EXPERIMENT 1: Study of NN toolbox

Neural network (NN) toolboxes involve understanding the features, functionalities, and applications of different NN libraries or frameworks. Here's a guide on how to effectively study NN toolboxes:

1. Choose a Neural Network Toolbox/Framework:

- **TensorFlow**: TensorFlow is an open-source deep learning library developed by Google. It offers a comprehensive ecosystem for building and deploying machine learning models, including neural networks.
- **PyTorch**: PyTorch is an open-source machine learning library developed by Facebook. It provides dynamic computational graphs and a flexible framework for building neural networks.
- **Keras**: Keras is a high-level neural networks API written in Python. It can run on top of TensorFlow, Theano, or Microsoft Cognitive Toolkit (CNTK). Keras offers a user-friendly interface for building and training neural networks.
- **Caffe**: Caffe is a deep learning framework developed by the Berkeley Vision and Learning Center (BVLC). It is known for its speed and efficiency, particularly in image classification tasks.
- **MXNet**: MXNet is an open-source deep learning framework developed by Apache. It offers scalability, flexibility, and support for multiple programming languages.

2. Learn the Basics:

- Familiarize yourself with the basic concepts of neural networks, including feedforward networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs), etc.
- Understand how neural networks learn through optimization algorithms like gradient descent and backpropagation.
- Learn about common activation functions, loss functions, and optimization algorithms used in neural network training.

3. Understand the Toolbox/Framework Features:

- Explore the documentation and tutorials provided by the chosen toolbox/framework to understand its features and functionalities.
- Learn about the APIs for defining neural network architectures, specifying layers, configuring training parameters, etc.
- Understand the compatibility with different hardware platforms (CPU, GPU, TPU) and programming languages (Python, C++, etc.).

4. Hands-on Practice:

- Start with simple examples and tutorials provided by the toolbox/framework documentation to get hands-on experience.

- Implement basic neural network models (e.g., feedforward networks for classification tasks) using the toolbox/framework.
- Experiment with different network architectures, hyperparameters, and optimization strategies to understand their effects on model performance.

5. Explore Advanced Topics:

- Dive deeper into advanced topics such as transfer learning, model interpretation, hyperparameter tuning, etc.
- Explore specialized architectures like recurrent neural networks (RNNs), long short-term memory (LSTM) networks, generative adversarial networks (GANs), etc.
- Learn about deployment options for deploying trained models in production environments.

6. Participate in Projects and Competitions:

- Join online communities, forums, or social media groups related to the chosen toolbox/framework to interact with other users and experts.
- Participate in machine learning projects, competitions (e.g., Kaggle), or open-source contributions to gain practical experience and exposure to real-world problems.

7. Stay Updated:

- Stay updated with the latest developments, releases, and advancements in the chosen toolbox/framework by following official announcements, blogs, research papers, etc.
- Continuously expand your knowledge and skills by exploring new features, techniques, and best practices in neural network development.

Experiment 2: Study of MATLAB functions

MATLAB functions involves understanding how to use built-in functions, creating your own functions, and exploring various functionalities offered by MATLAB's vast library. Below are some key aspects to consider when studying MATLAB functions:

1. Built-in Functions:

- Familiarize yourself with commonly used built-in functions in MATLAB, such as **plot**, **sum**, **mean**, **max**, **min**, **find**, **sort**, etc.
- Explore MATLAB's documentation or help resources to understand the syntax, usage, and options available for each function.
- Practice using built-in functions in different scenarios to manipulate data, perform calculations, plot graphs, and solve problems.

2. Function Syntax:

- Understand the syntax for calling MATLAB functions, including input arguments and output variables.
- Learn how to interpret function documentation to understand the purpose of each input argument and the expected output.
- Pay attention to optional arguments and how to specify them when calling functions.

3. Creating Your Own Functions:

- Learn how to create custom MATLAB functions using the **function** keyword.
- Understand the structure of a MATLAB function file, including the function signature, input/output arguments, and the body of the function.
- Practice writing simple functions to perform specific tasks or calculations.

4. Function Handles and Anonymous Functions:

- Explore function handles and how to create them using **@** symbol to reference functions by their names.
- Learn about anonymous functions, which are small, unnamed functions defined in a single line using the **@(arguments) expression** syntax.

