Machine Learning Homework 1

Filippo Tatafiore 1934988

2024-25

1 Datasets

This homework is composed of three forward kinematics problems for three different robots: a 2D robot with 2 joints (r2), 2D robot with 3 joints (r3) and a 3D robot with 5 joints (r5).

I generated 6 different datasets:

- r2_200_100k.csv
- r2_400_100k.csv
- r3_300_100k.csv
- r3_600_100k.csv
- r5_500_100k.csv
- r5_1000_100k.csv

For each robot there are 2 different datasets, one for training and one only for testing, generated with different seeds.

Every dataset has been saved in the drive, so that it is accesible from colab, using:

```
drive.mount('/content/drive')
```

The first step for every problem is loading the two corresponding files containing the datasets into two different pandas dataframes, *df_train* and *df_test*, and defining the features and target variables for the model.
Each problem has a different number of input features, since the robots have different number of joints. The r5 problem has additional target features with respect to r2 and r3, being in 3D instead of 2D like the others.

The first dataset is then splitted into two parts: training set and validation set. This way, we are able to evaluate the model on the validation set during the training done on the training set.
The final evaluation is done on the separate testing dataset, which is not used before.

Before training the model, the data is normalized. This way each feature has a mean of 0 and a standard deviation of 1, ensuring that all features are on the same scale. This practice makes it easier for different algorithms to process the data effectively.

All the generated datasets have 100'000 samples, but the models are trained only on 1000 samples, a randomly selected subset of the training set. The final evaluation is done on the full testing dataset.

2 R2

The datasets for the r2 problem have the following structure:

j0	j1	cos(j0)	cos(j1)	sin(j0)	sin(j1)	ft_x	ft_y	ft_qw	ft_qz
0.029	0.033	1.000	0.999	0.029	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...

Here j0, j1, cos(j0), cos(j1), sin(j0) and sin(j1) are considered the features in input for the model, while ft_x, ft_y, ft_qw and ft_qz are the target variables, for the model to predict.

For the r2 robot, I implemented two different models: Gradient Descent-based Linear Regression and Polynomial Regression.

2.1 Linear Regression

To implemet Linear Regression in tensorflow, it is necessary to define a single layer neural network. This layer uses no activation function, meaning the model remains linear with respect to the input. It applies a linear transformation to the data: $y = wx + b$, where w are the weights and b the bias.

Before the training, hyperparameter search is performed. Hyperparameter tuning is the process of selecting the best set of hyperparameters (parameters not learned from the data during training, but set before the learning process begins) for a machine learning model to improve its performance. They govern the behavior of the training algorithm or the structure of the model.
The hyperparameters chosen for the Linear Regression algorithm are learning rate and number of epochs.

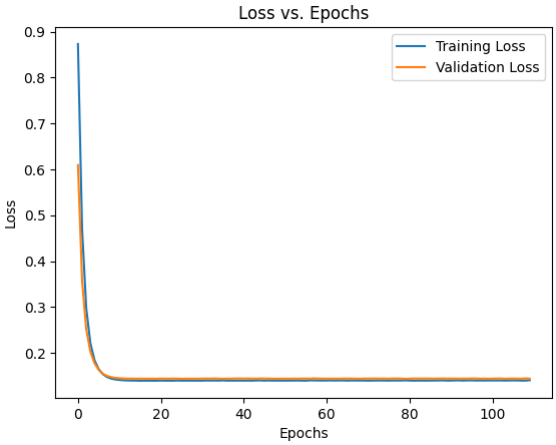
The learning rate determines the size of the steps the algorithm takes towards minimizing the loss function during the training of a model. In the context of Gradient Descent it is a scalar value that controls how much the weights of the model are updated in response to the gradient of the loss function with respect to those weights: $w = w - \eta \nabla E_n$ (η: learning rate).
If the learning rate is too small then the steps taken towards the optimal solution are very small, causing slow convergence. The algorithm may take a very long time to reach the global or local minimum. On the other hand, if the learning rate is too large, the optimization could oscillate around the minimum without converging.
This is why it is important to tune the learning rate, balancing the speed of convergence with stability.

During one epoch, the model predicts the target features of the validation set using the current weights, then computes the loss function (in this case the Mean Squared Error), and updates the weights. The number of epochs determines how many times this process is executed.
Having too few epochs results in underfitting, the model fails to generalize well.
Too many epochs may result in overfitting, where the model performs well on training data but poorly on unseen test data.

