaced by a

more efficient operation. As of right now, this operation is on average four times slower on a cuda device compared to a CPU due a bug in the PyTorch tensor concatenation function. With a lower inference time, the CNN LSTM models could potentially outperform the Encoder-Decoder LSTM. Additionally, hyperparameter optimization could be performed with specially designed algorithms such as HyperBand (Li et al, 2018) to fully optimize hyperparameters for each model individually. As a final suggestion for future work, the real-time prediction behavior of each architecture could be evaluated. If predictions are accurate enough, it could be possible that the model only needs to predict at for example a five or ten second interval rather than at the frequency of the data stream (2Hz for simulated data). Ideally, a model would only have to make a prediction at the end of the previous prediction's duration to achieve minimal computations.

References

- Abujoub, S. (2019). *Development of a Landing Period Indicator and the use of Signal Prediction to Improve Landing Methodologies of Autonomous Unmanned Aerial Vehicles on Maritime Vessels*. Carleton University.
- Boser, B. E., Guyon, I. M., & Vapnik, V. N. (1992). Training algorithm for optimal margin classifiers. *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, 144–152. <https://doi.org/10.1145/130385.130401>
- Brownlee, J. (2017). *Gentle Introduction to the Adam Optimization Algorithm for Deep Learning*. Machinelearningmastery. <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/>
- Brownlee, J. (2018). *Multi-Step LSTM Time Series Forecasting Models for Power Usage*. Deep Learning for Time Series. <https://machinelearningmastery.com/how-to-develop-lstm-models-for-multi-step-time-series-forecasting-of-household-power-consumption/>
- Canziani, A., Paszke, A., & Culurciello, E. (2016). *An Analysis of Deep Neural Network Models for Practical Applications*. <https://doi.org/10.48550/arxiv.1605.07678>
- Chen, J., Benesty, J., Huang, Y., & Doclo, S. (2006). New insights into the noise reduction Wiener filter. *IEEE Transactions on Audio, Speech and Language Processing*, 14(4), 1218–1233. <https://doi.org/10.1109/TSA.2005.860851>
- Claesen, M., & de Moor, B. (2015). *Hyperparameter Search in Machine Learning*. <https://doi.org/10.48550/arxiv.1502.02127>
- Cui, Z., Ke, R., Pu, Z., & Wang, Y. (2020). Stacked bidirectional and unidirectional LSTM recurrent neural network for forecasting network-wide traffic state with missing values. *Transportation Research Part C: Emerging Technologies*, 118. <https://doi.org/10.1016/J.TRC.2020.102674>
- de Cubber, G. (2019). *OPPORTUNITIES AND SECURITY THREATS POSED BY NEW TECHNOLOGIES*.
- de Masi, G., Gaggiotti, F., Bruschi, R., & Venturi, M. (2011). Ship motion prediction by radial basis neural networks. *IEEE SSCI 2011 - Symposium Series on Computational Intelligence - HIMA 2011: 2011 IEEE Workshop on Hybrid Intelligent Models and Applications*, 28–32. <https://doi.org/10.1109/HIMA.2011.5953967>
- Dhanya, J., & Raghukanth, S. T. G. (2018). Ground Motion Prediction Model Using Artificial Neural Network. *Pure and Applied Geophysics*, 175(3), 1035–1064. <https://doi.org/10.1007/S00024-017-1751-3/FIGURES/22>
- Doshi Sanket. (2019). *Various Optimization Algorithms For Training Neural Network*. Towards Data Science. <https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6>
- Fossen, S., & Fossen, T. I. (2018). EXogenous Kalman filter (XKF) for Visualization and Motion Prediction of Ships using Live Automatic Identification System (AIS) Data. *Modeling, Identification and Control*, 39(4), 233–244. <https://doi.org/10.4173/MIC.2018.4.1>
- Geifman, A. (2020). *How to Measure Inference Time of Deep Neural Networks / Deci*. Deci.Ai. <https://deci.ai/blog/measure-inference-time-deep-neural-networks/>
- Grewal, M. S., & Andrews, A. P. (2010). Applications of Kalman Filtering in Aerospace 1960 to the Present. *IEEE Control Systems*, 30(3), 69–78. <https://doi.org/10.1109/MCS.2010.936465>
- Guan, B., Yang, W., Wang, Z., & Tang, Y. (2018). Ship roll motion prediction based on ℓ_1 regularized extreme learning machine. *PLoS ONE*, 13(10). <https://doi.org/10.1371/JOURNAL.PONE.0206476>
- Ham, S.-H., Roh, M.-I., & Zhao, L. (2017). *Integrated method of analysis, visualization, and hardware for ship motion simulation*. <https://doi.org/10.1016/j.jcde.2017.12.005>

