University of Sheffield

Implementing Inverse Kinematics with a Neural Network

José Casimiro Revez

Supervisor: Stuart Wilson

A report submitted in fulfilment of the requirements for the degree of BSc in Computer Science

in the

Department of Computer Science

May 8, 2024

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: José Casimiro Revez

Signature:

Date:30/11/2023

,

Abstract

Inverse Kinematics is a problem in which we aim to obtain joint angles, from a desired end-effector position. This has been a prevalent problem in the field of robotics for decades as people are continuously trying to find ways to optimize a solution.

In this paper I will explore and analyze different approaches to solving this problem, starting with classical solutions, such as Tejomurtula and Kak's architecture, and slowly making my way to more state-of-the-art solutions that will subsequently be implemented and tested.

The solution proposed in this paper involved the implementation of a Forward Kinematics (FK) Model and Back Propagating Neural Network (BPNN), from scratch. These were then combined to have a Neural Network that solves the Forward Kinematics problem, and finally, this final Neural Network is inverted. Further research was done to ascertain how complex a robotic arm needs to be for the neural network to produce a small enough error so that the Neural Network can be reliably used to solve the Inverse Kinematics problem. '

Acknowledgements

I would like to thank my supervisor Stuart Wilson for all his help throughout the project, both with the actual project and by just making himself available for any problems and hardships that I was going through during the duration of this project. He was a quintessential part of the design process and ultimately, the sole reason I was able to finish this project.

Contents

List of Figures

Chapter 1

Introduction

Inverse Kinematics is one of the most important aspects of robot control systems as it enables us to work out the joint angles we need to reach a desired end effector position. Finding a solution for this problem is incredibly helpful as it has uses in almost every field: in medicine, this can be used to build robotic arms capable of precise surgeries that could not otherwise be performed by humans due to the risk; and in manufacturing, to improve the performance of robotic arms and subsequently production, just to name a couple. Building a good solution to this problem can be extremely complex as there are cases with no solutions, as well as cases with multiple solutions. This has made it so researchers are continuously trying to find more efficient ways of calculating the possible joint angles that accomplish a desired end effector position. The most popular approaches to this are algebraic, geometric, and iterative.[5]

The algebraic approach is the original Kinematics model that uses the Cartesian space and works by multiplying a 4x4 transform matrix with the desired end effector position to get the joint positions and use inverse trigonometric functions to work out the possible joint angles

The geometric approach[6, 7] is better suited for robotic arms with at most 2 degrees of freedom as anything more than that usually leads to trigonometric equations that are too complex. For instance, this approach can be useful for working out joint angles in a robotic arm with 2 links in a three-dimensional space by working in the plane created by the two lines, connecting them as a triangle, and using the law of cosines to calculate joint angles.

The iterative approach is the one that has the most flexibility as it calculates the angles until the desired end effector position is reached. Examples of this include the Cyclic Coordinate Descent[8], FABRIK[9] and Neural Networks. Most of these, however, are designed for 2D space, with most recent solutions being based on Neural Networks that need a lot of data and hidden layers to get an accurate solution. With this project, I aim to build a Neural Network solution for the Inverse Kinematics problem in 3D space that does not require a lot of training data and reduces the number of dimensions needed for the Neural Network.

1.1 Aims and Objectives

As mentioned before, my aim with this project is to build a simple Neural Network solution for the Inverse Kinematics problem, that cuts down on the amount of training data needed, as well as the amount of hidden layers in the Neural Network. I will implement this using Python and the library matplotlib to handle a 3D simulation of what I want to achieve.

With the successful implementation of this Neural Network solution, I hope to better understand how the complexity (number of joints) of a robotic arm can affect the learning capability of a Neural Network and whether or not a simple Neural Network can be realistically used to solve the Inverse Kinematics problem. The success of this project will be characterized by how well the Neural Network learns and whether or not it can be scaled in order to provide an Inverse Kinematics solution for robotic arms with multiple DOF.

1.2 Dissertation Overview

In the following chapters of this dissertation, I will extensively cover the literature on this topic, discussing different approaches to solving this problem and what is and is not useful for the purpose of this project. I will then go over the requirements and formally analyze the problem, discussing the final Neural Network structure that was decided on, how this was implemented and tested, and finally, discussing the results obtained, the success of the Neural Network implemented, and what could be done in the future (if anything) to make these results better.

Chapter 2

Literature Survey

In Forward Kinematics, when we want to move a robotic arm with 1-DOF (a singular joint) a certain amount, a 4x4 transform matrix is constructed based on the joint axis and the angle of rotation and then multiplied by the endpoint of the robotic arm to obtain the end effector position. This is to say, forward kinematics is transforming joint-space values into Cartesian-space values. This can be given by the following equation:[1]

$$r(t) = f(\theta(t)) \tag{2.1}$$

where $\theta(t)$ is an m-vector of joint variables, $r(t)$ is an n-vector of Cartesian variables, and $f(.)$ is a continuous nonlinear function whose structure and parameters are known. The inverse kinematics problem is to find the inverse mapping of equation (2.1):[1]

$$\theta(t) = f^{-1}(r(t)) \tag{2.2}$$

This problem depends on the existence of a unique solution and the effectiveness and efficiency of solution methods, thus making this much more convoluted and complex than the forward kinematics problem.[1] A lot of research has been put into this over the years in order to make the calculation process for this problem more efficient and accurate and many good solutions have been developed. Lots of these solutions, especially recently, have been centered on Neural-Network-based approaches. Going over a few of these, starting with old findings and initial solutions and making my way to the state of the art, I will go over each proposed solution's advantages and limitations, as well as how these affected and shaped my final implementation.

2.1 Early Findings That Paved The Way

In a conference from 1994, a recurrent neural network for computing pseudoinverse Jacobian matrices is proposed. This works by taking into account the following equation for the joint angle:

$$\theta(t) = J^{+}(e(t))w + [I - J^{+}(e(t))J(e(t))]k(t) \tag{2.3}$$

where $J^{+}(e(t))$ is the pseudoinverse of J(e(t)), I is an identity matrix, and k(t) is an m-vector of arbitrary time-varying variables.[1]

The proposed solution, assuming that at each sample interval, the Jacobian is approximately constant, computes the pseudoinverse of the Jacobian using a Dynamical Equation, that, under certain conditions can be simplified and defined as:

$$\dot{X} = -\mu_1 J^{\mathrm{T}} U J^{\mathrm{T}} - \mu_2 (2VJX - V) \tag{2.4}$$

This equation is equivalent to following the Neural Network structure:

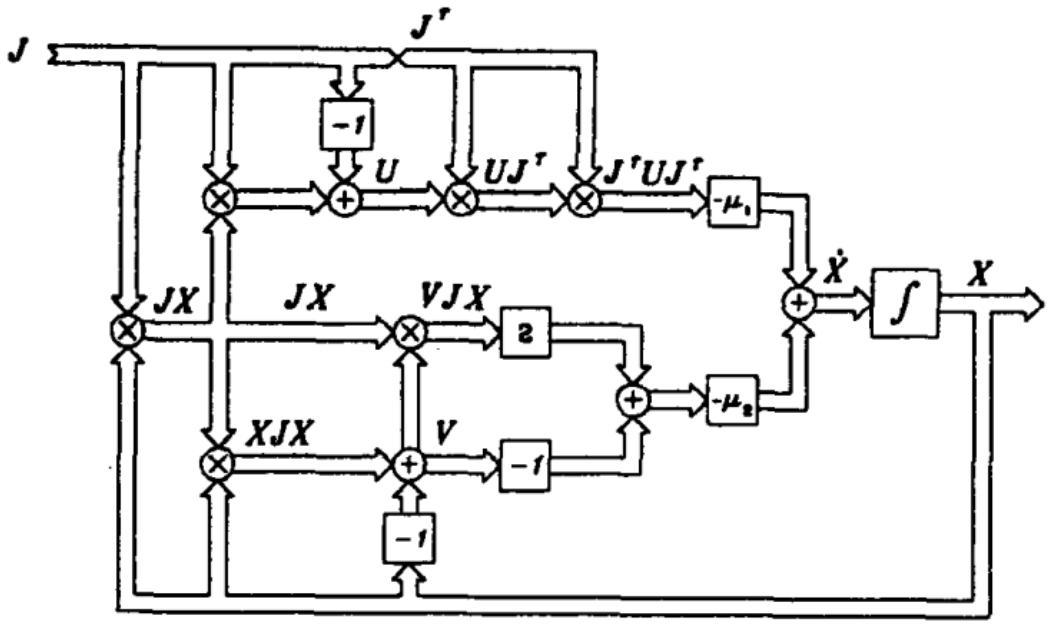

Figure 2.1: Neural Network structure as proposed by Wu and Wang[1]

Although it is capable of solving the Inverse Kinematics problem according to the simulations provided, it does so after making a lot of assumptions which in simulation space are easy to satisfy, but not so much when trying to use this in real life.

Another approach proposed later on was the error back-propagation Neural Network.[2] This works by deciding the amount of hidden layers in advance and connecting all the layers

in the Neural Network to each other. The weights of these connections are chosen at random and after the first pattern is fed, the result is compared to the desired output, and the error is propagated back. The weights are adjusted iteratively until it falls below a certain threshold. This, however, would take too long to train every time for it to be used in real-time.

With this in mind, Tejomurtula and Kak proposed a new network architecture for error back-propagation. They proposed that, since most of the calculations are made in joint space, some network weights be made of trigonometric functions as opposed to the conventional real numbers. The neural-network structure is as follows:[2]

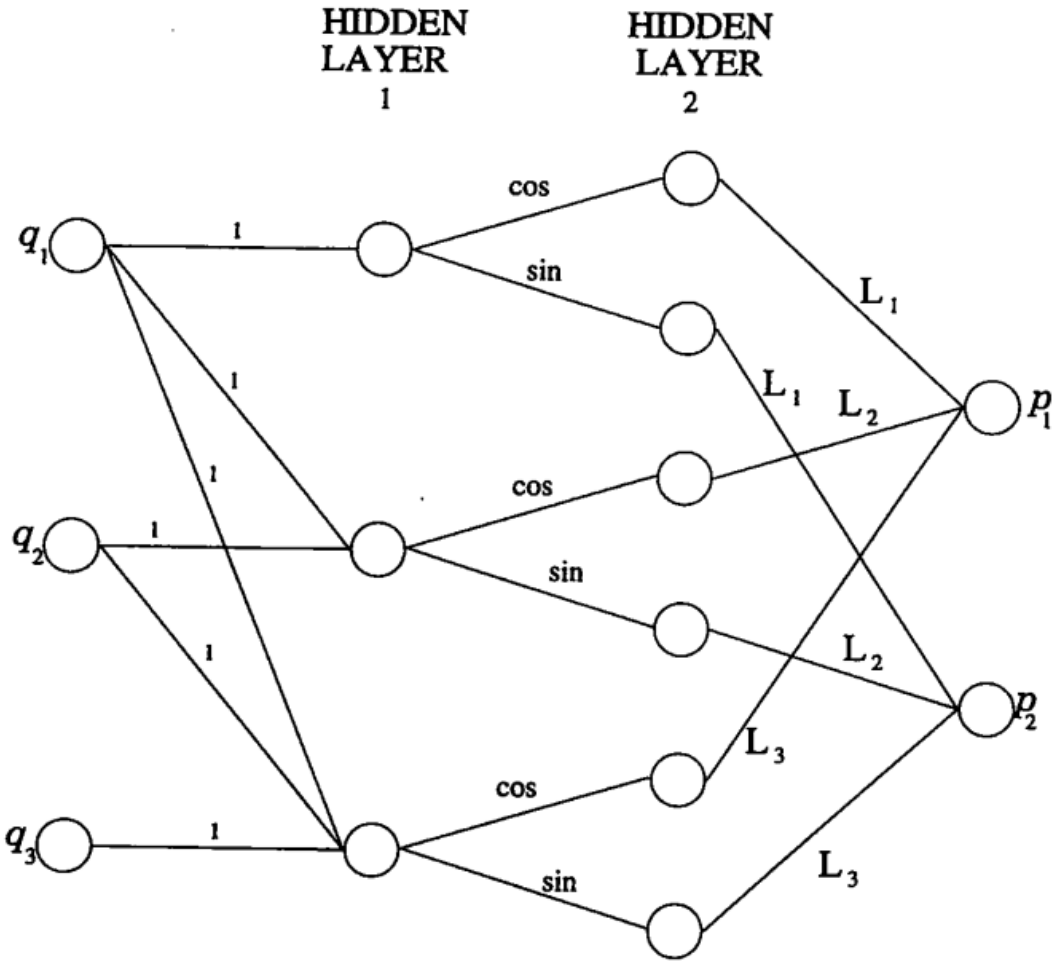

Figure 2.2: Neural Network structure as proposed by Tejomurtula and Kak[2]

This NN takes the desired end-effector position and joint ranges as inputs, an initial guess is made about the joint angles, and the coordinates of the end-effector pose are calculated according to the guesses. After this, the error is determined according to the desired pose and it is, then, back-propagated through the network according to certain conditions. The final joint values are checked against the joint ranges and adjusted accordingly and the

Forward Kinematics calculation is performed. This is repeated until the error falls below a certain threshold and the final adjusted joint angles are the solution. This implementation gets around the time problem but its efficacy is limited by the initial guesses which makes it better than the conventional back-propagation method, but still not ideal as in real life it is unlikely to reliably produce an accurate output.

One other solution to this problem was the implementation of a modular architecture neural network that learns a discontinuous Inverse Kinematics function by switching between multiple neural networks[10, 11, 12]. This approach differs to the previous ones mentioned as it computes a precise Inverse Kinematics Model for a robotic arm, as opposed to computing a solution for a desired end-effector position, on the fly. Although the training stage for this can be computationally taxing and take a long time, it will always produce a precise model for that robot in more or less the same amount of time, regardless of how many DOF it has. This can be very useful for stationary robots that always work in a never-changing environment as it makes for faster movements and decisions, in real-time. However, it cannot be used reliably in robots that move around or whose environment changes as it would have to compute the IK Model every time the robot is in a new position.

Through the years, multiple Neural Network structures have been proposed[13, 14, 15], all of them with their advantages and limitations, which will be discussed and compared later on. It's also relevant to mention that in 2014, Puheim and Madarász wrote a paper[16] detailing methods to normalize the inputs and outputs of a Neural network as this can improve efficiency and faster error minimization, especially in Neural Networks with simplistic structures.

