Robotics and Mechatronics

Homework Six

Mohammad Montazeri
School of Mechanical Engineering
College of Engineering, University of Tehran
Tehran, Iran; 810699269
mohammadmontazeri@ut.ac.ir

***Abstract*—This report explores the use of machine learning and neural networks in robotics, focusing on backpropagation, perceptron learning, CNNs, MLPs for solving IKP, and SVMs. These techniques improve robotic precision and autonomy by enhancing data processing and decision-making. They also facilitate the robotic issues unsolvable by classic analytical approaches.**
***Index Terms*—Neural Network, model, learning, training, loss, backpropagation, prediction, error**

I. Introduction

Machine learning has become essential in robotics for improving control and decision-making systems. Backpropagation is a key algorithm for training neural networks by minimizing prediction errors. Perceptron learning serves as the foundation for creating simple binary classifiers.
Convolutional Neural Networks (CNNs) are crucial for visual perception tasks, enabling robots to process and recognize spatial data. The Inverse Kinematics Problem (IKP), which determines joint angles for desired robot positions, can be effectively solved using Multi-Layer Perceptrons (MLPs).
Support Vector Machines (SVMs) offer robust solutions for classifying data by identifying the optimal separating hyperplane. This report provides an overview of these techniques and their applications in robotics, highlighting their role in enhancing robotic functionality and autonomy.

II. Problem 1: Back Propagation

In neural network notations, we can present the $z_i^{(j)}$ and $a_i^{(j)}$ respectively as the input and output of the i-th neuron on the j-th layer. If we present the predicted outcomes of each neuron on the the output layer of the network as $\hat{O}_i$ for $i = 1, 2$ and the true expected outcome as O_i, we can write the forward path for the network and calculate total error as below.

a) Forward Path (feedforward)

Calculate the input to each hidden node or output node and apply an activation function (e.g., sigmoid) to each sum.

$$z_1^{(1)} = w_1 x_1 + w_3 x_2 + b_1 = 0.15 \times 0.2 + 0.25 \times 0.1 + 0.3$$
$$a_1^{(1)} = \sigma(z_1^{(1)}) = \sigma(0.355) = \frac{1}{1 + \mathrm{e}^{-0.355}} = 0.58782$$
$$z_2^{(1)} = w_2 x_1 + w_4 x_2 + b_1 = 0.2 \times 0.2 + 0.3 \times 0.1 + 0.3$$
$$a_2^{(1)} = \sigma(z_2^{(1)}) = \sigma(0.37) = \frac{1}{1 + \mathrm{e}^{-0.37}} = 0.59145$$

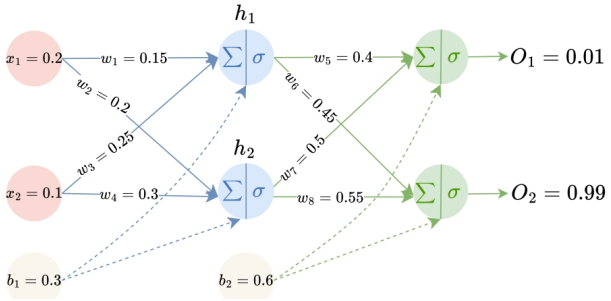


Fig. 1. Architecture of the neural network with its initial values.

$$z_1^{(2)} = w_5 a_1^{(1)} + w_7 a_2^{(1)} + b_2 = 0.4 \times 0.5878 + 0.5 \times 0.5914 + 0.6$$
$$a_1^{(2)} = \sigma(z_1^{(2)}) = \sigma(1.13082) = \frac{1}{1 + \mathrm{e}^{-1.13082}} = 0.75599$$
$$z_2^{(2)} = w_6 a_1^{(1)} + w_8 a_2^{(1)} + b_2 = 0.45 \times 0.5878 + 0.55 \times 0.5914 + 0.6$$
$$a_2^{(2)} = \sigma(z_2^{(2)}) = \sigma(1.18978) = \frac{1}{1 + \mathrm{e}^{-1.18978}} = 0.76670$$

$$\rightarrow \hat{O}_1 = a_1^{(2)} = 0.75599 \tag{1}$$
$$\rightarrow \hat{O}_2 = a_2^{(2)} = 0.76670 \tag{2}$$