- He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2016-December*, 770–778. <https://doi.org/10.48550/arxiv.1512.03385>
- Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735–1780. <https://doi.org/10.1162/NECO.1997.9.8.1735>
- Johansen, T. A., & Fossen, T. I. (2017). The eXogenous Kalman Filter (XKF). *International Journal of Control*, 90(2), 177–183. <https://doi.org/10.1080/00207179.2016.1172390>
- Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME--Journal of Basic Engineering*, 82(Series D), 35–45.
- Kaminskiy, N.-M. (2019a). *Deep learning models for ship motion prediction from images*.
- Kaminskiy, N.-M. (2019b). *Ship Ocean Simulation in Blender*. GitHub.Com. https://github.com/Nazotron1923/ship-ocean_simulation_BLENDER
- Kingma, D. P., & Ba, J. L. (2014). Adam: A Method for Stochastic Optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. <https://doi.org/10.48550/arxiv.1412.6980>
- Kingma, D. P., & Welling, M. (2019). An Introduction to Variational Autoencoders. *Foundations and Trends in Machine Learning*, 12(4), 307–392. <https://doi.org/10.1561/22000000056>
- Kouris, A., Venieris, S. I., Rizakis, M., & Bouganis, C.-S. (2019). *Approximate LSTMs for Time-Constrained Inference: Enabling Fast Reaction in Self-Driving Cars*. www.imperial.ac.uk/intelligent-digital-systems/approx-lstms/
- Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2016). Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research*, 18, 1–52. <https://arxiv.org/abs/1603.06560v4>
- Li, L., Jamieson, K., Rostamizadeh, A., & Talwalkar, A. (2018). Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research*, 18, 1–52. <http://jmlr.org/papers/v18/16-558.html>
- Lopez Pinaya, W. H., Vieira, S., Garcia-Dias, R., & Mechelli, A. (2020). Autoencoders. *Machine Learning: Methods and Applications to Brain Disorders*, 193–208. <https://doi.org/10.48550/arxiv.2003.05991>
- Luo, F. L., Unbehauen, R., & Cichocki, A. (1997). A Minor Component Analysis Algorithm. *Neural Networks*, 10(2), 291–297. [https://doi.org/10.1016/S0893-6080\(96\)00063-9](https://doi.org/10.1016/S0893-6080(96)00063-9)
- "MarLand." (2020). *MarLand – Robotics & Autonomous Systems*. <https://mecatron.rma.ac.be/index.php/projects/marland/>
- "MarSur." (2019). *MarSur – Robotics & Autonomous Systems*. <https://mecatron.rma.ac.be/index.php/projects/marsur/>
- Mealey, T., & Taha, T. M. (2018). Accelerating Inference in Long Short-Term Memory Neural Networks. *Proceedings of the IEEE National Aerospace Electronics Conference, NAECON, 2018-July*, 382–390. <https://doi.org/10.1109/NAECON.2018.8556674>
- Minsky, M., & Papert, S. A. (1969). Perceptrons: An Introduction to Computational Geometry. In *Perceptrons*. The MIT Press. <https://doi.org/10.7551/MITPRESS/11301.001.0001>
- Nayfeh, A. H., Mook, D. T., & Marshall, L. R. (2012). Nonlinear Coupling of Pitch and Roll Modes in Ship Motions. *https://doi.org/10.2514/3.62949*, 145–152. <https://doi.org/10.2514/3.62949>
- Peng, X., Zhang, B., & Rong, L. (2019). A robust unscented Kalman filter and its application in estimating dynamic positioning ship motion states. *Journal of Marine Science and Technology (Japan)*, 24(4), 1265–1279. <https://doi.org/10.1007/S00773-019-00624-5>

- Perez, T., & Blanke, M. (2017). Ship Roll Damping Control. *Annual Reviews in Control*, 36(1), 129–147. <https://doi.org/10.1016/j.arcontrol.2012.03.010>
- Perez, T., & Fossen, T. I. (2005). Kinematics of ship motion. *Advances in Industrial Control*, 9781852339593, 45–58. https://doi.org/10.1007/1-84628-157-1_3
- Portilla, J. (2019). *PyTorch for Deep Learning with Python*. Udemy.Com. <https://www.udemy.com/course/pytorch-for-deep-learning-with-python-bootcamp/#instructor-1>
- Q. Judge, C. (2019). Ship motion in waves. In *Seakeeping and Maneuvering*. https://www.usna.edu/NAOE/_files/documents/Courses/EN455/AY20_Notes/EN455CourseNotesAY20_FrontMaterial.pdf
- Ran, T., Tong, S., Yang, Y., & Zhang, H. (2021). Research and Application on Mathematical Model of Ship Manoeuvring Motion under Shallow water effect. *IOP Conference Series: Earth and Environmental Science*, 643(1). <https://doi.org/10.1088/1755-1315/643/1/012127>
- Rashid, M. H., Zhang, J., & Minghao, Z. (2021). Real-Time Ship Motion Forecasting Using Deep Learning. *The 2nd International Conference on Computing and Data Science*, 5(2021). <https://doi.org/10.1145/3448734>
- Ren, X., Yang, T., Erran Li, L., Alahi, A., & Chen, Q. (2021). Safety-aware Motion Prediction with Unseen Vehicles for Autonomous Driving. *IEEE Explore*. <https://github.com/>
- Ruder, S. (2016). *An overview of gradient descent optimization algorithms*. <http://caffe.berkeleyvision.org/tutorial/solver.html>
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature* 1986 323:6088, 323(6088), 533–536. <https://doi.org/10.1038/323533a0>
- Rumelhart E., D., Hinton E., G., & Williams J., R. (1986). Learning representations by back-propagating errors. *Letters To Nature*.
- Sharma, S., Sharma, S., & Athaiya, A. (2020). ACTIVATION FUNCTIONS IN NEURAL NETWORKS. *International Journal of Engineering Applied Sciences and Technology*, 4, 310–316. <http://www.ijeast.com>
- Shen Kevin. (2018). *Effect of batch size on training dynamics*. Medium. <https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e>
- Sherstinsky, A. (2018). Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. *Physica D: Nonlinear Phenomena*, 404. <https://doi.org/10.1016/j.physd.2019.132306>
- Silva, M. (2015). Ocean Surface Wave Spectrum. In *Physical Oceanography*. https://www.researchgate.net/publication/283722827_Ocean_Surface_Wave_Spectrum
- Skulstad, R., Li, G., Fossen, T. I., Wang, T., & Zhang, H. (2021). A Co-Operative hybrid model for ship motion prediction. *Modeling, Identification and Control*, 42(1), 17–26. <https://doi.org/10.4173/MIC.2021.1.2>
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56), 1929–1958. <http://jmlr.org/papers/v15/srivastava14a.html>
- Stewart, R. H. (2008). *Introduction to Physical Oceanography*. <http://www.madsci.org/posts/archives/2004-11/1101806651.Es.r.html>
- Stock, J. H., & Watson, M. W. (2001). Vector Autoregressions. *Journal of Economic Perspectives*, 15(4), 101–115. <https://doi.org/10.1257/JEP.15.4.101>

- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., & Rabinovich, A. (2014). Going Deeper with Convolutions. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 07-12-June-2015*, 1–9. <https://doi.org/10.48550/arxiv.1409.4842>
- Tang, Y., Ma, L., Liu, W., & Zheng, W. S. (2018). Long-Term Human Motion Prediction by Modeling Motion Context and Enhancing Motion Dynamic. *IJCAI International Joint Conference on Artificial Intelligence, 2018-July*, 935–941. <https://doi.org/10.48550/arxiv.1805.02513>
- Wang, Q., Ma, Y., Zhao, K., & Tian, Y. (2022a). A Comprehensive Survey of Loss Functions in Machine Learning. *Annals of Data Science*, 9(2), 187–212. <https://doi.org/10.1007/s40745-020-00253-5>
- Wang, Q., Ma, Y., Zhao, K., & Tian, Y. (2022b). A Comprehensive Survey of Loss Functions in Machine Learning. *Annals of Data Science*, 9(2), 187–212. <https://doi.org/10.1007/s40745-020-00253-5/TABLES/8>
- Wei, Y., Chen, Z., Zhao, C., Chen, X., Tu, Y., & Zhang, C. (2022). Big multi-step ship motion forecasting using a novel hybrid model based on real-time decomposition, boosting algorithm and error correction framework. *Ocean Engineering*, 256, 111471. <https://doi.org/10.1016/J.OCEANENG.2022.111471>
- Wilson, D. R., & Martinez, T. R. (2001). The need for small learning rates on large problems. *Proceedings of the International Joint Conference on Neural Networks*, 1, 115–119. <https://doi.org/10.1109/IJCNN.2001.939002>
- Wu, J. (2017). *Introduction to Convolutional Neural Networks*.
- Zhang, T., Zheng, X. Q., & Liu, M. X. (2021). Multiscale attention-based LSTM for ship motion prediction. *Ocean Engineering*, 230. <https://doi.org/10.1016/J.OCEANENG.2021.109066>
- Zhao, X., Xu, R., & Kwan, C. (2004). Ship-motion prediction: Algorithms and simulation results. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 5. <https://doi.org/10.1109/ICASSP.2004.1327063>
- Zhong-yi, Z. (2012). An adaptive ship motion prediction method based on parameter estimation. *Journal of Ship Mechanics*.

Appendix A. Individual predictions and LPF

For each multi-step model, a prediction was plotted for one IO-sequence. The results of these predictions are added in this appendix because they provide a good visual representation of how accurate a model is. The predicted sequence is always shown in red on top of the real sequence in blue. The specific IO-ratio and model that was used is mentioned in the caption. **Pitch** will always be shown in the **top** graph and **roll** in the **middle** graph. At the **bottom**, the LPF graph is shown.

