Dokumentacja Biblioteki do Sieci Neuronowych

Spis Treści

Sieć Neuronowa

Klasa: `Network`

Klasa `Network` implementuje główną strukturę sieci neuronowej.

- **Konstruktor: `__init__(self, loss, optimizer)`**
  - **Parametry**:
    - `loss` (funkcja straty): Funkcja straty do użycia podczas trenowania.
    - `optimizer` (optymalizator): Optymalizator do aktualizacji wag.
- **Metoda: `train_mode(self)`**
  - **Opis**: Ustawia tryb sieci na treningowy.
- **Metoda: `evaluation_mode(self)`**
  - **Opis**: Ustawia tryb sieci na ewaluacyjny.
- **Metoda: `add_layer(self, layer, layer_inputs, layer_outputs, distribution="normal")`**
  - **Opis**: Dodaje warstwę do sieci.
  - **Parametry**:
    - `layer` (klasa warstwy): Typ warstwy do dodania.
    - `layer_inputs` (int): Liczba neuronów wejściowych.
    - `layer_outputs` (int): Liczba neuronów wyjściowych.
    - `distribution` (str): Typ rozkładu do inicjalizacji wag (domyślnie “normal”).
- **Metoda: `add_dropout(self, layer_inputs, fraction=0.5)`**
  - **Opis**: Dodaje warstwę Dropout do sieci.
  - **Parametry**:
    - `layer_inputs` (int): Liczba neuronów wejściowych.
    - `fraction` (float): Ułamek jednostek do wyłączenia (domyślnie 0.5).
- **Metoda: `add_activation(self, activation)`**
  - **Opis**: Dodaje funkcję aktywacji do sieci.
  - **Parametry**:
    - `activation` (klasa funkcji aktywacji): Typ funkcji aktywacji do dodania.
- **Metoda: `evaluate(self, x)`**
  - **Opis**: Przeprowadza propagację w przód (forward pass) przez sieć.
  - **Parametry**:
    - `x` (numpy array): Dane wejściowe.
  - **Zwraca**: Wynik propagacji w przód.
- **Metoda: `train(self, x, y)`**
  - **Opis**: Trenuje sieć na podanych danych.
  - **Parametry**:
    - `x` (numpy array): Dane wejściowe.
    - `y` (numpy array): Prawdziwe wartości (etykiety).
  - **Zwraca**: Strata po jednej epoce treningu.

Przykład użycia

```
import numpy as np
from Network import Network
from Linear import Linear
from ActivationFunctions import Relu, Sigmoid
from LossFunctions import MSELoss
from Optimizers import SGD
```

```
# Inicjalizacja sieci
loss_function = MSELoss
optimizer = SGD(learning_rate=0.01)
network = Network(loss_function, optimizer)

# Dodawanie warstw do sieci
network.add_layer(Linear, 3, 5, distribution="normal_xavier")
network.add_activation(Relu())
network.add_layer(Linear, 5, 2, distribution="normal_xavier")
network.add_activation(Sigmoid())

# Tryb treningowy
network.train_mode()

# Przykładowe dane
x = np.random.rand(10, 3)
y = np.random.rand(10, 2)

# Trenowanie sieci
loss = network.train(x, y)
print("Loss:", loss)

# Tryb ewaluacyjny
network.evaluation_mode()

# Ewaluacja sieci
predictions = network.evaluate(x)
print("Predictions:", predictions)
```

Ładowanie Danych

Klasa: `DataLoader`

Klasa `DataLoader` umożliwia efektywne ładowanie i iterację przez zestawy danych podczas trenowania i ewaluacji.

- **Konstruktor: `__init__(self, inputs, labels, batch_size=1, shuffle=False, drop_last=False)`**
  - **Parametry**:
    - `inputs` (numpy array): Dane wejściowe.
    - `labels` (numpy array): Etykiety danych.
    - `batch_size` (int): Rozmiar batcha (domyślnie 1).
    - `shuffle` (bool): Czy przetasować dane (domyślnie False).
    - `drop_last` (bool): Czy zignorować ostatni batch, jeśli jest niepełny (domyślnie False).
- **Metoda: `__iter__(self)`**
  - **Opis**: Tworzy iterator dla batchy danych.
  - **Zwraca**: Iterator zwracający batche danych i etykiet.

Przykład użycia

```
import numpy as np
from DataLoader import DataLoader

# Przykładowe dane
inputs = np.random.rand(100, 3)
labels = np.random.rand(100, 1)

# Inicjalizacja DataLoader
data_loader = DataLoader(inputs, labels, batch_size=10, shuffle=True)

# Iteracja przez batchy danych
for batch_inputs, batch_labels in data_loader:
```

```
print("Batch inputs:", batch_inputs)
print("Batch labels:", batch_labels)
```

Warstwa Liniowa

Klasa: `Linear`

Klasa `Linear` definiuje warstwę liniową używaną w sieci.