Given the target values $O_1 = 0.01$ and $O_2 = 0.99$, we can calculate squared errors and sum them.

$$E_1 = |O_1 - \hat{O}_1| = |0.75599 - 0.01| = 0.74599$$
$$E_2 = |O_2 - \hat{O}_2| = |0.76670 - 0.99| = 0.2233$$

Assuming the cost function specified for this problem to be *sum of squared errors* (SSE), we can find the total loss like[1]:

$$J(w, b) = \frac{1}{2} \sum_{q=1}^{2} \left\| h_{w,b}^q(x) - y^q \right\|^2 \rightarrow \mathcal{L}_{SSE} = 0.5 \left(E_1^2 + E_2^2 \right)$$
$$E_{tot} = 0.5(0.55650 + 0.04986) = \mathbf{0.30318} \tag{3}$$

[1]It should be noted that some resources [6] have a gain of 0.5 in SSE loss formula which helps simplify the equation when a coefficient of 2 is multiplied in it while differentiating in backpropagation process. Although not accordant with some other references [7], this notation is selected in this problem.

b) Backward Path (back-propagation)

The overall procedure for updating the weights in such a network is like:

- Compute the error gradient with respect to the output layer:

$$\delta_1 = (\hat{O}_1 - O_1) \cdot \sigma'(W_5 h_1 + W_7 h_2 + b_2)$$
$$\delta_2 = (\hat{O}_2 - O_2) \cdot \sigma'(W_6 h_1 + W_8 h_2 + b_2)$$

  where (σ') is the derivative of the activation function (e.g., sigmoid).
- Update the weights for the output layer:

$$W_5 \leftarrow W_5 - \text{learning rate} \cdot \delta_1 \cdot h_1$$
$$W_6 \leftarrow W_6 - \text{learning rate} \cdot \delta_2 \cdot h_1$$
$$W_7 \leftarrow W_7 - \text{learning rate} \cdot \delta_1 \cdot h_1$$
$$W_8 \leftarrow W_8 - \text{learning rate} \cdot \delta_2 \cdot h_2$$

- Compute the error gradient with respect to the hidden layer:

$$\delta_{h1} = (\delta_1 \cdot W_5 + \delta_2 \cdot W_6) \cdot \sigma'(W_1 x_1 + W_3 x_2 + b_1)$$
$$\delta_{h2} = (\delta_1 \cdot W_7 + \delta_2 \cdot W_8) \cdot \sigma'(W_2 x_1 + W_4 x_2 + b_1)$$

- Update the weights for the hidden layer:

$$W_1 \leftarrow W_1 - \text{learning rate} \cdot \delta_{h1} \cdot x_1$$
$$W_2 \leftarrow W_2 - \text{learning rate} \cdot \delta_{h2} \cdot x_1$$
$$W_3 \leftarrow W_3 - \text{learning rate} \cdot \delta_{h1} \cdot x_2$$
$$W_4 \leftarrow W_4 - \text{learning rate} \cdot \delta_{h2} \cdot x_2$$

In this problem, to update only w_4 and w_5, we need to differentiate the total loss with respect to them. For this matter, we take learning rate $\eta = 1$ and note that the derivative of the sigmoid function is as below [3].

$$\sigma'(z) = \sigma(z)\,(1 - \sigma(z)) \tag{4}$$

1) Updating W_5:

$$\frac{\partial \mathcal{L}}{\partial w_5} = \frac{1}{2}\left[\frac{\partial E_1^2}{\partial w_5} + \frac{\partial E_2^2}{\partial w_5}\right] = E_1 \frac{\partial E_1}{\partial w_5} \tag{5}$$

$$= E_1 \frac{\partial E_1}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_5} \tag{6}$$

$$= E_1 \times 1 \times a_1^{(2)}\left(1 - a_1^{(2)}\right) \times a_1^{(1)} \tag{7}$$

$$= 0.74599 \times 0.75599(1 - 0.75599) \times 0.58782$$
$$= \mathbf{0.08089115564} \tag{8}$$

$$w_5|_{new} = w_5|_{old} - \eta \nabla_{w_5} \mathcal{L} = 0.4 - \frac{\partial \mathcal{L}}{\partial w_5} = \mathbf{0.31911}$$

