Neural Network as an Alternative to the Jacobian for Iterative Solution to Inverse Kinematics

Fabrício Julian Carini Montenegro, Ricardo Bedin Grando, Giovani Rubert Librelotto, Rodrigo da Silva Guerra
Universidade Federal de Santa Maria
spl.splinter@gmail.com, ricardo.grando@ecomp.ufsm.br, librelotto@inf.ufsm.br, rodrigo.guerra@ieee.org

***Abstract*—The inverse kinematics problem is generally very complex and many traditional solutions are targeted only to robots of certain specific topologies. The iterative method based on the (pseudo)inverse of the Jacobian matrix is a well-known, proven, and reliable general approach that can be applied to a wide variety of manipulators. However, it relies on linearizations that are only valid within a very tight neighborhood around the current pose of the manipulator. This requires the robot to move at very short steps, intensively recalculating its trajectory along the way, making this approach inefficient for certain applications. Neural networks, for their known capacity of modelling highly non-linear systems, appear as an interesting alternative. In this paper we demonstrate that neural networks can indeed be successfully trained to map task space displacements into joint angle increments, outperforming the method based on the inverse of the Jacobian when dealing with larger displacement increments. We validate our study showing comparative results for hypothetical 2-DOF and 3-DOF planar manipulators.**

***Index Terms*—Robotics, Neural Network, Jacobian, Differential Kinematics, Inverse Kinematics**

I. Introduction

To perform real-world tasks, usually, the joints of a robot must move in such a way that its end-effector reaches a certain goal, or follow a certain trajectory in space. These mappings from joint space into task space, and vice-versa, are studied, respectively, in robotic forward and inverse kinematics [1], [2]. More specifically, the inverse kinematics problem consists in determining the joint values in such a way as to place its end-effector at a desired position and orientation. The solution to this problem is of fundamental importance in robotics.

While the forward kinematics problem can be solved with simple rules, typically multiplying a chain of homogeneous transformation matrices, the inverse kinematics problem is much more complex for a number of reasons. Inverse kinematics usually involves a set of nonlinear, coupled, algebraic equations to which there is no general algorithmic solution [3], and often a closed-form solution can not be found [4]. For some configurations of robotic arms, more than one set of joint values may result in the same position and orientation of the end-effector. In the case of redundant manipulators, whole intervals (infinite set) of solutions may be found to map to a same position and orientation in task space [4]. Also, there might be no admissible solutions, in view of the manipulator kinematic structure [4], particularly, if the desired position and orientation for the end-effector is outside the robot's workspace.

One of the most popular approaches is to iteratively compute the (pseudo)inverse of the robot Jacobian at every pose increment along its trajectory. The Jacobian is a matrix-valued function, calculated for each given robot pose, and it can be thought of as the vector version of the ordinary derivative. The components of the Jacobian matrix are the partial derivatives that, for the given robot pose, relate the rate of changes in joint values to the corresponding rate of changes in the end-effector's position and orientation [1]. With this derivative in hand, we can use its (pseudo)inverse to compute small changes on the joint values that result in the given desired small incremental adjustments in the end-effector's position and orientation, so as to move it, ever so slightly, towards the desired goal, step by step.

However, due to the non-linear nature of the inverse kinematics problem, the Jacobian approximates the behavior of the system only for a narrow neighborhood surrounding the current configuration of joint values (linearization around the current robot's pose). As a consequence, the process of using the inverted Jacobian requires incrementally approaching the goal through many repeated iterations at very small steps.

Another complication is that calculating the inverse (or pseudoinverse) of a matrix can be a computationally intensive task. Moreover, the successive increments may lead the robot's pose to a path that passes near singularities causing severe numeric instability.

Meanwhile, in the last few years, we have witnessed a rebirth of neural networks, along with the emergence of deep learning approaches [5]. Today, neural networks are employed in a vast range of applications in many different fields, from gesture recognition [6] to speech recognition [7], from visual computing [8] to natural language processing [9]. Several software frameworks are available for training and deploying neural networks, and a myriad of specially customized hardware devices is widely available, making these very powerful and versatile tools for a broad variety of computational applications.

The rest of this paper is organized as follows: Section II talks about related works in the area; In Section III, we show our methodology, explaining how the neural network was built, how the data for its training was generated, and how it can be used; Then, in Section IV we present a study applying this approach on simulated 2-DOF and 3-DOF planar

This work is supported by CAPES.

manipulators; Finally, in Section V we discuss the results and future work.

