Perceptron

- Definicja

$$w_i^{(nowe)} = w_i^{(stare)} + \eta \cdot (y - \hat{y}) \cdot x_i$$

- Wzory
  - Aktualizacja wag
  - $w_i^{(nowe)}$ - nowa waga dla i-tego wejścia
  - $w_i^{(stare)}$ - poprzednia waga dla i-tego wejścia
  - η - współczynnik uczenia
  - y - rzeczywista etykieta klasy dla danego przykładu
  - $\hat{y}$ - predykcja modelu dla danego przykładu
  - x_i - i-ty element wektora danych wejściowych

Funkcja skokowa

Funkcja skokowa (znana również jako funkcja Heaviside'a) to prosta funkcja aktywacji używana w wielu modelach neuronowych. Jest to funkcja, która zwraca 1, gdy argument jest większy lub równy zeru, a w przeciwnym przypadku zwraca 0.

$$f(x) = \begin{cases} 1, & \text{jeśli } x \geq 0 \\ 0, & \text{w przeciwnym przypadku} \end{cases}$$

Przykład działania - Reguła uczenia

- Dane Wejściowe: (x_1, x_2)
- Wagi: (w_1, w_2)
- Bias: b
- Funkcja Aktywacji: funkcja skokowa

1. Iteracja 1:
   - Obliczenie ważonej sumy
   - Aktywacja
   - Predykcja
   - Aktualizacja wag
   - Aktualizacja bias
2. Iteracja 2 (dla kolejnego przykładu):
   - Analogicznie do iteracji 1

Funkcja skokowa - Wprowadzenie

Funkcja skokowa (Heaviside'a) jest jedną z podstawowych funkcji aktywacji w modelach neuronowych.

Funkcja skokowa - Definicja

Funkcja skokowa (Heaviside'a) jest zdefiniowana jako:

$$f(x) = \begin{cases} 1, & \text{jeśli } x \geq 0 \\ 0, & \text{w przeciwnym przypadku} \end{cases}$$

Funkcja Skokowa w Pythonie

Implementacje w Pythonie

```
import numpy as np

def heaviside(x):
    return np.heaviside(x, 0)

# Przykłady użycia
print(heaviside(2))  # Output: 1.0
print(heaviside(-3)) # Output: 0.0
```

Reguła Uczenia Perceptronu w Pythonie

Implementacje w Pythonie

```
class Perceptron:
    def __init__(self, num_inputs, learning_rate=0.1):
        self.num_inputs = num_inputs
        self.weights = np.zeros(num_inputs)
        self.bias = 0
        self.learning_rate = learning_rate

    def predict(self, inputs):
        weighted_sum = np.dot(self.weights, inputs) + self.bias
        return heaviside(weighted_sum)

    def train(self, training_inputs, labels):
        for inputs, label in zip(training_inputs, labels):
            prediction = self.predict(inputs)
            error = label - prediction
            self.weights += self.learning_rate * error * inputs
            self.bias += self.learning_rate * error
```

Reguła Uczenia Perceptronu w Pythonie

Implementacje w Pythonie

```
# Przykłady użycia
perceptron = Perceptron(2)
training_inputs = np.array([(2, 3), (-1, 2)])
labels = np.array([1, 0])
perceptron.train(training_inputs, labels)

# Testowanie na nowych danych
print(perceptron.predict((2, 3)))  # Output: 1.0
print(perceptron.predict((-1, 2))) # Output: 0.0
```

Funkcja Sigmoidalna

Wzór:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Zalety:

- Zakres wartości między 0 a 1, co jest przydatne w problemach klasyfikacji.
- Gładka i różniczkowalna, ułatwia propagację wsteczną.
- Reprezentuje prawdopodobieństwo, co jest użyteczne w problemach binarnej klasyfikacji.

Wady:

- Problem z zanikającymi gradientami w głębokich sieciach neuronowych.

