Sure, here's a concise explanation for your documentation:

NeuralNetwork Class

The `NeuralNetwork` class implements a basic feedforward neural network capable of training and making predictions. It accepts the following parameters during initialization:

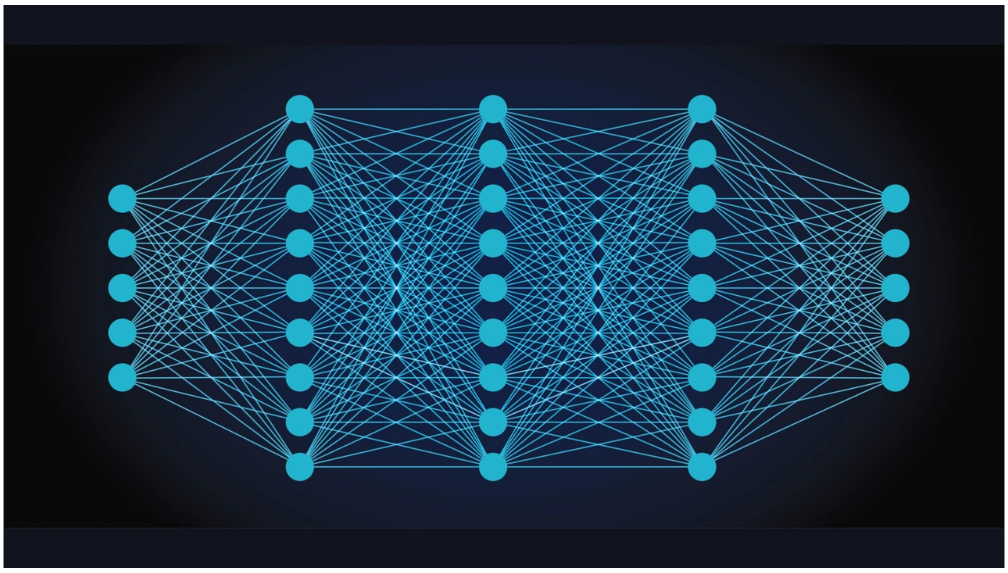
- **`input`**: Represents the input data or features for the neural network.
- **`hiddenlayer`**: Specifies the number of neurons in each hidden layer as a list of integers.
- **`activation_hidden`**: Tuple containing the activation function and its derivative for the hidden layers.
- **`outputlayer`**: Specifies the number of neurons in the output layer.
- **`idealValues`**: Target output values for training purposes.
- **`activation_output`**: Tuple containing the activation function and its derivative for the output layer.
- **`learning_rate`**: Determines the step size in gradient descent optimization during training.