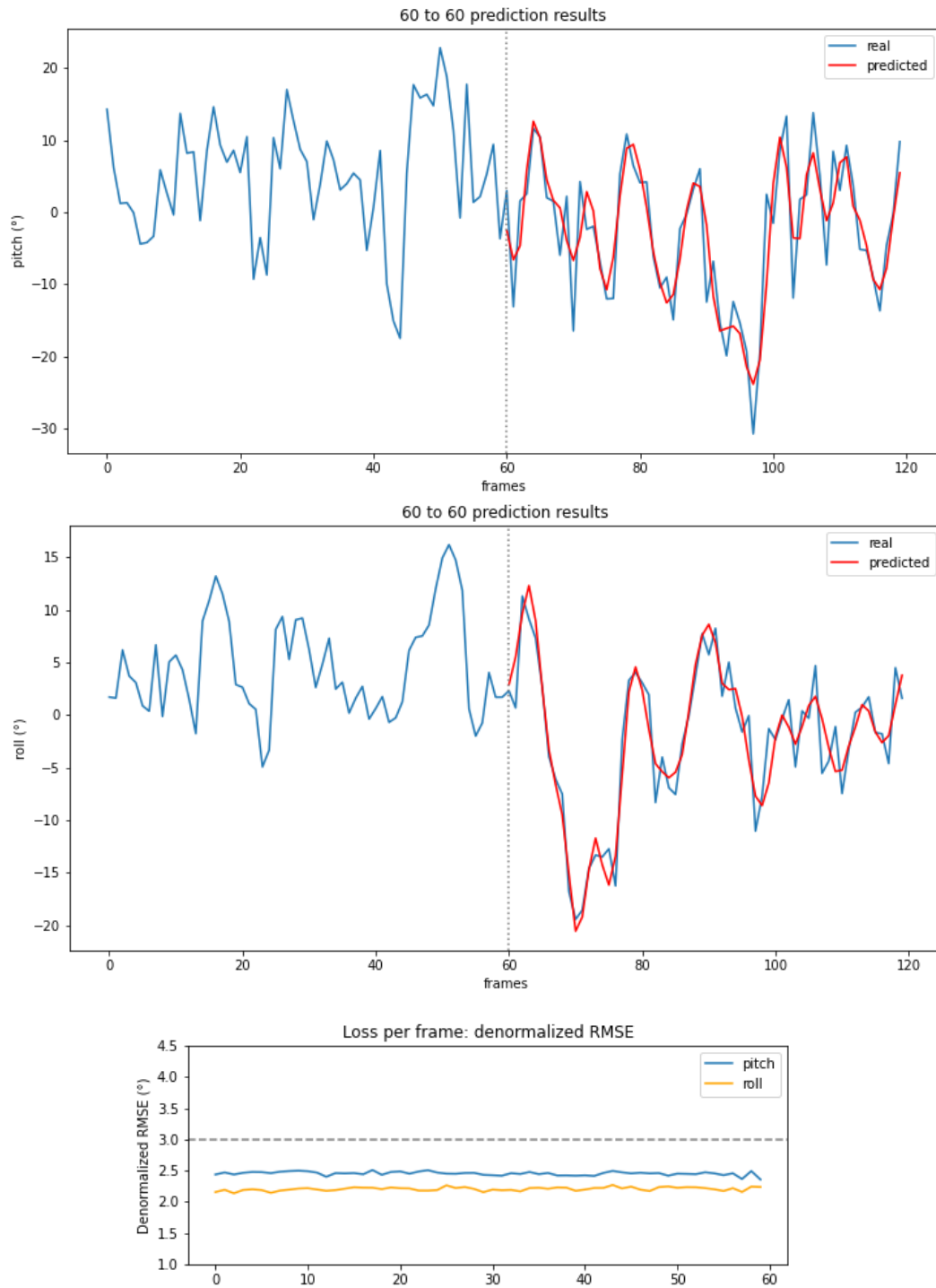


Figure A-1: Encoder-Decoder LSTM - 60/60

