Machine Learning - Homework 1: Robot Kinematics

Federico Tranzocchi
Matriculation code (seed): 1891909

All the code used for plotting is AI-generated and refined.

1 Forward Kinematics

As a first task, we want to learn the *forward kinematics* of 3 given robot manipulators:

- `reacher_v6` – a 2R planar robot;
- `reacher3_v6` – a 3R planar robot;
- `marrtino_arm` – a 5R spatial robot.

We are interested in learning both the (relative) position and orientation of the end-effector of each of these robots with respect to their base frame. It is clear that training a function approximator for learning a target function is a *regression* task, and as such it is wise to rely on a loss function such as the *mean squared error*, and not on accuracy.

1.1 Data generation

Before starting, we need to collect some data to generate the datasets that will be used to train the machine learning models. We can run the robots in their respective simulated MuJoCo environment and sample some random actions from the action space; each robot pose observed during the simulation results in precious data that we can extract and collect to build our (3) datasets. Such values are:

- **input:** joint angles
- **target:** end-effector position and orientation

Notice that we need 3 datasets, one for each robot, because the features and labels vary with respect to the number of joints and the size of the task space, respectively. Summarizing:

- `reacher_v6` has $n = 2$ revolute joints/features and its task space has dimension $m = 3$, since both the planar position and orientation are of interest;
- `reacher3_v6`: $n = 3$, $m = 3$ (input: 3 joints, target: planar pose);
- `marrtino_arm`: $n = 5$, $m = 6$ (input: 5 joints, target: spatial pose).

Observation The rotation of the end-effector collected during sampling is described in quaternions, from which a minimal representation is extracted. Such a representation of orientation uses 1 angle for $m = 2$, and a sequence of 3 Euler angles for $m = 3$.

Before finally starting off, consider that the analysis for each robot is carried out considering a number of 100000 samples (to be splitted into training and test set), so as to obtain more reliable statistics. When needed (e.g. plotting, grid search, ...), this number will be accordingly reduced to a proper quantity.

1.2 Planar 2R manipulator

1.2.1 Data

The environment for the 2R planar arm counts, as anticipated, 100000 samples for which we are "given" the values for the joint angles as input, and the corresponding end-effector pose as output.

Once collected the necessary data, we can split the final dataset into a *training* and a *test* set by following the standard rule that sees $\frac{2}{3}$ of the samples allocated to the training set. We should not forget that the goal in any machine learning task is to estimate *how well the model will perform on unseen data*. In general, the employed split rule allows to build an unbiased estimator of the error where the model is trained well enough but not so as to perfectly fit the dataset; therefore, it still has sufficient generalization power.

1.2.2 Training

Forward kinematics involves nonlinear relationship between joint angles and the resulting pose; an *artificial neural network* is well-suited for modeling these relationships. At the same time, one could think that an ANN might be overkill for such a "simple" task, and thus it is of interest to see how a more humble kernelized support vector machine performs.

Artificial Neural Network The ANN used here is based on a very simple architecture. It consists of a 3-layer network where the 2 hidden layers use a *ReLU* activation function which introduces nonlinearity while remaining computationally efficient, unlike sigmoid or tanh. These two layers use 64 and 32 neurons respectively, providing a good balance between performance and risk of overfitting. The "bottleneck" ($64 \rightarrow 32$ hidden units) encourages the network to focus on more important features of the input data, helping to prevent overfitting and therefore maintaining generalization power. The total number of trainable parameters of this network is therefore **only** 2371, making it a very simple model.

The number of input units corresponds to the number of features, namely the $n = 2$ robot's joint angles. The number of output units is the number of parameters that must be learned for our task about positioning the end-effector in a plane by also accounting for its orientation; therefore, $m = 3$ units are required in output.

As mentioned at the beginning of the analysis, the chosen loss function is MSE. *Adam* is used as the optimizer with a learning rate of 0.001 so as to ensure convergence.

The training process involves 50 epochs and a batch size of 32 samples.

Kernelized SVM for regression The very first thing to notice is that actually SVR can only learn one value. This means that we need 3 models, one for each target (x, y, $theta$), whose output will be combined after fitting.

Now, setting the hyperparameters for these models may be quite tough of a task, seen the fair number of them. In order to find the best ones, it is advisable to carry out some *hyperparameter search*. This is one of those cases in which a reduced number of samples makes sense. Carrying out a thorough grid search is computationally expensive, and for this reason, only 1000 samples will be considered for this model. The parameters of interest are:

- (the inverse of) the regularization factor C;
- the type of kernel used among linear, poly, and sigmoid (notice that RBF has been intentionally cut out for analysis purposes, that is to mainly focus on kernels of the polynomial family);
- the degree $(1, 2, 3)$ to be used in case of polynomial kernel.

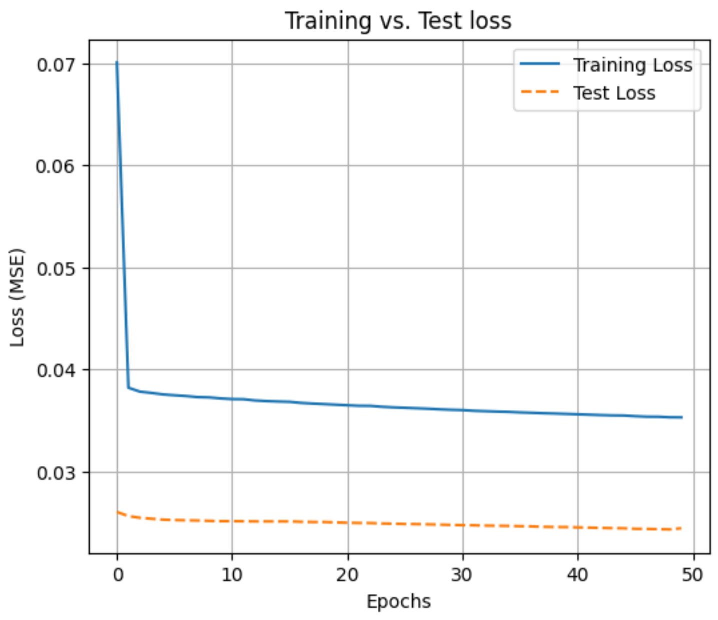

Figure 1: Comparison between training and test loss of the neural network used for the 2R planar arm.

