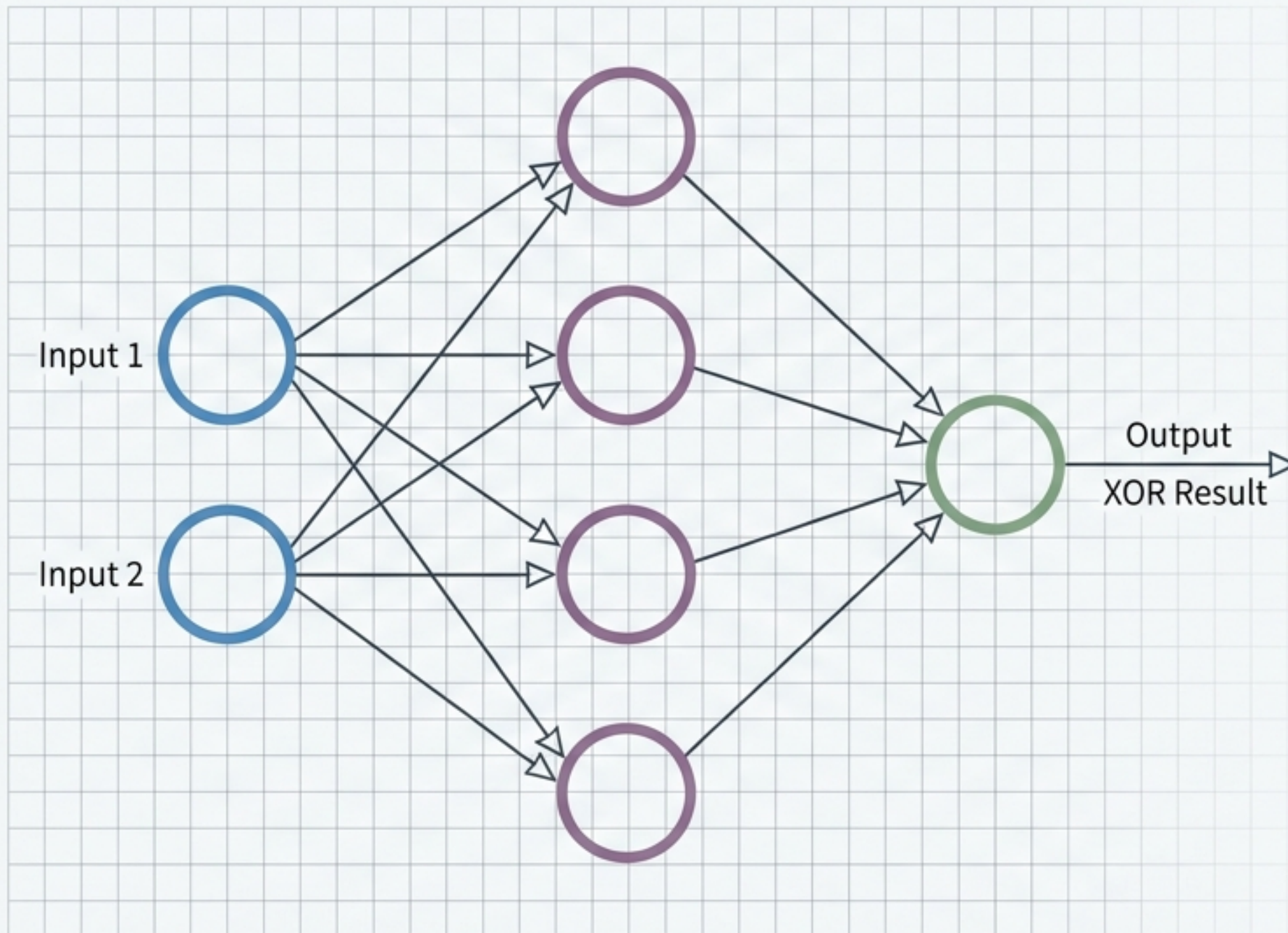


Pythonで解き明かす3層パーセプトロン

非線形問題（XOR演算）を解決するアルゴリズムの完全コード解説



```
import numpy as np

# Three-Layer Perceptron for XOR Problem

class ThreeLayerPerceptron:
    def __init__(self, input_size=2, hidden_size=4, output_size=1):
        # Initialize weights and biases
        self.W1 = np.random.randn(input_size, hidden_size)
        self.b1 = np.zeros(hidden_size)
        self.W2 = np.random.randn(hidden_size, output_size)
        self.b2 = np.zeros(output_size)

    def sigmoid(self, x):
        # Activation function
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def forward(self, X):
        # Forward pass
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1) # Hidden layer activation
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.sigmoid(self.z2) # Output layer activation
        return self.a2

