Alan Vu Sipiorski, 225810

Michał Respondek, 225784

Tomasz Budny, 208246

Dokumentacja projektu – Sieci Neuronowe

Perceptron

Najprostszy i najbardziej podstawowy, pojedyńczy matematyczny model sztucznego neuronu. Neuron rozumiemy jako podstawowa jednostka systemu nerwowego, np. człowieka, komórka nerwowa, która przetwarza i przewodzi informacje w postaci sygnałów elektrycznych. Jest on używany najczęściej do klasyfikacji binarnej elementów, które rozdzielone są granicą decyzyjną, jednak te obszary muszą być liniowo separowalne.

Perceptron składa się z:

- wejścia: $x_1, x_2, x_3, x_4 \dots, x_n$
- wag: : $w_1, w_2, w_3, w_4 \dots, w_n$
- stałej uczenia
- funkcji aktywacji
- wyjścia

Celem klasyfikacji używając perceptronu, jest znalezienie takich wag, aby wartości wyjściowe były odpowiednio takie same jak wartości oczekiwane.

Perceptron Algorithm

PA - Perceptron Algorithm, klasyczny i podstawowy algorytm perceptronu, który uczy się na podstawie iteracyjnego dobierania nowych wag.

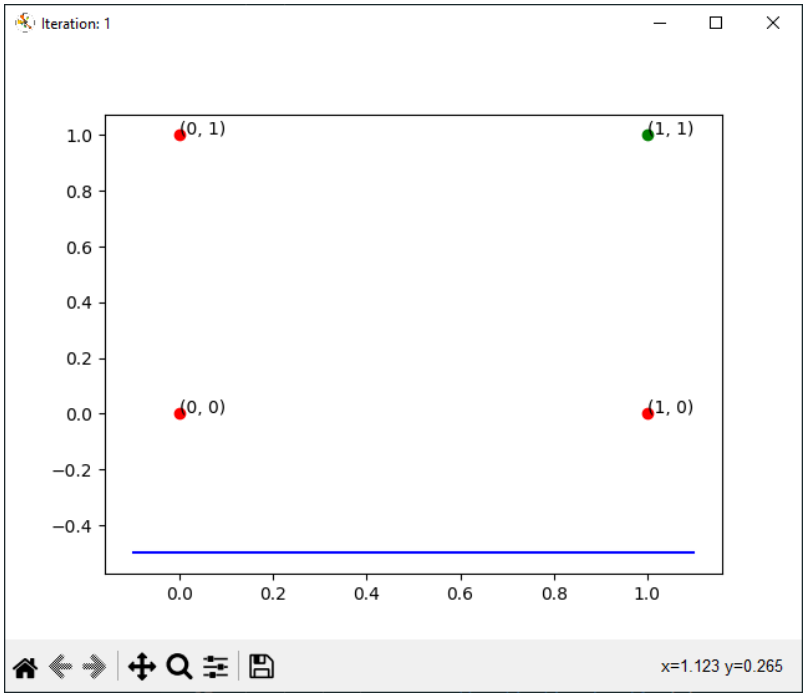
- Wagi w^1 mogą być wybrane losowo
- $w^1 = (w_1, w_2, w_3, w_4 \dots, w_n)$
- $w^{k+1} = w^k + \varrho\,(d^k - y^k)\,x^k$
- $\boldsymbol{\varphi} = (x^k)^T w^k$
- $y^k = g(\varphi)$ – funkcja aktywacji