2) Updating W_4:

$$\frac{\partial \mathcal{L}}{\partial w_4} = \frac{1}{2}\left[\frac{\partial E_1^2}{\partial w_4} + \frac{\partial E_2^2}{\partial w_4}\right] = \left[E_1 \frac{\partial E_1}{\partial w_4} + E_2 \frac{\partial E_2}{\partial w_4}\right] \tag{9}$$
$$= \left[E_1 \frac{\partial E_1}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_4} + E_2 \frac{\partial E_2}{\partial a_2^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \frac{\partial z_2^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_4}\right]$$

$$\frac{\partial \mathcal{L}}{\partial w_4} = \left[E_1 \times 1 \times a_1^{(2)}\left(1 - a_1^{(2)}\right) \times w_7 \times a_2^{(1)}\left(1 - a_2^{(1)}\right) \times x_2 + E_2 \times 1 \times a_2^{(2)}\left(1 - a_2^{(2)}\right) \times w_8 \times a_2^{(1)}\left(1 - a_2^{(1)}\right) \times x_2\right]$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_4} =& 0.74599 \times 0.75599(1 - 0.75599) \times 0.5 \\ & \times 0.59145(1 - 0.59145) \times 0.1 + \\ & 0.2233 \times 0.76670(1 - 0.76670) \times 0.55 \\ & \times 0.59145(1 - 0.59145) \times 0.1 \\ =& \mathbf{0.00219343754}\end{aligned}$$

$$w_4|_{new} = w_4|_{old} - \eta \nabla_{w_4} \mathcal{L} = 0.3 - \frac{\partial \mathcal{L}}{\partial w_4} = \mathbf{0.29781}$$

III. Problem 2: Perceptron Learning

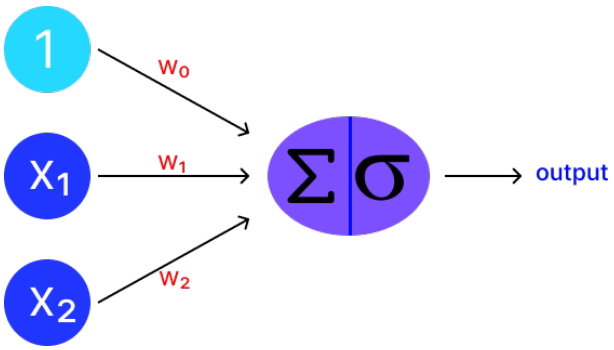


Fig. 2. Architecture of the perceptron used in problem 2.

a) Creating perceptron manually

With the given training samples, our intuition is bent towards the fact that the perceptron outputs positively for inputs which together are greater than or equal to 3. This satisfies all 3 examples. So, the simplest way to put this in perceptron is by giving the values below to the weights and bias.

$$w_0 = -3 \qquad w_1 = w_2 = 1$$
$$z = \sum_0^2 w_i x_i \qquad \sigma(z) = \begin{cases} +1, & z \geq 0 \\ -1, & z < 0 \end{cases} \tag{10}$$

b) Implementing learning algorithm

Now assuming we are unaware of the true weights of this perceptron and unable to derive them by mind, we use the following algorithm to modify weights step by step.

- Let
$$\mathcal{D} = \left(\left\langle \mathbf{x}^{[1]}, y^{[1]} \right\rangle, \ldots, \left\langle \mathbf{x}^{[n]}, y^{[n]} \right\rangle\right) \in (\mathbb{R}^m \times \{0,1\})^n$$
- Initialize $\mathbf{w} = [w_0, w_1, w_2] := [-1, 1, 0.5]$
- For every training epoch:
  - For every $\left\langle \mathbf{x}^{[i]}, y^{[i]} \right\rangle \in \mathcal{D}$:
    1) $\hat{y}^{[i]} := \sigma\left(\mathbf{x}^{[i]\top}\mathbf{w}\right)$
    2) $\text{err} := \left(y^{[i]} - \hat{y}^{[i]}\right)$
    3) $\mathbf{w} := \mathbf{w} + \text{err} \cdot \mathbf{x}^{[i]}$