    def backward(self, X, y, output, learning_rate=0.1):
        # Backpropagation
        self.output_error = y - output
        self.output_delta = self.output_error * self.sigmoid_derivative(output)

        self.hidden_error = np.dot(self.output_delta, self.W2.T)
        self.hidden_delta = self.hidden_error * self.sigmoid_derivative(self.a1)

        # Update weights and biases
        self.W2 += np.dot(self.a1.T, self.output_delta) * learning_rate
        self.b2 += np.sum(self.output_delta, axis=0) * learning_rate
        self.W1 += np.dot(X.T, self.hidden_delta) * learning_rate
        self.b1 += np.sum(self.hidden_delta, axis=0) * learning_rate

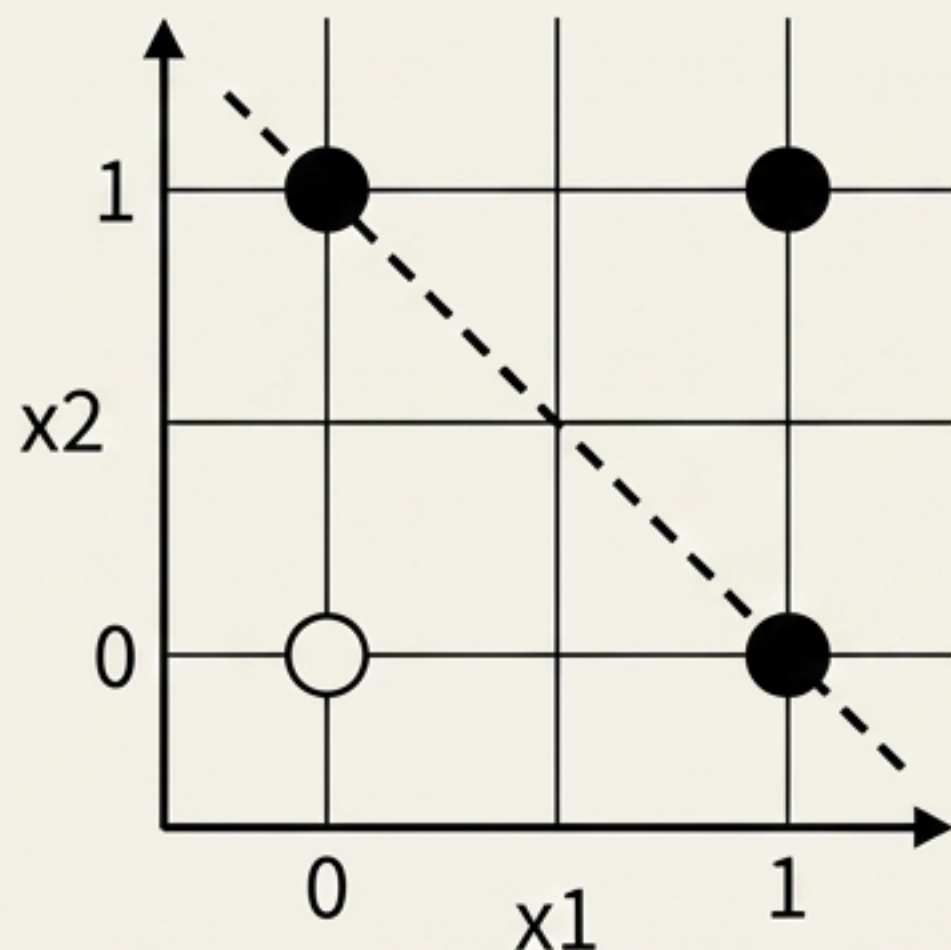
# Training the Perceptron for XOR
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

model = ThreeLayerPerceptron()

# Training loop omitted for brevity...
```

