ann-1

May 8, 2024

1 Write a python Program for Bidirectional Associative Memory with two pairs of vectors

Bidirectional Associative Memory (BAM) is a supervised learning model in Artificial Neural Network. This is hetero-associative memory, for an input pattern, it returns another pattern which is potentially of a different size.

BAM is a memory system designed to create associations between pairs of patterns. If it learns a pair of patterns (e.g., pattern A with pattern B), it can later retrieve pattern B when given pattern A, and vice versa.

Limitations of BAM:

Storage capacity of the BAM: In the BAM, stored number of associations should not be exceeded the number of neurons in the smaller layer.

Incorrect convergence: Always the closest association may not be produced by BAM

Bidirectional Associative Memory (BAM) is a type of neural network designed to store and retrieve pairs of patterns. It allows for the association of one pattern with another in such a way that presenting one pattern will automatically recall its associated pattern. This concept has applications in areas like pattern recognition, error correction, and associative learning.

The relationships between patterns are stored in a weight matrix.

BAM can be used in various domains where associative learning is relevant, such as pattern recognition, language processing, and associative learning tasks.

Translation and Language Processing: BAM can be used to create associative memories between words in different languages. Given a word in one language, BAM can recall its corresponding word in another language, enabling simple translation tasks.

[2]:
```python
import numpy as np

# Define two sets of patterns: Set A and Set B
# Set A: Input Patterns
x1 = np.array([1, 1, 1, 1, 1, 1]).reshape(6, 1)
x2 = np.array([-1, -1, -1, -1, -1, -1]).reshape(6, 1)

# Set B: Target Patterns
y1 = np.array([1, 1]).reshape(2, 1)
```

```python
y2 = np.array([-1, -1]).reshape(2, 1)

# Calculate the weight matrices: W and W_T
W = np.dot(x1, y1.T) + np.dot(x2, y2.T)
W_T = np.dot(y1, x1.T) + np.dot(y2, x2.T)

# Testing Phase
# Test for Input Patterns: Set A
print("Testing for input patterns: Set A")
def testInputs(x, weight):
    # Compute the output pattern
    y = np.dot(weight, x)
    y[y < 0] = -1
    y[y >= 0] = 1
    return y

print("\nOutput of input pattern 1:")
print(testInputs(x1, W_T))
print("\nOutput of input pattern 2:")
print(testInputs(x2, W_T))

# Test for Target Patterns: Set B
print("\nTesting for target patterns: Set B")
def testTargets(y, weight):
    # Compute the output pattern
    x = np.dot(weight, y)
    x[x <= 0] = -1
    x[x > 0] = 1
    return x

print("\nOutput of target pattern 1:")
print(testTargets(y1, W))
print("\nOutput of target pattern 2:")
print(testTargets(y2, W))
```

```
Testing for input patterns: Set A

Output of input pattern 1:
[[1]
 [1]]

Output of input pattern 2:
[[-1]
 [-1]]

Testing for target patterns: Set B
```

```
Output of target pattern 1:
[[1]
 [1]
 [1]
 [1]
 [1]
 [1]]

Output of target pattern 2:
[[-1]
 [-1]
 [-1]
 [-1]
 [-1]
 [-1]]
```

```
[ ]:
```

ann-2

May 8, 2024

1 Write a Python program to plot a few activation functions that are being used in neural networks

The activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

Why do we need Non-linear activation function? A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

The two main categories of activation functions are:

Linear Activation Function

Non-linear Activation Functions

Sigmoid :

It is a function which is plotted as 'S' shaped graph.

Equation : A = 1/(1 + e-x)

Nature : Non-linear

Value Range : 0 to 1

Uses : Usually used in output layer of a binary classification

Relu:

It Stands for Rectified linear unit. It is the most widely used activation function. Chiefly implemented in hidden layers of Neural network.

Equation :- A(x) = max(0,x). It gives an output x if x is positive and 0 otherwise.

Value Range :- [0, inf)

Nature :- non-linear

RELU learns much faster than sigmoid and Tanh function.

Tanh:

Tanh stands for Tangent Hyperbolic function

Value Range :- -1 to +1

Nature :- non-linear

Softmax:

The softmax function is also a type of sigmoid function but is handy when we are trying to handle multi- class classification problems.

Nature :- non-linear

Uses :- Usually used when trying to handle multiple classes. the softmax function was commonly found in the output layer of image classification problems.

Output:- The softmax function is ideally used in the output layer of the classifier where we are actually trying to attain the probabilities to define the class of each input.

Elu:

Exponential Linear Units (ELUs)

Exponential Linear Units are activation functions that are used to speed up the training of deep neural networks. The main advantage of the ELUs is that they can alleviate the vanishing gradient problem by using identity for positive values and also improves the learning characteristics.

Swish :

Swish Activation function as an alternative to the popular ReLU activation function.

Swish is smooth and differentiable everywhere, unlike ReLU, which has a discontinuity at $= 0$

x=0. This smoothness can lead to better gradient flow during backpropagation.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

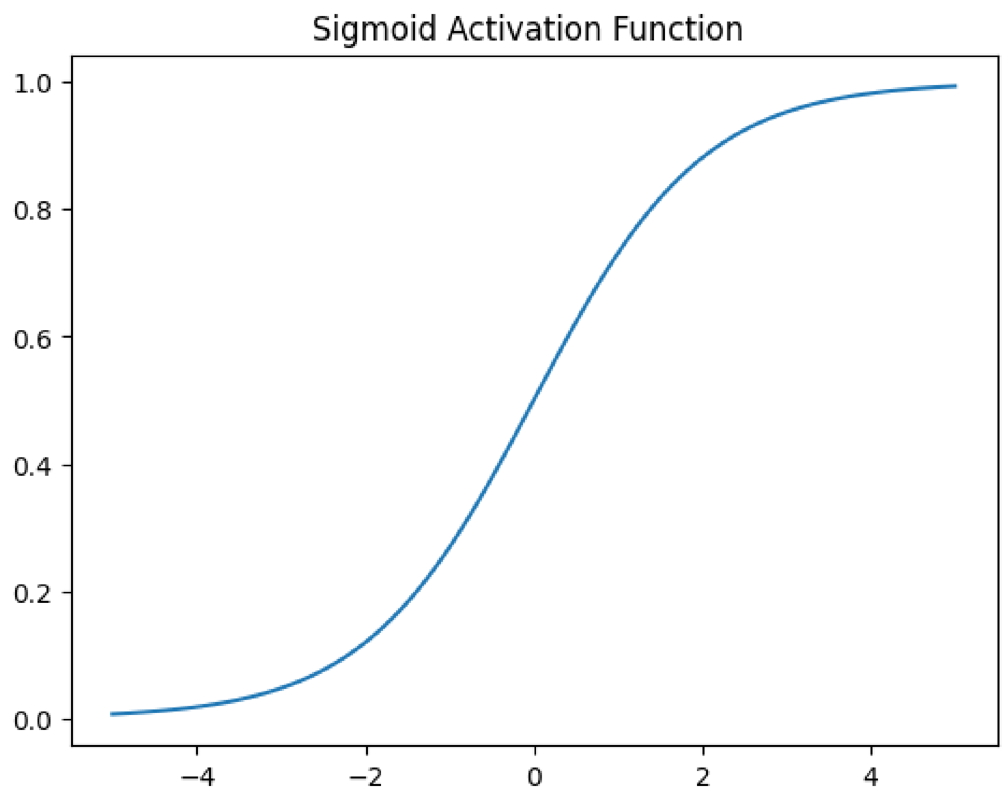
[2]:
```
def sigmoid(x):
    return 1/(1+np.exp(-x))

def relu(x):
    return np.maximum(0,x)

def tanh(x):
    return np.tanh(x)

def softmax(x):
    exp_x = np.exp(x-np.max(x))
    return exp_x / np.sum(exp_x,axis=0)

def elu(x, alpha=1.0):
    return np.where(x >= 0, x, alpha * (np.exp(x) - 1))

def swish(x, beta=1):
    return x / (1 + np.exp(-beta * x))
```

