Application of an MLP for Inverse Kinematics of a 6-DOF Robotic Arm

Jackie Le
University of Waterloo
MTE 203
jddle@uwaterloo.ca

I. Introduction

In the ever-evolving field of robotics, the precision and flexibility of robotic arms are paramount. The six degrees of freedom (6-DOF) robotic arm exemplifies this, capable of intricate maneuvers by controlling position—along the x, y, and z axes—and orientation, through pitch, yaw, and roll movements. This report delves into an innovative approach to the complex inverse kinematics (IK) problem for such an arm, employing a multi-layer perceptron (MLP) for function approximation.

An MLP can be thought of as a interconnected network of nodes for mapping input vectors to output vectors using multi-variable functions. The network mimics the biological neural network in brains by using layers of interconnected nodes that operate like neurons each performing a computation. This network comprises an input layer, which introduces the unique input variables; a hidden layer, which discerns intricate relationships through a weighted sum of inputs; and an output layer, which computes these relationships to produce a result. These computations are then integrated with calculus concepts such differentiation and optimization, to analyze the relationship between nodes and adjusts them to achieve desired outputs.

This leads us to the intended application: MLPs in addressing the IK problem. The application of an MLP for solving IK problems is unorthodox and experimental; however, if successful, it provides a superior solution. The complexity of the IK problem stems from the impressive flexibility achieved by having six rotating joints. The increased maneuverability introduces a characteristic that makes the solution to the IK problem ambiguous, i.e., there is no unique combination of joint angles for a given target position and orientation. This issue is commonly solved using iterative numerical methods that apply approximation techniques to produce any solution. This approach is computationally expensive for real-time results as opposed to a successfully trained MLP, which can provide an output in one computation of the MLP for any given input.

II. Background

In the foundational work by Rumel, D.,Hinton G., & Williams, R. (1986), the backpropagation algorithm was introduced as an innovative approach to training neural networks such as an MLP. This approach employs calculus concepts such as partial derivates, gradients, and optimization to adjust weights in relation to computed errors [1]. To explore how these principles are applied in the context of an MLP, understand that an MLP is a neural network that consists of an input layer for passing along inputs, a system of hidden layers that contains varying numbers of nodes and layers and an output layer that has a node for each unique output variable. The input layer is mapped to the output layer through the weights and biases stored by the nodes in the hidden layer and output layer. A depiction of these interconnected layers is shown in Fig. 1.

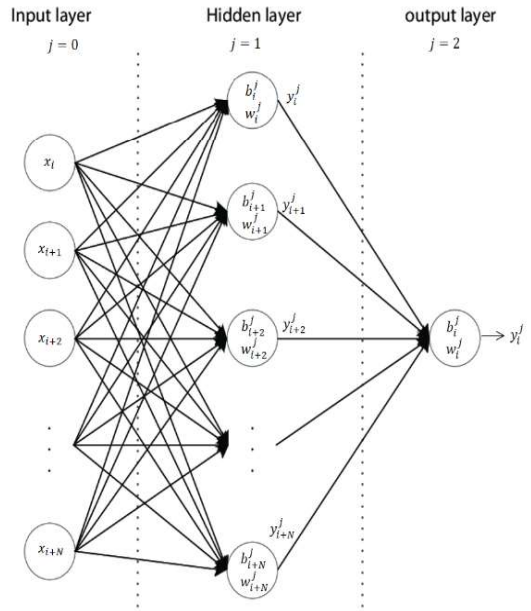

Fig. 1. Three Layered MLP Structure Labelled Weights and Biases for Each Node

Before understanding backpropagation, we must first understand forward propagation. The forward propagation of an MLP is the calculation of the output from each node by applying the weights and biases to the previous layers' outputs. Note that the output (y) from the input layer is simply the input into the MLP (x). Then the outputs from any node can be derived as:

$$z_i^{(j+1)} = \sum_{i=0}^{N}(w_i^{(j+1)} y_i^{(j)}) + b_i^{(j+1)} \rightarrow y_i^{(j+1)} = f\left(z_i^{(j+1)}\right) \qquad (1)$$

where:

- i is the current node,
- j is the current layer,
- z is the output before applying activation function
- b represents a bias from the node,
- w represents the weight of the node,
- y represents the vector output from a node
- f represents the activation function [2].

The activation function is a pivotal component of an MLP because it helps map abstract relationships through non-linear transformations. Without the activation function, the forward propagation equations could only produce linear relationships. The activation function can take the form of any linear or non-linear function to better represent the abstract relationships between layers [3].

