Perceptron:

Perceptron is a type of artificial neural network that is used for binary classification problems. It is a single-layer neural network that takes in a set of inputs, applies weights to those inputs, and then outputs a binary value indicating whether the input belongs to a particular class or not.

The perceptron learning algorithm involves iteratively updating the weights until the algorithm converges and correctly classifies all of the training data. The algorithm takes in a set of labeled training data, where each training sample consists of a set of inputs and a binary class label. It then initializes the weights to random values and iterates over the training data, adjusting the weights for each misclassified sample until all of the training data is correctly classified.

Algorithm:

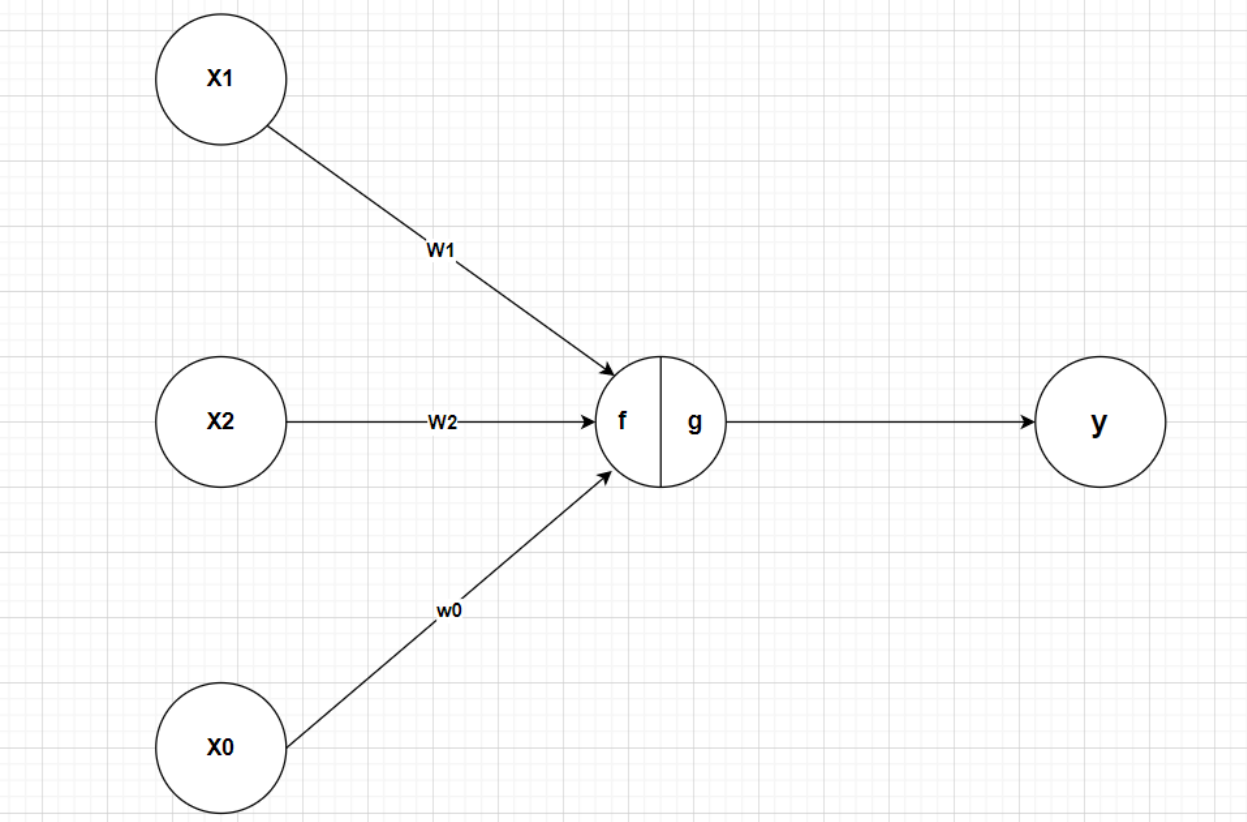
The algorithm works by iteratively updating a set of weights based on the input data and the known output classes, until all samples are correctly classified.

The algorithm can be summarized as follows:

- *Initialize the weight vector w to some arbitrary value.*
- *For each sample in the input data:*
- *If the sample is correctly classified (i.e., its predicted class matches its actual class), continue to the next sample.*
- *If the sample is misclassified:*
- *Update the weight vector by adding or subtracting the sample (depending on its actual class), multiplied by a learning rate.*
- *Repeat step 2 until all samples are correctly classified.*
- *Output the final weight vector.*

The code starts by defining the input data (x1 and x2) and their corresponding classes (y), which are combined into a single dataset. The mean of the input data is then subtracted from each sample to center the data around the origin, which helps improve the convergence of the algorithm.

The code then defines a function to plot the data points and decision boundary. The weight vector is initialized to [1, 1, 1], and the graph is plotted using this initial weight vector.
The algorithm then enters a loop that continues until all samples are correctly classified. For each iteration, the algorithm goes through each sample in the input data and checks if it is classified correctly. If it is, the algorithm continues to the next sample. If it is misclassified, the weight vector is updated by adding or subtracting the sample, depending on its actual class.
If the algorithm makes an update to the weight vector during an iteration, the graph is plotted using the updated weight vector. Once all samples are correctly classified, the final weight vector is outputted.



Calculation Step by step:

Input:

x1	x2	Class
1	1	1
-1	-1	-1
0	0.5	-1
0.1	0.5	-1
0.2	0.2	1
0.9	0.5	1

In perceptron learning, the goal is to find a linear decision boundary that can correctly classify the input data into two classes. The perceptron learning algorithm adjusts the weights of the linear function to minimize the error between the predicted and actual class labels.

The error function used in perceptron learning is the mean squared error (MSE), which measures the average squared difference between the predicted and actual class labels. This is because the MSE is a differentiable function that allows for the use of gradient descent optimization, which is used to adjust the weights of the linear function in order to minimize the error.

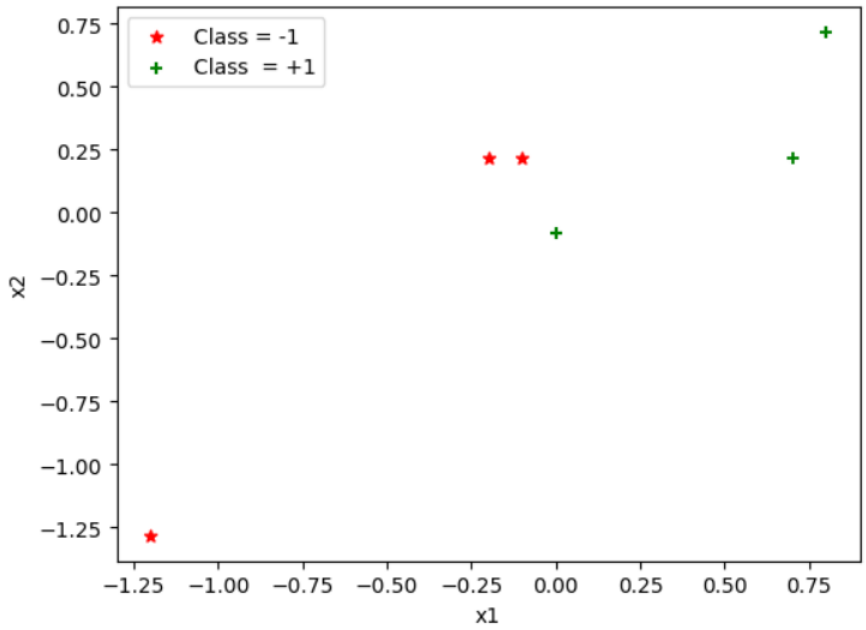
In summary, the use of mean squared error in perceptron learning allows for the optimization of the linear function weights using gradient descent, which helps to minimize the error between predicted and actual class labels.

MEAN SQUARED DATA :

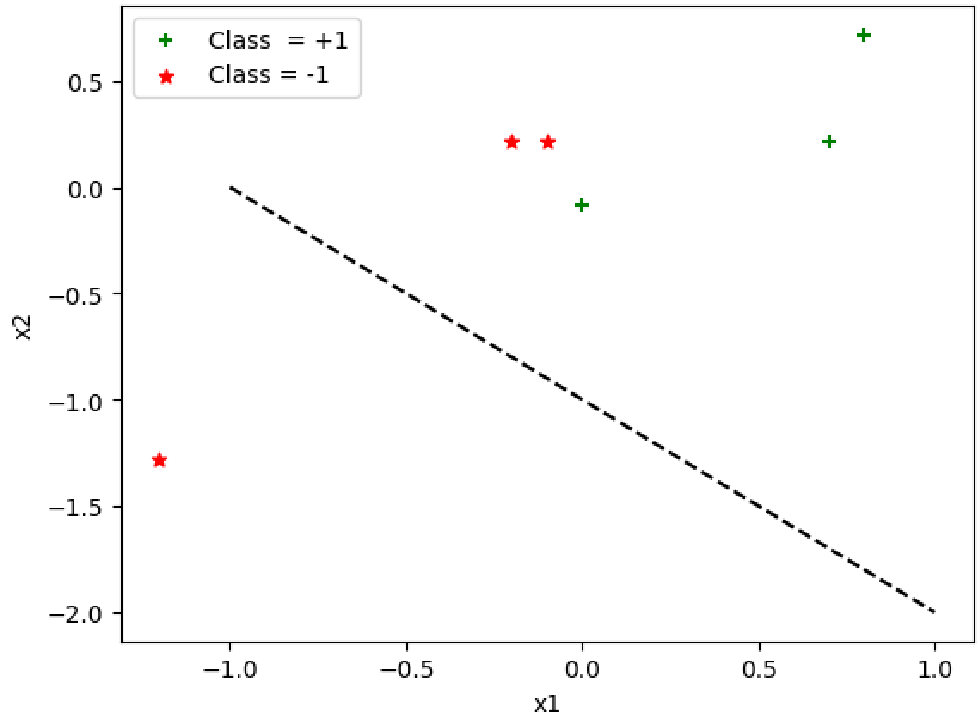
x1	x2	Class
0.8	0.716667	1
-1.2	-1.28333	-1
-0.2	0.216667	-1
-0.1	0.216667	-1

-2.77556e-17	-0.0833333	1
0.7	0.216667	1

plots a scatter plot of two-dimensional data points, where each point is colored and marked according to its class label. The x and y coordinates of the data points are stored in separate arrays. The code loops through the unique class labels and plots the points belonging to each class with different colors and symbols. The resulting plot has labeled axes and a legend indicating which class corresponds to each color and symbol.



Let us try to draw the decision boundary W^T X = 0 and consider the initial weights are [1,1,1] and begin the calculation



Iteration: 1 The Weights are : [1, 1, 1]

=======================================

X : [1. 0.8 0.71666667]

W^TX = 2.5166666666666666

Classified Correctly

****

X : [1. -1.2 -1.28333333]

W^TX = -1.4833333333333332

Classified Correctly

****

X : [1. -0.2 0.21666667]

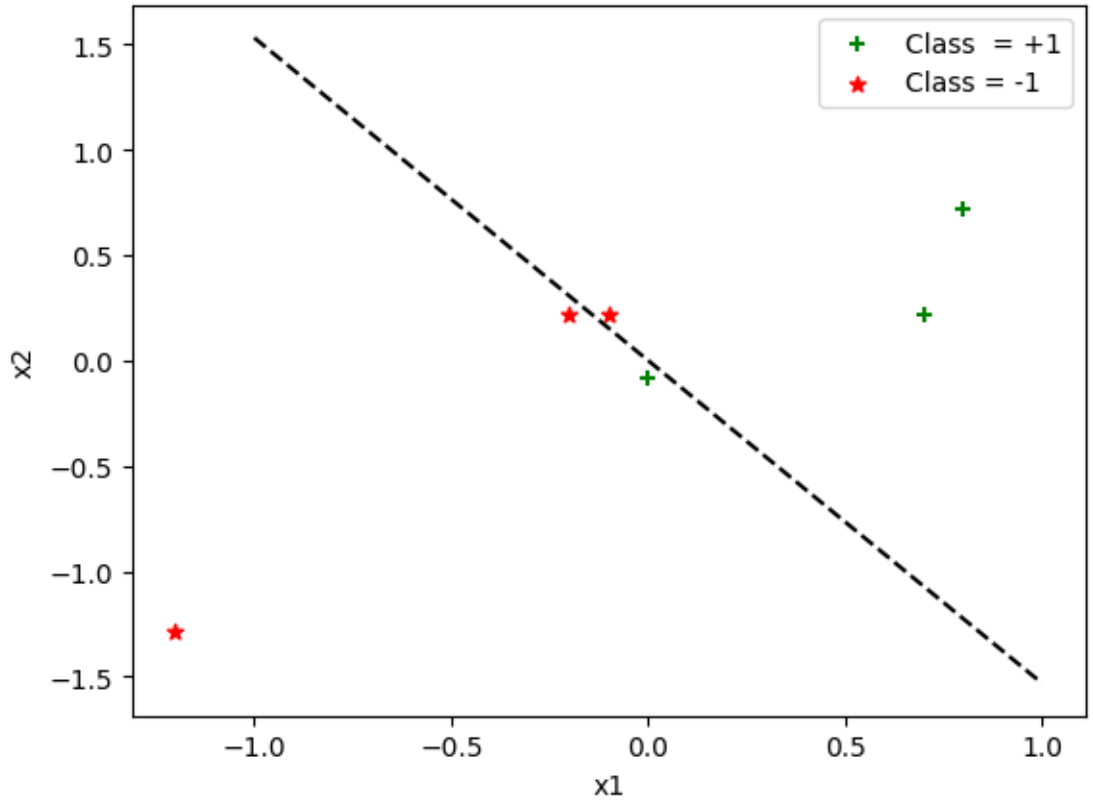
W^TX = 1.0166666666666666

MisClassified

****

w = w - x

The Updated Weights are : [0. 1.2 0.78333333]



Iteration: 2 The Weights are : [0. 1.2 0.78333333]

==========

X : [1. 0.8 0.71666667]

W^TX = 1.5213888888888887

Classified Correctly

****

X : [1. -1.2 -1.28333333]

W^TX = -2.4452777777777777

Classified Correctly

****

X : [1. -0.2 0.21666667]

W^TX = -0.07027777777777783

Classified Correctly

****

X : [1. -0.1 0.21666667]

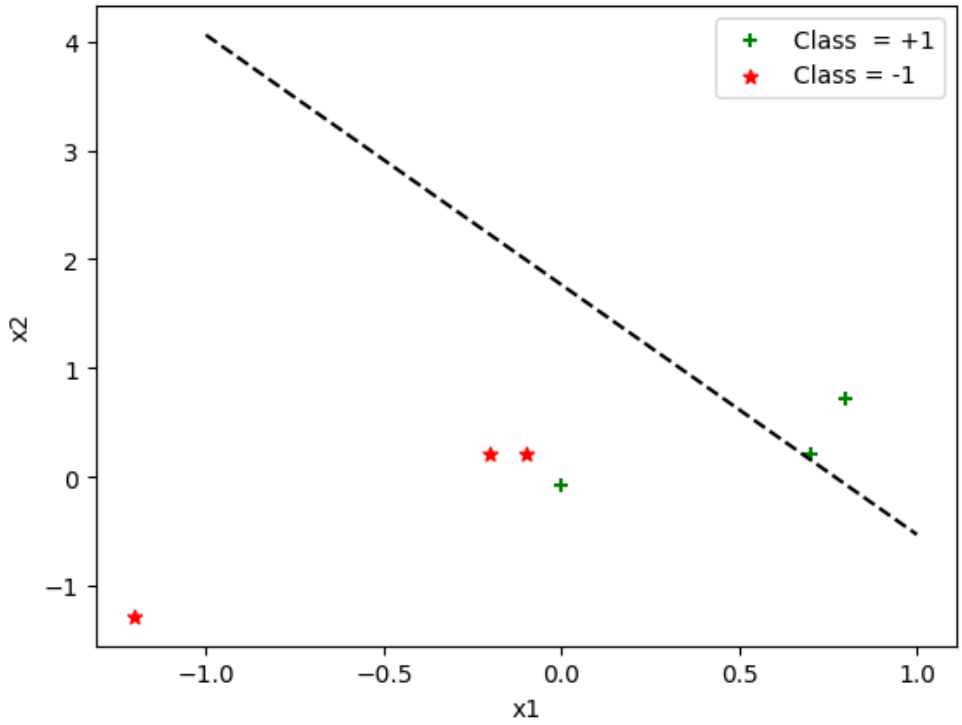
W^TX = 0.04972222222222218

MisClassified

****

w = w - x

The Updated Weights are : [-1. 1.3 0.56666667]



Iteration: 3 The Weights are : [-1. 1.3 0.56666667]

==========

X : [1. 0.8 0.71666667]

W^TX = 0.4461111111111111

Classified Correctly

****

X : [1. -1.2 -1.28333333]

W^TX = -3.2872222222222223

Classified Correctly

****

X : [1. -0.2 0.21666667]

W^TX = -1.1372222222222221

Classified Correctly

****

X : [1. -0.1 0.21666667]

W^TX = -1.0072222222222222

Classified Correctly

****

X : [1.00000000e+00 -2.77555756e-17 -8.33333333e-02]

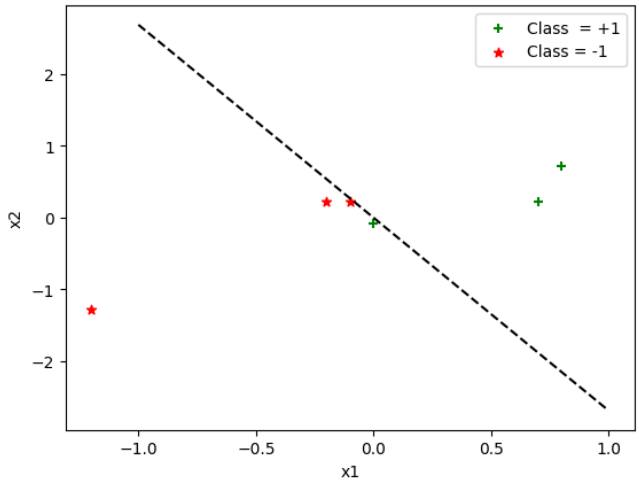
W^TX = -1.0472222222222223

MisClassified

****

w = w + x

The Updated Weights are : [0. 1.3 0.48333333]