[3]:
```
x = np.linspace(-5,5,100)
x
```

[3]:
```
array([-5.        , -4.8989899 , -4.7979798 , -4.6969697 , -4.5959596 ,
       -4.49494949, -4.39393939, -4.29292929, -4.19191919, -4.09090909,
       -3.98989899, -3.88888889, -3.78787879, -3.68686869, -3.58585859,
       -3.48484848, -3.38383838, -3.28282828, -3.18181818, -3.08080808,
       -2.97979798, -2.87878788, -2.77777778, -2.67676768, -2.57575758,
       -2.47474747, -2.37373737, -2.27272727, -2.17171717, -2.07070707,
       -1.96969697, -1.86868687, -1.76767677, -1.66666667, -1.56565657,
       -1.46464646, -1.36363636, -1.26262626, -1.16161616, -1.06060606,
       -0.95959596, -0.85858586, -0.75757576, -0.65656566, -0.55555556,
       -0.45454545, -0.35353535, -0.25252525, -0.15151515, -0.05050505,
        0.05050505,  0.15151515,  0.25252525,  0.35353535,  0.45454545,
        0.55555556,  0.65656566,  0.75757576,  0.85858586,  0.95959596,
        1.06060606,  1.16161616,  1.26262626,  1.36363636,  1.46464646,
        1.56565657,  1.66666667,  1.76767677,  1.86868687,  1.96969697,
        2.07070707,  2.17171717,  2.27272727,  2.37373737,  2.47474747,
        2.57575758,  2.67676768,  2.77777778,  2.87878788,  2.97979798,
        3.08080808,  3.18181818,  3.28282828,  3.38383838,  3.48484848,
        3.58585859,  3.68686869,  3.78787879,  3.88888889,  3.98989899,
        4.09090909,  4.19191919,  4.29292929,  4.39393939,  4.49494949,
        4.5959596 ,  4.6969697 ,  4.7979798 ,  4.8989899 ,  5.        ])
```

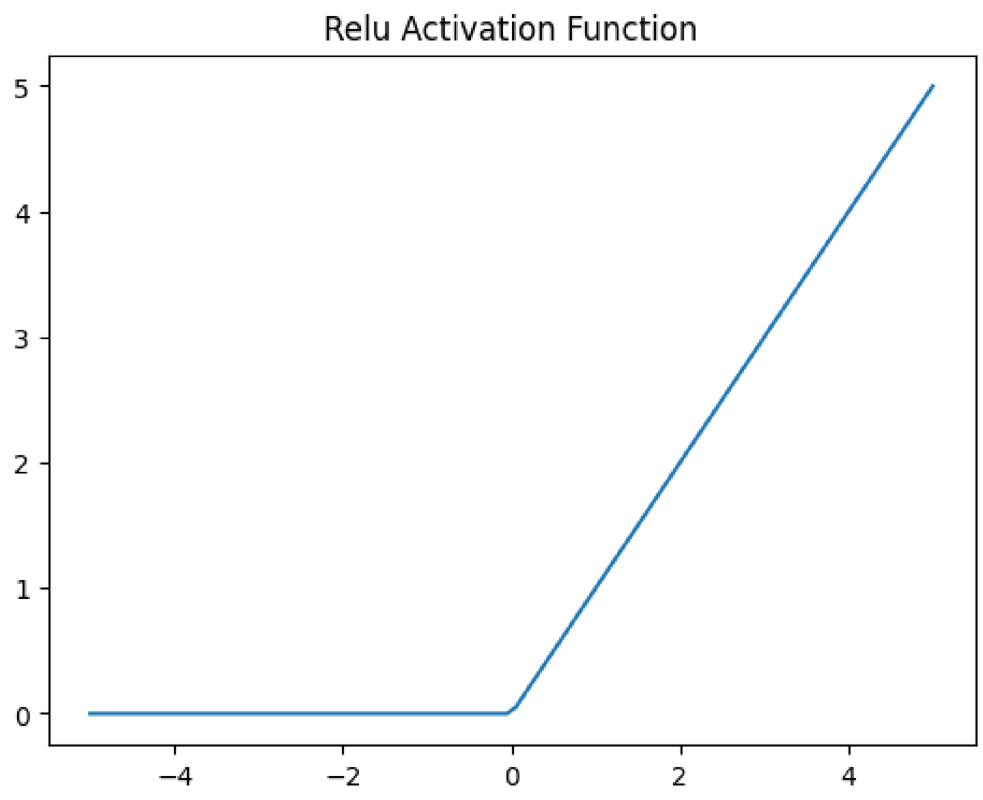
[4]:
```
plt.plot(x,sigmoid(x))
plt.title('Sigmoid Activation Function ')
```

[4]:
```
Text(0.5, 1.0, 'Sigmoid Activation Function ')
```



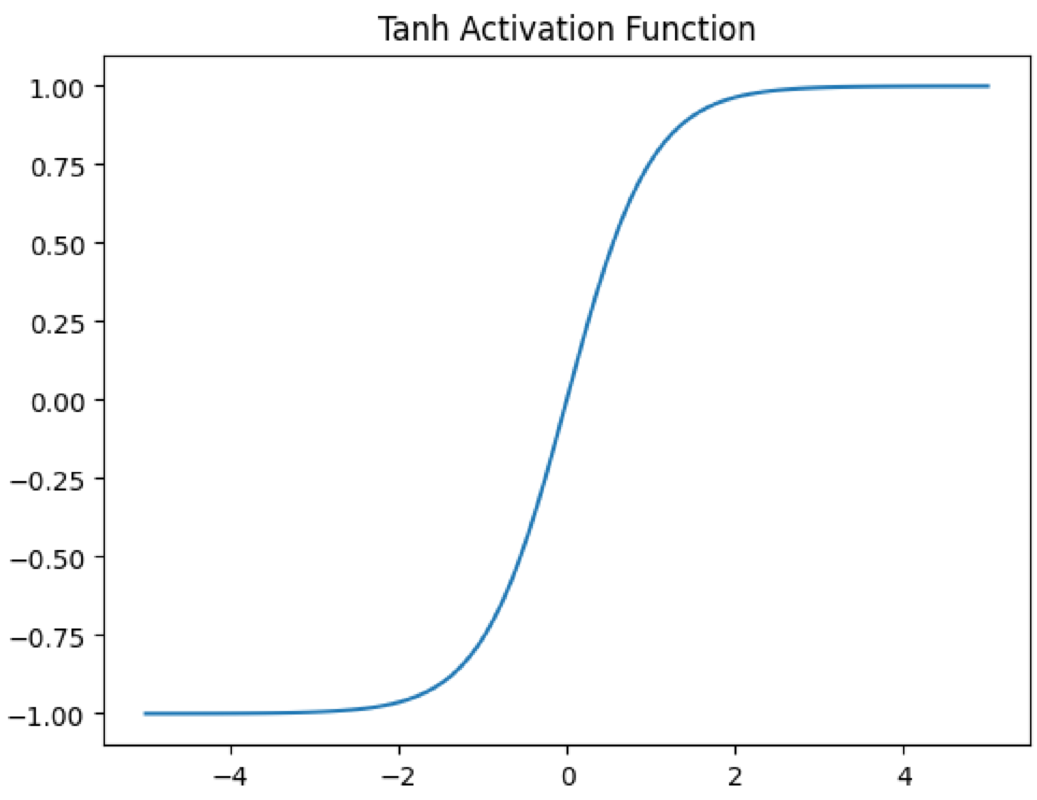
```
[5]: plt.plot(x,relu(x))
     plt.title('Relu Activation Function')
```

```
[5]: Text(0.5, 1.0, 'Relu Activation Function')
```



