```
import numpy as np

def mp_neuron(inputs, weights, threshold):
   weighted_sum = np.dot(inputs, weights)
   output = 1 if weighted_sum == threshold else 0
   return output

def and_not(x1, x2):
   weights = np.array([1, -1])
   threshold = 1
   inputs = np.array([x1, x2])
   output = mp_neuron(inputs, weights, threshold)
   formula = f"({x1} * 1) + ({x2} * 1) = {np.dot(inputs, weights)}"

   return output

print("X1\t X2 \t\tOutput")
for x1 in [0, 1]:
   for x2 in [0, 1]:
      output = and_not(x1, x2)
      print(f"{x1}\t{x2}\t\t{output}")          ..........explain this code line by line
```

Of course! I'll explain your code **line by line** in a very clear way:

```python
import numpy as np
```

- **Imports** the `numpy` library and gives it the alias `np`.
- `numpy` helps with mathematical operations like dot products, arrays, etc.

```python
def mp_neuron(inputs, weights, threshold):
```

- Defines a **function** `mp_neuron` that simulates a simple **McCulloch-Pitts (MP) neuron**.
- It takes three inputs:
  - `inputs`: array of input values.
  - `weights`: array of weights associated with inputs.
  - `threshold`: a value that decides neuron firing.

```python
    weighted_sum = np.dot(inputs, weights)
```

- Computes the **weighted sum**: multiplies each input with its corresponding weight and sums them up.
- `np.dot(inputs, weights)` → **dot product** of `inputs` and `weights`.

```python
    output = 1 if weighted_sum == threshold else 0
```

- If the `weighted_sum` **equals** the `threshold`, output is `1` (neuron fires).
- Else, output is `0` (neuron does not fire).

```python
    return output
```

- Returns the **output** (`1` or `0`) from the `mp_neuron` function.

```python
def and_not(x1, x2):
```

- Defines another function `and_not`, simulating a custom **logic gate** behavior.

```python
    weights = np.array([1, -1])
```

- Sets **weights**:
  - First input (`x1`) has a weight of `+1`.
  - Second input (`x2`) has a weight of `-1`.

```python
    threshold = 1
```

- Sets the **threshold** value to `1`.

```python
    inputs = np.array([x1, x2])
```

* Creates a **numpy array** of the two inputs `[x1, x2]`.

---

python

```python
    output = mp_neuron(inputs, weights, threshold)
```

* Calls the `mp_neuron` function with these inputs, weights, and threshold.
* Stores the **neuron output** (`0` or `1`) in `output`.

---

python

```python
    formula = f"({x1} * 1) + ({x2} * 1) = {np.dot(inputs, weights)}"
```

* (Note: There's a slight mistake here: second weight should be `-1`.)
* This line **creates a string** showing the mathematical formula (but it is not printed anywhere in the code).

---

python

```python
    return output
```

* Returns the **output** from the `and_not` function.

---

python

```python
print("X1\t X2 \t\tOutput")
```

* Prints the table **header** for displaying `X1`, `X2`, and their corresponding output.

---

python

```python
for x1 in [0, 1]:
    for x2 in [0, 1]:
```

* **Nested loop**:
    * First loop: `x1` will be `0` and then `1`.
    * Inside each `x1`, second loop: `x2` will be `0` and then `1`.

---

python

```python
        output = and_not(x1, x2)
```

* Calls the `and_not` function for every combination of `x1` and `x2`.
* Stores the result in `output`.

```python
        print(f"{x1}\t{x2}\t\t{output}")
```

- Prints the values of `x1`, `x2`, and the **output** neatly in tabular format.