2.2 More Recent, State-of-the-Art Solutions

Going forward in time now to some more recent papers, I saw that most recent solutions to this problem fell in one of two categories: Feed-Forward Back-Propagation Neural Networks[17, 4, 18, 19, 20] (from hereon referred to as FFBPNN) and Recurrent Neural Networks[21, 22] (from hereon referred to as RNN).

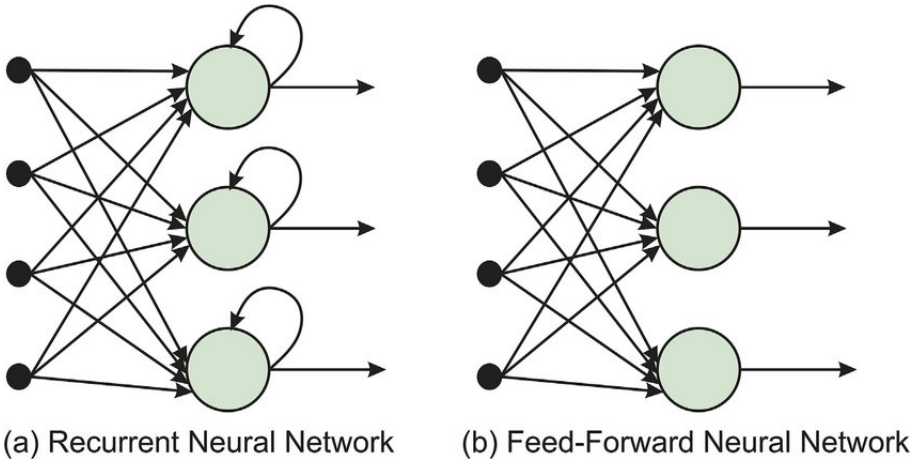

Figure 2.3: Difference Between FFBNN and RNN [3]

2.2.1 Feed-Forward Back-Propagation Neural Networks

FFBPNNs will usually have a more complex neural structure, as they have to have at least one hidden layer, in the most simple of cases(See Figure 2.4), but will most of the time need more layers as the DOF increases. These are usually trained according to the following:

- Setting random input values and weights for the hidden layers.
- Computing the real end-effector pose using Forward Kinematics and setting that as a goal.
- Feeding those values through the NN and obtaining an output.
- Computing the error between the goal and the output and propagating the error back through the NN so as to adjust the initial weights.
- Repeating steps 3 and 4 until the error is below a certain threshold.

Certain solutions use more than one FFBPNN, and use the solutions from the first NN as inputs for the second, and so on.[17, 4] This leads to accurate solutions and faster convergence of the error function, and it works incredibly well for robots with 3-DOF[17, 20] and it is easy to implement for higher DOF as you need only add one neuron to the first layer, for each DOF, without sacrificing efficiency or accuracy[4]

More complex solutions have been proposed also which use an Improved Particle Swarm Optimised FFBPNN[18] and the results showed an error margin, an order of magnitude higher than those obtained from regular FFBPNN, making this an ideal solution for robots that need a high degree of accuracy such as medical robots.

Lastly, I would like to mention Conditional Invertible Neural Networks as well, whose network structure achieves bi-directional mapping between the inputs and outputs, with the ability to perform different computations based on certain conditions[19]. This structure

makes for more efficient computations and an accuracy that fluctuates less than regular Inverse Kinematics Solvers.

In short, FFBPNN solutions are efficient and accurate making them ideal for most robots, with varying degrees of freedom.

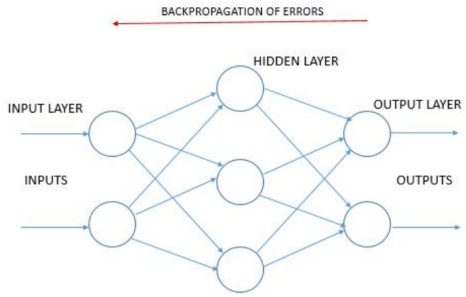

Figure 2.4: Basic Structure of a Feed-Forward Neural Network with back-propagation[4]

2.2.2 Recurrent Neural Networks

RNNs are usually Much simpler in structure since they are recursive, but this, conversely, makes training time much longer than other NNs, while having a higher accuracy overall[21] These work by having an extensive training stage that creates an Inverse Kinematics Model for the robot, i.e. a mapping of joint angles to poses. This model is then used for both Forward and Inverse Kinematics operations as it can quickly and accurately return the joint angles based on the desired pose, etc. These solutions are optimal for robots that are in a never-changing environment, as the training is done for a specific configuration and will need to be redone if there are changes to the environment. This means that an RNN will fall behind in terms of efficiency if the environment changes.

2.3 Summary

Several solutions have been proposed over the years for the Inverse Kinematics problem; This started with inefficient, 2D implementations of robots with fewer DOF and later went on to 3D implementations of both Forward-Feeding Neural Networks with Back Propagation and Recurrent Neural Networks. Even though they are inefficient, I feel it is important to look back at how development started so that we can discuss the limitations that we had back then and how we can get over them now. Nowadays the two most prevalent NN structures to solve the IK problem could not be any more different and it is important to look at and understand how both of them work. Both have their own limitations which make each of them suitable to different robots and scenarios so having a good grasp on how both of these work will help me decide which one to use for my project. On the one end, FBPNNs are reliable and fast in most cases which makes them optimal for making an IK Solver for general robots whose environment is prone to change. On the other end, RNNs are less complex in

structure and, while taking longer to train, have results that are more accurate than even some of the best FBPNNs.

Chapter 3

Requirements and Analysis

Since Kinematics in general was a fairly new topic to me, in order to thoroughly understand Inverse Kinematics, I first needed to understand Forward Kinematics, making the first requirement for this project, the implementation of a Forward Kinematics Model. Following this, the plan was to generate joint angles randomly, for each joint of the robotic arm, putting these through the FK Model to obtain the target outputs and then training the Neural Network with these values.

Looking at the possible Neural Network structures, we can see that each of them has its benefits and limitations, making the initial choice of NN structure the most important part of this project. While at first, the implementation of multiple structures was considered, in order to see which NN structure best handles the IK problem, this idea was quickly discarded due to time limitations as I felt implementing multiple Neural Networks from scratch was not feasible in the amount of time I had. This meant that a specific NN structure had to be chosen and the only requirements were to keep the NN complexity at a minimum, meaning only one hidden layer, while also reducing the amount of training data needed to train the Neural Network.