Example Usage

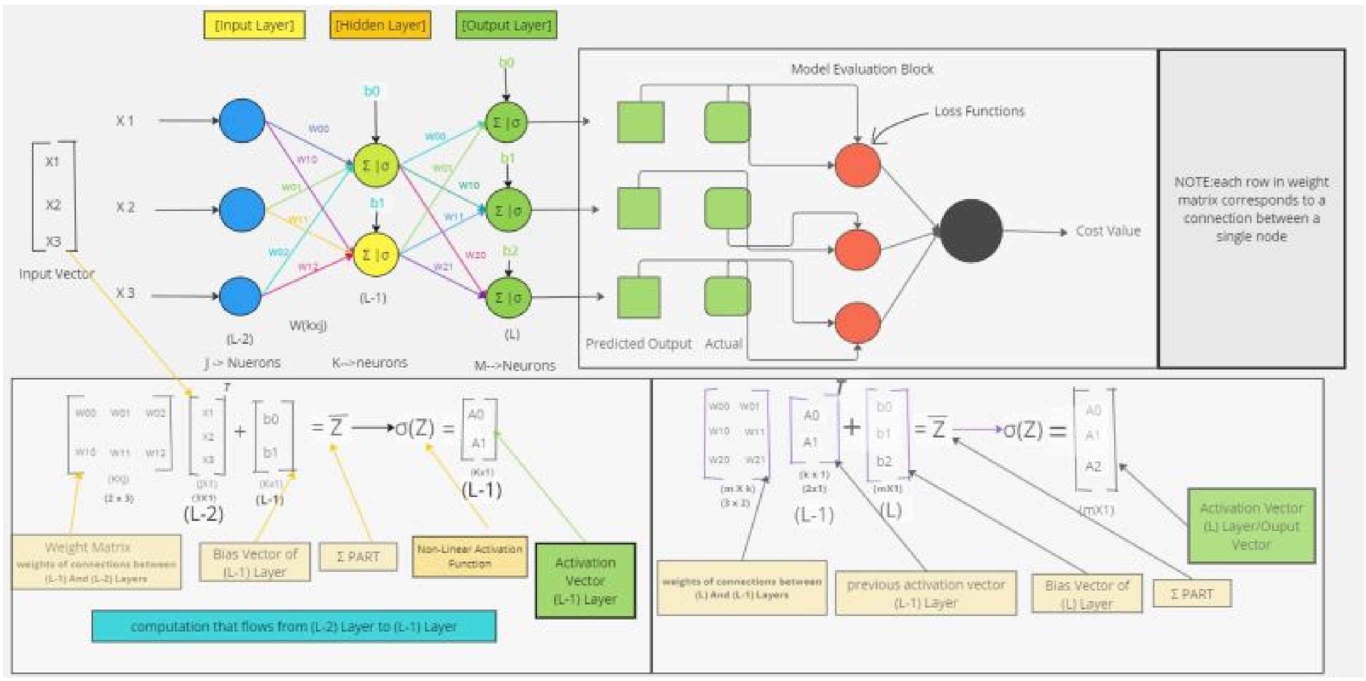
```python
# Example instantiation
NN = NeuralNetwork(
    input=[1, 2, 3],
    hiddenlayer=[4],
    activation_hidden=(activations.LeakyRelu, activations.LeakyRelu_derivative),
    outputlayer=2,
    idealValues=[1, 2],
    activation_output=(activations.sigmoid, activations.sigmoid_derivative),
    learning_rate=0.001
)
```

Purpose

The `NeuralNetwork` class encapsulates the functionality for creating and training a neural network model. It provides methods to initialize the network, perform forward and backward propagation, compute loss, and update model parameters using gradient descent.



Sure! Let's go through the forward pass code, explaining each line and the underlying mathematical concepts.


[Input Layer]
[Hidden Layer]
[Output Layer]
Input Vector
X1
X2
X3
X 1
X 2
X 3
b0
b1
b2
w00
w10
w01
w11
w02
w12
w20
w21
Σ |σ
(L-2)
(L-1)
(L)
W(kxj)
J -> Nuerons
K-->neurons
M-->Neurons
Model Evaluation Block
Loss Functions
Cost Value
Predicted Output
Actual
NOTE:each row in weight matrix corresponds to a connection between a single node
(kxj)
(2 x 3)
(jX1)
(3X1)
(Kx1)
= Z
σ(Z) =
A0
A1
A2
Weight Matrix
weights of connections between (L-1) And (L-2) Layers
Bias Vector of (L-1) Layer
Σ PART
Non-Linear Activation Function
Activation Vector (L-1) Layer
computation that flows from (L-2) Layer to (L-1) Layer
(m X k)
(3 x 2)
(k x 1)
(2x1)
(mX1)
weights of connections between (L) And (L-1) Layers
previous activation vector (L-1) Layer
Bias Vector of (L) Layer
Activation Vector (L) Layer/Ouput Vector

Forward Pass Explained

```python
def Forwardpass(self):
    for i in range(len(self.W) - 1):
        Mat_Vec_Mul = self.W[i] @ np.transpose(self.A[i])  # weight and
activation(L-1) multiplication , weight matrix of (L) and (L-1)

        shape = Mat_Vec_Mul.shape  # to adjust the shape from (i x 1 ) of vectors
to (i,) to reduce errors in computation
        Z = (Mat_Vec_Mul.reshape(shape[0])) + self.b[i]  # Z = wx + b --> weighted
sum
        self.Z.append(Z)
        A = self.activation_hidden[0](Z)
        self.A.append(A)
    Mat_Vec_Mul = self.W[-1] @ np.transpose(self.A[-1])
    shape = Mat_Vec_Mul.shape  # to adjust the shape from (i x 1 ) --> (i,) to
reduce errors in computation
    Z = (Mat_Vec_Mul.reshape(shape[0])) + self.b[-1]
    self.Z.append(Z)
    A = self.activation_output[0](Z)
    self.A.append(A)
    Loss = self.Loss()
    return Loss
```