Implementacje w Pythonie

```
import numpy as np
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Funkcja ReLU (Rectified Linear Unit)

Wzór:

$$f(x) = \max(0, x)$$

Zalety:

- Prosta w implementacji i obliczeniowo efektywna.
- Unikanie problemu zanikających gradientów.
- Szybkość zbieżności w uczeniu głębokich sieci neuronowych.

Wady:

- Niezerowa dla ujemnych argumentów, co może prowadzić do zjawiska zanikania neuronów.
- Brak różniczkowalności dla x=0, co może stanowić problem w niektórych metodach optymalizacyjnych.

Implementacje w Pythonie

```
def relu(x):
    return max(0, x)
```

Funkcja Tangens Hiperboliczny (tanh)

Wzór:

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Zalety:

- Zakres wartości między -1 a 1, co jest przydatne w problemach klasyfikacji.
- Szeroki i różnorodny zakres dynamiki, co może prowadzić do szybszej zbieżności w uczeniu.
- Gładka i różniczkowalna, co ułatwia propagację wsteczną.

Wady:

- Problem z zanikającymi gradientami dla wartości dalekich od zera.
- Może być mniej efektywna w uczeniu głębokich sieci neuronowych niż ReLU.

Implementacje w Pythonie

```
import numpy as np
def tanh(x):
    return np.tanh(x)
```

Implementacja Funkcji Aktywacji w Pythonie

Funkcja Skokowa (Heaviside'a):

Implementacje w Pythonie

```
def heaviside(x):
    if x >= 0:
        return 1
    else:
        return 0
```

Funkcja Sigmoidalna:

Implementacje w Pythonie

```
def sigmoid(x):
    return 1 / (1 + 2.71828 ** (-x))
```

Implementacja Funkcji Aktywacji w Pythonie

Funkcja ReLU (Rectified Linear Unit):

Implementacje w Pythonie

```
def relu(x):
    if x >= 0:
        return x
    else:
        return 0
```

Funkcja Tangens Hiperboliczny (tanh):

Implementacje w Pythonie

```
def tanh(x):
    return (2 / (1 + 2.71828 ** (-2 * x))) - 1
```

Neuron sigmoidalny

Definicja

Neuron sigmoidalny to rodzaj neuronu używanego w sztucznych sieciach neuronowych, który wykorzystuje funkcję aktywacji sigmoidalną do transformacji sygnałów wejściowych.

- Neuron sigmoidalny przyjmuje wejścia $x_1, x_2, ..., x_n$, każde z nich jest ważone przez odpowiednie wagi $w_1, w_2, ..., w_n$.
- Następnie suma ważona wejść jest przekształcana przez funkcję aktywacji sigmoidalną σ, aby uzyskać wyjście y.

Neuron sigmoidalny

Wzór dla wyjścia neuronu sigmoidalnego

$$y = \sigma\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right)$$

- x_i - wartość i-tego wejścia,
- w_i - waga przypisana do i-tego wejścia,
- b - bias (przesunięcie),
- σ - funkcja aktywacji sigmoidalna.

Reguła uczenia neuronu sigmoidalnego

Funkcja kosztu (błędu)

Do oceny jakości predykcji neuronu w stosunku do prawdziwych etykiet używamy funkcji kosztu. Przykładowe funkcje kosztu to błąd średniokwadratowy (MSE) lub entropia krzyżowa (cross-entropy).

- **Błąd średniokwadratowy (MSE)**:

```
def mean_squared_error(y_true, y_pred):
    return np.mean((y_true - y_pred)**2)
```

- **Entropia krzyżowa**:

```
def cross_entropy_loss(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(y_true * np.log(y_pred) +
                    (1 - y_true) * np.log(1 - y_pred))
```

Algorytm propagacji wstecznej

Algorytm

1. Oblicz błąd: $error = y_{\text{pred}} - y_{\text{true}}$
2. Oblicz gradient: $\nabla J = X^T \cdot error$
3. Aktualizuj wagi: $w = w - \alpha \cdot \nabla J$

Implementacje w Pythonie