II. Related Work

In this paper we study the use of neural networks as an alternative to the Jacobian approach, as an iterative solution to inverse kinematics.

Most works that use neural networks in the area of robot kinematics focus on trying to solve the general inverse kinematics problem, in some cases with relatively good results ([10], [11], [12]). Some works even use complementary techniques, like genetic algorithms[13]. But, from related works, we find that solving the inverse kinematics problem on a neural network is not a trivial task.

The inverse kinematics problem has too big of a non-linear space to be modeled into a simple neural network, even when testing with a simple 3D, 3-DOF robot [14]. As mentioned in [15] and [16], modeling the inverse kinematics of a simple robot with a neural network is a complicated task even for big and complex neural networks.

For this reason, using the Jacobian approach instead of the general inverse kinematics method, seems to be a good alternative. Some works model the Jacobian approach into neural networks, but they generally focus on a single aspect of the problem, as seen in [17], which focuses on the velocity of the movement without trying to outperform the classic approach.

Our work has emphasis on enhancing the performance of the inverse Jacobian method for larger step increments, improving on the limitations previously discussed, while still generating good results when near singularities.

III. Methodology

In our approach we use a neural network as a replacement for the inverse of the Jacobian. Our goal is to have neural network model that works well in situations in which the inverse of the Jacobian doesn't. We assume we are given the same inputs and outputs.

In this paper we used two different simulated planar arms to test our methodology: one with 2 rotational joints and the other with 3 rotational joints. We focused only on the position coordinates of the end-effector, without taking into account the orientation. Planar 2-DOF and 3-DOF arms have well known closed solutions for inverse kinematics. However, these arms were chosen for clarity, as they serve as a good benchmark environment for testing and demonstrating the approach developed in this paper. The technique showed in this paper can be extended to be applied in simulated and real-world 3D manipulators, with larger number of joints, controlling for both position and orientation of the end-effector. The environments used for the tests are detailed in Subsection III-E.

In the next subsections, we present the inputs and outputs of the network, show how the training data were generated, explain the neural network's architecture, and describe the tests we did in order to validate the methodology.

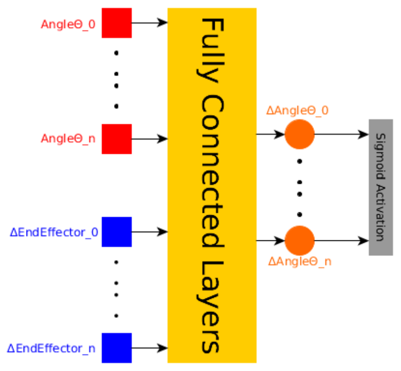


Fig. 1. Network Structure

A. *Neural Network's Inputs and Outputs*

In general, inverse kinematics algorithms receive an absolute goal position for the end-effector and compute the matching absolute joint values to bring the robot to that position. In the case of differential kinematics, we inform the algorithm the current pose and the desired incremental step change we want to apply to the end-effector. In the output we expect to see the corresponding incremental change in joint values that we ought to apply to move the robot the desired amount in the desired direction.

The inputs of our neural network are:

1) The current angular position θ of the rotational joints;
2) The desired change Δp in the end-effector's position.

For a generic robot arm, working in a 3D space, one would have as many components in θ as the number of joints, and up to six values in the Δp vector (for the desired step increments in position and orientation). For the 2-DOF and 3-DOF planar arms studied in this work, θ has either two or three values, respectively. Furthermore, since we only consider position, and the planar arms only move in 2D space, the Δp vector has only two values (Δx and Δy).

The output of our neural network is simply the vector $\Delta\theta$ of the corresponding incremental changes in the joint values that should result in the desired Δp. The same consideration applied to the number of inputs should be observed here, where a different number of joints means a different number of outputs.

Table I summarizes the inputs and outputs used for the cases studied in this paper. The neural network's structure is shown in Figure 1 and it's architecture is described in Subsection III-D.

TABLE I
Inputs and Outputs for the 2-joint and 3-joint planar arms

In/Out	2-joint Planar Arm	3-joint Planar Arm
Inputs	θ_1, θ_2, Δx, Δy	θ_1, θ_2, θ_3, Δx, Δy
Outputs	$\Delta\theta_1$, $\Delta\theta_2$	$\Delta\theta_1$, $\Delta\theta_2$, $\Delta\theta_3$

B. Generating Data

To train the neural network we generated large datasets with about 2 millions samples each. To generate the individual samples, we used the forward kinematics, proceeding as described below.

