Lab - 1 - Warm-Up: Vectors and Dot Product for Perceptron

1. Aim : The aim of this lab warm-up is to introduce the minimum mathematical and computational foundations required to understand and implement a Perceptron. This warm-up focuses on vectors, dot product, and linear combinations, which form the core computation in all neural networks and deep learning models.

Exercise 1: Scalars and Vectors

Objective: Understand how scalars and vectors are represented in NumPy.

• Define a scalar bias value

• Define an input vector and a weight vector

• Display their values

Observation:

• Identify which variables represent learnable parameters

In [1]:
```
import numpy as np

# Define a scalar bias value
b = 0.5

# Define an input vector and weight vector
x = np.array([1.0, 2.0, 3.0,-5.0, 6.4, 2, 34,199,29,504, 20.1,33.3])

# Define a weight vector (learnable parameter)
w = np.array([0.2, 0.5, 0.3, 0.1, -0.4, 0.6, 0.7, -0.2, 0.3, 0.8, -0.5, 0.9])

# Display their values
print("Bias:", b)
print("Input vector:", x)
print("Weight vector :", w)

# Observation: Learnable parameters are w and b, they can be changed during training.
# Input vector x comes from the data and is not learned
```

```
Bias: 0.5
Input vector: [  1.    2.    3.   -5.    6.4   2.   34.  199.   29.  504.   20.1  33.
3]
Weight vector : [ 0.2  0.5  0.3  0.1 -0.4  0.6  0.7 -0.2  0.3  0.8 -0.5  0.9]
```

Exercise 2: Dot Product

Objective: Compute the dot product between input and weight vectors.

• Use NumPy to compute the dot product

• Print the output

Interpretation:

• A positive value indicates the input lies on one side of the decision boundary

• A negative value indicates the opposite side

In [2]:
```python
# DOt product-
dot_product = np.dot(w, x)
print("Dot product (w · x):", dot_product)

# Interpretation
if dot_product > 0:
    print("The input lies on one side of the decision boundary.")
else:
    print("The input lies on the opposite side of the decision boundary.")
```

```
Dot product (w · x): 416.06000000000006
The input lies on one side of the decision boundary.
```

Exercise 3: Linear Combination

Objective: Compute the linear output of a neuron.

• Add bias to the dot product

• Observe how bias shifts the decision boundary

In [3]:
```python
def perceptron_output(w, x, b):
    dot_product = np.dot(w, x)
    z = dot_product + b
    return z

w1 = np.array([0.2, 0.5, 0.3, 0.1, -0.4, 0.6, 0.7, -0.2, 0.3, 0.8, -0.5, 0.9])
b1 = 0.5
x1 = np.array([1.0, 2.0, 3.0,-5.0, 6.4, 2, 34,199,29,504, 20.1,33.3])

z1 = perceptron_output(w1, x1, b1)
print(f"Original: z = {z1}")

# Checking for different biases
biases_demo = [0.0, 1.0, -1.0, 2.0]
for bias_val in biases_demo:
```

```
    z_demo = perceptron_output(w1, x1, bias_val)
    print(f"Bias {bias_val}: z = {z_demo}")
```

```
Original: z = 416.56000000000006
Bias 0.0: z = 416.06000000000006
Bias 1.0: z = 417.06000000000006
Bias -1.0: z = 415.06000000000006
Bias 2.0: z = 418.06000000000006
```

Exercise 4: Step Activation Function

Objective: Convert linear output into a binary decision.

• Implement a simple step activation function

• Apply it to the linear output

In [4]:
```
def step_activation(z):
    return 1 if z >= 0 else 0


x_new1 = np.array([0.5, -1.0, 2.0, 1.5, -0.5, 3.0])
w_new1 = np.array([0.3, -0.2, 0.4, 0.1, -0.6, 0.7])
b_new1 = 0.2


z1 = perceptron_output(w_new1, x_new1, b_new1)
output1 = step_activation(z1)
print(f"z = {z1}, Step output = {output1}")


biases_test = [0.0, 0.5, -0.5, 1.0]

for bias in biases_test:
    z_test = perceptron_output(w_new1, x_new1, bias)
    step_out = step_activation(z_test)
    print(f"Bias {bias}: z = {z_test}, Step output = {step_out}")
```

```
z = 3.8999999999999995, Step output = 1
Bias 0.0: z = 3.6999999999999993, Step output = 1
Bias 0.5: z = 4.199999999999999, Step output = 1
Bias -0.5: z = 3.1999999999999993, Step output = 1
Bias 1.0: z = 4.699999999999999, Step output = 1
```

Discussion Point: Why does a perceptron use an activation function?

Activation functions introduce non-linearity which helps the perceptron learn complex patterns and not just the linear ones

Exercise 5: Matrix as Batch of Samples

Objective: Understand batch processing.

• Define a matrix where each row is a data sample

• Print the matrix

In [5]:
```
# Define a matrix where each row represents a data sample
X_batch = np.array([
    [1.0, 2.0, 3.0],
    [0.5, -1.0, 2.0],
    [-0.5, 1.5, -2.0],
    [2.0, 0.0, 1.0]
])

print("Batch matrix X (each row is a sample):")
print(X_batch)
print(f"Shape: {X_batch.shape} (rows: samples, columns: features)")
```

```
Batch matrix X (each row is a sample):
[[ 1.   2.   3. ]
 [ 0.5 -1.   2. ]
 [-0.5  1.5 -2. ]
 [ 2.   0.   1. ]]
Shape: (4, 3) (rows: samples, columns: features)
```

Exercise 6: Batch Dot Product

Objective: Apply dot product to multiple samples simultaneously.

• Compute dot product between matrix and weight vector

• Add bias

In [6]:
```
w_batch = np.array([0.4, -0.2, 0.6])

dot_products_batch = X_batch @ w_batch
print("Dot products for batch:", dot_products_batch)

b_batch = 0.1
z_batch = dot_products_batch + b_batch
print("Linear outputs z for batch (after adding bias):", z_batch)

outputs_batch = np.array([step_activation(z) for z in z_batch])
print("Step activation outputs for batch:", outputs_batch)
```

```
Dot products for batch: [ 1.8  1.6 -1.7  1.4]
Linear outputs z for batch (after adding bias): [ 1.9  1.7 -1.6  1.5]
Step activation outputs for batch: [1 1 0 1]
```

7. Post-Lab Reflection Questions Answer the following questions in your lab record:

1 . What role do weights play in the dot product?

- Weights scale each input feature before summation; the dot product computes the weighted sum that determines the neuron's activation. Learning adjusts weights to change how features influence the output and thus reshape the decision boundary.

2. How does bias affect the decision boundary?

- Bias shifts the activation value (z) up or down, effectively translating the decision boundary without changing the weights. Adjusting bias moves the threshold at which the neuron fires.

3. Why is dot product fundamental to neural networks?

- The dot product is the core linear operation that combines inputs and weights into a single activation value. It's efficient, vectorizable, and composes naturally into matrix operations for layers and batches, forming the backbone of forward and backward passes.

4. How does batch computation help in training deep learning models?

- Batch computation processes multiple samples at once using matrix operations, improving computational efficiency (via vectorization and hardware acceleration), providing more stable gradient estimates, and increasing throughput during training.