Robot Kinematics
Machine Learning Report

Datasets
For each robot, I created two datasets. The first dataset was used in the traditional way, splitting the data into training and testing sets to train the model. The second dataset served as an additional validation test to evaluate the model's accuracy.

I also provided the option to consider or exclude sine and cosine data as input, as well as orientation quaternions as output, by modifying global variables. As shown in the function *parse.split_data*, the CSV file, transformed into an array using Pandas, is parsed with these options based on the specific robot under consideration.

Models
For the models, I used different types of neural networks with either 2 or 3 layers, evaluating various hyper-parameters through a search. These hyper-parameters included the number of neurons per layer, as well as the activation function, loss function, and optimizer used during network compilation, for a total of 160 combinations.

It was observed that dropout and regularization do not help with smaller numbers of neurons; in fact, they tend to increase the error on the test set.

For the 3DOF Robot, for example:

Configuration: Neurons = (12,12), with Activation = ReLU, Optimizer=Adam

- Loss=mean_squared_error, Dropout=True, Regularize=True
  Mean Pos Error: 0.03880353568175236
- Loss=mean_squared_error, Dropout=True, Regularize=False
  Mean Pos Error: 0.027752858171472213
- Loss=mean_squared_error, Dropout=False, Regularize=True
  Mean Pos Error: 0.02616633234204491
- Loss=mean_squared_error, Dropout=False, Regularize=False
  Mean Pos Error: 0.018233888439547506

On the other hand, by increasing the number of nodes in the network, regularization can actually improve the error, if the the mean absolute error is used as loss function:

Configuration: Neurons = (48, 48), with Activation = ReLU, Optimizer=Adam

- Loss=mean_absolute_error, Dropout=True, Regularize=True
  Mean Pos Error: 0.017697347351313755
- Loss=mean_absolute_error, Dropout=True, Regularize=False
  Mean Pos Error: 0.016369005529970143
- Loss=mean_absolute_error, Dropout=False, Regularize=True
  Mean Pos Error: 0.00598208213603072
- Loss=mean_absolute_error, Dropout=False, Regularize=False
  Mean Pos Error: 0.006165220054456053

2DOF Robot Best Model

The best-performing configuration from the hyper-parameters search was as follows:

- **Neurons:** *(32, 16)*
- **Activation:** *Tanh*
- **Loss:** *Mean Squared Error*
- **Optimizer:** Adam
- **Dropout:** False
- **Regularization:** *False*

Best Mean Total Error: 0.0036562471284967

3DOF Robot Best Model

The best-performing configuration from the hyper-parameters search was as follows:

- **Neurons:** *(48, 48)*
- **Activation:** *ReLU*
- **Loss:** *Mean Absolute Error*
- **Optimizer:** Adam
- **Dropout:** False
- **Regularization:** *True*

Best Mean Total Error: 0.0059820821360307

Model Evaluation

To evaluate the model, I implemented the following function, in the NeuralNetwork class:

```
def evaluate(self, X_test, y_test, verbose=2):
    print("Evaluation: ", self.model.evaluate(X_test, y_test, verbose=verbose))

    y_pred = self.model.predict(X_test)

    err_pos = tf.linalg.norm(y_pred[:, :self.dim] - y_test[:, :self.dim], axis=1)
    err_ori = None
    if self.out_orientation:
        err_ori = tf.linalg.norm(y_pred[:, self.dim:] - y_test[:, self.dim:], axis=1)

    if verbose:
        print(f"Mean error in position: {tf.reduce_mean(err_pos)}")
        if self.out_orientation:
            print(f"Mean error in orientation: {tf.reduce_mean(err_ori)}")

    return err_pos, err_ori
```

The model was evaluated using the following steps:

1. **Position Error:** The error in position was calculated as the Euclidean distance between the predicted and true positions.
2. **Orientation Error (if applicable):** If orientation was included as an output, the orientation error was similarly calculated as the L2 norm between the predicted and true orientation quaternions.
3. The mean errors for both position and orientation were then computed and displayed.

The model's performance is assessed based on these error metrics, providing insights into its accuracy and effectiveness in predicting both position and orientation.