1.2.3 Evaluation

Artificial Neural Network The training loss in Fig. 1 shows the typical shape that a "healthy" loss should have.

Actually, the model takes just the first epoch to train and converge, and this can be seen by looking at the test loss in yellow, but the learning rate (0.001) prevents the network from overfitting, even though it keeps training for 49 more epochs.

Observation In this case, even an untrained NN would perform better than multiple kernelized SVMs.

We can also confirm from other metrics how the model was very effective in this regression task:

- Mean Absolute Error = 0.0128
- R2-Score = 0.9918

It looks like the network manages to predict the direct kinematics of the robot correctly. Indeed, by plotting the true end-effector pose against the predicted one (see Fig. 2), it is clear that overall the output of the model and the target output are not that far from each other. Notice how sometimes the predicted position is "shifted" out of the workspace, which means that that position is not even actually reachable by the arm, resulting in an unacceptable behavior when it comes to a real manipulator. Nevertheless, the position and especially orientation of the model's outputs look mostly correct.

Observation Only around 3000 samples have been used for the plot in Fig. 2 for a better clarity of results. Nevertheless, the whole $\frac{1}{3}$ of all samples was used for testing.

Finally, we can compare the performance of the NN we have used so far by computing its derivative and comparing this **learned Jacobian** with the true one for a 2R planar arm. For the

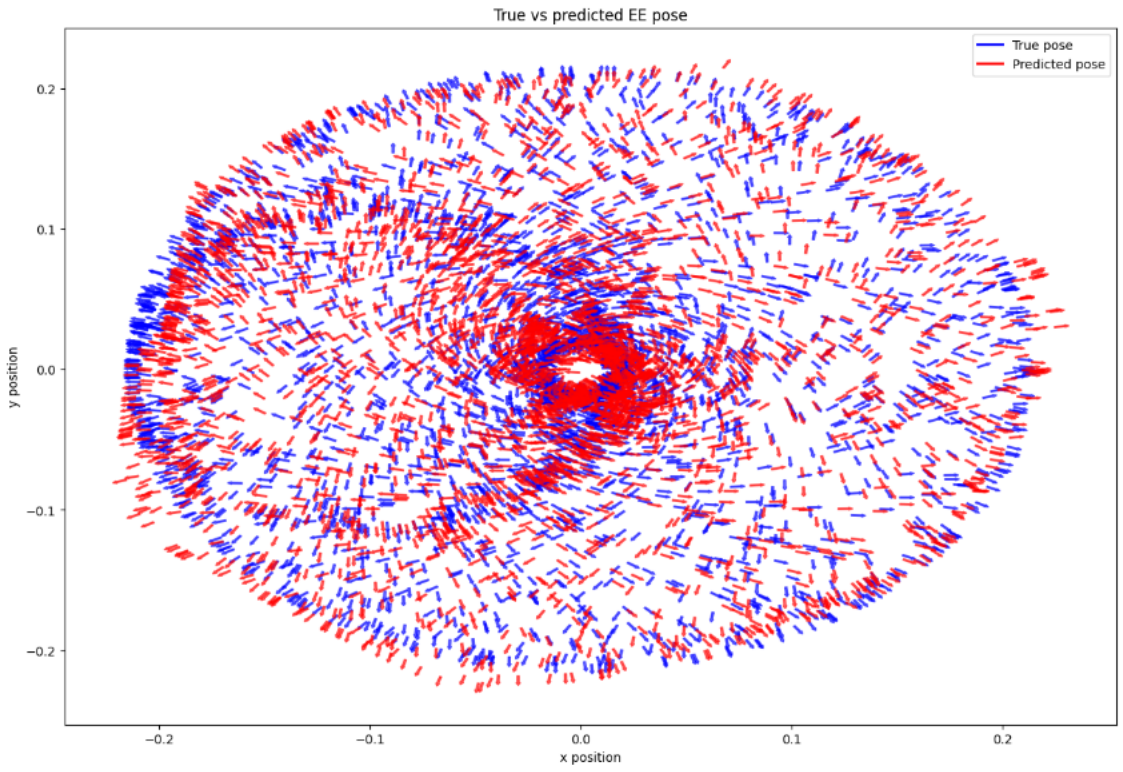

Figure 2: Predicted forward kinematics (red) vs. true forward kinematics (blue) in the `r2` environment.

configuration $\mathbf{q} = [0.5, 0.2]^T$, the two Jacobians are:

$$J_{\text{true}} = \begin{pmatrix} -0.1123 & -0.0644 \\ 0.1642 & 0.0765 \\ 1 & 1 \end{pmatrix} \qquad J_{\text{model}} = \begin{pmatrix} -0.1331 & -0.0607 \\ 0.1801 & 0.1144 \\ 0.9887 & 0.9671 \end{pmatrix}$$

leading to an error $E = ||J_{\text{true}}(\mathbf{q}) - J_{\text{model}}(\mathbf{q})|| = 0.0578$, which is a great result putting this model close to a real robot that uses true direct kinematics.

Kernelized SVM for regression The outcome of the grid search highlights how actually training the above neural network was more efficent both in terms of time, and thus resources, and performance. In fact, it took roughly 5 minutes for the hyperparameters to be tuned with only 1000 samples, and the R2-scores of the SVR for the x- and y-position show very poor performance (0.238 and 0.155, respectively). This means that with less time we can instead train an NN over 100 times the number of samples and even manage to perform definitely better.