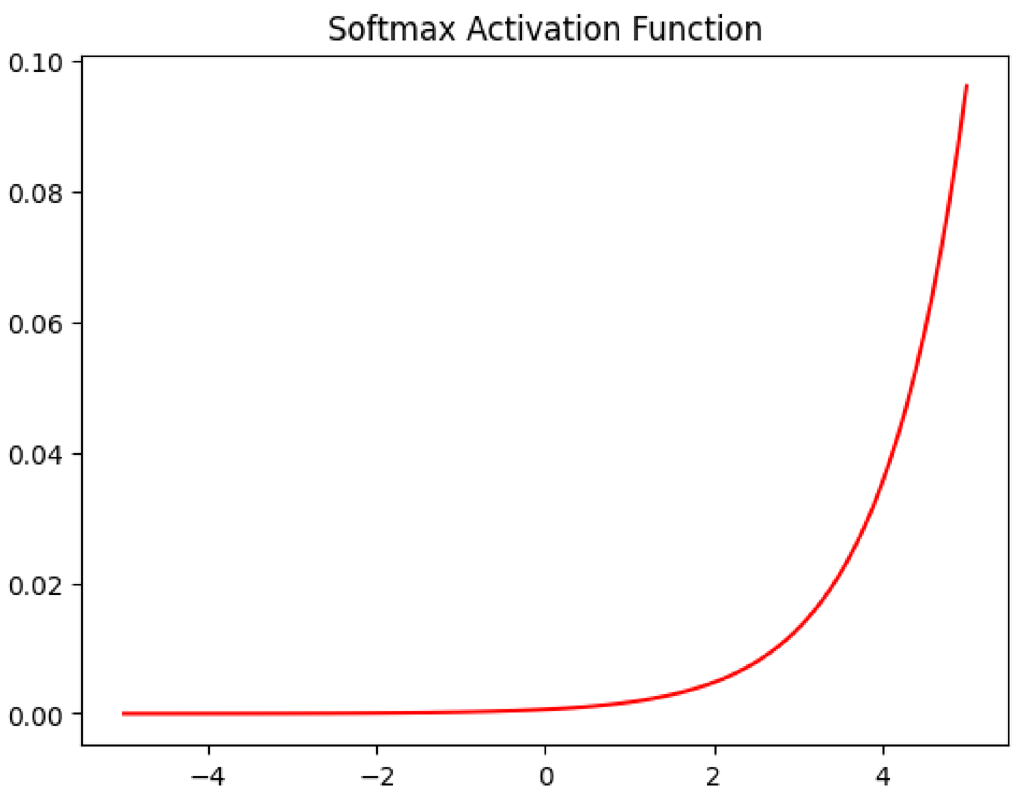
```
[6]: plt.plot(x,tanh(x))
     plt.title('Tanh Activation Function')
```

```
[6]: Text(0.5, 1.0, 'Tanh Activation Function')
```



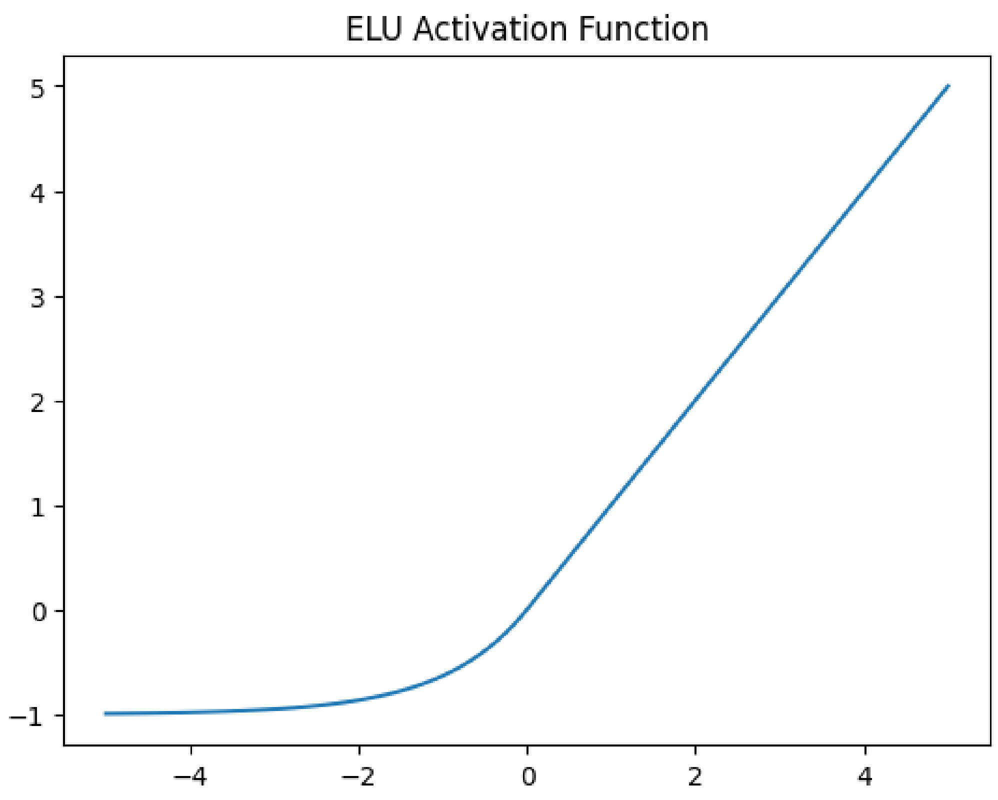
```
[7]: plt.plot(x,softmax(x) , color = 'red')
     plt.title('Softmax Activation Function')
```

```
[7]: Text(0.5, 1.0, 'Softmax Activation Function')
```



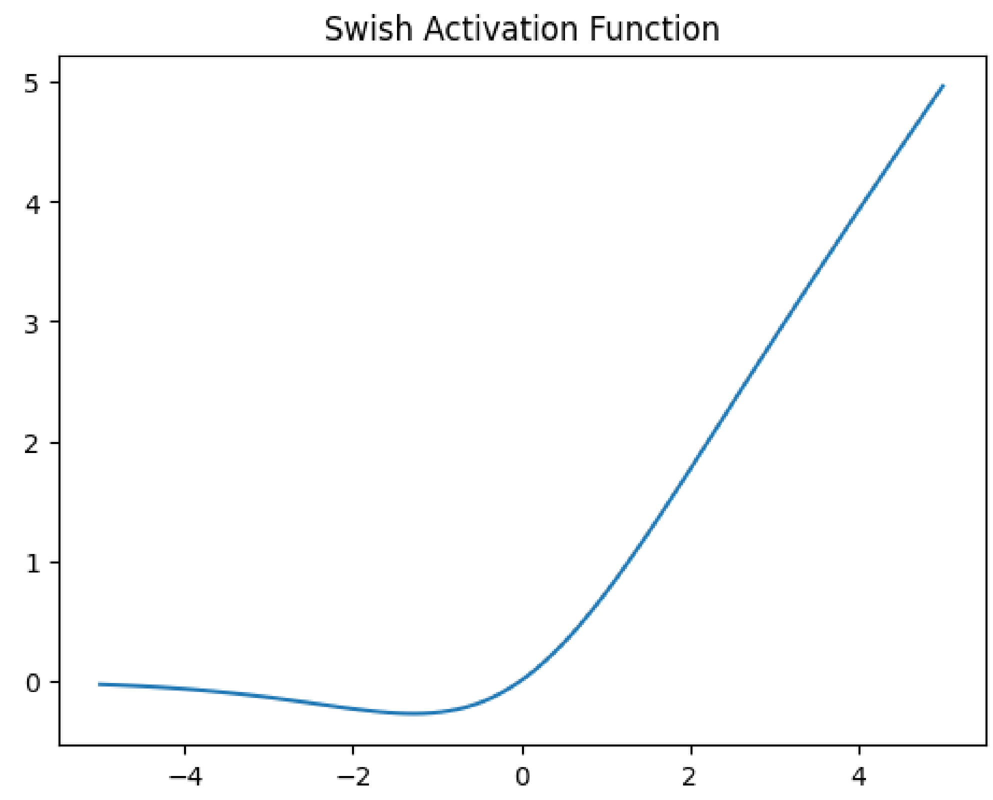
```
[8]: plt.plot(x, elu(x))
     plt.title('ELU Activation Function')
```

```
[8]: Text(0.5, 1.0, 'ELU Activation Function')
```



```
[9]: plt.plot(x, swish(x))
     plt.title('Swish Activation Function')
```

```
[9]: Text(0.5, 1.0, 'Swish Activation Function')
```


Swish Activation Function
5
4
3
2
1
0
−4
−2
0
2
4

ann-3

May 8, 2024

1 Generate ANDNOT function using McCulloch-Pitts neural net by a python program.

2 Choose Any one Code

3 MP MODEL

The inputs of the McCulloch-Pitts neuron could be either 0 or 1.

