Report

Homework 1

Learning Forward Kinematics Using Neural Networks for Robotic Manipulators

Andrea Massignan
1796802

GitHub Repository Link

Contents

1 Introduction

1.1 Goal and Problem Statement

The primary objective of this homework is to develop and train neural network models capable of learning the forward kinematics (FK) of robotic manipulators with varying degrees of freedom (DOF), specifically targeting 2, 3, and 5 DOF robotic arms.

Forward kinematics involves determining the position and orientation of a robot's end-effector (EE) based on given joint angles. By training neural networks to approximate the FK function, aiming to create enough accurate models that can predict the EE states for arbitrary joint configurations.

2 Dataset Creation

2.1 Simulated Environment

To train the neural network models, a comprehensive dataset of the relationship between joint angles and end-effector positions was extracted and elaborated by generating with a simulated robotic environment(provided on the github link in the homework slides), designed to emulate the kinematic behavior of robotic arms with 2, 3, and 5 DOF, from which the logs of joint angles and corresponding end-effector states were extracted, as the basis for training the models.

2.2 Data Logging

For each robotic configuration denoted as r2, r3, and r5 corresponding to 2, 3, and 5 DOF respectively—the simulation was executed to log various joint angles and the resulting end-effector positions and orientations.

The logged data included:

- **Joint Angles**: The angles of each joint, denoted as $\theta_1, \theta_2, \ldots, \theta_n$, where n corresponds to the number of joints (2, 3, or 5).
- **Trigonometric Values**: The sine and cosine of each joint angle, i.e., $\sin(\theta_i)$ and $\cos(\theta_i)$, providing additional features that help in modeling the nonlinear relationships between joint angles and end-effector positions.
- **End-Effector States**: The Cartesian coordinates (x, y, z) and orientation components (qw, qx, qy, qz) of the end-effector, which serve as the target outputs for the forward kinematics model.

Each robotic configuration's data was saved into separate CSV log files, specifically:

- `logfile(r2).csv` for the 2-DOF robot.
- `logfile(r3).csv` for the 3-DOF robot.
- `logfile(r5).csv` for the 5-DOF robot.

Furthermore the datasets provided by the professors were used to train the neural network models, 4 datasets for each robot configuration, with different seeds, to compare the metrics.

2.3 Dataset Preparation

The data obtained from the simulation logs underwent several preprocessing steps to ensure suitability for training the neural network models, alongside the datasets provided by the professors.

The preprocessing pipeline included:

1. **Data Loading**: Each CSV file was read into a **pandas** DataFrame, in order to have an easy manipulation and analysis of the data.
2. **Dynamic Column Naming**: Based on the number of joints in each robot (r2, r3, r5), column names were dynamically generated to represent joint angles and their trigonometric values.
3. **Feature and Label Separation**: The dataset was divided into input features and target labels. The input features comprised the joint angles and their sine and cosine values, while the labels consisted of the end-effector's positional and orientational data.
4. **Train-Test Split**: To evaluate the model's performance and generalization capabilities, the dataset was split into training and validation sets using a random split.
5. **Feature Standardization**: The input features were standardized to have zero mean and unit variance using the `StandardScaler` from scikit-learn.

This preprocessing method ensured that the datasets were as clean, well-structured, and primed as possible for effective later model training.

3 Model Development and Training

3.1 Importing Libraries

In this homework I leveraged several common Python libraries essential for data manipulation, machine learning, and neural network construction:

- **Pandas and NumPy**: Utilized for efficient data handling and numerical computations.
- **Scikit-learn**: Employed for data preprocessing tasks such as train-test splitting and feature scaling.
- **TensorFlow and Keras**: The primary frameworks used for building, training, and evaluating the neural network models.