The method chosen for the hyperparameter tuning is Random Search, where a random subset of hyperparameter combinations are selected from the search space.
To perform it I used keras tuner, testing 6 different combinations.
The tuning process shows that a higher value for the number of epochs is generally preferable in this case, usually over 100, and a learning rate about 10^{-3}.

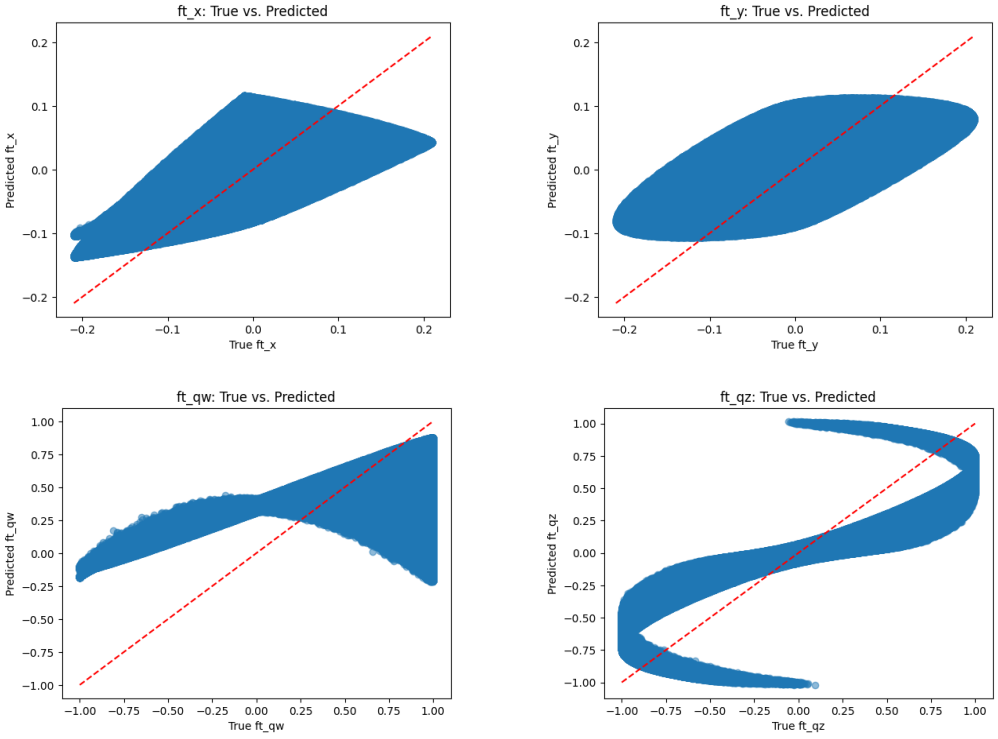
Having obtained the best values for learning rate and number of epochs, the model can be trained using these.

The following figure shows the loss value calculated during training. We can see it decreasing as the number of epochs increases.



After the training, the last step is evaluating the model on the entire 100'000 samples of the test dataset.
The evaluation is done calculating the Mean Squared Error: $\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
The MSE obtained for Linear Regression is 0.1430.

The figures below show a comparison of the model outputs to the ground truth. For a model that performs well, the points should align along the diagonal ($y = x$), but we can see here it is not the case.



In the end, I computed the Jacobian matrix of the learned forward kinematics function, considering

the input joint angles and the output cartesian coordinates.
Then I compared it to the analytical Jacobian, derived explicitly from the kinematics equations of the robot:

$$J = \begin{bmatrix} -L_1 \sin(\theta_1) - L_2 \sin(\theta_1 + \theta_2) & -L_2 \sin(\theta_1 + \theta_2) \\ L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2) & L_2 \cos(\theta_1 + \theta_2) \end{bmatrix}$$

