Absolutely, let's break down the code step-by-step:

Importing libraries:

Python

```
import numpy as np
```

This line imports the numpy library which is commonly used for scientific computing in Python. Neural networks rely heavily on numerical operations like matrix multiplication, so numpy provides efficient tools for these tasks.

Defining the NeuralNetwork Class:

Python

```
class NeuralNetwork:
  def __init__(self, input_size, hidden_size, output_size):
    # ...
```

This defines a class called NeuralNetwork. The __init__ function is the constructor of the class, which gets called whenever we create a new instance of the NeuralNetwork. It takes three arguments:

- input_size: This specifies the number of features in the input data.
- hidden_size: This specifies the number of neurons in the hidden layer.
- output_size: This specifies the number of output neurons, which determines the number of possible predictions the network can make.

Initializing Weights:

Python

```
  def __init__(self, input_size, hidden_size, output_size):
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size
    #initialize weights
    self.W1 = np.random.randn(self.input_size, self.hidden_size)
```

```
    self.W2 = np.random.randn(self.hidden_size, self.output_size)
```

Here, we initialize two weight matrices:

- self.W1: This weight matrix connects the input layer to the hidden layer. It has dimensions (input_size, hidden_size). Each element W1[i][j] represents the weight of the connection between the i-th neuron in the input layer and the j-th neuron in the hidden layer.
- self.W2: This weight matrix connects the hidden layer to the output layer. It has dimensions (hidden_size, output_size). Similar to W1, each element W2[i][j] represents the weight of the connection between the i-th neuron in the hidden layer and the j-th neuron in the output layer.

The weights are initialized with random values using np.random.randn. These random values will be adjusted during the training process.

Activation Function (Sigmoid):

Python

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

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

These two functions define the activation function used in the network. The code defines a sigmoid function and its derivative.

- sigmoid(x): This function takes an input x and applies the sigmoid activation function. The sigmoid function maps a real number between negative infinity and positive infinity to a value between 0 and 1.
- sigmoid_derivative(x): This function calculates the derivative of the sigmoid function. The derivative is used in the backpropagation algorithm during training.

Forward Pass Function:

Python

```
def forward(self, X):
    #calculate output of hidden Layer
    self.z = np.dot(X, self.W1)
    self.z2 = self.sigmoid(self.z)
    #calculate output of output Layer
    self.z3 = np.dot(self.z2, self.W2)
```

```
    output = self.sigmoid(self.z3)
    return output
```

This function defines the forward pass of the neural network. It takes the input data X as input and returns the predicted output. Here's what happens step-by-step:

1. self.z = np.dot(X, self.W1): This line calculates the weighted sum of the inputs in the first layer. It multiplies the input data X with the weight matrix W1 using matrix multiplication (np.dot). The result (self.z) represents the sum of the weighted inputs for each neuron in the hidden layer.
2. self.z2 = self.sigmoid(self.z): This line applies the sigmoid activation function to the weighted sum (self.z) for each neuron in the hidden layer. This introduces non-linearity to the network, allowing it to learn complex patterns. The result (self.z2) represents the activation values of the hidden layer neurons.
3. self.z3 = np.dot(self.z2, self.W2): Similar to step 1, this line calculates the weighted sum of the

Sources

1. https://stackoverflow.com/questions/54188873/valueerror-shapes-2-and-5-not-aligned-2-dim-0-5-dim-0
2. https://github.com/fayssalab/DataScience

Sure, continuing the breakdown of the code:

Forward Pass Function (continued):

- self.z3 = np.dot(self.z2, self.W2): Similar to step 1, this line calculates the weighted sum of the activations from the hidden layer (self.z2) with the weights connecting the hidden layer to the output layer (W2).
- output = self.sigmoid(self.z3): This line applies the sigmoid activation function again to the weighted sum (self.z3) for each neuron in the output layer. The final result (output) represents the network's prediction for the input data.