3.1 Project Plan and How It Will Be Evaluated

As mentioned above, this project starts with the implementation of a Forward Kinematics Model. Forward Kinematics works by calculating a rotation matrix for each joint, taking into account the desired angle of rotation. The rotation matrix for a unit vector $u = (u_{\mathrm{x}}, u_{\mathrm{y}}, u_{\mathrm{z}})$, and an angle of rotation θ can be constructed as follows[23, 24, 25]:

$$R = \begin{bmatrix} \cos\theta + u_x^2\,(1-\cos\theta) & u_x u_y\,(1-\cos\theta) - u_z\sin\theta & u_x u_z\,(1-\cos\theta) + u_y\sin\theta \\ u_y u_x\,(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2\,(1-\cos\theta) & u_y u_z\,(1-\cos\theta) - u_x\sin\theta \\ u_z u_x\,(1-\cos\theta) - u_y\sin\theta & u_z u_y\,(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2\,(1-\cos\theta) \end{bmatrix}.$$

Which can be written more concisely as[26]:

$$R = (\cos\theta)\, I + (\sin\theta)\, [\mathbf{u}]_\times + (1 - \cos\theta)\, (\mathbf{u} \otimes \mathbf{u})$$

With this, we can now create rotation matrices for each joint and desired rotation. This rotation matrix is to be then be multiplied by every point that is downstream from the joint serving as a rotation axis, in order to get a correct rotation of the robotic arm. On a robotic arm with multiple degrees of freedom, there will be a rotation matrix for every joint and these have to be multiplied in the correct sequence, starting with the first joint and moving down to the last, otherwise, the rotation will not be correct.

The Forward Kinematics model can be tested and validated using simple trigonometric functions to see whether or not the rotation matrix is correctly transforming the points.

With this done, an NN structure needs to be settled on. Since the main goal of this project is to cut down on NN complexity, this narrows down the possible options to Recurrent Neural Networks and Feed-Forward Back Propagating Neural Networks, since the structures that predate these two are far too complex and don't meet the requirements for this project. RNNs have a simpler structure when compared to most FFBPNNs, which would make them ideal for this project, were it not for the fact that they use recursive functions as activation functions. This can produce more accurate results but greatly expands the amount of training time needed, as well as the amount of training data. In contrast, the simplest FFBPNNs would have a structure that is as simple as that of RNNs, while allowing for simpler activation functions like the sigmoid function. RNNs are also much harder to fine-tune which means that even though they are simpler in structure, FFBPNNs can end up being much simpler and leave lots of room for improvement once implementation is done as the structure can very easily be scaled up for problems of a higher complexity.

For both NN structures, the neural network is going to need three inputs, which are the x, y and z coordinates of the desired end effector-position, and as many outputs as there are degrees of freedom, i.e. joints, in the robotic arm. These outputs would be the angles each joint needs to move to get the robotic arm to move the end effector to the desired position. When it comes to generating the inputs and the target outputs, there are two possible approaches: with the Forward Kinematics model, randomly generating the desired end effector positions to be used as inputs to the Neural Network, and implementing an Inverse Kinematics model to work out the angles that would get the robotic arm to the desired position, using these as the target outputs; or conversely, generating random angles that would serve as the target outputs, and using only the Forward Kinematics model, working out the positions at which the robotic arm would end up and using these positions as the inputs to the Neural Network.

Once implementation has been completed, it will be thoroughly tested and evaluated to ensure it is working as intended. The Neural Network will be evaluated using the error function that is used in the back-propagation. This error will be minimized and it will need to fall below a certain threshold so the NN can be considered to have solved the problem. Since we will be working in a geometric space, the error function will be something that will

have intrinsic meaning and can easily be described and visualized. I expect that this error function will not be hard to find as it is a simple trigonometric function. For the purpose of this project, the Neural Network will be considered to have solved the Inverse Kinematics problem if the average difference between the target angle and the output angle is less than five degrees.

I would also like to investigate how the error is minimized for Neural Networks of varying DOF, and how many neurons need to be in the hidden layer for the loss function to fall below a certain threshold, for a robotic arm of increasing DOF. I feel that both of these could open up interesting conversations about the complexity of the problem and discussing them can help bring us closer to understanding it.

Chapter 4

Design

Looking at the possible ways this could be implemented, we are left with four possible design choices. Two for each Neural Network structure: one using an Inverse Kinematics model to map desired end effector positions to joint angles and using these as inputs and target outputs to the NN, respectively; and another using the Forward Kinematics model to map joint angles to end effector positions and using these as target outputs and inputs to the NN, respectively.

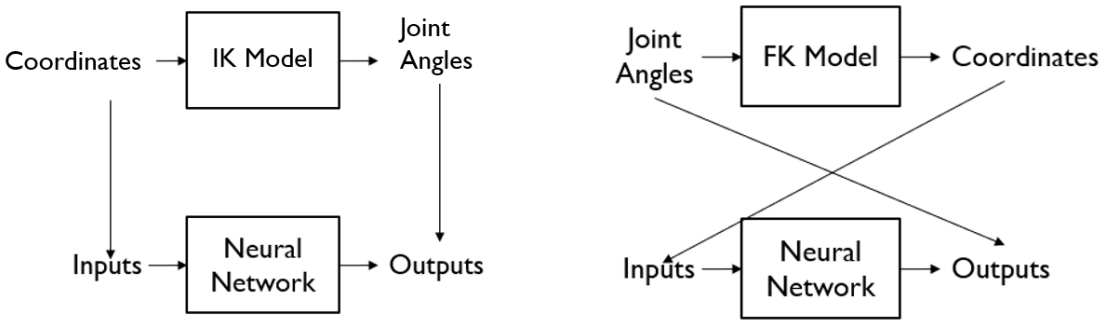

Figure 4.1: Illustration of how each possible design would be implemented

4.1 Deciding the Final Design

With the project's main aim of simplicity in mind, the Neural Network structure chosen to be implemented was a Feed-Forward Back Propagating Neural Network. With the structure decided and the possible design choices narrowed down to the two aforementioned options, which meant the final decision that had to be made was whether or not to build an Inverse Kinematics model. Just like the Forward Kinematics model that was implemented, which maps joint angles to end effector positions, this model would do the opposite, mapping desired end effector positions to the angles needed to get the robotic arm to that position.

The implementation of such a model would give me a better understanding of the intricacies of the problem and would make it easy to obtain optimal results; however, as helpful as this would be, it would also involve a lot of effort and would take away time from the implementation of the actual Neural Network so ultimately, it was decided that the Forward Kinematics model would be used and the inputs and target outputs of the Neural Network would be worked out according to Figure 4.2.

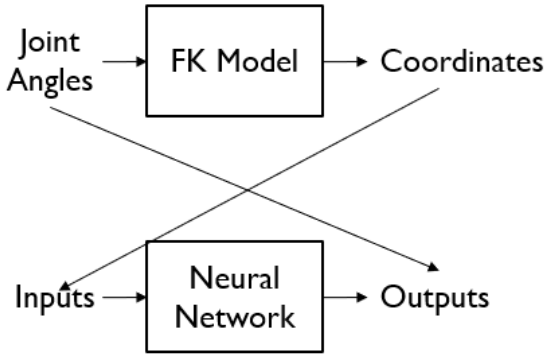

Figure 4.2: Final Design of the implementation

So firstly, the joint angles that go into the FK Model are randomly generated and used to obtain end effector positions. These end effector positions are then passed as inputs to the Neural Network and used to obtain some outputs (that initially are completely random due to the nature of Neural Networks). After this, the error between these and the joint angles that were randomly generated at the start and the outputs of the Neural Network is computed and the weights of the neural network are updated using the gradient descent.