Mathematical Explanation and Chain Rule Application

1. Initialize the Forward Pass

```
for i in range(len(self.W) - 1):
```

- **Explanation**: Iterate through each layer except the last one (output layer).

2. Weighted Sum Calculation

```
Mat_Vec_Mul = self.W[i] @ np.transpose(self.A[i])
```

- **Explanation**: Calculate the weighted sum of inputs from the previous layer. This is the dot product of the weights (`self.W[i]`) and the activations from the previous layer (`self.A[i]`).
- **Mathematical Notation**: (Z^{(l)} = W^{(l)} \cdot A^{(l-1)})
- **Chain Rule Application**: Not directly applicable here, but this prepares for the activation function.

3. Adjust Shape for Computation

```
shape = Mat_Vec_Mul.shape
Z = (Mat_Vec_Mul.reshape(shape[0])) + self.b[i]
```

- **Explanation**: Adjust the shape of the matrix-vector multiplication result and add the bias term.
- **Mathematical Notation**: (Z^{(l)} = W^{(l)} \cdot A^{(l-1)} + b^{(l)})

4. Activation Function

```
self.Z.append(Z)
A = self.activation_hidden[0](Z)
self.A.append(A)
```

- **Explanation**: Apply the activation function to the weighted sum `Z` to get the activation for the current layer. Store `Z` and `A` for later use.
- **Mathematical Notation**: ($A^{(l)} = \sigma(Z^{(l)})$)
- **Chain Rule Application**: This prepares the activations for the next layer and for backpropagation.

5. Output Layer Weighted Sum

```
Mat_Vec_Mul = self.W[-1] @ np.transpose(self.A[-1])
shape = Mat_Vec_Mul.shape
Z = (Mat_Vec_Mul.reshape(shape[0])) + self.b[-1]
```

- **Explanation**: Calculate the weighted sum for the output layer.
- **Mathematical Notation**: ($Z^{(L)} = W^{(L)} \cdot A^{(L-1)} + b^{(L)}$)

6. Output Layer Activation Function

```
self.Z.append(Z)
A = self.activation_output[0](Z)
self.A.append(A)
```

- **Explanation**: Apply the activation function to the weighted sum `Z` of the output layer to get the final output.
- **Mathematical Notation**: ($\hat{y} = \sigma(Z^{(L)})$)

7. Compute Loss

```
Loss = self.Loss()
return Loss
```

- **Explanation**: Calculate the loss function to measure the difference between the predicted values and the actual values.
- **Mathematical Notation**: ($L = \frac{1}{N} \sum_{i=1}^{N} (\hat{y} - y)^2$)

Summary

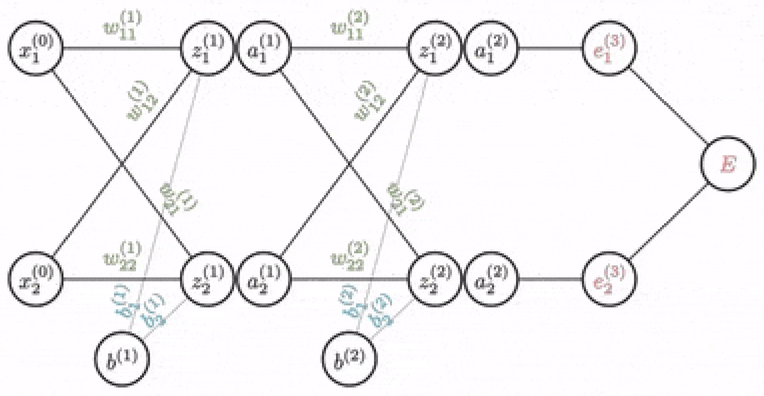
The forward pass involves computing the weighted sum and applying the activation function for each layer sequentially from the input to the output layer. Each line of code in the forward pass implements these steps, setting up the network for backpropagation by storing the activations and weighted sums needed for calculating gradients. The chain rule is implicitly applied in the sense that each layer's output becomes the input for the next layer, ensuring that the gradients can be propagated backward during the training process.