From the forward computations we can derive the following backpropagation formulas where L represents the loss function. The loss function is the error between the predicted and the desired outputs. Therefore, from the forward propagation equation we can define the function $L(\hat{y}, y)$, where $\hat{y}$ is the actual target value (a constant) and y is the output from the MLP which could be defined as a function of the variables from the weights and biases of each node. From this definition we can derive the gradient equations (2 and 3) necessary for backpropagation. To better understand the derivation of these gradients, the chain rule tree for the loss function was depicted, see Fig 2, for the MLP structure shown in Fig. 1.

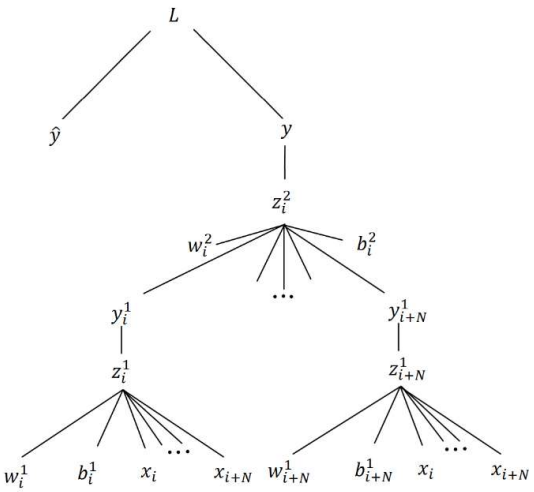

Fig. 2. Loss Function Chain Rule Tree Derived From Fig. 1 MLP

$$\nabla_{w_i^{(j+1)}} L = \frac{\partial L}{\partial w_i^{(j+1)}} = \frac{\partial L}{\partial y_i^{(j+1)}} \cdot \frac{\partial y_i^{(j+1)}}{\partial z_i^{(j+1)}} \cdot \underbrace{\frac{\partial z_i^{(j+1)}}{\partial w_i^{(j+1)}}}_{=y_i^j} \tag{2}$$

$$\nabla_{b_i^{(j+1)}} L = \frac{\partial L}{\partial b_i^{(j+1)}} = \frac{\partial L}{\partial y_i^{(j+1)}} \cdot \frac{\partial y_i^{(j+1)}}{\partial z_i^{(j+1)}} \cdot \underbrace{\frac{\partial z_i^{(j+1)}}{\partial b_i^{(j+1)}}}_{=1} \tag{3}$$

$\frac{\partial L}{\partial w_i^{(j+1)}}$ and $\frac{\partial L}{\partial b_i^{(j+1)}}$ are partial derivatives representing the gradient of the loss function given changes in $w_i^{(j+1)}$ and $b_i^{(j+1)}$ the weight and bias of the next layer, respectively. The gradient must be calculated for each weight and bias, which can be done by proceeding down through the tree and calculating the partial derivatives for each layer. By backpropagating through the MLP, the value of the loss function can be reduced by applying the negative value of these gradients to the weights and biases. This is referred to as gradient descent [4].

This process of training a MLP model through backpropagation and gradient descent is influenced by two notable features. First, the learning rate, which is multiplied by the negative gradient during gradient descent. The importance of this lies in the visualization and understanding of multi-variable functions. When performing gradient descent if the learning rate is too high, it is possible to overshoot the local minimum possibly leading to divergence as seen in Fig. 3. Second, batch size, which is the number of data points that are used for each backpropagation. If all the data points are used at once it is possible that the gradient descent leads to a local minimum or saddle point halting the optimization. To prevent this the data can be split into batches before backpropagating, reducing the effects of local minimums or saddle points at the cost of additional computations [5].

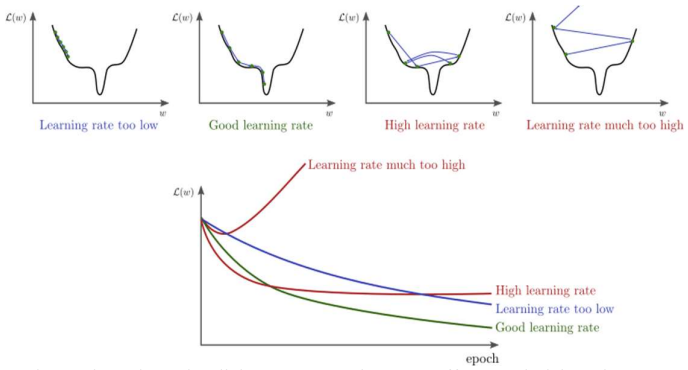

Fig. 3. Five Plots Visualizing How Learning Rate Effects Optimizing The Loss Function [5]

This general concept of MLPs and backpropagation will be applied in this report to solve the IK problem for a 6-DOF robotic arm. The mathematical implications of the forward and inverse kinematics of the robotic arm are not the focus of this report and will only be discussed when required. The application of an MLP for such a complex function approximation will prove difficult, however, if successful will produce an innovative solution that efficiently produces IK solutions [6].