With this implemented, we should theoretically have a Neural Network that can solve the Inverse Kinematics problem which then just needs to be tested and validated to ensure it is working as intended. The design implemented should, in theory, work well for a 1-DOF robotic arm and leave room for it to be scaled up to more complex problems, i.e. multi-DOF robotic arms by adding one hidden layer and increasing the amount of training data. This will not be attempted in this project because of time limitations but could be in a future project.

Chapter 5

Implementation and Testing

The Neural Network proposed in this paper will be designed using python and the library matplotlib to plot the relevant graphs. One of my biggest aims with this project was to be able to better understand how neural networks work and to be able to implement one, myself, from scratch, so the only library used to help with the actual implementation of the Neural Network and Forward Kinematics will be numpy.

5.1 Forward Kinematics Model

For the implementation of the forward Kinematics Model, I only had to implement the function $rotation_matrix(rotation_angle, axis_of_rotation)$, where $rotation_angle$ is the angle by which the joint is supposed to rotate and $axis_of_rotation$ is the joint. This allows me to create a rotation matrix that can be used to obtain the rotated end effector position. This function was implemented according to the following equation[26], for a unit vector $u = (u_\mathrm{x}, u_\mathrm{y}, u_\mathrm{z})$, and an angle of rotation θ:

$$R = (\cos\theta)\, I + (\sin\theta)\, [\mathbf{u}]_\times + (1 - \cos\theta)\, (\mathbf{u} \otimes \mathbf{u})$$

where $[u]_\mathrm{x}$ is the cross product matrix of u, the expression $u \otimes u$ is the outer product, and I is the identity matrix. Since the joint is not always going to be represented by a unit vector, the first thing this function does is normalize the vector. It then calculates all the different members of the equation and finally works out and returns the rotation matrix. This was made into a Python class so it can be used in other algorithms as needed.

Testing for this part of the implementation was fairly straightforward as checking whether the rotation was done properly or not could be done by connecting the initial end effector position to the end effector position, and using the law of cosines to calculate the length of this line. If the implementation is correct, this would mean that $\sqrt{(A_x - B_x)^2 + (A_y - B_y)^2 + (A_z - B_z)^2} = \sqrt{a^2 + b^2 - 2ab\cos(\theta)}$. With A and B being the initial and final end effector positions, respectively and a and b being the lengths of the lines from the axis to points A and B, respectively.

5.2 Neural Network

For the implementation of the Neural Network, I first implemented a simple FFBPNN to solve the XOR problem, taking inspiration from an implementation found in the Medium website[27].

X_1	X_2	$Y = X_1$ XOR X_2
0	0	0
0	1	1
1	0	1
1	1	0

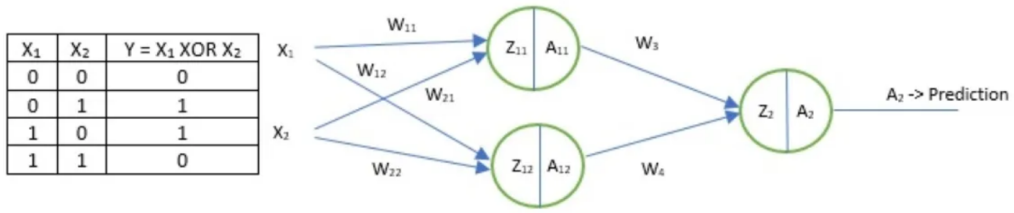

Figure 5.1: Structure of the XOR Neural Network

5.2.1 Forward Propagation

This Neural Network takes two inputs, has a hidden layer with two neurons, and produces one output. The weights that go from the input layer to the hidden layer are described as the weight vector W_1 and the weights that go from the neurons in the hidden layer to the output are described as the weight vector W_2. The values in the neurons in the hidden layer are described as a vector Z_1, these values are then activated and described as a vector A_1. The same goes for the output layer with the vectors Z_2 and A_2.

The Forward Propagation step of the neural network is the calculation of the vectors Z and A. These are calculated according to the following:

$$Z_1 = W_1 \cdot X + b_1 \tag{5.1}$$

$$A_1 = \frac{1}{1 + e^{-Z_1}} \tag{5.2}$$

$$Z_2 = W_2 \cdot A_1 + b_2 \tag{5.3}$$

$$A_2 = \frac{1}{1 + e^{-Z_2}} \tag{5.4}$$

Where Z_1 is the dot product between the inputs and weight vector W_1, ignoring bias, A_1 is the Z_1 vector, activated using the sigmoid function, Z_2 is the dot product between A_1 and the weight vector W_2, ignoring bias, and A_2 is the output prediction given by activating Z_2 using the sigmoid function.

5.2.2 Loss Calculation

The loss value is an estimate of how well the Neural Network is performing. If the loss value is decreasing over time it means the Neural Network is learning and doing its intended job. Deciding on a good loss function is the most essential part when implementing the neural network as this error is going to be propagated back through the network in order to adjust the weights. If the loss function does not represent the error accurately the NN will not be able to learn. Since the XOR problem is a binary problem, the loss function is given by the binary cross-entropy (equation 5.5).

$$L = -\frac{1}{m} * \sum (Y * log(A_2) + (1 - Y) * log(Y - A_2)) \tag{5.5}$$

Where L is the loss value, Y is the target outputs and A_2 is the outputs of the Neural Network.

5.2.3 Back-Propagation and Weight Adjustment

In the Back-Propagation stage, we take the loss value and calculate its gradient with respect to the weight vectors. Using the chain rule of derivative we have:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial A_2} * \frac{\partial A_2}{\partial Z_2} * \frac{\partial Z_2}{\partial W_2} \tag{5.6}$$

$$\frac{\partial L}{\partial A_2} = -(\frac{1}{m})(\frac{Y - A_2}{A_2(1 - A_2)} \tag{5.7}$$

$$\frac{\partial A_2}{\partial Z_2} = A_2(1 - A_2) \tag{5.8}$$

$$\frac{\partial Z_2}{\partial W_2} = A_1 \tag{5.9}$$

$$\frac{\partial L}{\partial W_1} = X * A_1 * (1 - A_1) * W_2 * (A_2 - Y) \tag{5.10}$$

Implementing this in python can be done by having a delta member dz for each weight vector, with dz_2 being the direct derivative of the loss function. This means that dz_2 is the most important component of the back-propagation algorithm as this is the member that directly propagates the gradient of the error through the network. Meaning that if we want to change the objective of the Neural Network and the error function is changed, dz_2 needs to be changed as well if we want the error to be minimized. The weights are adjusted according to the gradient's descent formula.

$$W = W - lr * \frac{\partial L}{\partial W} \tag{5.11}$$

Once this neural network was implemented and validated to be working correctly, it was

made into a Python class so it can be adapted to the Inverse Kinematics problem. The amount of inputs, outputs, and neurons in the hidden layer can be altered at will to adapt the neural network for any problem. Bias was also implemented later as they allow for the creation of non-linear decision boundaries. The biases are updated in the back-propagation by taking the mean of the delta members across the columns ensuring that they are adjusted uniformly based on the overall error propagated through the layer. Furthermore, the activation function on the output layer was changed to a linear function $A_2 = Z_2[28]$. This is because the sigmoid function squishes all the values between zero and one, making it an optimal activation function for a classification problem like the XOR problem, but a poor one in the context of Kinematics where the outputs can be any real value.