5. Vectorization and Element-wise Operations:

- Understand MATLAB's vectorized operations, which allow you to perform operations on entire arrays or matrices efficiently without using explicit loops.
- Explore element-wise operations using **.***, **./**, **.^**, etc., to perform element-wise multiplication, division, exponentiation, etc.

6. MATLAB Toolboxes and Libraries:

- Explore MATLAB's extensive collection of toolboxes for specialized tasks such as signal processing, image processing, optimization, statistics, etc.

- Learn how to use functions and tools provided by different toolboxes to solve specific problems or tasks.

7. Debugging and Troubleshooting:

- Practice debugging MATLAB code by using built-in debugging tools such as breakpoints, stepping through code, and inspecting variables.
- Learn how to interpret error messages and troubleshoot common issues encountered while writing or running MATLAB functions.

8. Optimization and Performance:

- Explore techniques for optimizing MATLAB code for improved performance, such as preallocating arrays, vectorizing operations, and using built-in optimization functions.
- Learn about MATLAB's profiling tools for identifying bottlenecks and optimizing code execution time.

9. Practice and Projects:

- Regularly practice writing MATLAB functions to reinforce your understanding and improve your programming skills.
- Work on projects or solve problems using MATLAB functions to apply what you've learned in practical scenarios.

Experiment 3: To perform basic matrix operations

To perform basic matrix operations in MATLAB, you can use built-in functions or operators. Here's a brief overview of some common matrix operations along with MATLAB code examples:

1. Addition and Subtraction: Perform element-wise addition/subtraction between matrices with the same dimensions.

```
% Define matrices
A = [1, 2, 3; 4, 5, 6];
B = [1, 2, 3; 4, 5, 6];
% Addition
```

```
>> C = A + B

C =

     2     4     6
     8    10    12
```

2. Scaler Multiplication: Multiply all elements of a matrix by a scalar (number).

```
>> D = 3 * A

D =

     3     6     9
    12    15    18
```

3. Transpose: Use the single quota " ' " operator to find the transpose (swapping rows and columns)

```
>> E = A'

E =

     1     4
     2     5
     3     6
```

4. Element-wise Multiplication and Division: Use the dot multiplication operator " .* " to multiply corresponding elements of two matrices with the same directions. To performs division between corresponding elements of two matrices with the same dimensions use " ./ "

```
A = [1, 2; 3, 4];
B = [5, 6; 7, 8];
% Element-wise multiplication
>> D = A.*B

D =

     5    12
    21    32
% Element-wise division
>> E = A./B

E =

    0.2000    0.3333
    0.4286    0.5000
```

5. Inverse: It uses the inv(A) function to calculate the inverse and stores the result in the variable. This function only works for square matrices (number of rows equals number of columns). If you provide a non-square matrix, MATLAB will throw an error. If the matrix is singular (determinant is 0), then it does not have and inverse

```
>> F = inv(A)

F =

   -2.0000    1.0000
    1.5000   -0.5000
```

6. Determinant: It uses the det(A) function to calculate the determinant. This will work for square matrices only. If you provide a non-square matrix, MATLAB will throw an error

```
>> G = det(A)

G =

    -2
```

Experiment 4: To plot the following

To plot a straight line and a sine curve in MATLAB, you can use the **plot** function. Below are examples demonstrating how to plot each:

(a) Plotting a Straight Line:

1. Define slope (m) and y-intercept (c): These values determine the equation of the line (y = mx + c). Change m and c to adjust the slope and y-intercept of your line.

2. Define x-axis range (linspace): The linspace function creates a linearly spaced vector 100 points between 0 and 10 (you can adjust these values). This represents the x-axis values of your plot.

3. Calculate y-values: The y vector is calculated using the line equation (y = mx + c) for each corresponding x-value in the x vector.

4. Plot the line: The plot(x, y) command creates the line plot using the x and y values we calculated. 5. Labels and Title: These lines add informative labels to the x and y-axes and a title to your plot.

