This document provides a detailed analysis of Homework 1 for Robot Kinematics. The primary objective is to explore forward and inverse kinematics, data-driven robot control methods, and reinforcement learning for robotic systems. The report covers data generation and preprocessing, the implementation of various kinematic algorithms, and advanced learning techniques for robotic control. Performance metrics and hyper-parameter optimization are discussed, with a focus on practical implementation and results analysis.

Forward Kinematics

Forward kinematics involves calculating the position and orientation of the robot's end-effector based on its joint parameters. It provides a deterministic solution to map joint angles to Cartesian coordinates. This chapter focuses on implementing forward kinematics for robotic arms with varying degrees of freedom (DOF), including 2 DOF, 3 DOF, and 5 DOF configurations. The section also discusses the algorithms utilized and the challenges encountered when extending kinematic models to higher DOF systems.

0.1 Models

0.1.1 2 DOF

In the case of a 2 DOF robotic arm, the forward kinematics problems involves predicting the Cartesian coordinates (x, y) of the end-effector and its orientation components (w, z), given two joint angles j_0, j_1 and their precomputed trigonometric functions. This study explores a data-driven approach using a **Linear Regression** model over a transformed feature space.

Kernel Trick and Linear Regression

The implemented model combines a **Radial Basis Function (RBF) kernel transformation** with a **Linear Regression model**. This approach transforms the original input space into a higher-dimensional feature space where linear regression can better capture complex, non-linear relationships. The RBF transformation is particularly suited for problems where the relationship between input features and target outputs is non-linear but can be approximated locally by a Gaussian kernel.It has a reduced computational overhead compared to traditional kernel methods like Support Vector Machines (SVMs).

Data Preparation

The data is divided into position data and rotation data and the data preparation for each one involved the following steps:

1. **Normalization**: All input features were scaled using *StandardScaler* scaling to ensure uniformity and to avoid dominance of features with larger magnitudes during model training.

2. **Splitting**: The dataset was divided into training and test sets to evaluate the generalization capability of the model. Typically, 80% of the data was used for training, and 20% was reserved for testing.

3. **RBF Transformation**: The input features were transformed into a higher-dimensional space using the RBF kernel. The parameters of the transformation, such as γ (controls kernel width) and $n\_components$ (number of features in the transformed space), were optimized during training.

Training and Grid Search

The model was trained using **Linear Regression** on the RBF-transformed input features. A grid search was performed to optimize the hyperparameters of the RBF transformation:

- γ: Determines the spread of the RBF kernel. Smaller values lead to a smoother transformation, while larger values capture fine-grained relationships.

- $n\_components$: Defines the dimensionality of the transformed feature space. Higher values generally allow better approximation of the RBF kernel but increase computational cost.

Grid Search Process:

1. A grid of hyperparameter combinations for γ and $n\_components$ was defined (e.g., $\gamma = \{0.001, 0.1, 0.5\}$,$n\_components = \{50, 100, 200\}$).

2. For each combination, the model was trained both for the position and orientation prediction, on the transformed training set.

3. The performance was evaluated on the test set using **Mean Squared Error (MSE)** and **R-squared ($\mathbf{R^2}$)** metrics.

4. The combination of γ and $n\_components$ that minimized MSE was selected as the best configuration.

Results and Discussion

The grid search results reveal significant insights about the relationship between the hyperparameters γ and $n\_components$ and the model's performance. **These are the results for the position**:

```
Params: gamma=0.001, n_components=50, MSE=0.000000185, R2=0.999977801
Params: gamma=0.001, n_components=100, MSE=0.000000180, R2=0.999978379
Params: gamma=0.001, n_components=200, MSE=0.000000178, R2=0.999978589
Params: gamma=0.1, n_components=50, MSE=0.000005046, R2=0.999398967
Params: gamma=0.1, n_components=100, MSE=0.000000207, R2=0.999975222
Params: gamma=0.1, n_components=200, MSE=0.000000179, R2=0.999978508
Params: gamma=0.5, n_components=50, MSE=0.000180775, R2=0.978953006
Params: gamma=0.5, n_components=100, MSE=0.000003542, R2=0.999574748
Params: gamma=0.5, n_components=200, MSE=0.000000197, R2=0.999976347

Best Parameters:
gamma: 0.001, n_components: 200
Best MSE: 0.000000178, Best R2: 0.999978589
```