The found hyperparameters for each model are the following:

- `svr_x`: `C`$= 10$, `degree`$= 2$, `kernel=’poly’`
- `svr_y`: `C`$= 100$, `degree`$= 3$, `kernel=’poly’`
- `svr_theta`: `C`$= 1$, `degree`$= 1$, `kernel=’poly’`

By plotting the results of the evaluation, we can see that the orientation has been predicted well once again on both joints. Actually, the first model predicted the x-position of the end-effector not too bad (see Fig. 3), but this case is about a robot manipulator which would ideally be mounted in a factory where precision is crucial, and thus I would not definitely employ SVR

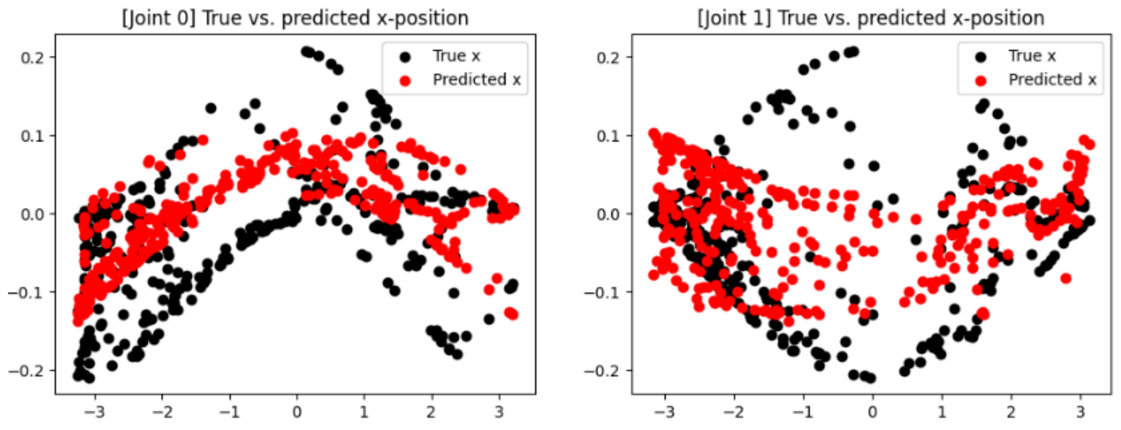

Figure 3: Predicted (red) vs. true (black) x-position of the end-effector in `r2` when using SVR.

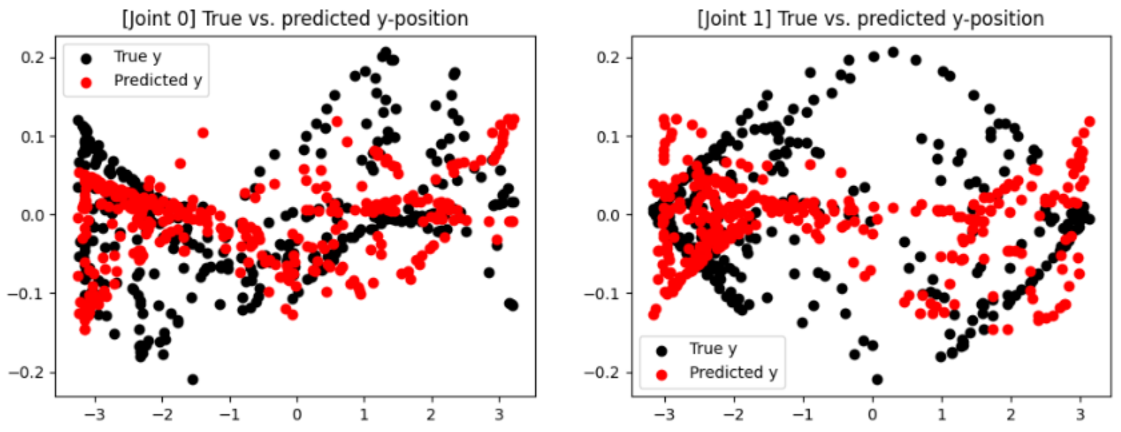

Figure 4: Predicted (red) vs. true (black) y-position in `r2` with `svr_y`.

in real-world conditions because the end-effector would completely be off position most times, despite being correctly oriented.

As anticipated, Fig. 4 shows how the predicted y-position is definitely more off and this may be due to the fact that we only searched among degrees 1, 2 and 3 for kernels of the polynomial family. We do not report the plot for the predicted orientation here, since `svr_theta` got an R2-score $\approx$ 1.

Personal consideration From this moment forth, I decided to proceed with only different architectures of neural networks, because other models do not look like possible real applications of machine learning for manipulators (to my knowledge). I wonder, when a closed-form solution for inverse kinematics cannot be found, or even just to retrieve direct kinematics in a lazy way, whether regression can be actually used or not and replace analytical solutions, knowing that the true function will never be perfectly learned (indeed, to learn = to estimate).

1.3 Planar 3R manipulator

1.3.1 Data

The number of samples 100000 and the training split strategy remain unchanged. This is intentional, as the model that we are going to use is a bit more complex, and an increase in training time is expected and wanted for this analysis.

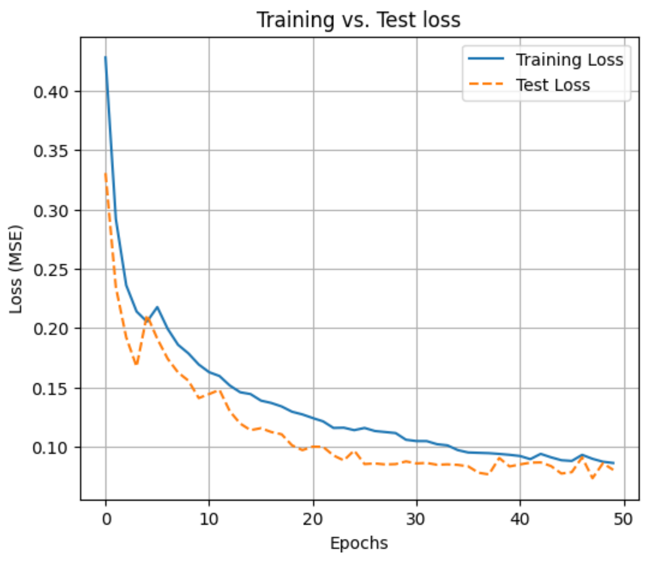

Figure 5: Training and test loss of `r3`.