Iteration: 4 The Weights are : [0. 1.3 0.48333333]

==

X : [1. 0.8 0.71666667]

W^TX = 1.3863888888888889

Classified Correctly

****

X : [1. -1.2 -1.28333333]

W^TX = -2.1802777777777775

Classified Correctly

****

X : [1. -0.2 0.21666667]

W^TX = -0.15527777777777785

Classified Correctly

****

X : [1. -0.1 0.21666667]

W^TX = -0.025277777777777783

Classified Correctly

****

X : [1.00000000e+00 -2.77555756e-17 -8.33333333e-02]

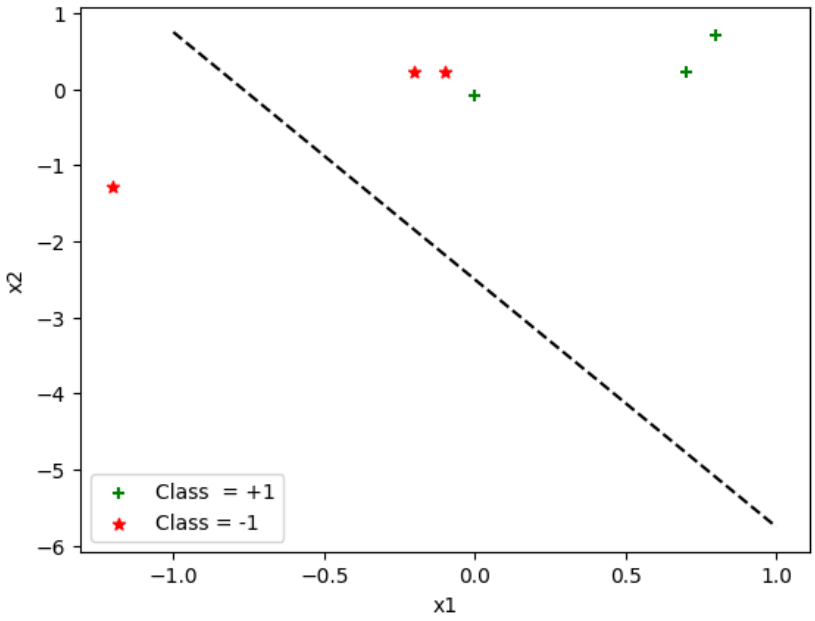
W^TX = -0.0402777777777778

MisClassified

****

w = w + x

The Updated Weights are : [1. 1.3 0.4]



Iteration: 5 The Weights are : [1. 1.3 0.4]

==========

X : [1. 0.8 0.71666667]

W^TX = 2.3266666666666667

Classified Correctly

****

X : [1. -1.2 -1.28333333]

W^TX = -1.0733333333333333

Classified Correctly

****

X : [1. -0.2 0.21666667]

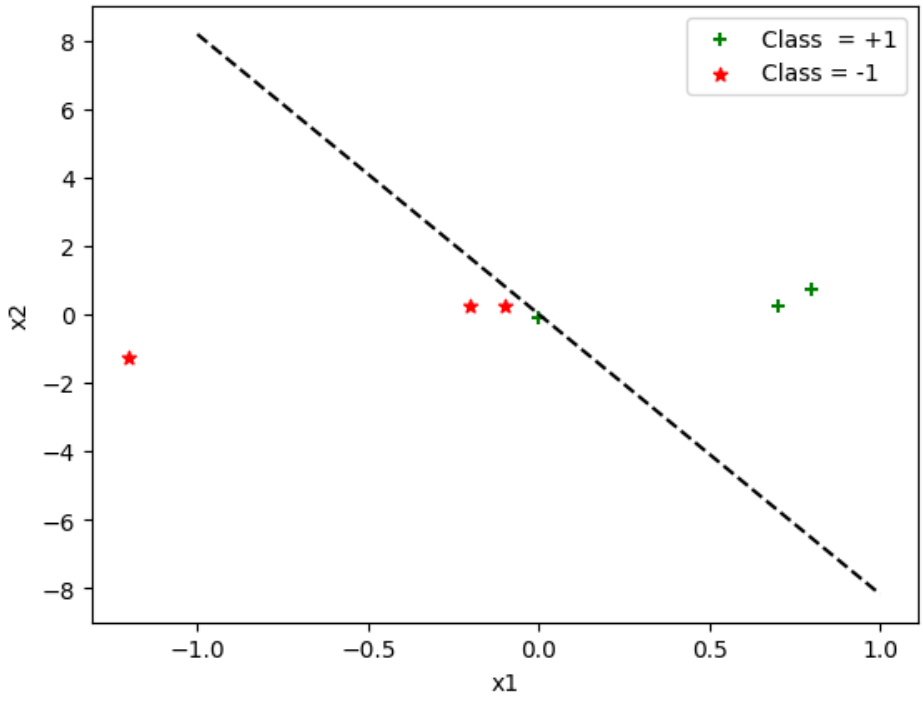
W^TX = 0.8266666666666667

MisClassified

****

w = w - x

The Updated Weights are : [0. 1.5 0.18333333]



Iteration: 6 The Weights are : [0. 1.5 0.18333333]

==========

X : [1. 0.8 0.71666667]

W^TX = 1.3313888888888887

Classified Correctly

****

X : [1. -1.2 -1.28333333]

W^TX = -2.0352777777777775

Classified Correctly

****

X : [1. -0.2 0.21666667]

W^TX = -0.26027777777777783

Classified Correctly

****

X : [1. -0.1 0.21666667]

W^TX = -0.11027777777777782

Classified Correctly

****

X : [1.00000000e+00 -2.77555756e-17 -8.33333333e-0

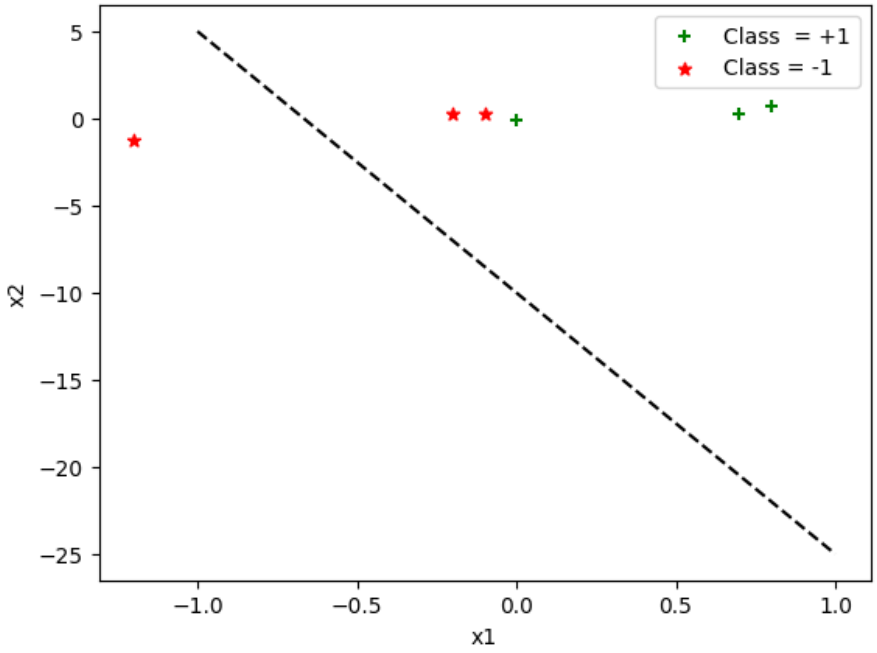
W^TX = -0.015277777777777817

MisClassified

****

w = w + x

The Updated Weights are : [1. 1.5 0.1]



Iteration: 7 The Weights are : [1. 1.5 0.1]

==========

X : [1. 0.8 0.716666

W^TX = 2.271666666666667

Classified Correctly

****

X : [1. -1.2 -1.28333333]

W^TX = -0.9283333333333332

Classified Correctly

****

X : [1. -0.2 0.21666667]

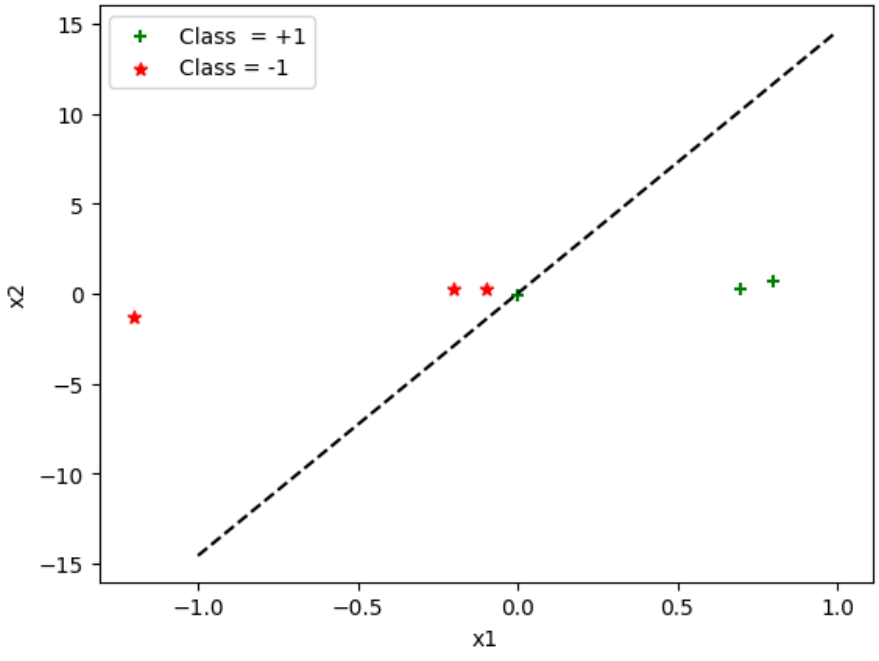
W^TX = 0.7216666666666667

MisClassified

****

w = w - x

The Updated Weights are : [0. 1.7 -0.11666667]



Iteration: 8 The Weights are : [0. 1.7 -0.11666667]

==========

X : [1. 0.8 0.71666667]

W^TX = 1.2763888888888888

Classified Correctly

****

X : [1. -1.2 -1.28333333]

W^TX = -1.890277777777778

Classified Correctly

****

X : [1. -0.2 0.21666667]

W^TX = -0.36527777777777787

Classified Correctly

****

X : [1. -0.1 0.21666667]

W^TX = -0.19527777777777783

Classified Correctly

****

X : [1.00000000e+00 -2.77555756e-17 -8.33333333e-02]

W^TX = 0.00972222222222217

Classified Correctly

****

X : [1. 0.7 0.21666667]

W^TX = 1.1647222222222222

Classified Correctly

****

Final weights: [0. 1.7 -0.11666667]

Note that in this algorithm, we don't use bias, which means that the decision boundary passes through the origin. Also, we don't use an error term to update the weights; instead, we update the weights based on the predicted and actual output values.

At the start of the algorithm, the weights are initialized to zero or a small random value. Then, we loop through the training data points and make predictions using the current weights. If a point is correctly classified, we move on to the next point. If a point is misclassified, we update the weights by adding or subtracting the features of the point, depending on whether it is a positive or negative point.

The intuition behind the weight updates is that we are moving the decision boundary in the direction of the misclassified point. If a positive point is misclassified as negative, then we add its features to the weights vector, which will shift the decision boundary towards the positive points. Similarly, if a negative point is misclassified as positive, then we subtract its features from the weights vector, which will shift the decision boundary towards the negative points.

Perceptron:

Given:

x_1	x_2	class
1	1	+1
-1	-1	-1
0	0.5	-1
0.1	0.5	-1
0.2	0.2	+1
0.9	0.5	+1

Mean of x_1: $\frac{1 - 1 + 0 + 0.1 + 0.2 + 0.9}{6} = 0.2$

mean of x_2: $\frac{1 - 1 + 0.5 + 0.5 + 0.2 + 0.5}{6} = 0.2833333$

Mean Centered datapoints

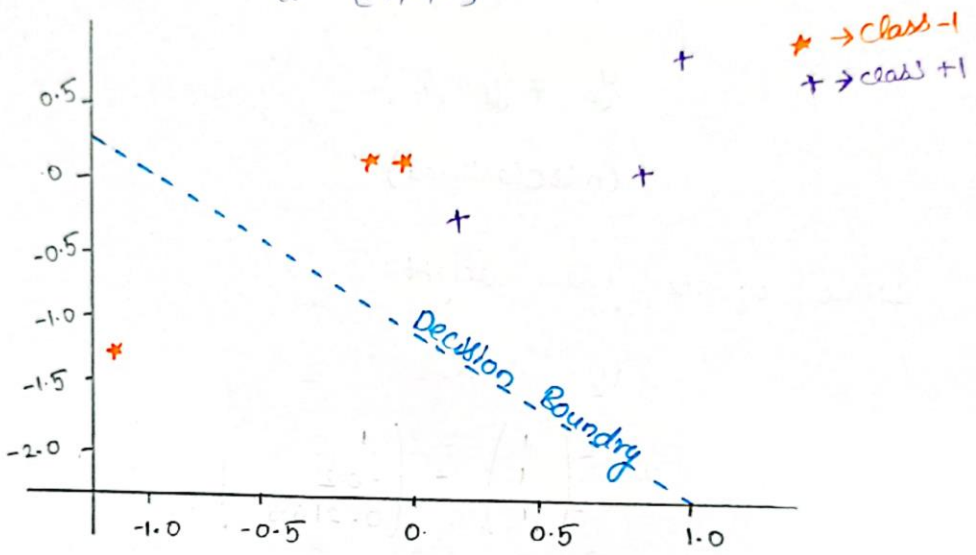
x_1	x_2	classes
0.8	0.716667	1
-1.2	-1.2833333	-1
-0.2	0.216667	-1
-0.1	0.216667	-1
0	-0.0833333	-1
0.7	0.216667	1

$w^T x = 0$

$w = [1, 1]$

Let us consider $w_0 = 1$ and $x_0 = 1$

$w = [1, 1, 1]$



Situation-1:-

(1) $w^T x \Rightarrow (1 \quad 1 \quad 1) \begin{pmatrix} 1 \\ 0.8 \\ 0.71666667 \end{pmatrix}$

$\Rightarrow 2.51667$

$y_a = y_p$ (classified correctly)

(2) $w^T x \Rightarrow (1 \quad 1 \quad 1) \begin{pmatrix} 1 \\ -1.2 \\ -1.2833333 \end{pmatrix}$

$\Rightarrow -1.48333$

$y_a = y_p$ (classified correctly)

(3) $w^T x \Rightarrow \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} +1. \\ -0.2 \\ 0.216667 \end{pmatrix}$

$\Rightarrow 1.016667$

$y_a \neq y_p$

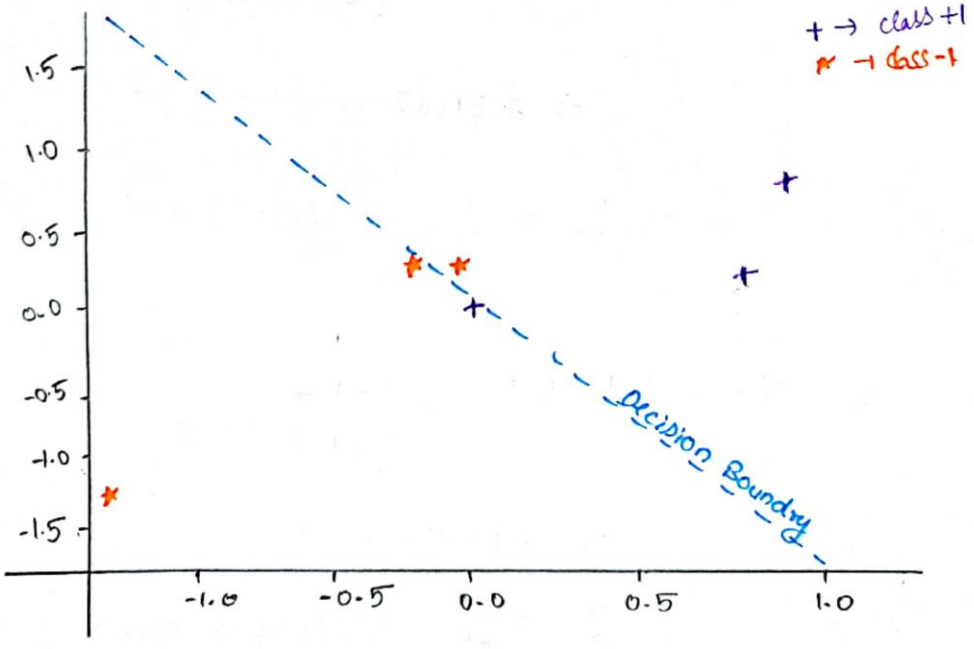
(misclassified)

Let us update the weight

$$w = w - x$$

$$w = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 1 \\ -0.2 \\ 0.216667 \end{pmatrix}$$

$$w = \begin{pmatrix} 0 \\ 1.2 \\ 0.78333 \end{pmatrix}$$



Iteration 2:

(1) $w^T x \Rightarrow \begin{pmatrix} 0 \\ 1.2 \\ 0.78333 \end{pmatrix}^T \begin{pmatrix} 1 \\ 0.8 \\ 0.7166667 \end{pmatrix}$

$\Rightarrow 1.512389$

$y_a = y_p$

correctly classified

(2) $w^T x \Rightarrow (0 \quad 1.2 \quad 0.78333) \begin{pmatrix} 1 \\ -1.2 \\ -1.2833333 \end{pmatrix}$

$\Rightarrow -2.445278$

$y_a = y_p$

Correctly classified

(3) $w^T x \Rightarrow (0 \quad 1.2 \quad 0.78333) \begin{pmatrix} 1 \\ -0.2 \\ 0.216667 \end{pmatrix}$

$\Rightarrow -0.070278$

$y_a = y_p$

Correctly classified

(4) $w^T x \Rightarrow (0 \quad 1.2 \quad 0.78333) \begin{pmatrix} 1 \\ -0.1 \\ 0.216667 \end{pmatrix}$