Backward Pass Function:

Python

```
  def backward(self, X, y, output):
    #calculate the error and derivative of error for output Layer
    self.output_error = y - output
    self.output_delta = self.output_error *
self.sigmoid_derivative(output)
    #calculate error and derivative of error for hidden Layer
    self.z2_error = self.output_delta.dot(self.W2.T)
```

```
    self.z2_delta = self.z2_error * self.sigmoid_derivative(self.z2)
    #update weights
    self.W1 += X.T.dot(self.z2_delta)
    self.W2 += self.z2.T.dot(self.output_delta)
```

This function defines the backward pass, which is used during training to update the weights of the network. Here's what happens:

1. self.output_error = y - output: This line calculates the difference between the actual target value (y) and the predicted output (output). This represents the error of the network's prediction.
2. self.output_delta = self.output_error * self.sigmoid_derivative(output): This line calculates the delta value for the output layer. The delta represents the contribution of each neuron in the output layer to the overall error. It's calculated by multiplying the error (output_error) with the derivative of the activation function applied to the output layer (sigmoid_derivative(output)).
3. self.z2_error = self.output_delta.dot(self.W2.T): This line calculates the error for the hidden layer. It considers how the weights between the hidden and output layers (W2.T - transposed) contribute to the overall error in the output layer (output_delta).
4. self.z2_delta = self.z2_error * self.sigmoid_derivative(self.z2): This line calculates the delta value for the hidden layer. Similar to the output layer, it considers the error in the hidden layer (z2_error) and the derivative of the activation function applied to the hidden layer (sigmoid_derivative(z2)).
5. self.W1 += X.T.dot(self.z2_delta): This line updates the weights between the input layer and the hidden layer (W1). It uses the learning rate (which isn't explicitly shown here but is typically a small value) multiplied by the product of the transposed input data (X.T) and the delta values of the hidden layer (z2_delta). This update adjusts the weights based on how much the hidden layer neurons contributed to the error.
6. self.W2 += self.z2.T.dot(self.output_delta): Similarly, this line updates the weights between the hidden layer and the output layer (W2). It uses the learning rate multiplied by the product of the transposed activations of the hidden layer (z2.T) and the delta values of the output layer (output_delta).

Training Function:

Python

```
  def train(self, X, y, epochs):
    for i in range(epochs):
      #forward propagation
      output = self.forward(X)
      #backward propagation
      self.backward(X, y, output)
```

This function defines the training process. It takes the training data (X - features, y - targets) and

the number of training epochs (epochs) as input. Here's what happens:

1. It iterates for the specified number of epochs (epochs).
2. Inside the loop, it performs a forward pass using self.forward(X) to calculate the network's prediction for the input data.
3. Then, it performs a backward pass using self.backward(X, y, output) to calculate the errors and update the weights based on the difference between the predictions and the actual targets.

Creating a Neural Network Object:

Python

```
#create a neural network object by specifying the number of inputs, hidden units,
nn = NeauralNetwork(input_size=2,hidden_size=3, output_size=1)
```

Here, we create an instance of the NeuralNetwork class. We specify the following arguments:

- input_size=2: This indicates that the input data has two features.
- hidden_size=3: This defines the number of neurons in the hidden layer, set to 3 in this example.
- output_size=1: This specifies that the network will make a single prediction (binary classification in this case).

Specifying Training Data:

Python

```
#specify train data 'X' and target output 'y'
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
```

Here, we define the training data:

- X: This is a NumPy array that holds the input data for training. It has a shape of (4, 2), where 4 represents the number of data points (samples) and 2 represents the two features for each data point.
- y: This is a NumPy array that holds the target outputs for each data point in X. It has a shape

of (4, 1), where each element represents the desired output (0 or 1) for the corresponding data point in X.

Training the Network:

Python

```
#train the network
nn.train(X, y, epochs=10000)
```

This line calls the train function of the neural network object (nn). It trains the network using the provided training data (X and y) for a specified number of epochs (epochs=10000). During training, the network performs forward and backward passes to adjust its weights and learn from the data.

Making Predictions on New Data:

Python

```
#make predictions on new data
new_data = np.array([[0, 0.5], [0, 0.8], [1, 0.2], [1, 0.6]])
predictions = nn.forward(new_data)
print(predictions)
```

Here, we define a new data set (new_data) with four data points, each having two features. Then, we call the forward function of the trained network (nn) on this new data. This performs a forward pass through the network using the learned weights, and the resulting predictions are stored in the predictions variable. Finally, the print(predictions) line displays the network's predicted outputs for the new data points.

Overall, this code demonstrates a basic implementation of a neural network for binary classification. By training the network on sample data, it learns to identify patterns and make predictions on unseen data.