Jacobian – 2DOF & 3DOF

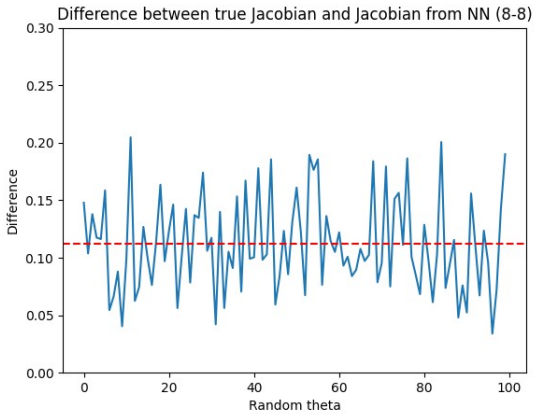
I created a set of 100 random theta vectors and used them as variables to evaluate the Jacobian of the trained model. I compared this Jacobian with the true Jacobian derived from the forward kinematics function, plotting the results using Matplotlib. As expected, it was observed that the average Jacobian error of the model changes depending on the number of neurons in each layer.

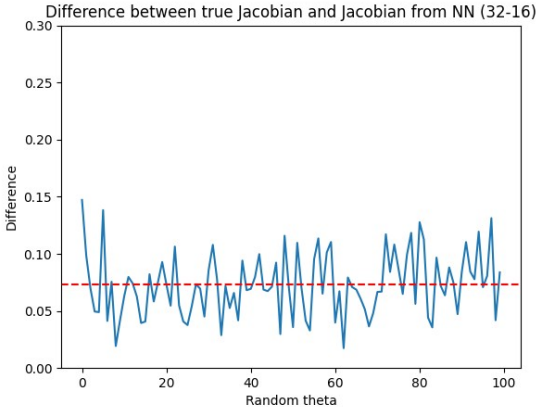
Jacobian Evaluation

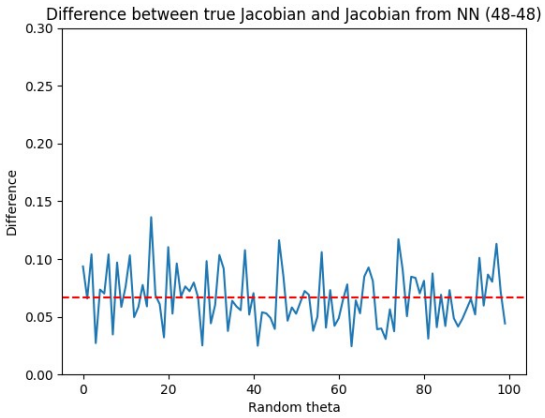
The function evaluates the difference between the Jacobian of a trained model and the true Jacobian computed from the forward kinematics function.

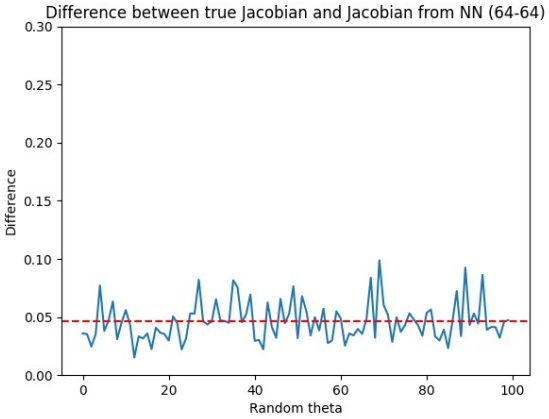
1. It iterates through each theta vector and calculates the model's Jacobian J;
2. It computes the true Jacobian using the forward kinematics function;
3. The norm of the difference between J and J_true is calculated as error for each theta.
4. The errors are appended to the returned list.

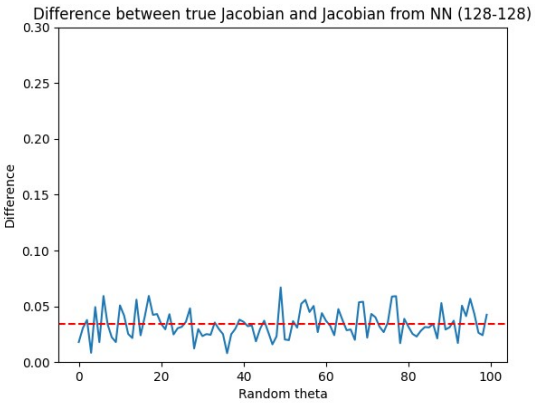
I observed that by increasing the number of neurons per layer, the Jacobian accuracy improves. Also, the 3DOF planar robot Neural Network model performs better than the 2DOF one. A shown in the figures, for the 3DOF robot the average error decreases from *0.11* for a (8,8) neural network to *0.075* for (32,16), and further down to *0.05* for (48,48). As the number of neurons continues to increase, the average error descrease, until a value of *0.032* for both (128,128) and (256,256).