The first step was generating a random valid pose. For simplicity purposes, this means generating any pose, without necessarily checking for self-collisions. It should be noted, however, that collision detection must be implemented for an application in real robot arms. To generate these random poses, we set the position of the joints to random angles, which, in turn, set the end-effector at a random position in space. The random angles generated for each joint were sampled from an uniform distribution ranging from $-\pi$ to π, as to cover the entire workspace of the robot arms. The random vector θ of initial joint positions generated in this first step was saved to be later normalized and fed to the neural network for training. These are our first set of inputs. With this random pose in hand, we applied the forward kinematics to find the corresponding position p of the end-effector. This position was also saved for later use.

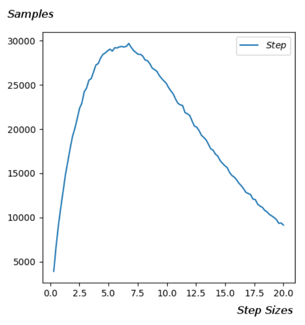
The second step was, starting from each of those random valid poses, generating a small random incremental change $\Delta\theta$ in the angles of the joints. The random angular displacements were generated using a normal distribution with a mean of 0 and standard deviation of 0.33. Values larger than 1 and lower than -1 were discarded. The resulting values were then scaled to the range between the negative and positive maximum value of $\Delta\theta$, e.g., when the maximum value of $\Delta\theta$ is $5°$, the range goes from $-5°$ to $5°$. These ranges are shown in Table II.

TABLE II
VALUES USED FOR GENERATING SAMPLES

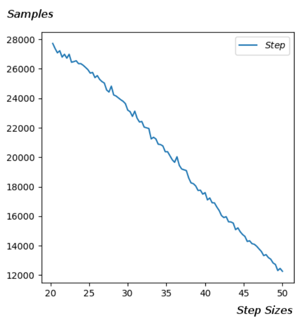
Maximum Value of $\Delta\theta$	Minimum Step Size	Maximum Step Size
$5°$	$0.1mm$	$20.0mm$
$15°$	$20.0mm$	$50.0mm$
$20°$	$50.0mm$	$100.0mm$
$20°$	$0.1mm$	$100.0mm$

The values of $\Delta\theta$ affect the amplitude of Δp, which is the resulting change in the end-effector's position in task space. Ideally we would like to choose in advance a range of angle displacement values $\Delta\theta$ that would result in the known desired sizes of Δp that we want to use for training. However, that's exactly the very inverse kinematics problem we want to solve. This means that we need to first provide a random angular increment $\Delta\theta$ and then validate the resulting vector Δp obtained through forward kinematics, by checking its size.

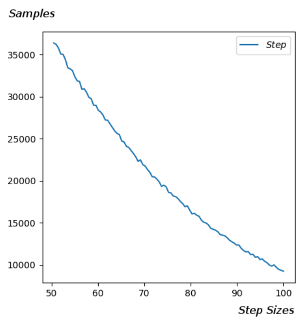
The distributions of resulting step sizes for the 2-DOF arm are shown[1] in Figure 2. The histogram's shape resembles that of a skewed normal distribution (see Figures 2a and 2d). This is due to the influence of the non-linearities in the kinematics of the arm. Some of the ranges that we used ended up truncating the histogram similar to a skewed normal shape, showing only higher values (tail of the curve), yielding a descending curve. That was the case for Figures 2b and 2c.



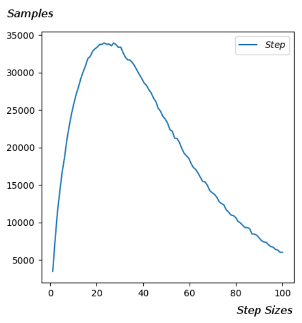
(a) Steps from $0.1mm$ to $20mm$



(b) Steps from $20mm$ to $50mm$



(c) Steps from $50mm$ to $100mm$



(d) Steps from $0.1mm$ to $100mm$

Fig. 2. Histogram showing the relation between number of samples and step sizes

This also resulted in a lower number of samples towards the end (and sometimes the beginning) of the range, if compared to values in the middle of the range. As we discuss later, in Section IV, since the problem approximates a linear function in the vicinity of the original pose, the traditional inverse Jacobian method is usually[2] very precise for small step increments. Therefore it's very hard to make a neural network model that performs better than the Jacobian for such small step increments. For this reason, we focused on training a neural network capable of executing the differential kinematics method at larger step increments, where the Jacobian fails. To get samples for larger step increments we used four different ranges for $\Delta\theta$, with different respective ranges of step sizes, as shown in Table II. These values are explained in more detail in Subsection III-E.