Key Observations for positions

1. **Impact of γ (Kernel Width)**:
   - Smaller values of γ ($\gamma = 0.001$) lead to better performance, with the lowest **MSE** ($1e^{-7}$) and highest $\mathbf{R^2}$ (0.999978).
   - Larger values of γ reduce performance, especially for lower *n_components*, indicating that excessively localized kernels fail to generalize well.
2. **Impact of *n_componets* (Dimensionality of Transformed Space)**:
   - Increasing *n_components* generally improves the performance, as shown by the case of $\gamma = 0.5$ and *n_components* $= 50, 100, 200$.
   - For smaller *n_components* (*n_components* $= 50$), performance drops, especially for larger γ values, as the feature space lacks sufficient dimensionality to capture the non-linear relationships.
3. **Best Parameters**: The combination $\gamma = 0.001$ and *n_components* $= 200$ achieves the best balance, with the lowest MSE ($1.78e^{-7}$) and highest R^2 (0.999978).

The results suggest that a moderately localized kernel and a sufficiently high-dimensional feature space are ideal for this task. These settings allow the model to approximate the non-linear relationships effectively without overfitting or losing generalization.

Results for the orientation:

```
Params: gamma=0.001, n_components=50, MSE=0.000046206, R2=0.999901035
Params: gamma=0.001, n_components=100, MSE=0.000001855, R2=0.999996037
Params: gamma=0.001, n_components=200, MSE=0.000001691, R2=0.999996391
Params: gamma=0.1, n_components=50, MSE=0.000379068, R2=0.999193927
Params: gamma=0.1, n_components=100, MSE=0.000002291, R2=0.999995104
Params: gamma=0.1, n_components=200, MSE=0.000001697, R2=0.999996378
```

```
Params: gamma=0.5, n_components=50, MSE=0.008425961, R2=0.981891810
Params: gamma=0.5, n_components=100, MSE=0.000104996, R2=0.999776346
Params: gamma=0.5, n_components=200, MSE=0.000002049, R2=0.999995630

Best Parameters:
    gamma: 0.001, n_components: 200
    Best MSE: 0.000001691, Best R2: 0.999996391
```

Key Observations for orientation

1. **Effect of γ on Model Performance**:
   - Lower values of γ (e.g., 0.001) consistently resulted in the best performance, achieving the lowest MSE ($1.691e^{-6}$) and highest R^2. A smaller γ value corresponds to a wider RBF kernel, which provides smoother approximations. This appears well-suited for the dataset used in this problem.
2. **Effect of *n_components* on Model Performance**:
   - For all γ values, increasing *n_components* consistently improved the model's performance.
   - For instance, with `gamma=0.5`, increasing *n_components* from 50 to 200 reduced the **MSE** from $8.43e^{-3}$ to $2.05e^{-6}$ and increased $\mathbf{R^2}$ from 0.981 to 0.999. This trend highlights the benefit of using more features in the transformed RBF space, likely because it enhances the model's ability to approximate complex relationships.
3. **Best Parameters**:
   - The optimal parameters were identified as $\gamma = 0.001$ and *n_components* = 200.
   - These parameters resulted in the lowest **MSE** and the highest $\mathbf{R^2}$, indicating the best balance between model complexity and predictive accuracy.

Jacobian Evaluation

To evaluate the accuracy and reliability of the learned kinematic model, I compared the **analytical Jacobian matrix**—derived from the theoretical kinematics equations of the robot—with the **learned Jacobian matrix**, computed numerically from the trained regression model. To compare the two Jacobians, the **Frobenius norm** of their difference was calculated. This metric quantifies the overall discrepancy between the learned and analytical Jacobians.

The result is

```
Frobenius norm of difference: 2.995346
```

A value of **2.995346** indicates that the learned model performs reasonably well in approximating the analytical Jacobian but has noticeable discrepancies. This may be given by the fact that with the Frobenius form the local derivatives discrepancies are highlighted or it may be that the jacobian matrix is not well calculated.

0.1.2 3 DOF

In the case of a 3 DOF robotic arm, the forward kinematics problems involves predicting the Cartesian coordinates (x, y) of the end-effector and its orientation components (w, z), given three joint angles j_0, j_1, j_2 and their precomputed trigonometric functions. This study explores a data-driven approach using the **Gradient Boosting** model.

