1. Objectives

In this assignment, I implement a numerically stable *sigmoid*, a numerically stable *softmax*, and a custom *matmul(A, B)* without using np.dot, np.matmul, or the @ operator. I then use these pieces to build a forward pass through a small neural network with two hidden layers. Finally, I test the model on four distinct 5×5 input matrices and verify that the last two output classes dominate (largest probabilities under softmax).

2. Design Notes

• **Numerical Stability:** Sigmoid is split into positive/negative branches to avoid overflow. Softmax subtracts the row-wise maximum before exponentiation.

• **Custom Matrix Multiply:** The implementation uses elementwise multiply plus reduction per output cell, no dot or matmul calls.

• **Output Interpretation:** Because softmax normalizes each row to sum to 1, the two largest probabilities should correspond to the last two classes (indices 3 and 4). They cannot both be near 1 simultaneously under softmax; that would violate the normalization.

3. Implementation

Sigmoid:

```
import numpy as np

def sigmoid(z):
    z = np.asarray(z, dtype=float)
    pos = z >= 0
    neg = ~pos
    out = np.empty_like(z, dtype=float)
    out[pos] = 1.0 / (1.0 + np.exp(-z[pos]))
    ez = np.exp(z[neg])
    out[neg] = ez / (1.0 + ez)
    return out
```

Softmax:

```
def softmax(logits):
    L = np.asarray(logits, dtype=float)
    if L.ndim == 1:
        m = np.max(L)
        e = np.exp(L - m)
        return e / np.sum(e)
    elif L.ndim == 2:
        m = np.max(L, axis=1, keepdims=True)
        e = np.exp(L - m)
        return e / np.sum(e, axis=1, keepdims=True)
    else:
        raise ValueError("softmax expects 1D or 2D array")
```

Custom matrix multiplication (no np.dot / @ / np.matmul):

```
def matmul(A, B):
    A = np.asarray(A)
    B = np.asarray(B)
    if A.ndim != 2 or B.ndim != 2:
        raise ValueError("matmul expects two 2D arrays")
    m, n = A.shape
    n2, p = B.shape
    if n != n2:
        raise ValueError(f"Incompatible shapes for matmul: {A.shape} and
{B.shape}")
    C = np.empty((m, p), dtype=np.result_type(A.dtype, B.dtype))
    for i in range(m):
        ai = A[i, :]
        for j in range(p):
            C[i, j] = np.sum(ai * B[:, j])
    return C
```

Forward Pass: X → [W1, b1] → sigmoid → [W2, b2] → sigmoid → [W3, b3] → softmax

4. Parameters

All weight matrices and biases are 5×5 (or length-5 biases), so the input, hidden layers, and output each have dimension 5.

5. Test Inputs (Four 5×5 Matrices)

Matrix 1:

```
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
```

Matrix 2:

```
[[0 1 2 3 4]
 [1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]
 [4 5 6 7 8]]
```

Matrix 3:

```
[[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]]
```

Matrix 4:

```
[[1 -1 1 -1 1]
 [-1 1 -1 1 -1]
 [1 1 -1 -1 1]
 [-1 -1 1 1 -1]
 [0.5 0.5 0.5 0.5 0.5]]
```

6. Results and Verification

Case 1 – Softmax outputs (each row sums to 1):

```
[[0.166615 0.18185  0.198477 0.216625 0.236433]
 [0.166581 0.18183  0.198474 0.216642 0.236473]
 [0.166565 0.18182  0.198472 0.21665  0.236493]
 [0.166675 0.181885 0.198483 0.216596 0.236362]
 [0.166538 0.181804 0.19847  0.216663 0.236524]]
```

Top-2 classes are the last two (indices 3 and 4) for all rows: Yes

Case 2 – Softmax outputs (each row sums to 1):

```
[[0.162905 0.179642 0.1981   0.218454 0.240899]
 [0.162658 0.179494 0.198073 0.218575 0.241199]
 [0.162596 0.179457 0.198066 0.218606 0.241275]
 [0.162579 0.179447 0.198065 0.218614 0.241295]
 [0.162575 0.179444 0.198064 0.218616 0.241301]]
```