1.3.2 Training

The output shape does not change because the task is still planar ; though, the input data consists now of 3 joint angles. The new architecture of the neural network involves 3 hidden layers with 64, 64, and 32 units, respectively. With respect to the previous model, the additional layer will provide some more parameters that can be trained, **theoretically** leading to a better estimation of the forward kinematics.; in fact, the total number of trainable parameters is 6595 almost 3 times more than those of `r2`. Despite we are not using any technique to address overfitting (such as dropout), we must never forget to account for it when building a neural network $\implies$ we do not want to exaggerate with the number of hidden units.

The last layer uses half of the units of the previous layers for the same reason as before, so as to let it focus more on important features and, in fact, helping to prevent overfitting. Again, we want to freeze the number of epochs and the size of the batch for all 3 environments to make a comparison out of this.

1.3.3 Evaluation

It took about 4 minutes to train the network, whose final training loss is shown in Fig. 5.

Making predictions with the trained model using the samples coming from the test set yields these results:

- Mean Absolute Error = 0.0425;
- R2-Score = 0.9762

It is pointless to plot the predicted vs. true pose of the end-effector once again, since it shares the same "problem" about points that fall outside of the robot's workspace with the previous model's plot.

The true and learned Jacobian, for instance for the robot configuration $\mathbf{q} = \begin{bmatrix}0.5 & 0.2 & 0.3\end{bmatrix}^T$, are

$$J_{\text{true}} = \begin{pmatrix} -0.1965 & -0.1486 & -0.0841 \\ 0.2183 & 0.1305 & 0.0540 \\ 1 & 1 & 1 \end{pmatrix} \qquad J_{\text{model}} = \begin{pmatrix} -0.1490 & 0.0071 & -0.1225 \\ 0.2844 & 0.1907 & 0.0048 \\ 0.9808 & 1.1226 & 0.9840 \end{pmatrix}$$

and they differ by an error $E = ||J_{\text{true}}(\mathbf{q}) - J_{\text{model}}(\mathbf{q})|| = 0.2324$.

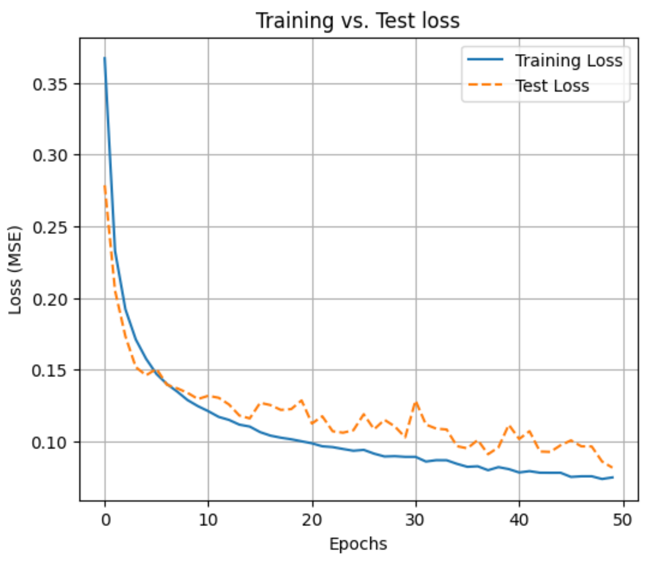

Figure 6: Training and test loss regarding `r5`.

1.4 Spatial 5R manipulator

1.4.1 Data

The final and most complex multi-target regression task involves 5 features and 6 target values, so we may expect more difficulties by the model when predicting unseen data. In order to face this last dataset, a neural network with 4 hidden layers consisting of 128, 128, 64 and 32 units, and they all employ a ReLU activation function once again.

1.4.2 Training

The used architecture leads to 27814 trainable parameters, which compared to the previous cases is a lot more: the model should be complex enough to learn the forward kinematics of the spatial robot. Training took around 4 minutes.

1.4.3 Evaluation

Fig. 6 marks that this time the test loss is constantly above the training loss, and thus worse. This means that the network reaches a certain performance on the training set that it will never achieve when predicting on unseen data. Still, the two losses are not that far from each other.

Fig. 7 highlights for the last time how the predicted orientation of the end-effector is averagely correct, while this latter is basically always shifted by a bit from its real position in space Notice that only 100 samples were plotted in Fig. 7 for a better understanding.

1.5 Hyperparameter search

Given each of the above discussed environments (`r2`, `r3`, `r5`) it is of interest to perform a hyperparameter search over different optimizers and learning rate, to see which one fits better what environment.

We have seen that `r2` immediately converges with Adam and relatively low learning rate (0.001), so we want to know what is the behavior of the model when using other (weaker) optimizers. On the other hand, `r3` and `r5` might have improved a little bit more.