$\Rightarrow 0.04972$

$y_a \neq y_p$

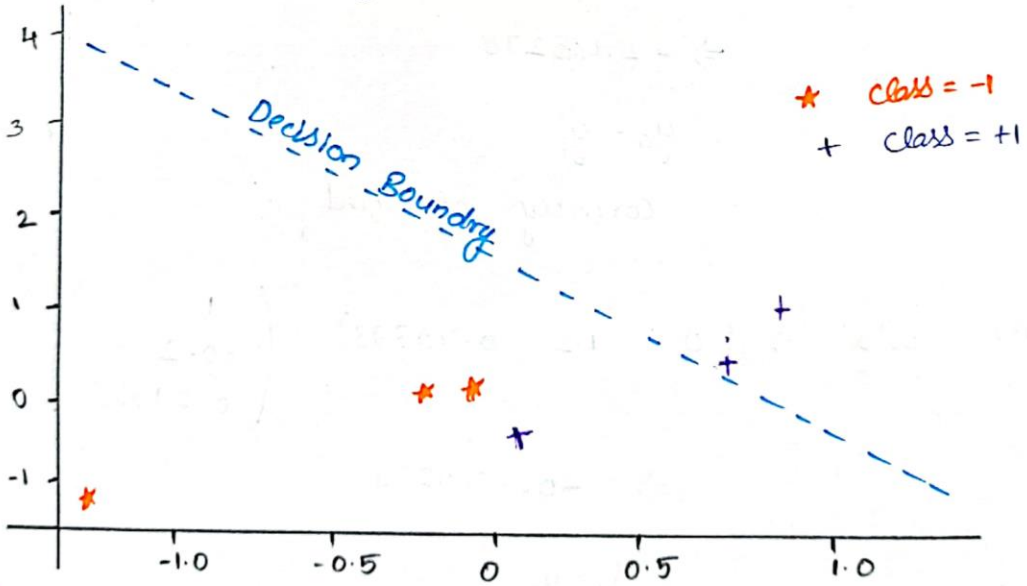
misclassified

Let us update the weights

$$w = w - x$$

$$w = \begin{pmatrix} 0 \\ 1.2 \\ 0.783333 \end{pmatrix} - \begin{pmatrix} 1 \\ -0.1 \\ 0.216667 \end{pmatrix}$$

$$w = \begin{pmatrix} -1 \\ 1.3 \\ 0.56667 \end{pmatrix}$$



Iteration 3:

(i) $w^T x \Rightarrow (-1 \quad 1.3 \quad 0.56667) \begin{pmatrix} 1 \\ 0.8 \\ 0.716667 \end{pmatrix}$

$\Rightarrow 0.4461$

$y_a = y_p$ correctly classified

(2) $w^T x = \begin{pmatrix} -1 \\ 1.3 \\ 0.56667 \end{pmatrix}^T \begin{pmatrix} 1 \\ -1.2 \\ -1.28333 \end{pmatrix}$

$\Rightarrow -3.28722$

$y_a = y_p$ (correctly classified)

(3) $w^T x = \begin{pmatrix} -1 & 1.3 & 0.56667 \end{pmatrix} \begin{pmatrix} 1 \\ -0.2 \\ 0.216667 \end{pmatrix}$

$\Rightarrow -1.137222$

$y_a = y_p$ (correctly classified)

(4) $w^T x = \begin{pmatrix} -1 & 1.3 & 0.56667 \end{pmatrix} \begin{pmatrix} 1 \\ -0.1 \\ 0.216667 \end{pmatrix}$

$\Rightarrow -1.00722$

$y_a = y_p$ (correctly classified)

(5) $w^T x = \begin{pmatrix} -1 & 1.3 & 0.56667 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ -0.08333 \end{pmatrix}$

$\Rightarrow -1.04722$

$y_a \neq y_p$

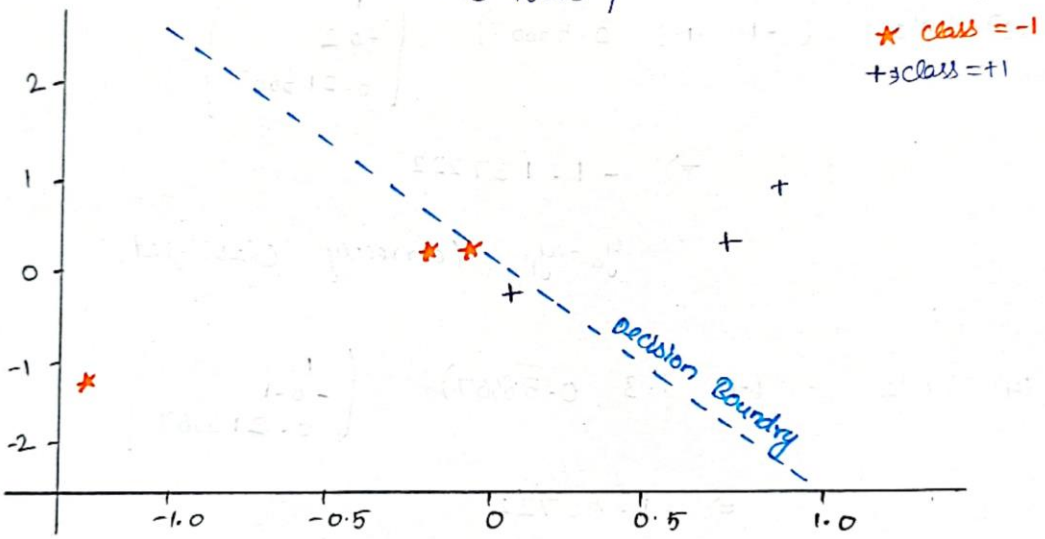
misclassified

Let us update the weight

$w = w + x$

$$w = \begin{pmatrix} -1 \\ 1.3 \\ 0.566667 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ -0.083333 \end{pmatrix}$$

$$w = \begin{pmatrix} 0 \\ 1.3 \\ 0.48333 \end{pmatrix}$$



Iteration 4:-

(1) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.3 & 0.48333 \end{pmatrix} \begin{pmatrix} 1 \\ 0.8 \\ 0.716667 \end{pmatrix}$

$\Rightarrow 1.38639$

$y_a = y_p$

correctly classified

(2) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.3 & 0.48333 \end{pmatrix} \begin{pmatrix} 1 \\ -1.2 \\ -1.28333 \end{pmatrix}$

$\Rightarrow -2.180278$

$y_a = y_p$ (correctly classified)

(3) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.3 & 0.48333 \end{pmatrix} \begin{pmatrix} 1 \\ -0.2 \\ 0.216667 \end{pmatrix}$

$\Rightarrow -0.155281$

$y_a = y_p$ Correctly classified

(4) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.3 & 0.48333 \end{pmatrix} \begin{pmatrix} 1 \\ -0.1 \\ 0.216667 \end{pmatrix}$

$\Rightarrow -0.025278$

$y_a = y_p$

Correctly classified

(5) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.3 & 0.48333 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ -0.08333 \end{pmatrix}$

$\Rightarrow -0.040278$

$y_a \neq y_p$

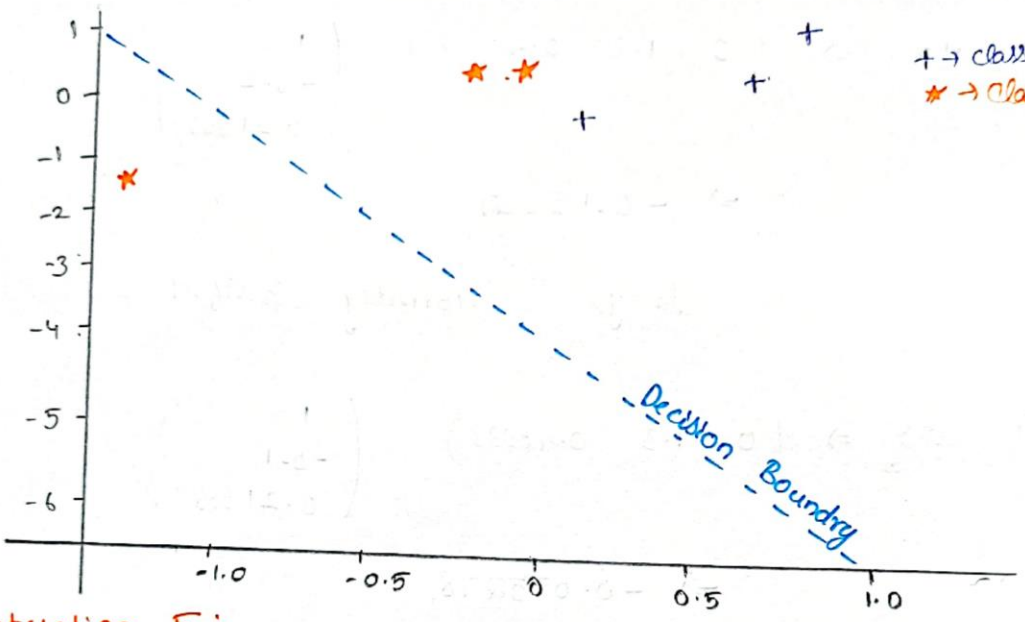
misclassified

Let us update the weights

$w = w + x$

$w = \begin{pmatrix} 0 \\ 1.3 \\ 0.48333 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ -0.08333 \end{pmatrix}$

$w = \begin{pmatrix} 1 \\ 1.3 \\ 0.4 \end{pmatrix}$



Iteration 5 :-

(1) $w^T x = (1 \quad 1.3 \quad 0.4) \begin{pmatrix} 1 \\ 0.8 \\ 0.716667 \end{pmatrix}$

$\Rightarrow 2.32667$

$y_a = y_p$ (correctly classified)

(2) $w^T x) = (1 \quad 1.3 \quad 0.4) \begin{pmatrix} 1 \\ -1.2 \\ -1.28333 \end{pmatrix}$

$\Rightarrow -1.07333$

$y_a = y_p$ (correctly classified

(3) $w^T x = (1 \quad 1.3 \quad 0.4) \begin{pmatrix} 1 \\ -0.2 \\ 0.216667 \end{pmatrix}$

$\Rightarrow 0.82667$

$y_a \neq y_p$

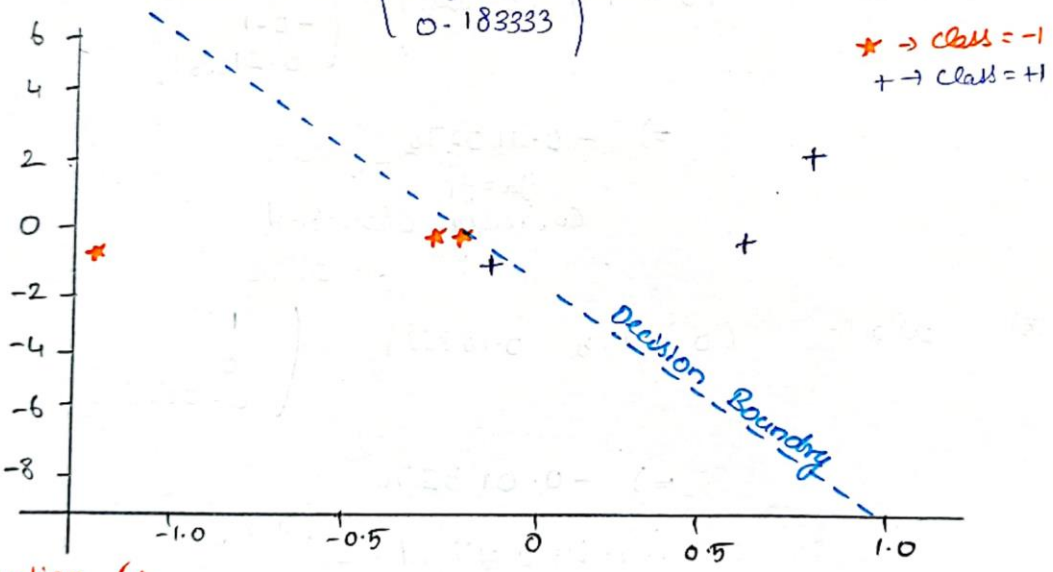
misclassified.

Let us update the weights

$$w = w - x$$

$$= \begin{bmatrix} 1 \\ 1.3 \\ 0.4 \end{bmatrix} - \begin{bmatrix} 1 \\ -0.2 \\ 0.216667 \end{bmatrix}$$

$$w = \begin{pmatrix} 0 \\ 1.5 \\ 0.183333 \end{pmatrix}$$



Iteration 6:-

(1) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.5 & 0.18333 \end{pmatrix} \begin{pmatrix} 1 \\ 0.8 \\ 0.716667 \end{pmatrix}$

$\Rightarrow 1.331387$

$y_a = y_p$ (correctly classified)

(2) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.5 & 0.18333 \end{pmatrix} \begin{pmatrix} 1 \\ -1.2 \\ -1.283333 \end{pmatrix}$

$\Rightarrow -2.03527$

$y_a = y_p$ (correctly classified)

(3) $w^T x = \begin{pmatrix} 0 & 1.5 & 0.18333 \end{pmatrix} \begin{pmatrix} 1 \\ -0.2 \\ 0.216667 \end{pmatrix}$

$\Rightarrow -0.2602778$

$y_a = y_p$

Correctly classified

(4) $w^T x = \begin{pmatrix} 0 & 1.5 & 0.18333 \end{pmatrix} \begin{pmatrix} 1 \\ -0.1 \\ 0.216667 \end{pmatrix}$

$\Rightarrow -0.110278$

$y_a = y_p$

Correctly classified

(5) $w^T x = \begin{pmatrix} 0 & 1.5 & 0.18333 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0.083333 \end{pmatrix}$

$\Rightarrow -0.015278$

$y_a \neq y_p$

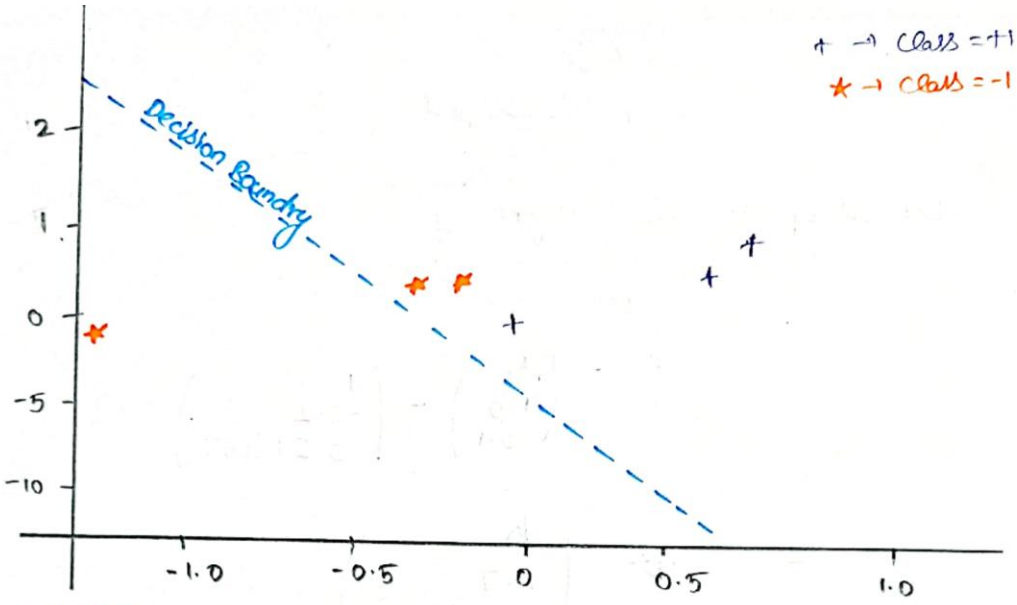
miss classified

Let us update the weights

$w_1 = w + x$

$= \begin{pmatrix} 0 \\ 1.5 \\ 0.18333 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \\ 0.08333 \end{pmatrix}$

$w = \begin{pmatrix} 1 \\ 1.5 \\ 0.1 \end{pmatrix}$



Itteration 7:

(1) $w^T x \Rightarrow (1 \quad 1.5 \quad 0.1) \begin{pmatrix} 1 \\ 0.8 \\ 0.716667 \end{pmatrix}$

$\Rightarrow 2.27167$

$y_a = y_p$

Correctly classified

(2) $w^T x \Rightarrow (1 \quad 1.5 \quad 0.1) \begin{pmatrix} 1 \\ -1.2 \\ -1.28333 \end{pmatrix}$

$\Rightarrow -0.92833$

$y_a = y_p$

Correctly classified

(3) $w^T x \Rightarrow (1 \quad 1.5 \quad 0.1) \begin{pmatrix} 1 \\ -0.2 \\ 0.216667 \end{pmatrix}$

$\Rightarrow 0.721667$

$y_a \neq y_p$

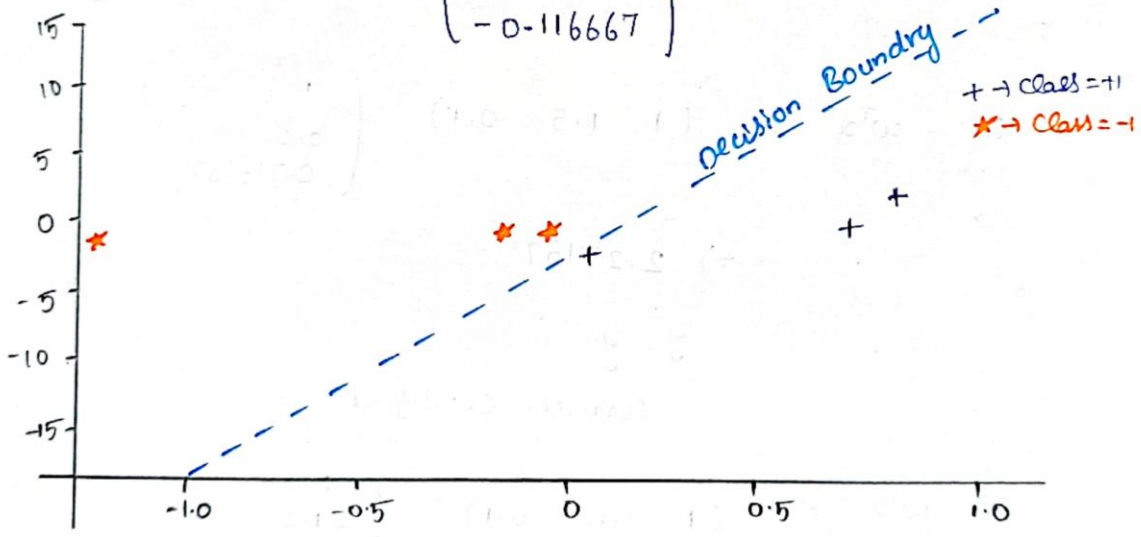
misclassified

Let us update the weight

$$w = w - x$$

$$= \begin{pmatrix} 1 \\ 1.5 \\ 0.1 \end{pmatrix} - \begin{pmatrix} 1 \\ -0.2 \\ 0.216667 \end{pmatrix}$$

$$w = \begin{pmatrix} 0 \\ 1.7 \\ -0.116667 \end{pmatrix}$$



Iteration 8:-

(i) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.7 & -0.116667 \end{pmatrix} \begin{pmatrix} 1 \\ 0.8 \\ 0.716667 \end{pmatrix}$

$\Rightarrow 1.276389$

$y_a = y_p$

Correctly classified

(2) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.7 & -0.116667 \end{pmatrix} \begin{pmatrix} 1 \\ -1.2 \\ -1.28333 \end{pmatrix}$

$\Rightarrow -1.890278$

$y_a = y_p$

Correctly classified

(3) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.7 & -0.116667 \end{pmatrix} \begin{pmatrix} 1 \\ -0.2 \\ 0.216667 \end{pmatrix}$

$\Rightarrow -0.365278$

$y_a = y_p$

Correctly classified

(4) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.7 & -0.116667 \end{pmatrix} \begin{pmatrix} 1 \\ -0.1 \\ 0.216667 \end{pmatrix}$

$\Rightarrow -0.195278$

$y_a = y_p$

Correctly classified

(5) $w^T x \Rightarrow \begin{pmatrix} 0 & 1.7 & -0.116667 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ -0.08333 \end{pmatrix}$

$\Rightarrow 0.009722$

$y_a = y_p$

Correctly classified

(6) $w^T x = \begin{pmatrix} 0 & 1.7 & -0.116667 \end{pmatrix} \begin{pmatrix} 1 \\ 0.7 \\ 0.216667 \end{pmatrix}$

$\Rightarrow 1.16472$

$y_a = y_p$ Correctly classified

In how many steps Perception algorithm will converge

Perceptron algorithm converge in 8 Iteration.

what is the final Decision boundy ?