3.2 Dataset Preprocessing

The preprocessing of the dataset in the `load_and_preprocess_dataset` function involves the following steps to ensure a clean, well-structured data as possible, for training and evaluation purposes:

- **Loading the Dataset**: The dataset is loaded using the `pandas` function `read_csv()` with a specified delimiter to correctly parse the data.
- **Dynamic Column Generation**: To account for variations in the number of joints (`num_joints`), the function dynamically generates lists of joint column names (`joint_cols`), `cosine` and `sine` columns (`cos_cols` and `sin_cols` respectively). These columns capture both the angle values and their trigonometric transformations, which are crucial for kinematic calculations.
- **End-Effector Columns and Feature Selection**: Based on `num_joints`, different columns are selected for the `end-effector` values (`ee_cols`), with specific `feature_cols` defined for each robot configuration. The feature columns encompass joint angles and trigonometric transformations, tailored to each `2`, `3`, or `5` joint setup.
- **Dataset Shuffling**: To promote `random distribution` and prevent ordering biases, the dataset is `shuffled` using `sample` with a fixed `random_state`.
- **Column Validation**: An `error check` is performed to confirm that each required column in `feature_cols` and `ee_cols` is present, ensuring data integrity.
- **Feature and Label Assignment**: The `features` (`X`) are set as the values in `feature_cols`, while the `labels` (`y`) correspond to the `end-effector` values in `ee_cols`.

- **Train-Validation-Test Split**: The dataset is divided into `training`, `Test`, and `test` sets, with `70%` allocated for training, `15%` for validation, and `15%` for testing. This split is done using `train_test_split` with specific proportions to ensure representative data distributions across sets.
- **Standardization**: Both features and labels are `standardized` to have a mean of zero and unit variance, a critical step for `model training`. The `StandardScaler` is fitted on the `training data` and subsequently applied to the `Test` and `test` sets to maintain consistency.

This preprocessing pipeline results in `X_train`, `X_val`, `X_test`, `y_train`, `y_val`, and `y_test` sets, along with scalers (`scaler_X` and `scaler_y`) for each of these data segments, ensuring a robust, well-prepared dataset.

3.3 Model Building Based on Number of Joints

The function `build_model` constructs and compiles a neural network model with architecture and complexity tailored to the specific `num_joints` of the robot. I adjusted the depth, layer configurations, and regularization techniques depending on the task complexity associated with different robotic configurations.

- **Input Layer**: The function begins by defining the `input` layer based on `input_size`, which varies depending on the dataset's feature dimensionality.
- **Model for 2 Joints** (`num_joints` = 2): For a `2-joint` system (r2), a simpler architecture is employed. The model consists of:
  - `Dense` layers with `64` neurons each and `ReLU` activation.
  - `Batch Normalization` layers applied after each dense layer to standardize the inputs for each batch, helping stabilize training.
  - `Dropout` layers with a rate of `0.2` to prevent overfitting by randomly setting a fraction of inputs to zero during training.
- **Model for 3 Joints** (`num_joints` = 3): For a `3-joint` system (r3), a medium complexity model is built, including:
  - A larger initial `Dense` layer with `128` neurons for improved representation capacity.
  - A `Residual Block` comprising a `Dense` layer with `128` neurons, followed by batch normalization and dropout. This residual connection is created using the `Add` layer to combine the output of the residual block with the main flow, enhancing gradient flow and model accuracy.
  - Additional layers including a smaller `Dense` layer with `64` neurons, batch normalization, and dropout for further processing.
- **Model for 5 Joints** (`num_joints` = 5): For a more complex `5-joint` system (r5), a deeper model with multiple residual connections is constructed:
  - An initial `Dense` layer with `256` neurons and `ReLU` activation, followed by batch normalization and a `Dropout` rate of `0.2`.
  - Two `Residual Blocks`, each containing:
    - A `Dense` layer with `256` neurons, batch normalization, and dropout layers to create the residual flow.
    - `Add` layers to combine the residual output with the main flow, allowing for more complex feature learning while preserving gradient flow.
  - Subsequent `Dense` and batch normalization layers, with an additional `Dropout` layer, followed by an `output` layer to fit the continuous target values.
- The **Output Layer** doesn't have an activation function, first I tried with a `linear` activation function, but then I noticed that the model was performing better without it, so I removed it.
- **Optimizer Selection**: Based on the `chosen_optimizer` argument, one of three optimizers is selected:

  - `Adam` optimizer with a learning rate of `0.001`.
  - `RMSprop` optimizer with a learning rate of `0.001`.
  - `SGD` with `momentum` and `Nesterov` updates.

