A Lagrange Dual Learning Framework for Solving Constrained Inverse Kinematics Tasks

Milo Knowles
mknowles@mit.edu

1 Introduction

Inverse kinematics (IK) is a fundamental problem for robotics and computer graphics. In the context of robotic manipulation, the goal of IK is to solve for a set of joint angles that would place the end-effector of a robotic arm at a given target pose. Oftentimes, there are also physical or mechanical constraints that must be respected, such as joint angle limits or obstacles in 2D or 3D space that the robot must avoid. Fast and accurate algorithms are needed to repeatedly solve IK problems with physical and safety-related constraints in real-time [1] .

However, solving high-dimensional inverse kinematics problems with constraints remains a challenging non-linear and non-convex task. Part of the challenge of finding optimal solutions to inverse kinematics tasks is that the problem may be under-constrained; many feasible configurations of joint angles can produce the same end-effector pose. For example, theoretical results have shown that a kinematic chain in 3D with 6 degrees-of-freedom may have up to 16 joint angle configurations that achieve the same end-effector pose [1].

Although analytical solutions exist for small kinematic chains (i.e 2 or 3 joints), they are difficult to derive for complex robotic manipulators with many degrees-of-freedom [2]. For these larger problems, numerical methods are commonly used, such as the Jacobian inverse method, cyclical coordinate descent, and the Forward and Backward Reachable Inverse Kinematics (FABRIK) method [3]. While effective for some manipulation problems, these approaches can be slow to compute, and have difficulty in incorporating problem constraints [2]. For example, the Jacobian inverse method, while widely used, may require many random initializations to converge to a feasible solution [2].

Recent work has shown that neural networks can provide fast approximations to challenging constrained optimization problems through a Lagrangian dual learning framework [4] [5]. This iterative approach interleaves two steps: training a neural network to approximate a Lagrangian relaxation of the constrained optimization problem, and updating the Lagrange multipliers to obtain a stronger relaxation. In this project, we demonstrate that this framework can be applied to constrained IK problems, producing learned models that can provide fast, approximate solutions while respecting problem constraints.

2 Background

2.1 Forward and Inverse Kinematics

We denote the joint angles of a serial-chain robotic manipulator with J joints as $\theta \triangleq (\theta^1, \theta^2, ..., \theta^J)$. Given this set of joint angles, the position and orientation (pose) of the end-effector can be easily computed using the *forward kinematics*:

Project for 6.881 Optimization for Machine Learning, Spring 2020.

$$T^{ee} = f(\theta) \tag{1}$$

where the pose T^{ee} is a transform containing a rotation R^{ee} and translation t^{ee}.

In general, the forward kinematics function also returns the position and orientation of all of the other joints $\{T^j \triangleq (R^j, t^j)\}_{j=1}^{J}$, although we are typically interested in the pose of the end-effector.

The goal of *inverse kinematics* is to find a vector of joint angles θ such T^{ee} is equal to the desired end-effector configuration T^{goal}:

$$\theta = f^{-1}(T^{goal}) \tag{2}$$

The desired configuration T^{goal} may be only a target position t^{ee}, or may include a target orientation r^{ee} as well. In this work, we will consider a target position only, although the proposed framework could be extended to include orientation as well.

2.2 Optimization Constraints

In addition, there may be explicit or implicit constraints over the solution space of θ. The most common constraints are *joint limits*, which are explicit bounds on the angles that the robot joints can be actuated to. Formally, joint limits can be expressed as:

$$\theta^j_{min} \leq \theta^j \leq \theta^j_{max} \quad 1 \leq j \leq J \tag{3}$$

Many IK solvers can also incorporate spatial constraints, such as obstacles in the workspace that the manipulator must not collide with, or "fences" that all joints must stay within. In this project, we will focus on obstacle avoidance constraints. In the most general form, constraining all joints to avoid obstacle S can be written as:

$$t^j \notin S \quad \forall j \in \{1, 2, ..., J\} \tag{4}$$

$$S \subset \mathbb{R}^d \tag{5}$$

where $d = 2$ for a 2D planar manipulator, $d = 3$ for a 3D manipulator, and S is a closed subset of $\mathbb{R}^d$.

2.3 The Lagrange Dual Learning Framework

In this section, we will briefly summarize the Lagrange Dual Learning (LDL) framework used in [4] and [5] before extending it to the inverse kinematics problem.