The third step is using forward kinematics to find the new position p' of the end-effector after applying the incremental joint angle changes to the original robot pose.

In the fourth step, we calculate the vector Δp representing the change in the end-effector's position, i.e., $p' - p$. This information is stored using Δx and Δy. In this step we check the size of the vector Δp, discarding samples that result in movements outside the range we previously established as valid. Finally we collect our second set of inputs – the vector Δp, with Δx and Δy – and our output – the generated $\Delta\theta$.

[1]For consiseness, we omit showing the distributions for the 3-DOF arm, but they are of very similar nature

[2]If not near a singularity

Putting it more succinctly, the steps are:

1) Set the arm to a random pose (initial θ) by generating random angles for each joint and store the resulting end-effector position (p)
2) Generate a small random change in the joint angles ($\Delta\theta$)
3) Calculate the new position of the end-effector (p')
4) Calculate the vector representing the change in the end-effector position (Δx, Δy). In this step we also discard samples with vectors outside the desired range.

After trained, the neural network should be fed the other way around. We give it the initial joint angles θ and our desired change in the end-effector position Δp. Then, we ask the network to give us the necessary change $\Delta\theta$ that results in the desired Δp for the initial θ provided.

C. *Normalization of Data*

To feed the data to the neural network we need to normalize the values of each sample to the range from 0.0 to 1.0. The values of the initial angles θ of the joints are normalized between the minimum and maximum valid angles of the joints, i.e., $-\pi$ and π. The values of the second set of inputs, the change Δp in the end-effector position, are normalized in the range of the valid movements decided in the step four of the respective data generation process. The values decided as a valid range are described in Table II and explained in further detail in Subsection III-E.

The vector Δp can point to any direction, so the values of Δx and Δy can vary considerably. We can still predict the maximum and minimum values of Δx and Δy. The maximum variation in the x axis will happen in samples where y doesn't change at all. That is, having a vector Δp of maximum size $100mm$ means Δx can range up to $100mm$, when Δy is 0. This happens in both directions, so both Δx and Δy will range from $-100mm$ to $100mm$. So we simply normalized Δx and Δy between the maximum and minimum values contained in the samples.

The output values of the neural network output, that is, the incremental change $\Delta\theta$ in angles of the joints that will cause the desired movement of the end-effector, are normalized between the angle ranges originally used to generate the random movements in step two. These values were already shown in Table II, although we do use radians instead of degrees for the normalization in the actual code.

D. *Neural Network Architecture*

The neural network we used is a simple multi-layered feed-forward fully connected network with 260,803 trainable parameters. The network has 3 hidden layers with 640, 320 and 160 neurons, respectively. We apply dropout in the outputs of the second and third hidden layers to avoid over-fitting. The hidden layers use rectified linear unit (ReLU) as activation function, while the output layer uses a sigmoid function.

We trained each model with MSE as loss metric, learning rate of 0.002, and used the Adam optimizer. We used 2 million different samples for each range, batch size of 2000, and trained for 100 epochs.

Four models of the neural network were trained for each of the two kinds of arm, one for each range of movements, as explained in Subsection III-E. In total, 8 models of neural networks were trained.

E. *Testing Environment*

As a proof of concept for testing our methodology, we created planar robotic arms simulated in a 2D environment designed using PyGame. In our simulation, we used two arms with 600 units of length, representing an arm of $60cm$ of length. This size is close to those of many robots, including robots we have in our robotics laboratory. We tested our method in two different simulated planar arms, one with two joints and two segments of $30cm$, and another with three joints and three segments of $20cm$.

The downside of this size is that the 2 joints give the robot arm a somewhat poor resolution, since moving the end-effector in a small amount requires using extremely small angles to move the relatively long ($30cm$) arm segments.