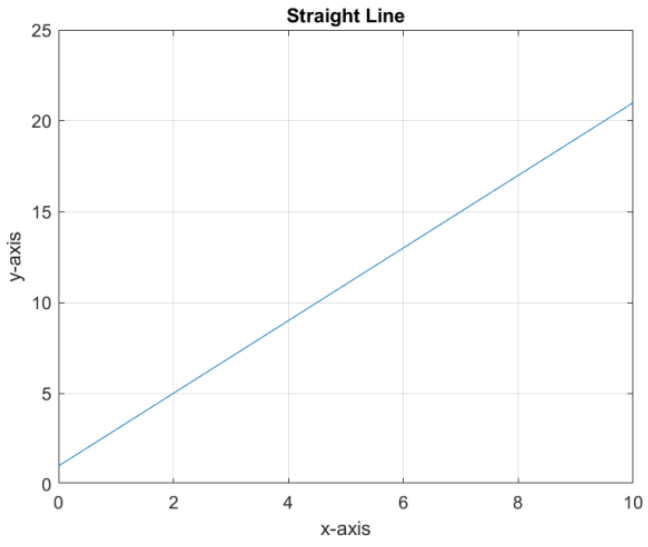
6. Grid (optional): The grid on command adds gridlines to your plot for better readability.

7. Show the plot: This line executes the plot and displays it in the MATLAB figure window.

Program:

```
>> m = 2;
>> c = 1;
>> x = linspace(0, 10, 100);
>> y = m*x + c; >> plot(x, y)
>> xlabel('x-axis')
>> ylabel('y-axis')
>> title("Straight Line")
>> grid on
```

Output:



(b) Plotting a Sine Curve:

1. sin(x): This calculates the sine of each element in the x vector, resulting in the corresponding y-values for the sine curve

2. plot(x, y) is used to create the plot, where **x** represents the x-coordinates and **y** represents the corresponding y-coordinates.

3. 'b-' or **'r-'** specifies the color and line style of the plot. **'b-'** represents a blue solid line, and **'r-'** represents a red solid line.

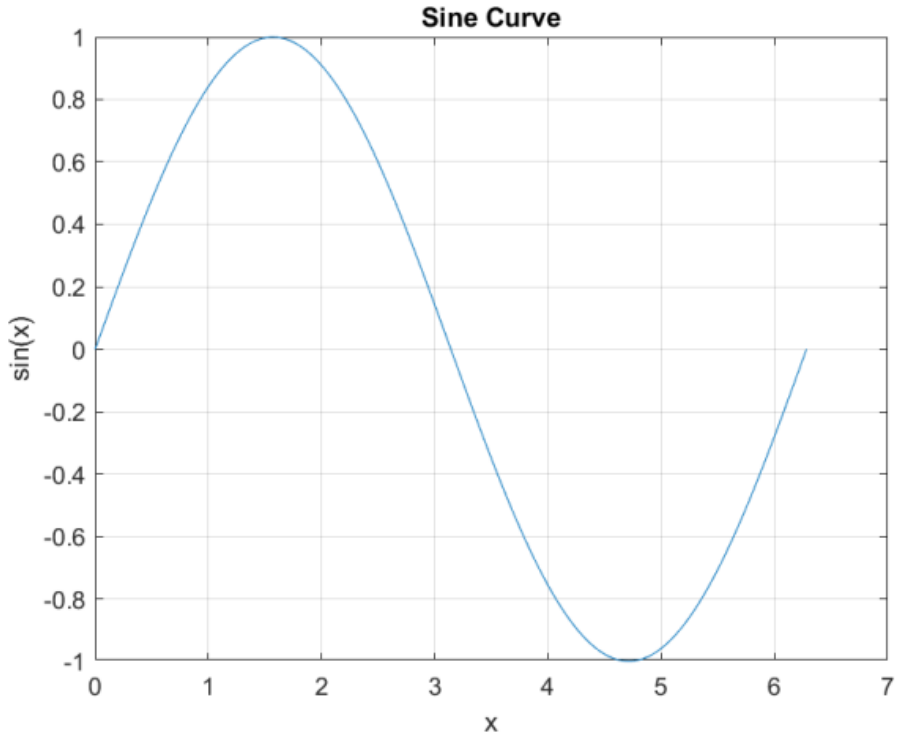
4. xlabel, **ylabel**, and **title** are used to label the x-axis, y-axis, and give the plot a title, respectively.