Final weights are

$$w = \begin{bmatrix} 0 \\ 1.7 \\ -0.116667 \end{bmatrix}$$

Gurmukhi Handwritten Digit Classification:

Our goal is to develop a neural network solution for classifying handwritten Gurmukhi digits. Gurmukhi is a popular Indian script widely used in the Indian state of Punjab, and the dataset provided includes images of handwritten digits from 0 to 9.

The dataset is divided into two folders: a train folder and a validation folder. The train folder contains nine subfolders, one for each digit, and each subfolder contains images of that digit. Similarly, the validation folder also contains nine subfolders, each with images of the corresponding digit.

Neural Network :

A simple neural network (NN) is a type of artificial neural network that is composed of a single hidden layer between the input and output layers. It is also known as a single-layer neural network or a perceptron. The main purpose of a simple NN is to perform binary classification or regression tasks.

The structure of a simple NN consists of an input layer, a hidden layer, and an output layer. The input layer is responsible for receiving the input data, which is then processed by the hidden layer. The hidden layer performs computations on the input data using a set of weights and biases. These weights and biases are adjusted during training to minimize the error in the output.

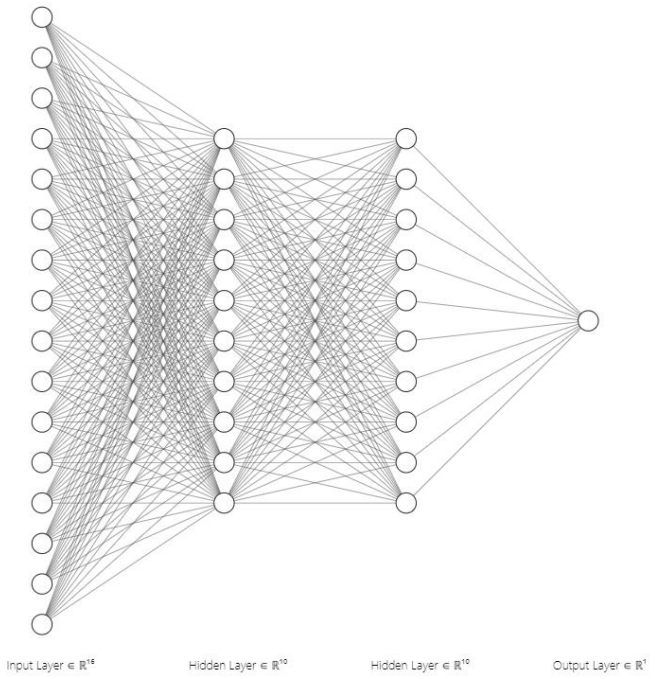
The output layer of a simple NN produces the final result, which can be a classification label or a numerical value. The output layer consists of a single neuron in the case of binary classification or regression tasks. The output of this neuron is transformed into a probability value using a sigmoid function, which is used to determine the final classification or regression output.

The training of a simple NN involves adjusting the weights and biases in the hidden layer to minimize the error in the output. This is done using an optimization algorithm such as stochastic gradient descent. The goal of the optimization algorithm is to find the set of weights and biases that minimize the error between the predicted output and the actual output.

There are various activation functions that can be used in a simple NN to introduce non-linearity into the model. The most commonly used activation function is the rectified linear unit (ReLU) function, which is a non-linear function that is used to introduce non-linearity into the model. Another popular activation function is the sigmoid function, which is used to produce the probability value for binary classification tasks.

The performance of a simple NN depends on several factors such as the number of neurons in the hidden layer, the number of input features, the activation function used, and the optimization algorithm used. Increasing the number of neurons in the hidden layer can improve the model's performance, but it can also lead to overfitting. Similarly, increasing the number of input features can improve the model's performance, but it can also increase the complexity of the model.

In summary, a simple NN is a type of artificial neural network that is used for binary classification or regression tasks. It consists of an input layer, a hidden layer, and an output layer. The weights and biases in the hidden layer are adjusted during training using an optimization algorithm to minimize the error in the output. The performance of a simple NN depends on several factors such as the number of neurons in the hidden layer, the number of input features, the activation function used, and the optimization algorithm used.



DATA SET EXPLANATION:

The dataset provided for the Gurmukhi Handwritten Digit Classification task is a collection of images of handwritten digits in the Gurmukhi script. The dataset is stored in a folder named "Dataset" which contains two sub-folders, "train" and "val".

The "train" folder contains images of digits divided into nine sub-folders, one for each digit from 0 to 9. Each image is named using a unique identifier and has a ".png" extension. The sub-folder for a particular digit contains all images of that digit. For example, the sub-folder "0" contains all images of the digit "0".

Similarly, the "val" folder also contains images of digits divided into nine sub-folders, one for each digit from 0 to 9. Each image is named using a unique identifier and has a ".png" extension. The sub-folder for a particular digit contains all images of that digit.

The name of the sub-folder containing an image of a digit indicates the class or label of that image. For example, an image of the digit "0" stored in the "0" sub-folder is labeled as class "0".

The goal of the Gurmukhi Handwritten Digit Classification task is to develop a neural network model that can accurately classify the digits in the provided dataset.

Process the Dataset and convert to CSV

The process of reading and preprocessing image data for NN models. The goal of this code is to prepare a dataset of handwritten digits for training a NN model that can classify new images of digits.

The first step is to mount a Google Drive, which is where the dataset is stored. Once the drive is mounted, the path to the train data directory is defined. This directory contains the images that will be used for training the NN model.

Next, we will creates empty lists to store the images and their respective labels. The script then loops through each folder in the train directory and reads all of the images in each folder. The images are read using OpenCV, which is a library used for computer vision tasks. Each image is then resized to a specified size using the OpenCV function cv2.resize().

After the images have been read and resized, they are flattened and normalized. Flattening the images means that they are converted from a 2D array to a 1D array, which is easier to work with in machine learning models. Normalization is the process of scaling the pixel values of the images to be between 0 and 1. This is important because it helps the NN model to learn more efficiently.

As the images are being read, their corresponding labels are also being read and stored in a separate list. The labels indicate the actual digit that is represented in each image. For example, if an image contains the digit 5, its label will be 5.

Once all of the images and their labels have been read and stored in lists, they are converted to numpy arrays. Numpy is a library for working with arrays and matrices in Python, and it is commonly used in machine learning. Converting the images and labels to numpy arrays allows them to be easily manipulated and used as input for NN models.

Finally, we saves the images and labels as a CSV file. CSV (Comma Separated Values) files are a common way of storing data in a tabular format that can be easily read by other programs. Saving the data in this format allows it to be easily loaded into other machine learning libraries or programs for further analysis.

So we have two csv file

which represents the flattened image data. Each column name is a string containing the word "pixel" and a number that ranges from 0 to 783. This is because each image in the dataset is a 28x28 grayscale image, and when flattened, it becomes a 1D array of length 784. Each element in the array represents a pixel in the image. Therefore, each column in the CSV file will represent a pixel in the image, and its corresponding value will represent the intensity of that pixel. The column names are created to help identify the pixel's location in the image when analyzing the data.

Train.csv

label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782
5	0.996078	0.929412	0.929412	0.462745	0.0	0.000000	0.000000	0.066667	0.945098	...	1.0	1.0	1.0	0.0	0.200000	0.952941	1.000000	1.000000	1.000000
5	0.996078	0.729412	0.000000	0.000000	0.0	0.000000	0.862745	0.933333	1.000000	...	1.0	1.0	1.0	1.0	1.000000	0.972549	0.462745	0.000000	0.784314
5	0.929412	0.729412	0.000000	0.000000	0.0	0.729412	0.929412	1.000000	1.000000	...	1.0	1.0	1.0	1.0	1.000000	1.000000	1.000000	1.000000	0.215686
5	0.862745	0.000000	0.000000	0.000000	0.0	0.729412	0.929412	1.000000	1.000000	...	1.0	1.0	1.0	1.0	0.984314	0.596078	0.000000	0.596078	0.984314
5	1.000000	0.984314	0.929412	0.462745	0.0	0.000000	0.000000	0.000000	0.200000	...	1.0	1.0	1.0	1.0	1.000000	1.000000	0.964706	0.333333	0.000000

Val.csv

label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782
5	0.996078	0.929412	0.929412	0.462745	0.0	0.000000	0.000000	0.066667	0.945098	...	1.0	1.0	1.0	0.0	0.200000	0.952941	1.000000	1.000000	1.000000
5	0.996078	0.729412	0.000000	0.000000	0.0	0.000000	0.862745	0.933333	1.000000	...	1.0	1.0	1.0	1.0	1.000000	0.972549	0.462745	0.000000	0.784314
5	0.929412	0.729412	0.000000	0.000000	0.0	0.729412	0.929412	1.000000	1.000000	...	1.0	1.0	1.0	1.0	1.000000	1.000000	1.000000	1.000000	0.215686
5	0.862745	0.000000	0.000000	0.000000	0.0	0.729412	0.929412	1.000000	1.000000	...	1.0	1.0	1.0	1.0	0.984314	0.596078	0.000000	0.596078	0.984314
5	1.000000	0.984314	0.929412	0.462745	0.0	0.000000	0.000000	0.000000	0.200000	...	1.0	1.0	1.0	1.0	1.000000	1.000000	0.964706	0.333333	0.000000

Implement Using keras Library:

This code is an example of a machine learning model that uses the Keras library to build a neural network to recognize handwritten digits. The dataset which is a collection of training and testing images of handwritten digits.

The code is divided into several sections:

1. Importing necessary libraries: *The code imports pandas, numpy, tensorflow, scikit-learn, and matplotlib.pyplot libraries. Pandas is used for data manipulation, numpy for numerical computation, tensorflow for building and training neural networks, scikit-learn for splitting the data into training and validation sets, and matplotlib for data visualization.*

2. Loading the data*: The code loads the training and testing data from CSV files using the read_csv() function from the pandas library. The training data has 785 columns, where the first column represents the label (the digit the image represents), and the remaining 784 columns represent the pixel values (28x28 grayscale images). The testing data has 784 columns representing the pixel values of the images.*

3. Preprocessing the data*: The code separates the input features (pixels) and target variable (labels) from the loaded data using iloc function. Then, it normalizes the pixel values by dividing each pixel value by 255.0 to get the pixel values between 0 and 1.*

4. Splitting the data*: The code splits the training data into training and validation sets using the train_test_split() function from the scikit-learn library. The test_size parameter is set to 0.2, which means that 20% of the training data will be used for validation. The random_state parameter is set to 42 to ensure that the same split is obtained each time the code is run.*

5. Defining the model architecture*: The code defines the model architecture using the Sequential class from the Keras library. The model consists of three dense layers. The first layer*

has 512 neurons with a ReLU activation function, the second layer has 256 neurons with a ReLU activation function, and the output layer has 10 neurons with a softmax activation function. The softmax function outputs probability scores for each class, representing the probability that the input image belongs to that class. The dropout function is used to prevent overfitting by randomly dropping out some neurons during training.

***6. Compiling the model**: The code compiles the model by specifying the loss function, optimizer, and evaluation metric to be used during training. The loss function used is sparse_categorical_crossentropy, which is used when the target variable is integer-encoded (in this case, the digit labels). The optimizer used is Adam, which is an adaptive learning rate optimization algorithm. The metric used to evaluate the model during training is accuracy.*

***7. Training the model**: The code trains the model on the training data using the fit() function. The number of epochs is set to 200, which means that the model will be trained on the entire training data 200 times. The batch size is set to 32, which means that the model will be trained on 32 images at a time. The validation_data parameter is set to (X_val, y_val), which means that the model will be evaluated on the validation set after each epoch.*

***8. Evaluating the model**: The code evaluates the trained model on the test data using the evaluate() function. It prints the test loss and test accuracy, which indicate how well the model performs on the unseen test data.*

***9. Making predictions**: The code makes predictions on the test set using the predict() function. It also displays a few random images from the test set along with their predicted and actual labels using the matplotlib library. The predicted label is the digit that the model predicts the image represents, and the actual label is the true label of the image. This allows us to see how well the model performs on unseen data.*

Test loss: 0.2437712848186493

Test accuracy: 0.9606741666793823

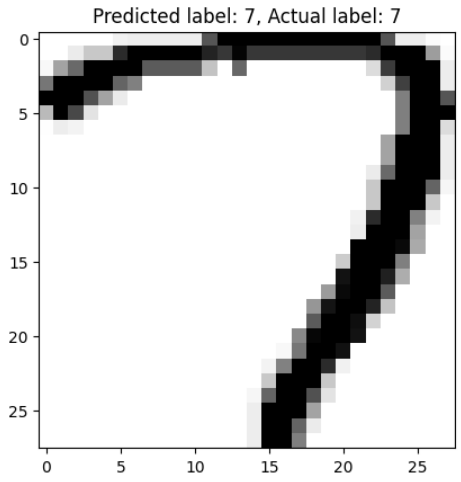
The output shows the test loss and test accuracy of the trained neural network model on the test set.

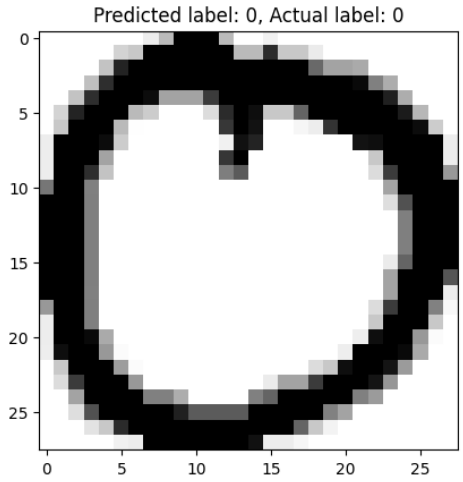
The test loss is a measure of how well the model predicts the correct label for each image in the test set. It represents the difference between the predicted and actual labels of the images in the test set. The lower the test loss, the better the model is performing on the test set.

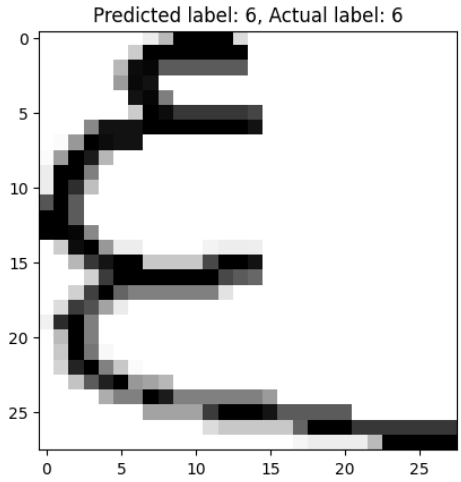
In this case, the test loss is 0.243, which means that the model is making an average error of 0.243 on each image in the test set.

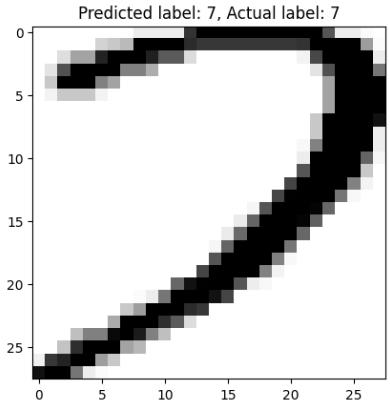
The test accuracy represents the percentage of correctly classified images in the test set. In this case, the test accuracy is 0.961, which means that the model correctly classifies 96.1% of the images in the test set. This indicates that the model has a high level of accuracy and is performing well in recognizing handwritten digits.

Predicted Vs Actual









Implement From the Scratch:

The neural network is trained on the training images and then tested on the testing images to evaluate its accuracy.

The neural network implemented in the code has two hidden layers. The first hidden layer has 10 neurons, and the second hidden layer also has 10 neurons. The input layer of the neural network has 784 neurons, which is the number of pixels in each 28x28 grayscale image in the MNIST dataset. The output layer has 10 neurons, each representing a digit from 0 to 9.