This framework considers optimization problems of the form:

$$\mathcal{O} = \arg\min_y f(y) \tag{6}$$

$$s.t. \quad g_i(y) \leq 0 \quad \forall i \in 1, ..., m \tag{7}$$

where each $g_i(y)$ is a function that quantifies the amount that the constraints are violated. In [4], these g_i can be negative, indicating that a constraint is "over-satisfied".

The Lagrangian objective function can be written as:

$$f_\lambda(y) = f(y) + \sum_{i=1}^{m} \lambda_i g_i(y) \tag{8}$$

When using the Lagrangian objective function, we are solving a *relaxation* of the original optimization problem:

$$\tilde{\mathcal{O}}_\lambda = \arg\min_y \{f_\lambda(y)\} \tag{9}$$

The goal of the Lagrangian dual optimization is to obtain the **strongest** such relaxation:

$$\mathcal{D} = \arg\max_{\lambda \geq 0} \{ \arg\min_y f_\lambda(y)\} \tag{10}$$

The insight in [4] and [5] is that a neural network can learn a parametric approximation of the inner optimization. To do this, the LDL framework of [4] alternates between training a neural network model on a Lagrangian relaxation objective for some *fixed* multipliers λ, and then updating the multipliers using a sub-gradient method to obtain a stronger relaxation.

Formally, a neural network model $\mathcal{M}[w]$ with learnable weights w is trained using a dataset of solved problem instances $\{(d_l, y_l = \mathcal{O}(d_l))\}_{l=1}^n$, consisting of inputs d and solutions y from an oracle.

The Lagrangian loss function for a particular relaxation (with fixed λ) is:

$$\mathcal{L}_\lambda(\tilde{y}_l, y_l, d_l) \triangleq \mathcal{L}(\tilde{y}_l, y_l, d_l) + \lambda^T g(\tilde{y}_l, y_l, d_l) \tag{11}$$

where $\tilde{y}$ are the network's predicted solutions, d_l are the inputs for a given problem instance, and y is the desired solution (obtained from an oracle that supervises the learning).

The training process can then be written as:

$$w^*(\lambda) = \arg\min_w \sum_{l=1}^{n} \mathcal{L}_{\lambda^k}(\mathcal{M}[w](d_l), y_l, d_l) \tag{12}$$

which produces an approximation $\tilde{\mathcal{O}}_\lambda = \mathcal{M}[w^*(\lambda)]$.

Next, the multipliers are updated as follows:

$$\lambda_i^{k+1} = \lambda_i^k + s_k \sum_{l=1}^{n} g_i(\mathcal{M}[w^*(\lambda)](d_l), y_l, d_L) \quad \forall i \tag{13}$$

where s_k is a tunable step size. This step is a *gradient ascent* update to the Lagrange multipliers λ based on Equation 8.

To summarize, the LDL framework consists of repeating two steps:

1. Train a neural network using the Lagrangian relaxation in Equation 11 using fixed multipliers λ^k
2. Update the Lagrange multipliers using Equation 13, obtaining a stronger relaxation with multipliers λ^{k+1}.

3 Method

3.1 Overview: LDL for Inverse Kinematics

The goal of this project is to demonstrate that a neural network can be trained to produce fast, approximate solutions for constrained inverse kinematics problems using the LDL framework. As a proof-of-concept, we will consider a planar, three-link manipulator, although the approach could be generalized to manipulators with more degrees-of-freedom in 3D.

This is challenging because (1) IK problems are highly non-linear and non-convex, and (2) IK problems are severely under-constrained. In general, there can be infinitely many feasible solutions to a particular task. The second problem is particularly challenging when training a neural network to learn an IK mapping from supervised examples, because the training set may contain contradictory examples that have nearby end-effector poses, but very different joint angles that achieve that pose. Essentially, our goal is to learn a particular inverse kinematics mapping among the many possible mappings [6].

For example, the authors of [6] observe that a *direct modelling* approach, where an IK network is trained to mimic joint angle examples from an oracle, can produce unsatisfactory results. If a sum-of-squared error loss function is used (see Equation 15), then different sets of joint angles that map to the same end-effector pose will cause the network to learn an arithmetic average in joint space. An average in joint space does not necessarily yield a correct or feasible result in Cartesian space [6].