Using the Jacobian to move the arms was important to compare its performance with that of the neural network. The two-link planar arm has a square Jacobian matrix that is very easy to calculate. For the 3-joint arm, we used the pseudoinverse of the Jacobian matrix.

After training the network we designed a task to compare its performance against that of the inverse Jacobian method. The task was to control the arms to follow a circular trajectory of radius of $100mm$ at different step increments. The trajectory for the 2-joint arm is centered at coordinates $(200, 300)$, while the center of the trajectory for the 3-joint arm is at coordinates $(300, 200)$. This difference in positioning of the desired trajectory is so that we could easily put the robot arm in the beginning of the trajectory to start our tests. Figure 3 shows the initial poses for the 2-joint and 3-joint arms and their positions relative to the respective trajectories. Both arms should follow the respective trajectories depicted in blue in a counter-clockwise motion.

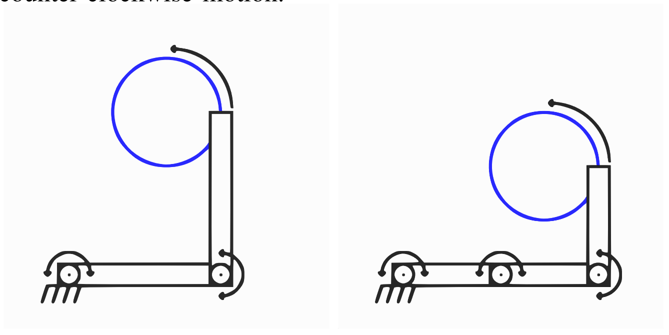


Fig. 3. Initial poses of the arms. Black lines represent the segments of the arm. The blue circle represents the trajectory.

We repeated the task in rounds of different step sizes. At each round, we ran the task for a total of 1000 uniformly spaced points in the trajectory, setting one point after another as the goal end-effector position, changing the step size incrementally in between rounds.

In the first round, we tested the minimum step size. After the arms go through all the 1000 steps, the round is finished. Then we calculate the mean squared error (MSE) for the both control methods, and restart the task for the next round, with trajectory points spaced at incrementally larger distances, and repeating the process.

For our first set of tests, we used trajectories with step sizes ranging from $0.01mm$ to $20mm$, and changing them in increments of $0.01mm$ at a time. From these initial tests we confirmed, as discussed in Section IV, that the Jacobian control easily outperforms the neural network in such small intervals. For this reason, in later tests, we used trajectories with larger step size ranges. In total, we trained and compared the precision of our neural network control with that of the inverse Jacobian method for trajectories with the following ranges: (1) from $0.01mm$ to $20mm$, (2) from $20mm$ to $50mm$, (3) from $50mm$ to $100mm$, and an overarching trajectory with steps from (4) $0.01mm$ to $100mm$. For each range we used a neural network with weights trained specifically for that interval.

IV. Results

Figure 4 shows the training and validation loss of the networks trained for the 3-joint arm in the 4 different ranges.