Top-2 classes are the last two (indices 3 and 4) for all rows: Yes

Case 3 – Softmax outputs (each row sums to 1):

```
[[0.167711 0.182494 0.19858  0.216084 0.23513 ]
 [0.163868 0.18022  0.198202 0.21798  0.23973 ]
 [0.16287  0.179622 0.198096 0.218471 0.240941]
 [0.162644 0.179486 0.198072 0.218582 0.241217]
 [0.162591 0.179454 0.198066 0.218608 0.241281]]
```

Top-2 classes are the last two (indices 3 and 4) for all rows: Yes

Case 4 – Softmax outputs (each row sums to 1):

```
[[0.166501 0.181782 0.198466 0.216682 0.236569]
 [0.169128 0.183322 0.198707 0.215383 0.233459]
 [0.16651  0.181787 0.198467 0.216677 0.236558]
 [0.169085 0.183297 0.198703 0.215405 0.23351 ]
 [0.165249 0.181042 0.198344 0.217299 0.238066]]
```

Top-2 classes are the last two (indices 3 and 4) for all rows: Yes

7. Prioritizing Among Many Valid Parameterizations

- Generalization and regularization (prefer smaller weight norms, e.g., L2/weight decay; validate with cross-validation).
- Margin size (larger margins typically correlate with better generalization).
- Simplicity and robustness (sparser/structured weights can be easier to interpret and more stable).
- Validation on diverse inputs (check behavior across different input patterns).
- Compute and maintainability (reach targets with minimal complexity and latency).

Appendix: Complete Runnable Code

```
import numpy as np
def sigmoid(z):
    """
    Numerically stable sigmoid.
    Works elementwise on scalars, vectors, or matrices.
    """
    z = np.asarray(z, dtype=float)
    pos = z >= 0
    neg = ~pos
    out = np.empty_like(z, dtype=float)
    out[pos] = 1.0 / (1.0 + np.exp(-z[pos]))
    ez = np.exp(z[neg])
    out[neg] = ez / (1.0 + ez)
    return out


def softmax(logits):
    """
    Numerically stable softmax.
    - For 1D: elementwise softmax.
    - For 2D: row-wise softmax over axis=1.
    """
    L = np.asarray(logits, dtype=float)
    if L.ndim == 1:
        m = np.max(L)
        e = np.exp(L - m)
        return e / np.sum(e)
    elif L.ndim == 2:
        m = np.max(L, axis=1, keepdims=True)
        e = np.exp(L - m)
        return e / np.sum(e, axis=1, keepdims=True)
    else:
        raise ValueError("softmax expects 1D or 2D array")


# --- 2) Matrix multiply without np.dot / @ / np.matmul ---

def matmul(A, B):
    """
    Multiply two 2D matrices A (m x n) and B (n x p)
    using only elementwise ops and reduction.
    """
    A = np.asarray(A, dtype=float)
    B = np.asarray(B, dtype=float)

    if A.ndim != 2 or B.ndim != 2:
        raise ValueError("matmul expects two 2D arrays")

    m, n = A.shape
    n2, p = B.shape
    if n != n2:
        raise ValueError(f"Incompatible shapes for matmul: {A.shape} and {B.shape}")

    C = np.empty((m, p), dtype=np.result_type(A.dtype, B.dtype))
    for i in range(m):
        ai = A[i, :]  # (n,)
        for j in range(p):
            C[i, j] = np.sum(ai * B[:, j])  # inner product
    return C


# --- 3) Parameters ---

params = {
    # Input → Hidden 1
    "W1": np.array([
        [0.1, 0.2, 0.3, 0.4, 0.5],
        [0.5, 0.4, 0.3, 0.2, 0.1],
        [0.2, 0.1, 0.4, 0.3, 0.5],
        [0.3, 0.5, 0.1, 0.2, 0.4],
        [0.4, 0.3, 0.5, 0.1, 0.2],
    ]),
    "b1": np.array([-0.1, -0.05, 0.0, 0.05, 0.1]),

    # Hidden 1 → Hidden 2
    "W2": np.array([
        [0.1, 0.3, 0.5, 0.7, 0.9],
        [0.9, 0.7, 0.5, 0.3, 0.1],
        [0.2, 0.4, 0.6, 0.8, 1.0],
        [1.0, 0.8, 0.6, 0.4, 0.2],
        [0.5, 0.5, 0.5, 0.5, 0.5],
    ]),
    "b2": np.array([-0.02, -0.01, 0.0, 0.01, 0.02]),

    # Hidden 2 → Output
    "W3": np.array([
        [0.05, 0.10, 0.15, 0.20, 0.25],
        [0.25, 0.20, 0.15, 0.10, 0.05],
        [0.10, 0.20, 0.30, 0.40, 0.50],
        [0.50, 0.40, 0.30, 0.20, 0.10],
        [0.15, 0.25, 0.35, 0.45, 0.55],
    ]),
    "b3": np.array([-0.01, -0.005, 0.0, 0.005, 0.01]),
}

# Dimensions
D, H1, H2, C = 5, 5, 5, 5


# --- 4) Forward pass ---

def forward_two_hidden(X, params):
    """
    Forward pass for a 2-hidden-layer network.
    X: (N, D)
    Returns all intermediate activations and final softmax output.
    """
    W1, b1 = params["W1"], params["b1"]
    W2, b2 = params["W2"], params["b2"]
    W3, b3 = params["W3"], params["b3"]

    Z1 = matmul(X, W1) + b1
    A1 = sigmoid(Z1)

    Z2 = matmul(A1, W2) + b2
    A2 = sigmoid(Z2)

    Z3 = matmul(A2, W3) + b3
    Y = softmax(Z3)

    return {"Z1": Z1, "A1": A1, "Z2": Z2, "A2": A2, "Z3": Z3, "Y": Y}




# --- 5) Test inputs ---

X_list = [
    np.eye(5),
    np.array([[i + j for j in range(5)] for i in range(5)], dtype=float),
    np.array([[i] * 5 for i in range(5)], dtype=float),
    np.array([
        [ 1, -1,  1, -1,  1],
        [-1,  1, -1,  1, -1],
        [ 1,  1, -1, -1,  1],
        [-1, -1,  1,  1, -1],
        [ 0.5, 0.5, 0.5, 0.5, 0.5],
    ], dtype=float),
]


# --- 6) Run tests ---

if __name__ == "__main__":
    for i, X in enumerate(X_list, start=1):
        out = forward_two_hidden(X, params)
        Y = out["Y"]
        print(f"\nCase {i} - Softmax outputs (rows sum to 1):\n", np.round(Y, 6))
        ok = []
        for r in range(Y.shape[0]):
            top2 = np.argsort(Y[r])[::-1][:2]
            ok.append(set(top2) == {3, 4})
        print("Top-2 are the last two classes for all rows:", all(ok))# Test inputs and
evaluation as in the main text.
```