3.2 Distal Supervision for Inverse Kinematics

Fortunately, the authors of [6] also propose a *distal learning framework* that addresses the non-uniqueness of the inverse kinematics problem. In the distal learning framework, a learner outputs an action (a *proximal* variable), which is then transformed by the dynamics of the environment into an outcome (a *distal* variable). In our case, a neural network learner outputs a set of joint angles (the proximal variable), and the environment transforms these joint angles into an end-effector pose (the distal outcome) using the forward kinematics of the robot arm.

Because the forward kinematics have a differentiable closed-form, we can define a loss function over the distal end-effector pose rather than the proximal joint angles (see Equation 14). This allows the learner to converge to a *particular* inverse kinematic mapping, and avoids the "joint space averaging" pitfall of the direct-modelling approach.

$$\mathcal{L}_{distal}(\tilde{\theta}, t^{ee}) = \sum_{l=1}^{n} ||f(\tilde{\theta}_l) - t_l^{ee}||_2^2 \qquad \text{Distal loss function} \tag{14}$$

$$\mathcal{L}_{direct}(\tilde{\theta}, \theta) = \sum_{l=1}^{n} ||\tilde{\theta}_l - \theta_l||_2^2 \qquad \text{Direct modelling loss function} \tag{15}$$

where f is the forward kinematics function, $\tilde{\theta}$ are the network's predicted joint angles, t_l^{ee} is the desired end-effector position, and θ is a (non-unique) solution to the inverse kinematics problem from an oracle.

3.3 Constraints

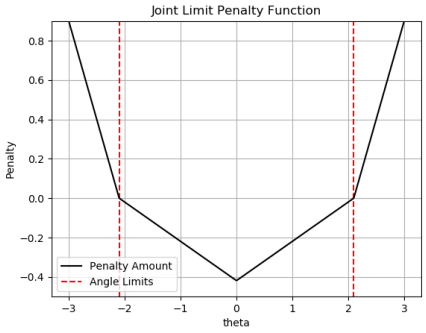
3.3.1 Static Joint Limit Constraints

We require that $\tilde{\theta}^j$, the network's predicted angle for joint j, must be within the range $[\theta^j_{min}, \theta^j_{max}]$. For simplicity, we use a range of $[-\frac{2\pi}{3}, \frac{2\pi}{3}]$ for all joints, although joint-specific limits could be imposed as well.

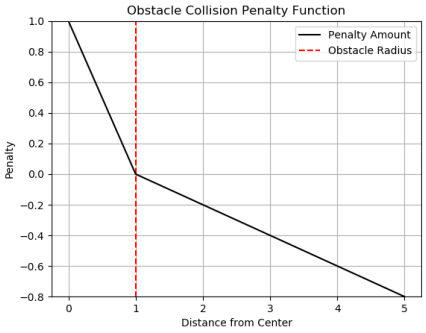
To encode this constraint in the LDL framework, we define a piecewise-linear penalty function g_{JL}

$$g_{JL}(\tilde{\theta}_i) = \begin{cases} \alpha_{in}(-\theta_{i,max} + |\tilde{\theta}_i|) & |\tilde{\theta}_i| \leq \theta_{i,max} \\ \alpha_{out}(-\theta_{i,max} + |\tilde{\theta}_i|) & |\tilde{\theta}_i| > \theta_{i,max} \end{cases} \tag{16}$$

where $\alpha_{out} > 0$ controls how much of a penalty is given for violating the angle limits, and $\alpha_{in} > 0$ controls how much of a "bonus" is given for being within the angle limits. Note that g_{JL} is zero at the joint limit boundaries, negative inside, and positive outside. Our intuition is that a nonzero gradient everywhere will encourage solutions to be well-within the joint limits.



(a) The joint limit penalty function g_{JL} from Equation 16, with θ^j_{max} set to $\frac{2\pi}{3}$, $\alpha_{in} = 0.2$, and $\alpha_{out} = 1$.



(b) The obstacle penalty function g_{OB} from Equation 17, with $\alpha_{in} = 1$ and $\alpha_{out} = 0.2$. The obstacle is a circle with radius 1.0.

Figure 1: A visualization of the piecewise-linear penalty functions used for joint limits (1a) and circular obstacles (1b).

See Figure 1a for an illustration of g_{JL} with $\alpha_{in} = 0.2$ and $\alpha_{out} = 1$.