To implement this, *Python* is used to automate the learning procedure. The code provided for this matter is appended to this report while its result shows:

```
--- epoch 1 ---
w0, w1, w2 = [-1, 1, 0.5]
w0, w1, w2 = [-1, 1, 0.5]
w0, w1, w2 = [-1.5, 1.0, -0.5]
w0, w1, w2 = [-1.5, 1.0, -0.5]
total error = 2
--- epoch 2 ---
w0, w1, w2 = [-1.5, 1.0, -0.5]
w0, w1, w2 = [-2.0, 0.5, 0.0]
w0, w1, w2 = [-2.0, 0.5, 0.0]
w0, w1, w2 = [-1.5, 1.5, 0.5]
total error = 4
--- epoch 3 ---
w0, w1, w2 = [-1.5, 1.5, 0.5]
w0, w1, w2 = [-1.5, 1.5, 0.5]
w0, w1, w2 = [-1.5, 1.5, 0.5]
w0, w1, w2 = [-1.5, 1.5, 0.5]
total error = 0
```

This shows that with the initial weights of `w0, w1, w2 = [-1, 1, 0.5]` and a learning rate of `0.5`, perceptron reaches an admirable answer for weights, in just 3 epochs. Dividing the learning rate by 2 causes the perceptron to reach another true answer with one epoch less, like:

```
--- epoch 1 ---
w0, w1, w2 = [-1, 1, 0.5]
w0, w1, w2 = [-1, 1, 0.5]
w0, w1, w2 = [-1.25, 1.0, 0.0]
w0, w1, w2 = [-1.25, 1.0, 0.0]
total error = 2
--- epoch 2 ---
w0, w1, w2 = [-1.25, 1.0, 0.0]
w0, w1, w2 = [-1.25, 1.0, 0.0]
w0, w1, w2 = [-1.25, 1.0, 0.0]
w0, w1, w2 = [-1.25, 1.0, 0.0]
total error = 0

```

These results _although true_ were obtained using a learning algorithm which updates weights like below and was derived from [6].

$$\mathbf{w} := \mathbf{w} + \mathrm{y}_{\text{true}} \cdot \mathbf{x}^{[i]} \tag{11}$$

After changing the code a little bit to use our own previously-discussed updating rule derived from [7] as $\mathbf{w} := \mathbf{w} + \text{err} \cdot \mathbf{x}^{[i]}$, we get the following result which leads to final weights of **`w0, w1, w2 = [-2, 2, 0.5]`**

```
--- epoch 1 ---
w0, w1, w2 = [-1.0, 3.0, -0.5]
total error = 4
--- epoch 2 ---
w0, w1, w2 = [-2.0, 2.0, 0.5]
total error = 2
--- epoch 3 ---
w0, w1, w2 = [-2.0, 2.0, 0.5]
total error = 0
```

IV. Problem 3: SVM

Support Vector Machines are powerful supervised learning algorithms for both classification and regression. It is a discriminative classifier that is formally defined by a separating hyperplane. So given labelled training data, the algorithm outputs an optimal hyperplane that categorizes new examples.

a) Linear Separability

In this 1D space, it's impossible to draw a single line that separates the positive and negative samples. Therefore, the data is not linearly separable. SVM proposes a solution called *Kernel Trick* which helps moving the data into a higher dimensional space where the data samples would be separated with a hyper-plane.

b) Kernel and Transfer Functions

Given the kernel function:

$$K(x_1, x_2) = \cos\left(\frac{\pi}{4}x_1\right)\cos\left(\frac{\pi}{4}x_2\right) + \sin\left(\frac{\pi}{4}x_1\right)\sin\left(\frac{\pi}{4}x_2\right)$$

This looks like the expression of the cosine of the difference of two angles, which suggests a trigonometric transformation. We can use this to find the transformation function ϕ.
Rewriting the kernel function using the trigonometric identity for the cosine of a difference gives

$$\cos(A - B) = \cos(A)\cos(B) + \sin(A)\sin(B)$$

Let $A = \frac{\pi}{4}x_1$ and $B = \frac{\pi}{4}x_2$. Thus,

$$K(x_1, x_2) = \cos\left(\frac{\pi}{4}x_1 - \frac{\pi}{4}x_2\right) = \cos\left(\frac{\pi}{4}(x_1 - x_2)\right)$$

