Importing libraries:

Python

```
import numpy as np
```

This line imports the numpy library which is commonly used for scientific computing in Python. It provides functions and data structures for efficient numerical operations.

Perceptron Class:

Python

```
class Perceptron:
    def __init__(self, input_size, lr=1, epochs=100):
        self.W = np.zeros(input_size+1)
        self.epochs = epochs
        self.lr = lr
```

This defines a class called Perceptron which represents a single Perceptron neuron. The __init__ function is the constructor that gets called whenever you create a new Perceptron object. It takes three arguments:

- input_size: This is the number of input features the Perceptron will receive (in this case, 10 for the 10 digits).
- lr (optional): This is the learning rate, which controls how much the weights are adjusted during training (defaults to 1).
- epochs (optional): This is the number of times the training data will be iterated through during training (defaults to 100).

Inside the constructor:

- self.W = np.zeros(input_size+1): This creates a weight vector W with size input_size + 1 filled with zeros. The extra element (bias) is added at index 0.
- self.epochs = epochs: This stores the number of training epochs.
- self.lr = lr: This stores the learning rate.

Activation Function:

Python

```
def activation_fn(self, x):
    return 1 if x >= 0 else 0
```

This defines a function called activation_fn that takes an input x and applies the activation function. In this case, it's a simple threshold function. It returns 1 if x is greater than or equal to 0, otherwise it returns 0.

Prediction Function:

Python

```
def predict(self, x):
    z = self.W.T.dot(x)
    a = self.activation_fn(z)
    return a
```

This defines a function called predict that takes an input vector x and returns the predicted output of the Perceptron.

1. z = self.W.T.dot(x): This calculates the weighted sum of the input x and the weight vector W. Here, W.T is the transpose of W and dot performs the dot product.
2. a = self.activation_fn(z): This applies the activation function (activation_fn) to the weighted sum z to get the output a.

Training Function:

Python

```
def fit(self, X, d):
    for _ in range(self.epochs):
        for i in range(d.shape[0]):
            x = np.insert(X[i], 0, 1)
            y = self.predict(x)
            e = d[i] - y
            self.W = self.W + self.lr * e * x
```

This defines a function called fit that trains the Perceptron on the provided data. It takes two arguments:

- X: This is a 2D numpy array where each row represents an input data point.

- d: This is a 1D numpy array containing the desired output (labels) for each data point in X.

The training loop iterates through the following steps for the specified number of epochs:

1. Loop through each data point (i) in the training data (X, d).
2. x = np.insert(X[i], 0, 1): This adds a bias term (set to 1) to the beginning of each input data point X[i]. The Perceptron expects a weight for the bias term as well.
3. y = self.predict(x): This predicts the output for the current data point x using the predict function.
4. e = d[i] - y: This calculates the error (e) which is the difference between the desired output (d[i]) and the predicted output (y).
5. self.W = self.W + self.lr * e * x: This updates the weight vector W using the learning rate (lr), the error (e), and the current input data

Sources

1. https://gamedevacademy.org/perceptrons-the-first-neural-networks/

Training Data Preparation:

Python

```
X_train = np.array([
    [0,0,0,0,0,0,0,0,0,0],  # ASCII representation of 0
    [0,0,0,0,0,0,0,0,0,1],  # ASCII representation of 1
    [0,0,0,0,0,0,0,0,1,0],  # ASCII representation of 2
    # ... (all digits 0 to 9)
])

d_train = np.array([0,1,0,1,0,1,0,1,0,1])
```

Here, we define the training data:

- X_train: This is a 2D numpy array where each row represents the ASCII representation of a digit (0 to 9) as a one-hot encoded vector. Each element in the row corresponds to a bit in the ASCII representation. A value of 1 indicates the corresponding bit is set, and 0 indicates it's not set.
- d_train: This is a 1D numpy array containing the desired output (labels) for each data point in X_train. It represents whether the corresponding digit is even (0) or odd (1).

Creating and Training the Perceptron:

Python

```
perceptron = Perceptron(input_size=10)
```

```
perceptron.fit(X_train, d_train)
```

1. perceptron = Perceptron(input_size=10): This creates a new Perceptron object with input_size set to 10 (matching the size of the input data).
2. perceptron.fit(X_train, d_train): This calls the fit function of the Perceptron object to train it on the provided training data (X_train, d_train). The Perceptron will learn the weights that can differentiate between even and odd digits based on their ASCII representations.