The comparison is done calculating the elementwise difference of the two matrices and the Frobenius norm.
The output of this process for a single sample is the following:

```
input sample:    [-0.02098687 -0.03345539  1.7309977   1.6827176  -0.08761874 -0.00471709]

ANALYTICAL JACOBIAN MATRIX:

┌────────────┬────────────┐
│ 0.00754007 │ 0.00544154 │
├────────────┼────────────┤
│ 0.19983    │ 0.0998518  │
└────────────┴────────────┘

Forward Kinematics Output (FK): [ 4.2555992e-02 -1.3798475e-05]

LEARNED JACOBIAN MATRIX:
┌─────────────┬────────────┐
│ -0.0117037  │ 0.00470325 │
├─────────────┼────────────┤
│ 0.00446344  │ -0.00050724│
└─────────────┴────────────┘


COMPARISON between learned jacobian and analytical jacobian:

Frobenius norm of the difference: 0.22047864250617313

Element-wise difference:

┌────────────┬─────────────┐
│ -0.0192438 │ -0.000738288│
├────────────┼─────────────┤
│ -0.195366  │ -0.100359   │
└────────────┴─────────────┘
```

The performance of Linear Regression for forward kinematics is not particulary good, this is because Linear Regression assumes that the output is a linear combination of the input features, so it cannot capture the complex nonlinearity in forward kinematics.
In fact, in the forward kinematics problem, the relationship between the position and orientation of the end-effector of the robot and the joint angles is typically governed by trigonometric functions, which are nonlinear.
This is the reason why I also trained a Polynomial Regression model for the same r2 problem.

2.2 Polynomial Regression

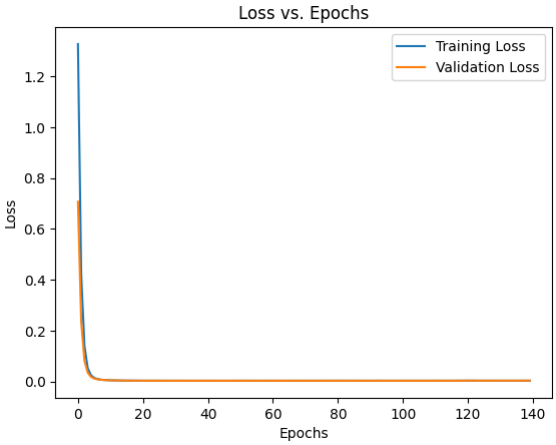
Polynomial Regression is more suitable than the Linear Regression, being non-linear. It still approximates the relationship between input and output as a fixed-degree polynomial, which may not align with the true trigonometric nature of forward kinematics.

Similarly to the Linear Regression case, to implement Polynomial Regression in tensorflow, it is necessary to define a single layer neural network with no activation function.

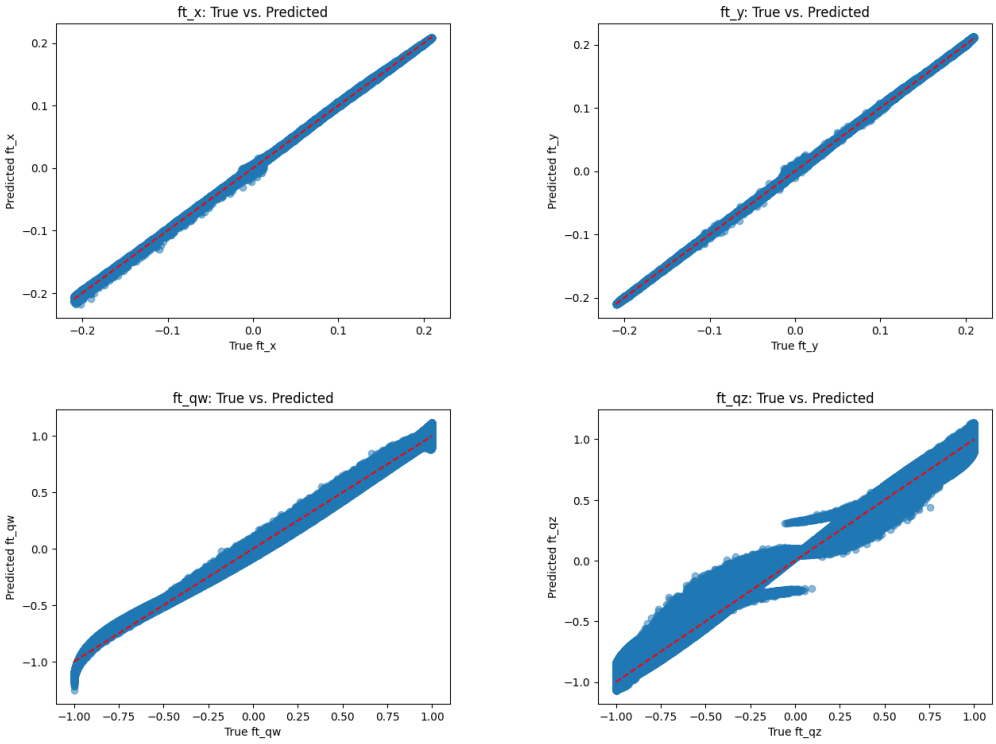
After defining the model, the next step is transform the inputs into a higher-dimensional feature space. This way the input is expanded into polynomial terms of a specified degree.