- **Konstruktor: `__init__(self, input_neurons, output_neurons, optimizer, distribution)`**
  - **Parametry**:
    - `input_neurons` (int): Liczba neuronów wejściowych.
    - `output_neurons` (int): Liczba neuronów wyjściowych.
    - `optimizer` (optymalizator): Optymalizator do aktualizacji wag.
    - `distribution` (str): Typ rozkładu do inicjalizacji wag ("normal_xavier", "uniform_xavier", "normal").
- **Metoda: `forward(self, x)`**
  - **Opis**: Przeprowadza propagację w przód przez warstwę.
  - **Parametry**:
    - `x` (numpy array): Dane wejściowe.
  - **Zwraca**: Wynik propagacji w przód.
- **Metoda: `backward(self, gradient, layer_input)`**
  - **Opis**: Przeprowadza propagację wstecz (backpropagation) przez warstwę.
  - **Parametry**:
    - `gradient` (numpy array): Gradient z następnej warstwy.
    - `layer_input` (numpy array): Dane wejściowe do warstwy podczas propagacji w przód.
  - **Zwraca**: Gradient dla poprzedniej warstwy.

Przykład użycia

```
import numpy as np
from Linear import Linear
from Optimizers import SGD

# Przykładowe dane wejściowe
x = np.random.rand(10, 3)

# Inicjalizacja optymalizatora
optimizer = SGD(learning_rate=0.01)

# Inicjalizacja warstwy liniowej
linear = Linear(input_neurons=3, output_neurons=2, optimizer=optimizer, distribution="normal_xavier")

# Propagacja w przód
output = linear.forward(x)
print("Linear layer output:", output)

# Przykładowy gradient
gradient = np.random.rand(10, 2)

# Propagacja wstecz
grad_input = linear.backward(gradient, x)
print("Gradient input:", grad_input)
```

Funkcje Aktywacji

Klasa: `Sigmoid`

Klasa `Sigmoid` implementuje funkcję aktywacji sigmoid.

- **Metoda: `sig(self, layer_input)`**
  - **Opis**: Oblicza wartość funkcji sigmoid.
  - **Parametry**:

        * `layer_input` (numpy array): Dane wejściowe.
    - **Zwraca**: Wartość funkcji sigmoid.
- **Metoda: `forward(self, layer_input)`**
    - **Opis**: Przeprowadza propagację w przód.
    - **Parametry**:
        * `layer_input` (numpy array): Dane wejściowe.
    - **Zwraca**: Wynik propagacji w przód.
- **Metoda: `backward(self, gradient, layer_input)`**
    - **Opis**: Przeprowadza propagację wstecz.
    - **Parametry**:
        * `gradient` (numpy array): Gradient z następnej warstwy.
        * `layer_input` (numpy array): Dane

wejściowe do warstwy podczas propagacji w przód. - **Zwraca**: Gradient dla poprzedniej warstwy.

Klasa: `Tanh`

Klasa `Tanh` implementuje funkcję aktywacji tanh.

- **Metoda: `tanh(self, layer_input)`**
    - **Opis**: Oblicza wartość funkcji tanh.
    - **Parametry**:
        * `layer_input` (numpy array): Dane wejściowe.
    - **Zwraca**: Wartość funkcji tanh.
- **Metoda: `forward(self, layer_input)`**
    - **Opis**: Przeprowadza propagację w przód.
    - **Parametry**:
        * `layer_input` (numpy array): Dane wejściowe.
    - **Zwraca**: Wynik propagacji w przód.
- **Metoda: `backward(self, gradient, layer_input)`**
    - **Opis**: Przeprowadza propagację wstecz.
    - **Parametry**:
        * `gradient` (numpy array): Gradient z następnej warstwy.
        * `layer_input` (numpy array): Dane wejściowe do warstwy podczas propagacji w przód.
    - **Zwraca**: Gradient dla poprzedniej warstwy.

Klasa: `Relu`

Klasa `Relu` implementuje funkcję aktywacji ReLU.

- **Metoda: `forward(self, layer_input)`**
    - **Opis**: Przeprowadza propagację w przód.
    - **Parametry**:
        * `layer_input` (numpy array): Dane wejściowe.
    - **Zwraca**: Wynik propagacji w przód.
- **Metoda: `backward(self, gradient, layer_input)`**
    - **Opis**: Przeprowadza propagację wstecz.
    - **Parametry**:
        * `gradient` (numpy array): Gradient z następnej warstwy.
        * `layer_input` (numpy array): Dane wejściowe do warstwy podczas propagacji w przód.
    - **Zwraca**: Gradient dla poprzedniej warstwy.