5. grid on is used to display a grid on the plot.

Program:

```
>> x = linspace(0, 2*pi, 200);
>> y = sin(x);
>> plot(x, y);
>> xlabel('x')
>> ylabel('sin(x)')
>> title("Sine Curve")
>> grid on
```

Output:

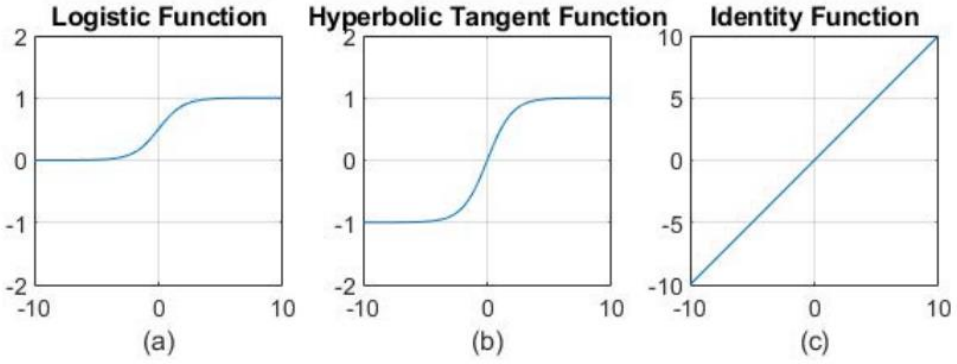


Experiment 5: To generate a few activation functions that is used in neural networks in MATLAB.

Here are implementations of a few commonly used activation functions in neural networks, implemented in MATLAB:

```
x = -10:0.1:10;
tmp = exp(-x);
y1 = 1./(1+tmp);
y2 = (1-tmp)./(1+tmp);
y3 = x;
subplot(231); plot(x, y1);grid on;
axis([min(x) max(x) -2 2]);
title('Logistic Function');
xlabel('(a)');
axis('square');
subplot(232); plot(x, y2); grid on;
axis([min(x) max(x) -2 2]);
title('Hyperbolic Tangent Function');
xlabel('(b)');
axis('square');
subplot(233); plot(x, y3); grid on;
axis([min(x) max(x) min(x) max(x)]);
title('Identity Function');
xlabel('(c)');
axis('square')
```

Output:



Experiment 6: To plot hard limit transfer function.

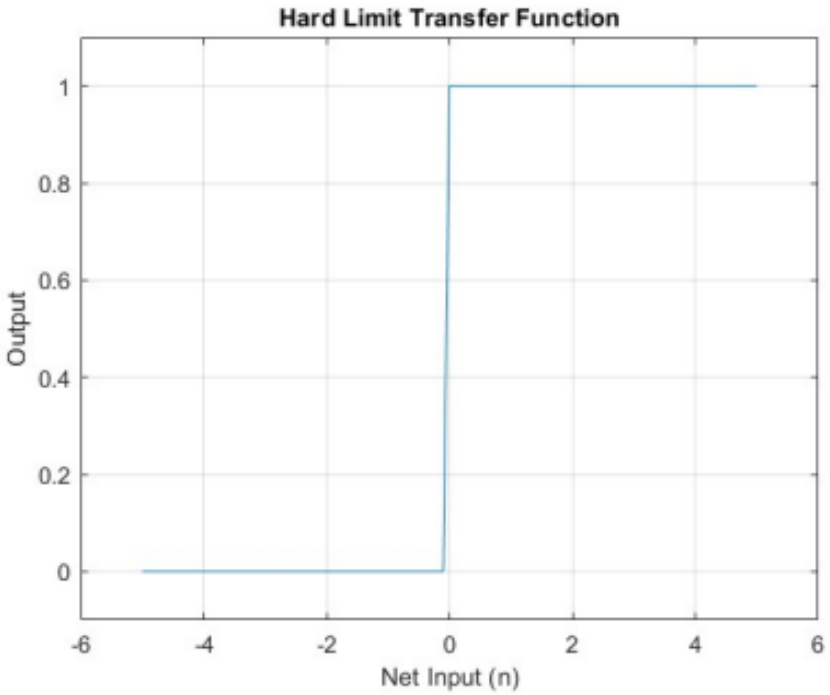
The hard limit transfer function simply converts any input to either 0 or 1 based on a threshold.
The hard limit function actually outputs:

- 1 if the net input (n) is greater than zero (0).
- 0 otherwise (including negative inputs and zero itself)

Program:

```
n = -5:0.1:5;
y = hardlim(n);
plot(n, y);
xlabel('Net Input (n)');
ylabel('Output');
title('Hard Limit Transfer Function');
xlim([-6 6]);
ylim([-0.1 1.1]);
grid on;
```

Output:



Experiment 7: To generate XOR function using McCullock Pitts Neural Network in MATLAB

Creating a McCulloch-Pitts neural network for XOR function in MATLAB involves defining the architecture of the network and implementing the logic gates for the neurons. Here's how you can do it:

Program:

Create the file name "mcculloch_pitts_xor.m" in MATLAB Current Folder and write the code in that file:

```
function output = mcculloch_pitts_xor(x1, x2)
% Define weights and thresholds
w11 = 1; % Weight for x1 to hidden neuron 1
w12 = 1; % Weight for x2 to hidden neuron 1
b1 = -1.5; % Threshold for hidden neuron 1
w21 = 1; % Weight for x1 to hidden neuron 2
w22 = 1; % Weight for x2 to hidden neuron 2
b2 = -1.5; % Threshold for hidden neuron 2
w31 = -1; % Weight for hidden neuron 1 to output
w32 = -1; % Weight for hidden neuron 2 to output
b3 = -0.5; % Threshold for output neuron

% Hidden Neuron
y1 = step_function(w11*x1 + w12*x2 + b1);
y2 = step_function(w21*x1 + w22*x2 + b2);

% Output neuron
output = step_function(w31*y1 + w32*y2 + b3);
end

function output = step_function(x)
   output = x >= 0;
end
```

Execute the code in the command window:

```
% Test XOR function
fprintf('XOR(0,0) = %d\n', mcculloch_pitts_xor(0,0));
fprintf('XOR(0,1) = %d\n', mcculloch_pitts_xor(0,1));
fprintf('XOR(1,0) = %d\n', mcculloch_pitts_xor(1,0));
fprintf('XOR(1,1) = %d\n', mcculloch_pitts_xor(1,1));
```

Output:

```
XOR(0,0) = 0
XOR(0,1) = 0
XOR(1,0) = 0
XOR(1,1) = 0
```

Experiment 8: To generate AND-NOT function using McCullock Pitts Neural Network in MATLAB.

To implement the AND-NOT function using the McCulloch-Pitts neural network in MATLAB, you need to define the architecture of the network and set appropriate weights and thresholds. In this implementation:
Neuron 1 implements the AND gate, which takes inputs x1 and x2.
Neuron 2 implements the NOT gate, which takes the output of the AND gate as input.
The weights and thresholds are set manually for simplicity. The step_function is used as the activation function.
You can test the AND-NOT function by calling mcculloch_pitts_and_not with different input combinations.

Program:

```
function output = mcculloch_pitts_and_not(x1, x2)
    % Define weights and thresholds
    w1 = -1; % Weight for x1
    w2 = -1; % Weight for x2
    b = 1;   % Bias

 % Neuron 1 (AND gate)
 y1 = step_function(w1*x1 + w2*x2 + b);

% Neuron 2 (NOT gate)
y2 = step_function(-2*y1 + 1); % NOT gate using -2*y1 + 1

% Output neuron
output = y2;
end

function output = step_function(x)
    output = x >= 0;
end

% Test AND-NOT function
fprintf('AND-NOT(0,0) = %d\n', mcculloch_pitts_and_not(0,0));
fprintf('AND-NOT(0,1) = %d\n', mcculloch_pitts_and_not(0,1));
fprintf('AND-NOT(1,0) = %d\n', mcculloch_pitts_and_not(1,0));
fprintf('AND-NOT(1,1) = %d\n', mcculloch_pitts_and_not(1,1));
```

Output:

```
AND-NOT(0,0) = 0
AND-NOT(0,1) = 0
AND-NOT(1,0) = 0
AND-NOT(1,1) = 1
```

Experiment 9: To use Hebbian Network to classify 2-Dimentional input pattern.