Gradient Boosting Regressor

Gradient boosting regressor is a machine learning model used for regression tasks. It works by combining multiple weak learners (decision trees) to create a strong predictive model.

Gradient Boosting builds the model sequentially, adding one tree at a time. Each new tree is trained to correct the errors made by the previous trees. Each new tree is trained on the residuals (errors) of the previous trees. This means that the model focuses on the data points that were previously predicted incorrectly, gradually improving its accuracy. The predictions of all the trees are combined using weighted averaging. Trees that perform better on the training data are given higher weights in the final prediction. Gradient Boosting is known for its high predictive accuracy, often outperforming other machine learning algorithms.

Data Preparation

Since the target data are the position and the orientation of the end-effector, these two can be handled separately. Thus the data is divided into two parts: *position data* and *orientation data*. This helps to better focus on the single type of data.

Training

Since the model is sensitive to the hyperparameters, it is important to set the latter:

number of estimators This sets the number of boosting stages (decision trees) to be used. Increasing this value can improve accuracy but also increases computation time and overfitting.

learning rate It controls the contribution of each tree to the final prediction. A lower value leads to slower learning but potentially better generalization.

max depth This limits the maximum depth of each decision tree, preventing overfitting.

I experimented with various combinations of the number of estimators and maximum tree depth for both position and orientation prediction tasks, while keeping the learning rate constant. The tested configurations are summarized in the following table:

Number of estimators	max depth
1000	2
500	4
250	8
125	16

Results for position

Number of estimators	max depth	MSE x	MSE y
1000	2	$4.7e^{-3}$	$4.5e^{-3}$
500	4	$8.13e^{-5}$	$9e^{-5}$
250	8	$7.27e^{-5}$	$3.55e^{-5}$
125	16	$6.11e^{-5}$	$1.7e^{-}5$

Results for orientation

Number of estimators	max depth	MSE w	MSE z
1000	2	0.37	0.39
500	4	0.29	0.32
250	8	0.29	0.32
125	16	0.29	0.33

Discussion

The results for predicting the position (x, y) using Gradient Boosting indicate a clear improvement in performance as the model complexity increases (i.e., higher max_depth and fewer $n\_estimators$): The mean squared error decreases significantly as the maximum depth of the trees increases from 2 to 16, with the lowest MSE achieved when $max\_depth = 16$ and $n\_estimators = 125$. This suggest that deeper trees are better able to capture the non-linear relationships between the joint angles and the Cartesian coordinates.

The results for predicting the orientation (w, z) show a less pronounced improvement in performance as max_depth increases. This could be due to limitations in the dataset or the model's inability to fully capture the non-linear relationships required for precise quaternion predictions.

0.1.3 5 DOF

In the case of a 5 DOF robotic arm, the forward kinematics problem involves predicting the Cartesian coordinates (x, y) of the end-effector and its orientation components (w, z), given the five joint angles $j_0, j_1, \ldots, j_4$ and their precomputed trigonometric functions. This study explores a data-driven approach using a **feedforward neural network (FNN)**.

Neural Network Architecture

Input Layer The input layer consists of **15 neurons**, corresponding to the following inputs:

$$j_0, j_1, \ldots, j_4, \cos(j_0), \cos(j_1), \ldots, \cos(j_4), \sin(j_0), \ldots, \sin(j_4)$$

Hidden Layers Based on the **Universal Approximation Theorem**, we know that a neural network with at least one hidden layer can approximate any Borel measurable function. However, a single hidden layer with a large number of neurons may lead to overfitting and optimization difficulties. To balance complexity and performance, I tested multiple architectures, varying the width and number of hidden layers:

- 2 hidden layers with 32, 16 neurons, respectively.
- 3 hidden layers with 64, 32, and 16 neurons, respectively.
- 4 hidden layers with 128, 64, 32 and 16 neurons, respectively.

By increasing the number of layers and neurons, the model is expected to be able to represent more complex non-linear functions, but this may also lead to the risk of overfitting. In general, a more complex network may be able to better capture the nuances in the data, but it may also overfit the training set, losing generalization on the test data. All hidden layers use the **ReLU activation function**, which prevents saturation, enables the model to capture nonlinear relationships, and mitigates the vanishing gradient problem.