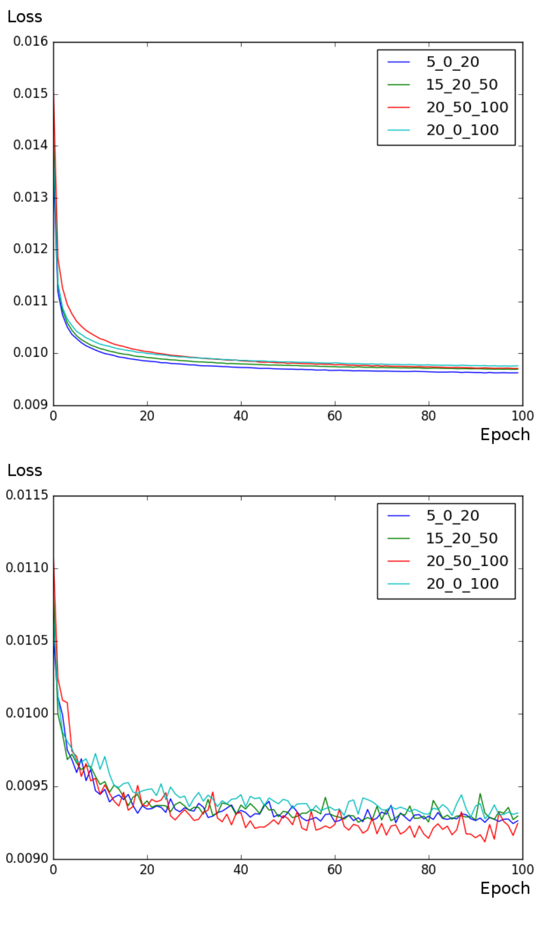


Fig. 4. Training (top) and validation (bottom) loss of the neural networks

In general, what our tests showed is that the neural network is much more stable than the inverse of the Jacobian for the given interval to which it was trained. We can see this behavior very clearly in Figure 5, which shows the comparison between the precision of both arms in the tested trajectory when controlled by the (pseudo)inverse of the Jacobian method and by our neural network in the interval between $50mm$ and $100mm$.

As expected, the neural network indeed performed worse than the inverse of the Jacobian for small step sizes, as shown in the beginning of the range in Figure 5. On the other hand,the neural network performed much better for larger step sizes. Even when the neural network has its worst performance, toward the end of the range, it still has MSE much lower than the inverse of the Jacobian. As the step increments increase, the MSE for the Jacobian grows exponentially, while the MSE of the neural network remains almost constant, increasing only slightly faster towards the end of the trained interval.

As a final test, we compared the behavior of both methods near a singularity. We set the initial position of the 2-DOF arm outstretched to the right and created a trajectory that goes horizontally to left, up to the origin of the system and then goes upward, creating an “L” shape. The origin is also the location of the first joint, and since both links have equal lengths, this configures a singularity of the system. As can be seen in Figure 6, the neural network is much more stable than the inverse Jacobian method when near singularities. The trajectory described by the end-effector when controlled by the Jacobian suffers acute spikes when near the origin.

V. Conclusion

In this paper, we showed a method for training a neural network to compute the differential inverse kinematics of robot manipulators. To validate our results we compared the precision of the the neural network to that of the inverse Jacobian method. We used a couple of planar arms with two and three joints, and set them to move through a circular trajectory using both methods. For each arm, we set four ranges of interest and trained four different neural networks with unique data. In total, eight neural networks were trained.

Based on the observed results we propose a neural network approach based on the concept presented in this work could present some advantages if applied to a real robot in a hybrid approach. In a hybrid approach the neural network could be used to incrementally move the robot at larger steps while it’s far from its goal. As it gets closer, the system could automatically switch to the inverse Jacobian method for fine control at smaller step increments until the goal is reached. The arm would get to the target in less steps, allowing a lower number of iterations. Taking larger initial steps could possibly lower the total time of the task, depending on the hardware acceleration and on the complexity of the robot. Using this approach also means one can use a GPU to run the neural network, possibly lightening the workload on the CPU. Moreover, larger steps imply in lower data bandwidth requirements for the control loop, in the system that communicates with the servo actuators.

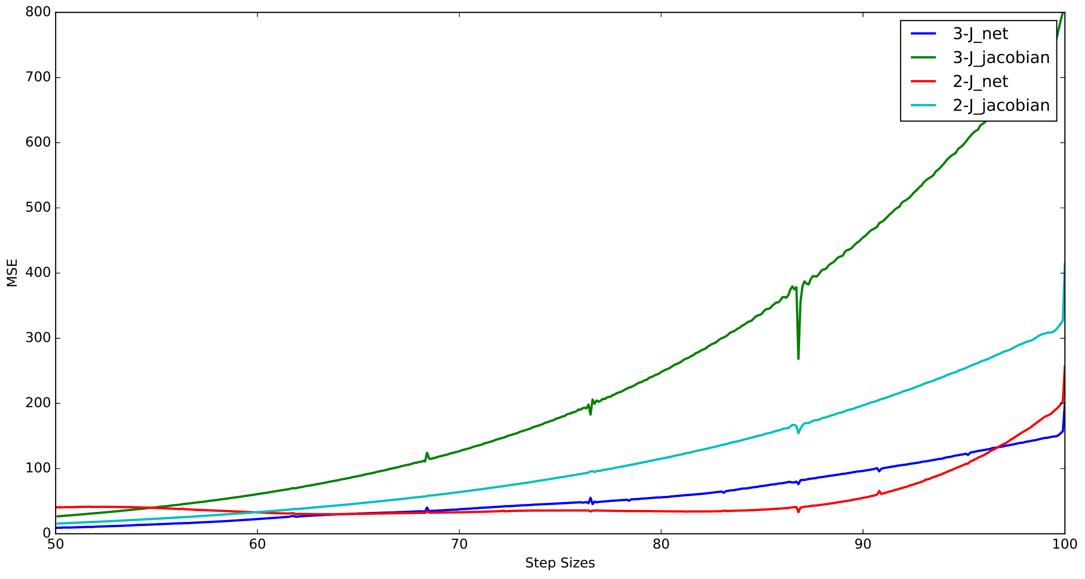
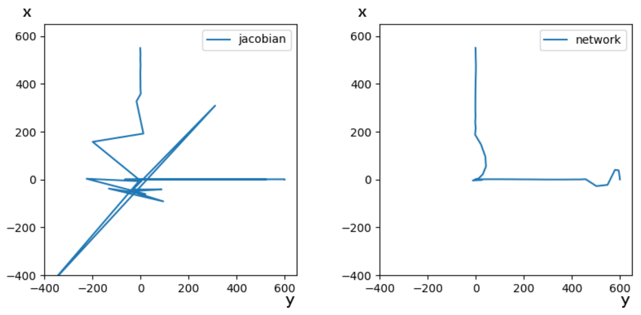