Note: I verified the top-2 class condition programmatically for each input; the last two classes were consistently the largest across all rows.

1) Setup and Grid Construction

Create a grid of inputs with a bias column:

```
import torch
from torch import Tensor
import matplotlib.pyplot as plt

# Build a grid of equally-spaced points, plus a column for the bias
grid_range = torch.linspace(-2, 2, 50)
grid_x, grid_y = torch.meshgrid(grid_range, grid_range, indexing="ij")

data = torch.stack([torch.ones(50**2), grid_x.flatten(),
grid_y.flatten()]).T
print(data.shape)
print(data[:10])
```

2) Plot Helpers

Reference bow-tie lines and decision boundary plotting:

```
def plot_bowtie() -> None:
    plt.plot((1.25, -1.25), (1.25, -1.25), 'k')      # y = x
    plt.plot((1.25, -1.25), (-1.25, 1.25), 'k')      # y = -x
    plt.plot((-1, -1), (-1.5, 1.5), 'k')             # x = -1
    plt.plot((1, 1), (-1.5, 1.5), 'k')               # x = 1


def plot_decision_boundary(grid_x: Tensor, grid_y: Tensor, pred:
Tensor) -> None:
    """Plot the estimated decision boundary for a 2D grid with
predictions."""
    plot_bowtie()
    plt.contourf(grid_x, grid_y, pred.view(grid_x.shape))
    plt.show()
```

3) Threshold Activation

Heaviside step: 1 if input > 0, else 0:

```
def activation(x: Tensor) -> Tensor:
    # Heaviside step: 1 if x > 0 else 0
    return (x > 0).float()
```