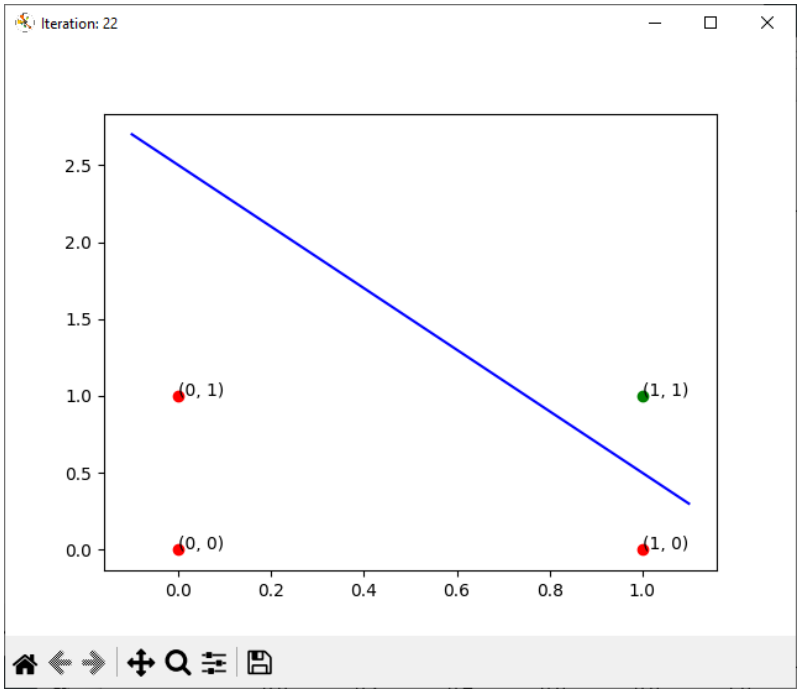
Przykładowa funkcja aktywacji:

- $g(x) = \begin{cases} 1\ jeżeli\ x > 0 \\ 0\ jeżeli\ x \leq 0 \end{cases}$

Wygenerowany wykres (PA)

Z uwzględnieniem linii decyzyjnej oraz położenie wektorów wejściowych. Wykresy w iteracji 1 oraz 22.





Batch Update Perceptron Algorithm

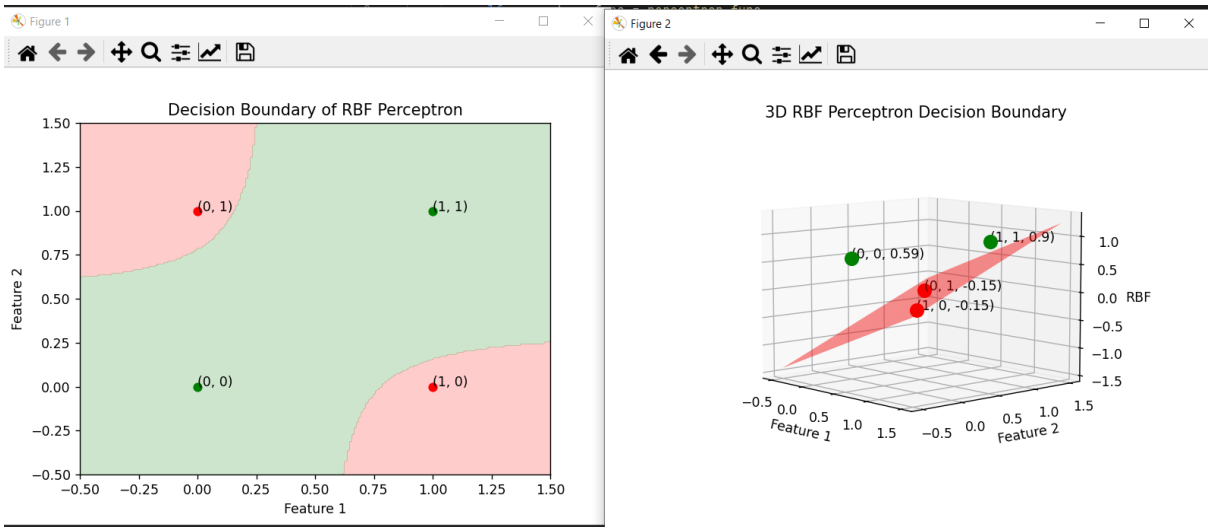
BUPA - Batch Update Perceptron Algorithm, grupowo odświeżany algorytm perceptronu. Różni się od klasycznej wariacji algorytmu tym, że zmiana wag następuje w inny sposób.