🔥 Quick Summary of the Logic:

- It models a **logical operation** where **output is 1 only when** `x1 = 1` **AND** `x2 = 0`.
- Which matches **X1 AND NOT X2** truth table:

X1	X2	Output (X1 AND NOT X2)
0	0	0
0	1	0
1	0	1
1	1	0

Would you also like me to draw a small diagram showing how this neuron looks internally? 🎯 (optional)

```
import numpy as np

step_function = lambda x: 1 if x >= 0 else 0

training_data = [
    {'input': [1, 1, 0, 0, 0, 0], 'label': 1},
    {'input': [1, 1, 0, 0, 0, 1], 'label': 0},
    {'input': [1, 1, 0, 0, 1, 0], 'label': 1},
    {'input': [1, 1, 0, 0, 1, 1], 'label': 0},
    {'input': [1, 1, 0, 1, 0, 0], 'label': 1},
    {'input': [1, 1, 0, 1, 0, 1], 'label': 0},
    {'input': [1, 1, 0, 1, 1, 0], 'label': 1},
    {'input': [1, 1, 0, 1, 1, 1], 'label': 0},
    {'input': [1, 1, 1, 0, 0, 0], 'label': 1},
    {'input': [1, 1, 1, 0, 0, 1], 'label': 0},
]

weights = np.zeros(6)
for _ in range(10):
    for data in training_data:
        inputs = np.array(data['input'])
        label = data['label']
        output = step_function(np.dot(inputs, weights))
        weights += 0.1 * (label - output) * inputs

while True:
```

```
user_input = input("Enter a Number (0-9) or 'exit' to quit: ")
if user_input.lower() == 'exit':
    break
if user_input.isdigit() and 0 <= int(user_input) <= 9:
    inputs = [int(x) for x in '{0:06b}'.format(int(user_input))]
    result = "even" if step_function(np.dot(inputs, weights)) == 1 else "odd"
    print(user_input, "is", result)
else:
    print("Invalid input. Try again.")          ........explain this code line by line
```

Sure! Let's go **line by line** and break down everything very clearly:

```python
import numpy as np
```

- **Imports** the `numpy` library as `np`.
- We'll use `numpy` for **arrays** and **dot product** calculations.

```python
step_function = lambda x: 1 if x >= 0 else 0
```

- Defines a **simple step activation function** using `lambda` (an anonymous function):
  - If input `x` is **greater than or equal to 0**, return `1`.
  - Otherwise, return `0`.
- It's like a basic **thresholding** operation used in perceptrons.

```python
training_data = [
```

- Creates a **list of training examples**.
- Each example is a **dictionary** with:
  - `'input'`: a list of 6 binary values (like 0 or 1).
  - `'label'`: the **target output** (1 for even, 0 for odd).

(list continues...)

Example inside `training_data`:

```python
{'input': [1, 1, 0, 0, 0, 0], 'label': 1}
```

- Input: `[1,1,0,0,0,0]` (binary format of a number).

- Label: `1` means "even".

```python
weights = np.zeros(6)
```

- Initializes **weights** as a numpy array of six **zeros** (`[0,0,0,0,0,0]`).
- One weight for each input feature.

```python
for _ in range(10):
```

- A **loop** that will run 10 times to **train** the weights better.
- `_` is a dummy variable — used when you don't need the loop counter.

```python
    for data in training_data:
```

- For each **training sample** in the `training_data` list:

```python
        inputs = np.array(data['input'])
```

- Extracts the `input` part of the sample and converts it into a **numpy array**.

```python
        label = data['label']
```

- Extracts the **correct output** (label) for that input.

```python
        output = step_function(np.dot(inputs, weights))
```

- Calculates:
  - **Dot product** of inputs and weights (`np.dot(inputs, weights)`).
  - Passes the result to `step_function` to get `0` or `1`.

```python
        weights += 0.1 * (label - output) * inputs
```

- **Perceptron learning rule**:
  - `` `(label - output)` `` calculates the **error**.
  - Multiply by **inputs** → determine how to update each weight.
  - Multiply by **0.1** → learning rate (small changes).
  - Update `` `weights` `` to improve prediction.