Knowing

$$K(\mathbf{x_1}, \mathbf{x_2}) = \phi(\mathbf{x_1}) \cdot \phi(\mathbf{x_2}) \tag{12}$$

we can deduce that the transformation function ϕ that maps 1D points to a 2D space is given by:

$$\boldsymbol{\phi}(x) = \left[\cos\left(\frac{\pi}{4}x\right), \sin\left(\frac{\pi}{4}x\right)\right] \tag{13}$$

c) Points and Support Vectors Displayed

Using a Python script which is appended to this file, we can transfer the data points from the 1-D to 2-D space and plot it. We can also use `svm` API from `sklearn` package to fit SVM to these points and plot the separation line along with its gutters (support vectors). As Figure 3 shows the result, it can be seen that some data points happen to overlap on each other wen transferred to 2D space; thus, each point has a number label which shows how many points actually exist on that coordinate. With this module, in addition to all above, we can derive the equation for the separating line as:

```
The equation of the separator line is:
y = 0.00x + -0.00
```

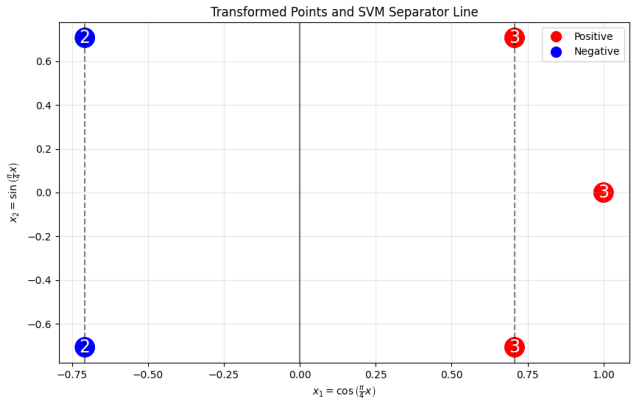


Fig. 3. Transformed points in 2D space with separation and supporting lines.

d) Equation of the Separator Line

Figure 3 shows that the nearest samples of two classes are kind of symmetrically spread about the vertical line $y = 0$ which happens to be the line that causes the biggest margin between the two classes. Therefore, as derived in the previous section, the separator line in the transformed space is obtained as a linear function like:

$$\mathbf{X_1} = \mathbf{0} \tag{14}$$

V. Problem 4: CNN Forward Process

The inputs, filter weights and other parameters of the convolutional neural network in this problem are specified to me as shown in Figure 4.

In other words, after adding one row/column of zero padding around (all over the) inputs, we'll have the following channels as the $2@(5 \times 5)$ input image.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{15}$$

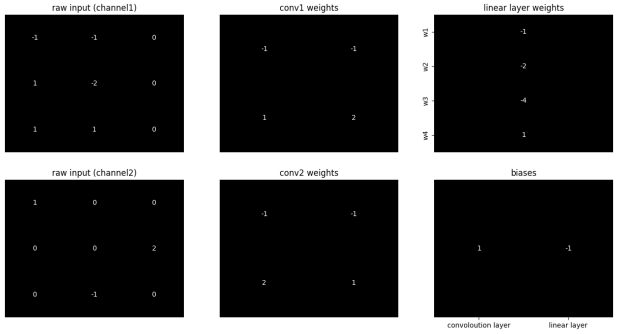


Fig. 4. The specified parameters to the author's specific CNN.

At first we need to convolve this input channels with the filters below, a bias of 1 and a stride of 1 for each. Then we need to add them together to get a uni-channel output.

$$\begin{bmatrix} -1 & -1 \\ 1 & 2 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 \\ 2 & 1 \end{bmatrix} \tag{16}$$

To do so, we use the following convolution rule. (Note that this equation is used for each convolution _with a filter and its corresponding input channel together_ separately.)

$$z = \mathbf{x} \cdot \mathbf{w} + b \tag{17}$$

where z is the scalar output of one filtered part, $\mathbf{x}$ is the 2×2 section of the input, $\mathbf{w}$ is the 2×2 filter weight matrix, and b is the bias.