It has a threshold function as an activation function.

So, the output signal yout is 1 if the input ysum is greater than or equal to a given threshold value, else 0

Limitations

While the McCulloch-Pitts model was groundbreaking, it has several limitations:

Simplicity: It deals only with binary inputs and outputs, which doesn't capture the complexity of real-world data or biological neurons.

No Learning: The model doesn't have learning or adaptation capabilities; it doesn't adjust weights based on experience.

No Time Dependency: The model does not account for time-based processes or recurrent connections.

bias is a learnable parameter that allows models to make flexible predictions by shifting the activation functions' output

4 Perceptron

A perceptron is a fundamental building block of neural networks, modeled after a simple neuron.

It takes a weighted sum of its inputs, applies a linear transformation, and uses an activation function to produce an output, typically for binary classification tasks.

The perceptron serves as a basic example of how neural networks process and learn from data.

[ ]:

```

```

[4]:
```
import numpy as np
def and_not(inputs,weights):
    print('ANDNOT Truth Table')
    print()

    print('X1\tX2\t Y')
    print('------------------')

    for i in inputs:
        # weighted_sum = i[0]*weights[0] + i[1]*weights[1]
        weighted_sum = np.dot(i[0] , weights[0]) + np.dot(i[1] , weights[1])
        if weighted_sum >=1 :
            print(i[0],"\t",i[1],"\t",1)
        else :
            print(i[0],"\t",i[1],"\t",0)

inputs = [[0,0],[0,1],[1,0],[1,1]]
weights = [1,-1]
and_not(inputs,weights)
```

```
ANDNOT Truth Table

X1      X2       Y
------------------
0        0       0
0        1       0
1        0       1
1        1       0
```

[ ]:

ann-4

May 8, 2024

1 Write a Python Program using Perceptron Neural Network to recognize even and odd numbers. Given numbers are in ASCII form 0 to 9

[4]:
```
import numpy as np

class Perceptron:
    def predict(self, test_data):
        predictions = np.where(np.sum(test_data.reshape(-1, 1), axis=1) % 2 ==␣
 ↪0, 0, 1)
        return predictions

# Test the code
```

[5]:
```
test_data = np.array([48, 49, 50, 51, 52, 53, 54, 55, 56, 57])
p = Perceptron()
res = p.predict(test_data)
print("Predictions:", res)
```

```
Predictions: [0 1 0 1 0 1 0 1 0 1]
```

[8]:
```
for i , j in zip(test_data,res):
    number = chr(i)
    output = 'even' if j==0 else 'Odd'
    print(f'Number {number} , ASCII value : {i} , Output : {output}')
```

```
Number 0 , ASCII value : 48 , Output : even
Number 1 , ASCII value : 49 , Output : Odd
Number 2 , ASCII value : 50 , Output : even
Number 3 , ASCII value : 51 , Output : Odd
Number 4 , ASCII value : 52 , Output : even
Number 5 , ASCII value : 53 , Output : Odd
Number 6 , ASCII value : 54 , Output : even
Number 7 , ASCII value : 55 , Output : Odd
Number 8 , ASCII value : 56 , Output : even
Number 9 , ASCII value : 57 , Output : Odd
```

[ ]:
```

```

ann-5

May 8, 2024

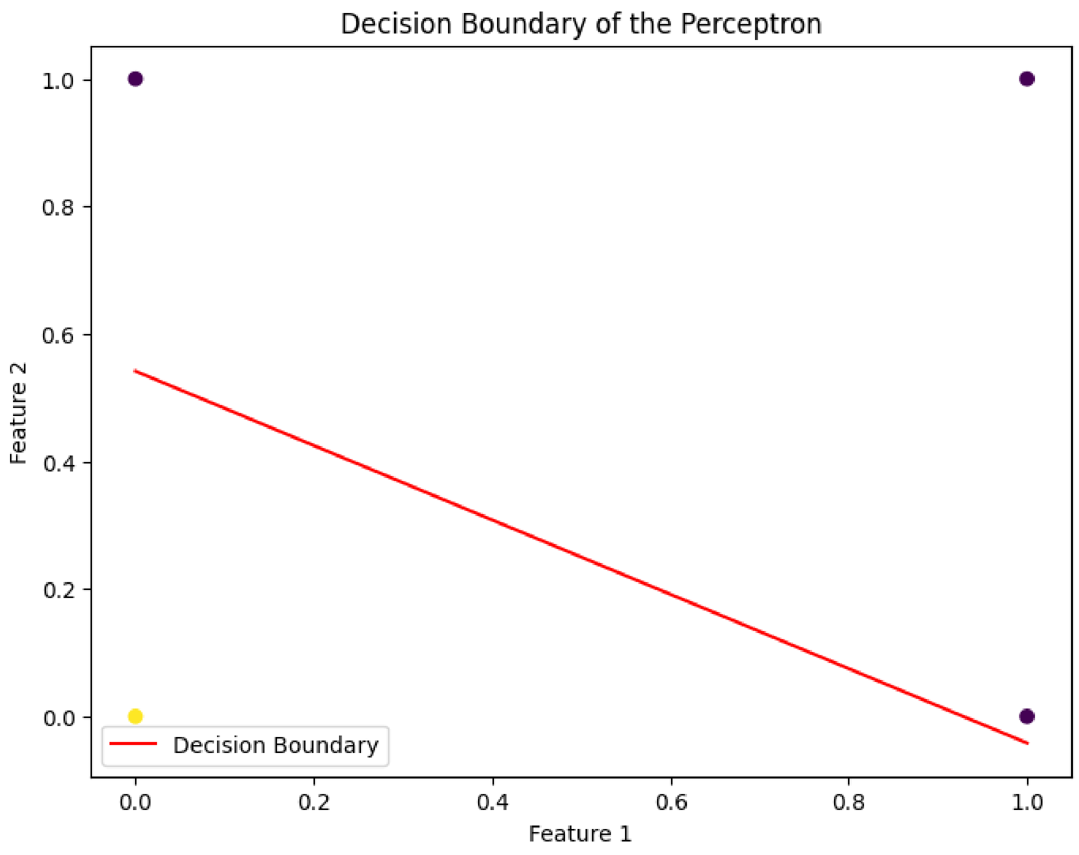
1 With a suitable example demonstrate the perceptron learning law with its decision regions using python. Give the output in graphical form

The Perceptron Learning Rule is a simple algorithm used to train perceptrons for linear classification tasks. It adjusts the weights and bias of a perceptron based on its prediction errors.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

# Perceptron Class
class Perceptron:
    def __init__(self, input_size, learning_rate=0.1, epochs=100):
        self.weights = np.random.rand(input_size)
        self.bias = np.random.rand()
        self.learning_rate = learning_rate
        self.epochs = epochs

    def activation(self, x):
        return 1 if x > 0 else 0

    def predict(self, inputs):
        summation = np.dot(inputs, self.weights) + self.bias
        return self.activation(summation)

    def train(self, training_inputs, labels):
        for epoch in range(self.epochs):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                error = label - prediction
                self.weights += self.learning_rate * error * inputs
                self.bias += self.learning_rate * error

    def plot_decision_region(self, training_inputs, labels):
        # Plotting training data points
        plt.figure(figsize=(8, 6))
```

```python
        plt.scatter(training_inputs[:, 0], training_inputs[:, 1], c=labels, cmap='viridis')

        # Plotting decision boundary
        slope = -self.weights[0] / self.weights[1]
        intercept = -self.bias / self.weights[1]
        x = np.linspace(np.min(training_inputs[:, 0]), np.max(training_inputs[:, 0]), 100)
        y = slope * x + intercept
        plt.plot(x, y, 'r', label='Decision Boundary')

        plt.xlabel('Feature 1')
        plt.ylabel('Feature 2')
        plt.title('Decision Boundary of the Perceptron')
        plt.legend()
        plt.show()