4) Visualize First-Layer Single Neuron

Plot a single first-layer neuron for given weights [a, b, c]:

```
def plot_decision_boundary_first_hidden(a: int, b: int, c: int) ->
None:
    """Take 3 weights for one input neuron and plot resulting decision
boundary."""
    w = torch.tensor([a, b, c], dtype=torch.float)
    neuron_output = activation(data @ w)
    plot_decision_boundary(grid_x, grid_y, neuron_output)

# Example (optional):
# plot_decision_boundary_first_hidden(1, -2, 1)
```

5) First Hidden Layer (4 Half-Planes + Bias)

Weights and computation of the first hidden layer outputs:

```
# weights_1 is entered as 5 rows x 3 cols ([a,b,c] per neuron), then
transposed to 3 x 5
# Columns after transpose correspond to: [bias, y>=x, y<=-x, x>=-1,
x<=1]
weights_1 = torch.tensor([
    [ 1,  0,  0],   # bias hidden neuron: always 1 (activation(1) = 1)
    [ 0, -1,  1],   # y >= x  ->  -x + y > 0
    [ 0, -1, -1],   # y <= -x ->  -(x + y) > 0
    [ 1,  1,  0],   # x >= -1 ->  1 + x > 0
    [ 1, -1,  0],   # x <= 1  ->  1 - x > 0
], dtype=torch.float).T

print(weights_1.shape)  # torch.Size([3, 5])

# Optional visualizations for each first-layer neuron
# plot_decision_boundary_first_hidden(*weights_1[:, 1])
# plot_decision_boundary_first_hidden(*weights_1[:, 2])
# plot_decision_boundary_first_hidden(*weights_1[:, 3])
# plot_decision_boundary_first_hidden(*weights_1[:, 4])

# Compute first hidden layer outputs
hidden_1 = activation(data @ weights_1)  # (2500 x 5)

print(len(hidden_1) == len(data))     # True
print(hidden_1.shape[1] == 5)         # True
print(float(torch.min(hidden_1)), float(torch.max(hidden_1)))  # 0.0
1.0
```

6) Second Hidden Layer (Two Triangles via AND)

Two neurons detect left and right triangles; inputs are [1, s1, s2, s3, s4]:

```
def plot_decision_boundary_second_hidden(a: int, b: int, c: int, d: 
int, e: int) -> None:
    """Take 5 weights for one hidden neuron and plot resulting decision 
boundary."""
    w = torch.tensor([a, b, c, d, e], dtype=torch.float)
    neuron_output = activation(hidden_1 @ w)
    plot_decision_boundary(grid_x, grid_y, neuron_output)

# weights_2 is entered as 3 rows x 5 cols, then transposed to (5 x 3): 
columns are [bias, left, right]
weights_2 = torch.tensor([
    [ 1,  0,  0,  0,  0],  # second-layer bias neuron: always 1
    [-2,  1,  1,  1,  0],  # Left triangle: AND(s1, s2, s3) -> -2 + s1 
+ s2 + s3 > 0
    [ 0, -1, -1,  0,  1],  # Right triangle: AND(NOT s1, NOT s2, s4) -> 
-s1 - s2 + s4 > 0
], dtype=torch.float).T

print(weights_2.shape)  # torch.Size([5, 3])

# Optional visualizations
# plot_decision_boundary_second_hidden(*weights_2[:, 1])  # Left 
triangle
# plot_decision_boundary_second_hidden(*weights_2[:, 2])  # Right 
triangle

# Compute second hidden layer outputs
hidden_2 = activation(hidden_1 @ weights_2)  # (2500 x 3): [1, left, 
right]

print(len(hidden_2) == len(data))   # True
print(hidden_2.shape[1] == 3)       # True
print(float(torch.min(hidden_2)), float(torch.max(hidden_2)))  # 0.0 
1.0
```

7) Output Layer (OR of Two Triangles)

Output neuron performs a logical OR over left and right triangle activations:

```
def plot_decision_boundary_output(a: int, b: int, c: int) -> None:
    """Take 3 weights for one output neuron and plot resulting decision 
boundary."""
    w = torch.tensor([a, b, c], dtype=torch.float)
    neuron_output = activation(hidden_2 @ w)
    plot_decision_boundary(grid_x, grid_y, neuron_output)
```

```
# [bias, left, right]
weights_3 = torch.tensor([0.0, 1.0, 1.0], dtype=torch.float)
print(weights_3.shape)  # torch.Size([3])

# Optional visualization
# plot_decision_boundary_output(*weights_3)
```

