Perceptron

Weronika Smagór, Aleksandra Motor, Patryk Polczyk, Krzysztof Konieczny
Fizyka Techniczna III rok
Wydział Inżynierii Materiałowej i Fizyki Politechniki Krakowskiej

Czerwiec 2021
Kraków

Spis treści

1 Wstęp

Warrech McCulloch oraz Walter Pitts, chcąc zrozumieć mechanizm mózgu, zaprezentowali koncepcję uproszczonego modelu komórki nerwowej, tzw. neuronu McCullocha-Pittsa. Opisali komórkę nerwową jako prostą bramkę logiczną, która zawiera binarne wyjścia. Kilka lat później Frank Rosenblatt na podstawie modelu neuronu MCP opublikował pierwszą koncepcję reguły uczenia perceptronu. Zapronował algorytm zdolny do automatycznego uczenia się za pomocą optymalnych współczynników wag, które są przemnażane przez wartości wejściowe. Taki proces pozwala określić czy neuron prześle sygnał dalej.
Podstawowym założeniem w neuronie MCP i modelu perceptronu progowego jest wprowadzenie uproszczonego mechanizmu naśladującego działanie pojedynczej komórki nerwowej. Reguła Rosenblatta jest opisywana poszczególnymi etapami:

- Wprowadzenie wag o wartości 0 lub niewielkich, losowych wartościach
- Dla każdej próbki należy obliczyć wartość wyjściową oraz zaktualizować wagi

Perceptron to najprostsza forma sieci neutronowej stosowana do klasyfikacji specjalnych typów wzorców, które są liniowo rozłączne. Składa się z pojedynczego neuronu z regulowanymi wagami synaptycznymi w_i oraz progiem θ.
Jeżeli całkowite pobudzenie z danej próbki $x^{(i)}$ jest wyższe od zdefiniowanej wartości progowej θ, to przewidujemy, że dany obiekt przynalezy do klasy pozytywnej 1, a w przeciwnym razie do klasy negatywnej -1. W algorytmie perceptronu funkcja aktywacji $\Phi(z)$ jest funkcją skoku jednostkowego, zwaną również funkcją skokową Heaviside'a.

$$\Phi(z) = \begin{cases} 1 & \text{gdy } z \geqslant 0 \\ -1 & \text{gdy } z < 0 \end{cases}$$

Możemy dla uproszczenia przenieść wartość progową θ na lewą stronę równania i zdefiniować początkową wagę jako $w_0 = \theta$, a $x_0 = 1$, dzięki czemu całkowite pobudzenie z przybierze prostszą postać $z = w_0x_0 + w_1x_1, ..., w_nx_n = w^Tx$ zakładając, że

$$\Phi(z) = \begin{cases} 1 & \text{gdy } z \geqslant 0 \\ -1 & \text{gdy } z < 0 \end{cases}$$

Przykład:
Samolot: $x_1 + 2x_23x_3 = 4$ który, dzieli R^3 na dwie półprzestrzenie.
W wielu przypadkach wygodniej jest zajmować się tylko perceptronami o progu równym zero. Odpowiada to separacjom liniowym, które są zmuszone do przejścia przez początek przestrzeni wejściowej. Próg perceptora został zamieniony na wagę θ dodatkowego kanału wejściowego połączonego ze stałą 1. Ta dodatkowa waga połączona ze stałą nazywa się obciążeniem elementu. Zatem wektor wejściowy $(x_1, x_2, ..., x_n)$ musi być rozszerzony o dodatkową 1 i w rezultacie otrzymamy $(n + 1)$-wymiarowy wektor:

$$(1, x_1, x_2, ..., x_n)$$

Wektor ten nazywa się rozszerzonym wektorem wejściowym, gdzie $x_0 = 1$. Rozszerzony wektor wagowy związany z tym perceptronem to:

$$(w_0, w_1, ..., w_n)$$

w wyniku czego: $w_0 = \theta$
Obliczenie progowe perceptronu zostanie wyrażone za pomocą iloczynów skalarnych. Zatem test arytmetyczny obliczony przez perceptron wyraża się następująco:

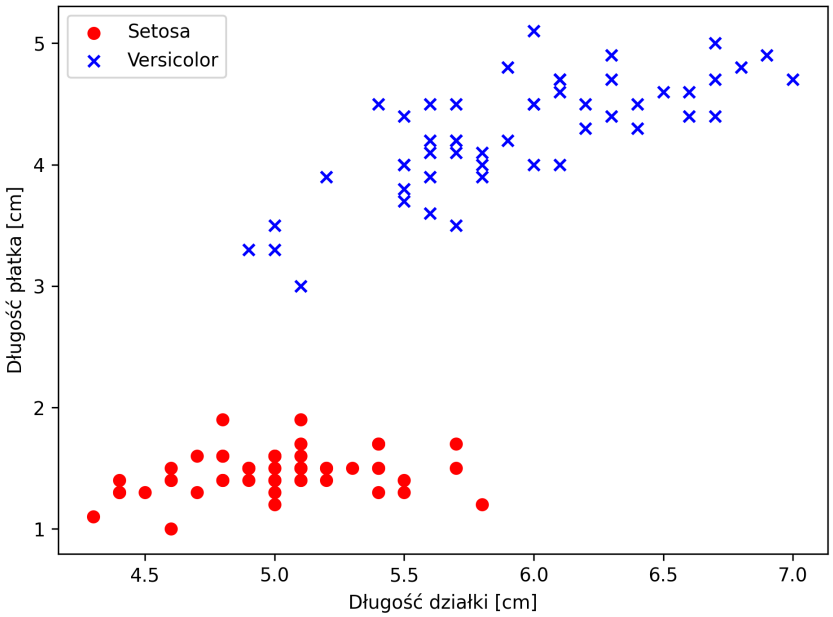
$$w^tx \geqslant \theta$$

wtedy, gdy w i x są wagami i wektorami wejściowymi, a gdy w i x są rozszerzoną wagą i wektorami wejściowymi możemy zapisać to następująco:

$$w^tx \leqslant \theta$$

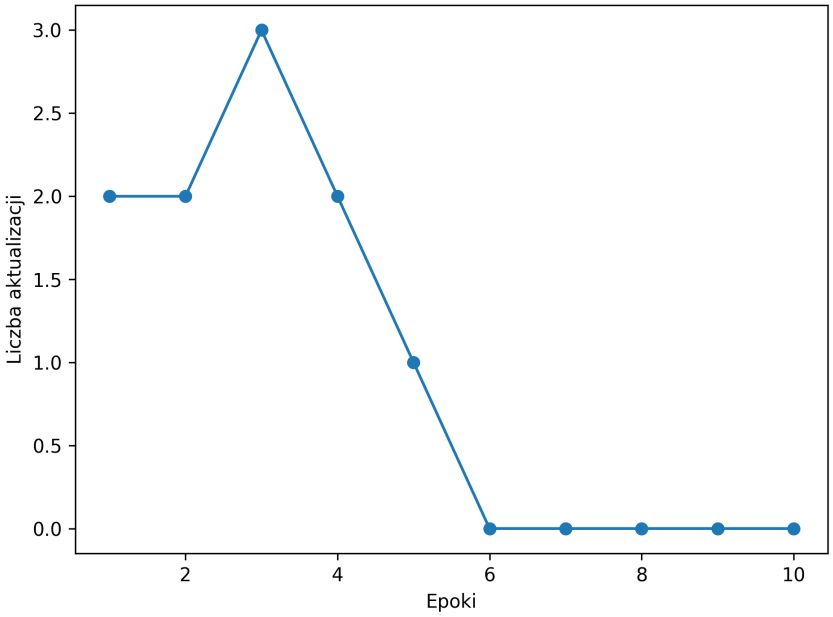
2 Nauka perceptronu

Gdy zaimplementowaliśmy już nasz perceptron, następnie musimy dostarczyć mu dane do nauki. W naszym przypadku, będą to standardowe dane na temat irysów, które pobraliśmy ze strony:
https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
Dane te posiadają 4 parametry, jednak do uczenia perceptronu wykorzystamy tylko dwa. Będą to mianowicie długość działki oraz wielkość płatka. Pobraliśmy dane na temat 50 kwiatów setosa i 50 kwiatów versicolor. Nadamy im odpowiednio wartości -1 i 1. Algorytm oddziela dane prostą i zbiega się do lokalnego minimum, aby znaleźć odpowiednią wagę.