- Wagi w^1 mogą być wybrane losowo
- $w^1 = (w_1, w_2, w_3, w_4 \dots, w_n)$
- $w^{k+1} = w^k + \varrho \, \Sigma \, Z_i$

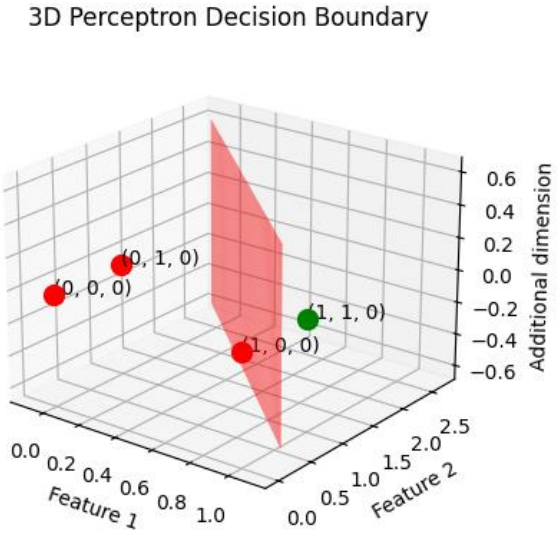
Funkcja XOR

W porównaniu do funkcji AND, której elementy są liniowo separowalne, z funkcją xor jest większy problem, ponieważ nie jest on liniowo separowalny. W związku z tym, podstawowe obliczenia perceptronu nie są wystarczające do rozwiązania tego problemu.

Przykładowym rozwiązaniem tego problemu jest na przykład podniesienie zbióru o jeden wymiar wyżej, z zastosowaniem kernelu RBF (radialna funkcja bazowej RBF).



Wyświetlanie granicy decyzyjnej w przestrzeni $\mathbb{R}^3$



Sieć Hopfielda

Jest to sieć neuronowa, która tak samo jak perceptron posiada pewne neurony, one natomiast są połączone w obu kierunkach oraz symetryczne, w sposób rekurencyjny. To oznacza, że połączenia między dwoma neuronami mają taką samą wagę, a każdy neuron jest połączony z każdym innym neuronem. Wzajemne połączenia neuronów wraz z wagami, są zobrazowane w postaci macierzy.

Ten rodzaj sieci przechowuje i zapamiętuje pewne n-wymiarowe binarne wektory, które w późniejszych fazach są używane do rozpoznawania i odtwarzania pewnych wzorców. Jest to związane z pojęciem nazywanym pamięcią asocjacyjną. Proces zapamiętywania wzorców polega na wykorzystaniu reguły Hebba do odpowiedniego zmieniania wag. Sieci Hopfielda są na przykład używane do procesów związanych z rozpoznawaniem poszczególnych literek alfabetu, oczywiście będąc wcześniej nauczonym. Celem jest doprowadzenie do stabilności sieci lub jej zbieżności.

Poszczególnymi krokami procesu charakterystycznego dla tej sieci jest:

- Inicjalizacja stanu neuronów
- Aktualizacja tych stanów
- Powtórzenie powyższych aż do stabilności

Tryb synchroniczny

W trybie synchronicznym, modyfikujemy wszystkie neurony na raz, w tym samym czasie.

```python
def synchronous_mode(x, w, f, sig):
    x_prime = x.copy()

    for i in range(len(x)):
        x_prime[i] = 0. + sig
        for j in range(len(x)):
            x_prime[i] += w[i][j] * x[j]
        x_prime[i] = f(x_prime[i])

    x = x_prime

    return x
```

Warunki stabilizacji w trybie synchronicznym

- Macierz wag jest symetryczna:

$$w_{i,j} = w_{j,i}$$

- Na diagonali są elementy większe lub równe 0

$$w_{i_1 j} \geq 0$$

- Macierz jest dodatnio określona

Tryb asynchroniczny

W trybie asynchronicznym neurony są aktualizowane niezależnie od siebie, nie w tym samym czasie, kolejne po sobie. Możliwy jest losowy lub cykliczny wybór następnych neuronów do aktualizacji.

```python
def asynchronous_mode(x, w, f, sig):
    for i in range(len(x)):
        x_prime = 0. + sig
        for j in range(len(x)):
            x_prime += w[i][j] * x[j]
        x_prime = f(x_prime)
        x[i] = x_prime

    return x
```