# Example Usage
training_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
labels = np.array([1, 0, 0, 0])

# Create a Perceptron
perceptron = Perceptron(input_size=2)

# Train the Perceptron
perceptron.train(training_inputs, labels)

# Test the trained Perceptron
test_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
for inputs in test_inputs:
    print(f"Input: {inputs}, Predicted Output: {perceptron.predict(inputs)}")

# Plot the decision region
perceptron.plot_decision_region(training_inputs, labels)
```

```
Input: [0 0], Predicted Output: 1
Input: [0 1], Predicted Output: 0
Input: [1 0], Predicted Output: 0
Input: [1 1], Predicted Output: 0
```


Decision Boundary of the Perceptron
Feature 1
Feature 2
Decision Boundary
0.0
0.2
0.4
0.6
0.8
1.0

```
[ ]:
```

ann-9

May 8, 2024

1 Write a python program to illustrate ART neural network

[3]:
```python
import numpy as np

def initialize_weights(input_dim):
    # Initialize weights uniformly and normalize
    weights = np.ones(input_dim) / input_dim
    return weights

def calculate_similarity(input_pattern, weights):
    # Calculate similarity as the normalized dot product (cosine similarity)
    return np.minimum(input_pattern, weights).sum() / np.sum(input_pattern)

def update_weights(input_pattern, weights, learning_rate=0.1):
    # Update weights using the weighted geometric mean
    return (learning_rate * np.minimum(input_pattern, weights) +
            (1 - learning_rate) * weights)

def ART_neural_network(input_patterns, vigilance, learning_rate=0.1):
    num_patterns, input_dim = input_patterns.shape
    categories = []

    for pattern in input_patterns:
        matched_category = None
        # Check existing categories for a match
        for category in categories:
            similarity = calculate_similarity(pattern, category["weights"])
            if similarity >= vigilance:
                matched_category = category
                break

        if matched_category is None:
            # Create a new category if none matched
            weights = initialize_weights(input_dim)
            matched_category = {"weights": weights, "patterns": []}
            categories.append(matched_category)

        # Add the pattern to the matched category
        matched_category["patterns"].append(pattern)
        # Update the category's weights with the new pattern
        matched_category["weights"] = update_weights(
            pattern, matched_category["weights"], learning_rate)

    return categories

# Example usage
input_patterns = np.array([[1, 0, 1, 0], [0, 1, 0, 1], [1, 1, 1, 0]])
vigilance = 0.8

categories = ART_neural_network(input_patterns, vigilance)

# Display the learned categories
for i, category in enumerate(categories):
    print(f"Category {i+1}:")
    print("Patterns:")
    for pattern in category["patterns"]:
        print(pattern)
    print("Weights:")
    print(category["weights"])
    print()
```

```
Category 1:
Patterns:
[1 0 1 0]
Weights:
[0.25  0.225 0.25  0.225]

Category 2:
Patterns:
[0 1 0 1]
Weights:
[0.225 0.25  0.225 0.25 ]

Category 3:
Patterns:
[1 1 1 0]
Weights:
[0.25  0.25  0.25  0.225]
```

[ ]:

ann-11

May 8, 2024

1 11. Write a python program to design a Hopfield Network which stores 4 vectors

2 Hopfield Network

A Hopfield Network is a type of recurrent artificial neural network that is used to store and recall patterns

Consists of one layer of 'n' fully connected recurrent neurons

It is generally used in performing auto-association and optimization tasks

The neurons in a Hopfield Network are binary; they can be in one of two states: 0 (off) or 1 (on).

Neurons update their state based on a specific rule, typically using a threshold function

A common method for training the network is Hebbian learning, where the weights are adjusted based on a set of patterns.

Hebbian learning is a learning rule in neural networks that suggests that the connection strength between two neurons increases when they are activated simultaneously.

[4]:
```python
import numpy as np

# Define the 4 vectors to be stored in the Hopfield Network
vectors = np.array([[1, 1, -1, -1],
                    [-1, -1, 1, 1],
                    [1, -1, 1, -1],
                    [-1, 1, -1, 1]])

# Initialize the weight matrix
num_neurons = vectors.shape[1]
weights = np.zeros((num_neurons, num_neurons))

# Calculate the weights using Hebbian learning
for vector in vectors:
    outer_product = np.outer(vector, vector)  # Outer product for Hebbian␣
 ↪learning
    weights += outer_product
```

```python
# Set the diagonal to 0 (no self-connections)
np.fill_diagonal(weights, 0)

# Define the activation function
def activation(x):
    return np.where(x >= 0, 1, -1)

# Define the Hopfield network function
def hopfield(input_vector, weights, iterations=10):
    # Iteratively update the network to reach a stable state
    for _ in range(iterations):
        output_vector = activation(np.dot(weights, input_vector))
    return output_vector

# Test the Hopfield Network with one of the stored vectors as input
input_vector = np.array([1, -1, 1, -1])  # Example of an initial input
output_vector = hopfield(input_vector, weights)

print("Input vector:")
print(input_vector)
print("Output vector after Hopfield network:")
print(output_vector)
```

```
Input vector:
[ 1 -1  1 -1]
Output vector after Hopfield network:
[ 1 -1  1 -1]
```

[ ]:

```
```

ann-12

May 8, 2024

# How to Train a Neural Network with TensorFlow / Pytorch and evaluation of logistic regression using tensorflow.

[1]:
```
# import tensorflow as tf
# from tensorflow import keras
# from tensorflow.keras import layers

# # Load and preprocess data
# (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
# x_train, x_test = x_train / 255.0, x_test / 255.0

# # Model definition
# model = keras.Sequential([
#     layers.Flatten(input_shape=(28, 28)),  # Flatten input images
#     layers.Dense(128, activation='relu'),  # Hidden layer
#     layers.Dropout(0.2),  # Dropout for regularization
#     layers.Dense(10, activation='softmax')  # Output layer for 10 classes
# ])

# # Model compilation
# model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
 ↪metrics=['accuracy'])

# # Model training
# model.fit(x_train, y_train, epochs=5, validation_split=0.2)

# # Model evaluation
# test_loss, test_accuracy = model.evaluate(x_test, y_test, verbose=2)
# print(f"Test accuracy: {test_accuracy:.2f}")
```

[2]:
```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.layers import Dense
from tensorflow.keras import Sequential
```

[3]:
```
data = load_breast_cancer()
```

[4]:
```
X_train, X_test, y_train, y_test = train_test_split(data.data , data.target ,␣
↪test_size=0.25, random_state=42)
```

[5]:
```
X_train.shape , X_test.shape , y_train.shape , y_test.shape
```

[5]:
```
((426, 30), (143, 30), (426,), (143,))
```

[6]:
```
X_train
```

[6]:
```
array([[1.289e+01, 1.312e+01, 8.189e+01, …, 5.366e-02, 2.309e-01,
        6.915e-02],
       [1.340e+01, 2.052e+01, 8.864e+01, …, 2.051e-01, 3.585e-01,
        1.109e-01],
       [1.296e+01, 1.829e+01, 8.418e+01, …, 6.608e-02, 3.207e-01,
        7.247e-02],
       …,
       [1.429e+01, 1.682e+01, 9.030e+01, …, 3.333e-02, 2.458e-01,
        6.120e-02],
       [1.398e+01, 1.962e+01, 9.112e+01, …, 1.827e-01, 3.179e-01,
        1.055e-01],
       [1.218e+01, 2.052e+01, 7.722e+01, …, 7.431e-02, 2.694e-01,
        6.878e-02]])
```

[7]:
```
st = StandardScaler()
X_train = st.fit_transform(X_train)
X_test = st.fit_transform(X_test)
```

[8]:
```
X_train.shape
```

[8]:
```
(426, 30)
```

[9]:
```
model = Sequential()
model.add(Dense(128,activation='relu' ,  input_shape = (30,)))
model.add(Dense(1,activation='sigmoid'))
```

```
C:\Users\vedan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_qbz5n2
kfra8p0\LocalCache\local-packages\Python39\site-
packages\keras\src\layers\core\dense.py:86: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

[10]:
```
model.summary()
```

Model: "sequential"

```
 Layer (type)                           Output Shape                               ␣
↪Param #

 dense (Dense)                          (None, 128)                                  ␣
↪3,968

 dense_1 (Dense)                        (None, 1)                                      ␣
↪129


 Total params: 4,097 (16.00 KB)


 Trainable params: 4,097 (16.00 KB)


 Non-trainable params: 0 (0.00 B)
```

None means that the batch size is not fixed in the model's architecture and can be set later when the model is run.