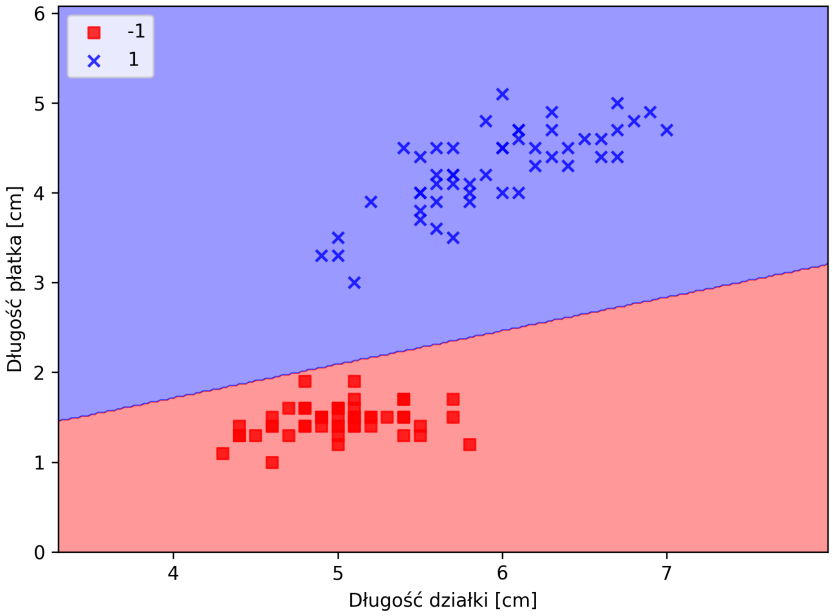
Rysunek 1: Zestawienie danych uczących

W następnym kroku przyjrzymy się błędom klasyfikacji dla każdej iteracji(epoki).



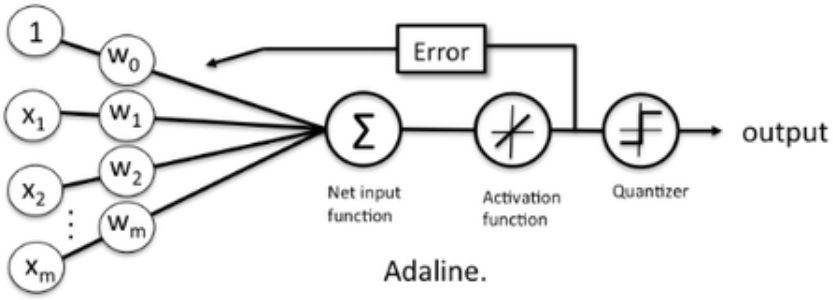
Rysunek 2: Wykres zbieżności algorytmu

Z wykresu wynika, że algorytm znalazł granicę decyzyjną już w 6 epoce i nauczył się poprawnie klasyfikować kwiaty. Rosenblatt dowiódł matematycznie, że reguła uczenia się perceptronu wykazuje zbieżność, jeśli dwie klasy mogą zostać rozdzielone liniową hiperpłaszczyzną. Jeśli nie można idealnie odseparować tych klas granicą decyzyjności, wagi będą cały czas aktualizowane, chyba, że ustalimy maksymalną liczbę epok.



Rysunek 3: Wykres regionów decyzyjnych

ADELINE czyli adaptacyjne liniowe neurony, są rozwinięciem koncepcji perceptronu. Opiera się on na koncepcji definiowania i minimalizowania funkcji kosztu. Zmianie uległa aktualizacja wag, w tym modelu wykorzystujemy liniową funkcję aktywacji, a nie jak poprzednio skok jednostkowy.

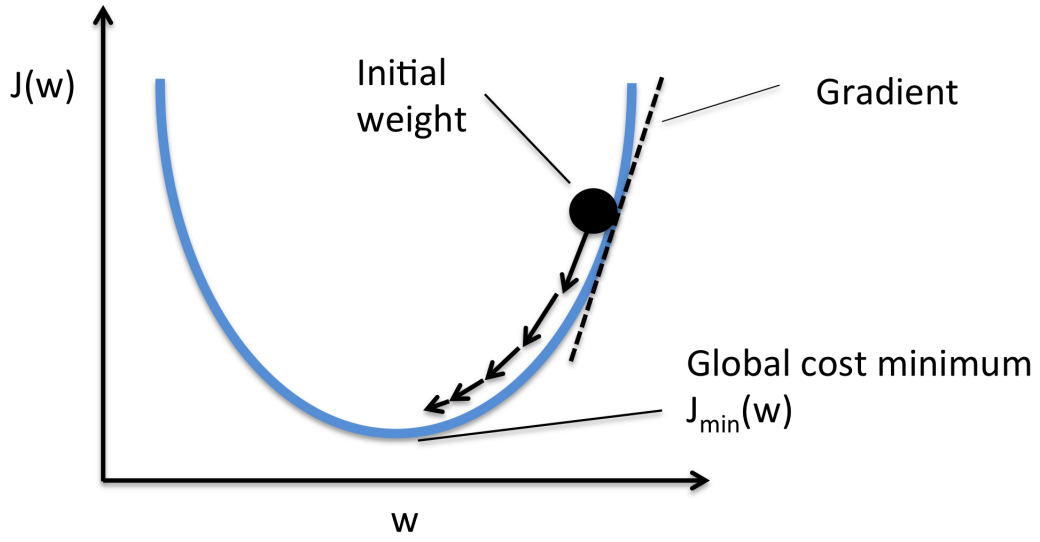


Rysunek 4: Schemat modelu ADELINE

Jak wspomnieliśmy wcześniej, chcemy zdefiniować i zminimalizować funkcję kosztów. Do jej zdefiniowania posłużymy się sumą kwadratów błędów, która wygląda następująco:

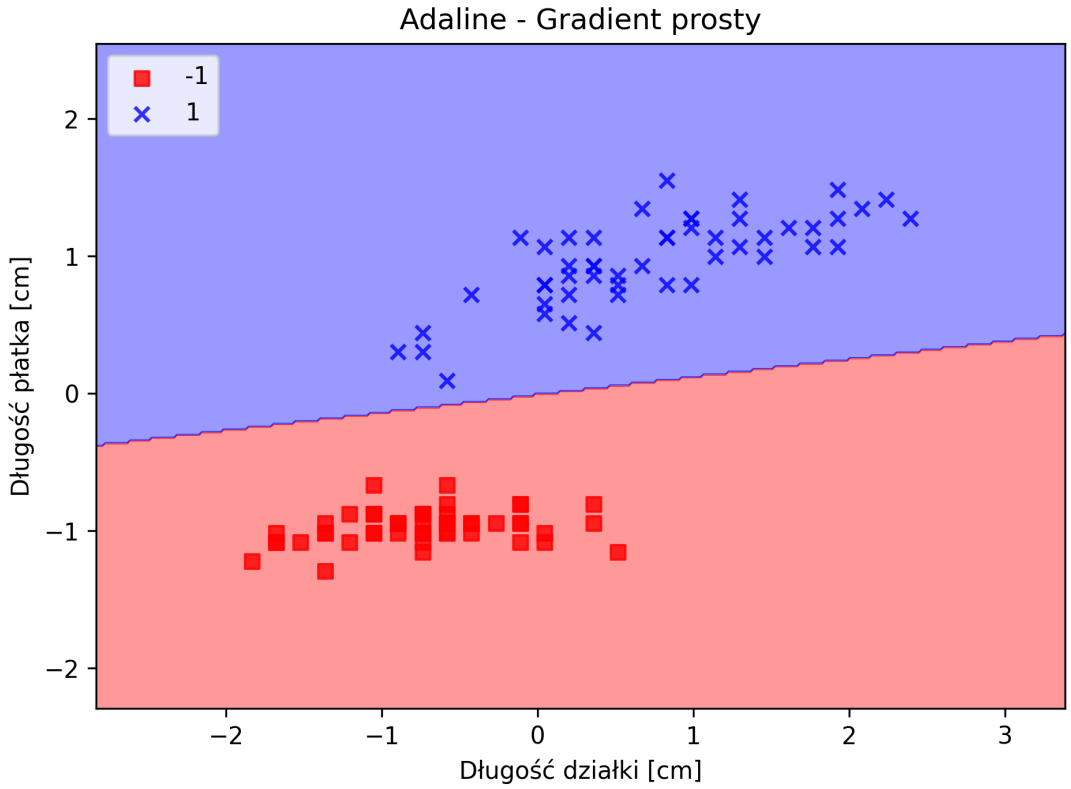
$$J(w) = 1/2 \sum_i (target(i) - output(i))2$$

Możemy ją również nazwać funkcją aktywacji. W odróżnieniu od skoku jednostkowego ta funkcja jest różniczkowalna oraz pozwala na wykorzystanie gradientu prostego, który umożliwia znajdowanie wag minimalizujących funkcję kosztu. Na rysunku 5 przedstawiono jak waga początkowa zmierza do minimum globalnego funkcji kosztu.