III. Approach

By applying the mathematical concepts outlined in the background, this study aims to train and optimize an IK solution for a 6-DOF robotic arm using an MLP. The MLP will have three nodes for the input layer, which represent the target position in cartesian coordinates (x, y, z). There will be one hidden layer with twenty nodes. Using one hidden layer simplifies the backpropagation, making it easier to intuitively understand how certain parameters affect the MLP when designing a solution. Twenty hidden nodes were chosen to sufficiently map the complex relationship between the robotic arm position and joint angles. The same number of hidden nodes was applied in a research study for a similar application performed at the Memorial University of Newfoundland, and produced viable results [6]. The output layer will have five nodes, one for each of the first five joints, since the sixth joint only affects the orientation. By focusing only on position, the number of input and output nodes is reduced, significantly simplifying the problem.

The dataset that will be used for training the MLP will include 10^5 random datapoints which will provide 10 points for each output. This keeps the number of data points manageable while sufficiently mapping the boundaries of the IK solution. In the case that the boundaries are mapped inaccurately, the number of datapoints can be increased or potentially collected methodically to cover all parameters. The activation function that will be used for the MLP is the hyperbolic tangent function:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \rightarrow \frac{d}{dx}\tanh(x) = 1 - \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)^2 \quad (4)$$

This activation function was chosen over other commonly used functions, such as the rectified linear unit function or sigmoid function, because of its zero-centered behavior. Unlike the other two functions mentioned, the tanh function includes negative values with an output range of [-1,1] as opposed to [0,1]. This is crucial in the application of robotics because the x and y input values will be mapped evenly in the positive and negative directions [3].

Lastly, the loss function for this application will be unorthodox, however, more intuitive in the application of kinematics. In place of the conventional mean squared error function which compares each output with the expected output and squares the difference, a Euclidean distance function will be applied. This function will use the predicted five joint angle outputs to calculate error as the distance between the target and resulting forward kinematic position. The Euclidean distance function is an imperfect approach as it fails to incorporate the joint angles from the dataset, thus ignoring any relation to the limitations of the robotic arms. However, the MSE function is unsuitable for this application because any given target position will have a complex range of possible output angles, making it difficult to minimize the function.

IV. Implementation

A. Collecting and Processing Data

The training dataset was compiled by importing a 3D model of the robotic arm into a web application using JavaScript and the Three.js library [7]. This environment was utilized to apply the forward kinematics of the robotic arm with its physical attributes to validate random joint configurations. Axis-Aligned Bounding Boxes (AABBs) were generated around each robotic arm component in yellow using Three.js (see Fig. 3) to provide a rough but adequate spatial representation of each joint's area. To avoid self-collision the end-effector must remain outside the AABBs of the base and link 2 which are the three most left boxes in Fig. 3. Additionally, the minimum point of each AABB verified that the arm remained above zero height (y-value). These method, combined with the specific joint angle constraints listed in Table 1, compiled a dataset with 100,000 (10^5) unique joint angle configurations and the position in millimeters. While this approach simplified the validation, it also limited the arm's range of motion. Note that the coordinate system used assigns the y-axis upwards.

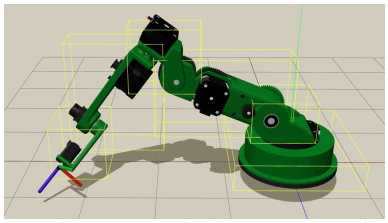

Fig. 4 AABBs of the 6-DOF Robotic Arm Demonstrating How the Boxes Are Exaggerated Visualisation of Each Component [7]

TABLE II. Joint Angle Ranges

Joint	*Min [Degrees°]*	*Max [Degrees °]*
1	0	360
2	-90	90
3	90	270
4	0	360
5	-150	150

After collecting a sufficient dataset, the implementation was programmed using Python and the NumPy library for mathematical calculations [8]. The process for coding the MLP began by importing the dataset and normalizing it. Normalizing the dataset optimizes the training process by creating a consistent range for each feature in the dataset [2]. The data is then split into two arrays for inputs and outputs. The code block for this implementation is as follows:

```
X = DATA[['x', 'y', 'z']].values  # Input features
y = DATA[['j1', 'j2', 'j3', 'j4', 'j5']].values  # Target values
# Calculate the mean and standard deviation for features
X_mean = np.mean(X, axis=0)
X_std = np.std(X, axis=0)
# Calculate the mean and standard deviation for labels
y_mean = np.mean(y, axis=0)
y_std = np.std(y, axis=0)
# Normalize the data
X = (X - X_mean) / X_std
y = (y - y_mean) / y_std
```