#The first Dense layer has 3,968 parameters. This is calculated as

(30×128)+128=3840+128=3968

(30×128)+128=3840+128=3968.

The 128 added comes from the bias term for each neuron.

The second Dense layer has 129 parameters. This comes from

(128×1)+1=128+1=129

(128×1)+1=128+1=129.

Again, the "+1" represents the bias term

[11]:
```
model.compile(loss='binary_crossentropy' , optimizer = 'adam' , metrics =␣
 ↪['accuracy'])
```

Binary cross-entropy is a common loss function for binary classification tasks, where the output is expected to be either 0 or 1.

It measures the divergence between the predicted probability and the true binary label, penalizing misclassifications. The goal of training is to minimize this loss.

The accuracy metric is straightforward: it measures the percentage of correct predictions over all predictions made.

Adam (Adaptive Moment Estimation) is a popular choice because it combines the benefits of two other optimization algorithms (AdaGrad and RMSprop).

It adjusts learning rates adaptively based on gradients and is known for its good performance across various tasks.

[12]:
```
model.fit(X_train,y_train , epochs=10 , validation_data=(X_test,y_test))
```

```
Epoch 1/10
14/14             1s 22ms/step -
accuracy: 0.6901 - loss: 0.6007 - val_accuracy: 0.9371 - val_loss: 0.2864
Epoch 2/10
14/14             0s 5ms/step -
accuracy: 0.8921 - loss: 0.3161 - val_accuracy: 0.9650 - val_loss: 0.1698
Epoch 3/10
14/14             0s 6ms/step -
accuracy: 0.9286 - loss: 0.2104 - val_accuracy: 0.9650 - val_loss: 0.1273
Epoch 4/10
14/14             0s 5ms/step -
accuracy: 0.9525 - loss: 0.1543 - val_accuracy: 0.9720 - val_loss: 0.1057
Epoch 5/10
14/14             0s 5ms/step -
accuracy: 0.9604 - loss: 0.1552 - val_accuracy: 0.9790 - val_loss: 0.0919
Epoch 6/10
14/14             0s 6ms/step -
accuracy: 0.9807 - loss: 0.1190 - val_accuracy: 0.9790 - val_loss: 0.0825
Epoch 7/10
14/14             0s 5ms/step -
accuracy: 0.9799 - loss: 0.1026 - val_accuracy: 0.9860 - val_loss: 0.0754
Epoch 8/10
14/14             0s 5ms/step -
accuracy: 0.9789 - loss: 0.0926 - val_accuracy: 0.9860 - val_loss: 0.0705
Epoch 9/10
14/14             0s 4ms/step -
accuracy: 0.9757 - loss: 0.0987 - val_accuracy: 0.9860 - val_loss: 0.0666
Epoch 10/10
14/14             0s 4ms/step -
accuracy: 0.9801 - loss: 0.0921 - val_accuracy: 0.9860 - val_loss: 0.0637
```

[12]:
```
<keras.src.callbacks.history.History at 0x1b1a4845310>
```

epochs=10:

An epoch is one complete pass through the entire training dataset.

Specifying 10 epochs means the model will go through the training data 10 times, adjusting its weights each time based on the loss and gradients calculated by the optimizer.

[13]:
```
loss , accuracy = model.evaluate(X_test,y_test)
print('Test Loss : ',loss)
print('Test Accuracy : ',accuracy)
```

```
5/5             0s 0s/step -
```

```
accuracy: 0.9814 - loss: 0.0752
Test Loss :  0.06372378021478653
Test Accuracy :  0.9860140085220337
```

[14]:
```
y_pred = model.predict(X_test)
y_pred
```

```
5/5              0s 8ms/step
```

[14]:
```
array([[9.0326011e-01],
       [2.7961724e-03],
       [6.3731954e-02],
       [9.9206311e-01],
       [9.9877328e-01],
       [1.9892698e-06],
       [1.7032802e-05],
       [1.4307170e-01],
       [7.0264971e-01],
       [9.9561769e-01],
       [9.5213825e-01],
       [9.0098292e-02],
       [9.8348665e-01],
       [2.5817543e-01],
       [9.9756938e-01],
       [1.2956608e-02],
       [9.9274290e-01],
       [9.9976707e-01],
       [9.9997520e-01],
       [5.2031607e-04],
       [8.7778890e-01],
       [9.8389065e-01],
       [2.7799329e-05],
       [9.9952620e-01],
       [9.9571592e-01],
       [9.9221468e-01],
       [9.9472421e-01],
       [9.9188697e-01],
       [9.9054694e-01],
       [8.3075830e-04],
       [9.9334449e-01],
       [9.9901521e-01],
       [9.9474019e-01],
       [9.9016017e-01],
       [9.9892765e-01],
       [9.9601507e-01],
       [2.9945546e-01],
       [9.9683821e-01],
       [9.2748851e-03],
       [8.9807260e-01],
       [9.9799728e-01],
       [4.8475288e-02],
       [9.9151194e-01],
       [9.9580765e-01],
       [9.6894908e-01],
       [9.8009109e-01],
       [9.9885076e-01],
       [9.9808377e-01],
       [9.6279496e-01],
       [9.9226487e-01],
       [1.2670841e-02],
       [1.4999480e-04],
       [8.1056511e-01],
       [8.9975202e-01],
       [9.9808782e-01],
       [9.8947698e-01],
       [9.9873912e-01],
       [6.1337971e-07],
       [4.0831488e-01],
       [9.9914187e-01],
       [9.8007345e-01],
       [3.9471651e-04],
       [5.1079249e-05],
       [9.4434774e-01],
       [9.9810541e-01],
       [9.2107481e-01],
       [3.8435170e-03],
       [8.1618691e-06],
       [9.9754351e-01],
       [9.5289934e-01],
       [1.0609056e-01],
       [2.9067965e-02],
       [9.9436051e-01],
       [6.9631554e-02],
       [9.9971682e-01],
       [9.8128164e-01],
       [9.6632069e-01],
       [6.0486364e-01],
       [9.9965447e-01],
       [9.8086321e-01],
       [4.5455992e-02],
       [9.9956268e-01],
       [4.4945437e-01],
       [6.7535286e-05],
       [5.7003818e-02],
       [2.3282316e-02],
       [1.2650962e-02],
       [9.8700281e-03],
       [9.9782550e-01],
       [9.8859966e-01],
       [9.8874420e-01],
       [8.0150485e-01],
       [9.2279583e-01],
       [9.9747163e-01],
       [9.9862534e-01],
       [9.9937165e-01],
       [7.7867310e-04],
       [1.3927210e-03],
       [9.9960464e-01],
       [1.7557584e-02],
       [6.9271140e-02],
       [9.9995542e-01],
       [1.4529071e-03],
       [1.3956904e-02],
       [9.8688632e-01],
       [9.6051800e-01],
       [9.8528606e-01],
       [7.2660350e-06],
       [9.1482532e-01],
       [9.5319599e-01],
       [4.1344974e-02],
       [9.9527621e-01],
       [6.1325043e-01],
       [9.9899898e-07],
       [9.9018753e-01],
       [8.5383126e-06],
       [9.9977905e-01],
       [9.7979033e-01],
       [9.9878597e-01],
       [6.6816756e-03],
       [9.7553384e-01],
       [9.9818456e-01],
       [9.9230820e-01],
       [5.0185737e-03],
       [9.4179964e-01],
       [3.7760517e-04],
       [1.6997973e-02],
       [9.9699420e-01],
       [9.9571502e-01],
       [4.7879521e-04],
       [5.1876174e-05],
       [1.8753920e-03],
              [9.8584455e-01],
              [9.9787354e-01],
              [9.7924328e-01],
              [4.8910797e-02],
              [8.1421250e-01],
              [9.9619395e-01],
              [7.0357001e-01],
              [6.0802117e-02],
              [9.9513108e-01],
              [9.3429800e-05],
              [9.9972105e-01]], dtype=float32)
```

```
[ ]:
```

ann-13

May 8, 2024

1 13. TensorFlow / Pytorch implementation of CNN.