3.3.2 Static Obstacle Constraints

In addition, we consider static obstacles in the workspace of the manipulator. Here, we make a number of simplifying assumptions:

- Obstacles are circular, parameterized by a center position $c_k \triangleq (x_k, y_k)$ and radius r_k, where k is the index of the object.
- **Only the robot joints are prohibited from colliding with obstacles**. The straight limb components that connect joints are allowed to pass through obstacles, as shown in Figure 2. Clearly, this is not representative of physical collision constraints, but enables a simpler implementation.

Similar to the joint limit constraints, we define a piecewise-linear penalty function g_{OB}:

$$g_{OB,k}(p_i) = \begin{cases} \alpha_{in}(||c_k - p_i||_2 - r_k) & ||c_k - p_i||_2 \leq r_k \\ \alpha_{out}(||c_k - p_i||_2 - r_k) & ||c_k - p_i||_2 > r_k \end{cases} \tag{17}$$

3.3.3 Dynamic Obstacle Constraints

Finally, we would like to avoid dynamic obstacles whose position may change in each task instance. The penalty function used for dynamic obstacles is identical to that of a static obstacle (see Equation 17 and Figure 1b), but the position (x_k, y_k) is randomized for each obstacle in each problem instance.

3.4 Generating Datasets Without an IK Solver

One advantage to the distal supervision approach described in Section 3.2 is that ground-truth joint configurations θ are not needed to supervise the IK network. The training set only contains end-effector positions $\{t^{ee}_l\}^n_{l=1}$ for which *some* feasible solution θ exists. Whereas the direct-modelling approach requires a numerical IK method (i.e the oracle) to provide labels, distal supervision only requires a forward-kinematics model, which is simple to implement in closed-form.

In practice, it is easier to ensure that a feasible IK solution exists by generating training examples using the forward kinematics. We use the following procedure for generating a dataset $\{t^{ee}_l\}^n_{l=1}$:

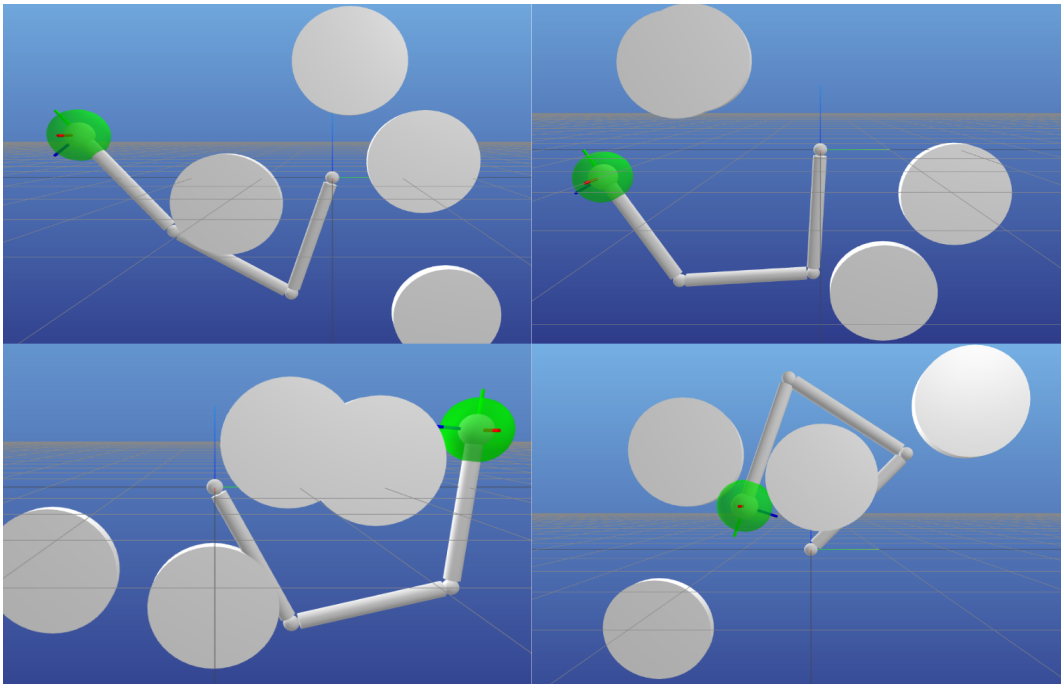


Figure 2: Four examples of randomly generated obstacles and ground-truth joint angle configurations from the training set, generated using the procedure from Section 3.4. The gray circles are obstacles, and the green spheres are centered at the desired end-effector position. These examples correspond to a task with four dynamic obstacle avoidance constraints. As stated in Section 3.3.2, we only consider collisions between joints and obstacles, so these examples do not violate collision constraints, even though some of the limbs pass through objects. The robot arm is shown here for visualization purposes only, and the ground-truth joint angles are not required at training time. Visualization is performed using **PyDrake** [7] and the **meshcat-python** library (https://github.com/rdeits/meshcat-python).