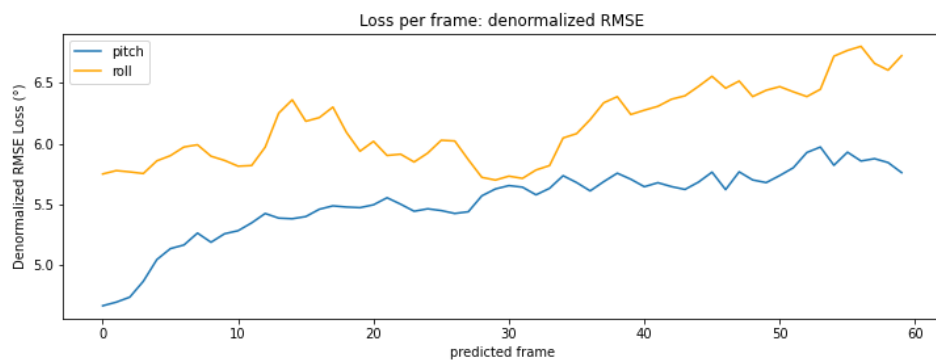
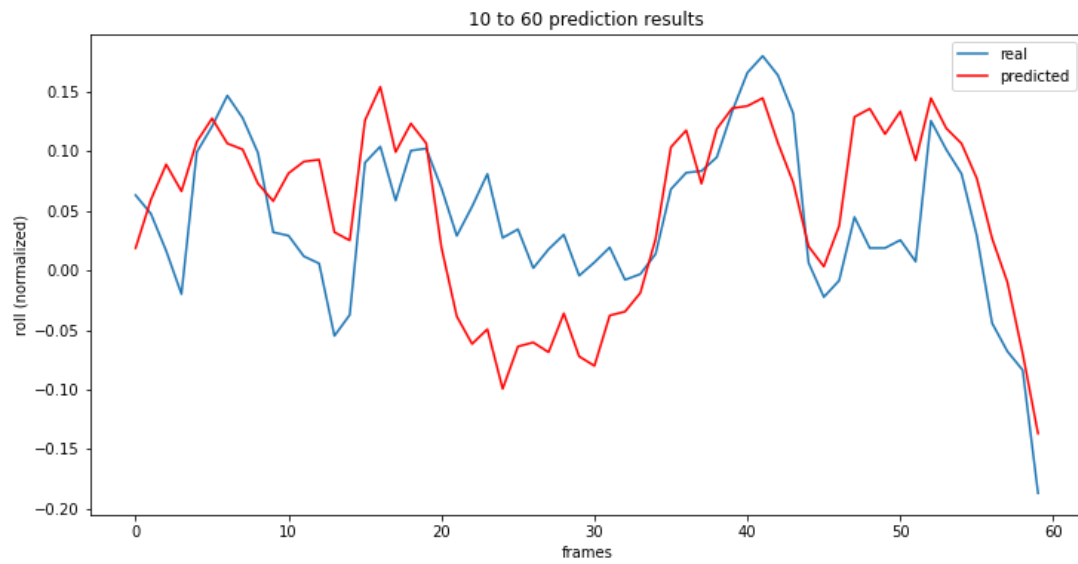
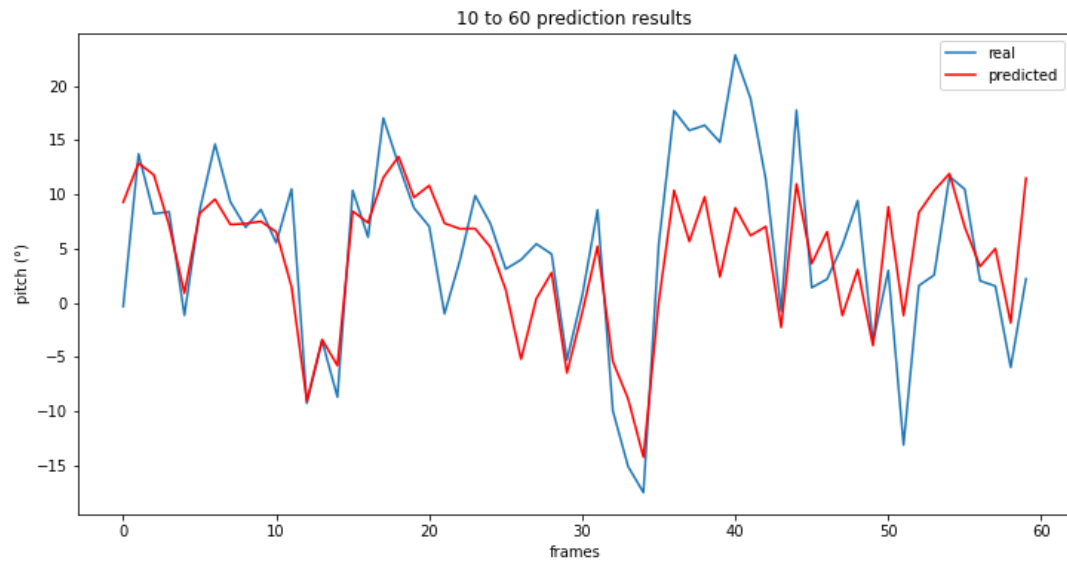