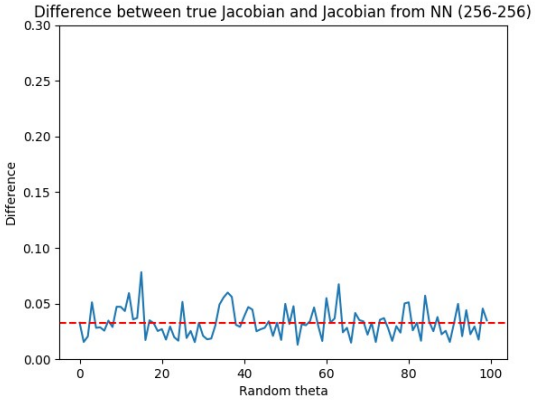












Inverse Kinematics

I wrote a function that, given an initial joint configuration and a target final position, returns the final joint configuration that positions the end-effector at the desired location. The function implements the two required algorithms and executes one based on the specified parameters:

- **Newton-Raphson:** $x_{n+1}=x_n-J^{-1}\cdot(x_n-r)$
  At each step, the Jacobian is evaluated at the current theta, then inverted and multiplied by the error between the model's forward kinematics at the current theta and the target position.
- **Levenberg-Marquardt:** $x_{n+1}=x_n-(J^T\cdot J+\lambda\cdot I)^{-1}\cdot J^T\cdot(x_n-r)$
  Instead of inverting the Jacobian directly, this method inverts $J^T\cdot J+\lambda\cdot I$ to avoid singularities and rank deficiencies. The factor λ could also be dynamic, adjusting based on the error to allow for larger or smaller steps. However, in this implementation, I left λ static.

Random Theta

By using random theta values and positions, I observed that the second method (Levenberg-Marquardt) converges more often than the first. Moreover, in the Newton-Raphson method, increasing the number of iterations does not improve the convergence of the algorithm.

- 100 iterations, Newton-Raphson
  Max iteration reached: 64 / 100
  Average error at max iteration: 0.318262
  Average error at break: -4.885498e-07
- 500 iterations, Newton-Raphson
  Max iteration reached: 70 / 100
  Average error at max iteration: 0.31624046
  Average error at break: -1.8805721e-07

- 100 iterations, Levenberg-Marquardt
  Max iteration reached: 82 / 100
  Average error at max iteration: 0.06085123
  Average error at break: 0.00099721
- 500 iterations, Levenberg-Marquardt
  Max iteration reached: 33 / 100
  Average error at max iteration: 0.118702054
  Average error at break: 0.00098943

In bounds Theta

However, when considering the robot's workspace and boundaries—i.e., using theta values within the limits and target positions within the feasible region—the behavior of both methods may differ.

- 100 iterations, Newton-Raphson
  Max iteration reached: 0 / 100
  Average error at max iteration: 0
  Average error at break: 7.303093851078299e-05

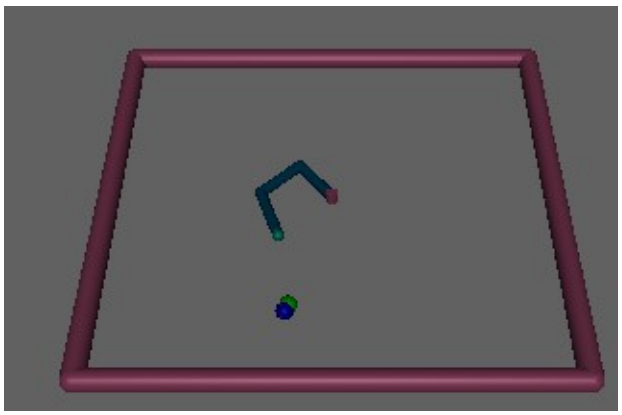
- 100 iterations, Levenberg-Marquardt
  Max iteration reached: 35 / 100
  Average error at max iteration: 0.024166390619107653
  Average error at break: 0.0009541043993802025

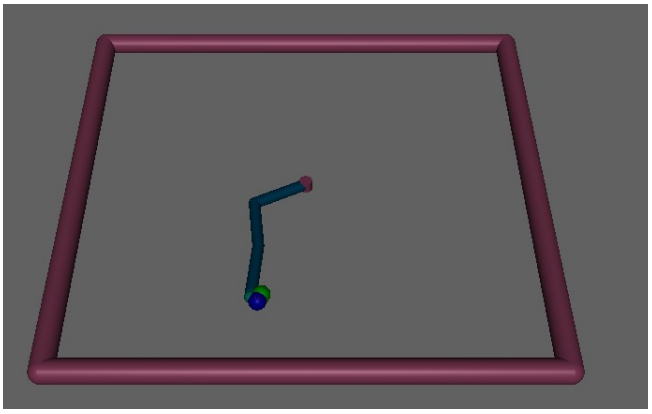
PID Controller

Given the initial random *theta* configuration and the final *theta* (obtained from the inverse kinematics algorithm), I created and fine-tuned a controller that moves the robotic arm from one configuration to another. The controller takes the current configuration as input and calculates the error relative to the desired configuration. Then, using the proportional, integral, and derivative gains (Kp, Ki, Kd), it computes an "action" to apply to the joint torques of the arm.