Klasa: `LeakyRelu`

Klasa `LeakyRelu` implementuje funkcję aktywacji Leaky ReLU.

- **Konstruktor: `__init__(self, a=0.01)`**
    - **Parametry**:
        * `a` (float): Wartość nachylenia dla ujemnych wartości wejściowych (domyślnie 0.01).
- **Metoda: `forward(self, layer_input)`**
    - **Opis**: Przeprowadza propagację w przód.
    - **Parametry**:
        * `layer_input` (numpy array): Dane wejściowe.

  - **Zwraca**: Wynik propagacji w przód.
- **Metoda: `backward(self, gradient, layer_input)`**
  - **Opis**: Przeprowadza propagację wstecz.
  - **Parametry**:
    - `gradient` (numpy array): Gradient z następnej warstwy.
    - `layer_input` (numpy array): Dane wejściowe do warstwy podczas propagacji w przód.
  - **Zwraca**: Gradient dla poprzedniej warstwy.

Przykład użycia

```
import numpy as np
from ActivationFunctions import Sigmoid, Tanh, Relu, LeakyRelu

# Przykładowe dane wejściowe
x = np.random.rand(10, 3)

# Funkcja aktywacji Sigmoid
sigmoid = Sigmoid()
sigmoid_output = sigmoid.forward(x)
print("Sigmoid output:", sigmoid_output)

# Funkcja aktywacji Tanh
tanh = Tanh()
tanh_output = tanh.forward(x)
print("Tanh output:", tanh_output)

# Funkcja aktywacji ReLU
relu = Relu()
relu_output = relu.forward(x)
print("ReLU output:", relu_output)

# Funkcja aktywacji Leaky ReLU
leaky_relu = LeakyRelu(a=0.1)
leaky_relu_output = leaky_relu.forward(x)
print("Leaky ReLU output:", leaky_relu_output)
```

Dropout

Klasa: `Dropout`

Klasa `Dropout` implementuje warstwę dropout do sieci, która pomaga w zapobieganiu przeuczeniu.

- **Konstruktor: `__init__(self, input_neurons, fraction, training)`**
  - **Parametry**:
    - `input_neurons` (int): Liczba neuronów wejściowych.
    - `fraction` (float): Ułamek jednostek do wyłączenia (domyślnie 0.5).
    - `training` (lista bool): Lista zawierająca wartość określającą tryb trenowania (True) lub ewaluacji (False).
- **Metoda: `forward(self, layer_input)`**
  - **Opis**: Przeprowadza propagację w przód.
  - **Parametry**:
    - `layer_input` (numpy array): Dane wejściowe.
  - **Zwraca**: Wynik propagacji w przód po zastosowaniu dropout.
- **Metoda: `backward(self, gradient, *_)`**
  - **Opis**: Przeprowadza propagację wstecz.
  - **Parametry**:
    - `gradient` (numpy array): Gradient z następnej warstwy.
  - **Zwraca**: Gradient dla poprzedniej warstwy po zastosowaniu dropout.

Przykład użycia

```
import numpy as np
from Dropout import Dropout
```

```python
# Przykładowe dane wejściowe
x = np.random.rand(10, 3)

# Inicjalizacja Dropout
dropout = Dropout(input_neurons=3, fraction=0.5, training=[True])

# Propagacja w przód
dropout_output = dropout.forward(x)
print("Dropout output (training):", dropout_output)

# Tryb ewaluacyjny
dropout.training[0] = False
dropout_output_eval = dropout.forward(x)
print("Dropout output (evaluation):", dropout_output_eval)
```

Funkcje Straty

Klasa: `MSELoss`

Klasa `MSELoss` implementuje funkcję straty Mean Squared Error (MSE).

- **Metoda:** `loss(self, y_pred, y_true)`
  - **Opis**: Oblicza stratę MSE.
  - **Parametry**:
    - `y_pred` (numpy array): Przewidywane wartości.
    - `y_true` (numpy array): Prawdziwe wartości.
  - **Zwraca**: Wartość straty MSE.
- **Metoda:** `loss_gradient(self, y_pred, y_true)`
  - **Opis**: Oblicza gradient straty MSE.
  - **Parametry**:
    - `y_pred` (numpy array): Przewidywane wartości.
    - `y_true` (numpy array): Prawdziwe wartości.
  - **Zwraca**: Gradient straty MSE.

Klasa: `MAELoss`

Klasa `MAELoss` implementuje funkcję straty Mean Absolute Error (MAE).