```
def backpropagation(X, y_true, y_pred, weights, learning_rate):
    error = y_pred - y_true
    gradient = np.dot(X.T, error)
    weights -= learning_rate * gradient
    return weights
```

Funkcja aktywacji sigmoidalna

Funkcja sigmoidalna jest często używana jako funkcja aktywacji w neuronach sztucznych sieci neuronowych. Jest to funkcja nieliniowa, która przekształca sumę ważoną wejść na wartość w zakresie od 0 do 1.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Implementacje w Pythonie

```
import numpy as np

def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

Przykład obliczeń oraz implementacje w Pythonie

Przykład obliczeń

```
# Inicjalizacja wag
weights = np.random.rand(3)

# Przekazanie sygnału
z = np.dot(X, weights)

# Funkcja aktywacji
y_pred = sigmoid(z)

# Obliczenie błędu
cost = mean_squared_error(y_true, y_pred)

# Propagacja wsteczna
weights = backpropagation(X, y_true, y_pred, weights, learning_rate)
```

Przykład obliczeń oraz implementacje w Pythonie

Implementacje w Pythonie

```
import numpy as np

def mean_squared_error(y_true, y_pred):
    return np.mean((y_true - y_pred)**2)

def cross_entropy_loss(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true)
    *
    np.log(1 - y_pred))

def backpropagation(X, y_true, y_pred, weights, learning_rate):
    error = y_pred - y_true
    gradient = np.dot(X.T, error)
    weights -= learning_rate * gradient
    return weights
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

Uczenie neuronu sigmoidalnego

Implementacje w Pythonie

```
import numpy as np
class SigmoidNeuron:
    def __init__(self, input_size):
        self.weights = np.random.uniform(-0.1, 0.1, size=input_size)
        self.bias = np.random.uniform(-0.1, 0.1)

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

    def feedforward(self, inputs):
        z = np.dot(inputs, self.weights) + self.bias
        return self.sigmoid(z)
```

Uczenie neuronu sigmoidalnego

Implementacje w Pythonie

```
def backpropagation(self, inputs, target, learning_rate):
    z = np.dot(inputs, self.weights) + self.bias
    predicted = self.sigmoid(z)
    error = predicted - target
    dC_dw = np.dot(inputs.T, error)
    dC_db = np.sum(error)
    self.weights -= learning_rate * dC_dw
    self.bias -= learning_rate * dC_db

def train(self, X, y, epochs, learning_rate):
    for epoch in range(epochs):
        total_loss = 0
        for inputs, target in zip(X, y):
            inputs = np.array(inputs, ndmin=2)
            self.backpropagation(inputs, target, learning_rate)
            total_loss += self.loss(inputs, target)
        if (epoch + 1) % 100 == 0:
            print(f"Epoka {epoch + }/{epochs},
            całkowita strata: {total_loss}")
```

Uczenie neuronu sigmoidalnego

Implementacje w Pythonie

```
    def loss(self, inputs, target):
        predicted = self.feedforward(inputs)
        return 0.5 * np.square(predicted - target)

# Przykładowe dane treningowe
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

# Normalizacja danych wejściowych
X_normalized = X / np.max(X)
```

Uczenie neuronu sigmoidalnego

Implementacje w Pythonie

```
# Utworzenie instancji neuronu
neuron = SigmoidNeuron(input_size=2)

# Uczenie neuronu
neuron.train(X_normalized, y, epochs=100, learning_rate=0.1)

# Testowanie na nowych danych
new_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
new_data_normalized = new_data / np.max(new_data)
expected_values = np.array([0, 0, 0, 1])
for data, target in zip(new_data_normalized, expected_values):
    prediction = neuron.feedforward(data)
    print(f"Dla danych {data}, oczekiwana wartość:
    {target}, predykcja: {prediction}")
```