1. (Optional) If the IK task includes dynamic objects, randomly sample their positions (x_k, y_k), for $k \in \{1, 2, ..., K\}$.
2. Uniformly sample angles for all of the joints from their allowable range: $\theta^j \sim [\theta^j_{min}, \theta^j_{max}] \quad \forall j \in \{1, 2, ..., J\}$
3. Apply the forward kinematics to determine the resulting joints positions $\{t^1, t^2, ..., t^J\}$. If any of the joints collide with a static or dynamic obstacle, repeat Step 2.

This procedure guarantees that all end-effector poses are feasible given the problem constraints, and is very fast to compute because it does not require an IK solver in the loop.

3.5 Implementation Details

The code for this project is available at: https://github.com/miloknowles/lagrange-dual-learning

3.5.1 Inverse Kinematics Tasks

We train an IK network using the LDL framework on the following constrained inverse kinematics tasks:

- **Position only**: IK solutions must place the end-effector at the desired position (i.e such that $f(\tilde{\theta}) = s^{ee}$)
- **Position + joint limits**: Predicted joint angles must be between $-\frac{2\pi}{3}$ and $\frac{2\pi}{3}$ (see Section 3.3.1)

- **Position + joint limits + (1) static obstacle**: All joints must avoid a single static obstacle.
- **Position + joint limits + (4) static obstacles**: All joints must not collide with (4) static obstacles.
- **Position + joint limits + (4) dynamic obstacles**: All joints must not collide with (4) dynamic obstacles, whose position is randomized for each training example.

For each task, we generate a training dataset with **40,000** examples and a validation dataset with **4,000** examples according to the procedure in Section 3.4, and with the task-specific constraints.

We evaluate the effectiveness of our method on each of these tasks in Section 4.

3.5.2 Network Architecture

We spent very little time tuning the model architecture for this project, since the focus is on constrained optimization rather than deep learning.

The network used in all experiments is a feed-forward neural network with **8 fully-connected layers** of **100 hidden units** each and **ReLU** nonlinearities. The model is implemented using the **PyTorch** library [8], and has a relatively small number of trainable parameters (**73,103**).

3.5.3 Encoding Constraints as Network Inputs

The network is provided with the parameters of each task instance as input. For our experiments, **we limit the number of total obstacles (static and dynamic) to be at most 4**, although this arbitrary choice could be easily changed. This enables us to pass a fixed-size input vector with **16** values to the network:

- (2) The coordinates of the desired end-effector position $s^{ee} = (x^{ee}, y^{ee})$
- (2) The joint limits θ_{min} and θ_{max}
- (3) Parameters of obstacle 1 (x_1, y_1, r_1)
- (3) Parameters of obstacle 2 (x_2, y_2, r_2)
- (3) Parameters of obstacle 3 (x_3, y_3, r_3)
- (3) Parameters of obstacle 4 (x_4, y_4, r_4)

For tasks where not all constraint inputs are needed, a constant input value of -1 is used.

3.5.4 Optimization Hyperparameters

As in the LDL framework of [4], we use the **Adam** optimizer [9] with a learning rate ($\alpha = 5 \times 10^{-5}$) and β values $(0.9, 0.999)$ to train the neural network $\mathcal{M}$. The step-size s_k used for Lagrange multiplier updates (see Equation 13) is set to 10^{-4}.

In preliminary experiments, we found that training with larger batch sizes tended to result in higher rates of constraint violation. Empirically, a **batch size of 8** led to good performance while still speeding up training due to parallelization.

4 Results

For each task, we train an IK network using the LDL framework and evaluate its performance on a held-out validation set of **4000** task instances. We summarize the *average end-effector position error* as well as the *constraint violation rates* for each model in Table 1. In all tasks, the end-effector position error is less than 5cm, which is very small relative to the length of the three-link arm (3 meters total length). Interestingly, there is very little drop in position error in the more difficult tasks with larger numbers of obstacles to avoid, and position error *improves* with dynamic obstacles. There may be a regularizing effect due to obstacles and joint limits that helps convergence.