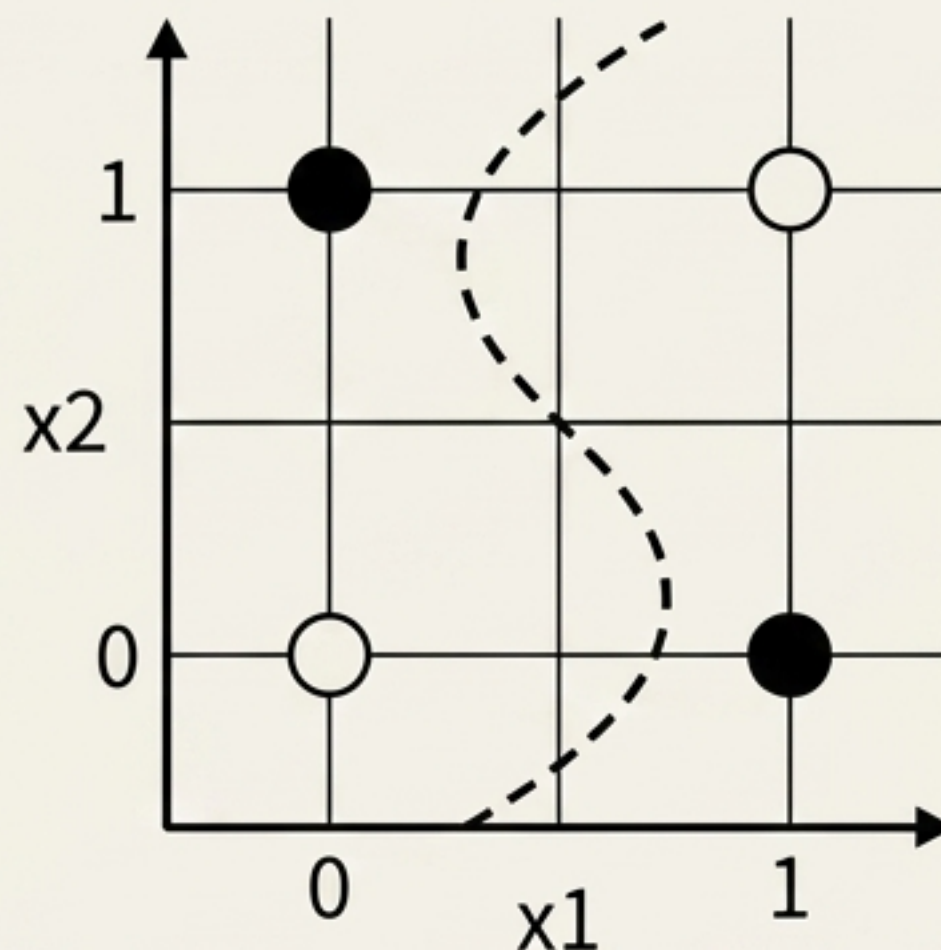
単純パーセプトロンの限界：非線形問題の壁

線形問題（OR演算など）



1本の直線で分類可能（単純パーセプトロンで解決できる）。

非線形問題（XOR演算）

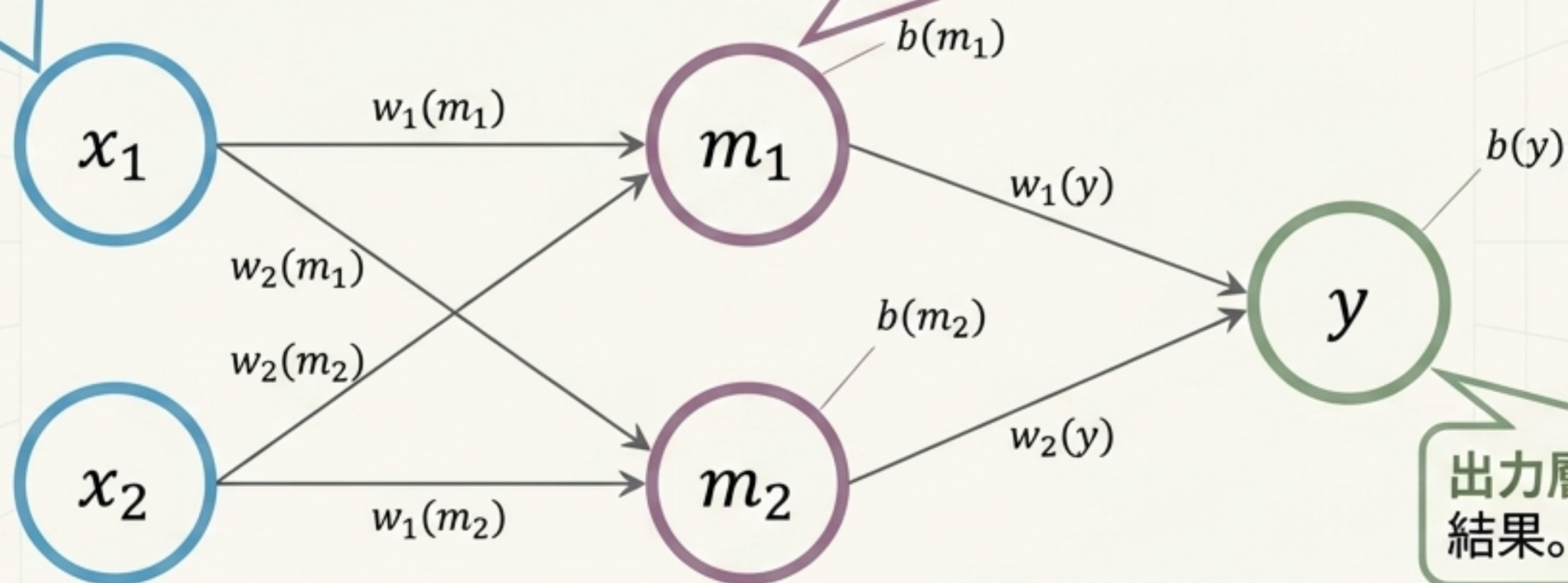


1本の直線では絶対に分類できない。層を増やす必要がある。

解決の青写真：3層パーセプトロンのアーキテクチャ

入力層: 外部からの論理演
の入力値 (0または1)。

ここが鍵！入力を非線形空間
に変換する新しいノード。



出力層: 最終的な演算
結果。

中間ノード (m_1, m_2) を追加することで、複数の直線境界を組み合わせ、
XORのような複雑な境界線を引くことが可能になる。

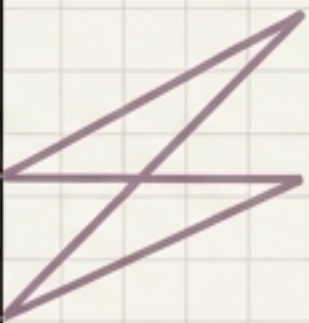
理論値からPython変数へのマッピング

理論値（テーブル）

中間層（Intermediate Layer）

ノード	w1	w2	b
m1	0.5	0.5	-0.2
m2	-0.5	-0.5	0.7

Pythonコード（変数の定義）



```
WM = [[0.5, 0.5],  
      [-0.5, -0.5]] # 中間ノードの重み  
BM = [-0.2, 0.7]    # 中間ノードのバイアス
```

出力層（Output Layer）

ノード	w1	w2	b
y	0.5	0.5	-0.6
			-0.6



```
WY = [0.5, 0.5] # 出力ノードyの重み  
BY = -0.6       # 出力ノードyのバイアス
```

アルゴリズムの全貌：3つの計算ブロック

【全体のループ】
全ての入力パターン
(4通り) を順番に
処理する。

```
def three_layer_perceptron(X, Y, WM, BM, WY, BY, NC, NM, NI):  
    M = [[0] * NM for _ in range(NC)] # 中間ノードの出力  
    for out in range(NC):  
        ytemp = BY  
        for mid in range(NM):  
            mtemp = BM[mid]  
            for jin in range(NI):  
                mtemp += WM[mid][jin] * X[out][jin]  
            if mtemp > 0:  
                M[out][mid] = 1  
            else:  
                M[out][mid] = 0  
        ytemp += WY[mid] * M[out][mid]  
        if ytemp > 0:  
            Y[out] = 1  
        else:  
            Y[out] = 0
```