- **Model Compilation**: The model is compiled with a `Mean Squared Error (MSE)` loss and `Mean Absolute Error (MAE)` as a metric for performance monitoring.

3.4 Optimizer and Dataset Selection

The model training pipeline includes a user-driven selection process for both the `optimizer` and `dataset`. This flexibility enables tailored training based on specific requirements and computational constraints.

- **Optimizer Selection**: The optimizer is a critical component influencing the model's convergence speed and stability. Users are prompted to choose from three popular optimizers:
  - `Adam`: Adaptive Moment Estimation, known for fast convergence and minimal hyperparameter tuning requirements.
  - `RMSprop`: Root Mean Square Propagation, effective in managing rapidly changing gradients.
  - `SGD`: Stochastic Gradient Descent, chosen with momentum and Nesterov updates to enhance convergence and escape saddle points.

  An `input` prompt collects the user's choice, and an error check ensures only valid optimizers are selected, raising an exception if an unsupported optimizer is chosen.

- **Dataset Directory Selection**: Users select between two directories, `datasets` and `other_datasets`, for training data. Each directory caters to different needs:
  - `datasets`: Contains a single dataset per joint configuration, specifically designed for training on `2`, `3`, and `5-joint` robot configurations. This option was considered for faster training and initial model evaluation.
  - `other_datasets`: Contains `10` datasets per joint configuration, each named in the format `r2_20_100k.csv` to `r2_30_100k.csv` (for the 2-joint robot), and similarly for `r3` and `r5`. This directory is used to obtain a larger sample set for extensive metric calculations. The `input` prompt ensures that users are aware of the increased time requirements when opting for `other_datasets`.

  Error handling is in place to prevent invalid selections, raising a `ValueError` if the dataset input is not among the allowed options.

- **File Path Construction and Training Initiation**: Depending on the selected dataset directory:
  - If `datasets` is chosen, a single file path is constructed for each `num_joints` configuration (`2`, `3`, or `5`). This file path is then used to call `train_model`, where `single` is set to `True`.
  - If `other_datasets` is chosen, multiple paths are constructed for each `num_joints` configuration, iterating over a range to include all files from `20` to `23` for each joint configuration. This option calls `train_model` with `single` set to `False`, enabling multi-dataset training.

3.5 Training and Evaluation Process

The training and evaluation of the model are managed by the `train_model` and `evaluate_model` functions. These functions handle dataset loading, model training, validation, and testing, with systematic logging and checkpointing. The key steps are as it follows:

- **Directory Setup**: To organize outputs, the following directories are created if they do not already exist: `checkpoints`, `models`, `results`, and `models/scalers`. This ensures proper storage of model checkpoints, trained models, and results.

- **Dataset Loading and Preprocessing**: The dataset is loaded and preprocessed via the `load_and_preprocess_dataset` function as we described before, which standardizes the data for optimal model performance. Furthermore, data augmentation is performed to increase the dataset's robustness (adding noise), followed by shuffling to ensure random distribution.
- **Model Compilation and Callbacks**: A model is built and compiled based on the specified `num_joints` using the `build_model` function as previously specified. Several `callbacks` are used to enhance training efficiency:
  - `EarlyStopping`: Monitors `val_loss` and halts training if no improvement is observed for `10` epochs, restoring the best weights.
  - `ReduceLROnPlateau`: Reduces the learning rate by a factor of `0.5` if validation loss plateaus for `5` epochs, with a minimum learning rate of `1e-6`.
  - `ModelCheckpoint`: Saves the best model checkpoint based on `val_loss`, which is then reloaded after training.
- **Model Training**: The model is trained on the training set with a validation split for monitoring generalization. `Batch size` is set to `64` with an increased epoch count of `100` for a single dataset and `200` when training on multiple datasets, allowing for thorough learning and convergence, even if the callbacks halt training early.
- **Evaluation on Test Set**: After training, the `evaluate_model` function is called to assess the model's performance on the `test set`. The following metrics are calculated:
  - `Mean Absolute Error (MAE)`: Measures the average absolute difference between predictions and true values, reflecting model accuracy.
  - `Mean Squared Error (MSE)`: Assesses prediction error magnitude, penalizing larger errors more heavily than MAE.
  - `R-squared (`R^2`)`: Indicates the proportion of variance in the data explained by the model, with values closer to 1.0 showing better fit.