5.3 Inverse Kinematics Neural Network

Now with FK and NN classes implemented, these can be combined to implement a Neural Network that solves the Inverse Kinematics problem. It was decided that I would first implement a Neural Network to solve for the Forward Kinematics problem so I could see if my Neural Network class was working as intended, as the FK problem is not as complex as the IK problem. To do this, both classes are imported, the initial angles, weight and bias vectors are all generated randomly, the inputs are then passed through the FK model to obtain coordinates of end-effector positions which are used as the target outputs of the Neural Network. The error function used was the mean squared error.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - A2_i)^2 \tag{5.12}$$

Using MSE as the error function, we have $dz_2 = (y - a2)$. Y being the target outputs and A2 being the outputs of the Neural Network. This was quickly validated to be correctly implemented as the error was correctly minimized over time. For a 1-DOF robotic arm, setting the learning rate to 0.1 and the number of neurons in the hidden layer to 16, the network minimizes the loss to $8.21 \times 10^{-}3$ after ten thousand iterations.

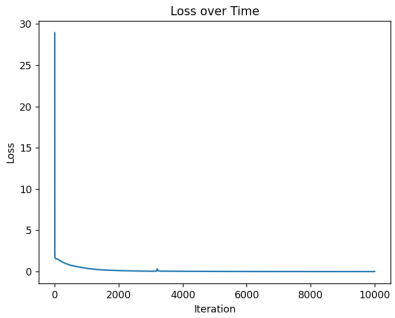

Figure 5.2: Loss over time graph for the NN configuration mentioned above.

Further testing was done by taking ten random samples from the training data, putting them through the NN after it has been trained, and comparing the outputs with the target outputs. This can be seen in Figure 5.3.

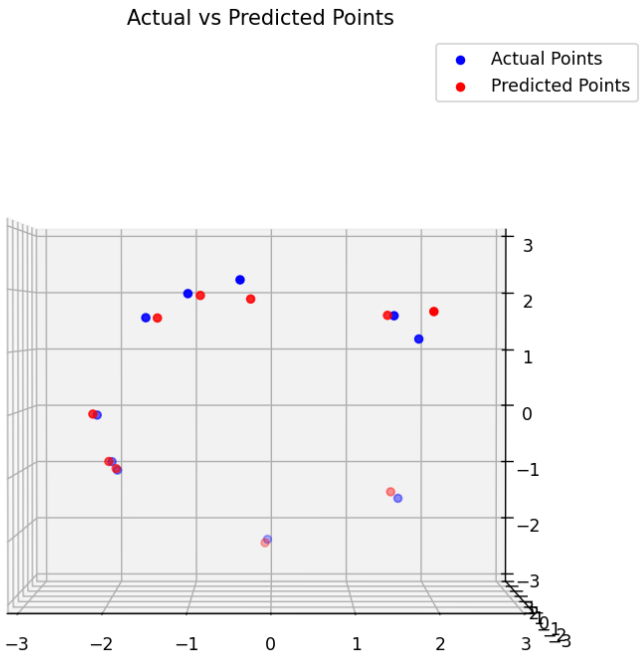

Figure 5.3: Plot of the predicted outputs against the target outputs.

Having implemented a Neural Network that can solve the FK problem, I can now easily adapt this to solve the IK problem, as their structure is going to be more or less the same. In theory, this can be achieved by taking the randomly generated joint angles that serve as inputs to the FK Model, and using them as the target outputs of the NN, while the outputs of the FK Model are passed through as inputs to the NN. Doing this should theoretically be enough, however, I wanted the error function to have meaning geometrically. While MSE would be an adequate error function to use, I feel that having an error function that has geometric meaning can help understand how the error is being minimized and give a clear understanding of the magnitude of the error.

With this in mind, a loss function that accurately represents the error between a target angle and a predicted angle would be maximized when the angles are opposite (i.e. the difference between them is π) and minimized when the angles are equal (i.e. the difference between them is 0). With the help of my supervisor, the following function was settled upon:

$$\sin\left(\frac{x}{2}\right)^2 \tag{5.13}$$

Where x is the difference between the target joint angle and the predicted angle. This is a periodic function that is maximum and equal to one when $x = \pi$ and minimum and equal

to zero when $x = 0$. This function is represented by the following graph.

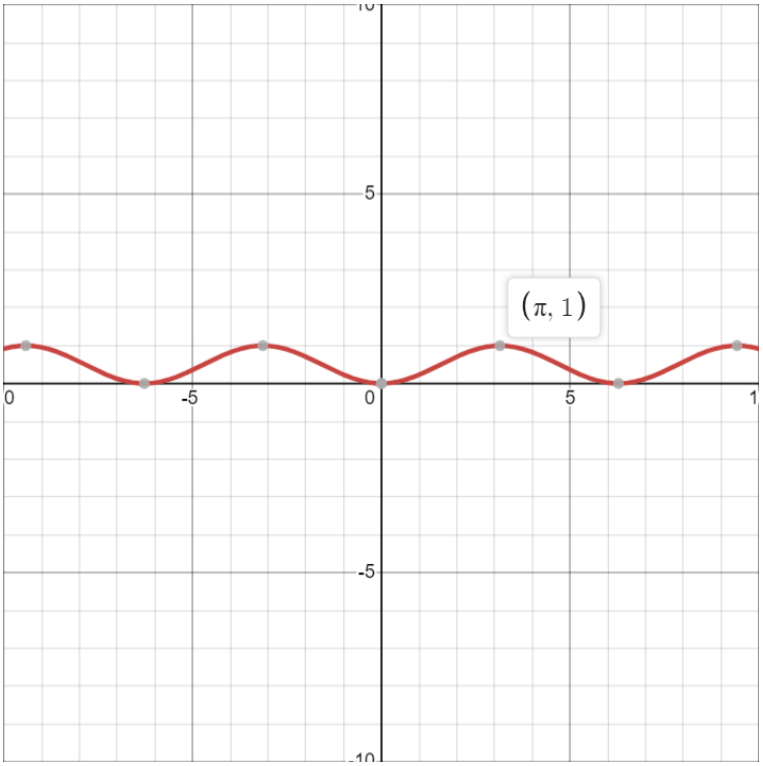

Figure 5.4: Graph of the loss function used for the IKNN

Testing the final IKNN implementation followed the same strategy taken when testing the FKNN. The Neural Network was considered to be correctly implemented as the error was minimized over time. For a 1-DOF robotic arm, setting the learning rate to 0.25 and the number of neurons in the hidden layer to 16, the network minimized the loss to $2.05 \times 10^{-}2$ after ten thousand iterations.

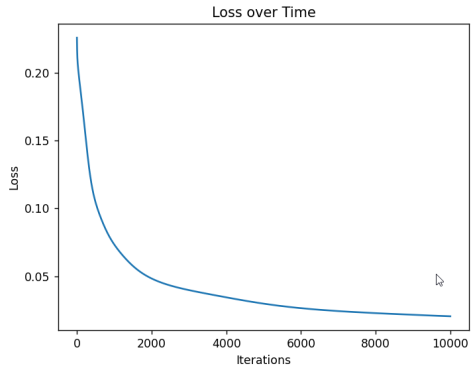

Figure 5.5: Loss over time graph for the NN configuration mentioned above.