Therefore, we can calculate the first or the top-left $(0, 0)$ index of the convolved result like:

$$\begin{aligned} z_{(0,0)} = &(0 \times -1 + 0 \times -1 + 0 \times 1 + -1 \times 2 + 1) \\ &+ (0 \times -1 + 0 \times -1 + 0 \times 2 + 1 \times 1 + 1) = 1 \end{aligned}$$

With the same procedure, we obtain

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 4 & 0 & 3 & 6 \\ 3 & 5 & 1 & 0 \\ 1 & 1 & 2 & 2 \end{bmatrix} \tag{18}$$

Dividing the 4×4 result into 4 of 2×2 pieces and extracting each one's maximum value, we can achieve:

$$\begin{bmatrix} 1 & 1 & 1 & 2 \\ 4 & 0 & 3 & 6 \\ 3 & 5 & 1 & 0 \\ 1 & 1 & 2 & 2 \end{bmatrix} \xrightarrow{\text{max pooling}} \begin{bmatrix} \mathbf{4} & \mathbf{6} \\ \mathbf{5} & \mathbf{2} \end{bmatrix}$$

ReLU stands for **Rectified Linear Unit**, which is a commonly used activation function in neural networks and deep learning. It is also used in the CNN under review and is defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{19}$$

Applying this function to the last matrix, we'll get[2]:

$$\begin{bmatrix} 4 & 6 \\ 5 & 2 \end{bmatrix} \tag{20}$$

Now we flatten the 2×2 matrix into a 1-D 1×4 output.

$$\begin{bmatrix} 4 & 6 & 5 & 2 \end{bmatrix} \tag{21}$$

And now we use the linear layer weights specified as

$$\begin{bmatrix} -1 & -2 & -4 & 1 \end{bmatrix}$$

and a bias of -1, to convert the result to a 1-D scalar output.

$$(4 \times -1) + (6 \times -2) + (5 \times -4) + (2 \times 1) - 1 = -35$$

Thus, the final output of this CNN would be:

$$\begin{bmatrix} \mathbf{-35} \end{bmatrix}$$

VI. Problem 5: IKP with MLP

To construct a fully connected neural network with the specified architecture, we'll use Python and one of its popular deep learning libraries, TensorFlow and its high-level API, Keras. The network will have three input features, two hidden layers with 50 neurons each, and an output layer with 3 neurons using a linear activation function. The code is provided in Jupyter Notebook and is attached to this report. Some notable aspects of this code are:

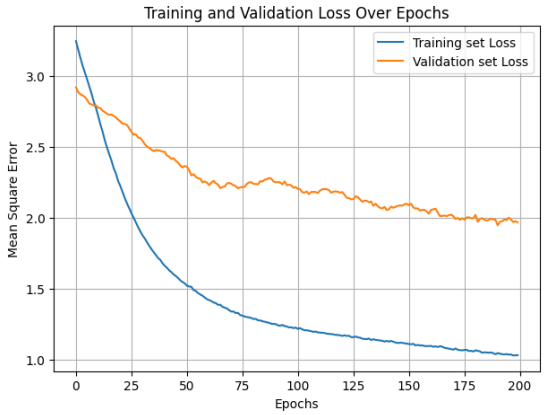
- Sequential: This class from Keras allows you to build a linear stack of layers.
- Dense: This class creates a fully connected layer. The `input_dim` parameter specifies the number of input features (3 in this case).
- ReLU activation: The relu function is a common activation function for hidden layers.
- Linear activation: The linear activation function is used for regression tasks or when you need the output to be a range of real numbers.
- Normalization: You may want to normalize your input data to improve training performance.
- Random state: the keyword `random_state` is used to control the randomness in the algorithms to ensure reproducibility. Here, `random_state=20` guarantees that the training and test sets will contain the same samples each time you run the code, which is critical for reproducibility.

[2]By introducing a non-zero slope for negative inputs, some variants of ReLU function are defined such as *leaky ReLU, PReLU, ELU, etc* which maintain non-linearity and help avoid the dying ReLU problem. They provide a gradient even for negative inputs, which helps maintain the learning process across all neurons. In the case of this problem, since there's no negative index in the original matrix, there would be no difference if any of these functions were applied to it.