In the simulation, I placed three spheres:

- The **red sphere** indicates the desired position, which is input to the *INV KIN* algorithm.
- The **blue sphere** indicates the position calculated using the true forward kinematics (*FK*) from the final theta, output by the inverse kinematics algorithm.
- The **green sphere** indicates the position calculated using the model's *FK* from final theta.



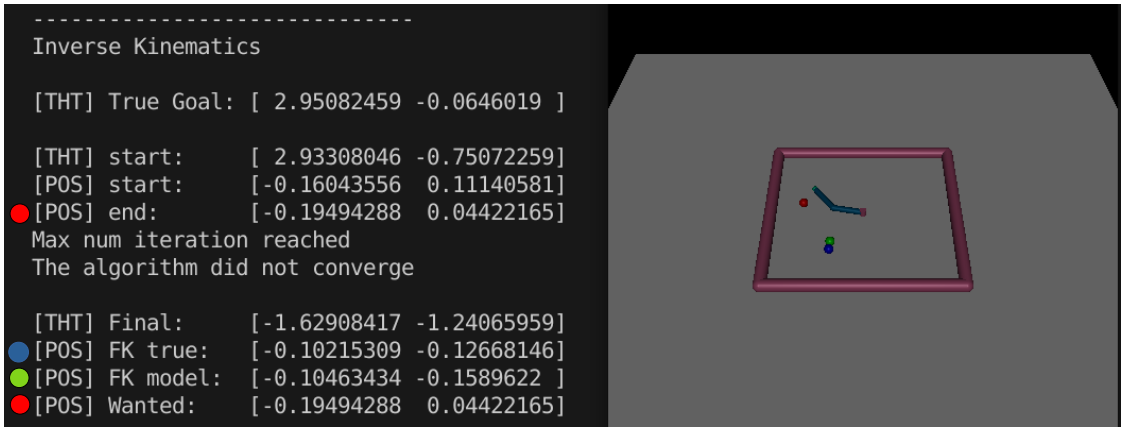


In the case shown, the red sphere is invisible because it is covered by the blue sphere, indicating a well-trained neural network capable of aligning the true *FK* with the learned *FK*.

Encountered Problems

It is observed that as the number of neurons in the network increases, these three spheres get closer to each other, indicating improved model accuracy. However, there are occasional errors:

1. **The three spheres are far apart**: This occurs when the inverse kinematics algorithm fails to converge, returning an incorrect final theta that is far from the desired position (the red sphere). A simple solution is to increase the number of iterations or change the algorithm.