B. *Loss and Activation Function*

After processing the data, the activation function (tanh) and the loss function along with their derivatives/gradients were programmed. The code for these functions is as follows:

```
# Activation function -> Hyperbolic Tangent (tanh)
def tanh(x):
    return np.tanh(x)
# Derivative of tanh for use in backpropagation
def tanh_derivative(x):
    return 1.0 - np.tanh(x)**2
# Euclidean distance from 3D Distance between two points as loss function
def euclidean_loss(X_true, y_pred):
    # Calculate the predicted positions for each pair of joint angles
    y_pred_positions = np.array([FK.forward_kinematics(pred_angles) for pred_angles in y_pred])
    # Calculate the Euclidean distances for each pair of points
    distances = np.sqrt(np.sum((X_true - y_pred_positions) ** 2, axis=1))
    # Calculate the mean of the distances
    mean_distance = np.mean(distances)
    return mean_distance
# Function from autograd library to compute the gradient of the loss function
# wrt to input[1] = predicted joint angles
euclidean_loss_grad = grad(euclidean_loss, 1)
```

The tanh function is implemented with the NumPy library and the derivative of the tanh function is derived as $1 - \tanh(x)^2$. The Euclidean distance loss function is implemented using the forward kinematics equation from a library for the 6-DOF robotic arm. To calculate the gradient with respect to the predicted angles *autograd* library was used. This library calculates the gradient of any function passed into the *grad* function by tracking the usage of mathematical operations performed using the NumPy library [9]. The *grad* function builds a tree consisting of each computation during the forward pass of the loss function. This tree is then used to calculate the gradient by applying the chain rule, product rule, and function composition for higher-order partial derivatives.

C. *Forward and Backward Propagation*

Applying the functions defined above the forward propagation function was implemented as follows:

```
def forward(X, W1, b1, W2, b2):
# W1 has shape (#input nodes, #hidden nodes), b1 has shape (#hidden nodes,)
# W2 has shape ((#hidden nodes, #output nodes), b2 has shape (#output nodes,)
    Z1 = np.dot(X, W1) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = np.tanh(Z2)
    return A1, A2
```

The forward propagation function applies the weights and biases to calculate the outputs for each node which are stored in A1 and A2, where A1 stores the output for each input node and A2 stores the output for each hidden layer node. This calculation is performed for each datapoint in X, and stored in A1, A2. Next, we can implement the backward propagation function in a similar manner. The function was coded as follows:

```
def backprop(X, X_denormalized, A1, A2, A2_denormalized, W1, b1, W2, b2, lr):
    # ∂L/∂A2
    grad_loss_joint = euclidean_loss_grad(X_denormalized, A2_denormalized)
    # ∂L/∂Z2 = ∂L/∂A2 * ∂A2/∂Z2
    dZ2 = grad_loss_joint * tanh_derivative(A2)
    # ∂L/∂W2 = ∂L/∂A2 * ∂A2/∂Z2 * ∂Z2/∂W2 || A1.T is ∂Z2/∂W2
    dW2 = np.dot(A1.T, dZ2)
    # ∂L/∂b1 = ∂L/∂A2 * ∂A2/∂Z2
    db2 = np.sum(dZ2, axis=0)
    # ∂L/∂A1 = ∂L/∂Z2 * ∂Z2/∂A1 || W2.T is ∂Z2/∂A1
    dA1 = np.dot(dZ2, W2.T)
    # ∂L/∂Z1 = ∂L/∂A1 * ∂A1/∂Z1
    dZ1 = dA1 * tanh_derivative(A1)
    # ∂L/∂W1 = ∂L/∂Z1 * ∂Z1/∂W1 || X.T is ∂Z1/∂W1
    dW1 = np.dot(X.T, dZ1)
    # ∂L/∂b1 = ∂L/∂A1 * ∂A1/∂Z1
    db1 = np.sum(dZ1, axis=0)
# Update weights and biases
    W1 -= lr * dW1
    b1 -= lr * db1
    W2 -= lr * dW2
    b2 -= lr * db2
    return W1, b1, W2, b2
```

The code includes comments that match lines of code to the mathematical derivations made in the background sections. The backward propagation in summary is calculating the gradient of the loss function for each node with respect to the weights and biases and then adjusting them by subtracting the gradients by a learning rate factor to iteratively minimize the loss. The subtraction of the gradient to minimize a loss function is commonly referred to as gradient descent.