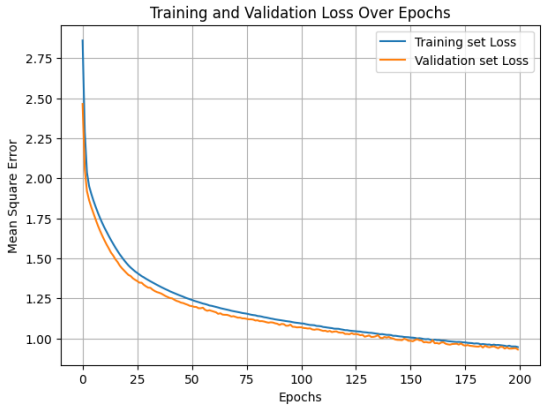
- Overfitting: One of the challenges I faced, was the fact that the more I tried modifying and re-fitting the model, the more overfitting used to emerge. In fact, I split the training part of the code in the jupyter notebook into a separate cell and kept modifying and re-running it with a naive hope to get better answers. In contrast, it used to show more overfitting every time, getting more precise on the training set and worse predictions on the test data. Turns out my model **resumed** training on that dataset in continuation of previous trains; so it used to get into the overfitting trouble.
  In other words, when you repeatedly run the training code in a Jupyter notebook, the model continues training from its current state rather than starting from scratch unless explicitly reinitialized. This continuous training can lead to overfitting, where the model becomes excessively tuned to the training data, reducing its ability to generalize to new, unseen data. Eventually, I found this `tf.keras.backend.clear_session()` command which initializes the model so I can confidently re-train my model every time I execute the program.
- Batch: Machine learning models, especially deep neural networks, are typically trained using a method called *mini-batch gradient descent*, where The training dataset is divided into small subsets called batches. Each batch is used to compute the gradient and update the model weights.
  Key Points to Consider:
  - Memory and Computational Resources: Larger batch sizes require more memory. If you are constrained by memory, you might need to choose a smaller batch size.
  - Training Time: While smaller batch sizes result in more updates per epoch and might appear to train faster in terms of epochs, they might take longer in actual time due to less efficient computation.
  - Generalization: There's evidence suggesting that smaller batch sizes may help with generalization, potentially leading to better performance on unseen data.

  Impact of Batch Size:
  - Small Batch Size: Leads to more frequent updates, which can help the model converge faster but might result in more noisy updates.
  - Large Batch Size: Leads to less frequent updates, which can stabilize the convergence but might require more epochs to converge.
- Dataset size: The model in this problem is trained twice, once with 80% of only 20 data points and once, $20^3 = 8000$ data points. Both of the results are included in this report to demonstrate how larger amount of dataset can increase the agent's reproducibility and accuracy.

Finally, the overall result of the model(s) is depicted in Figure 5. These plots show how the loss (MSE) of the model evolves through each epoch on both training set and validation set, which helps identify overfitting or underfitting. It is also concluded that both of loss curves have a descending trend which end in low errors, although it might decrease even more if the training could surpass a little more than 200 epochs, or the training set were a bit larger. Then, Figure 6 resembles the true values of three outputs of the IKP problem[3] and predicted ones via the model trained with less than 20 samples (Figure 5(a)) in separate figures. These plots provide a visual insight into how well our model is performing. The *actual vs. predicted* plots also give a clear picture of the smaller model's performance on the test data, which might not be acceptably good.



(a) Model trained on 80% of $20 = 16$ data points



(b) Model trained on 80% of $20^3 = 6400$ data points

Fig. 5. Result of the neural network model: Evolution of loss function (MSE) over epochs

In the end, as a demonstration of the agent's performance, its prediction is compared with the actual values derived from the following IKP equations at time step $t = 0$. Here's the *inverse kinematic problem* equations of the 3-DOF serial robot under review in this problem, obtained in *problem 1* of *homework 4*:

[3]joint space variables θ_1, θ_2 and θ_3

$$\theta_1 = \frac{\pi}{4} + \frac{\pi}{9}\sin\left(\frac{\pi}{5}t\right) \tag{22}$$

$$\theta_2 = \frac{\pi}{6} + \frac{\pi}{18}\cos\left(\frac{\pi}{10}t\right) \tag{23}$$

$$\theta_3 = -\frac{\pi}{9} - \frac{\pi}{36}\sin\left(\frac{\pi}{15}t\right) \tag{24}$$

And here's the results obtained from the first _weaker_ model trained with 16 samples:

```
Predicted thetas at t:0 = [0.6419082
↪ 0.78249395 0.01114409]
Actual thetas at t:0  = [ 0.7854   0.69813
↪ -0.34907]
Mean Square Error for the test sample at
↪ t:0 = 0.052487
```

