```
#Linear Activation Function

import numpy as np
def linear_function(x):
  return 2*x

input = np.array([3, -3, 0, 6])
output = linear_function(input)
print(output)

[ 6 -6  0 12]

#Non linear act func
#Sigmoid Activation Func
import numpy as np
def sigmoid_func(x):
  return 1/(1+np.exp(-x))

input = np.array([3, -3])
output = sigmoid_func(input)
print(output)

[0.95257413 0.04742587]

0.95257413 + 0.04742587

1.0

#Softmax Func
def soft_max(x):
  a = np.exp(x)
  b = a/a.sum()
  return b

print(soft_max([3, -3, 0]) )

[0.95033021 0.00235563 0.04731416]

0.95033021 + 0.00235563 +  0.04731416

1.0

#Tanh Activation func
def tanh_func(x):
  return 2*(1/(1+np.exp(-2 * x)))-1

print(tanh_func(3))
print(tanh_func(-3))
print(tanh_func(0))
print(tanh_func(6))
```

```
0.9950547536867307
-0.9950547536867305
0.0
0.9999877116507956
```

```python
#Relu Activation func
def relu_func(x):
  return max(0, x)

relu_func(3), relu_func(-3), relu_func(9), relu_func(-7)
```

```
(3, 0, 9, 0)
```

```python
# ANDNOT
def and_not_mcp(A, B):
    # weights
    w1 = 1  # for A
    w2 = -1  # for B
    threshold = 1

    # weighted sum
    net_input = w1 * A + w2 * B

    # activation
    output = 1 if net_input >= threshold else 0
    return output

# Test all combinations
inputs = [(0,0), (0,1), (1,0), (1,1)]

print("A B | A AND NOT B")
print("------------------")
for A, B in inputs:
    result = and_not_mcp(A, B)
    print(f"{A} {B} |     {result}")
```

```
A B | A AND NOT B
------------------
0 0 |     0
0 1 |     0
1 0 |     1
1 1 |     0
```