Figure A-2: Sequential CNN - 10/60

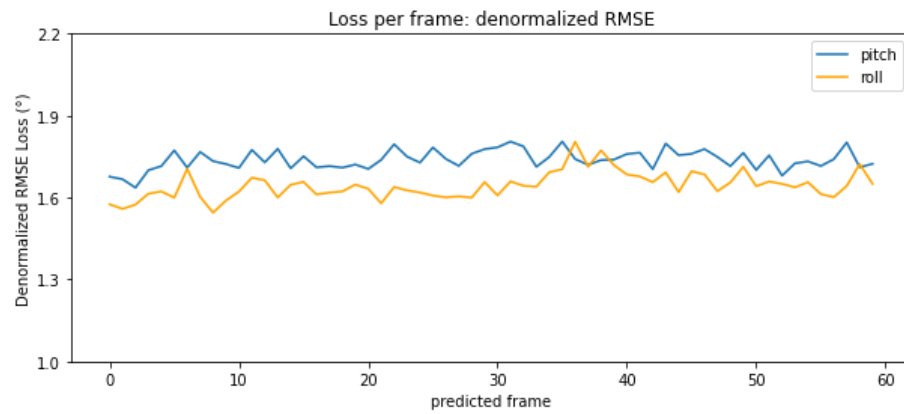
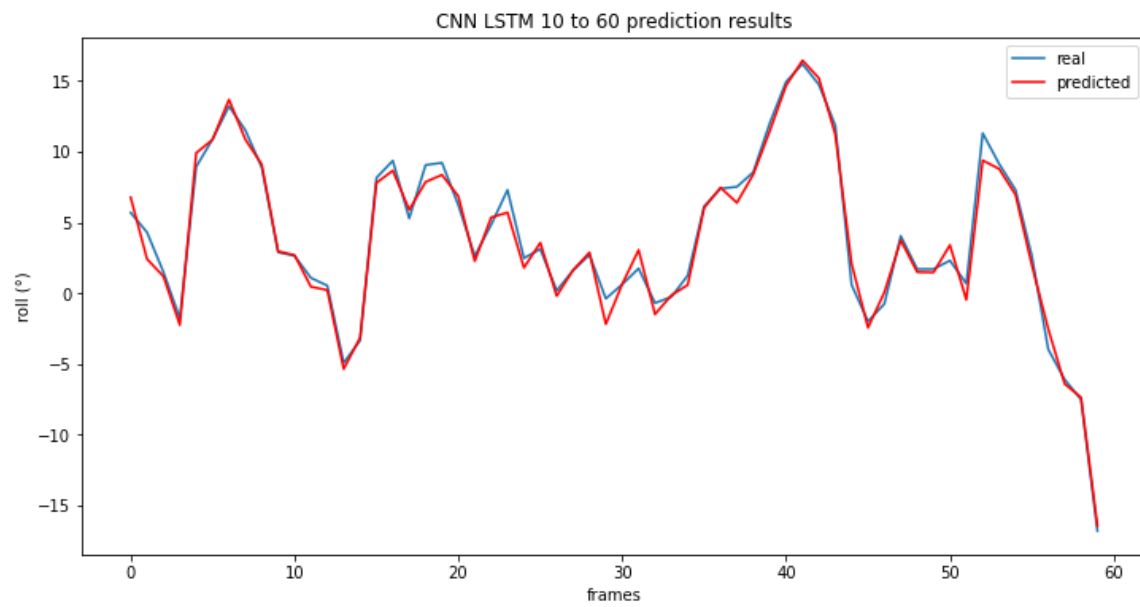
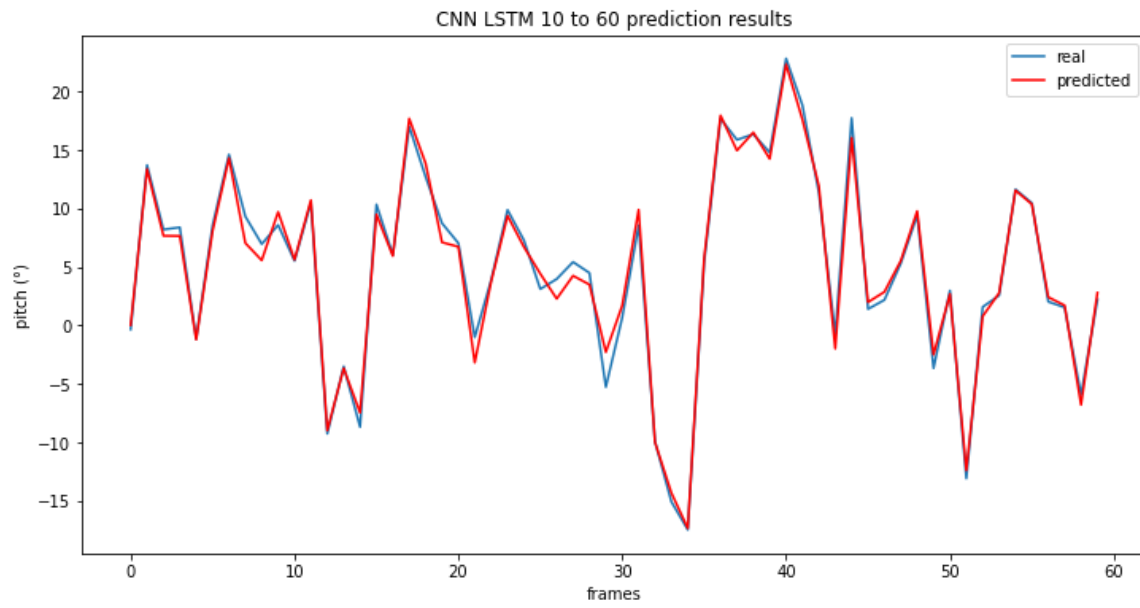