D. Assembled Multi-Layer-Perceptron

Finally, all the functions can be integrated to create an MLP for training a model that can produce joint angles for the 6-DOF robotic arm given a cartesian position. This integration is programmed as follows:

```
# Define the MLP architecture
input_size = 3  # Number of input features
hidden_size = 20  # Number of nodes in the hidden layer
output_size = 5  # Number of output values
# Initialize weights and biases
W1 = np.random.randn(input_size, hidden_size) * 0.1  # random values for weights
b1 = np.zeros((hidden_size,)) # zeros for biases
W2 = np.random.randn(hidden_size, output_size) * 0.1  # random values for weights
b2 = np.zeros((output_size,)) # zeros for biases
def train(X_train, y_train, W1, b1, W2, b2, X_mean, X_std, y_mean, y_std, epochs, lr, batch_size=32):
    # Loop over the specified number of epochs
    for epoch in range(epochs):
        # Determine the total number of data points
        n_samples = X_train.shape[0]
        # Calculate the number of batches
        n_batches = n_samples // batch_size
        # Loop over each batch
        for batch in range(n_batches):
            # Determine the start and end indices of the current batch
            begin = batch * batch_size
            end = min(begin + batch_size, n_samples)
            # Extract the batch data
            X_batch = X_train[begin:end]
            # Forward Propagation: Compute the Node Outputs Using Equation (1)
            A1, A2 = forward(X_batch, W1, b1, W2, b2)
            # Denormalize the input and output data
            X_batch_denormalized = X_batch * X_std + X_mean
            A2_denormalized = A2 * y_std + y_mean
            # Backward pass: Update weights and biases based on the gradient
            W1, b1, W2, b2 = backprop(X_batch, X_batch_denormalized, A1, A2, A2_denormalized, W1, b1,
W2, b2, lr)
      # Return the final weights and biases
      return W1, b1, W2, b2
W1, b1, W2, b2 = train(X, y, W1, b1, W2, b2, X_mean, X_std, y_mean, y_std, epochs=1000, lr=0.01)
```

This code encapsulates the complete integration of an MLP for training an IK solution. To summarize, the process begins with defining the MLP structure and preparing the dataset. Initial weights and biases are randomized. Upon calling the training function, the MLP undergoes repeated cycles of forward and backpropagation. During forward propagation, each node's output is computed. In backpropagation, these outputs (A2) are denormalized to apply the Forward Kinematics (FK) equation within the loss function. Here, gradients are calculated to update weights and biases, aiming to minimize the loss function.

This cycle is executed across numerous batches and epochs - a batch representing a subset of the dataset, while an epoch encompasses a full pass through the entire dataset. Implementing batching is particularly crucial in our complex IK problem, as it enhances convergence efficiency. Smaller batches reduce the likelihood of gradients getting trapped in local minima or stalling at saddle points, leading to faster convergence with a large dataset.

E. Experiments

The process of training a successful MLP requires extensive optimization and iterations. The backpropagation is influenced by multiple factors with effects that are best understood through experimentation. To analyze and optimize the backpropagation, the *train* function defined above was slightly altered to include code for outputting the loss, weights, bias, and training time for each epoch in a text file. The code for producing these outputs are trivial given the results are interpretable.

The experiments that were conducted included training the MLP on a smaller dataset of 10,000 data points, varying only the learning rate. The loss for each epoch during training was collected and analyzed to tune the learning rate. After tuning the learning rate, multiple MLP models were trained with varying initial values. These models were trained until the rate of minimization of the loss was negligible.

To assess the results of these experiments the trained MLP models were saved by storing the weights and biases in a text file. These weights and biases were then imported and used on a separate data set to generate results for analysis. The code for this is as follows:

```
# Import the weights and biases from the saved model as (W1, b1, W2, b2)
# Perform Forward Propagation to calculate the predicted joint angles
A1, A2 = forward(X, W1, b1, W2, b2)
A2_denormalized = A2 * y_std + y_mean
X_denormalized = X * x_std + x_mean
for i in range(X.shape[0]):
    distance_error.append(euclidean_loss(X_denormalized[i], A2_denormalized[i]))
    y_pred_positions.append(FK.forward_kinematics(A2_denormalized[i]))
# Output all data to a CSV file (target/predicted positions, joint angles, and distance error)
```

V. Results and Discussion

A. Experiment Results

Following the implementation and approach described in the previous sections, multiple MLP models were trained with varying learning rates to optimize the backpropagation. The experiments began with an arbitrary learning rate of 0.01. To interpret the results from these experiments, the loss at each epoch was graphed as seen in Figure X. This graph and all following data visualisation were generated by providing the results from the experiments to ChatGPT. From Fig. 5, it can be observed that a learning rate value of 0.01 resulted in inconsistent backpropagation. These poor results are likely due to the value being too high, thus causing the model to overshoot the minimum. Following, this assumption lower learning rates were tested and graphed. The optimal learning rate was determined to be 1e-05 because it converged the fastest at a consistent rate.