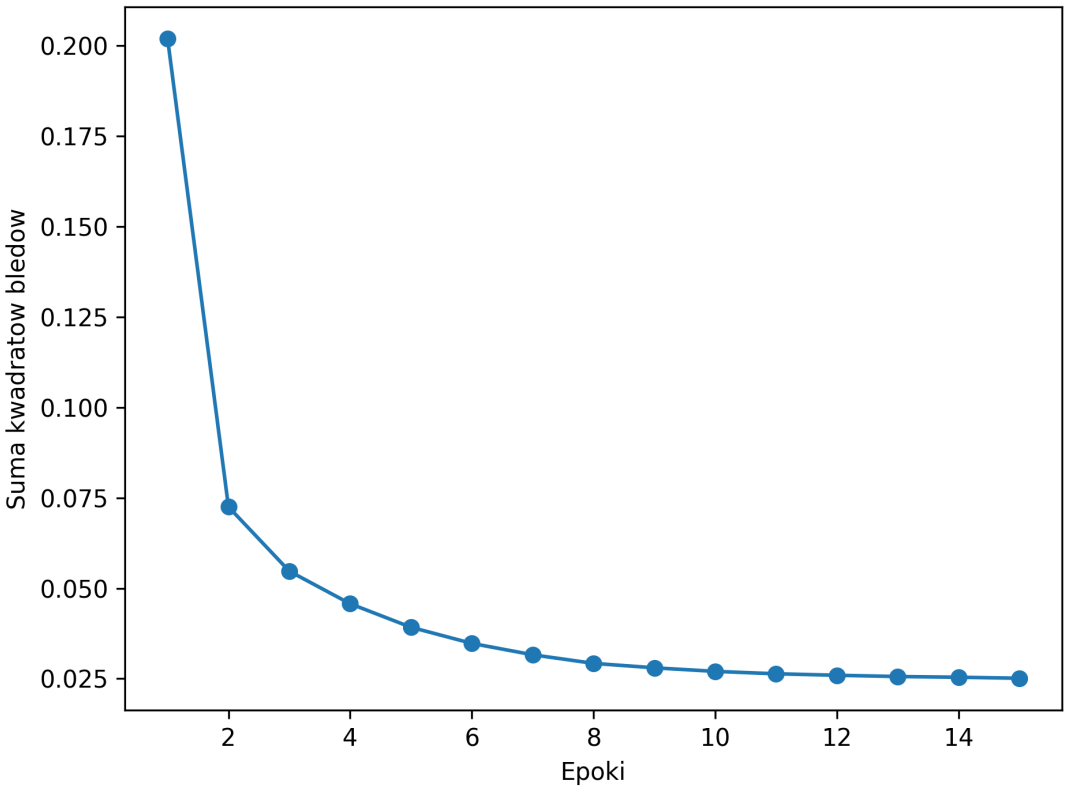


Rysunek 5: Schemat działania algorytmu gradientu prostego

Po standaryzowaniu cech otrzymujemy następujące wykresy:



Rysunek 6: Wykres regionów decyzyjnych dla algorytmu ADELINE

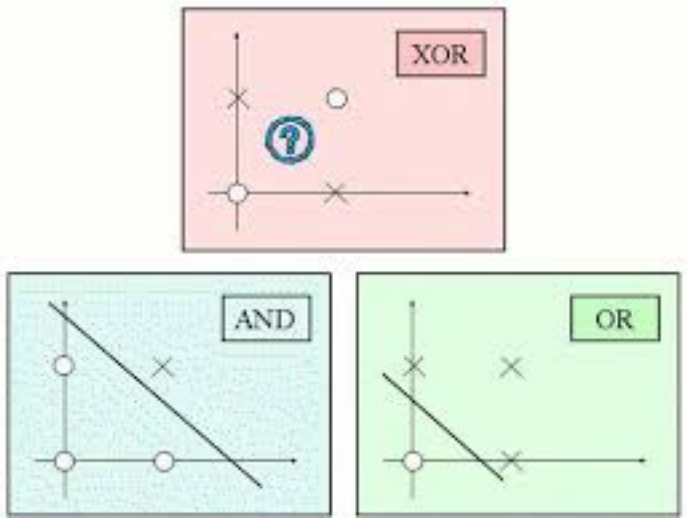


Rysunek 7: Wykres zbieżnosci algorytmu adeline

Pomimo sklasyfikowania wszystkich próbek, suma kwadratów błędów nie zeruje się.

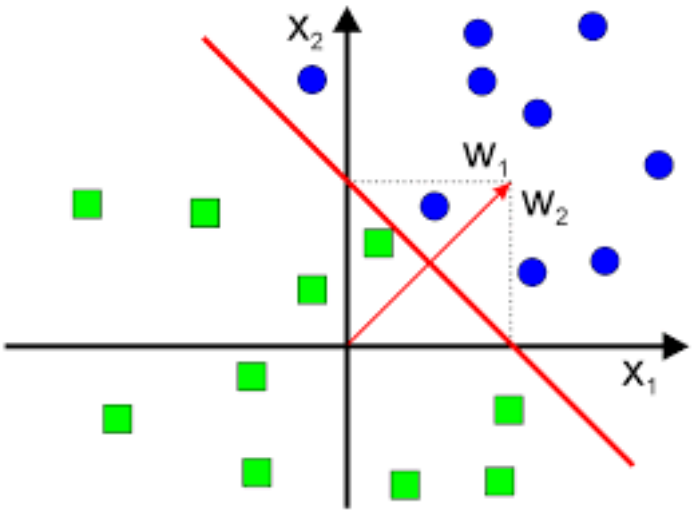
3 Problem XOR

Siec jednokierunkowa jaką jest perceptron prosty, zbudowana jest jedynie z warstwy wejściowej i warstwy wyjściowej. Między warstwami wyjściowymi nie istnieją powiązania, a więc możemy je traktować niezależnie. W 1969 roku Minsky i Papert zauważyli, że wiele interesujących funkcji nie może być modelowanych metodą perceptrona prostego, ponieważ nie zostaje spełniony warunek wektora wag z tw. Rosenblatta.