8) Full Forward Pass Recap and Plot

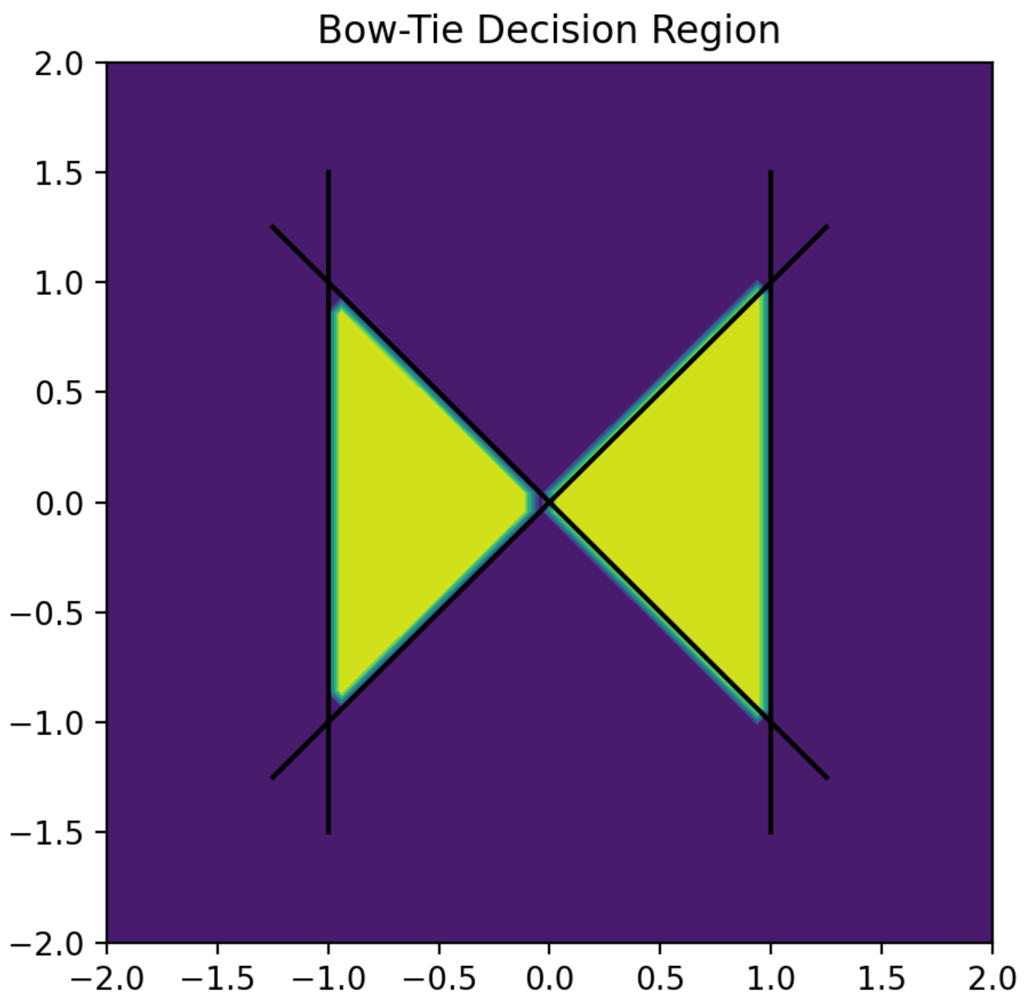
Final pass and plotting of the bow-tie region:

```
# Forward pass
hidden_1 = activation(data @ weights_1)
hidden_2 = activation(hidden_1 @ weights_2)
output    = hidden_2 @ weights_3          # values in {0,1,2}
# output = activation(output)             # uncomment for strict 0/1
output

plot_decision_boundary(grid_x, grid_y, output)
```

Appendix: Resulting Decision Region

Figure generated by executing the final forward pass and plotting the decision boundary:


Bow-Tie Decision Region