Figure A-3: CNN LSTM single input (colored) - 10/60

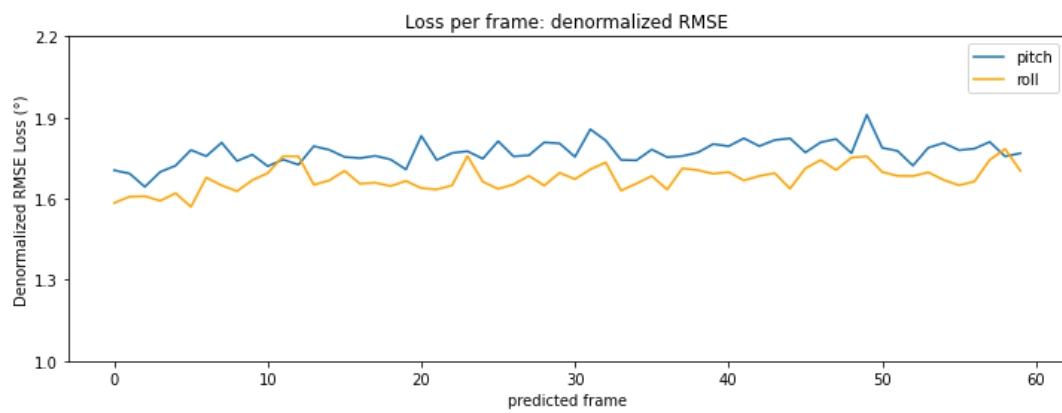
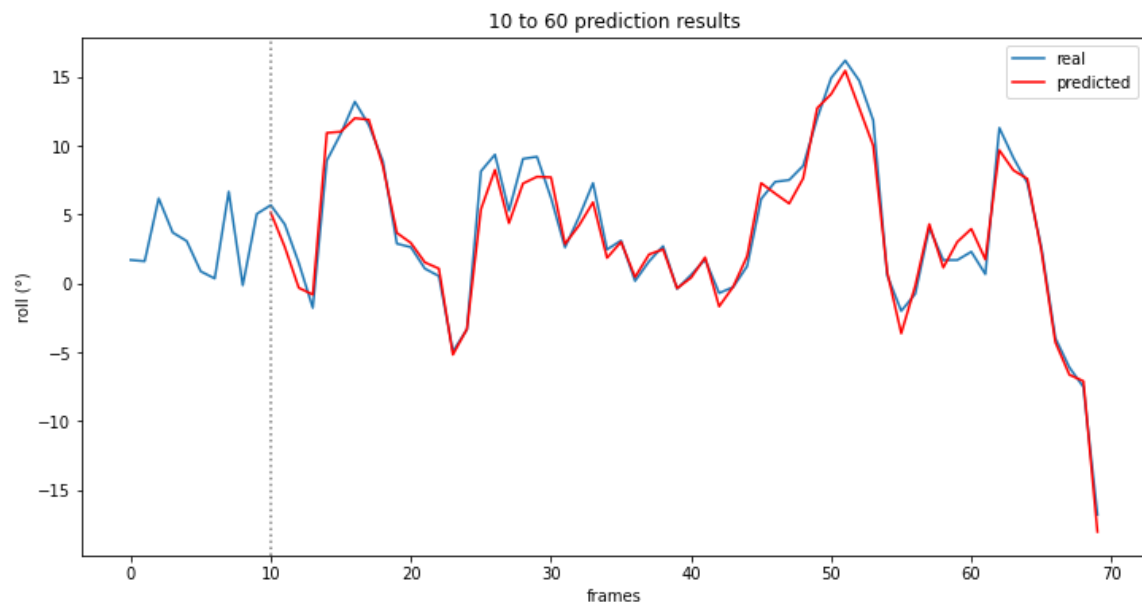
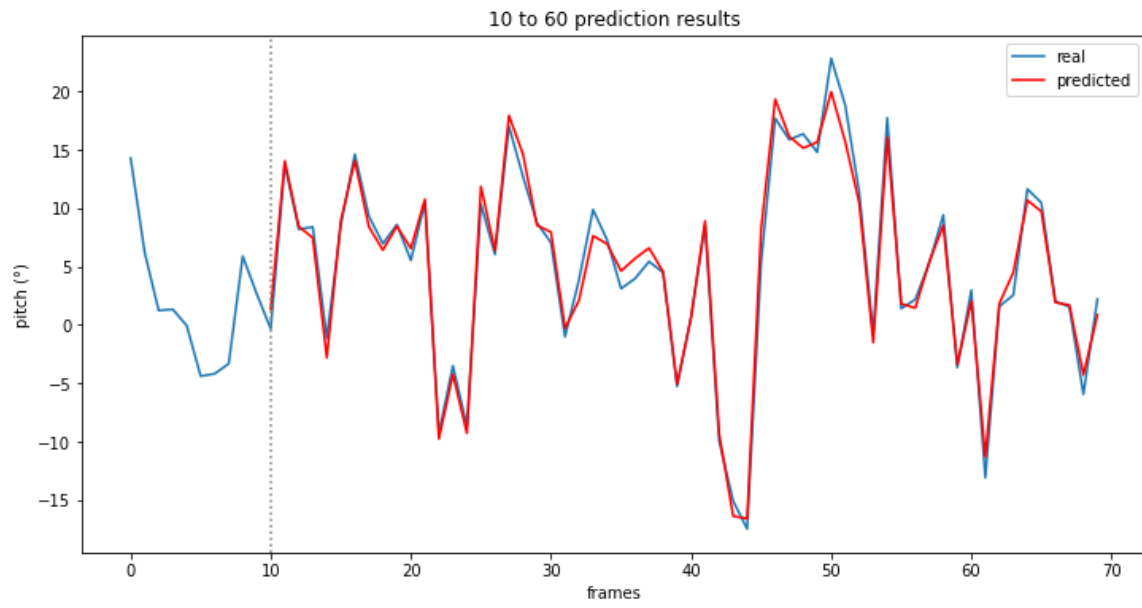