  The results are saved to a text file in the `results` directory, providing a record of each experiment's performance.
- **Model and Scaler Saving**: After training, the model is saved as a `Keras` file in the `models` directory. Additionally, `scaler` objects (`scaler_X` and `scaler_y`) used for data standardization are saved with `joblib` for future inverse transformations, facilitating reproducibility.
- **Optional Multi-Dataset Training**: If training across multiple datasets, each dataset undergoes the same loading, preprocessing, augmentation, and shuffling processes. The model architecture remains constant, but results for each dataset variation (e.g., different seeds or sample sizes) are documented individually in the results file.

3.6 Models Performance Metrics by Optimizer

The models were evaluated across different optimizers (Adam, RMSprop, and SGD) for each robot configuration (2-joint, 3-joint, and 5-joint). The following tables display the obtained metrics for each optimizer, including Test Mean Absolute Error (MAE), Test Mean Squared Error (MSE) and the R-squared (R^2) score.

2-Joint Robot Metrics

Optimizer: Adam	**Test MAE**	**Test MSE**	$\boldsymbol{R^2}$
Sample 1 (Seed 20)	0.010551	0.000263	0.998447
Sample 2 (Seed 21)	0.010837	0.000278	0.998406
Sample 3 (Seed 22)	0.010315	0.000250	0.998473
Sample 4 (Seed 23)	0.010794	0.000272	0.998390

Table 1: 2-Joint Robot Metrics with Adam Optimizer

Optimizer: RMSprop	**Test MAE**	**Test MSE**	$\boldsymbol{R^2}$
Sample 1 (Seed 20)	0.010460	0.000261	0.998517
Sample 2 (Seed 21)	0.011091	0.000291	0.998410
Sample 3 (Seed 22)	0.010421	0.000250	0.998469
Sample 4 (Seed 23)	0.010350	0.000251	0.998535

Table 2: 2-Joint Robot Metrics with RMSprop Optimizer

Optimizer: SGD	**Test MAE**	**Test MSE**	$\boldsymbol{R^2}$
Sample 1 (Seed 20)	0.012602	0.000358	0.997878
Sample 2 (Seed 21)	0.012174	0.000356	0.998108
Sample 3 (Seed 22)	0.012596	0.000385	0.997970
Sample 4 (Seed 23)	0.012617	0.000373	0.997937

Table 3: 2-Joint Robot Metrics with SGD Optimizer

3-Joint Robot Metrics

Optimizer: Adam	**Test MAE**	**Test MSE**	R^2
Sample 1 (Seed 20)	0.011360	0.000282	0.998794
Sample 2 (Seed 21)	0.011070	0.000246	0.998814
Sample 3 (Seed 22)	0.011508	0.000270	0.998529
Sample 4 (Seed 23)	0.010749	0.000235	0.998693

Table 4: 3-Joint Robot Metrics with Adam Optimizer

Optimizer: RMSprop	**Test MAE**	**Test MSE**	R^2
Sample 1 (Seed 20)	0.011299	0.000264	0.998593
Sample 2 (Seed 21)	0.011063	0.000261	0.998665
Sample 3 (Seed 22)	0.011511	0.000272	0.998587
Sample 4 (Seed 23)	0.010731	0.000241	0.998755

Table 5: 3-Joint Robot Metrics with RMSprop Optimizer

Optimizer: SGD	**Test MAE**	**Test MSE**	R^2
Sample 1 (Seed 20)	0.010322	0.000251	0.998861
Sample 2 (Seed 21)	0.010068	0.000234	0.998855
Sample 3 (Seed 22)	0.010449	0.000269	0.998750
Sample 4 (Seed 23)	0.009747	0.000227	0.998868

Table 6: 3-Joint Robot Metrics with SGD Optimizer

5-Joint Robot Metrics

Optimizer: Adam	**Test MAE**	**Test MSE**	R^2
Sample 1 (Seed 20)	0.008955	0.000145	0.998761
Sample 2 (Seed 21)	0.009026	0.000144	0.998716
Sample 3 (Seed 22)	0.023160	0.000876	0.988867
Sample 4 (Seed 23)	0.020046	0.000657	0.991606