	Constraint Violation		
Task	**Position Error (cm)**	**Joint Limit (%)**	**Obstacle Collision (%)**
P	1.99	-	-
P + JL	2.77	3.35	-
P + JL + 1S	2.55	4.15	5.00
P + JL + 4S	4.57	4.25	10.93
P + JL + 4D	3.06	1.67	19.43

Table 1: The average end-effector position error in centimeters and constraint violation rates across a validation set of 4000 instances for each task. A joint limit violation occurs if *any* of the three joints exceed their maximum angle, and a collision occurs if any joint is within the boundaries of any object. The constraint violation rate is the fraction of examples in which a particular type of violation occurs. The task shorthand **P + JL + 4S** means that there is an end-effector position constraint, joint-limit constraints, and four dynamic obstacles that must be avoided. We show qualitative examples of IK solutions to the **P + JL + 4S** task in Figure 3 and Figure 4, as well as examples of the **P + JL + 1S** and **P + JL + 4S** in Figure 5.

Batch Size	1	8	64	512
Inference Time (ms/ex)	0.329	0.0387	0.00478	0.000595

Table 2: Model inference time for different batch sizes on an NVIDIA GeForce GTX 1050M GPU. One advantage of the proposed approach is that many inverse kinematics tasks can be solved in parallel, reducing the amount of time per-example. Even without parallelization, our model takes less than one millisecond to solve one task instance.

In addition, joint limit constraints are seldom violated by the IK network's solutions (less than 5 percent of the time for all tasks), and these constraints are not violated much more frequently for tasks with more obstacles. The obstacle collision constraints are substantially harder to satisfy, especially for dynamic obstacles. In the **P + JL + 4S** task, which has 4 static obstacles, the IK solution places a joint inside a joint about 11 percent of the time. In the **P + JL + 4D** task, which has 4 dynamic obstacles, the violation rate is much worse, at about 19.5 percent. We show examples of obstacle and joint limit constraint violations in Figure 4.

Finally, in Table 2, we summarize the runtime of our network for solving IK problems. One key advantage of using a neural network is that many IK tasks can be solved in parallel. For trajectory optimization, where IK solutions are needed for many intermediate end-effector poses, this could be particularly useful.

5 Conclusions and Future Work

This project was a proof-of-concept to demonstrate that a neural network can learn fast approximations to inverse kinematics problems despite the non-uniqueness, non-convexity, and non-linearity of the tasks. In Section 4, we showed that the network can produce IK solutions with very low position error, and modest rates of constraint violation. Additional work is needed to deal with obstacle collision constraints more effectively, possibly by changing the neural network architecture or the form of the constraint penalty functions. In addition to reducing collision rates, further work is needed to incorporate more complex and possibly non-convex obstacle geometries.

One compelling use for our method is as an initialization step for numerical methods that require a good starting point to converge. A single pass through the network is very fast, as shown in Table 2, and could substantially reduce the number of iterations needed by a numerical method to achieve a good solution. As a post-processing step, iterative refinement with a numerical method would likely reduce constraint violation rates substantially and improve position error even further.