The code first initializes the parameters of the neural network randomly using a uniform distribution between -0.5 and 0.5. These parameters include the weight matrices and bias vectors for each layer.

The code then defines two activation functions, ReLU and softmax. ReLU (Rectified Linear Unit) is used as the activation function for the first hidden layer, and softmax is used as the activation function for the output layer. ReLU is a simple activation function that outputs the input value if it is positive and 0 otherwise. Softmax is used to normalize the output of the neural network into a probability distribution over the 10 possible classes.

The forward propagation function takes in the input data, applies the weight matrices and bias vectors to the input data to generate the output of each layer, and applies the activation function to the output of each layer to generate the final output of the neural network.

The backward propagation function calculates the gradients of the parameters of the neural network with respect to the loss function. The loss function used in the code is the cross-entropy loss function. The gradients are calculated using the chain rule of differentiation and the derivative of the activation functions.

The update parameters function updates the parameters of the neural network using the gradients calculated in the backward propagation function and a learning rate alpha.

The code then defines a function to make predictions on new input data using the trained neural network. The function applies the trained neural network to the input data and outputs the class with the highest probability.

Finally, the code tests the accuracy of the neural network on the testing images in the MNIST dataset. It uses the make predictions function to generate predictions on the testing images and calculates the accuracy by comparing the predicted classes with the true classes. The accuracy of the neural network on the testing images is used as a measure of its performance.

Explain All Functions in the code:

1. `init_params()`: *This method initializes the parameters of a neural network. It takes no input arguments and returns four variables - two matrices and two vectors - each containing random values between -0.5 and 0.5. The first matrix contains 10 rows and 784 columns, the first vector contains 10 elements, the second matrix contains 10 rows and 10 columns, and the second vector contains 10 elements.*

2. `ReLU()`: *This method is the implementation of the Rectified Linear Unit (ReLU) activation function. It takes one input argument, which can be a scalar or a matrix, and returns the same shape as the input argument. For each element of the input argument, ReLU returns the element if it is greater than or equal to zero, and zero otherwise.*

3. `softmax()`: *This method is the implementation of the softmax activation function. It takes one input argument, which can be a scalar or a matrix, and returns a probability distribution over the input argument. For a vector, the softmax function calculates the exponential of each element, normalizes them by the sum of exponential elements, and returns the normalized vector. For a matrix, the softmax function calculates the exponential of each element for each row, normalizes the rows by the sum of exponential elements for each row, and returns the normalized matrix.*

4. `forward_prop()`: *This method calculates the forward propagation of a neural network given the input data and parameters. It takes five input arguments - four matrices and one vector - and returns four matrices. The first matrix is the dot product of the first and fifth input arguments, added to the second input argument, followed by the ReLU activation function. The second matrix is the dot product of the third and the output of the ReLU activation function, added to the fourth input argument, followed by the softmax activation function.*

5. `ReLU_deriv()`: *This method calculates the derivative of the ReLU activation function. It takes one input argument, which can be a scalar or a matrix, and returns the same shape as the input argument. For each element of the input argument, ReLU_deriv returns 1 if it is greater than or equal to zero, and zero otherwise.*

6. `one_hot()`: *This method converts a vector of integers into a one-hot encoded matrix. It takes one input argument, which is a vector of integers, and returns a matrix where each row represents an integer in the input vector and has a value of 1 in the column corresponding to the integer and 0 in all other columns.*

7. `backward_prop()`: *This method calculates the backward propagation of a neural network given the output of the forward propagation and the input data and parameters. It takes six input arguments - four matrices and two vectors - and returns four matrices. The first matrix is the dot product of the transpose of the third input argument and the first input argument,*

multiplied by the derivative of the ReLU activation function applied to the second input argument. The second matrix is the dot product of the transpose of the third input argument and the fourth input argument, multiplied by 1/m, where m is the number of samples. The third matrix is the dot product of the transpose of the fifth input argument and the fourth output argument, multiplied by the derivative of the softmax activation function applied to the third output argument. The fourth matrix is the dot product of the third output argument and the transpose of the first input argument, multiplied by 1/m.

8. `update_params()`: *This method updates the parameters of a neural network using the gradients calculated in the backward propagation step. It takes eight input arguments - four matrices, four vectors, and one scalar - and returns four matrices. The updated values of the first matrix, the first vector, the second matrix*

The neural network consists of an input layer, two hidden layers, and an output layer. The input layer has 784 nodes, each representing one pixel of the input image. The two hidden layers have 10 nodes each, and the output layer has 10 nodes, each representing one digit from 0 to 9. The neural network uses the ReLU activation function for the hidden layers and the softmax activation function for the output layer.

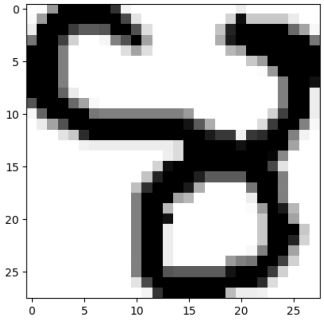
The model is trained using the gradient descent algorithm, which involves forward propagation to calculate the output of the neural network, and backward propagation to calculate the gradients of the weights and biases. The weights and biases are then updated using the gradients and a learning rate. This process is repeated for a number of iterations until the model reaches a good accuracy.

The accuracy of the model is evaluated using the accuracy metric, which is the number of correct predictions divided by the total number of predictions. The model is also tested on a separate set of images to ensure that it is not overfitting to the training set. Finally, the model can be used to make predictions on new images by passing them through the neural network and getting the output.

PREDICTED VS ACTUAL:

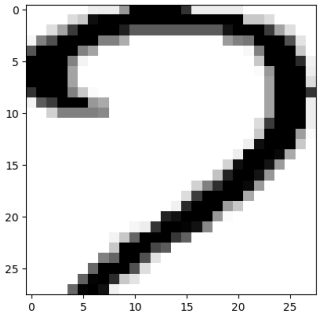
Prediction: [4]

Label: 4



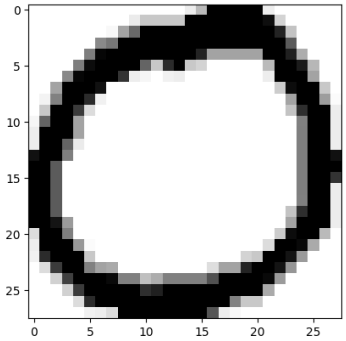
Prediction: [7]

Label: 7



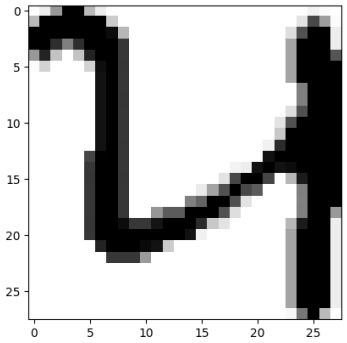
Prediction: [0]

Label: 0



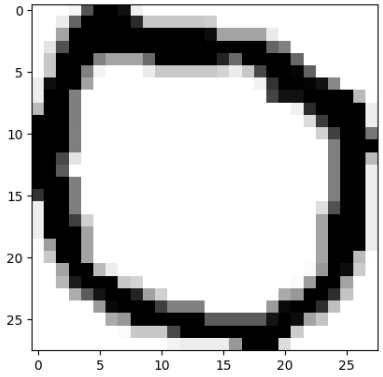
Prediction: [5]

Label: 5



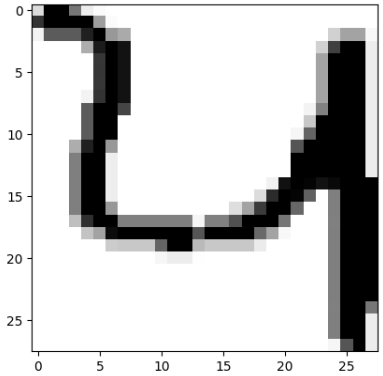
Prediction: [0]

Label: 0



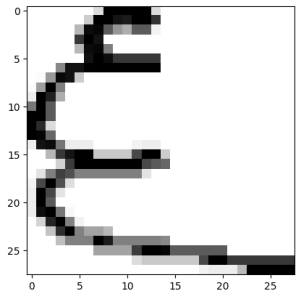
Prediction: [5]

Label: 5



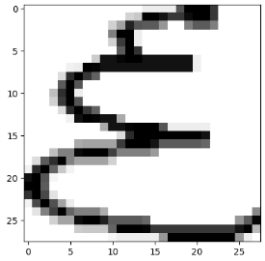
Prediction: [6]

Label: 6



Prediction: [6]

Label: 6



Accuracy:

Evaluate the performance of the neural network model: Test Accuracy and Val Accuracy.

Test Accuracy is the measure of the model's accuracy on the test dataset. This metric is calculated by comparing the model's predicted outputs with the actual outputs on the test dataset. An accuracy of 1.0 means that the model correctly predicted all the outputs on the test dataset.

Val Accuracy is the measure of the model's accuracy on the validation dataset. This metric is calculated in the same way as the Test Accuracy metric, but it uses the validation dataset instead of the test dataset. An accuracy of 0.92 means that the model correctly predicted 92% of the outputs on the validation dataset.

Both Test Accuracy and Val Accuracy are important metrics to evaluate the performance of a neural network model. The Test Accuracy metric tells us how well the model is likely to perform

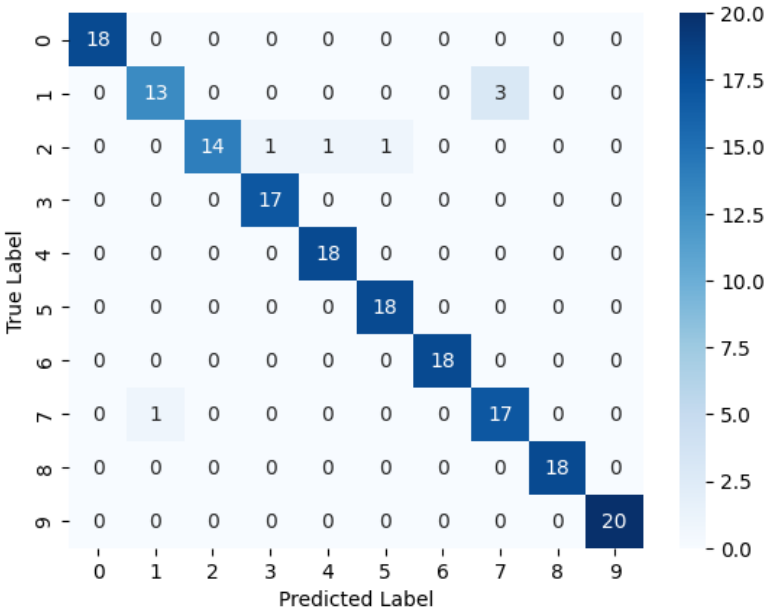
on new, unseen data. The Val Accuracy metric tells us how well the model is performing during training, and can help us identify if the model is overfitting (performing well on the training data, but poorly on new, unseen data).
In this case, the model has a perfect Test Accuracy of 1.0, which indicates that it is likely to perform well on new, unseen data. The Val Accuracy of 0.92 indicates that the model is performing well during training and is not overfitting the training data.

Conf Matrix :

The confusion matrix is a table that is often used to evaluate the performance of a classification model. It shows the number of correct and incorrect predictions made by the classification model compared to the actual outcomes (or true values) in the test data.
In the given confusion matrix, the rows represent the actual or true classes of the test data, and the columns represent the predicted classes by the classification model. For example, the first row represents the true values of the first class, and the model predicted that all the samples in that class were correctly classified.
The diagonal of the matrix represents the correctly classified samples, and the off-diagonal elements represent the misclassified samples. In this matrix, we can see that all the samples in the first class were correctly classified (18 samples), and the same is true for all other classes. Therefore, we can say that the model has a perfect accuracy of 100% on the test data.

However, we can also see that there are a few misclassifications in the confusion matrix. For example, the second column of the third row shows that one sample from the third class was predicted as the fourth class, and one sample from the third class was predicted as the fifth class. Similarly, in the second row, the eighth column shows that three samples from the second class were predicted as the eighth class. These misclassifications are reflected in the overall validation accuracy of 92%, which means that out of all the samples in the test data, 92% were correctly classified by the model.

With Keras Library:



Without any Library:

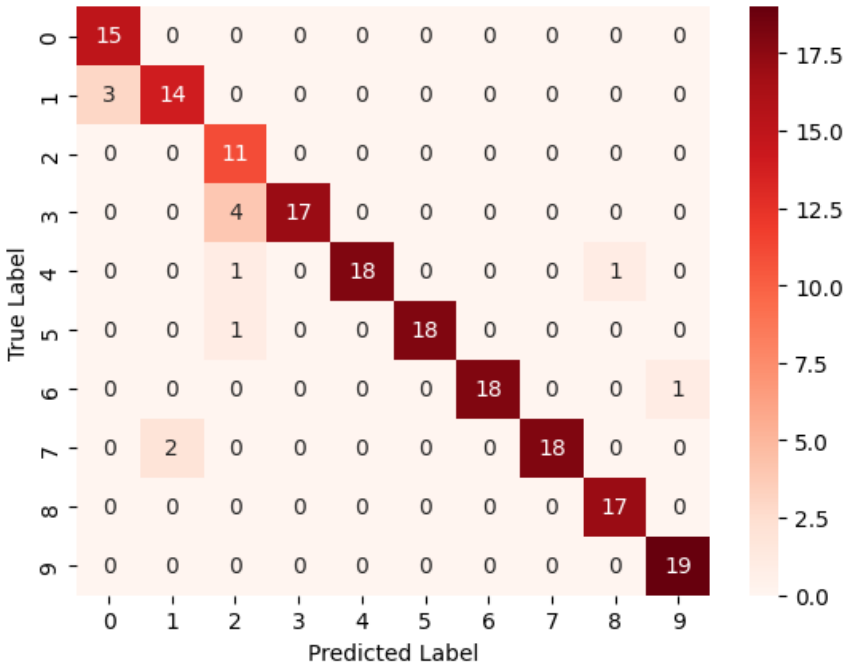


Chart Image Classification using CNN

Task 1:

The task involves downloading a dataset from the provided link which contains chart images belonging to five different classes - Line, Dot Line, Horizontal Bar, Vertical Bar, and Pie chart. The dataset also includes a CSV file containing the corresponding labels for each image. The task requires the use of 80% of the images for training and 20% of the images for validation.

Task 2:

The task involves implementing a two-layer Convolutional Neural Network (CNN) to classify the given chart images into their respective classes. The CNN should be trained on the training set and evaluated on the validation set. The accuracy, loss, and loss plot should be calculated and submitted along with the code. Observations should also be briefly written about the performance of the model.

Task 3:

The task involves fine-tuning a pre-trained network, such as AlexNet, for chart image classification. The pre-trained network should be modified to fit the requirements of the task, and the performance of the model should be reported.

CNN:

A Convolutional Neural Network (CNN) is a type of deep learning neural network that is commonly used for image classification and recognition tasks. Unlike a fully connected neural network, which treats input data as a one-dimensional vector, a CNN takes in input data as a two-dimensional matrix, which is often referred to as an image.

The main building block of a CNN is the convolutional layer, which applies a set of filters (also called kernels or weights) to the input image. Each filter applies a mathematical operation to a small region of the image, called a receptive field, to extract a specific feature. The output of the filter is then passed through an activation function to produce a feature map, which represents the presence of the feature in the input image.

A pooling layer is often applied after a convolutional layer to downsample the feature maps and reduce the dimensionality of the input. The most common type of pooling is max pooling, which takes the maximum value of each non-overlapping rectangular region of the feature map.
The output of the convolutional and pooling layers is then passed through one or more fully connected layers, which perform a final classification of the input image. The final layer of the network typically uses a softmax activation function to produce a probability distribution over the output classes.
The training of a CNN involves adjusting the weights of the filters and fully connected layers to minimize the loss between the predicted and actual output labels. This is typically done using backpropagation and gradient descent, where the gradients of the loss with respect to the weights are calculated and used to update the weights in the opposite direction of the gradient.

In practice, CNNs are often pre-trained on large datasets, such as ImageNet, using transfer learning. This involves using the weights learned from the pre-training as initialization for the CNN on the new task, and fine-tuning the weights on the new dataset to improve performance. Overall, CNNs have proven to be highly effective in a variety of image classification tasks, and continue to be an active area of research in the field of deep learning.

Observation on Given Dataset:

The "charts" dataset consists of a single folder called "charts" that contains two main components:
1. "train_val" subfolder: This subfolder contains 1000 images that are used for both training and validation purposes. These images are used to train and evaluate the performance of the machine learning model.
2. "train_val.csv" file: This is a CSV (Comma Separated Values) file that contains two columns. The first column, "imageIndex", lists the names of all the image files in the "train_val" subfolder. The second column, "type", indicates the type of graph depicted in the corresponding image.

The dataset is designed to train and test models for image classification tasks. The task in this specific dataset is to classify chart images into one of five categories: "Line", "Dot Line", "Horizontal Bar", "Vertical Bar", and "Pie" charts.

The dataset is useful for training and testing various machine learning models such as Convolutional Neural Networks (CNNs) for image classification tasks. The size of the dataset is relatively small (1000 images), which is beneficial for training models with less computational resources.

Create Train Folder:

To split the images in the `train_val` folder into 5 different folders based on their chart type, we can use the `train_val.csv` file which contains two columns - `imageIndex` and `type`.

First, we can read the `train_val.csv` file and create a dictionary with keys as the chart types and values as empty lists. Then, we can iterate through the rows of the `train_val.csv` file and append the image names to the respective chart type's list in the dictionary.

Once we have the image names for each chart type, we can create 5 different folders - "Line", "Dot Line", "Horizontal Bar", "Vertical Bar", and "Pie" charts in the `train` directory. We can use the `os` module to create the directories.

Next, we can iterate through the dictionary we created earlier and copy the images from the `train_val` directory to the respective chart type's folder in the `train` directory. We can use the `shutil` module to copy the images.

After this process is complete, we will have 5 folders in the `train` directory, each containing the images for the respective chart type. We can then use these folders to train our model for chart image classification.