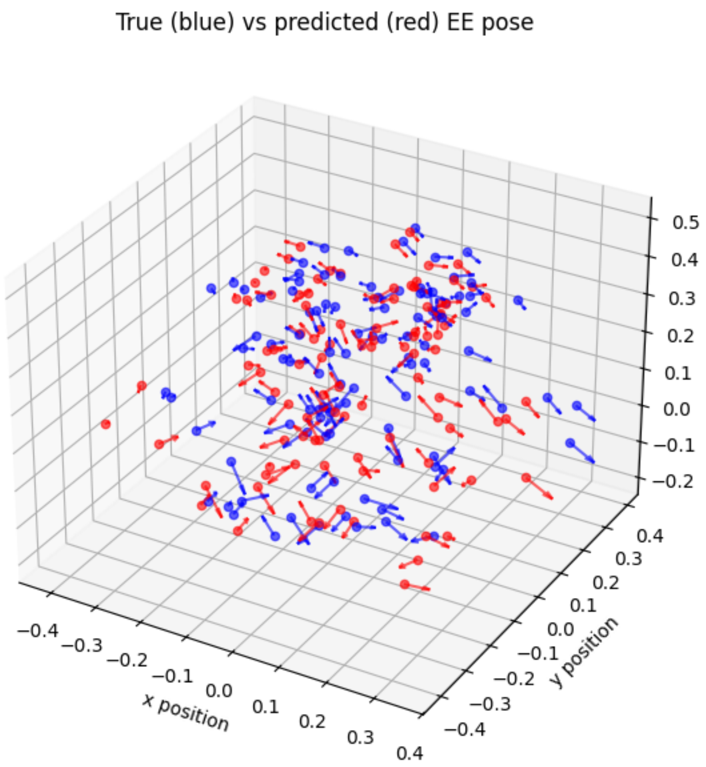

Figure 7: Predicted pose (red) and true pose (blue) of `r5`.

Since running for more epochs would require much time, we are going to train the models in just 10 epochs: they should be enough to understand the behavior of the losses. Finally, we consider only 10000 samples for simplicity.

The settings of the search are reported below:

- Optimizers: SGD, RMSprop, Adam
- Learning rates: 0.001, 0.005, 0.01

The whole process took only 3 minutes. Fig. 8 depicts the training and test losses of `r2` at different learning rates. We can see how, in general, SGD is the optimizer that starts worse. In the end though, all optimizers converge to the same value, which is around 0.350 for the train

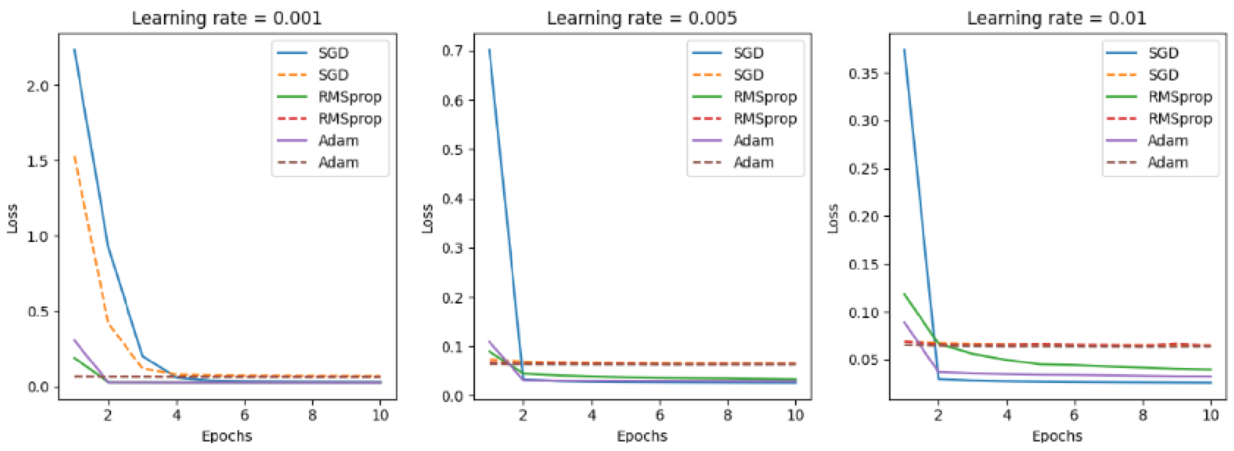

Figure 8: Training and test losses for environment `r2`.

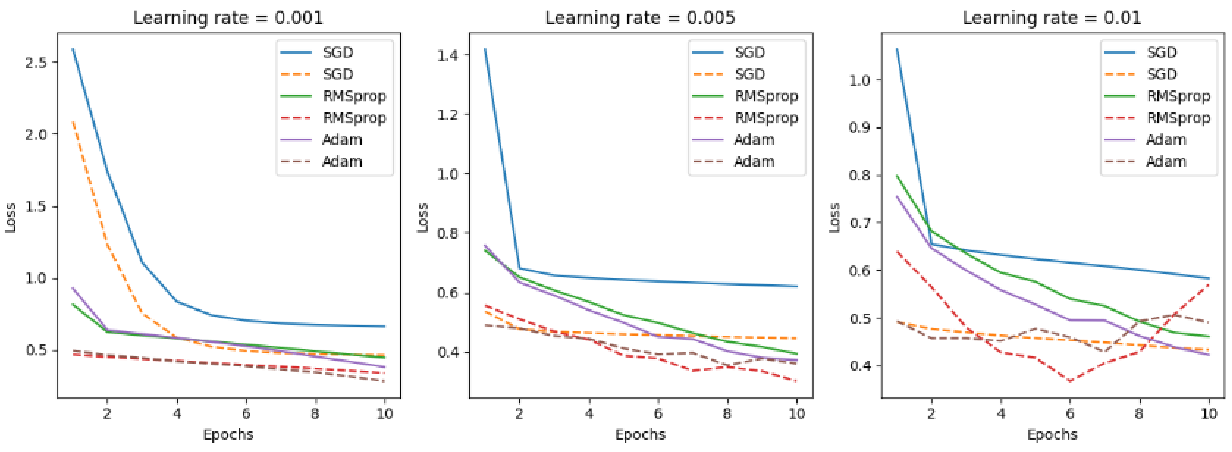

Figure 9: Training and test losses for environment r3.