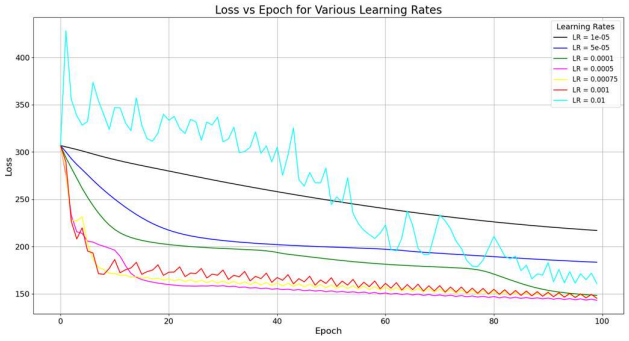

Fig. 5. Graph of Loss vs Epoch for Training With Varying Learning Rates [10]

After finalizing the learning rate, 10 MLPs were trained with different initial values and produced average losses varying from 35.92 to 127.26. This suggests that the initial values used to train the MLP highly influences the success of the backpropagation. It can be assumed that this is due to the nature of the problem, which has multiple solutions for any given target position, resulting in a significant number of local minimums or saddle points. As explained in the background section, this hinders backpropagation. To improve the results, the batch size could be optimized by following the same process used for the learning rate. In addition, considering the assumed abundance of local minimums and saddle points, a method of varying the learning rate while backpropagating could be implemented. This method would make it possible to escape local minimums or saddle points by increasing the learning rate temporarily.

B. MLP In-Depth Error Analysis

From the ten MLPs trained through backpropagation a MLP with an average loss of 35.92 was achieved. The results of this MLP represent an IK function capable of outputting five joint angles that place the end-effector within an average of 35.92mm from the target position.

For deeper analysis of these results, ChatGPT was used to calculate the maximum, minimum, and quartiles values for the loss from 10^5 data points [10]. The results can be found in Fig. 6, which also includes a histogram of the loss generated with ChatGPT. Immediately it was observed from the histogram that the results are severely right skewed, indicating significant outliers. This observation was supported by the low quartile values and high maximum value shown in Fig. 6. This analysis suggests that results from the MLP could be viable if the outliers are removed.

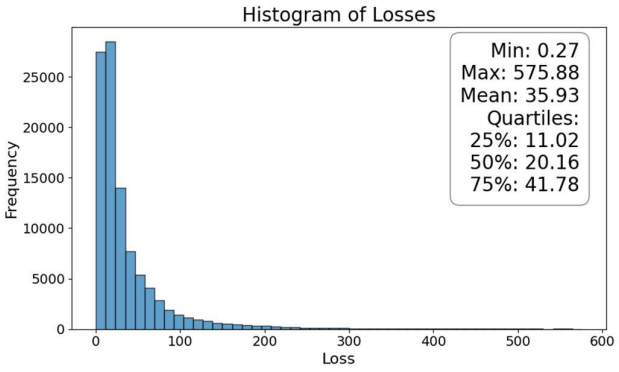

Fig. 6. Histogram of Loss for MLP model tested on 10^5 Data Points with Statistical Values of the Loss [10]

To better understand the cause of the outliers and the success of the MLP, ChatGPT was used to graph the target vs predicted position for each axis (x,y,z), see Fig. 7. These graphs include linear lines to indicate the error (distance from predicted positions to target).