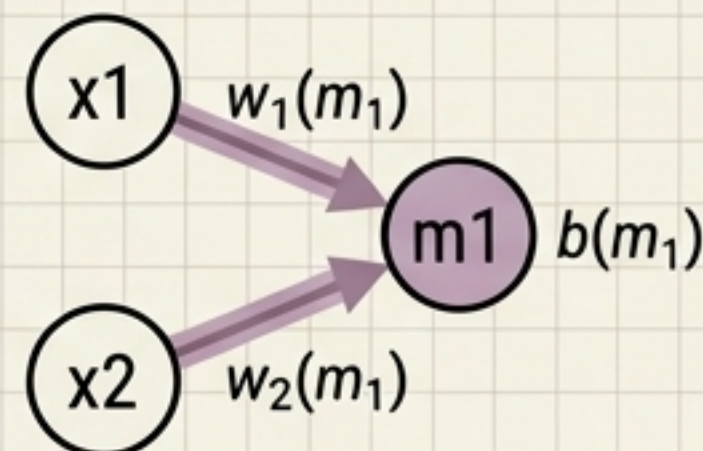
【Step 1】
入力値(X)と重み(WM)・
バイアス(BM)から中間ノード
(m1, m2)の値を計算。

【Step 2】
中間ノードの結果(M)と
重み(WY)・バイアス(BY)から
最終出力(Y)を決定。

中間ノード演算：数式とコードの完全な一致

MATH
EQUATION

$$m1 = x_1 \times w_1(m_1) + x_2 \times w_2(m_1) + b(m_1)$$



(入力値の配列)

(中間ノードへの重み配列)

(中間ノードのバイアス)

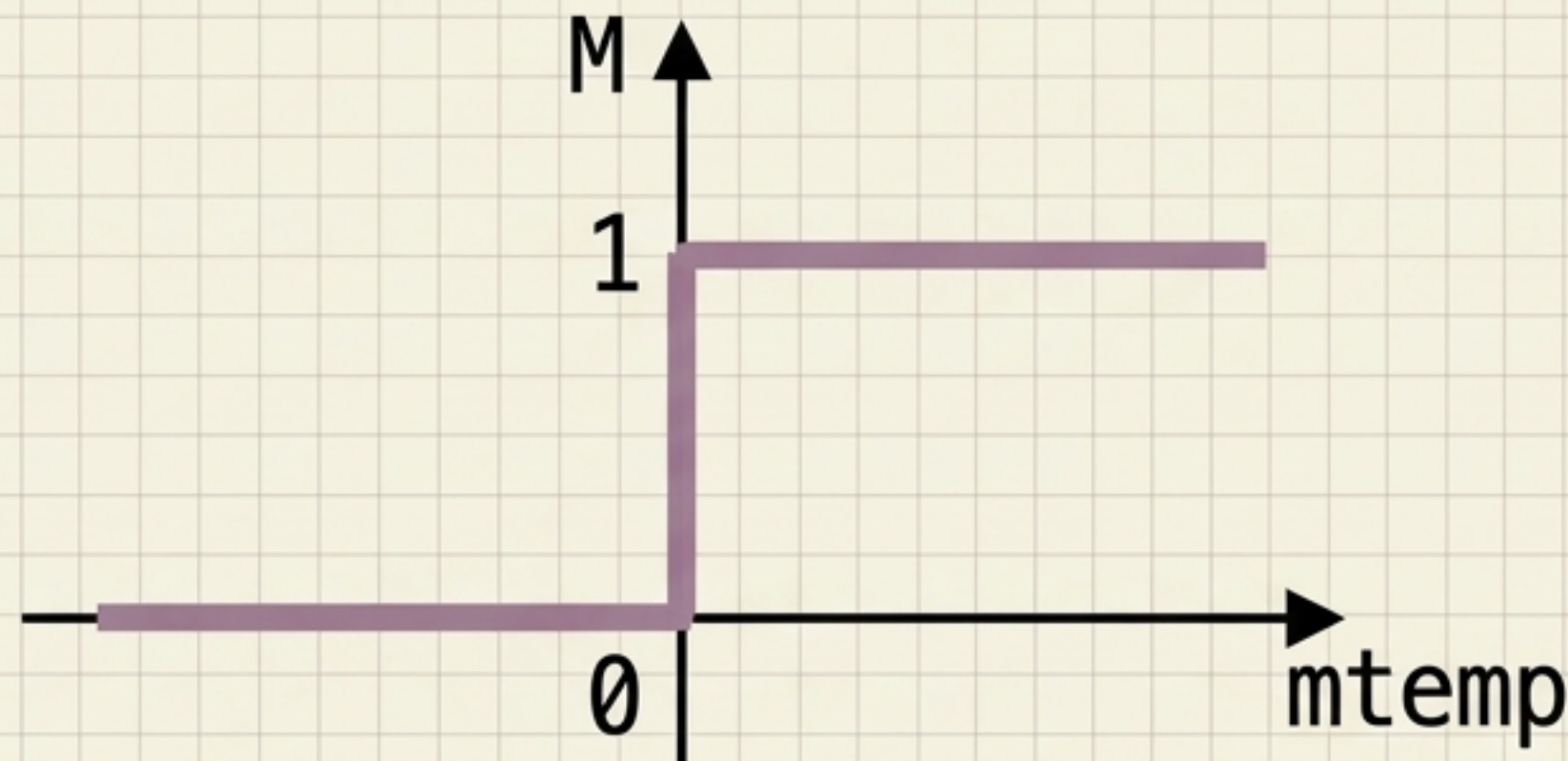
PYTHON
CODE

```
mtemp = BM[mid]
for jin in range(NI):
    mtemp += WM[mid][jin] * X[out][jin]
```