Observation on Train Folder:

The folder "vbar_categorical" contains 200 images that belong to the category "Vertical Bar" charts.

The folder "hbar_categorical" contains 200 images that belong to the category "Horizontal Bar" charts. The folder "line" contains 200 images that belong to the category "Line" charts.

The folder "pie" contains 200 images that belong to the category "Pie" charts.

Finally, the folder "dot_line" contains 200 images that belong to the category "Dot Line" charts.

Overall, the five folders contain a total of 1000 images from the original "train_val" folder. Each folder is dedicated to one of the five types of charts that the model is trained to classify. This type of organization makes it easy to feed the images into a machine learning algorithm in a structured and organized manner. It also enables the model to learn the unique features and patterns of each type of chart and make accurate predictions when presented with new, unseen images.

Task 1: Split Train and Val Set

The task is to split the dataset into training and validation sets for the chart image classification task.

It creates two new directories named train and val in a specified location. It then creates five subdirectories inside each of the new directories. These subdirectories correspond to the five different types of chart images.

The dataset is split into the training and validation sets by shuffling the images in each subdirectory and assigning 80% of the images to the training set and 20% of the images to the validation set.

Finally, the script copies the images from the original dataset to their corresponding directory in the training or validation set. This process is repeated for each of the five subdirectories.

The result of splitting the original dataset into a training set and a validation set, where the images are categorized into five different types: vbar_categorical, hbar_categorical, line, pie, and dot_line.

In the training set, there are 800 images (80% of the original 1000 images), with 160 images for each type of chart. The images are further divided into subfolders based on their chart type. For example, the "vbar_categorical" folder contains 160 images of vertical bar charts.

In the validation set, there are 200 images (20% of the original 1000 images), with 40 images for each type of chart. The images are also divided into subfolders based on their chart type, following the same structure as the training set.

This way, the dataset is split into a training set and a validation set, with an 80:20 ratio for training and validation respectively. The images are categorized by their chart type, making it easier to train a model to classify them accurately.

```
Folder vbar_categorical contains 160 images
Folder hbar_categorical contains 160 images
Folder line contains 160 images
Folder pie contains 160 images
Folder dot_line contains 160 images
```

```
Folder vbar_categorical contains 40 images
Folder hbar_categorical contains 40 images
Folder line contains 40 images
Folder pie contains 40 images
Folder dot_line contains 40 images
```

Taks 2: Implement a two-layer Convolutional Neural Network

A two-layer CNN is a simple variant of the CNN architecture that consists of two convolutional layers followed by a fully connected (dense) layer for classification.

The first layer in a CNN is a convolutional layer. This layer applies a set of filters to the input image, which extracts features from the image. Each filter is a small matrix of numbers that is convolved with the input image, which results in a new feature map. The size of the feature map is determined by the size of the input image, the size of the filters, the number of filters, and the stride.

The second layer in a CNN is another convolutional layer that applies a second set of filters to the output of the first layer. This layer further extracts more complex features from the image. After the two convolutional layers, we add a fully connected (dense) layer that takes the output of the convolutional layers and classifies the input image into one of the five classes. The dense layer has one node per class, and the output is the predicted probability for each class.

The input to the network is an image, which is usually preprocessed by resizing and normalization. The output of the network is a predicted probability distribution over the five classes.

The parameters of the CNN, including the weights of the filters, are learned during training using backpropagation and stochastic gradient descent.

The two-layer CNN is a simple and relatively fast network that can achieve reasonable accuracy on image classification tasks. However, deeper CNN architectures with more layers, such as

VGG, ResNet, and Inception, have achieved state-of-the-art performance on many image classification benchmarks.

The code trains a two-layer convolutional neural network (CNN) to classify images of different types of charts. The data is split into a training set and a validation set, and each set is generated using an ImageDataGenerator object. The CNN architecture consists of two convolutional layers with pooling layers in between, followed by a flattening layer and two fully connected layers. The model is compiled with the Adam optimizer and categorical cross-entropy loss, and accuracy is used as the metric. The model is trained for 20 epochs and the training and validation accuracies are saved in a history object.

1. Loading the Data

The first step is to load the data. We will use the train and validation data created in the previous step. The data is loaded using the `ImageDataGenerator` class from the `keras.preprocessing.image` module. We also define the batch size and the image size.

2. Defining the Model

Next, we define our convolutional neural network model. We will use the `Sequential` class from the `keras.models` module to create our model. Our model will have two convolutional layers followed by a flattening layer and a dense output layer. The first convolutional layer will have 32 filters with a kernel size of 3x3 and a ReLU activation function. The second convolutional layer will have 64 filters with a kernel size of 3x3 and a ReLU activation function. The flattening layer will convert the output of the second convolutional layer to a one-dimensional array. Finally, the dense output layer will have 5 units, one for each chart class, with a softmax activation function.

3. Compiling the Model

Once our model is defined, we need to compile it. We will use the `categorical_crossentropy` loss function, the `adam` optimizer, and the `accuracy` metric.

4. Training the Model

With the model defined and compiled, we can now train it using the `fit` method. We specify the training data, the validation data, the number of epochs, and the batch size.

5. Evaluating the Model

Once training is complete, we can evaluate the model on the validation data using the `evaluate` method. We print the validation accuracy and loss.

6. Saving the Model

Finally, we can save the trained model for future use.

Train Vs Test Accuracy:

Train Accuracy : 100

Test Accuracy : 98.5

After training the model for 20 epochs, the training accuracy is reported as 1.0000, which means that the model was able to correctly classify all the training data points. However, perfect training accuracy does not necessarily mean that the model is performing well on unseen data. The validation accuracy is reported as 0.9850, which means that the model was able to correctly classify 98.5% of the validation data points. This suggests that the model is performing well on unseen data, and has not overfit to the training data.

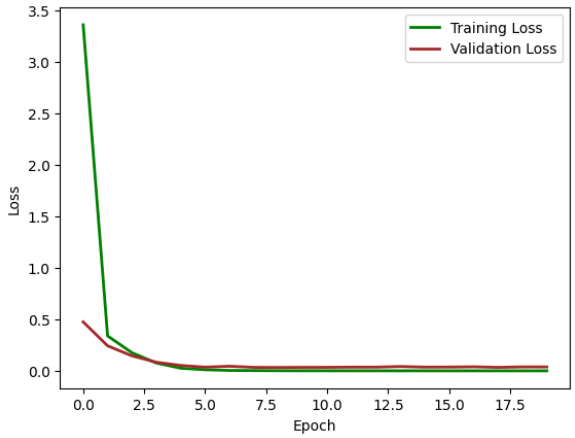
Overall, the reported training and validation accuracies suggest that the CNN model is performing well on the given chart dataset. However, it is important to keep in mind that accuracy is just one metric, and it is important to evaluate the model using other metrics as well, such as precision, recall, and F1 score.

Plot Training loss vs Validation loss:

The plot shows the training loss and validation loss values over the training epochs of the CNN model. The training loss is the value of the loss function calculated on the training dataset during each epoch of training. The validation loss is the value of the loss function calculated on the validation dataset during each epoch of training.

The purpose of plotting these two loss values is to visually analyze the performance of the model during training. Ideally, we want the training loss to decrease and the validation loss to decrease as well, indicating that the model is learning to generalize to new data. However, if the

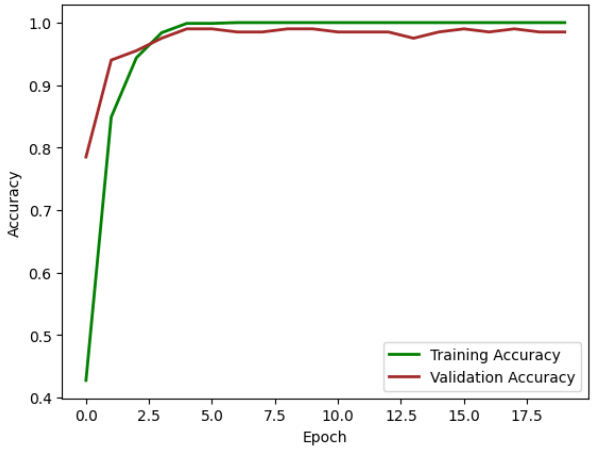
training loss decreases too much and the validation loss does not decrease or even increases, it can be an indication of overfitting.



Plot Training Accuracy vs Validation Accuracy:

Plotting the training accuracy and validation accuracy over the epochs gives an insight into how well the model is performing during training and validation. If the training accuracy keeps improving while the validation accuracy stops improving, it could be an indication of overfitting. On the other hand, if both the training accuracy and validation accuracy stop improving, it could be an indication of underfitting.

In this plot, the y-axis represents the accuracy while the x-axis represents the number of epochs. The training accuracy curve represents how the accuracy improves during the training process, while the validation accuracy curve shows how well the model performs on the validation set. Ideally, we want both curves to increase and become stable. In this case, the training accuracy curve increases and eventually stabilizes at 100%, while the validation accuracy curve increases and stabilizes at around 98%. This indicates that the model has learned the features of the training data well, and it can generalize well to unseen data. However, if the validation accuracy curve was decreasing while the training accuracy curve increased, it would indicate overfitting, and steps should be taken to prevent it.



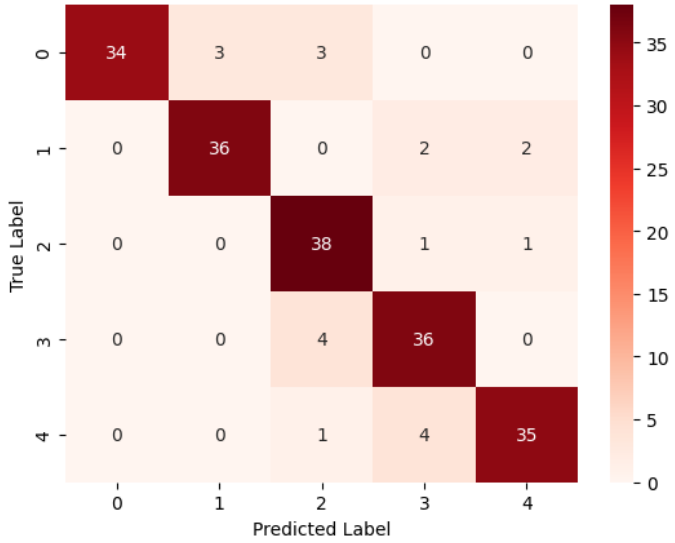
Conf Matrix :

The confusion matrix is a table that is often used to evaluate the performance of a classification model. It shows the number of correct and incorrect predictions made by the classification model compared to the actual outcomes (or true values) in the test data.

In the given confusion matrix, the rows represent the actual or true classes of the test data, and the columns represent the predicted classes by the classification model. For example, the first row represents the true values of the first class, and the model predicted that all the samples in that class were correctly classified.

The diagonal of the matrix represents the correctly classified samples, and the off-diagonal elements represent the misclassified samples. In this matrix, we can see that all the samples in the first class were correctly classified (18 samples), and the same is true for all other classes. Therefore, we can say that the model has a perfect accuracy of 100% on the test data.

However, we can also see that there are a few misclassifications in the confusion matrix. For example, the second column of the third row shows that one sample from the third class was predicted as the fourth class, and one sample from the third class was predicted as the fifth class. Similarly, in the second row, the eighth column shows that three samples from the second class were predicted as the eighth class. These misclassifications are reflected in the overall validation accuracy of 98%, which means that out of all the samples in the test data, 98% were correctly classified by the model.



Task 3: Finetune a pretrained network

Here is a list of some popular pre-trained Convolutional Neural Networks:

1. AlexNet

2. VGG16

3. VGG19

4. ResNet50

5. InceptionV3

6. InceptionResNetV2

7. MobileNet

8. DenseNet

9. Xception

10. EfficientNet

Each of these networks has its own architecture and has been pre-trained on large-scale image datasets such as ImageNet. These pre-trained models can be used as feature extractors or fine-tuned for a specific task, such as image classification, object detection, or segmentation.

AlexNet:

The architecture of AlexNet consists of eight layers. The first five layers are convolutional layers, and the last three are fully connected layers. The following is a brief overview of each layer:

1. Input layer: The input layer accepts a 227x227x3 RGB image, where 227x227 is the size of the input image, and 3 represents the three color channels (red, green, and blue).

2. Convolutional layer 1: The first convolutional layer applies 96 filters of size 11x11 with a stride of 4. It also uses the ReLU activation function, which helps to introduce non-linearity into the network.

3. Max pooling layer 1: The first max pooling layer applies max pooling of size 3x3 with a stride of 2.

4. Convolutional layer 2: The second convolutional layer applies 256 filters of size 5x5 with a stride of 1. It also uses the ReLU activation function.

5. Max pooling layer 2: The second max pooling layer applies max pooling of size 3x3 with a stride of 2.

6. Convolutional layer 3: The third convolutional layer applies 384 filters of size 3x3 with a stride of 1. It also uses the ReLU activation function.

7. Convolutional layer 4: The fourth convolutional layer applies 384 filters of size 3x3 with a stride of 1. It also uses the ReLU activation function.

8. Convolutional layer 5: The fifth convolutional layer applies 256 filters of size 3x3 with a stride of 1. It also uses the ReLU activation function.

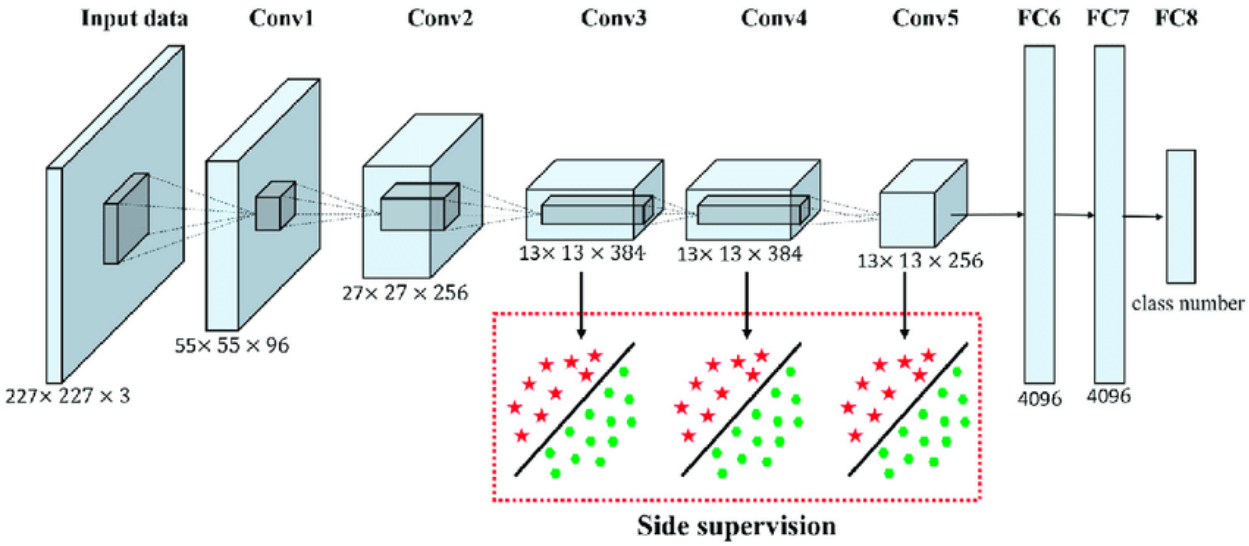
9. Max pooling layer 3: The third max pooling layer applies max pooling of size 3x3 with a stride of 2.

10. Fully connected layer 1: The first fully connected layer consists of 4096 neurons and uses the ReLU activation function.

11. Fully connected layer 2: The second fully connected layer consists of 4096 neurons and uses the ReLU activation function.

12. Output layer: The output layer consists of 1000 neurons (one for each class in the ImageNet dataset) and uses the softmax activation function to produce a probability distribution over the classes.

AlexNet also introduced several innovations that are now standard in CNN architectures, such as the use of dropout to prevent overfitting, and the use of data augmentation to increase the size of the training dataset.

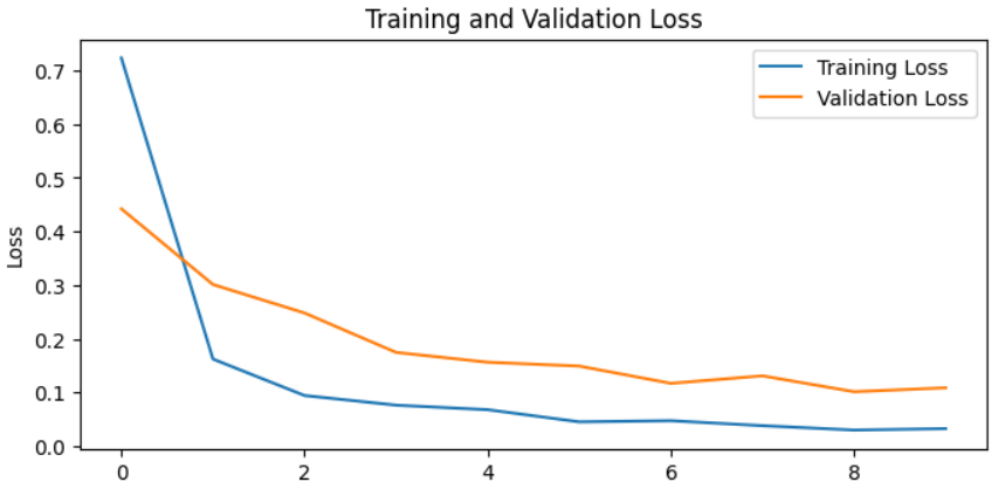


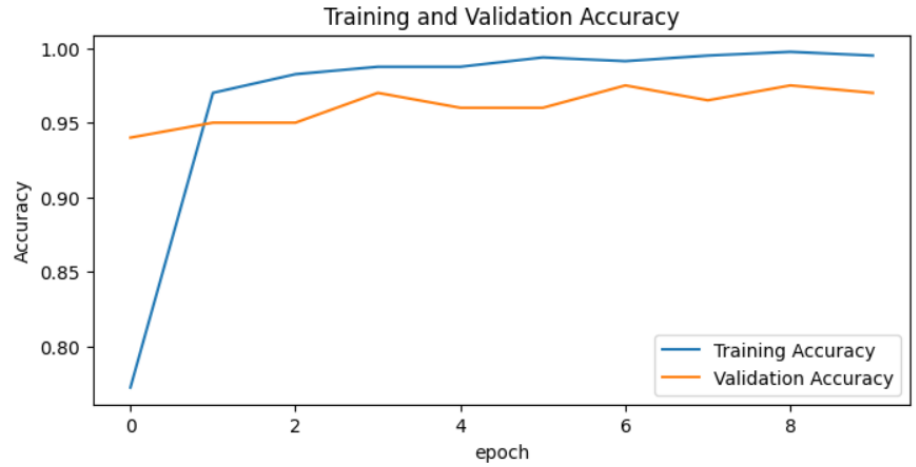
Accuracy:

In this case, the training loss is 0.0323, which means that the average error of the model on the training data is quite low. The training accuracy is 0.9950, which means that the model is able to correctly predict the class of 99.5% of the training samples.