The hyperparameters chosen for this model are learning rate and number of epochs, as before, with the addition of the degree of the polynomial.
Just as before, hyperparameter tuning was executed with Random Search, testing 6 different combinations.
With the tuning we can see how the polynomial degree influences heavily the model, since having it too high causes the model to overfit on the train data, meaning it doesn't generalize well on unseen data.
The best results were obtained with a smaller polynomial degree, like 2 or 3.

With polynomial degree 2, the loss observed during training:



When evaluating the model with the separate testing dataset, the MSE obtained is 0.003337.
This shows an improvement with respect to Linear Regression. We can also see it in the following comparison of the model outputs to the ground truth.



Here we see that the points align much closer to the diagonal.

In the end, like before, I computed the Jacobian matrix of the learned forward kinematics function, considering the input joint angles and the output cartesian coordinates.
The calculation of the Jacobian has an extra step due to the fact that in polynomial regression a polynomial transformation is applied at the input features. Because of this, it is necessary to compute the derivative of the output with respect to the polynomial input, $\frac{dy}{dx\_poly}$, and the derivative of the polynomial input with respect to the original input, $\frac{dx\_poly}{dx}$.
This way we can obtain the learned Jacobian as the dot product of the two matrices:
$\frac{dy}{dx\_poly}\frac{dx\_poly}{dx} = \frac{dy}{dx}$

3 R3

The two datasets for the r3 problem have a similar structure to the r2 ones:

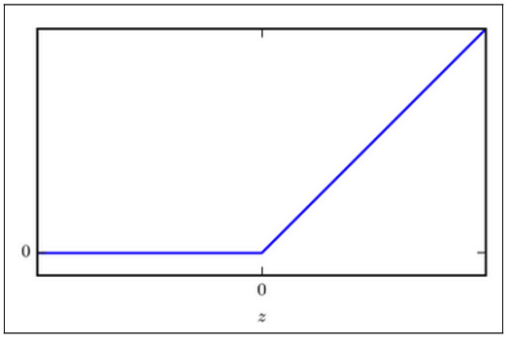
j0	j1	j2	cos(j0)	cos(j1)	cos(j2)	sin(j0)	sin(j1)	sin(j2)	ft_x	ft_y
0.034	0.012	-0.024	0.999	1.000	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...

ft_qw	ft_qz
...	...
...	...

Here we have one additional joint, so our input will have higher dimension, considering also the sine and cosine of the new joint. The output will stay the same since we are still considering a 2D space.

The chosen model to predict the forward kinematics of this robot is a shallow neural network, with a single hidden layer.
This was implemented in tensorflow, like the other models. The activation function chosen for the hidden layer, which defines neuron outputs, is the *relu* function.

$$relu(z) = max(0, z)$$



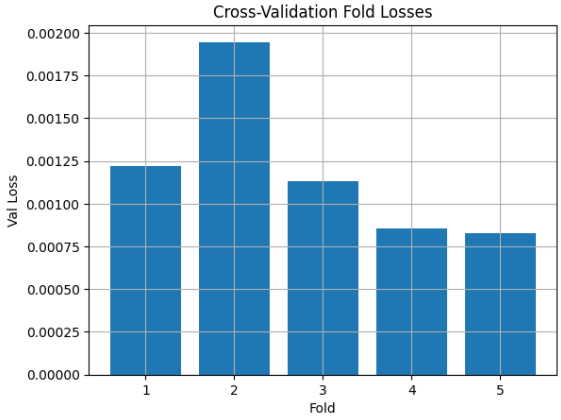
The hyperparameters for this model are learning rate, number of epochs and the number of neurons present in the single hidden layer (width).
Increasing the number of neurons increases the model's capacity, meaning it can learn more complex patterns, but entails a risk of overfitting and higher computational cost.