Pythonの二重ループ (midとjin) は、数学的な「入力と重みの内積＋バイアス」をプログラム上で忠実に再現している。

活性化関数：計算値を「0」か「1」に変換する

```
if (mtemp > 0):  
    M[out][mid] = 1  
else:  
    M[out][mid] = 0
```



配列 M[out][mid] の役割

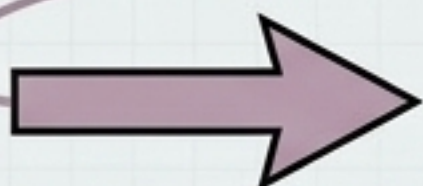
4通りの入力パターン(out)それぞれに対して、2つの中間ノード(mid)の発火状態(0か1)を一時保存する2次元配列。これが次の出力層への「新たな入力」となる。

実行トレース(前編): 入力 [1, 0] を中間層に通す

条件: 入力値 $X = [1, 0]$ (つまり $x_1=1, x_2=0$)

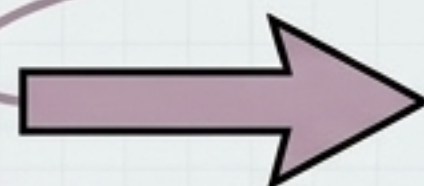
1. m1 の計算

$$\underset{x_1}{1} \times \underset{w_1}{0.5} + \underset{x_2}{0} \times \underset{w_2}{0.5} - \underset{b}{0.2} = 0.3$$

 $0.3 > 0$ なので、
 $m_1 = 1$ (発火)

2. m2 の計算

$$\underset{x_1}{1} \times \underset{w_1}{(-0.5)} + \underset{x_2}{0} \times \underset{w_2}{(-0.5)} + \underset{b}{0.7} = 0.2$$