As was done with the FKNN, further testing was done by taking random samples from the training data, putting them through the NN after it had been trained, putting the predicted joint angles through my FK model to obtain an end effector position, and comparing these with the inputs of the NN (target end effector position). This can be seen in Figure 5.5.

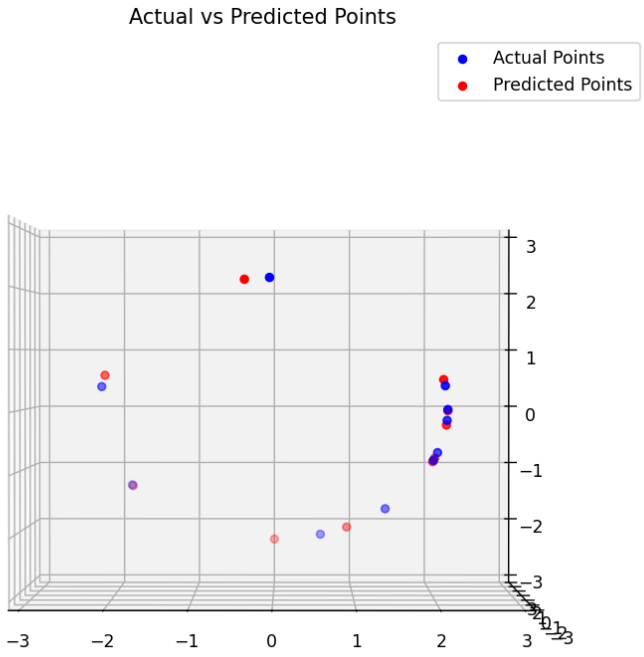

Figure 5.6: Plot of the position obtained using the predicted outputs against the desired end-effector position.

Chapter 6

Results and Discussion

With the final IKNN implementation finalized and validated, came time to put this NN to the test and see the extent of its capabilities. Doing this would help me confirm whether or not a design this simple would be feasible for robotic arms of varying DOF, as well as give insight to the true complexity of the Inverse Kinematics problem. For the purpose of this project, the Neural Network was to be considered a feasible implementation if it could bring the error below $1.90 \times 10^{-}3$.

6.1 How far can it be pushed?

I first wanted to see how much the NN could minimize the loss after ten thousand iterations for a fixed learning rate of 0.25 and 12 neurons in the hidden layer. As we increase the DOF in a robotic arm, the IK solutions become harder and harder to compute as each DOF added, adds one level of complexity to the problem. With this in mind, it is expected that the final loss after training increases with each DOF added and it will be interesting to see the rate at which this increases.

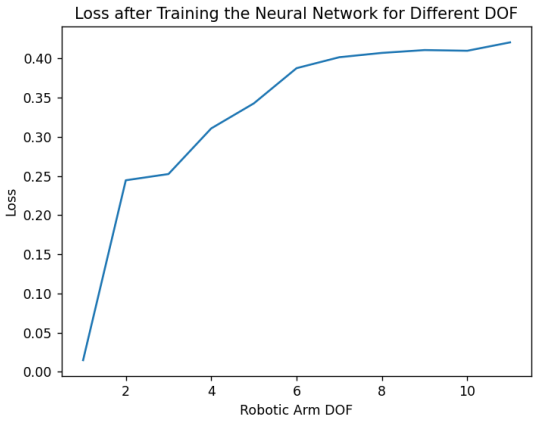

Figure 6.1: Loss after training for robotic arms of increasing DOF

The graph displays a curve that resembles a logarithmic curve which can be described as a curve that increases at an ever-decreasing rate as x gets larger. Although the final loss after training, increasing with each DOF added was expected, this resembling a logarithmic curve was not. This can be explained by both the nature of the problem and how Neural Networks work. Since we are trying to obtain joint angles as outputs, these will increase as the robotic arm DOF increases, in turn, making the error go down slower. The nature of a NN makes it harder to update weight vectors that produce n outputs as opposed to only one, as there are n many more outputs to be accounted for.

After this, I wanted to know how many neurons the hidden layer needed to have so that, for a fixed learning rate, the Neural Network could bring the loss below a certain threshold after ten thousand iterations, for robotic arms with varying DOF. Ideally, the error threshold should be the value set at the start of this chapter; however, when collecting these results, the network could not bring the error below that threshold even for a 2-DOF robotic arm. Thus the threshold ended up being settled upon after some investigation so that the graph produced could accurately describe the complexity of the problem. The learning rate was fixed at 0.25 and the threshold was set at 0.42.

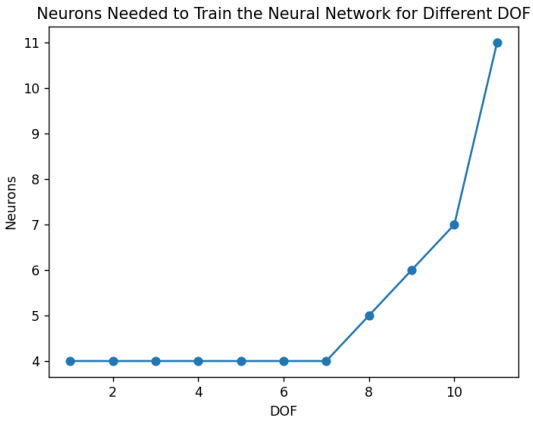

Figure 6.2: Neurons needed to train the Neural Network for Varying DOF

We can see that this graph is the exact opposite of the other one, resembling an exponential curve, This is exactly what was expected and, again, can be explained by the intrinsic nature of Neural Networks. As the robotic arm DOF increases, so do the number of outputs and, therefore, the number of neurons needed to solve the problem.

6.2 Is it feasible and how could it have been made better?

According to the personal goal outlined in the requirements, the NN structure I implemented is feasible and can be considered a good solution to the Inverse Kinematics problem as it brought the loss below the intended threshold in about one hundred and fifty thousand iterations, for a 1-DOF robotic arm. However, the solution proposed cannot be used in robotic

arms with a DOF superior to one as it cannot bring the error below the intended threshold. Overall I would consider this project a success as the initial goal of implementing an Inverse Kinematics Neural Network by inverting the inputs and outputs of a Forward Kinematics Neural Network, was ultimately achieved. Furthermore, this project gave me a lot of insight into both the fields of Kinematics and Machine Learning, allowing me to understand the complexity of the Inverse Kinematics problem and how a simple Neural Network can be implemented to solve this.

The initial aim of keeping the Neural Network structure as simple as possible was good for the purpose of this project but it created limitations that could not be overcome deeper into the implementation. An example of this is the amount of hidden layers and training data used. If both had been increased, the complexity and scope of the NN would increase, possibly making it easier to minimize the error below an acceptable threshold. It would be interesting to see how this affected the network and its performance for robotic arms with a DOF superior to one. Since this Neural Network works well for a 1-DOF robotic arm, my theory is that for each DOF, the number of hidden layers needs to increase by one and the amount of samples needs to be increased to n^m, where n is the number of samples for a 1-DOF robotic arm and m is the DOF of the new robotic arm.

Chapter 7

Conclusions

In this project, the implementation of a Neural Network for solving the Inverse Kinematics problem in robotic arms was explored. The project began with a thorough analysis of the requirements and considerations necessary for developing an effective solution. Understanding the fundamentals of Forward Kinematics being a crucial prerequisite, this lead to the implementation of a FK model to establish a foundation for subsequent work.