Hyperparameter tuning is implemented with scikeras, using the Grid Search method.
Grid Search involves defining a grid of all possible combinations of hyperparameter values, then the model is trained and validated for every combination in the grid.

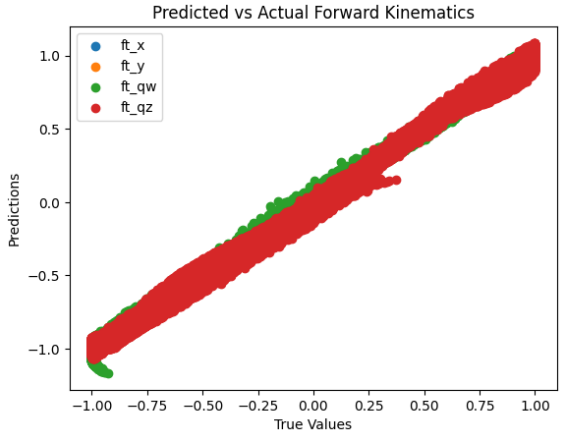
The tuning showed that a too small value for learning rate, like 10^{-5} caused worse performance, while increasing the number of neurons improved it. The best results were obtained with a higher number of epochs.

Both the tuning and the training are performed using k-Fold Cross Validation.
Cross Validation works by partitioning the dataset into k subsets (folds). For each fold, it is treated as the test set, using the remaining k-1 as the training set. The model is therefore trained on the training set and evaluated on the test set, ensuring that each fold is used once as a test set.
At the end we can calculate the average performance across all folds.

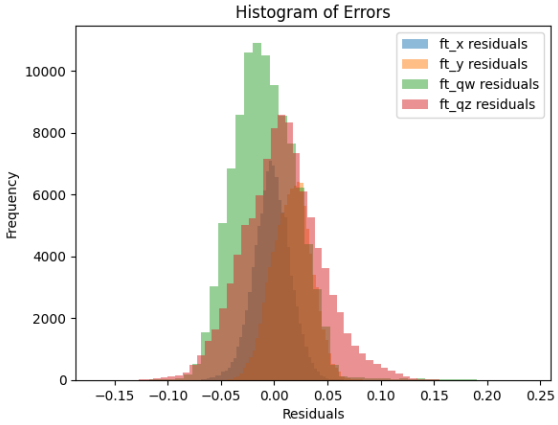
After getting the best hyperparameters with Grid Search, I built and trained the model using these, on the 1000 samples of the training dataset, then calculated the average loss on the validation dataset.
The losses obtained at each fold of the cross validation are on the following image.



At the end, the MSE calculated on the separate dataset for testing is 0.0008.
The neural network model shows a better performance rather than the previous ones. It is more suitable for the forward kinematics problem.
This shows in the comparison of the model output to the ground truth too. As we can see below, the points align closely along the diagonal, indicating a well performing model.



Follows a representation of the residuals (errors) between the actual outputs and the predicted outputs.
For a well performing model, the residuals for each feature should be centered around zero, indicating unbiased predictions, with a narrow spread, indicating a consistenly small error. It should also have the structure of a Gaussian function, symmetrical around zero, having very few residuals distant from it.



The computation of the learned Jacobian matrix is done the same way as before.

4 R5

The datasets for the r5 problem have the same structure:

j0	j1	j2	j3	j4	cos(j0)	cos(j1)	cos(j2)	cos(j3)	cos(j4)	sin(j0)
0.005	0.027	0.011	-0.007	-0.002	1.000	1.000	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...

sin(j1)	sin(j2)	sin(j3)	sin(j4)	ft_x	ft_y	ft_z	ft_qw	ft_qx	ft_qy	ft_qz
...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...