[1]:
```python
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Dense , Conv2D ,  Flatten , MaxPool2D
from tensorflow.keras import Sequential
from tensorflow.keras.utils import to_categorical
```

Conv2D: A convolutional layer used for extracting features from 2D data like images. It applies convolution operations using a defined number of filters (also known as kernels) across the input data.

MaxPool2D: This layer applies max-pooling, which is used to downsample the data by taking the maximum value from a region, reducing the spatial dimensions while keeping important features.

Flatten: This layer takes multidimensional data (like images) and flattens it into a 1D array, making it ready for dense (fully connected) layers.

Dense: A dense (fully connected) layer where every input node is connected to every output node. It is typically used for classification and output layers in neural networks.

[2]:
```python
(x_train,y_train) , (x_test,y_test) = mnist.load_data()
```

[3]:
```python
x_train.shape , x_test.shape , y_train.shape , y_test.shape
```

[3]:
```
((60000, 28, 28), (10000, 28, 28), (60000,), (10000,))
```

[4]:
```python
x_train = x_train.reshape(-1,28,28,1) / 255
x_test = x_test.reshape(-1,28,28,1) / 255
```

Reshaping the Data: x_train = x_train.reshape(-1,28,28,1):

This code reshapes the x_train data to have four dimensions: (number of samples, height, width, channels).

-1 indicates that the reshape should maintain the same number of samples.

This is a flexible way to ensure you don't need to manually specify the number of training samples.

28, 28 are the height and width of each image in the MNIST dataset.

1 represents the number of channels, indicating grayscale images. If these were color images, the channels would be 3 (RGB).

The reshaping is necessary because CNNs work with 2D (or 3D, with multiple color channels) image data. Each filter in a convolutional layer operates on the height, width, and channel(s) of the input data. Normalizing the Data:

/ 255: This operation scales the pixel values from a range of 0-255 to a range of 0-1. This normalization is critical for several reasons:

It makes the model's learning more stable and can speed up training.

It ensures consistent input data, which is important for model convergence.

By bringing all values to a similar scale, you avoid potential issues with different input scales.

[5]:
```
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

The to_categorical function from tensorflow.keras.utils is used to convert class labels (integers) into a one-hot encoded matrix.

A one-hot encoded representation means that each class label is represented as a vector (or an array) with the length equal to the number of classes.

The vector contains all zeros except for a single 1 at the index corresponding to the class label.

[6]:
```
model = Sequential()
model.add(Conv2D(64 , (3,3) , activation='relu' , input_shape = (28 , 28,1)))
model.add(MaxPool2D())
model.add(Conv2D(64,(3,3) , activation='relu'))
model.add(Flatten())
model.add(Dense(128 , activation='relu'))
model.add(Dense(10,activation='softmax'))
```

```
C:\Users\vedan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_qbz5n2
kfra8p0\LocalCache\local-packages\Python39\site-
packages\keras\src\layers\convolutional\base_conv.py:99: UserWarning: Do not
pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model
instead.
  super().__init__(
```

model.add(Conv2D(64, (3,3), activation='relu', input_shape=(28,28,1))):

This line adds a 2D convolutional layer to the model with the following parameters:

64: The number of filters (also known as kernels) in this convolutional layer. More filters can capture more features from the input data.

(3,3): The size of the convolutional kernels. A 3x3 kernel is common in CNNs because it effectively captures spatial features.

activation='relu': Specifies the activation function used after applying the convolution. ReLU (Rectified Linear Unit) is a popular activation function in CNNs because it introduces non-linearity without causing vanishing gradient issues.

input_shape=(28,28,1): This is the shape of the input data. Since MNIST images are 28x28 pixels with a single grayscale channel, this input shape is appropriate.

model.add(MaxPool2D()): Adds a MaxPooling layer to the model.

MaxPooling is used to reduce the spatial dimensions of the feature maps, helping to decrease the computational load and capture higher-level features.

model.add(Flatten()): This layer flattens the 3D feature maps into a 1D vector. This step is necessary before feeding the data into dense (fully connected) layers.

model.add(Dense(128, activation='relu')): This adds a dense layer with 128 units and ReLU activation. Dense layers allow for deeper learning and abstraction, with each unit connected to every other unit in the preceding and following layers.

model.add(Dense(10, activation='softmax')): This is the output layer. It contains 10 units because MNIST has 10 classes (digits 0-9). The softmax activation function ensures that the output is a probability distribution across the 10 classes, indicating the model's confidence in its predictions.

[7]:
```
model.summary()
```

```
Model: "sequential"


 Layer (type)                       Output Shape                     ␣
↪Param #

 conv2d (Conv2D)                    (None, 26, 26, 64)                   ␣
↪640

 max_pooling2d (MaxPooling2D)       (None, 13, 13, 64)                   ␣
↪  0

 conv2d_1 (Conv2D)                  (None, 11, 11, 64)                ␣
↪36,928

 flatten (Flatten)                  (None, 7744)                         ␣
↪  0

 dense (Dense)                      (None, 128)                      ␣
↪991,360

 dense_1 (Dense)                    (None, 10)                         ␣
↪1,290


Total params: 1,030,218 (3.93 MB)
```

```
Trainable params: 1,030,218 (3.93 MB)

Non-trainable params: 0 (0.00 B)
```

[8]:
```
model.compile(loss='categorical_crossentropy',metrics=['accuracy'] , optimizer␣
↪= 'adam')
```

loss='categorical_crossentropy': The loss function is a measure of how well the model is performing. It calculates the error or difference between the predicted output and the true labels.

"Categorical Cross-Entropy" is used for multi-class classification tasks with one-hot encoded labels, like MNIST (where there are 10 classes representing the digits 0-9). It computes the negative log-likelihood between the predicted probabilities and the true distribution (one-hot encoded labels).

[9]:
```
model.fit(x_train,y_train,epochs=10,validation_data=(x_test,y_test))
```

```
Epoch 1/10
1875/1875              40s 21ms/step -
accuracy: 0.9248 - loss: 0.2416 - val_accuracy: 0.9875 - val_loss: 0.0398
Epoch 2/10
1875/1875              36s 19ms/step -
accuracy: 0.9890 - loss: 0.0348 - val_accuracy: 0.9879 - val_loss: 0.0356
Epoch 3/10
1875/1875              37s 20ms/step -
accuracy: 0.9927 - loss: 0.0230 - val_accuracy: 0.9903 - val_loss: 0.0331
Epoch 4/10
1875/1875              36s 19ms/step -
accuracy: 0.9953 - loss: 0.0152 - val_accuracy: 0.9886 - val_loss: 0.0351
Epoch 5/10
1875/1875              37s 20ms/step -
accuracy: 0.9971 - loss: 0.0102 - val_accuracy: 0.9910 - val_loss: 0.0298
Epoch 6/10
1875/1875              36s 19ms/step -
accuracy: 0.9975 - loss: 0.0079 - val_accuracy: 0.9894 - val_loss: 0.0464
Epoch 7/10
1875/1875              37s 20ms/step -
accuracy: 0.9973 - loss: 0.0078 - val_accuracy: 0.9889 - val_loss: 0.0434
Epoch 8/10
1875/1875              36s 19ms/step -
accuracy: 0.9983 - loss: 0.0051 - val_accuracy: 0.9906 - val_loss: 0.0375
Epoch 9/10
1875/1875              36s 19ms/step -
accuracy: 0.9986 - loss: 0.0043 - val_accuracy: 0.9903 - val_loss: 0.0459
Epoch 10/10
1875/1875              37s 20ms/step -
accuracy: 0.9986 - loss: 0.0039 - val_accuracy: 0.9895 - val_loss: 0.0503
```