Following the FK model's implementation, the focus shifted to designing and implementing the NN structure for solving the IK problem. Various NN architectures were considered, with a Feed-Forward Back Propagating Neural Network (FFBPNN) ultimately being chosen for its simplicity and potential effectiveness. This decision took into account complexity, training data requirements, and computational efficiency.

The implementation phase involved developing the NN architecture and integrating it with the FK model. A rigorous testing and validation process was conducted to ensure the correctness of the solution and decide whether the implementation was successful. Both the FKNN and IKNN were thoroughly evaluated using the appropriate error functions.

Results from the experimentation phase provided valuable insights into the IKNN's capabilities and limitations. It was found that while the NN structure proved successful for 1-degree-of-freedom (DOF) robotic arms, its performance diminished as the DOF increased. The complexity of the IK problem increased exponentially with higher DOF, making it necessary to adjust the NN structure and length of training data, for scalability to be possible.

By reflecting on the project's outcomes the significance of gaining insights into both kinematics and machine learning domains is highlighted. Despite encountering challenges in achieving low error thresholds for higher DOF configurations, the project can overall be labeled a success when it comes to implementing a working IKNN from scratch and gaining valuable insight into the field of Robotics and Machine Learning. I also feel that the ideas discussed in this paper plenty of room for future improvement and can lead to advancements in the area.

In conclusion, this study contributes to the understanding of Inverse Kinematics problem-solving methodologies using Neural Networks and highlights the importance of balancing simplicity with effectiveness in algorithm design. While the current solution

demonstrates feasibility for certain robotic arm configurations, further research and refinement are warranted to address challenges associated with higher DOF systems. By embracing a collaborative and iterative approach, future endeavors in this area hold promise for unlocking new possibilities in robotics and automation.

Bibliography

[1] G. Wu and J. Wang, "A recurrent neural network for manipulator inverse kinematics computation," *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, 1994.

[2] S. Tejomurtula and S. Kak, "Inverse kinematics in robotics using neural networks," *Information Sciences*, vol. 116, p. 147–164, Mar 1998.

[3] A. Eliasy and J. Przychodzen, "The role of ai in capital structure to enhance corporate funding strategies," *Array*, vol. 6, p. 100017, 2020.

[4] G. Bhardwaj, N. Sukavanam, R. Panwar, and R. Balasubramanian, "An unsupervised neural network approach for inverse kinematics solution of manipulator following kalman filter based trajectory," *2019 IEEE Conference on Information and Communication Technology*, Dec 2019.

[5] S. Kucuk and Z. Bingul, "Robot kinematics: Forward and inverse kinematics," *Industrial Robotics: Theory, Modelling and Control*, Dec 2006.

[6] R. Paul and B. Shimano, "Kinematic control equations for simple manipulators," *1978 IEEE Conference on Decision and Control including the 17th Symposium on Adaptive Processes*, Jan 1978.

[7] J. J. Craig, *Introduction to robotics: Mechanics and control*. Pearson, 2008.

[8] L.-C. Wang and C. Chen, "A combined optimization method for solving the inverse kinematics problems of mechanical manipulators," *IEEE Transactions on Robotics and Automation*, vol. 7, p. 489–499, Aug 1991.

[9] A. Aristidou and J. Lasenby, "Fabrik: A fast, iterative solver for the inverse kinematics problem," *Graphical Models*, vol. 73, p. 243–260, Sep 2011.

[10] E. Oyama and S. Tachi, "Inverse kinematics learning by modular architecture neural networks," *IJCNN'99. International Joint Conference on Neural Networks. Proceedings (Cat. No.99CH36339)*, Jul 1999.

[11] E. Oyama, T. Maeda, J. Gan, E. Rosales, K. MacDorman, S. Tachi, and A. Agah, "Inverse kinematics learning for robotic arms with fewer degrees of freedom by modular

neural network systems," *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Aug 2005.

[12] E. Oyama, T. Maeda, J. Gan, E. Rosales, K. MacDorman, S. Tachi, and A. Agah, "Inverse kinematics learning for robotic arms with fewer degrees of freedom by modular neural network systems," *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005.

[13] T. Ogawa, H. Matsuura, and H. Kanada, "A solution of inverse kinematics of robot arm using network inversion," *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*.

[14] Y. Zhou, W. Tang, and J. Zhang, "Algorithm for multi-joint redundant robot inverse kinematics based on the bayesian - bp neural network," *2008 International Conference on Intelligent Computation Technology and Automation (ICICTA)*, Oct 2008.

[15] T. Yuan and Y. Feng, "A new algorithm for solving inverse kinematics of robot based on bp and rbf neural network," *2014 Fourth International Conference on Instrumentation and Measurement, Computer, Communication and Control*, Sep 2014.

[16] M. Puheim and L. Madarasz, "Normalization of inputs and outputs of neural network based robotic arm controller in role of inverse kinematic model," *2014 IEEE 12th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, Jan 2014.

[17] P. Srisuk, A. Sento, and Y. Kitjaidure, "Inverse kinematics solution using neural networks from forward kinematics equations," *2017 9th International Conference on Knowledge and Smart Technology (KST)*, Feb 2017.

[18] C. Liu, X. Wang, H. Jiang, X. Wang, and H. Guo, "Inverse kinematics solution of manipulator based on ipso-bpnn," *2022 5th International Conference on Pattern Recognition and Artificial Intelligence (PRAI)*, Aug 2022.

[19] S. Wang, G. He, and K. Xu, "A novel solution to the inverse kinematics problem in robotics using conditional invertible neural networks," *2023 IEEE 16th International Conference on Electronic Measurement amp;amp; Instruments (ICEMI)*, Aug 2023.

[20] A.-N. Sharkawy and S. S. Khairullah, "Forward and inverse kinematics solution of a 3-dof articulated robotic manipulator using artificial neural network," *International Journal of Robotics and Control Systems*, vol. 3, p. 330–353, May 2023.

[21] R. Hao, "Inverse kinematic analysis and simulation of the scara robot based on recurrent neural networks," *MEMAT 2022; 2nd International Conference on Mechanical Engineering, Intelligent Manufacturing and Automation Technology*, 2022.

[22] A. Shaar and J. A. Ghaeb, "Intelligent solution for inverse kinematic of industrial robotic manipulator based on rnn," *2023 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, May 2023.

[23] V. Balakrishnan, "How is a vector rotated?," *Resonance*, vol. 4, p. 61–68, Oct 1999.

[24] A. Morawiec, *Orientations and rotations: Computations in crystallographic textures*. Springer, 2004.

[25] A. Palazzolo, "Formalism for the rotation matrix of rotations about an arbitrary axis," *American Journal of Physics*, vol. 44, p. 63–67, Jan 1976.

[26] J. Mathews, "Coordinate-free rotation formalism," *American Journal of Physics*, vol. 44, p. 1210–1210, Dec 1976.

[27] S. A. Bhatti, "Coding a neural network for xor logic classifier from scratch," Mar 2020.

[28] A.-V. Duka, "Neural network based inverse kinematics solution for trajectory tracking of a robotic arm," *Procedia Technology*, vol. 12, p. 20–27, 2014.