Fig. 5. 2-joint and 3-joint arms performing in the range between $50mm$ and $100mm$



(a) Inverse Jacobian method (b) Neural Network method

Fig. 6. Comparison of inverse Jacobian method and neural network performing near a singularity

For future work, our project includes applying this technique in 3D simulated robot arms, and in real robots. We are also planning a benchmark task to test the hybrid approach proposed here.

References

[1] Mark W Spong, Seth Hutchinson, Mathukumalli Vidyasagar, et al. *Robot modeling and control*, volume 3. Wiley New York, 2006.

[2] John J Craig. *Introduction to robotics: mechanics and control*, volume 3. Pearson/Prentice Hall Upper Saddle River, NJ, USA:, 2005.

[3] Frederic Chapelle and Philippe Bidaud. Closed form solutions for inverse kinematics approximation of general 6r manipulators. *Mechanism and machine theory*, 39(3):323–338, 2004.

[4] Lorenzo Sciavicco and Bruno Siciliano. *Modelling and control of robot manipulators*. Springer Science & Business Media, 2012.

[5] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[6] Kouichi Murakami and Hitomi Taguchi. Gesture recognition using recurrent neural networks. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 237–242. ACM, 1991.

[7] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[8] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[9] Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in neural information processing systems*, pages 545–552, 2009.

[10] Ahmed RJ Almusawi, L Canan Dülger, and Sadettin Kapucu. A new artificial neural network approach in solving inverse kinematics of robotic arm (denso vp6242). *Computational intelligence and neuroscience*, 2016, 2016.

[11] Adrian-Vasile Duka. Neural network based inverse kinematics solution for trajectory tracking of a robotic arm. *Procedia Technology*, 12:20–27, 2014.

[12] Luv Aggarwal, Kush Aggarwal, and Ruth Jill Urbanic. Use of artificial neural networks for the development of an inverse kinematic solution and visual identification of singularity zone (s). *Procedia Cirp*, 17:812–817, 2014.

[13] RaşIt KöKer. A genetic algorithm approach to a neural-network-based inverse kinematics solution of robotic manipulators based on error minimization. *Information Sciences*, 222:528–543, 2013.

[14] Raşit Köker, Cemil Öz, Tarık Çakar, and Hüseyin Ekiz. A study of neural network based inverse kinematics solution for a three-joint robot. *Robotics and autonomous systems*, 49(3-4):227–234, 2004.

[15] Z Bingul, HM Ertunc, and C Oysu. Comparison of inverse kinematics solutions using neural network for 6r robot manipulator with offset. In *Computational Intelligence Methods and Applications, 2005 ICSC Congress on*, pages 5–pp. IEEE, 2005.

[16] Eimei Oyama, Arvin Agah, Karl F MacDorman, Taro Maeda, and Susumu Tachi. A modular neural network architecture for inverse kinematics model learning. *Neurocomputing*, 38:797–805, 2001.

[17] Ali T Hasan, N Ismail, Abdel Magid S Hamouda, Ishak Aris, MH Marhaban, and HMAA Al-Assadi. Artificial neural network-based kinematics jacobian solution for serial manipulator passing through singular configurations. *Advances in Engineering Software*, 41(2):359–367, 2010.