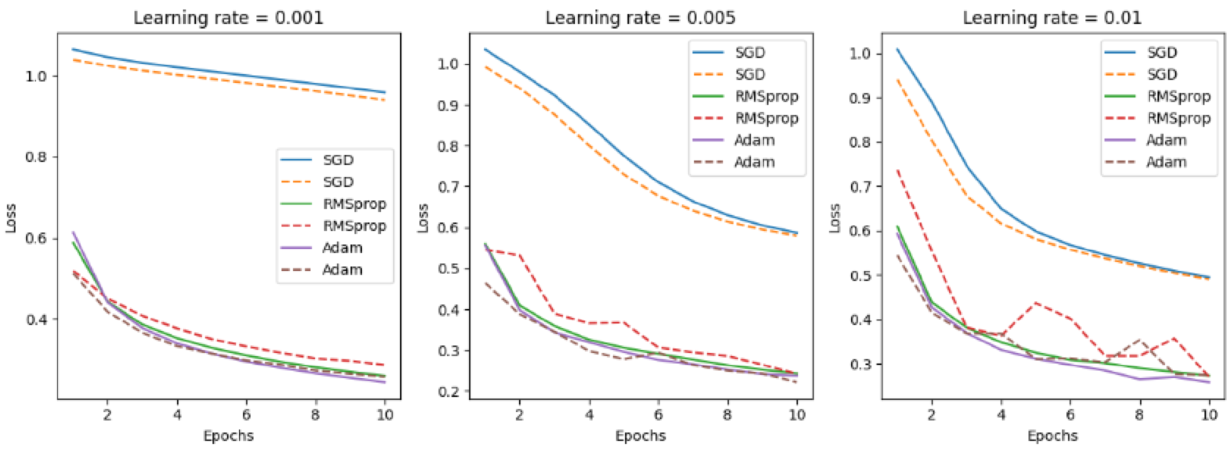

Figure 10: Training and test losses for environment r5.

loss and 0.650 for the test loss. It must be noticed how the test loss is basically always above the train loss for every optimizer and learning rate.

Moving to r3, it is clear from Fig. 9 that SGD is not only the worse at the beginning, but also at convergence. When the learning rate is not too high (< 0.01), Adam and RMSprop are almost equivalent and the best choice among the three. When the step size increases to 0.01, it is very curious how SGD seriously underperforms during training but at the end wins against the other two optimizers: Adam's test loss just increases and goes away from convergence, while RMSprop seems doing great at the beginning but eventually overshoot after epoch 6, leading itself to be the worst choice.

In contrast to r2, the test loss is on average lower than the training loss for r3. Both the best training loss (0.3404) and test loss (0.2819) are achieved by the configuration using Adam with step size 0.001, that is exactly the one we have used for our analysis above.

Finally, the losses for the last environment r5 are shown in Fig. 10 where it is even easier to recognize how SGD gets outperformed by the rest. Let us analyze the plots for each learning rate:

- $0.001 \implies$ RMSprop and Adam are (almost) equivalent; SGD is on its own path very far from the other two.

- $0.005 \implies$ SGD starts getting closer to Adam and RMSprop, but they are still the best choice.

- $0.01 \implies$ RMSprop oscillates quite a lot and it is not reliable anymore; Adam had just a single spike which could indicate a worse behavior over time; SGD is seriously getting better and promising for greater numbers of epochs.

We can not exclude the fact that SGD might reach or even surpass the performance of RMSprop and Adam when considering longer periods of training, especially for higher learning rates. In fact, it looks like these latters struggle in general when the learning rate approaches values greater than 0.005 where they start overshooting/oscillating, whereas SGD starts taking better values.

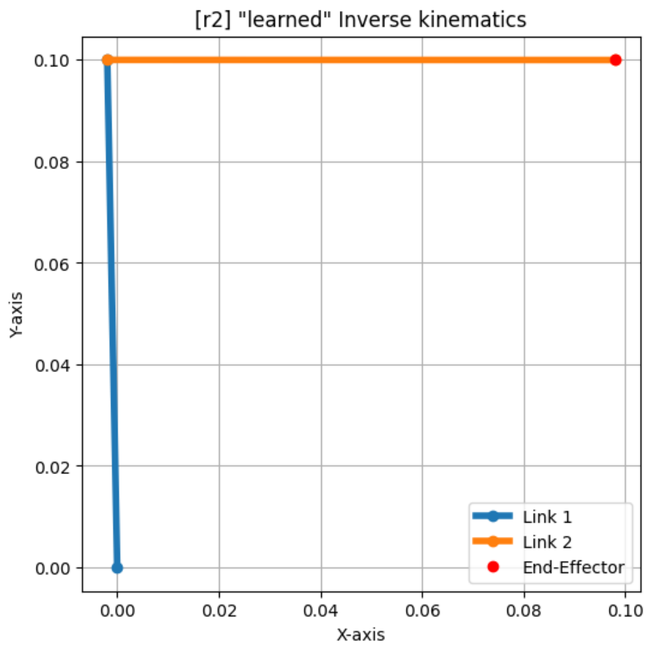

Figure 11: A first tentative of inverse kinematics with `r2`.

2 Inverse Kinematics

We want now to exploit the learned Jacobian matrices of `r2` and `r3` to solve the *inverse kinematics* problem for the related robots.

The first thing to do is to load the models that were trained before so that we can compute the forward kinematics and the Jacobians J_{r2} and J_{r3} as needed.

The *Levenberg-Marquardt* algorithm is perfect for problems regarding curve fitting. It uses a damping factor $\lambda \geq 0$ adjusted at each iteration finding the trade-off between Gauss-Newton and gradient descent. Here, we start with an initial $\lambda = 0.1$ that is either multiplied or divided by a factor 10 based on the next step error.