Table 7: 5-Joint Robot Metrics with Adam Optimizer

Optimizer: RMSprop	**Test MAE**	**Test MSE**	R^2
Sample 1 (Seed 20)	0.012284	0.000252	0.996614
Sample 2 (Seed 21)	0.012497	0.000260	0.996594
Sample 3 (Seed 22)	0.012492	0.000258	0.996556
Sample 4 (Seed 23)	0.008146	0.000124	0.998907

Table 8: 5-Joint Robot Metrics with RMSprop Optimizer

Optimizer: SGD	Test MAE	Test MSE	R^2
Sample 1 (Seed 20)	0.031060	0.001732	0.973374
Sample 2 (Seed 21)	0.033296	0.002000	0.971397
Sample 3 (Seed 22)	0.031931	0.001816	0.972986
Sample 4 (Seed 23)	0.031745	0.001766	0.972918

Table 9: 5-Joint Robot Metrics with SGD Optimizer

4 Jacobian Calculation and Comparison Across 2, 3, and 5 DOF Robots

Jacobians were computed for 2, 3, and 5 Degrees of Freedom (DOF) robot configurations to assess the model's ability to approximate kinematics across different robot structures. For each configuration, the **learned Jacobians** from the trained models were compared to the **analytical Jacobians** derived directly from the kinematic equations. The following sections outline the methods for calculating both Jacobian types and the error metrics used to quantify their similarity.

Learned Jacobian Calculation for R2, R3, and R5

The learned Jacobians were computed through `automatic differentiation` in TensorFlow. For each robot, the forward kinematics (FK) model provided an end-effector (EE) pose prediction based on joint angles. Key steps in the learned Jacobian computation include:

1. Defining joint angles q_1, q_2 (and q_3, q_4, q_5 for higher DOFs) as `tf.Variable` types to enable differentiation.
2. Constructing an **input feature vector** that includes joint angles and their trigonometric values (cosines and sines), tailored to the model's input requirements.
3. Scaling the input features using the `scaler_X` to match the model's expected input range.
4. Using `GradientTape` to track joint angle variables and compute partial derivatives of each component of the EE pose with respect to each joint angle, thus forming the learned Jacobian matrix.

The learned Jacobian dimensions for each configuration were:

- **R2 (2 DOF)**: A 2×2 Jacobian matrix for EE position (x, y) and orientation components (qw, qz).
- **R3 (3 DOF)**: A 3×3 Jacobian matrix with similar pose components as R2, adapted for three joints.
- **R5 (5 DOF)**: A 3×5 Jacobian matrix accounting for EE position (x, y, z) and orientation components (qw, qx, qy, qz), considering the 5-joint configuration.

Analytical Jacobian Calculation for R2, R3, and R5

The **analytical Jacobians** were derived based on each robot's forward kinematic equations. Given link lengths $l_1, l_2, \ldots, l_n$ for each configuration, the end-effector's position and orientation were expressed in terms of cumulative angles. Partial derivatives were then computed to form the Jacobian matrix for each joint angle.

R2 Analytical Jacobian

For the 2 DOF planar robot, the analytical Jacobian matrix included derivatives of x, and y with respect to q_1 and q_2, resulting in a 2×2 matrix. Link lengths were provided (on the slides) as constants to compute the position derivatives $l_1 = 0.1, l_2 = 0.1$.

R3 Analytical Jacobian

For the 3 DOF robot, the analytical Jacobian extended to include derivatives with respect to x and y, resulting in a 2×3 matrix. The cumulative joint angles were used to compute the position derivatives, and link lengths $l_1 = 0.1, l_2 = 0.1, l_3 = 0.1$.

R5 Analytical Jacobian

For the 5 DOF configuration, the analytical Jacobian was a 5×5 matrix capturing the EE position and orientation derivatives with respect to each joint angle. The cumulative joint angles were used to compute the position derivatives again from the provided link lengths $l_1 = 0.1, l_2 = 0.1, l_3 = 0.1, l_4 = 0.1, l_5 = 0.1$.