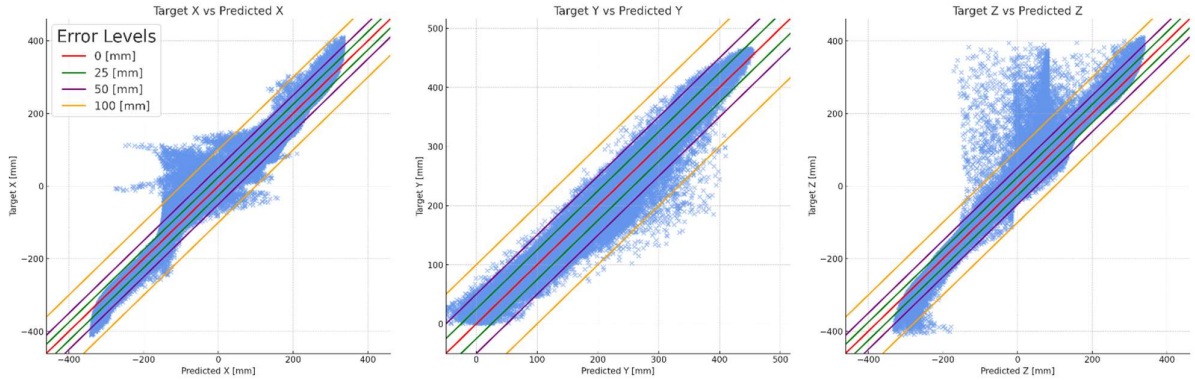

Fig. 7. 2D Scatter Plots of The Target vs. Predicted Position[mm] For Each Axis (x,y,z), Including Bounds to Indicate the Error [10]

These graphs show that the outliers primarily stem from the z-axis position which has a significant number of errors greater than one hundred millimeters for target positions greater than zero. Additionally, it is notable that the x-axis resulted in significant errors for target positions close to zero and one hundred. It is difficult to pin the cause of these results as the MLP maps a complex relationship between the joint angles and target positions, however a likely factor could be biases from the MLP that favor certain joint angles that limit the range of operation causing greater errors.

C. MLP Applicability for Real World Application

To fully assess the applicability of the MLP, the physical constraints of the robotic arm must be considered. Using ChatGPT the range of the predicted joint angles were calculated and tabulated in Table 2. Immediately, it is observed that the results fall outside the limitations outlined in Table 1.

TABLE II. MLP MODEL PREDICTED JOINT ANGLE RANGE

Joint	*Min [Degrees°]*	*Max [Degrees °]*
1	179.91	492.12
2	10.75	197.72
3	188.30	282.99
4	179.82	416.98
5	0.21	147.64

However, if the predicted angles are translated by factors of 360° as required then they all fall within the robotic arms limitation. Passing these translated predicted joint angles into the validation process used when collecting data resulted in 0.1512% of the results being invalid. This is an interesting result considering that the implementation utilized a loss function that did not incorporate the joint angles. It suggests that the validation process for collecting data and the forward kinematic equation provided enough insight through the dataset to map the physical constraints. These results suggest that an MLP can feasibly map the physical constraints for the IK of a 6-DOF robotic arm.

Building on the observations regarding the errors from the MLP, it is possible that a lower average error can be achieved by limiting the physical operating range. From Fig. 7, it can be observed that the errors at the edge of the operational range are greater, likely due to a lack of data points. Considering this and previous observations about the error, ChatGPT was used to generate 2D scatter plot graphs of the target positions with losses below the third quartile (41.78), overlapped with a dotted line that represents the robotic arm's actual operating range, see Fig. 8.

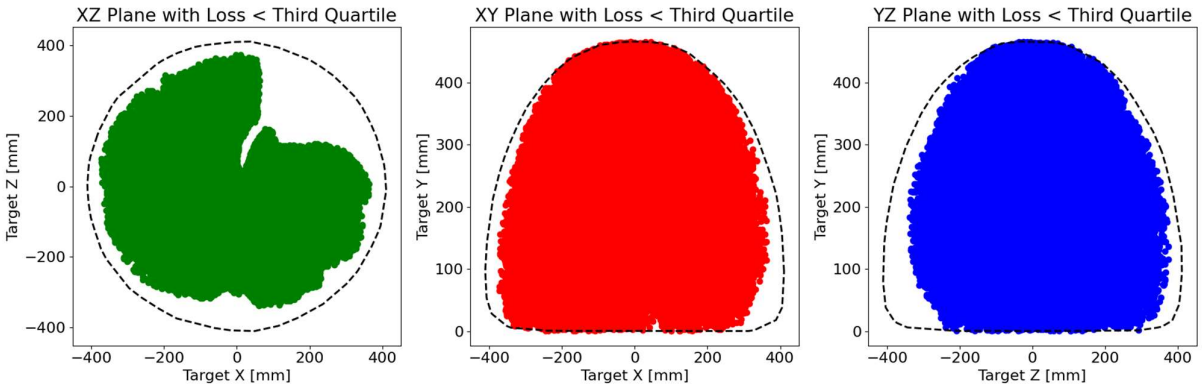

Fig. 8. 2D Scatter Plots of the Target Positions With a Loss Value Less Than The Third Quartile (41.78) With a Dotted Line Indicating The Robotic Arms Operational Range For Each Plane [10]

These graphs demonstrate that the model can provide a usable operating range while producing results with a maximum error of 41.78mm. Using ChatGPT, the average error for data points below the third quartile was calculated as 17.07mm. These results, alongside the range of operation seen in Fig. 8 are impressive. Additionally, from Fig. 8, it is seen that the operational range loss is primarily from the boundaries which could improve by increasing the number of data points for the edge.