Rysunek 8: Bramki logiczne dla perceptronu prostego

Jeżeli do perceptronu prostego wchodzi m elementów to przestrzeń dwuwymiarowa zostaje przedzielona na dwie półprzestrzenie wzdłuż granicy decyzyjnej.

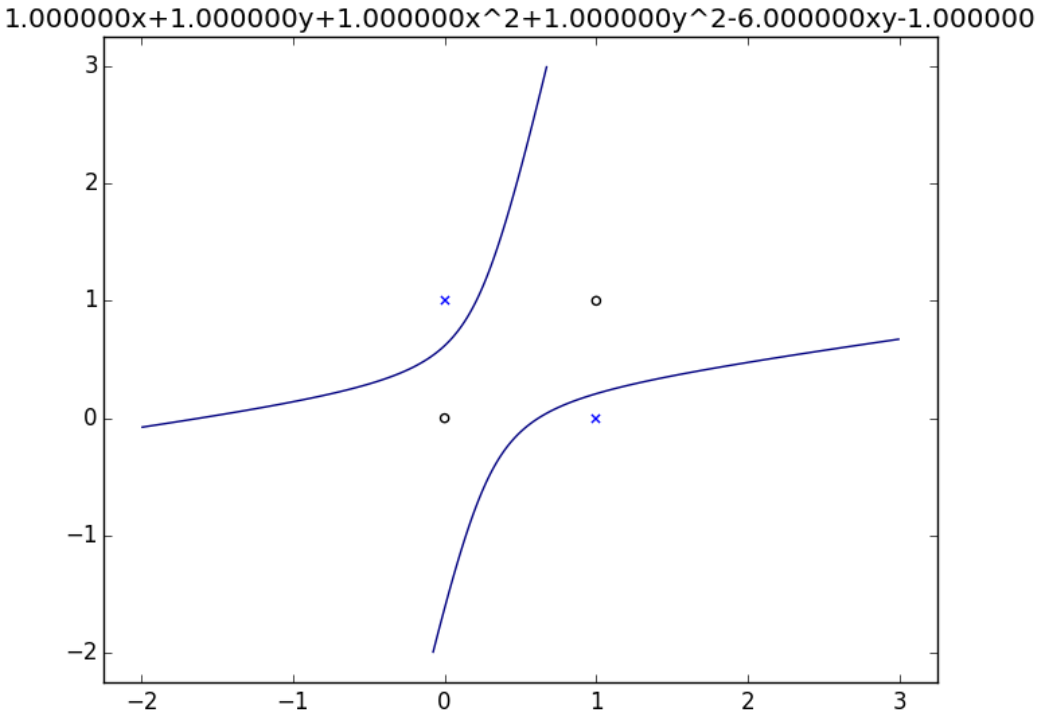


Rysunek 9: Granica decyzyjna

Wniosek z tego jest taki, że zbiory wartości 0 i 1 muszą być separowane liniowo od siebie i leżeć w osobnych półpłaszczyznach. A takiej zasady nie spełnia bramka logiczna XOR. Nie istnieje , bowiem taka prosta, która rozdzieliłaby wartości funkcji XOR równych 0 od wartości 1.