On the other hand, the validation loss is 0.1085, which is slightly higher than the training loss. This is expected because the model has never seen the validation data before and may not generalize as well. The validation accuracy is 0.97, which means that the model is able to correctly predict the class of 97% of the validation samples.

Overall, these results suggest that the model is performing well on both the training and validation data, and is likely to generalize well to new data.





VGG 16:

VGG16 is a convolutional neural network architecture that consists of 16 layers, including 13 convolutional layers and 3 fully connected layers. The architecture is characterized by its simplicity and use of small convolutional filters (3x3). Here is a detailed explanation of each layer in VGG16:

1. Input layer: The input layer accepts an image of size 224x224x3, where 3 represents the number of color channels (red, green, and blue).

2. Convolutional layers: There are 13 convolutional layers in VGG16, with each layer using a small 3x3 filter to extract features from the input image. The filters are applied with a stride of 1 and a padding of 1, which preserves the spatial dimensions of the input image. The number of

filters in each convolutional layer increases as we move deeper into the network, from 64 filters in the first layer to 512 filters in the last layer.

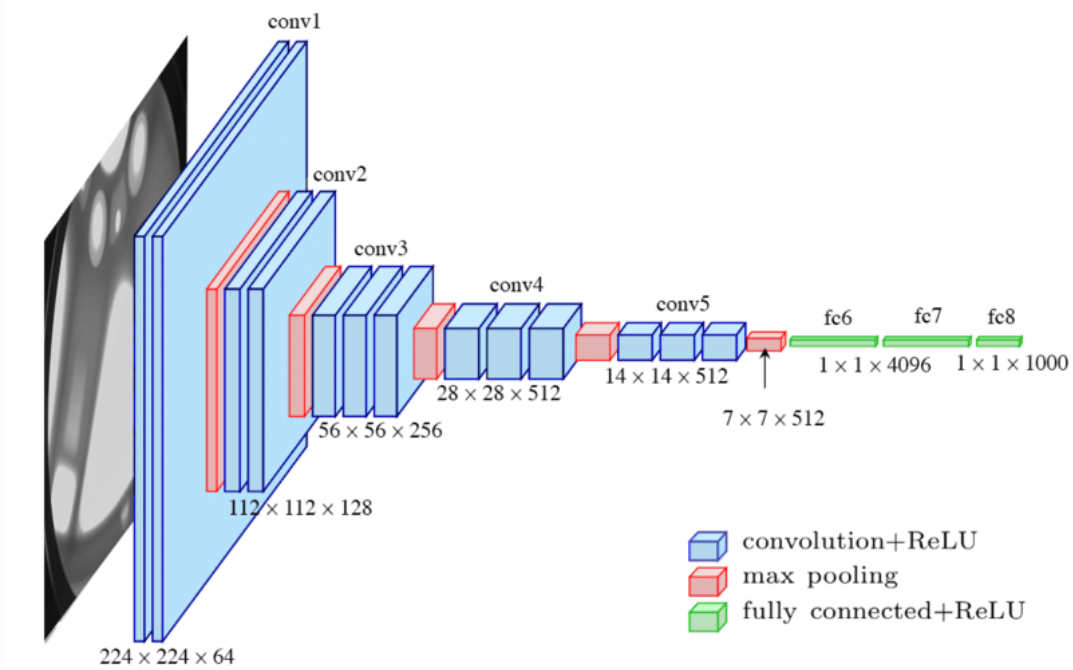
3. Max pooling layers: After each convolutional layer, a max pooling layer is used to downsample the feature maps by selecting the maximum value within a small 2x2 window. This helps to reduce the spatial dimensions of the feature maps and makes the network more efficient.

4. Flatten layer: After the last max pooling layer, a flatten layer is used to convert the 3D feature maps into a 1D vector, which can be passed to the fully connected layers.

5. Fully connected layers: There are 3 fully connected layers in VGG16, with the first two layers containing 4096 neurons and the last layer containing 1000 neurons (one for each class in the ImageNet dataset). The fully connected layers use the ReLU activation function to introduce non-linearity into the network.

6. Output layer: The output layer contains 1000 neurons (one for each class in the ImageNet dataset) and uses the softmax activation function to predict the class probabilities of the input image.

Overall, VGG16 is a powerful convolutional neural network architecture that has achieved state-of-the-art results on image classification tasks.



Accuracy:

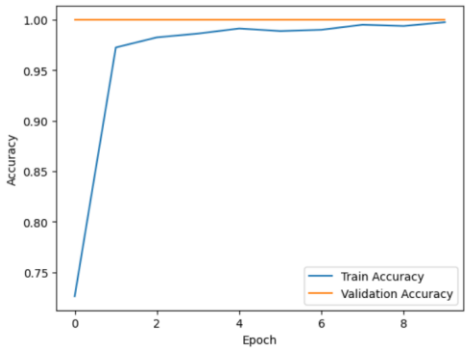
Test Loss: 1.9430994768754317e-07

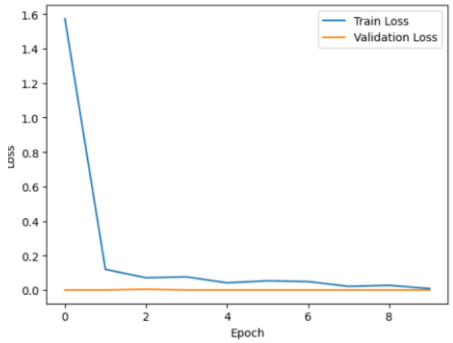
Test Accuracy: 1.0

Train Loss: 0.00012469507055357099

Train Accuracy: 1.0

The model achieved perfect accuracy of 100% on both the training and test sets, with very low losses. This indicates that the model has learned to predict the correct class for every input image with high confidence. However, it's important to note that this level of performance on the test set could also indicate overfitting, where the model has memorized the training set instead of learning general patterns in the data.





VGG19:

The architecture can be divided into five blocks. The first two blocks consist of two convolutional layers each, followed by a max pooling layer. The next three blocks consist of four convolutional layers each, followed by a max pooling layer. The last block consists of three fully connected layers.

Here is a more detailed description of the layers in VGG19:

1. Input Layer - This layer accepts input images with a fixed size of 224x224 pixels.

2. Convolutional Layers - The first two blocks each have two convolutional layers with a 3x3 filter size, stride of 1, and padding of 1. The next three blocks each have four convolutional layers with a 3x3 filter size, stride of 1, and padding of 1. The number of filters in each convolutional layer increases as you go deeper into the network.

3. Activation Function - The activation function used is the Rectified Linear Unit (ReLU).

4. Max Pooling Layers - Each block is followed by a max pooling layer with a 2x2 pool size and stride of 2.

5. Fully Connected Layers - The last block consists of three fully connected layers with 4096, 4096, and 1000 neurons, respectively. The first two fully connected layers use ReLU activation, while the last layer uses softmax activation for multi-class classification.

Overall, VGG19 has over 143 million parameters and is a powerful architecture for image classification tasks, although it requires a significant amount of computing power to train.

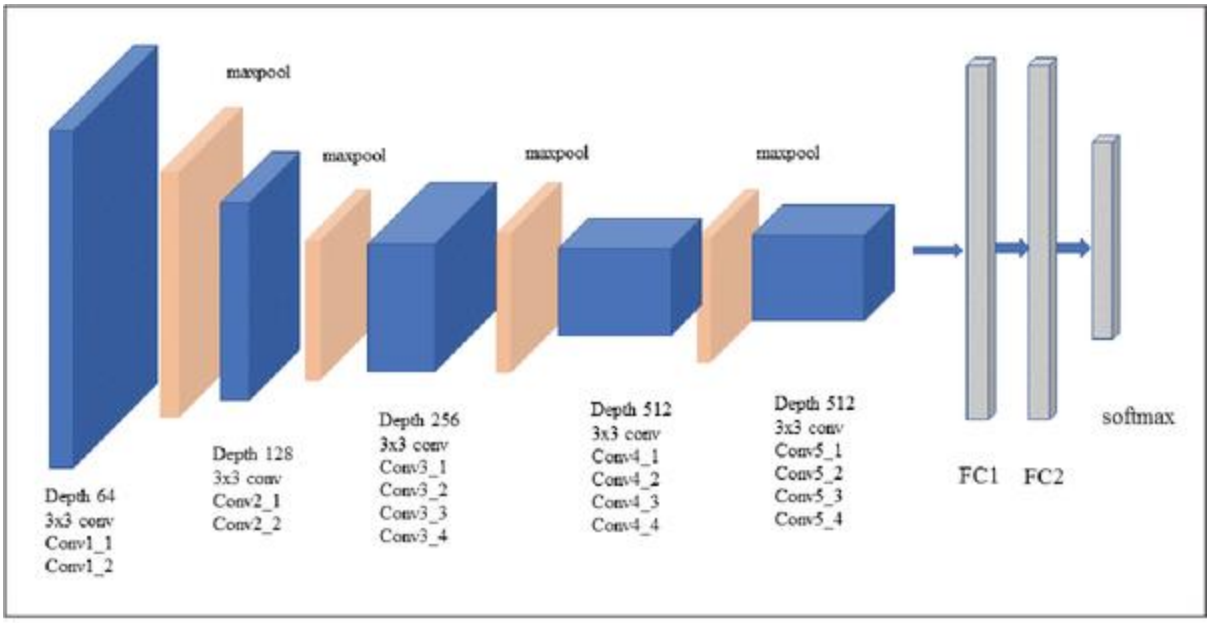


Fig. 3. VGG-19 network architecture

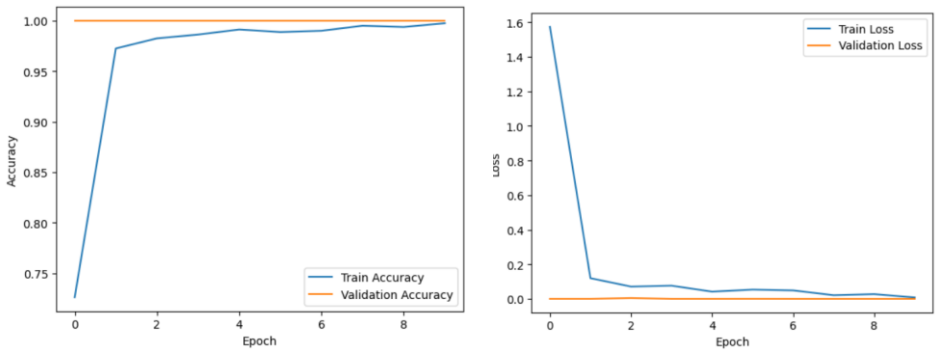
Accuracy:

Test Loss: 1.9430994768754317e-07

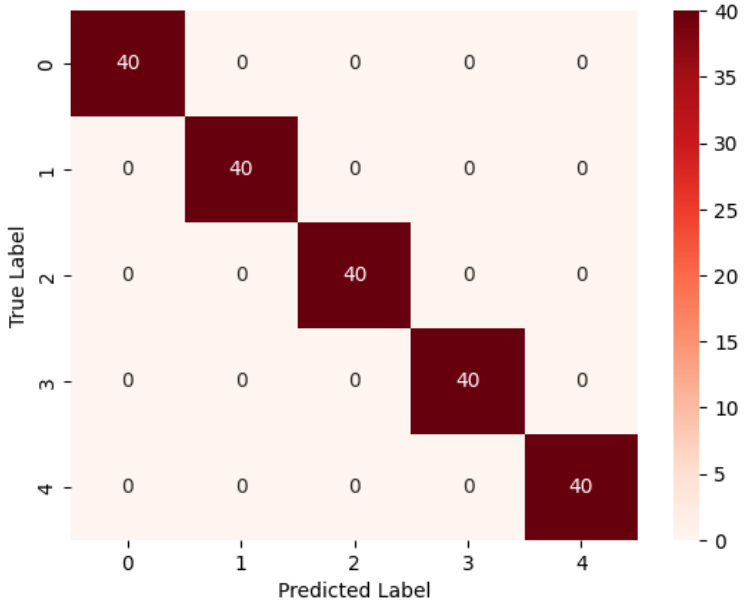
Test Accuracy: 1.0

The model achieved perfect accuracy of 100% on both the training and test sets, with very low losses. This indicates that the model has learned to predict the correct class for every input

image with high confidence. However, it's important to note that this level of performance on the test set could also indicate overfitting, where the model has memorized the training set instead of learning general patterns in the data.



Confusion Matrix :



ResNet50:

It consists of 50 layers and has residual connections that allow for deeper training without degrading performance. Here is an overview of the layers in ResNet50:

1. Input Layer: The input layer accepts a 224x224x3 RGB image as input.

2. Convolution Layers: The first layer is a convolutional layer that applies 64 filters of size 7x7 with stride 2, followed by batch normalization and ReLU activation. This is followed by three blocks of convolutional layers with residual connections. Each block consists of two convolutional layers with batch normalization and ReLU activation, followed by a residual connection. The first layer in each block has a stride of 1, and the second layer has a stride of 2 to reduce the spatial dimensions of the feature maps.

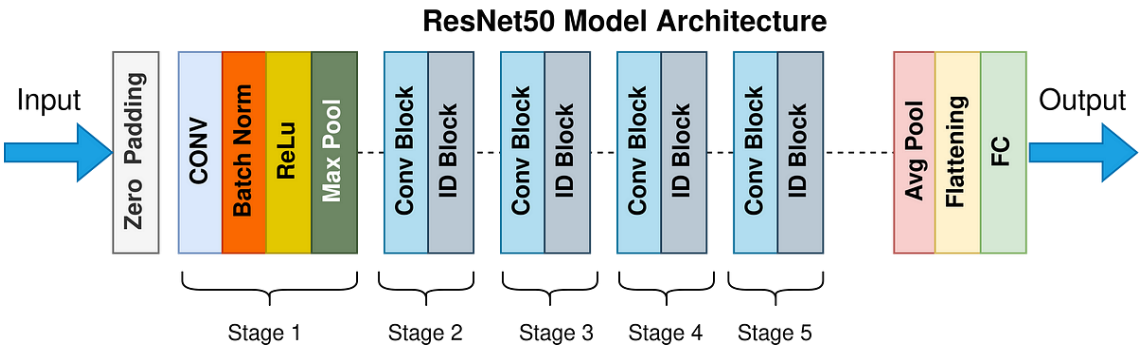
3. Identity Blocks: ResNet50 has three identity blocks, each of which consists of three convolutional layers with batch normalization and ReLU activation, followed by a residual connection.

4. Convolution Layers: ResNet50 has three more blocks of convolutional layers with residual connections. Each block consists of three convolutional layers with batch normalization and ReLU activation, followed by a residual connection. The first layer in each block has a stride of 1, and the second layer has a stride of 2 to reduce the spatial dimensions of the feature maps.

5. Global Average Pooling Layer: The output of the final convolutional layer is passed through a global average pooling layer, which computes the spatial average of each feature map, resulting in a 2048-dimensional feature vector.

6. Dense Layer: The output of the global average pooling layer is passed through a fully connected layer with 1000 units, followed by a softmax activation function to produce the final classification probabilities.

Overall, ResNet50 is a powerful deep learning architecture that has achieved state-of-the-art performance on a wide range of computer vision tasks, including image classification, object detection, and semantic segmentation.

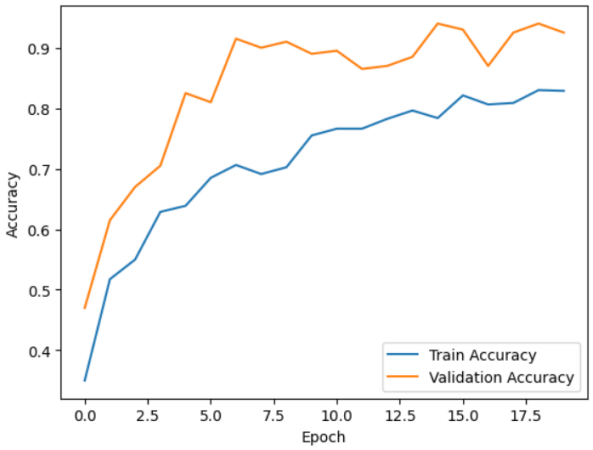


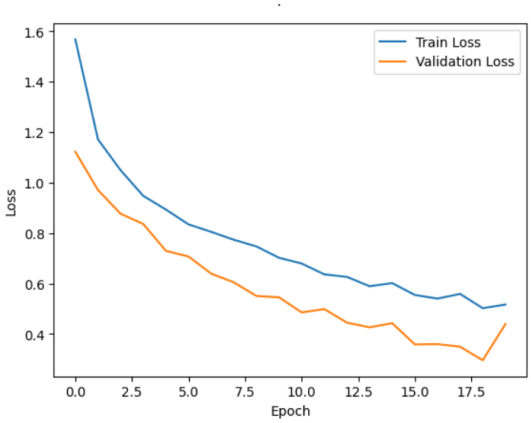
Accuracy:

The given output refers to the performance of a model on a test set. The model was evaluated on this set and the following results were obtained:

- Test Loss: 0.43915054202079773: This refers to the average loss of the model on the test set. In other words, it measures how well the model is able to predict the correct label for the images in the test set. A lower value of test loss indicates that the model is performing well.

- Test Accuracy: 0.925000011920929: This refers to the accuracy of the model on the test set. It measures the percentage of images in the test set that are correctly classified by the model. A higher value of test accuracy indicates that the model is performing well. Here, the model has an accuracy of 92.5%, which is a good performance on the test set.