And here's the results obtained from the second _stronger_ model trained with 6400 samples:

```
Predicted thetas at t:0 = [ 1.2475674
↪ 0.1718376  -0.17899337]
Actual thetas at t:0  = [ 0.7854   0.69813
↪ -0.34907]
Mean Square Error for the test sample at
↪ t:0 = 0.173169
```

VII. Conclusion

The use of machine learning techniques such as backpropagation, perceptron learning, CNNs, MLPs for IKP, and SVMs in robotics significantly improves data processing and decision-making capabilities. These methods enable robots to perform complex tasks with greater precision and autonomy. Continued development in these areas is essential for future advancements in robotic systems, leading to more intelligent and adaptable robots.

References

[1] J. Angeles, "Fundamentals of Robotic Mechanical Systems", Theory, Methods, and Algorithms, 4th edition, Springer.

[2] J. J. Craig, "Introduction to Robotics", 3d edition, Pearson Education, Inc.

[3] A. Chakraborty, Arc - Towards Data Science, "Derivative of the Sigmoid function", [Online]. Available: https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e

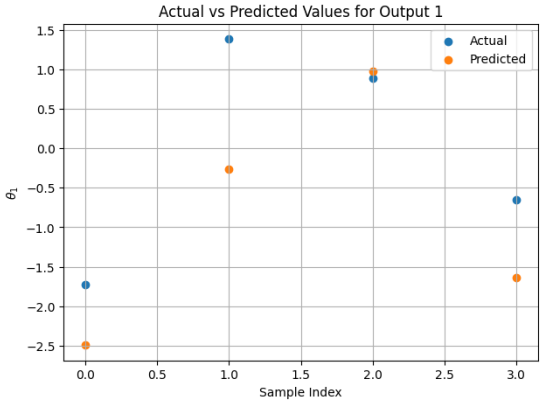
[4] Wikipedia. 2024. "Rectifier (Neural Networks)." Wikimedia Foundation. Last modified April 30, 2024. https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[5] B. Krishnamurthy, "An Introduction to the ReLU Activation Function", https://builtin.com/machine-learning/relu-activation-function

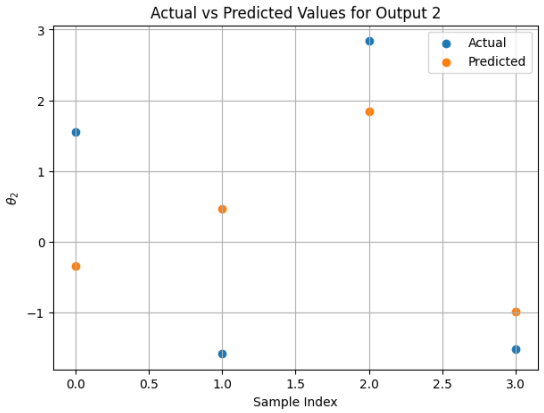
[6] A. Kalhor, "Neural Networks and Deep Learning", course pamphlet, School of Electrical and Computer Engineering, University of Tehran, Spring 2023

[7] H. Hoseini, "AI in Robotics", course pamphlet, University of Tehran, Spring 2024

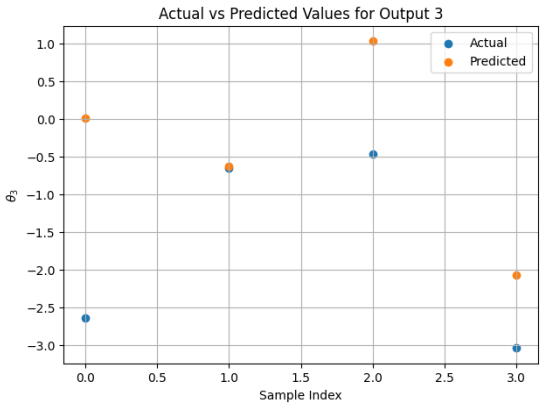
[8] Simplilearn, "Support Vector Machine", https://www.youtube.com/watch?v=TtKF996oEl8



(a) Comparison of predicted and actual values of θ_1 as the first output of the model



(b) Comparison of predicted and actual values of θ_2 as the second output of the model



(c) Comparison of predicted and actual values of θ_3 as the third output of the model

Fig. 6. Result of the neural network model: Investigation of the accuracy of predictions