We will try to solve the IK problem for some "random" desired poses (given that they are in the robot's workspace). To this end, we must remember that the length of all links is 0.1 for both robots. Whatever desired poses we are considering, we are giving the algorithm the same initial guess $\mathbf{q} = 0$ so as to obtain more realistic results, since basically every robot has a home configuration which can usually be either $\mathbf{q} = 0$ (maybe if mounted on a table or on the floor) or another one.

Observation An error < 0.01 will be considered as convergence.

2.1 2R planar arm

Random (reachable) point in the plane Let us first consider the target pose $\begin{bmatrix} 0.1 & 0.1 & 0.0 \end{bmatrix}^T$. The final "learned" inverse kinematics for such a desired vector of values is depicted in Fig. 11. Link 2 is perfectly aligned with the x-axis, but Link 1 is a bit off trajectory and this leads the end-effector to be slightly retracted from the desired position. As for the orientation, it was perfectly achieved.

Retracted arm While carrying out the analysis on `r2`, we did not mentioned anything about the hole in the final plot of end-effector poses. That hole means that an action that brought

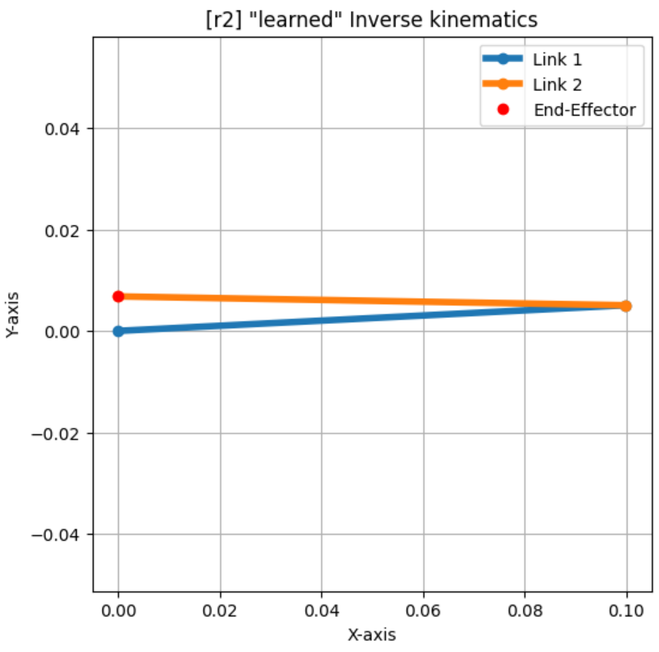

Figure 12: Inverse kinematics of `r2` for fully retracted arm.

the end-effector in the origin (or near there) has never been sampled during the generation of the dataset. At the same time, `r2` has been trained on **that** dataset, and thus it does not know anything about such a position. Fig. 12 highlights this fact, where `r2` was not able to bend Link 2 over Link 1 completely in order to reach the origin with orientation π.

Out-stretched arm We have seen that **all** above models did not really have any problem with orientation; instead, they kept predicting shifted positions of the end-effector, leading to planar coordinates that were not achievable in any way by the end-effector. Now, we can finally test this out and see whether it is a real problem or not. By feeding the inverse kinematics function with the configuration of joint angles corresponding to the out-stretched arm, i.e. $\mathbf{q} = \begin{bmatrix} 0.2 & 0 & 0 \end{bmatrix}^T$, the LM algorithm converges to an error$= 0.01444$ which never gets smaller than the tolerance threshold, meaning the returned solution (see Fig. 13) is discarded. We can see that the robot is stuck in a position where it is probably trying to reach a point which is outside of its primary workspace. Indeed, by plotting the "current pose" (which the robot thinks to be at) we obtain an x-position > 0.21, which will never be possibly achieved since $l_1 + l_2 = 0.2$.

2.2 3R planar arm

Out-stretched arm We remember from the training of `r3` that model's predictions were very good but not optimal, just a little bit more off than those of `r2`. As mentioned in the observations done throughout this study, robot kinematics is a strict and precise task. The easiest way to show this gap is to try computing the joint angles for reaching the point at $(0.3, 0.0)$ with orientation 0: a very simple task, specifically the same as above, but that we already noticed being problematic. In this case, the outcome of such a computation is reported in Fig. 14. The arm is definitely too much retracted and in a real-world scenario it would not be able to grasp a hypothetical object with the maximum precision.

Random point in the plane When the robot tries to reach the point $(0.25, 0.15)$ in the plane with orientation $\pi/4$, it misses the goal by just a little bit (Fig. 15). This amount of error is

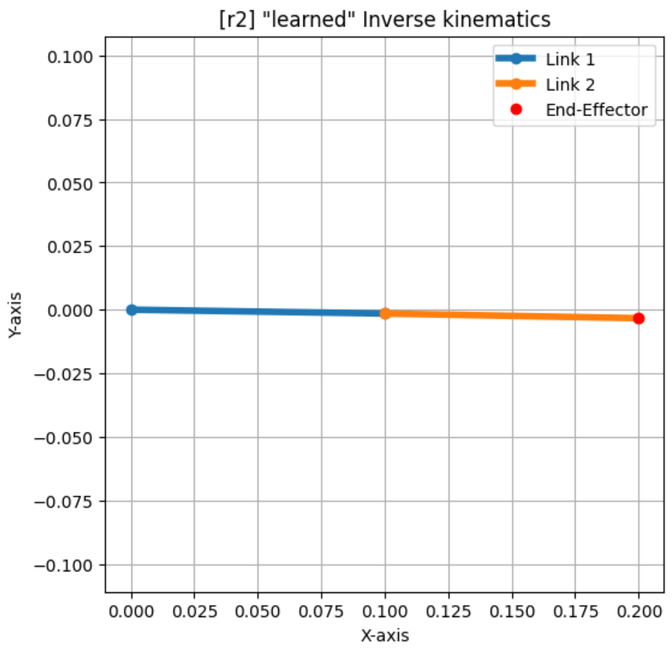

Figure 13: Inverse kinematics of `r2` for outstretched arm.