- **Metoda:** `loss(self, y_pred, y_true)`
  - **Opis**: Oblicza stratę MAE.
  - **Parametry**:
    - `y_pred` (numpy array): Przewidywane wartości.
    - `y_true` (numpy array): Prawdziwe wartości.
  - **Zwraca**: Wartość straty MAE.
- **Metoda:** `loss_gradient(self, y_pred, y_true)`
  - **Opis**: Oblicza gradient straty MAE.
  - **Parametry**:
    - `y_pred` (numpy array): Przewidywane wartości.
    - `y_true` (numpy array): Prawdziwe wartości.
  - **Zwraca**: Gradient straty MAE.

Klasa: `BCELoss`

Klasa `BCELoss` implementuje funkcję straty Binary Cross-Entropy (BCE).

- **Metoda:** `loss(self, y_pred, y_true)`
  - **Opis**: Oblicza stratę BCE.
  - **Parametry**:
    - `y_pred` (numpy array): Przewidywane wartości.
    - `y_true` (numpy array): Prawdziwe wartości.
  - **Zwraca**: Wartość straty BCE.
- **Metoda:** `loss_gradient(self, y_pred, y_true)`

  - **Opis**: Oblicza gradient straty BCE.
  - **Parametry**:
    * `y_pred` (numpy array): Przewidywane wartości.
    * `y_true` (numpy array): Prawdziwe wartości.
  - **Zwraca**: Gradient straty BCE.

Przykład użycia

```
import numpy as np
from LossFunctions import MSELoss, MAELoss, BCELoss

# Przykładowe przewidywane i prawdziwe wartości
y_pred = np.random.rand(10, 1)
y_true = np.random.rand(10, 1)

# Funkcja straty MSE
mse_loss = MSELoss()
mse_value = mse_loss.loss(y_pred, y_true)
mse_grad = mse_loss.loss_gradient(y_pred, y_true)
print("MSE Loss value:", mse_value)
print("MSE Loss gradient:", mse_grad)

# Funkcja straty MAE
mae_loss = MAELoss()
mae_value = mae_loss.loss(y_pred, y_true)
mae_grad = mae_loss.loss_gradient(y_pred, y_true)
print("MAE Loss value:", mae_value)
print("MAE Loss gradient:", mae_grad)

# Funkcja straty BCE
bce_loss = BCELoss()
bce_value = bce_loss.loss(y_pred, y_true)
bce_grad = bce_loss.loss_gradient(y_pred, y_true)
print("BCE Loss value:", bce_value)
print("BCE Loss gradient:", bce_grad)
```

Optymalizatory

Klasa: `SGD`

Klasa `SGD` implementuje optymalizator Stochastic Gradient Descent (SGD).

- **Konstruktor: `__init__(self, learning_rate)`**
  - **Parametry**:
    * `learning_rate` (float): Współczynnik uczenia.
- **Metoda: `optimize(self, weights, biases, grad_weights, grad_biases)`**
  - **Opis**: Aktualizuje wagi i biasy za pomocą algorytmu SGD.
  - **Parametry**:
    * `weights` (numpy array): Aktualne wagi.
    * `biases` (numpy array): Aktualne biasy.
    * `grad_weights` (numpy array): Gradient wag.
    * `grad_biases` (numpy array): Gradient biasów.
  - **Zwraca**: Zaktualizowane wagi i biasy.

Klasa: `SGDMomentum`

Klasa `SGDMomentum` implementuje optymalizator SGD z momentum.

- **Konstruktor: `__init__(self, learning_rate, momentum)`**
  - **Parametry**:
    * `learning_rate` (float): Współczynnik uczenia.
    * `momentum` (float): Współczynnik momentum.

- **Metoda:** `optimize(self, weights, biases, grad_weights, grad_biases)`
  - **Opis**: Aktualizuje wagi i biasy za pomocą algorytmu SGD z momentum.
  - **Parametry**:
    - `weights` (numpy array): Aktualne wagi.
    - `biases` (numpy array): Aktualne biasy.
    - `grad_weights` (numpy array): Gradient wag.
    - `grad_biases` (numpy array): Gradient biasów.
  - **Zwraca**: Zaktualizowane wagi i biasy.

Przykład użycia

```
import numpy as np
from Optimizers import SGD

# Przykładowe wagi i biasy
weights = np.random.rand(3, 2)
biases = np.random.rand(2)
grad_weights = np.random.rand(3, 2)
grad_biases = np.random.rand(2)

# Optymalizator SGD
optimizer_sgd = SGD(learning_rate=0.01)
new_weights, new_biases = optimizer_sgd.optimize(weights, biases, grad_weights, grad_biases)
print("SGD updated weights:", new_weights)
print("SGD updated biases:", new_biases)

# Optymalizator SGD z momentum
optimizer_sgdm = SGD(learning_rate=0.01, momentum=0.9)
new_weights_m, new_biases_m = optimizer_sgdm.optimize(weights, biases, grad_weights, grad_biases)
print("SGDM updated weights:", new_weights_m)
print("SGDM updated biases:", new_biases_m)
```