```
[9]: <keras.src.callbacks.history.History at 0x1a8815289a0>
```

```
[10]: loss , accuracy = model.evaluate(x_test,y_test)
      print('Test Loss : ',loss)
      print('Test Accuracy : ',accuracy)
```

```
313/313              1s 4ms/step -
accuracy: 0.9876 - loss: 0.0643
Test Loss :  0.05028357729315758
Test Accuracy :  0.9894999861717224
```

```
[ ]:
```

ann-14

May 8, 2024

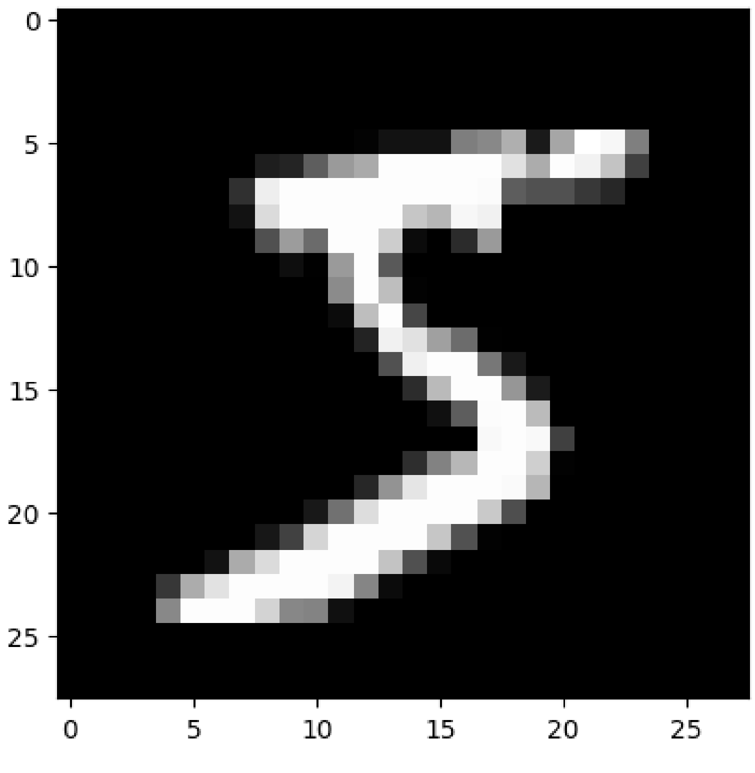
1 MNIST Handwritten Character Detection using Keras and Tensorflow

[11]:
```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Dense , Flatten
from tensorflow.keras import Sequential
import matplotlib.pyplot as plt
```

[12]:
```
(x_train , y_train) , (x_test , y_test) = mnist.load_data()
```

[14]:
```
plt.imshow(x_train[0], cmap='gray')  # Display in grayscale
plt.title(f"Label: {y_train[0]}")  # Display the corresponding label
plt.show()
```

Label: 5



[3]:
```
x_train.shape , y_train.shape , x_test.shape , y_test.shape
```

[3]:
```
((60000, 28, 28), (60000,), (10000, 28, 28), (10000,))
```

[4]:
```
model = Sequential()
model.add(Flatten(input_shape  = (28,28)))
model.add(Dense(128,activation='relu'))
model.add(Dense(64,activation='relu'))
model.add(Dense(10,activation='softmax'))
```

```
C:\Users\vedan\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_qbz5n2
kfra8p0\LocalCache\local-packages\Python39\site-
packages\keras\src\layers\reshaping\flatten.py:37: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

[5]:
```
model.summary()
```

```
Model: "sequential"


 Layer (type)                          Output Shape                          ␣
↪Param #

 flatten (Flatten)                     (None, 784)                               ␣
↪  0

 dense (Dense)                         (None, 128)                           ␣
↪100,480

 dense_1 (Dense)                       (None, 64)                              ␣
↪8,256

 dense_2 (Dense)                       (None, 10)                                ␣
↪650


 Total params: 109,386 (427.29 KB)


 Trainable params: 109,386 (427.29 KB)
 Non-trainable params: 0 (0.00 B)
```

[6]:
```
model.compile(loss='sparse_categorical_crossentropy' , optimizer = 'Adam' ,␣
 ↪metrics=['accuracy'])
```

[9]:
```
model.fit(x_train,y_train , epochs=10 , validation_data=(x_test,y_test))
```

```
Epoch 1/10
1875/1875              8s 4ms/step -
accuracy: 0.8133 - loss: 4.9431 - val_accuracy: 0.9034 - val_loss: 0.4814
Epoch 2/10
1875/1875              7s 4ms/step -
accuracy: 0.9275 - loss: 0.3457 - val_accuracy: 0.9365 - val_loss: 0.2510
Epoch 3/10
1875/1875              5s 3ms/step -
accuracy: 0.9480 - loss: 0.1957 - val_accuracy: 0.9454 - val_loss: 0.1987
Epoch 4/10
1875/1875              5s 3ms/step -
accuracy: 0.9550 - loss: 0.1617 - val_accuracy: 0.9509 - val_loss: 0.1929
Epoch 5/10
1875/1875              5s 3ms/step -
accuracy: 0.9609 - loss: 0.1380 - val_accuracy: 0.9584 - val_loss: 0.1587
Epoch 6/10
1875/1875              6s 3ms/step -
accuracy: 0.9674 - loss: 0.1128 - val_accuracy: 0.9535 - val_loss: 0.1677
Epoch 7/10
1875/1875              5s 3ms/step -
accuracy: 0.9687 - loss: 0.1045 - val_accuracy: 0.9603 - val_loss: 0.1551
Epoch 8/10
1875/1875              8s 4ms/step -
accuracy: 0.9732 - loss: 0.0949 - val_accuracy: 0.9625 - val_loss: 0.1435
Epoch 9/10
1875/1875              5s 3ms/step -
accuracy: 0.9754 - loss: 0.0858 - val_accuracy: 0.9628 - val_loss: 0.1464
Epoch 10/10
1875/1875              6s 3ms/step -
accuracy: 0.9752 - loss: 0.0823 - val_accuracy: 0.9611 - val_loss: 0.1601
```

[9]:
```
<keras.src.callbacks.history.History at 0x2321e092550>
```

[10]:
```
loss , accuracy = model.evaluate(x_test , y_test)
print('Test Loss : ',loss)
print('Test Acuuracy : ',accuracy)
```

```
313/313              1s 2ms/step -
accuracy: 0.9557 - loss: 0.1830
Test Loss :  0.16008248925209045
Test Acuuracy :  0.9610999822616577
```

```
[ ]:
```

Define ANN ?

An Artificial Neural Network (ANN) is a computational model inspired by the human brain's neural structure, designed to process information and learn from data. It consists of interconnected nodes, or "neurons," organized in layers, with each connection having an associated weight.

ANN VS BNN ?

Biological neural networks consist of neurons in the brain and nervous system that communicate through complex electrochemical signals.In contrast, artificial neural networks (ANNs) are mathematical models inspired by these biological systems, implemented in software or hardware. ANNs use simplified units (neurons) with weighted connections and are trained through algorithms to process information or recognize patterns.

Backpropagation Neural Network

A backpropagation neural network (or simply, a neural network that uses backpropagation) is a type of artificial neural network where learning occurs through an algorithm that adjusts the network's weights based on the error of the output. It calculates gradients of the loss function with respect to each weight by applying the chain rule, allowing the network to update weights to minimize the error during training.

ART

Adaptive Resonance Theory (ART) is a framework for neural networks that emphasizes stability and plasticity, allowing them to learn new information while retaining existing knowledge. ART achieves this balance by adjusting how much new patterns must resemble known patterns before they are incorporated into existing categories, enabling continuous learning without catastrophic forgetting.

CNN

A Convolutional Neural Network (CNN) is a type of deep learning neural network designed for processing structured grid-like data, such as images. It utilizes convolutional layers to automatically extract hierarchical features from the input, followed by pooling layers to reduce dimensionality, and dense layers for classification or other tasks. CNNs are commonly used in computer vision applications like image recognition, object detection, and facial recognition.

ASCII

AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE

Upper CASE A-Z : 65 - > 90

Lower Case a-z - > 97 - > 122