Sure! Let's delve into the backward pass of the neural network, explaining each line and the underlying mathematics, including the application of the chain rule of derivatives.

Backward Pass Explained

$$\frac{\partial E}{\partial w_{11}^{(2)}} = \frac{\partial e_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{11}^{(2)}} \qquad \frac{\partial E}{\partial w_{12}^{(2)}} = \frac{\partial e_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}}$$

$$\frac{\partial E}{\partial w_{21}^{(2)}} = \frac{\partial e_2^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \frac{\partial z_2^{(2)}}{\partial w_{21}^{(2)}} \qquad \frac{\partial E}{\partial w_{22}^{(2)}} = \frac{\partial e_2^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \frac{\partial z_2^{(2)}}{\partial w_{22}^{(2)}}$$



```
def Backwardpass(self):
    # Output layer error
    output_error = (self.A[-1] - self.idealValues)
    output_delta = output_error * self.activation_output_derivative(self.Z[-1])
## delta(L) = Error * d(sigma(L))/dz = activation_output_derivative(Z(L))

    # Reshape for correct dimensions
    output_delta = output_delta.reshape(-1, 1)
    self.A[-2] = self.A[-2].reshape(1, -1)

    # Gradients for output layer
    dW = output_delta @ self.A[-2]
    db = np.sum(output_delta, axis=1)

    self.W[-1] -= self.learning_rate * dW
    self.b[-1] -= self.learning_rate * db

    # Backpropagate through hidden layers
    delta = output_delta
    for i in range(len(self.W) - 2, -1, -1):
        delta = (self.W[i + 1].T @ delta).reshape(-1) *
self.activation_hidden_derivative(self.Z[i])  ## W(L+1,L)^T * delta(L) *
d(sigma(L-i))/dz = activation_hidden_derivative(self.Z[i])

        # Reshape for correct dimensions
        delta = delta.reshape(-1, 1)
        self.A[i] = self.A[i].reshape(1, -1)

        dW = delta @ self.A[i]
        db = np.sum(delta, axis=1)
```

```
            self.W[i] -= self.learning_rate * dW
            self.b[i] -= self.learning_rate * db
```

Mathematical Explanation and Chain Rule Application

1. Output Layer Error

```
output_error = (self.A[-1] - self.idealValues)
```

- **Explanation**: The output layer error is the difference between the predicted values (`self.A[-1]`) and the ideal (target) values (`self.idealValues`).
- **Mathematical Notation**: (E = \hat{y} - y)

2. Output Delta

```
output_delta = output_error * self.activation_output_derivative(self.Z[-1])
```

- **Explanation**: Multiply the error by the derivative of the activation function of the output layer. This gives the gradient of the loss with respect to the weighted sum (Z) of the output layer.
- **Mathematical Notation**: (\delta^{(L)} = E \cdot \sigma' (Z^{(L)}))
- **Chain Rule Application**:
  - Error: (\frac{\partial L}{\partial \hat{y}})
  - Activation derivative: (\frac{\partial \hat{y}}{\partial Z^{(L)}})
  - Combined: (\delta^{(L)} = \frac{\partial L}{\partial Z^{(L)}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial Z^{(L)}})

3. Reshape for Correct Dimensions

```
output_delta = output_delta.reshape(-1, 1)
self.A[-2] = self.A[-2].reshape(1, -1)
```

- **Explanation**: Reshape the delta and the activations for matrix multiplication.