Warunki stabilizacji w trybie asynchronicznym

- Macierz wag jest symetryczna:

$$w_{i,j} = w_{j,i}$$

- Na diagonali są elementy większe lub równe 0

$$w_{i_1 j} \geq 0$$

Algorytm Propagacji Wstecznej

Posiadając dużą sieć neuronową z wieloma warstwami, używając algorytmu propagacji wstecznej jesteśmy w stanie modyfikować wagi we wszystkich jej warstwach. Korzystając z tego algorytmu cofamy się do tyłu, warstwa po warstwie, tak jak to nazwa wskazuje, dochodząc do wybranej przez nas warstwy, aby zmienić jej wagę. Elementami sieci używającej opisywany algorytm jest:

- Warstwa wejściowa
- Warstwa ukryta
- Warstwa wyjścia

Poszczególnymi krokami algorytmu są:

- Inicjalizacja wag
- Propagacja wprzód (z ang. forward propagation)
- Obliczanie błędu
- Propagacja wstecz (z ang. backward propagation)
- Aktualizacja wag

Warstwa ukryta posiadająca dwa neurony pozwala na rozwiązanie problemu XOR, co byłoby niemożliwe posiadając tylko jeden neuron.

Energia całkowita

Korzystając z energii całkowitej, wagi są aktualizowane po prezentacji wszystkich wektorów wejściowych. Wtedy posiadamy wszystkie błędy, na podstawie których możemy obliczyć ich sumę.

```
W2, W3 = initial_weights()

for epoch in range(epochs):
    # forward propagation
    X2 = sigmoid(np.dot(X1, W2))
    X2 = np.hstack((np.ones((4, 1)), X2))  #add dummy input
    X3 = sigmoid(np.dot(X2, W3))

    # loss
    loss = np.mean(0.5 * (Y - X3) ** 2)
    loss_values.append(loss)

    # backward propagation
    delta3 = (Y - X3) * sigmoid_derivative(X3)
    delta2 = delta3.dot(W3.T) * sigmoid_derivative(X2)

    # weights
    W3 += X2.T.dot(delta3) * learning_rate
    W2 += (X1.T.dot(delta2))[:, 1:] * learning_rate
```

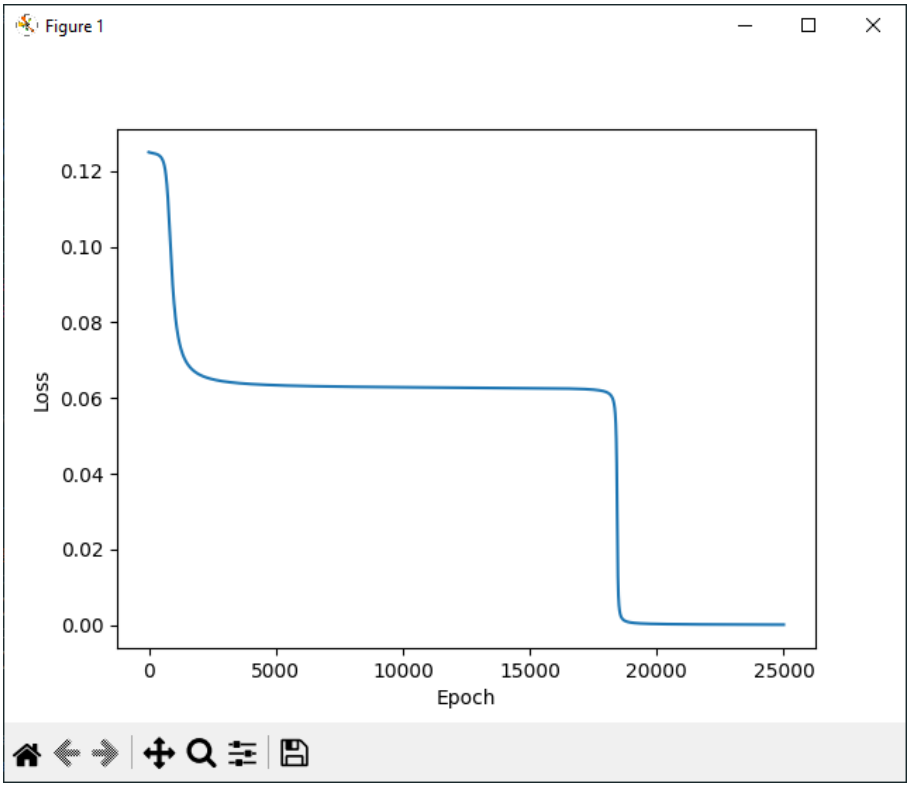


Figure 1
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Loss
0
5000
10000
15000
20000
25000
Epoch