✅ **If output is wrong, weights are adjusted slightly.**

Now after training, the code waits for **user input**:

```python
while True:
```

- **Infinite loop** — keeps asking user for a number unless they type 'exit'.

```python
    user_input = input("Enter a Number (0-9) or 'exit' to quit: ")
```

- Asks the user to **enter a number** between 0 and 9 or type `` `'exit'` ``.

```python
    if user_input.lower() == 'exit':
        break
```

- If user types `` `'exit'` `` (in any case: `` `Exit` ``, `` `EXIT` ``, etc.), **breaks** the loop and stops.

```python
    if user_input.isdigit() and 0 <= int(user_input) <= 9:
```

- Checks if:
  - The input is **a digit**.
  - It is between **0 and 9**.

```python
        inputs = [int(x) for x in '{0:06b}'.format(int(user_input))]
```

- **Convert the number into 6-bit binary format**.
  - Example: if user enters `3`, binary is `000011`.
- Then creates a list of integers from the binary string.

```python
        result = "even" if step_function(np.dot(inputs, weights)) == 1 else "odd"
```

- Again:
  - Take the dot product of `inputs` and trained `weights`.
  - Apply `step_function`.
  - If result is `1`, say "even"; otherwise, say "odd".

```python
        print(user_input, "is", result)
```

- Prints the **result** whether the number is **even or odd**.

```python
    else:
        print("Invalid input. Try again.")
```

- If input is not a valid number (0-9), shows an error message.

🔥 Quick Summary:

- It **trains** a **simple perceptron** to classify numbers (0–9) as **even** or **odd** based on **6-bit binary representation**.
- Then it **takes user input** and predicts even/odd after training.

Would you like me to also show you the **truth table** of binary inputs and their even/odd classification that this code is learning internally? 🚀 (It will make things even easier!)

import numpy as np
import matplotlib.pyplot as plt

X = np.array([[0, 0], [1, 0], [0, 1], [1, 1]])
Y = np.array([-1, -1, -1, 1])

w = np.zeros(X.shape[1])
b = 0

for _ in range(6):
    for i in range(X.shape[0]):
        y_pred = np.sign(np.dot(X[i], w) + b)

        if y_pred != Y[i]:
            w += 0.3 * Y[i] * X[i]
            b += 0.3 * Y[i]

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))

Z = np.sign(np.dot(np.c_[xx.ravel(), yy.ravel()], w) + b)
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.8, cmap="coolwarm")
plt.scatter(X[:, 0], X[:, 1], c=Y)
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Perceptron Decision Regions')
plt.show()   ......explain this code line by line

Certainly! Let's go **line by line** and break down the code:

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

- **Imports** the necessary libraries:
  - `numpy` as `np` for numerical operations (arrays, dot product).
  - `matplotlib.pyplot` as `plt` for plotting graphs.

```python
X = np.array([[0, 0], [1, 0], [0, 1], [1, 1]])
Y = np.array([-1, -1, -1, 1])
```

- **X** is a 2D array with 4 input points, representing all possible combinations of binary values (0 and 1) for two variables `X1` and `X2`:
  - `[[0, 0], [1, 0], [0, 1], [1, 1]]`

- **Y** is the target labels corresponding to the input points `` `X` ``:
    - `` `[-1, -1, -1, 1]` `` → The perceptron will learn to classify the points such that:
        - First three points (`` `(0,0), (1,0), (0,1)` ``) have label `` `-1` ``.
        - Last point (`` `(1,1)` ``) has label `` `1` ``.

```python
w = np.zeros(X.shape[1])
b = 0
```

- `` `w` `` initializes the weights of the perceptron as a vector of zeros.
    - `` `X.shape[1]` `` gives the number of input features (2 in this case), so `` `w = [0, 0]` ``.
- `` `b` `` initializes the bias as `` `0` ``.

```python
for _ in range(6):
```

- A **training loop** that runs for 6 iterations. Each iteration will update the weights and bias.

```python
    for i in range(X.shape[0]):
```