Comparison and Error Metrics

For each configuration, the learned and analytical Jacobians were compared by calculating the difference matrix, defined as:

$$\text{Difference} = J_{\text{learned}} - J_{\text{analytical}}$$

The **Frobenius norm** of this difference matrix provided a quantitative measure of error, computed as:

$$\|J_{\text{learned}} - J_{\text{analytical}}\|_F$$

The Frobenius norm reflects the overall deviation between the two Jacobians, with lower values indicating better approximation by the learned Jacobian. Results for each sample included joint angles, predicted and actual EE poses, learned and analytical Jacobians, and error norms, demonstrating the model's accuracy in approximating robot kinematics across configurations.

Results were recorded in a CSV file for each robot configuration, documenting the computed Jacobians and error norms for reproducibility and further analysis.

Metrics Comparison Across R2, R3, and R5 Robots

The following tables present the mean of Frobenius norm error metrics for the learned Jacobians compared to the analytical Jacobians across the 2, 3, and 5 DOF robot configurations. The error norms provide insights into the model's kinematic approximation accuracy for different robot structures:

Robot Configuration	**Mean Frobenius Norm Error**
2 DOF Robot (R2)	0.16005
3 DOF Robot (R3)	0.05674
5 DOF Robot (R5)	0.74656

Table 10: Mean Frobenius Norm Error Across R2, R3, and R5 Robots

We can notice that the mean error increases with the number of DOFs, indicating a higher complexity in approximating kinematics for the 5 DOF robot compared to the 2 and 3 DOF robots.

4.1 (Optional) Inverse Kinematics for R2

This subsection details the implementation of Inverse Kinematics (IK) for a two-degree-of-freedom (2 DOF) planar robotic arm (R2). The IK problem involves determining the joint angles that position the robot's end-effector (EE) at a desired Cartesian coordinate. Two numerical methods are used: the Newton-Raphson method and the Levenberg-Marquardt method.

4.1.1 Inverse Kinematics Methods

The core IK functionalities are encapsulated within two functions: `inverse_kinematics_newton_raphson` and `inverse_kinematics_levenberg_marquardt`.

Newton-Raphson Method The Newton-Raphson method iteratively approximates the solution by linearizing the nonlinear system around the current estimate. It seeks to find joint angles $\mathbf{q} = [q_0, q_1]^T$ such that the resulting EE pose $\mathbf{p} = [p_x, p_y]^T$ matches the target pose $\mathbf{p}_{\text{target}}$.
Function Signature:

```
def inverse_kinematics_newton_raphson(model, jacobian_func, target_pose, initial_guess,
                                      scaler_X, scaler_y, max_iterations=100,
                                      tolerance=0.03, max_delta_q=0.1):
```

Procedure:

1. **Initialization**: Initialize joint angles $\mathbf{q}$ with `initial_guess`.
2. **Iterative Refinement**: For each iteration:
   (a) **Forward Kinematics**: Compute the current EE pose $\mathbf{p} = \text{FK}(\mathbf{q})$.
   (b) **Error Calculation**: Determine the error vector $\mathbf{e} = \mathbf{p}_{\text{target}} - \mathbf{p}$.
   (c) **Convergence Check**: If $\|\mathbf{e}\| <$ `tolerance`, return the current joint angles.
   (d) **Jacobian Computation**: Calculate the Jacobian matrix $\mathbf{J}$.
   (e) **Delta Computation**: Compute the update $\Delta\mathbf{q} = \mathbf{J}^{\dagger}\mathbf{e}$.
   (f) **Step Size Limiting**: Constrain $\Delta\mathbf{q}$ within `max_delta_q`.
   (g) **Joint Angle Update**: Update joint angles $\mathbf{q} \leftarrow \mathbf{q} + \Delta\mathbf{q}$.
3. **Non-Convergence Handling**: If not converged, return the current joint angles.