4. Gradients for Output Layer

```
dW = output_delta @ self.A[-2]
db = np.sum(output_delta, axis=1)
```

- **Explanation**: Calculate the gradients for the weights and biases in the output layer.
- **Mathematical Notation**:

  - Weight gradients: (\frac{\partial L}{\partial W^{(L)}} = \delta^{(L)} \cdot A^{(L-1)})
  - Bias gradients: (\frac{\partial L}{\partial b^{(L)}} = \delta^{(L)})
- **Chain Rule Application**:
  - For weights: (\frac{\partial L}{\partial W^{(L)}} = \frac{\partial L}{\partial Z^{(L)}} \cdot \frac{\partial Z^{(L)}}{\partial W^{(L)}})
  - For biases: (\frac{\partial L}{\partial b^{(L)}} = \frac{\partial L}{\partial Z^{(L)}} \cdot \frac{\partial Z^{(L)}}{\partial b^{(L)}})

5. Update Output Layer Weights and Biases

```
self.W[-1] -= self.learning_rate * dW
self.b[-1] -= self.learning_rate * db
```

- **Explanation**: Update the weights and biases using the calculated gradients and the learning rate.
- **Mathematical Notation**:
  - Weights update: (W^{(L)} \leftarrow W^{(L)} - \eta \cdot \frac{\partial L}{\partial W^{(L)}})
  - Biases update: (b^{(L)} \leftarrow b^{(L)} - \eta \cdot \frac{\partial L}{\partial b^{(L)}})

6. Backpropagate Through Hidden Layers

```
delta = output_delta
for i in range(len(self.W) - 2, -1, -1):
    delta = (self.W[i + 1].T @ delta).reshape(-1) *
self.activation_hidden_derivative(self.Z[i])
```

- **Explanation**: Backpropagate the delta through each hidden layer.
- **Mathematical Notation**:
  - For layer (l): (\delta^{(l)} = (\delta^{(l+1)} \cdot W^{(l+1)}) \cdot \sigma' (Z^{(l)}))
- **Chain Rule Application**:
  - For hidden layers: (\delta^{(l)} = \frac{\partial L}{\partial Z^{(l)}} = \left(\frac{\partial L}{\partial A^{(l+1)}} \cdot \frac{\partial A^{(l+1)}}{\partial Z^{(l+1)}} \cdot \frac{\partial Z^{(l+1)}}{\partial A^{(l)}} \right) \cdot \frac{\partial A^{(l)}}{\partial Z^{(l)}})

7. Reshape for Correct Dimensions

```
delta = delta.reshape(-1, 1)
self.A[i] = self.A[i].reshape(1, -1)
```

- **Explanation**: Reshape the delta and the activations for matrix multiplication.

8. Gradients for Hidden Layers

```
dW = delta @ self.A[i]
db = np.sum(delta, axis=1)
```

- **Explanation**: Calculate the gradients for the weights and biases in the hidden layers.
- **Mathematical Notation**:
  - Weight gradients: $\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)} \cdot A^{(l-1)}$
  - Bias gradients: $\frac{\partial L}{\partial b^{(l)}} = \delta^{(l)}$
- **Chain Rule Application**:
  - For weights: $\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial Z^{(l)}} \cdot \frac{\partial Z^{(l)}}{\partial W^{(l)}}$
  - For biases: $\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial Z^{(l)}} \cdot \frac{\partial Z^{(l)}}{\partial b^{(l)}}$

9. Update Hidden Layer Weights and Biases

```
self.W[i] -= self.learning_rate * dW
self.b[i] -= self.learning_rate * db
```

- **Explanation**: Update the weights and biases using the calculated gradients and the learning rate.
- **Mathematical Notation**:
  - Weights update: $W^{(l)} \leftarrow W^{(l)} - \eta \cdot \frac{\partial L}{\partial W^{(l)}}$
  - Biases update: $b^{(l)} \leftarrow b^{(l)} - \eta \cdot \frac{\partial L}{\partial b^{(l)}}$

Summary

The backward pass involves computing the error and gradients for each layer starting from the output layer and moving backward through the hidden layers. The chain rule is applied to propagate the error gradients backward, allowing the network to update its weights and biases to minimize the loss function. Each line of code in the backward pass implements a step in this gradient descent optimization process.