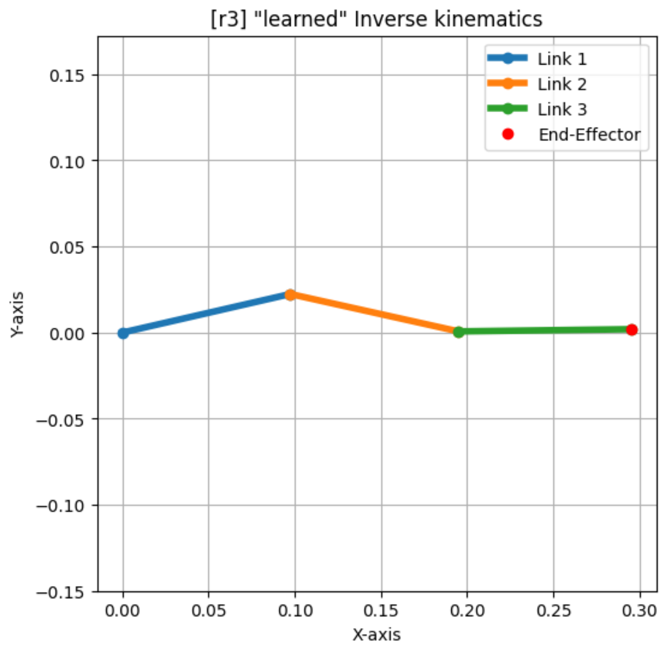

Figure 14: Inverse kinematics of `r3` for outstretched arm.

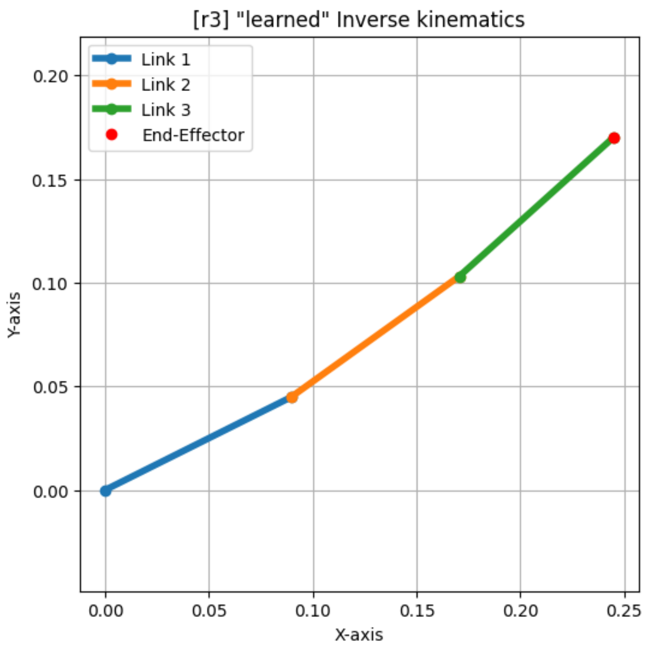

Figure 15: Inverse kinematics of `r3` for a random reachable point.

acceptable in a simulated environment, but not in industry or wherever the robot is going to be mounted.

2.3 Results

We have seen how the two models' output are not too far from the desired solutons, and with just a little bit more tuning the end-effector could be correctly placed in the target pose. This certainly sounds like a control problem, which will be addressed in the next section.

3 PID Controller (2R only)

We are going to employ a simple *PID controller* to get rid of the small errors we have been bumping into. Precisely, we need one controller for each joint so as to regulate the torque applied to them, but we are actually going to set the corresponding parameters of both joints to the same value. After some tuning, these are the final parameters of the two ontrollers:

- $K_p = 3$
- $K_i = 0.05$
- $K_d = 0.12$

This choice leads to a smooth and fast enough movement but that does not overshoot or oscillates. In total, 6 different end-effector poses were tried to be achieved with the aid of the PID's action, and for each of them there is a related video. Below follows the values of x, y, ϑ for the target poses, with their respective outcome:

- Pose 1: $(x = 0.1, y = 0.1, \vartheta = 0) \implies$ The robot managed to converge to the target pose in a few iterations, with no overshoot or any oscillation.
- Pose 2: $(x = 0.1, y = 0.1, \vartheta = \pi/2) \implies$ The robot presented a similar behavior to Pose 1.
- Pose 3: $(x = 0.2, y = 0.0, \vartheta = 0) \implies$ The robot was already close to the target pose because the initial guess for the joint angles is (0.0), and thus it just had to fix a small offset.
- Pose 4: $(x = 0.0, y = 0.2, \vartheta = \pi/2) \implies$ The robot acted in a very similar fashion to Pose 1 and Pose 2, reaching the point with no hesitation.
- Pose 5: $(x = 0.1, y = 0.0, \vartheta = \pi/4) \implies$ The robot did not manage to achieve the desired pose, most likely because of the constraint given by the orientation. (2R).
- Pose 6: $(x = 0.0, y = 0.0, \vartheta = \pi) \implies$ The robot seems not able to reach the fully retracted configuration with these parameters. For further details, read the observation in the orange rectangle below.

Personal observation I manually tried to tune the parameters of the controller with different target poses, and for some of them I was not able to find the perfect solution. The movement is always smooth and controlled, but there might be some steady-state error that I did not manage to remove even by tuning the incremental action. Specificalyl, in order to reach some specific poses, I had to increase the proportional action which basically always led to a very uncontrolled, unstable and unreliable movement from the robot side. I tried to increase the derivative action with no success. Finally, I thought that, in a real-world scenario, such a fast and dangerous movement would not have been correct, and thus I decided to leave the parameters the way they were so that the robot's end-effector can reach most poses correctly, some others with just a little offset, but in any case with a "pleasant" transition.