Despite these impressive results, the actual applicability of this MLP for real world application is currently limited. The current MLP is designed only for position. This makes the current MLP useless for real world applications since the orientation of end-effector is key for performing any task. To include orientation into the current MLP introduces a significant number of additional variables and relationships. This will drastically increase the complexity and computations required to produce a successful MLP through backpropagation. However, given the impressive results and basic machine learning methods implemented it is reasonable to state that further research could produce a MLP capable of solving the IK position and orientation.

VI. Conclusion

This report has demonstrated the applicability and challenges of employing an MLP to solve the inverse kinematics of a 6-DOF robotic arm, revealing insights that could potentially shape future research. The MLP developed in this report is incapable of use for real-world tasks; however, it demonstrates the feasibility of developing a practical MLP. Utilizing a simple MLP structure and basic backpropagation the MLP produced an average error suitable for tasks requiring an average accuracy of seventeen millimeters at the expense of slightly reducing the operating range.

The implementation and approach utilized in this report represent early research and investigation into the feasibility and applicability of a MLP to model the complex relationships required to solve the IK problems for a 6-DOF robotic arm. The backpropagation implemented in this report employed a constant learning rate incapable of tackling local minimums and saddle points often seen with gradient descent. Additionally, the batch size utilized for backpropagation was unrefined and employed with little research. Furthermore, the MLP structure is unrefined despite being the crux of this innovative solution. It was concluded highly likely that the major errors seen close to the operational range is a result of the unrefined dataset. These insights indicate that further research could possibly produce an MLP capable of solving IK positions and orientation.

The challenges and implications of producing such an MLP would require extensive experimentation to optimize the MLP structure, backpropagation algorithm, dataset, and application of the MLP. However, overcoming such endeavors could produce an innovative method of solving complex IK problems such as the one seen for the 6-DOF robotic arm. A solution that provides incomparable real-time performance at the cost of extensive development. The application of such a solution could introduce optimized robotic arms that require less computational power. The practicality of this application requires an approach capable of replicating successful results for different tasks. This would primarily impact the manufacturing industry that utilizes robotic arms in a highly predictable environment.

Research into the application of an MLP for solving IK problems of a 6-DOF robotic arm stemmed from curiosity and was quickly hindered by the complexity of the issue and unorthodox method. However, by simplifying the problem and applying my current knowledge of mathematical concepts and theories such as multi-variable calculus, gradients, and optimization I grasped a significant understanding of the fundamentals of machine learning. This experience has shed light on the true complexity of the problem initially proposed and has also yielded results inspiring further development into this topic. The results of this report not only provided insights into areas for improvement, but also new interest and curiosity for the mathematical concepts that have built unfathomable machine learning solutions.

REFERENCES

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986. doi:10.1038/323533a0

[2] K. Sanjar, A. Rehman, A. Paul, and K. JeongHong, “Weight dropout for preventing neural networks from overfitting,” *2020 8th International Conference on Orange Technology (ICOT)*, 2020. doi:10.1109/icot51877.2020.9468799

[3] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, “Activation functions in Deep learning: A comprehensive survey and benchmark,” *Neurocomputing*, vol. 503, pp. 92–108, 2022. doi:10.1016/j.neucom.2022.06.111

[4] C. D. Adeza, C. B. Manarin, J. S. Polinar, P. J. Cabrera, and A. B. Santos, “Performance analysis of gradient descent and backpropagation algorithms in classifying areas under community quarantine in the Philippines,” *Advances in Intelligent Systems and Computing*, pp. 77–85, 2022. doi:10.1007/978-981-19-0475-2_7

[5] “What learning rate should I use?,” B. D. Hammel, https://www.bdhammel.com/learning-rates/ (accessed Nov. 23, 2023).

[6] J. Lu, T. Zou, and X. Jiang, “A neural network based approach to inverse kinematics problem for general six-axis robots,” *Sensors*, vol. 22, no. 22, p. 8909, 2022. doi:10.3390/s22228909

[7] "Three.js, version r123," 2021. [Online]. Available: https://threejs.org/. (accessed Nov. 23, 2023).

[8] "NumPy, version 1.21," 2021. [Online]. Available: https://numpy.org/. (accessed Nov. 23, 2023).

[9] "Autograd," 2021. [Online]. Available: https://github.com/HIPS/autograd. (accessed Nov. 23, 2023).

[10] “ChatGPT, OpenAI," [Online]. Available: https://openai.com/chatgpt. (accessed Nov. 23, 2023).