- Inner loop iterating over the 4 samples in `` `X` `` (since `` `X.shape[0]` `` is 4, the loop runs for 4 data points).

```python
        y_pred = np.sign(np.dot(X[i], w) + b)
```

- **Prediction** step:
    - `` `np.dot(X[i], w)` `` calculates the **dot product** of input `` `X[i]` `` and the weight vector `` `w` ``.
    - Add the **bias** `` `b` ``.
    - `` `np.sign()` `` applies the **sign function**, returning:
        - `` `1` `` if the result is positive or zero.
        - `` `-1` `` if the result is negative.
    - This gives the predicted label `` `y_pred` `` for the `` `i` ``-th sample.

```python
        if y_pred != Y[i]:
```

- **Condition** to check if the predicted label `y_pred` is **not equal** to the actual label `Y[i]`.

```python
            w += 0.3 * Y[i] * X[i]
            b += 0.3 * Y[i]
```

- **Update rule** if the prediction is wrong:
  - **Weights**: `w += 0.3 * Y[i] * X[i]`
    - `Y[i]` ensures the weight adjustment aligns with the correct class.
    - `0.3` is the **learning rate** (controls the magnitude of the weight update).
  - **Bias**: `b += 0.3 * Y[i]`
    - Adjusts the **bias** in the same manner.

```python
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
```

- **Determine the plot boundaries** for both `X1` and `X2`:
  - `x_min`, `x_max` are the minimum and maximum values of the first feature (`X1`), plus a small margin.
  - `y_min`, `y_max` are the minimum and maximum values of the second feature (`X2`), plus a small margin.

```python
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))
```

- Creates a **grid** of points (`xx`, `yy`) over the entire plot area using `np.meshgrid`.
  - `np.arange(x_min, x_max, 0.01)` creates a range of values from `x_min` to `x_max` with a step size of `0.01`.
  - Similarly for `y_min` to `y_max`.
  - `xx` and `yy` are **grid coordinates** over which we will evaluate the decision boundary.

```python
Z = np.sign(np.dot(np.c_[xx.ravel(), yy.ravel()], w) + b)
Z = Z.reshape(xx.shape)
```

- **Decision boundary**:
  - `np.c_[xx.ravel(), yy.ravel()]` flattens the grid arrays `xx` and `yy` and **combines them** into a 2D array of input coordinates.

    - `np.dot(...)` computes the dot product of the grid points with the weights, adding the bias.
    - `np.sign(...)` applies the sign function to get `1` or `-1` based on whether the dot product + bias is positive or negative.
- `Z.reshape(xx.shape)` reshapes `Z` into the shape of the grid so it can be used for contour plotting.

```python
plt.contourf(xx, yy, Z, alpha=0.8, cmap="coolwarm")
```

- **Plots the decision boundary**:
    - `plt.contourf()` creates **filled contour plots**, showing decision regions (where the perceptron predicts different classes).
    - `alpha=0.8` makes the contours slightly transparent.
    - `cmap="coolwarm"` applies a color map to differentiate the regions.

```python
plt.scatter(X[:, 0], X[:, 1], c=Y)
```

- **Plots the data points**:
    - `X[:, 0]` is the first feature (`X1`), `X[:, 1]` is the second feature (`X2`).
    - `c=Y` colors the points according to their labels (`-1` or `1`).

```python
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Perceptron Decision Regions')
```

- **Adds labels** and **title** to the plot:
    - `'X1'` and `'X2'` for the axes.
    - `'Perceptron Decision Regions'` as the title.

```python
plt.show()
```

- **Displays the plot**.

🔥 Quick Summary of the Code:

- This code implements a **simple perceptron** algorithm that learns to classify two-dimensional data points into two classes (`-1` and `1`).

- The perceptron is trained using the **Perceptron Learning Rule** and updated weights and bias.
- After training, the **decision boundary** is visualized on a plot, showing the regions classified as `-1` or `1`, and the actual training data points are plotted as well.

Let me know if you'd like more details on how to modify this or plot other decision boundaries!

Is this conversation helpful so far?