Output Layer The output layer consists of 4 neurons, corresponding to the predicted Cartesian coordinates (x, y) and the quaternion coordinates (w, z). The **linear activation function** is used here, as this is a regression problem and we need to avoid saturation and disappearing gradient issues.

Loss Function The **Mean Squared Error (MSE)** is used as the loss function, as it minimizes the squared differences between the predicted and actual values. Furthermore, minimizing MSE corresponds to maximizing the likelihood under the assumption of Gaussian noise in the errors. This assumption is reasonable, as the random errors are likely generated by sensor noise.

Optimizer To efficiently optimize the network, I used the **Adam optimizer**, which is an advanced variant of gradient descent. Adam combines momentum and adaptive learning rates, providing faster convergence and robustness to noisy gradients.

Data Preparation

The data is normalized to ensure that all input features are on a similar scale, facilitating efficient training. Since the neural network uses gradient descent for optimization, when features have different scales, the gradients for features with larger values can dominate the learning process, leading to slower convergence or even getting stuck in local minima. Therefore, normalization helps bring all features to a similar scale, allowing the optimization algorithm to converge more efficiently. This also makes all the features contribute equally to the learning process, preventing any single feature from dominating the others.

Training

The described architectures were trained by specifying the epoch number, batch size and validation subdivision:

Epochs *Increasing* epochs the model gets to see the training data more times, potentially leading to better learning and improved performance (lower training loss). The cons is that it increases the risk of overfitting. Train time also increases. On the other side, *decreasing* epochs leads to faster training time, reduced risk of overfitting but the model may not have enough time to learn the patterns in the data well, leading to underfitting and poor performance.

Batch Size *Increasing* this parameter leads to more stable gradient estimates during training, which potentially leads to faster convergence. Using parallel processing more efficiently, leading to faster training times. On the other hand, it requires more memory, may lead to getting stuck in local minima, and might generalize less well to unseen data. *Decreasing* this parameter allows for more exploration of the loss landscape, potentially finding better solutions and better generalization but the training is slower, due to the more frequent weight update.

Validation Split *Increasing* this parameter leads to more data, of the training set, is used to evaluate the model during training, providing a more reliable estimate of its performance on unseen data. This helps to better detect overfitting. The cons are that less data is available for training, which could potentially hurt the model's overall performance if the dataset is small. A common choice is 20-30% of the training data.

The models were trained with the mean squared error (MSE) loss function and the mean absolute error (MAE) metric. The choice of MSE was motivated by the need to penalize large errors more severely, which is useful when dealing

with predictions of continuous values as in the case of robotic arm positions and orientations. The use of MSE allows the observation of mean absolute errors, making the analysis more intuitive and easy to interpret.

Results

`Model 32,16,4`:

- MSE (x, y, w, z): [$2.9282e^{-4}$, $4.3075e^{-4}$, 0.0452, 0.0172]
- MAE (x, y, w, z): [0.01347436, 0.01669511, 0.15035603, 0.08391704]
- Mean Euclidean Error (x,y): 0.0237
- Mean Angular Error (w, z): 1.992425 (degrees)

