Classification using the Hebb learning rule

1. Importing necessary libraries

We begin by importing the `numpy` library required for the numerical operations involved in our analysis.

```
import numpy as np
```

2. Defining the Hebbian network class

Next, we define a class implementing a neural network based on the Hebb learning rule.

- It initialises the network with a weight matrix set to zero, where the size corresponds to the number of input features.
- The `hebbian_learning` method applies the Hebb learning rule by adjusting the weights according to the outer product of the input patterns and their corresponding target patterns, incrementally updating the weight matrix.
- The `activate` method computes the weighted sum of inputs and applies a threshold function to produce binary outputs. Here, it uses a bipolar representation, where the output is 1 for non-negative sums and -1 for negative sums.

```
class Hebbian_Network:
  def __init__(self, input_size):
    self.weights = np.zeros((input_size, input_size))

  def hebbian_learning(self, inputs, targets):
    for input_pattern, target_pattern in zip(inputs, targets):
      self.weights += np.outer(input_pattern, target_pattern)

  def activate(self, input_pattern):
    net_input = np.dot(self.weights, input_pattern)
    return np.where(net_input >= 0, 1, -1)
```

3. Defining the training data

Now, we generate the training data for two letters (`A` and `B`) represented as 3x3 grids. Each pattern is flattened into a 9-dimensional array. Bipolar data is created by converting binary representations to bipolar values (1 and -1). The function returns both bipolar and binary inputs and their corresponding target patterns, which serve as identity mappings for training.

```
def generate_training_data():
  A_bipolar = np.array([[1, 1, -1],
                        [1, 1, -1],
                        [1, 1, 1]]).flatten() * 2 - 1
  B_bipolar = np.array([[1, 1, -1],
                        [1, 1, 1],
                        [1, 1, -1]]).flatten() * 2 - 1
  A_binary = (A_bipolar + 1) // 2
  B_binary = (B_bipolar + 1) // 2

  inputs_bipolar = np.array([A_bipolar, B_bipolar])
  targets_bipolar = np.array([A_bipolar, B_bipolar])

  inputs_binary = np.array([A_binary, B_binary])
  targets_binary = np.array([A_binary, B_binary])

  return (inputs_bipolar, targets_bipolar), (inputs_binary, targets_binary)
```

4. Training and testing the Hebbian network

Finally, we define the main execution block which:

- calls the `generate_training_data` function to create the training data for both bipolar and binary representations,
- initialises the Hebbian network with an input size of 9 (for the flattened 3x3 grid),
- trains the network using the bipolar training data and then tests it by activating the network with the training inputs, printing the results, and,
- resets the weights and trains the network again using the binary data, followed by testing and printing the outputs for the binary input patterns.

```
if __name__ == "__main__":
  (inputs_bipolar, targets_bipolar), (inputs_binary, targets_binary) = generate_training_data()
  hebbian_network = Hebbian_Network(input_size=9)
  hebbian_network.hebbian_learning(inputs_bipolar, targets_bipolar)

  print("Testing with bipolar data:")
```

```python
for input_pattern in inputs_bipolar:
  output = hebbian_network.activate(input_pattern)
  print(f'Input pattern: {input_pattern.reshape(3,3)}\nOutput: {output.reshape(3,3)}\n')

hebbian_network.weights = np.zeros((9,9))
hebbian_network.hebbian_learning(inputs_binary, targets_binary)

  # Test the network with binary data
print("Testing with Binary Data:")
for input_pattern in inputs_binary:
    output = hebbian_network.activate(input_pattern)
    print(f"Input Pattern: {input_pattern.reshape(3, 3)}\nOutput: {output.reshape(3, 3)}\n")
```

```
Testing with bipolar data:
Input pattern: [[ 1  1 -3]
 [ 1  1 -3]
 [ 1  1  1]]
Output: [[ 1  1 -1]
 [ 1  1 -1]
 [ 1  1 -1]]

Input pattern: [[ 1  1 -3]
 [ 1  1  1]
 [ 1  1 -3]]
Output: [[ 1  1 -1]
 [ 1  1 -1]
 [ 1  1 -1]]

Testing with Binary Data:
Input Pattern: [[ 1  1 -1]
 [ 1  1 -1]
 [ 1  1  1]]
Output: [[ 1  1 -1]
 [ 1  1 -1]
 [ 1  1  1]]

Input Pattern: [[ 1  1 -1]
 [ 1  1  1]
 [ 1  1 -1]]
Output: [[ 1  1 -1]
 [ 1  1  1]
 [ 1  1 -1]]
```