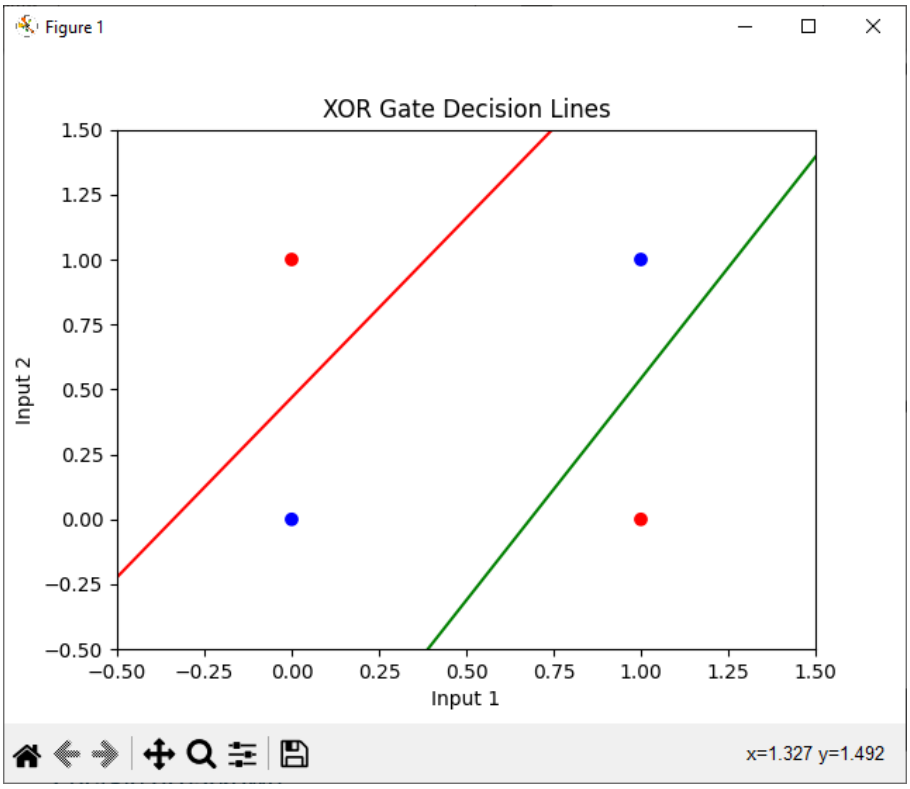


Figure 1
XOR Gate Decision Lines
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
Input 2
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
1.25
1.50
Input 1
x=1.327 y=1.492

Energia cząstkowa

W przypadku korzystania z energii cząstkowej, wagi są aktualizowane po każdym pojedynczym prezentowanym wektorze wejściowym. W porównaniu do trybu energii całkowitej, nie sumujemy wszystkich błędów, a wystarczy nam jeden pojedynczy.

```python
W2, W3 = initial_weights()

loss_values = []

for epoch in range(epochs):
    loss = 0.

    for i in range(len(X1)):
        # forward propagation
        X2 = sigmoid(np.dot(X1, W2))
        X2 = np.hstack((np.ones((4, 1)), X2))  #add dummy input
        X3 = sigmoid(np.dot(X2, W3))

        # loss
        loss += (Y[i] - X3[i]) ** 2

        # backward propagation
        delta3 = (Y[i] - X3[i]) * sigmoid_derivative(X3)
        delta2 = delta3.dot(W3.T) * sigmoid_derivative(X2)

        # weights
        W3 += X2.T.dot(delta3) * learning_rate
        W2 += (X1.T.dot(delta2))[:, 1:] * learning_rate

    loss_values.append(loss / len(X1))
```