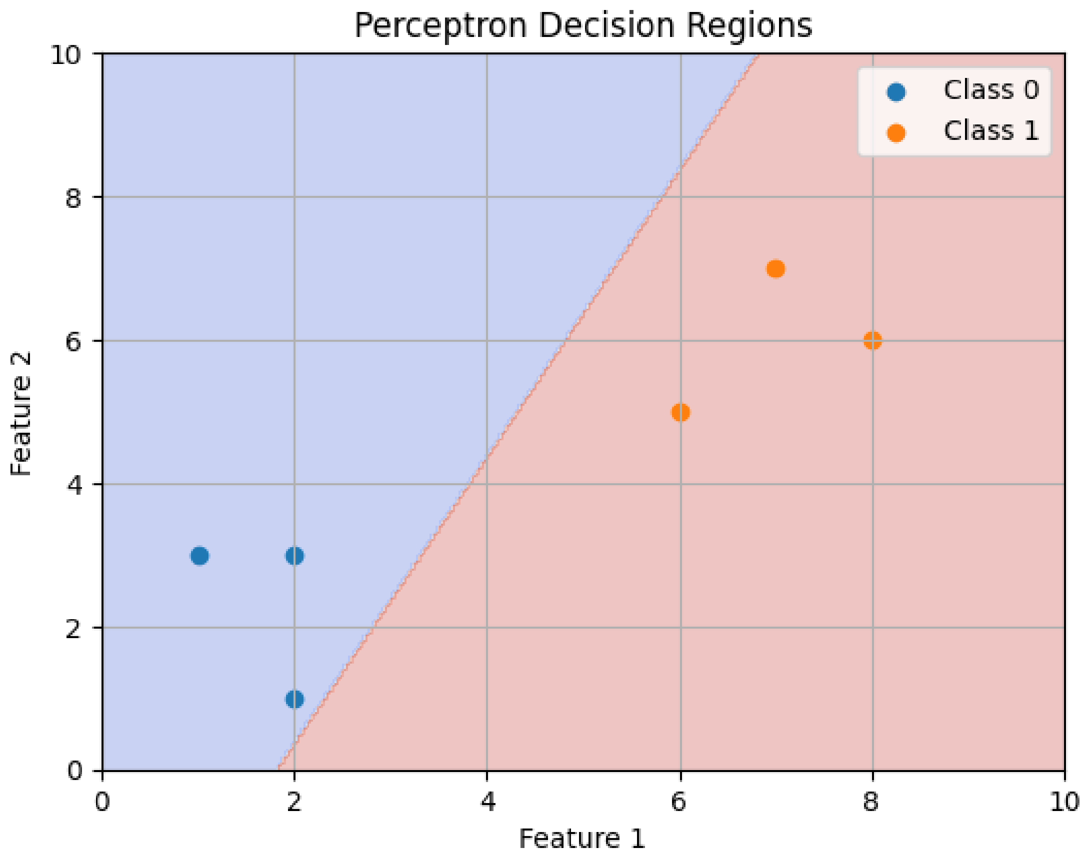
```python
##even and odd numbers

import numpy as np

# Prepare data
X = [list(map(int, format(ord(str(i)), '08b'))) for i in range(10)]
y = [1 if i % 2 == 0 else 0 for i in range(10)]  # 1 = even, 0 = odd
```

```python
# Initialize
w = np.zeros(8)
b = 0

# Train
for _ in range(20):
    for xi, yi in zip(X, y):
        z = np.dot(w, xi) + b
        pred = int(z >= 0)
        w += (yi - pred) * np.array(xi)
        b += (yi - pred)

# Test
for i, xi in enumerate(X):
    pred = int(np.dot(w, xi) + b >= 0)
    print(f"Digit {i} is predicted to be {'even' if pred else
'odd'}.")
```

```
Digit 0 is predicted to be even.
Digit 1 is predicted to be odd.
Digit 2 is predicted to be even.
Digit 3 is predicted to be odd.
Digit 4 is predicted to be even.
Digit 5 is predicted to be odd.
Digit 6 is predicted to be even.
Digit 7 is predicted to be odd.
Digit 8 is predicted to be even.
Digit 9 is predicted to be odd.
```

```python
import numpy as np
import matplotlib.pyplot as plt

# Sample 2D dataset (linearly separable)
X = np.array([
    [2, 1], [1, 3], [2, 3],  # Class 0
    [6, 5], [7, 7], [8, 6]   # Class 1
])
y = np.array([0, 0, 0, 1, 1, 1])  # Labels

# Initialize weights and bias
w = np.zeros(2)
b = 0
lr = 0.1

# Perceptron training
for _ in range(10):  # epochs
    for xi, yi in zip(X, y):
        z = np.dot(w, xi) + b
        y_pred = int(z >= 0)
        error = yi - y_pred
```

```
        w += lr * error * xi
        b += lr * error

# Plotting decision regions
x_min, x_max = 0, 10
y_min, y_max = 0, 10

xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
                     np.linspace(y_min, y_max, 200))
Z = (w[0] * xx + w[1] * yy + b >= 0).astype(int)

# Plot decision region
plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm)

# Plot original points
for class_value in [0, 1]:
    plt.scatter(
        X[y == class_value][:, 0],
        X[y == class_value][:, 1],
        label=f'Class {class_value}'
    )

plt.title("Perceptron Decision Regions")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.grid(True)
plt.show()
```




```python
import numpy as np

# Two pattern pairs
A = [[1, -1, 1], [-1, 1, -1]]
B = [[1, -1], [-1, 1]]

# Compute weight matrix
W = sum(np.outer(a, b) for a, b in zip(A, B))

# Sign function
def sign(x): return [1 if i >= 0 else -1 for i in x]

# Recall B from A
print("Recall B from A:")
for a in A:
    print(f"A: {a} -> B: {sign(np.dot(a, W))}")

# Recall A from B
print("\nRecall A from B:")
for b in B:
    print(f"B: {b} -> A: {sign(np.dot(b, W.T))}")
```

```
Recall B from A:
A: [1, -1, 1] -> B: [1, -1]
A: [-1, 1, -1] -> B: [-1, 1]

Recall A from B:
B: [1, -1] -> A: [1, -1, 1]
B: [-1, 1] -> A: [-1, 1, -1]
```

```python
import numpy as np

def sigmoid(x): return 1 / (1 + np.exp(-x))
def deriv(x): return x * (1 - x)

# XOR inputs and outputs
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])

# Initialize weights and biases
w1 = np.random.rand(2, 2)
w2 = np.random.rand(2, 1)
b1 = np.random.rand(1, 2)
b2 = np.random.rand(1, 1)

# Train
for _ in range(10000):
    h = sigmoid(X @ w1 + b1)
    o = sigmoid(h @ w2 + b2)
    e = y - o
    d = e * deriv(o)
    w2 += h.T @ d
    b2 += d.sum(axis=0, keepdims=True)
    dh = d @ w2.T * deriv(h)
    w1 += X.T @ dh
    b1 += dh.sum(axis=0, keepdims=True)

# Output
print("Output after training:")
print(np.round(o, 3))
```

```
Output after training:
[[0.013]
 [0.989]
 [0.989]
 [0.012]]
```

```python
##XOR Backpropogation
import numpy as np

# Activation and derivative
def sigmoid(x): return 1 / (1 + np.exp(-x))
def deriv(x): return x * (1 - x)
```

```python
# XOR inputs and outputs (binary)
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])

# Random weights and biases
w1 = np.random.rand(2, 2)
w2 = np.random.rand(2, 1)
b1 = np.random.rand(1, 2)
b2 = np.random.rand(1, 1)

# Training
for _ in range(10000):
    h = sigmoid(X @ w1 + b1)
    o = sigmoid(h @ w2 + b2)
    d = (y - o) * deriv(o)
    w2 += h.T @ d
    b2 += d.sum(axis=0, keepdims=True)
    dh = d @ w2.T * deriv(h)
    w1 += X.T @ dh
    b1 += dh.sum(axis=0, keepdims=True)

# Result
print("XOR Output:")
print(np.round(o))
```

```
XOR Output:
[[0.]
 [1.]
 [1.]
 [0.]]
```

```python
##HOP FIELD NETWORK
import numpy as np

class Hopfield:
    def __init__(self, size):
        self.weights = np.zeros((size, size))

    def train(self, patterns):
        for p in patterns:
            self.weights += np.outer(p, p)
        np.fill_diagonal(self.weights, 0)

    def recall(self, pattern, steps=5):
        for _ in range(steps):
            pattern = np.sign(np.dot(self.weights, pattern))
            pattern[pattern == 0] = 1  # Convert 0s to 1s
        return pattern
```

```python
# Define binary patterns
patterns = [
    [1, -1, 1, -1],
    [-1, 1, -1, 1],
    [1, 1, -1, -1],
    [-1, -1, 1, 1]
]

# Create and train Hopfield network
hopfield = Hopfield(4)
hopfield.train(patterns)

# Test with a noisy pattern
input_pattern = [1, -1, 1, 1]
output_pattern = hopfield.recall(np.array(input_pattern))

print("Input Pattern:", input_pattern)
print("Recalled Pattern:", output_pattern)
```

```
Input Pattern: [1, -1, 1, 1]
Recalled Pattern: [-1. -1.  1. -1.]
```

```python
##MNIST with PyTorch (Simplified)

import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Load and preprocess dataset
transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])
train_loader = DataLoader(datasets.MNIST('.', train=True,
download=True, transform=transform), batch_size=64, shuffle=True)
test_loader = DataLoader(datasets.MNIST('.', train=False,
download=True, transform=transform), batch_size=64)

# Simple model
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```