Hebbian learning is a type of unsupervised learning that strengthens the connection between neurons that fire simultaneously. In this code:
patterns contains the input patterns. Each row represents a pattern, and each column represents a dimension.

weights is the weight matrix between input neurons and output neurons. It's initialized randomly.

learning_rate is the rate at which the weights are updated during learning.

The network is trained using Hebbian learning, where the weights are updated based on the product of the input pattern and the activations.

Finally, the network is tested by computing the activations for each input pattern.

Here's a basic implementation of a Hebbian network for classifying 2-dimensional input patterns in MATLAB:

Program:

```
% Define input patterns
patterns = [
 [0 0]; % Class 1 (pattern 1)
 [0 1]; % Class 1 (pattern 2)
 [1 0]; % Class 2 (pattern 1)
 [1 1]; % Class 2 (pattern 2)
];

% Number of inputs and neurons (adjust based on your data)
num_inputs = size(patterns, 2);
num_neurons = 2; % Two neurons for two classes

% Initialize weights randomly
weights = rand(num_neurons, num_inputs);
% Define learning rate
learning_rate = 0.1;

% Train the network using Hebbian learning
for i = 1:size(patterns, 1)
```

```
    % Compute activations (no activation function for Hebbian learning)
    activations = patterns(i, :) * weights';

    % Update weights based on Hebbian rule (outer product)
    weight_updates = learning_rate * activations' * patterns(i, :);
    weights = weights + weight_updates;
end

% Test the network
for i = 1:size(patterns, 1)
    % Compute activations
    activations = patterns(i, :) * weights';

    % Find the winning neuron (highest activation)
    [~, winning_neuron] = max(activations);
    % Print classification result (replace with class labels if applicable)
    fprintf('Pattern %d: Classified as Class %d\n', i, winning_neuron);
end
```

Output:

```
Pattern 1: Classified as Class 1
Pattern 2: Classified as Class 2
Pattern 3: Classified as Class 2
Pattern 4: Classified as Class 2
```

Experiment 10: Write a MATLAB program for perceptron net for and function with bipolar input and targets.

patterns contains the bipolar input patterns. Each row represents a pattern, and each column represents a dimension.
targets contains the bipolar target outputs.
We initialize the weights randomly.
The perceptron network is trained using the perceptron learning rule for a specified number of epochs.
We then test the trained network and print out the inputs, targets, and outputs.

Program:

```
% Define bipolar input patterns (4 patterns with 2 dimensions each)
patterns = [-1 -1; -1 1; 1 -1; 1 1];

% Define bipolar target outputs
targets = [-1; -1; -1; 1];

% Initialize weights randomly
num_inputs = size(patterns, 2);
weights = rand(num_inputs, 1);

% Define learning rate
learning_rate = 0.1;

% Define number of epochs
num_epochs = 100;

% Train the perceptron network
for epoch = 1:num_epochs
   for i = 1:size(patterns, 1)
      % Compute the activation of the perceptron
      activation = patterns(i, :) * weights;

      % Compute the error
      error = targets(i) - sign(activation);

      % Update weights
      weights = weights + learning_rate * error * patterns(i, :)';
   end
end

% Test the perceptron network
fprintf('Trained weights: %s\n', mat2str(weights'));

% Evaluate trained network
```

```
fprintf('\nEvaluation:\n');
for i = 1:size(patterns, 1)
    % Compute the activation of the perceptron
    activation = patterns(i, :) * weights;

    % Compute the output
    output = sign(activation);

    fprintf('Input: %s, Target: %d, Output: %d\n', mat2str(patterns(i, :)), targets(i),
output);
end
```

Output:

```
Trained weights: [0.07692298496089 0.246171390631154]

Evaluation:
Input: [-1 -1], Target: -1, Output: -1
Input: [-1 1], Target: -1, Output: 1
Input: [1 -1], Target: -1, Output: -1
Input: [1 1], Target: 1, Output: 1
```