Input		Output
A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

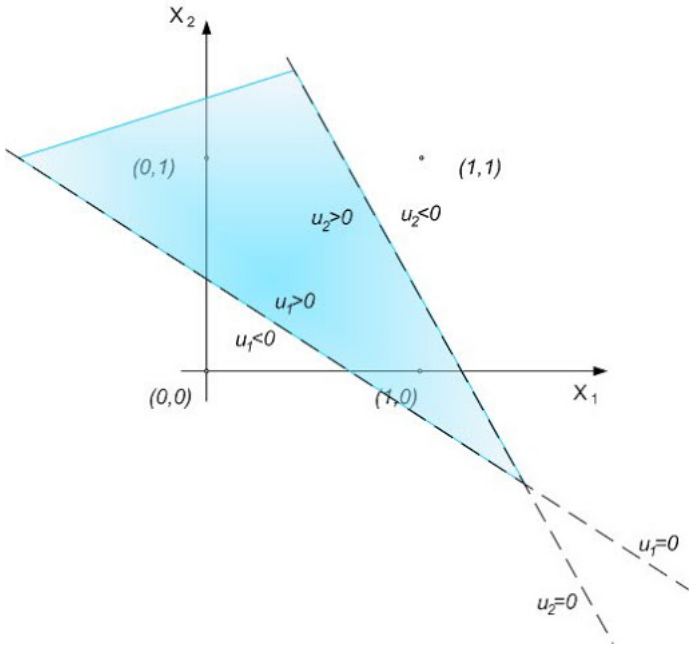
Rysunek 10: Opis funkcji logicznej XOR



Rysunek 11: Ilustracja graficzna problemu XOR

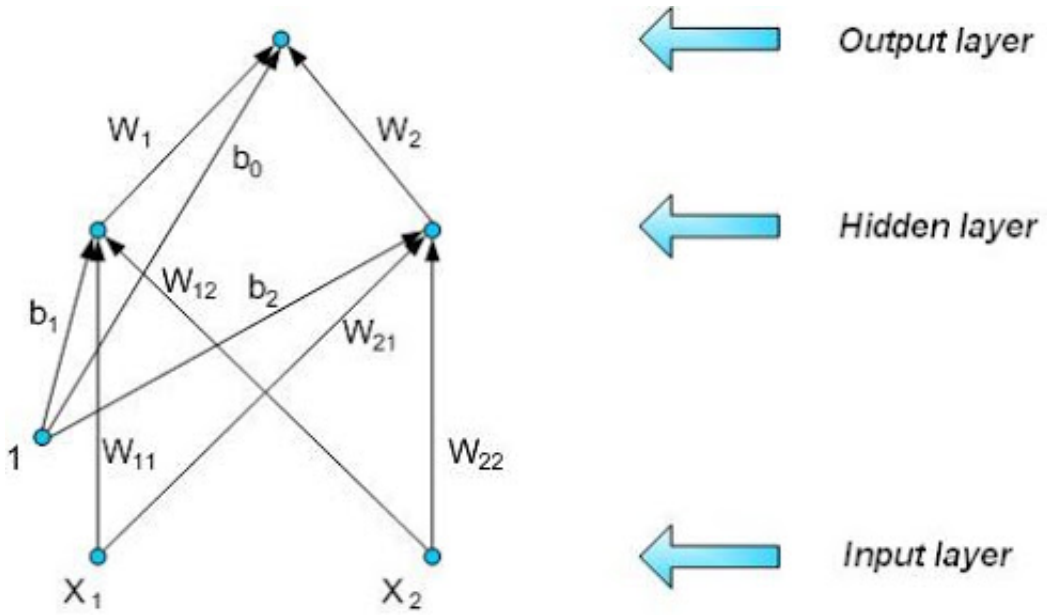
Ciekawostka

Przyjmując, że wartości wejściowe są binarne to dla 2 wejść istnieje 16 funkcji o wartościach binarnych, w tym dwie liniowo nieseparowane: XOR i jego logiczne zaprzeczenie. Dla 32 wejść można wyróżnić 4.3×10^9 funkcji, ale tylko 94572 z nich są liniowo separowane. Rozwiązaniem tego problemu jest rozszerzenie sieci neuronowej w taki sposób, by dodając jeden neuron w warstwie stworzyć sieć neuronów o wagach realizujących następujący podział przestrzeni:

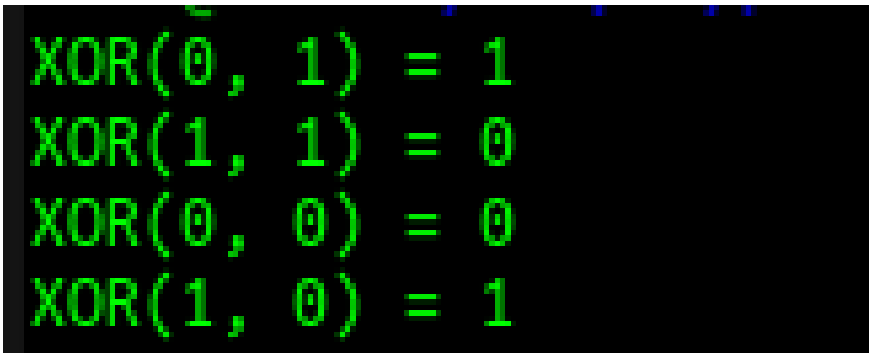


Rysunek 12: Sposób realizacji funkcji XOR za pomocą sieci wielowarstwowej

Dodanie dodatkowej warstwy z neuronem umożliwia zrealizowanie tej sumy logicznej, która jest częścią wspólną zbiorów u_1 i u_2 (rys 11). Każdy dodatkowy neuron umożliwia dokonanie liniowego podziału na granicy $U_i > 0$ i $U_i < 0$ zależnej od wag neuronu. Warstwą wyjściową jest tutaj obszar, który jest kombinacją obszarów mniejszych powstałych z podziału obszarów danych wejściowych.