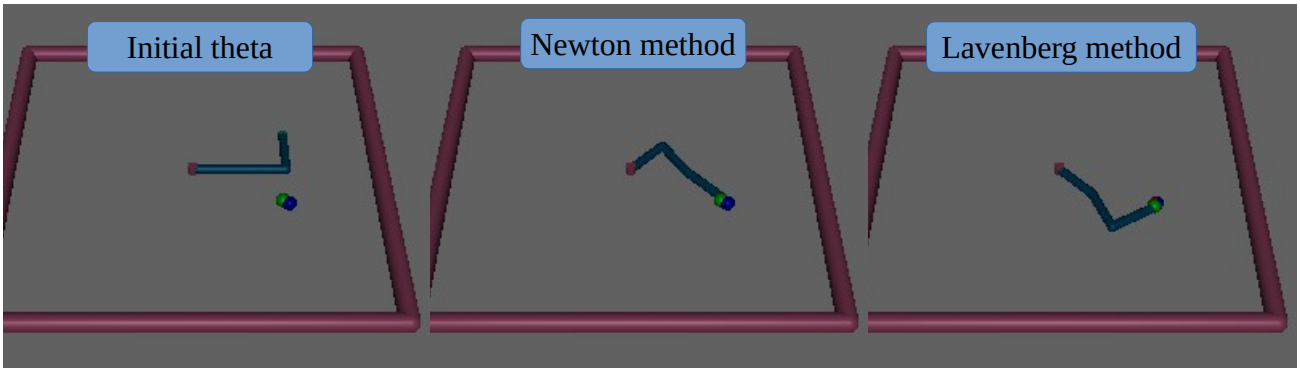
2. **Angles error despite convergence**: Occasionally, even when the inverse kinematics algorithm converges (i.e., when the spheres are close to each other), the returned configuration may be correct in terms of position but yield invalid angle values. This issue arises due to the redundancy of the robot for the given task: a 3DOF planar robot has multiple possible solutions for the joint angles corresponding to a single position. Additionally, the Jacobian does not account for joint boundaries, which can lead to invalid solutions in the inverse kinematics process. This problem does not occur with a 2DOF planar robot, where there are only two possible solutions: if the goal position is within the workspace, it's straightforward to retrieve the second solution from the first.

```
----------------------------
Inverse Kinematics

[THT] True Goal: [ 1.10141966 -1.46162337 -1.29524973]

[THT] start:     [-0.27351992 -0.54812819  1.14354608]
[POS] start:     [ 0.268376   -0.06602252]
[POS] end:       [ 0.13035995 -0.04570332]
Break at iteration 3

[THT] Final:     [-1.43097048 -2.31309507 -2.88471047]
[POS] FK true:   [ 0.02563094 -0.07623111]
[POS] FK model:  [ 0.13036004 -0.04570325]
[POS] Wanted:    [ 0.13035995 -0.04570332]
Position out of workspace
```

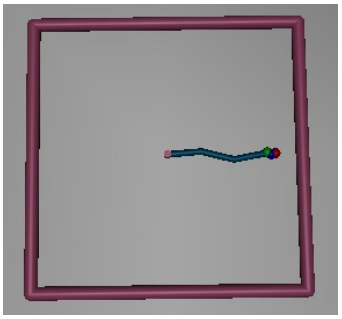
Here, for example, the generated theta (which respects the bounds) is different from the one returned by the inverse kinematics algorithm (which is out of bounds), even though the algorithm converged after only 3 iterations and with a minimal error on the position.

Singularities

It's also interesting to observe how the different algorithms behave. When initialized with $\theta=(0,0,\pi/2)$ and given the corresponding goal position for $\theta=(0,0,-\pi/2)$ based on the forward kinematics, neither algorithm reaches the expected configuration. The Newton method returns $(0.78,-1.82,0.182)$, while the other algorithm produces $(-0.79,-0.34,1.71)$, with both successfully avoiding the singularity at $(0,0,0)$.



Furthermore, when the position corresponding to the singularity is provided, neither algorithm converges—likely due to insufficient data in these edge cases. As shown in the figure on the right, although the arm can reach the red sphere, the inverse kinematics algorithm only returns an approximation, closer to the origin.



How the code works

There are several files in the project:

- **main.py**: This file handles dataset splitting, model compilation, training, evaluation, and implements the inverse kinematics and PID controller. It uses various global variables that can be adjusted as execution parameters:
    - *DIM* and *NJOINT*: Define the workspace dimension and the number of joints. Values can be (2,2) for a 2DOF planar robot, (2,3) for a 3DOF planar robot, and (3,5) for a 5DOF robot.
    - *IN_SINCOS* and *OUT_ORIENTATION*: Flags to use sine and cosine for input angles and orientation quaternions as output.
    - *NN*: A tuple specifying the number of nodes per layer for the neural network.
    - *VALIDATION*: When true, uses a separate validation dataset.
    - *NEWTON*: When true, enables the Newton method; otherwise, the Levenberg-Marquardt algorithm is used as algorithm for the inverse kinematic.
    - *NUM_IT*: Specifies the maximum number of iterations for the algorithm.
    - *STRESS_TEST*: Repeats the inverse kinematics algorithm 100 times for statistical analysis of convergence and errors.
- **models.py**: Contains the neural network class to create and evaluate the model. It provides utility functions to make the main code more readable.
- **inverse_kin.py**: Implements the two inverse kinematics algorithms.
- **pid.py**: Defines functions for creating the Mujoco environment and the PID controller class, which calculates joint torques based on a feedback chain with the current theta. The default PID constants are tuned for 2DOF and 3DOF robots.
- **hyper_tuning_NN.py:** is used to search the best hyper parameters for the neural network.

TODO

Here are a few improvements worth considering to enhance the results:

- Implement kinematics and PID control for the 5DOF robot.
- Incorporate orientation into the inverse kinematics and subsequent PID control.
- Develop an inverse kinematics algorithm that accounts for joint boundaries.