Levenberg-Marquardt Method The Levenberg-Marquardt (LM) method combines the Gauss-Newton method and gradient descent. It introduces a damping factor to control the step size and direction.
Function Signature:

```
def inverse_kinematics_levenberg_marquardt(model, jacobian_func, target_pose, initial_guess,
                                           scaler_X, scaler_y, max_iterations=100,
                                           tolerance=0.03, max_delta_q=0.1, damping_factor=0.01):
```

Procedure:

1. **Initialization**: Initialize joint angles $\mathbf{q}$ with `initial_guess` and set damping factor λ.
2. **Iterative Refinement**: For each iteration:
   (a) **Forward Kinematics**: Compute the current EE pose $\mathbf{p} = \text{FK}(\mathbf{q})$.
   (b) **Error Calculation**: Determine the error vector $\mathbf{e} = \mathbf{p}_{\text{target}} - \mathbf{p}$.
   (c) **Convergence Check**: If $\|\mathbf{e}\| <$ `tolerance`, return the current joint angles.
   (d) **Jacobian Computation**: Calculate the Jacobian matrix $\mathbf{J}$.

(e) **Normal Equations Formation**: Compute $\mathbf{J}^T\mathbf{J}$ and $\mathbf{J}^T\mathbf{e}$.

(f) **Damping Application**: Modify the normal equations by adding $\lambda\mathbf{I}$.

(g) **Delta Computation**: Solve $(\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})\Delta\mathbf{q} = \mathbf{J}^T\mathbf{e}$.

(h) **Step Size Limiting**: Constrain $\Delta\mathbf{q}$ within `max_delta_q`.

(i) **Joint Angle Prediction**: Predict new joint angles $\mathbf{q}_{\text{new}} = \mathbf{q} + \Delta\mathbf{q}$.

(j) **Error Assessment**: Calculate new error $\mathbf{e}_{\text{new}} = \mathbf{p}_{\text{target}} - \mathbf{p}_{\text{new}}$.

(k) **Update Decision**: Accept or reject the update based on error reduction.

3. **Non-Convergence Handling**: If not converged, return the current joint angles.

4.1.2 Useful Utility Functions

Enforce Joint Limits **Function Signature:**

```
def enforce_joint_limits(q, lower_limits, upper_limits):
```

Purpose: Ensures joint angles remain within physical constraints.

4.1.3 Main Execution Flow

The main execution flow orchestrates loading models and data, and applying IK methods to sample target poses.

1. **Path Definitions and Validations**: Define file paths and check for existence.
2. **Model and Scaler Loading**: Load the trained model and scalers.
3. **Dataset Preparation**: Read the dataset and sample target poses.
4. **Link Lengths Specification**: Define robot link lengths.
5. **Target Poses and Initial Guesses**: Extract target EE poses and define initial guesses.
6. **Joint Limits Definition**: Specify joint angle bounds.
7. **Inverse Kinematics Execution**: Apply IK methods to each target pose.
8. **Iteration Logging**: Provide real-time feedback on convergence status.

4.1.4 Workflow Analysis

Initialization Initialize joint angles $\mathbf{q}$ with `initial_guess` and maintain a history list.

Error Computation and Convergence Check Compute the current EE pose and error vector $\mathbf{e}$. Check if $\|\mathbf{e}\|$ is below `tolerance`.

Jacobian Matrix Calculation Compute the Jacobian matrix $\mathbf{J}$ using `jacobian_func`.

Delta Computation and Joint Angle Update In Newton-Raphson, compute $\Delta\mathbf{q} = \mathbf{J}^{\dagger}\mathbf{e}$. In LM, compute $\Delta\mathbf{q} = (\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I})^{-1}\mathbf{J}^T\mathbf{e}$.

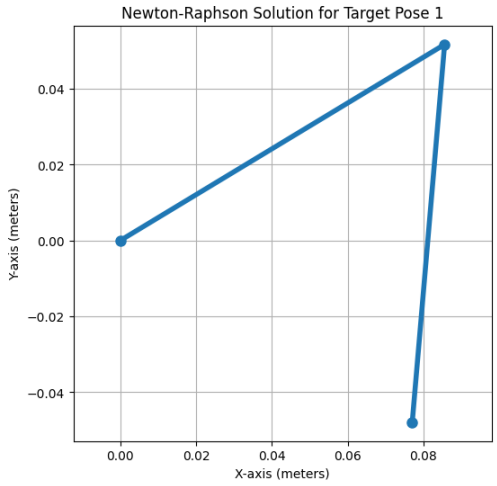
Step Size Limiting Constrain $\Delta\mathbf{q}$ within `max_delta_q`.