Figure A-4: CNN LSTM dual input (colored) - 10/60

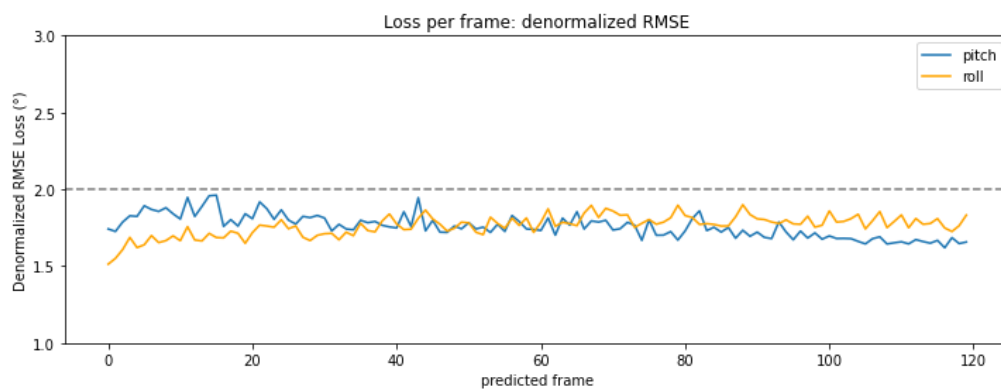
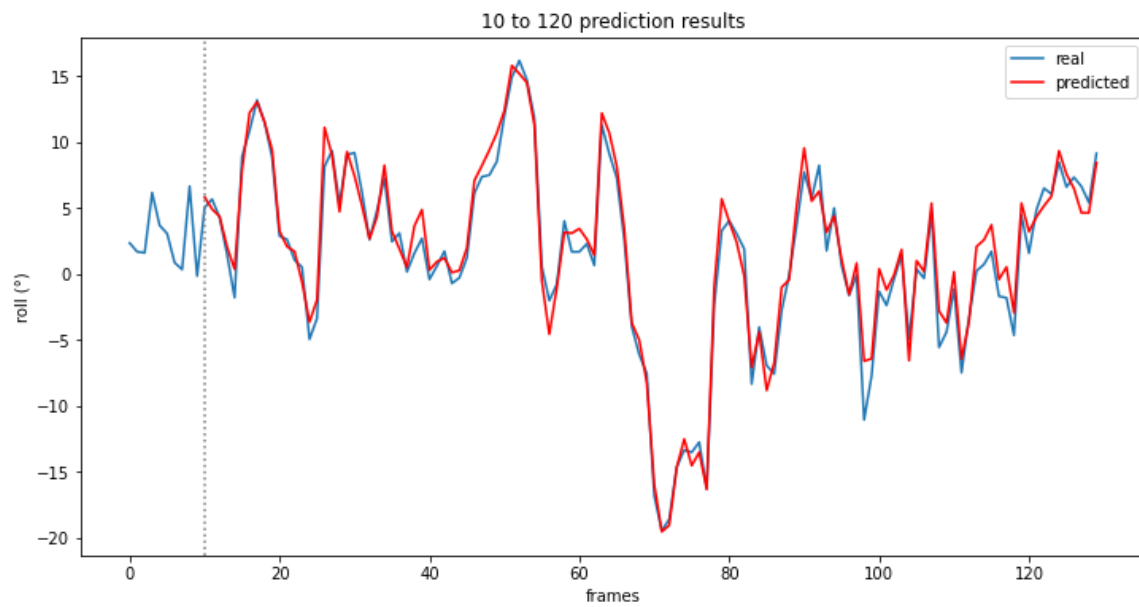
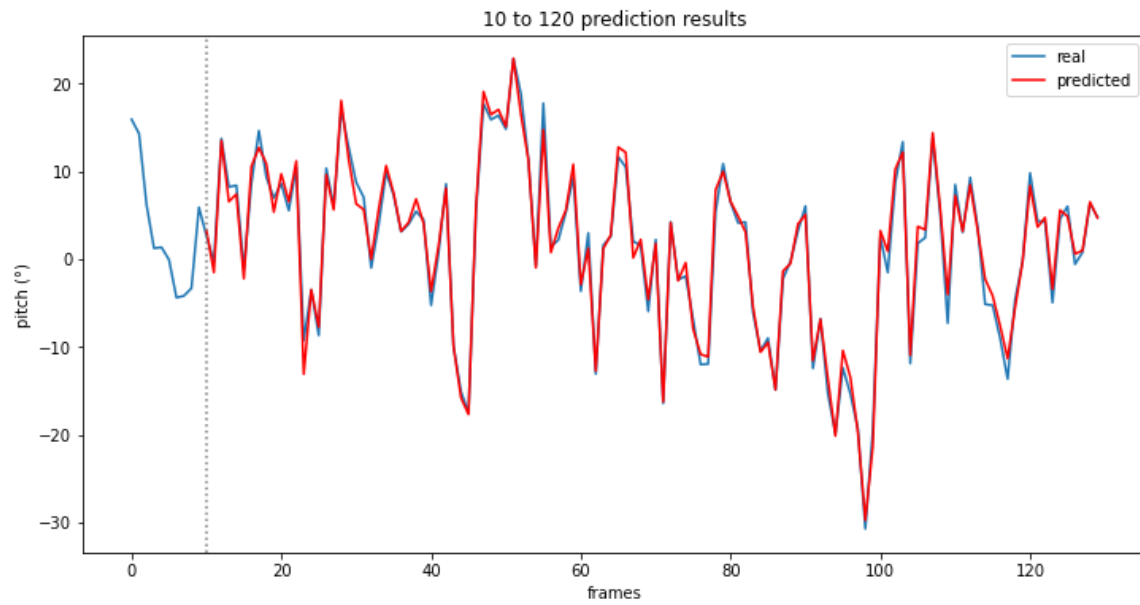


Figure A-5: CNN LSTM dual input (colored) - 10/120