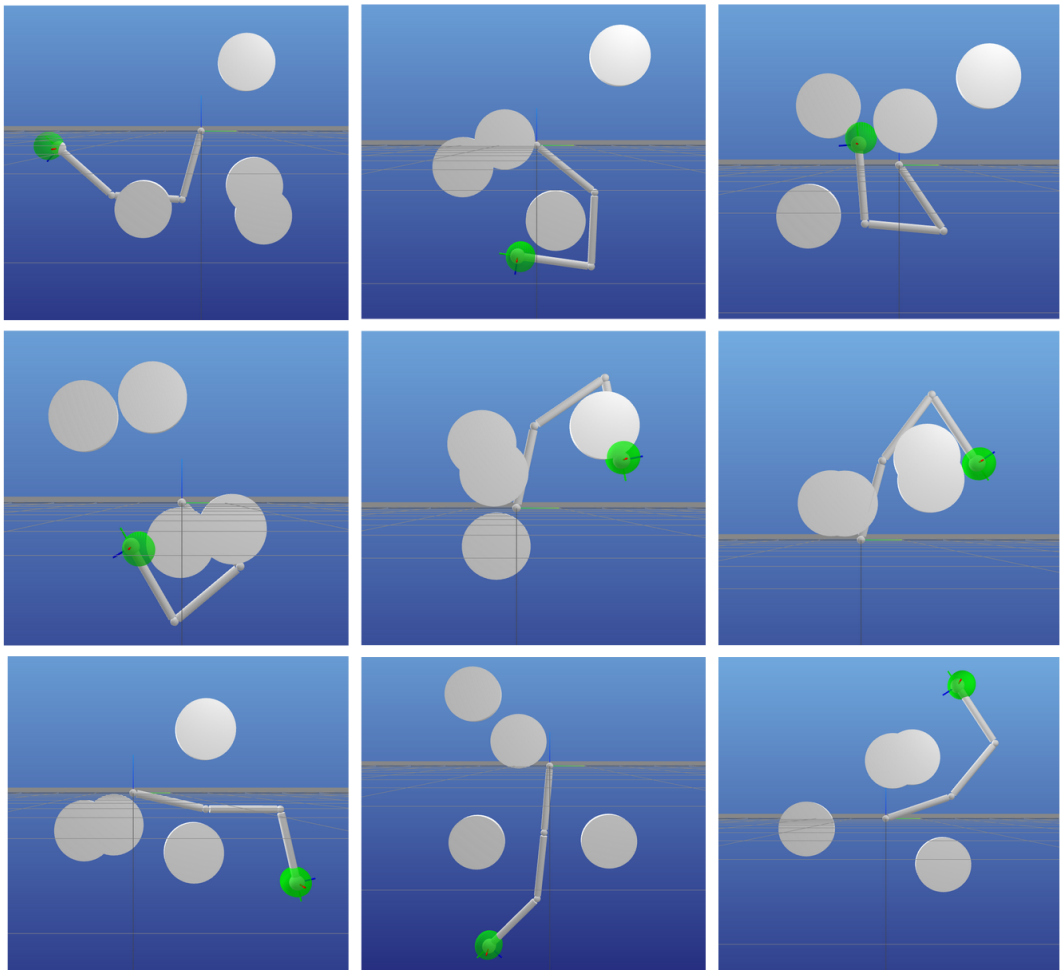


Figure 3: Examples of successful IK solutions from the **P + JL + 4D** task in Table 1. In these examples, all joint limits are satisfied and none of the joints are in collision with obstacles. As stated in Section 3.3.2, we only consider collisions between joints and obstacles, so these examples do not violate collision constraints, even though some of the limbs pass through objects. In fact, it appears that the IK network has learned to exploit this constraint, and often places joints on either side of an obstacle.