User Input and Prediction:

Python

```
test_number = input("Enter a number in ASCII form (0 to 9): ")
test_input = np.array([int(c) for c in test_number])
test_input = np.pad(test_input, (0, 10 - len(test_input)), 'constant')
prediction = perceptron.predict(np.insert(test_input, 0, 1))

test_number = int(test_number, 2)
print(f"Test Number: {test_number}")
if prediction == 1:
    print("Odd")
else:
    print("Even")
```

1. test_number = input("Enter a number in ASCII form (0 to 9): "): This prompts the user to enter a number in ASCII form (0 to 9).
2. test_input = np.array([int(c) for c in test_number]): This converts the user input string (test_number) into a list of integers, where each integer represents the value of a character in the ASCII representation.
3. test_input = np.pad(test_input, (0, 10 - len(test_input)), 'constant'): This checks the length of the converted list (test_input). If it's less than 10 (shorter ASCII representation), it pads the list with zeros at the end ((0,)) to ensure a consistent size of 10 for the Perceptron.
4. prediction = perceptron.predict(np.insert(test_input, 0, 1)): This creates a copy of the padded input (test_input) and inserts a bias term (1) at the beginning using np.insert. Then, it calls the predict function of the trained Perceptron to get the predicted output (prediction) for the user-provided number.
5. test_number = int(test_number, 2): This converts the original user input string (test_number) from its ASCII representation to its actual integer value using base 2 conversion (int(, 2)).
6. print(f"Test Number: {test_number}"): This prints the actual integer value of the user-provided number.
7. if prediction == 1:: This checks if the predicted output (prediction) from the Perceptron is 1 (odd).
    - print("Odd"): If the prediction is 1, it prints "Odd". 8

In this code, self is the first argument passed to the constructor (__init__) and methods (functions) within the Perceptron class. It's a special parameter that refers to the current instance of the Perceptron object itself.

Here's a breakdown of how self is used:

1. **Constructor (__init__):**
   - When you create a new Perceptron object (e.g., perceptron = Perceptron(input_size=10)), Python automatically passes the newly created object as the first argument (self) to the __init__ function.
   - Inside the __init__ function, you can access and modify the attributes (like W, epochs, and lr) of the specific Perceptron object using self. For instance, self.W = np.zeros(input_size+1) assigns a weight vector of zeros to the current object (perceptron).
2. **Methods:**
   - All methods defined within the Perceptron class (like activation_fn, predict, and fit) also take self as the first argument.
   - These methods use self to access the object's attributes and perform operations specific to that particular object.

For example, in the predict function:

Python

```
def predict(self, x):
  z = self.W.T.dot(x)
  a = self.activation_fn(z)
  return a
```

- self.W.T: Here, the method accesses the weight vector (W) attribute of the current object (perceptron) using self.
- Similarly, self.activation_fn(z) calls the activation_fn function on the object (perceptron).

In essence, self allows each Perceptron object to have its own set of weights and perform calculations independently while sharing the same functionality defined within the class methods.

Absolutely, let's break down the line self.W = np.zeros(input_size+1) step-by-step and explain it with an example:

1. self.W:

- This refers to an attribute of the current object (instance) of the Perceptron class. When you create a new Perceptron object, it will have various attributes associated with it, and W is one such attribute.
- In this context, W will represent the weight vector of the Perceptron. The weights are crucial for the Perceptron to learn and make predictions.

2. np.zeros(input_size+1):

- This part creates a new numpy array filled with zeros.
- np.zeros is a function from the numpy library used to generate arrays of zeros.
- (input_size+1) is the argument passed to np.zeros which specifies the size of the array. In this case, the size is determined by input_size (which is the number of input features) plus 1.

Example:

Imagine you create a Perceptron object with input_size=2 (meaning it takes 2 input features). In this case:

- self.W = np.zeros(input_size+1) becomes self.W = np.zeros(2+1).
- The np.zeros(3) function will create a 1D numpy array with 3 elements, all initialized to zero.
- So, after this line executes, the self.W attribute of your Perceptron object will hold a weight vector [0, 0, 0].

Why input_size + 1?

The reason for input_size + 1 is to accommodate the bias term in the Perceptron. The bias term is a constant value added to the weighted sum of the inputs before applying the activation function. It allows the Perceptron to shift the decision boundary and improve its ability to learn certain patterns. By allocating an extra element in the weight vector (self.W), the first element can be dedicated to the bias term (often set to 1), and the remaining elements hold the weights for the actual input features.