Rysunek 13: Struktura sieci mogąca realizować funkcję XOR



Rysunek 14: Wynik działania implementacji funkcji XOR

4 Opis kodu

```
class Perceptron(object):
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Rysunek 15: Fragment kodu, w którym zostaje zaimplementowany perceptron.

Funkcja fit tworzy macierz zerową, lepszym rozwiązaniem byłoby zaiplementowanie tego fragmentu z wagami losowymi. Następnie pętla przypisuje dane do odpowiednich kwiatów. Pętla ma także wbudowany licznik błędów, jeżeli kwiat nie zostanie przypisany, zostanie dodany do licznika błędów.

```
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=None)
df.tail()

y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[0:100, [0, 2]].values

plt.scatter(X[:50, 0], X[:50, 1],color='red', marker='o', label='Setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],color='blue', marker='x', label='Versicolor')

plt.xlabel('Długość działki [cm]')
plt.ylabel('Długość płatka [cm]')
plt.legend(loc='upper left')

plt.tight_layout()
plt.savefig('./1.png', dpi=300)
plt.show()
```

Rysunek 16: Fragment kodu, który za pomocą biblioteki pandas pobiera dane na temat kwiatów oraz rysuje wykres z podziałem na rodzaj kwiatów.

```
def plot_decision_regions(X, y, classifier, resolution=0.02):
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])


    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx), marker=markers[idx], label=cl)
```

Rysunek 17: Funkcja, która rysuje na wykresie linie decyzyjną dla naszych pobranych danych i rozdziela dwa rodzaje kwiatów.

```python
class AdalineSGD(object):
    def __init__(self, eta=0.01, n_iter=50, shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        if random_state:
            seed(random_state)

    def _initialize_weights(self, m):
        self.w_ = np.zeros(m + 1)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        output = self.net_input(xi)
        error = target - output
        self.w_[0] += self.eta * error
        self.w_[1:] += self.eta * xi.dot(error)
        cost = 0.5 * error ** 2
        return cost

    def _shuffle(self, X, y):
        seq = np.random.permutation(len(y))
        return X[seq], y[seq]

    def fit(self, X, y):

        self._initialize_weights(X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X,y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost)/len(y)
            self.cost_.append(avg_cost)
        return self

    def partial_fit(self, X, y):
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self
```

Rysunek 18: Klasa perceptronu, który korzysta z metody uczenia ADELINE. Jest to rozszerzenie naszego perceptronu, w którym po kazdej iteracji aktualizujemy wagę decyzyjną.

```python
def NOT_logicFunction(x):
    wNOT = -1
    bNOT = 0.5
    return perceptronModel(x, wNOT, bNOT)

def AND_logicFunction(x):
    w = np.array([1, 1])
    bAND = -1.5
    return perceptronModel(x, w, bAND)

def OR_logicFunction(x):
    w = np.array([1, 1])
    bOR = -0.5
    return perceptronModel(x, w, bOR)

def XOR_logicFunction(x):
    y1 = AND_logicFunction(x)
    y2 = OR_logicFunction(x)
    y3 = NOT_logicFunction(y1)
    final_x = np.array([y2, y3])
    finalOutput = AND_logicFunction(final_x)
    return finalOutput
```

Rysunek 19: Implementacja problemu XOR.

Do rozwiązania tego problemu potrzebowaliśmy aż trzech perceptronów, co pozwoliło na rozbudowanie strefy, na której spełnione zostaja założeenia tej funkcji.

5 Podsumowanie

Projekt ten umożliwił nam zapoznanie się z koncepcją uczenia maszynowego i jego podstawowym elementem, czyli perceptronem. Na przykładzie irysa widać w praktyce jak działa podział i czym jest linia decyzyjna dla prostego elementu tego typu. Ważnym fragmentem była funkcja kosztu, ponieważ usprawniła ona algorytm rozdzielający kwiaty. Na przykładzie XOR-u mieliśmy okazję zobaczyć ograniczenia takiego prostego podziału liniowego. Nie udało się rozwiązać problemu jak w przykładzie z kwiatami, jednak wprowadzenie dodatkowego perceptronu ograniczyło strefę decyzyjną, dzięki czemu można rozwiązywać problemy liniowo nieseparowane.

Literatura

[1] "The Nonlinear Workbook", Willi-Hans Steeb

[2] "Python. Uczenie maszynowe.", Sebastian Raschka, Vahid Mirjalili