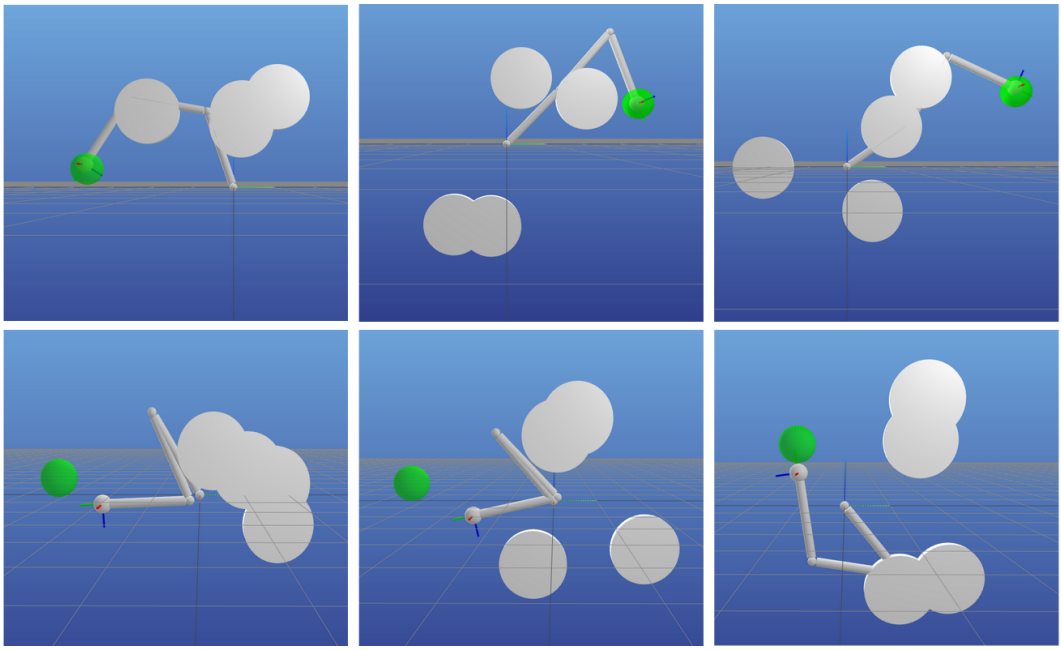


Figure 4: Examples of failed IK solutions from the **P + JL + 4D** task in Table 1. In the top row, we show three examples where obstacle collision constraints are violated. In the bottom row, we show three examples with high end-effector position error and joint limit violations. We notice that these two types of constraint violations tend to co-occur with joint limit violations. Note that the **green spheres contain a 20cm radius** around the goal position.

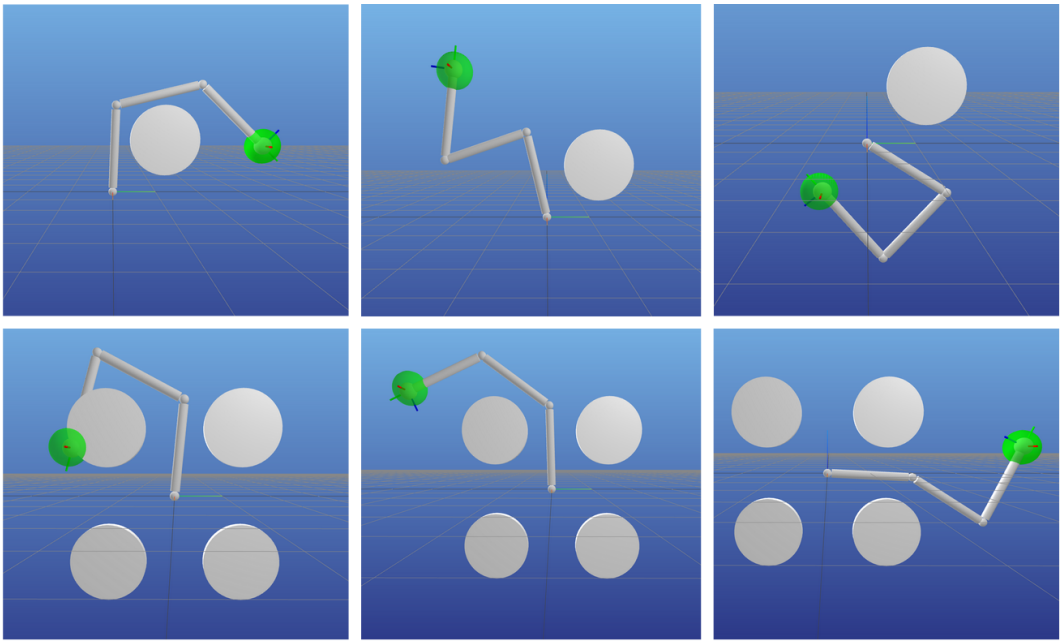


Figure 5: Examples of successful solutions for the **P + JL + 1S** (top row) and **P + JL + 4S** (bottom row) tasks from Table 1.

References

[1] Filip Maric, Matthew Giamou, Soroush Khoubyarian, Ivan Petrovic, and Jonathan Kelly. Inverse Kinematics for Serial Kinematic Chains via Sum of Squares Optimization. *arXiv*, 2019.

[2] A. Aristidou, J. Lasenby, Y. Chrysanthou, and A. Shamir. Inverse Kinematics Techniques in Computer Graphics: A Survey. *Computer Graphics Forum*, 37(6):35–58, 2018.

[3] Andreas Aristidou and Joan Lasenby. FABRIK: A fast, iterative solver for the Inverse Kinematics problem. *Graphical Models*, 73(5):243–260, 2011.

[4] Ferdinando Fioretto, Terrence WK Mak, Federico Baldo, Michele Lombardi, and Pascal Van Hentenryck. A Lagrangian Dual Framework for Deep Neural Networks with Constraints. *arXiv*, 2020.

[5] Servey Levine and Vladlen Koltun. Learning Complex Neural Network Policies with Trajectory Optimization. *ICML*, 32:232 p., 2014.

[6] Michael I. Jordan and David E. Rumelhart. Forward models: Supervised learning with a distal teacher. *Cognitive Science*, 16(3):307–354, 1992.

[7] Russ Tedrake and the Drake Development Team. Drake: Model-based design and verification for robotics, 2019.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.