Adaptive Damping in Levenberg-Marquardt Adjust damping factor λ based on error reduction.

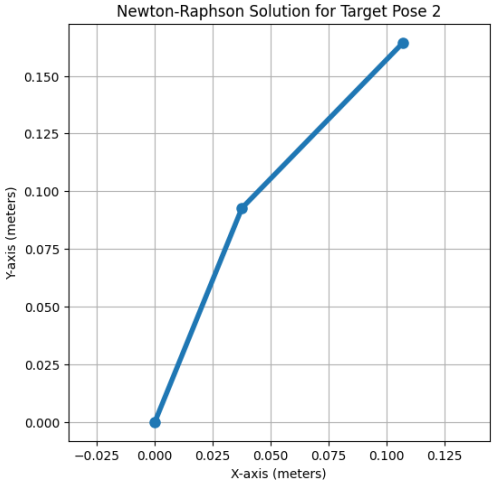
Joint Limits Enforcement Ensure joint angles remain within bounds using `enforce_joint_limits`.

4.1.5 Verification and Visualization

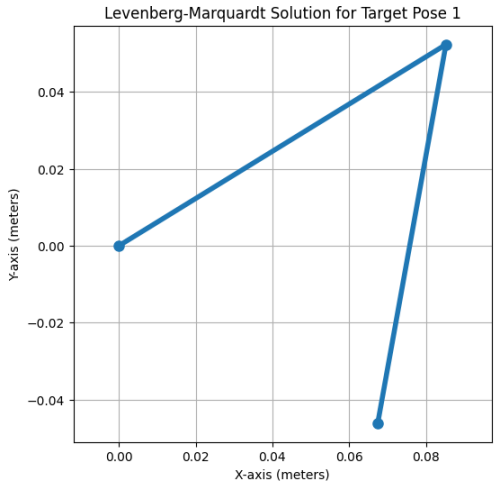
Verify the final EE pose and visualize the robot's configuration. The results, on two of the example target poses, are displayed as following:



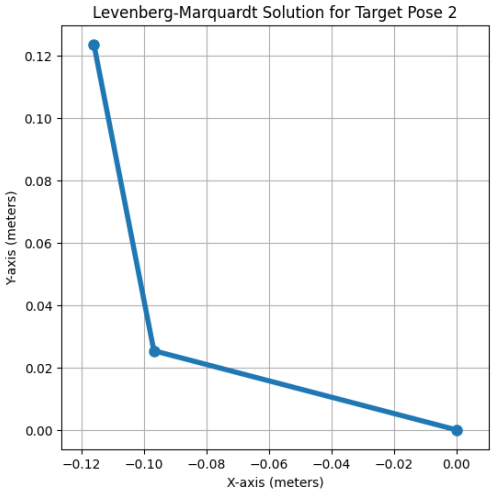
(a) Newton-Raphson Method for Target Pose 1



(b) Newton-Raphson Method for Target Pose 2 (Not Converging)



(c) Levenberg-Marquardt Method for Target Pose 1



(d) Levenberg-Marquardt Method for Target Pose 2

Figure 1: Comparison of Newton-Raphson and Levenberg-Marquardt Methods for Target Poses

5 Conclusion

In this report, I have presented a comprehensive overview of the robotic arm kinematics learning and inverse kinematics problem. I have detailed the model architecture, training pipeline, optimizer selection, dataset handling, and evaluation metrics. The performance of the models across different optimizers and robot configurations was analyzed, providing insights into the model's accuracy and generalization capabilities.

Furthermore, the Jacobian calculation and comparison across 2, 3, and 5 DOF robots were discussed, highlighting the model's kinematic approximation accuracy. The mean error metrics indicated higher complexity in approximating kinematics for the 5 DOF robot compared to the 2 and 3 DOF robots.

Lastly, I have demonstrated the implementation of Inverse Kinematics for a 2 DOF planar robot using the Newton-Raphson and Levenberg-Marquardt methods. The iterative refinement process, error computation, Jacobian calculation, and joint angle updates were discussed, showcasing the methods' convergence and accuracy in reaching target poses, even if with a higher tolerance.