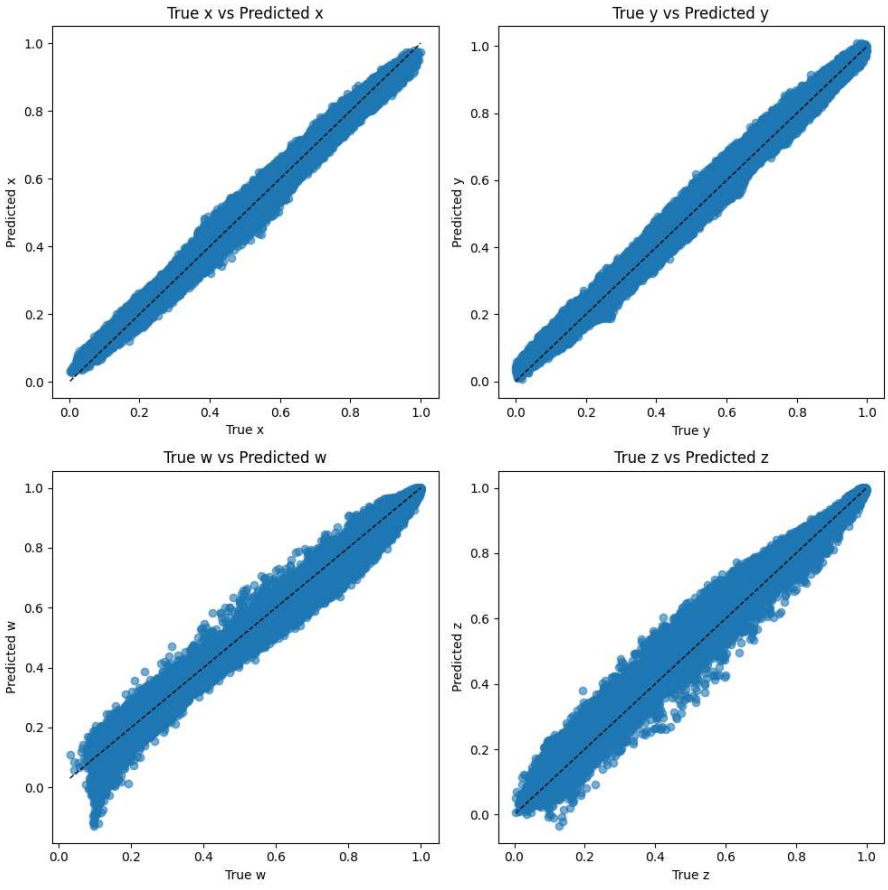

Figure 1: Model 32,16,4

`Model 64,32,16,4`:

- MSE (x, y, w, z): [$7.91954891e^{-5}$, $1.02309204e^{-4}$, $4.39072406e^{-2}$, $1.78959305e^{-2}$]
- MAE (x, y, w, z): [0.00706984, 0.00807612, 0.14752543, 0.08629945]
- Mean Euclidean Error (x, y): 0.0119
- Mean Angular Error (w, z): 0.849078

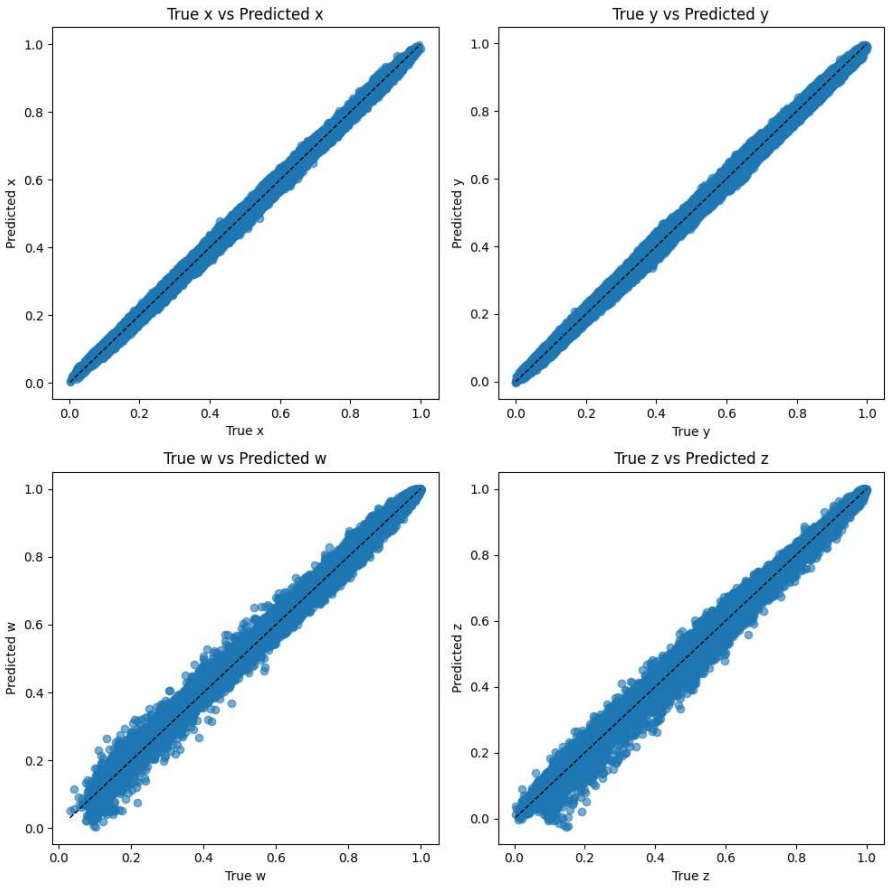

Figure 2: Model 64,32,16,4

`Model 128,64,32,16,4`:

- MSE (x, y, w, z): [$5.04603384e-05$, $4.64981485e-05$, $4.45449022e-02$, $1.78516821e-02$]
- MAE (x, y, w, z): [0.00581828 0.00557029 0.14828673 0.08612341]

- Mean Euclidean Error (x,y): 0.0088
- Mean Angular Error (w, z): 0.588519

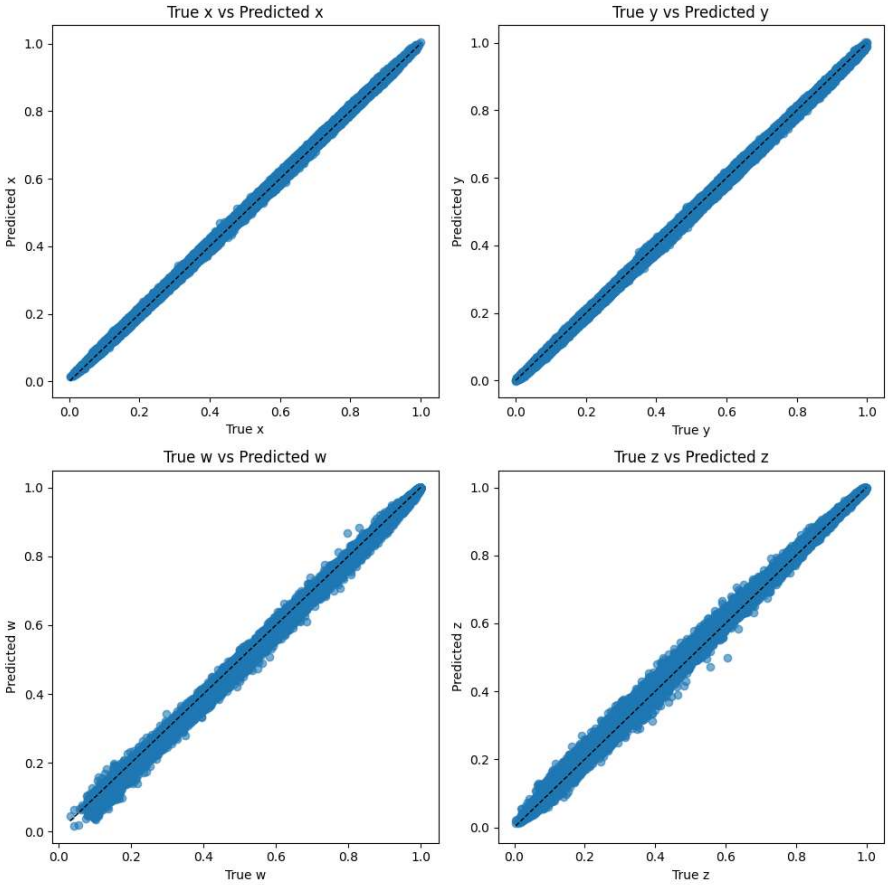

Figure 3: Model 128,64,32,16,4

Grid Search

Trial	Learning Rate	Epochs	Value
0	0.00057	11	$2.30e^{-5}$
1	0.00314	12	$6.26e^{-5}$
2	0.00758	19	$1.52e^{-4}$
3	0.00193	11	$6.08e^{-5}$
4	0.00015	12	$1.73e^{-5}$
5	0.00752	14	$1.38e^{-4}$
6	0.00056	13	$1.60e^{-5}$
7	0.00977	15	$3.85e^{-4}$
8	**0.00055**	**16**	$\mathbf{1.48e^{-5}}$
9	0.00074	19	$2.79e^{-5}$

Table 1: Grid Search results

Discussion

Using a feedforward neural network for forward kinematics provides flexibility for handling systems where the mapping between joint angles and Cartesian coordinates may be complex or noisy. However:

1. Simpler architectures may struggle to capture the high-dimensional non-linear relationships.

2. Larger architectures perform better but increase the risk of overfitting.

The **128, 64, 32, 16, 4 architecture** achieved the best overall performance, with a **Mean Euclidean Error of 0.0088** for position predictions and a **Mean Angular Error of 0.588 degrees** for orientation. Since position and orientation are two different problems and we see that in general all these models performs better on the position values, future work might consider using two different models that try to predict position and orientation respectively and exploring alternative loss functions to improve orientation predictions.