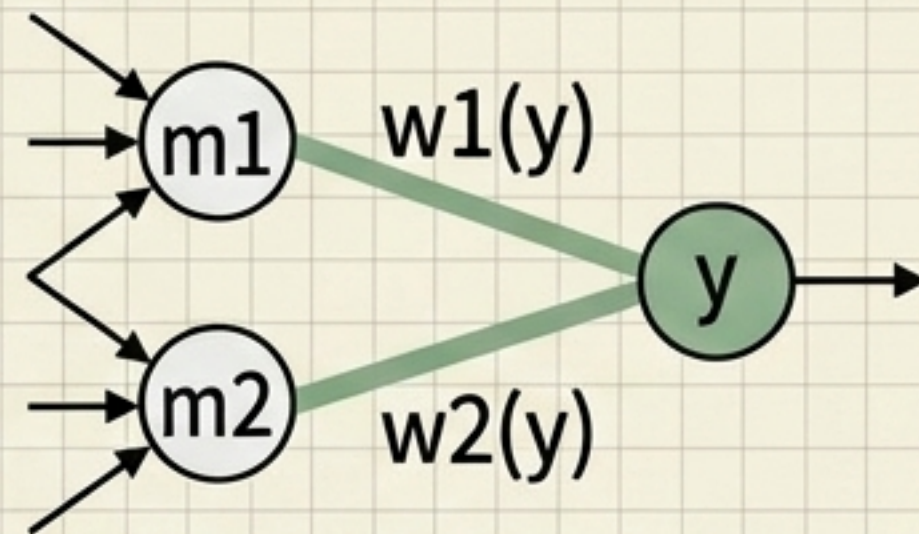
 $0.2 > 0$ なので、
 $m_2 = 1$ (発火)

配列 M のこのパターンの結果は $[1, 1]$ となる。元の入力 $[1, 0]$ が、中間層によって $[1, 1]$ という新しい特徴空間に変換された！

出力層演算：変換されたデータから最終回答を導く

Math

$$y = m1 \times w1(y) + m2 \times w2(y) + b(y)$$



Code

```
ytemp = BY
for mid in range(NM):
    ytemp = ytemp + M[out][mid] * WY[mid]
```

(先ほど計算した中間ノードの配列)

(出力ノードへの重み)

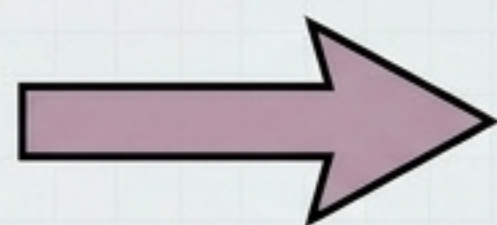
Key Insight: 中間結果を格納した M が、今度は出力層にとっての「入力値」として機能している。

実行トレース(後編): 最終出力 y を確定する

条件: 中間層の出力 $M = [1, 1]$ (つまり $m1=1, m2=1$)

3. y の計算

$$\underset{\substack{\uparrow \\ m1}}{1} \times \underset{\substack{\uparrow \\ w1(y)}}{0.5} + \underset{\substack{\uparrow \\ m2}}{1} \times \underset{\substack{\uparrow \\ m2}}{0.5} - \underset{\substack{\uparrow \\ w2(y)}}{0.6} = \underset{\substack{\uparrow \\ b(y)}}{0.4}$$



**$0.4 > 0$ なので、
 $y = 1$ (発火)**

出力 $y = 1$

入力 $[1, 0]$ に対して出力が 1 となった。これは見事にXOR (排他的論理和) の正しい結果と一致している！

完璧な非線形分離：プログラムの実行結果

--- 3層パーセプトロン (XOR演算) の結果 ---

入力: [0, 0] → 出力: 0

入力: [0, 1] → 出力: 1

入力: [1, 0] → 出力: 1

入力: [1, 1] → 出力: 0

同じ入力 → 0

異なる入力 → 1

3層アーキテクチャによって非線形問題が完全にコードで解決された瞬間。

XOR演算
成功

結論：なぜ層を増やすと「解けない問題」が解けるのか

	単純パーセプトロン (1層)	3層パーセプトロン
対応可能な問題	線形問題 (AND, OR, NAND)	非線形問題 (XOR, 同値(EQ))
コード構造 (Python)	1段階のループ処理 (ytempの計算のみ)	中間配列 M を介した2段階のループ処理 (mtemp → ytemp)
空間の捉え方		

Pythonコードの二重ループと中間配列 M は、単なる計算の繰り返しではない。入力データを「線形分離可能な新しい次元」へと変換するための、数学的に裏付けられた空間操作である。