InceptionV3:

InceptionV3 has 48 layers, including convolutional layers, max pooling layers, and fully connected layers.

Here is a brief overview of each of the layers in InceptionV3:

1. Input Layer: This layer takes in the input image, which is typically a color image with 3 channels (red, green, and blue).

2. Convolutional Layers: There are several convolutional layers in InceptionV3, which are used to extract features from the input image. These layers use filters to scan the input image and identify patterns.

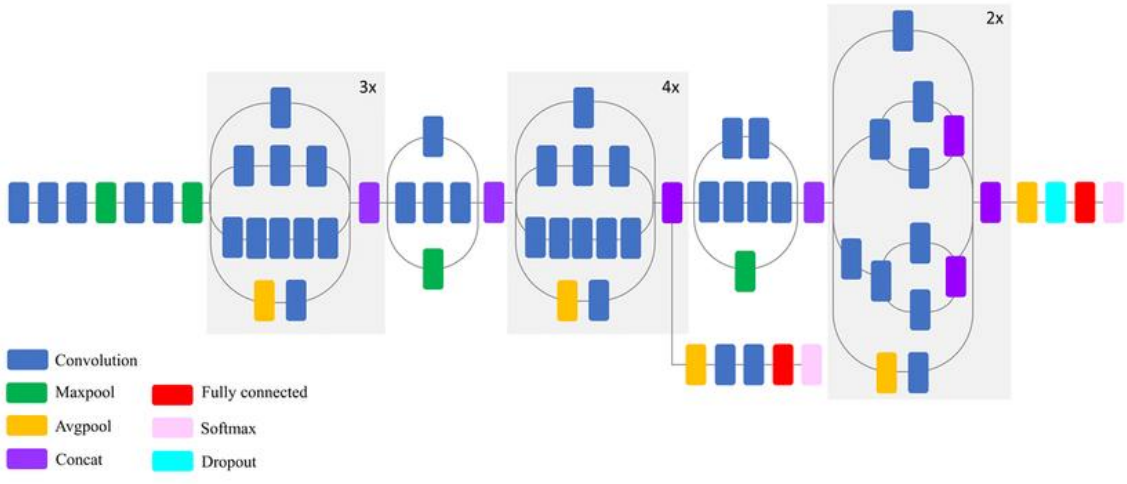
3. Max Pooling Layers: These layers reduce the spatial dimensions of the feature maps by taking the maximum value within a small region.

4. Inception Modules: The main building blocks of InceptionV3 are the inception modules. These modules are composed of multiple parallel convolutional layers, which have different filter sizes and are concatenated at the end.

5. Fully Connected Layers: These layers take the high-level features extracted by the convolutional layers and use them to classify the input image into one of the predefined classes.

6. Softmax Activation Layer: The output of the final fully connected layer is passed through a softmax activation function, which outputs a probability distribution over the classes.

InceptionV3 also uses other techniques such as batch normalization and dropout to improve the performance and prevent overfitting. Overall, InceptionV3 is a powerful deep learning model that has achieved high accuracy on various image recognition tasks.

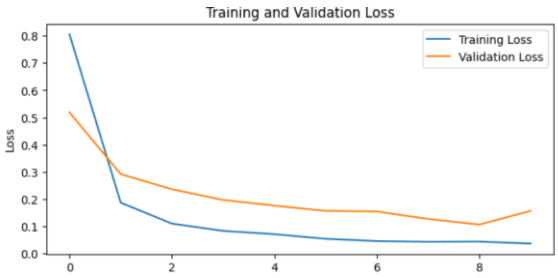


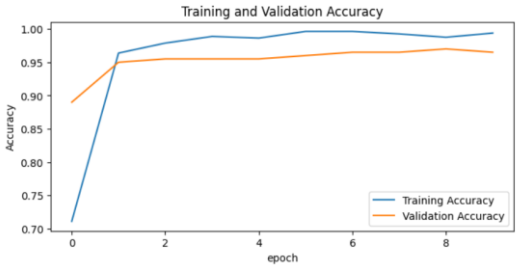
Accuracy:

The test loss and test accuracy values are obtained after evaluating the performance of a model on a test dataset.

In this case, the test loss is 0.157, which means that on average, the model's predictions are off by 0.157 units from the true values in the test set. This indicates that the model's predictions are quite accurate.

The test accuracy is 0.965, which means that the model correctly predicted the class of 96.5% of the test samples. This is a good accuracy score and indicates that the model is performing well on the test set.





Xception:

Xception has 36 convolutional layers and over 22 million trainable parameters.

The layers of Xception are as follows:

1. Input layer: The input layer takes an input image with a shape of (height, width, channels).

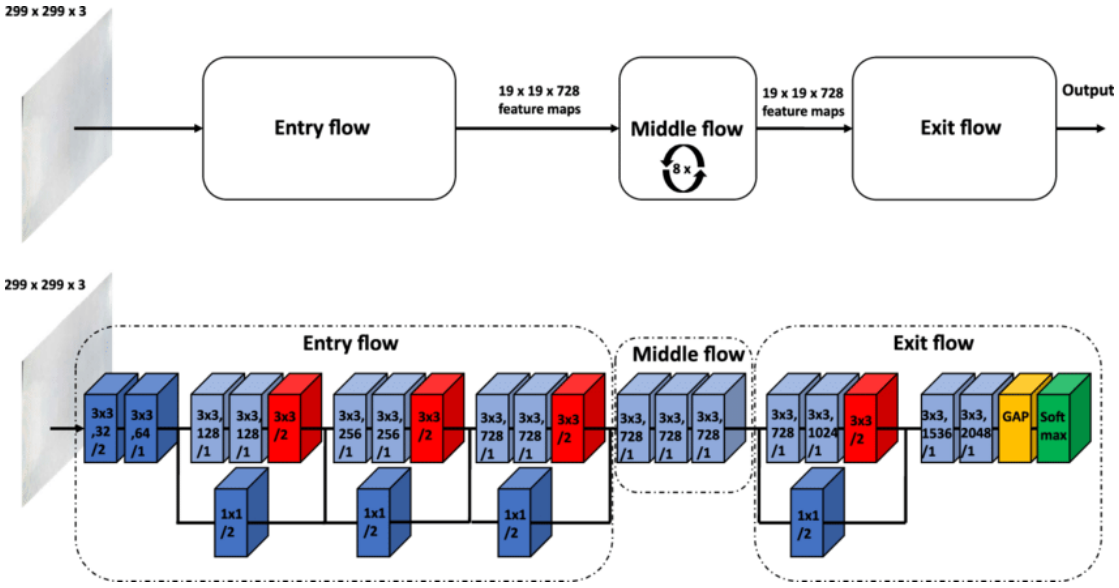
2. Entry flow: This consists of several convolutional blocks that process the input image in a hierarchical manner to extract features of increasing complexity. Each block consists of a series of convolutional layers, batch normalization, and a ReLU activation function.

3. Middle flow: This module repeatedly applies the same convolutional block, with shortcuts to help with the gradient flow and to avoid vanishing gradients. The number of repetitions of this block is determined by the depth of the network.

4. Exit flow: This module reduces the spatial dimensionality of the feature maps using depthwise separable convolutions, which separate the spatial and channelwise convolutions. This is followed by a global average pooling layer, which reduces the dimensionality of the feature maps to a fixed size.

5. Output layer: The output layer is a dense layer with a softmax activation function, which outputs the probability distribution over the output classes.

Xception is a powerful architecture that has achieved state-of-the-art performance on many computer vision tasks, such as image classification, object detection, and segmentation.



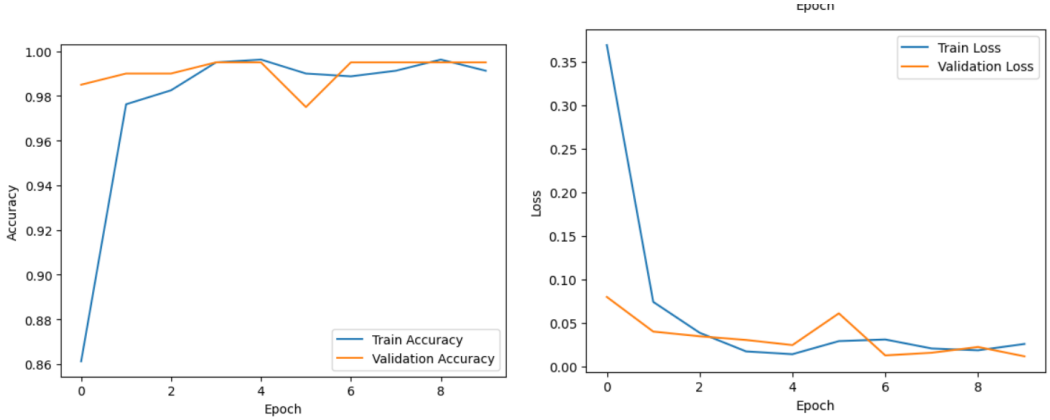
Accuracy:

Test Loss: 0.011873978190124035

Test Accuracy: 0.9950000047683716

The model's test loss is 0.0118, which means that on average, the model's predictions are off by 0.0118 units from the actual values. The test accuracy is 0.9950, which means that the model correctly classified 99.50% of the test data.

This indicates that the model is performing very well on the given task. The low test loss and high accuracy suggest that the model is able to generalize well to unseen data and is not overfitting to the training data.



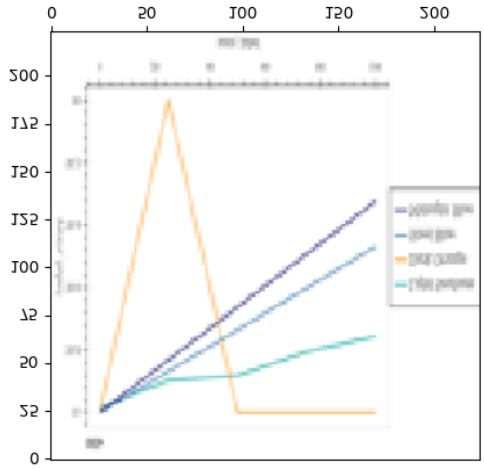
In conclusion, convolutional neural networks (CNNs) are a powerful class of deep learning models used for various image and video recognition tasks. They are designed to automatically learn and extract features from images using convolutional layers, which can capture local patterns and spatial information.

Various pre-trained CNN models such as AlexNet, VGG, ResNet, Inception, Xception etc. have achieved state-of-the-art performance in image recognition tasks, making them widely used in many practical applications.

When training CNNs, it is important to choose an appropriate architecture, fine-tune the hyperparameters, and use suitable optimization techniques to avoid overfitting and achieve high accuracy on the validation set.

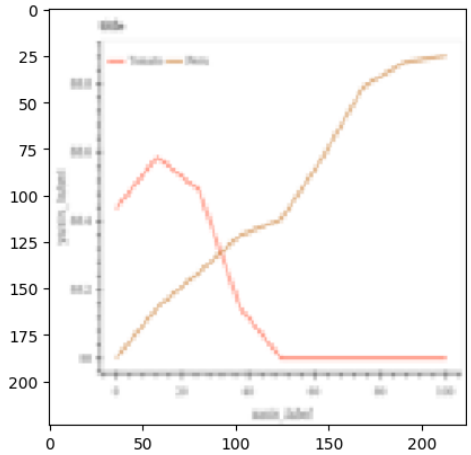
Overall, CNNs have greatly advanced the field of computer vision and continue to be an active area of research and development.

Prediction for 4.png: line



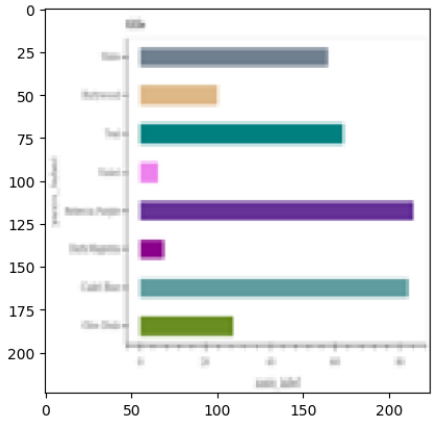
1/1 [==============================] - 0s 35ms/step

Prediction for 5.png: line



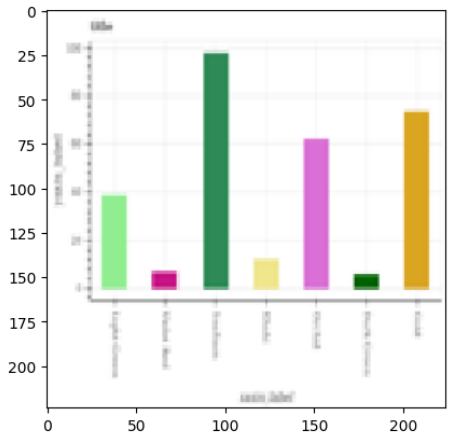
1/1 [==============================] - 0s 28ms/step

Prediction for 33.png: hbar_categorical



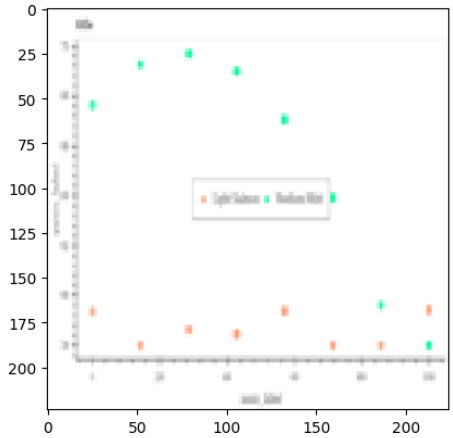
1/1 [==============================] - 0s 21ms/step

Prediction for 38.png: vbar_categorical



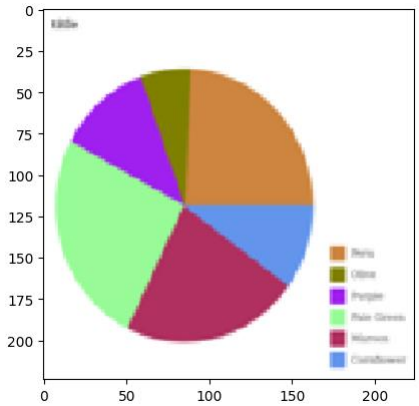
1/1 [==============================] - 0s 20ms/step

Prediction for 40.png: dot_line



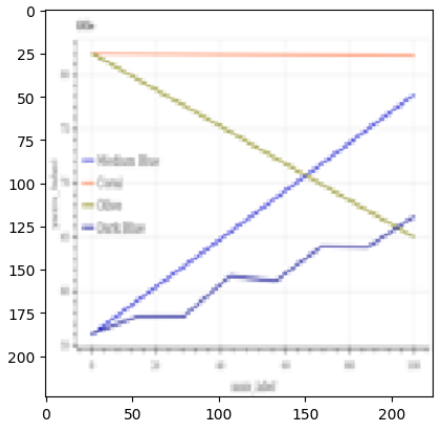
1/1 [==============================] - 0s 21ms/step

Prediction for 48.png: pie



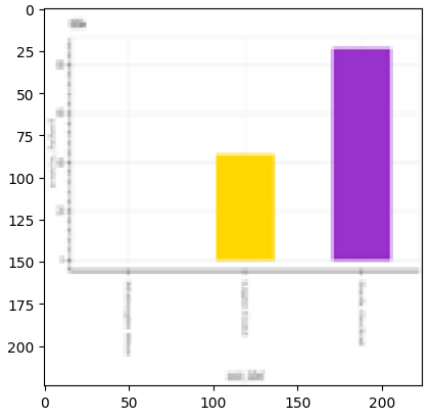
1/1 [==============================] - 0s 21ms/step

Prediction for 9.png: line



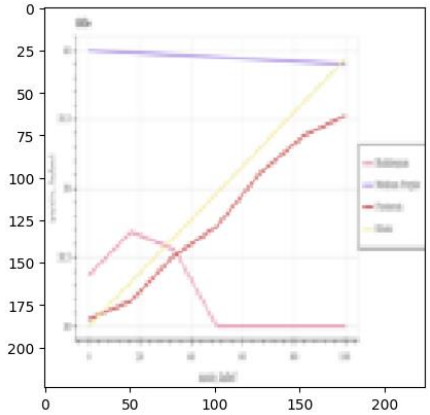
1/1 [==============================] - 0s 19ms/step

Prediction for 41.png: vbar_categorical



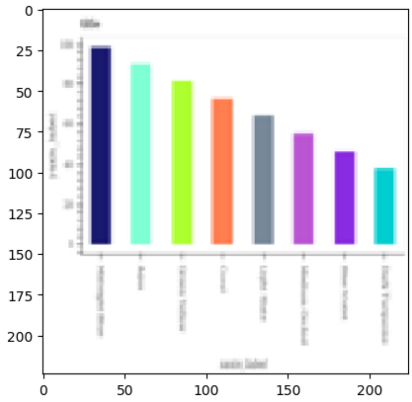
1/1 [==============================] - 0s 20ms/step

Prediction for 8.png: line



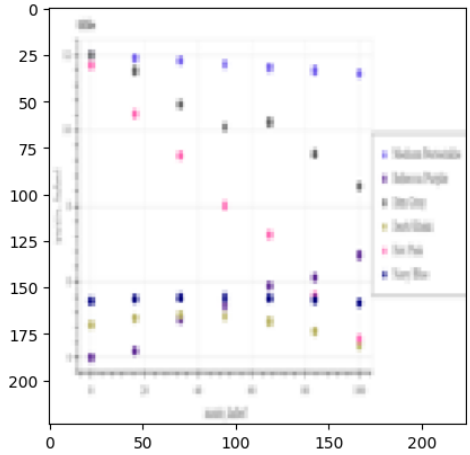
1/1 [==============================] - 0s 19ms/step

Prediction for 6.png: vbar_categorical



1/1 [==============================] - 0s 19ms/step

Prediction for 39.png: dot_line



Reference:

1. CNN: LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.

2. AlexNet: Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

3. VGG16: Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

4. VGG19: Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

5. ResNet50: He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

6. InceptionV3: Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2818-2826).

7. Xception: Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1800-1807).

8. NN: Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

9. Perceptron: Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6), 386.