```
# Train and evaluate
model = Net()
optimizer = optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss()

for epoch in range(5):
    for data, target in train_loader:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

# Test accuracy
correct, total = 0, 0
with torch.no_grad():
    for data, target in test_loader:
        output = model(data)
        _, predicted = torch.max(output, 1)
        correct += (predicted == target).sum().item()
        total += target.size(0)
print(f"Accuracy: {100 * correct / total}%")
```

```
Accuracy: 97.07%
```

```
##MNIST with Keras (TensorFlow Backend)
import tensorflow as tf
from tensorflow.keras.datasets import mnist

# Load and preprocess data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train.reshape(-1, 28*28)
x_test = x_test.reshape(-1, 28*28)

# Model
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28*28,)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Compile and train
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)

# Evaluate
```

```python
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc}")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

/usr/local/lib/python3.11/dist-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)

Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - accuracy: 0.8568 - loss: 0.4800
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 17s 4ms/step - accuracy: 0.9534 - loss: 0.1537
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9677 - loss: 0.1063
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9734 - loss: 0.0859
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.9772 - loss: 0.0728
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9743 - loss: 0.0853
Test accuracy: 0.9779999852180481
```

```python
##MNIST with TensorFlow (Low-Level)

import tensorflow as tf
from tensorflow.keras.datasets import mnist

# Load and preprocess data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train.reshape(-1, 28*28)
x_test = x_test.reshape(-1, 28*28)

# Model
inputs = tf.keras.Input(shape=(28*28,))
x = tf.keras.layers.Dense(128, activation='relu')(inputs)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = tf.keras.layers.Dense(10, activation='softmax')(x)
model = tf.keras.Model(inputs=inputs, outputs=outputs)

# Compile and train
```

```python
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5)

# Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_acc}")
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8607 - loss:
0.4842
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9563 - loss:
0.1481
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9668 - loss:
0.1087
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 3ms/step - accuracy: 0.9736 - loss:
0.0866
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9765 - loss:
0.0754
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9719 - loss:
0.0887
Test accuracy: 0.9761000275611877
```