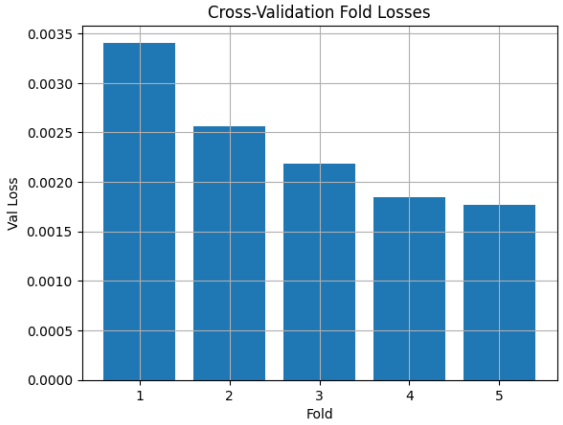
In this last problem we have 5 different joints, so the input space increases accordingly. The output dimension increases too, since now the robot is in 3D.

The model this time is a deep neural network, having multiple hidden layers. In theory a short and wide network can approximate any function, but in practice it is found that a deep and narrow network is easier to train and provides better results in generalization.
The activation function chosen for the hidden layers is *relu*.

For this model, other than learning rate and number of epochs, the hyperparameters include the number of layers and the number of neurons for each hidden layer.

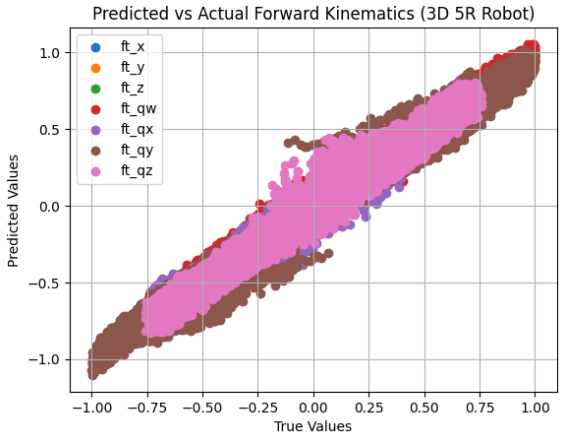
During the tuning, done again with Grid Search, it is found that is preferable a higher width (number of neurons) and number of epochs, to improve the model performance.

Follows a representation of the losses obtained at each fold of the cross validation computed during the training of the model.

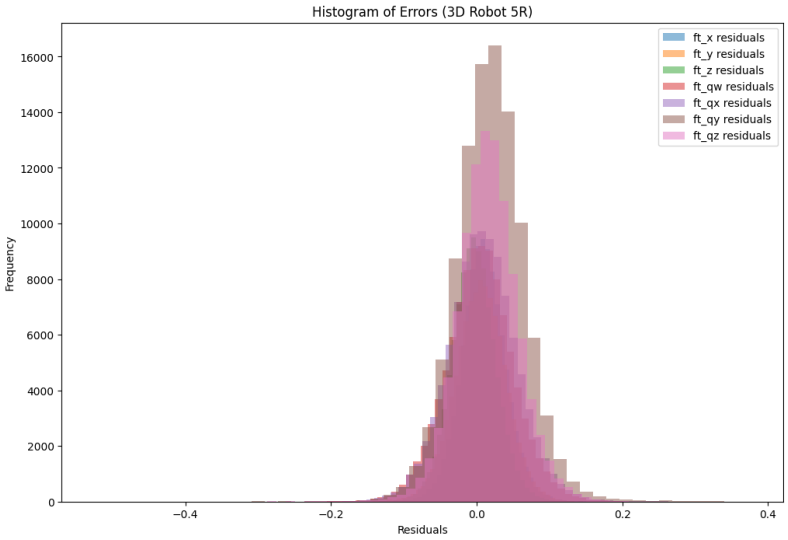


The evaluation done on the testing dataset report an MSE of 0.0018.

Once again, the comparison of the model output to the ground truth shows points aligning along the diagonal.



Then, as before, we can see a representation of the residuals showing a gaussian function centered around zero with a narrow spread.



At the end, as for the other models, I calculated the Jacobian matrix of the learned forward kinematics function.

5 Conclusion

The models adopted for the first r2 problem, Linear Regression and Polynomial Regression, are computationally less expensive than the neural networks. They are easier to implement and take less time to train, but they are also worse suited to represent the relationship between the input and output data of the forward kinematics problem, showing worse performance.

For the r3 and r5 problem, respectively are used a shallow neural network and a deep neural network.
Both these show better performance, proving the neural network to be a better suited model for this specific problem.
The shallow takes generally less time to train, since it is simpler and has less units than the deep neural network. Anyway both are more complex and take considerably more time to train than Linear and Polynomial Regression.