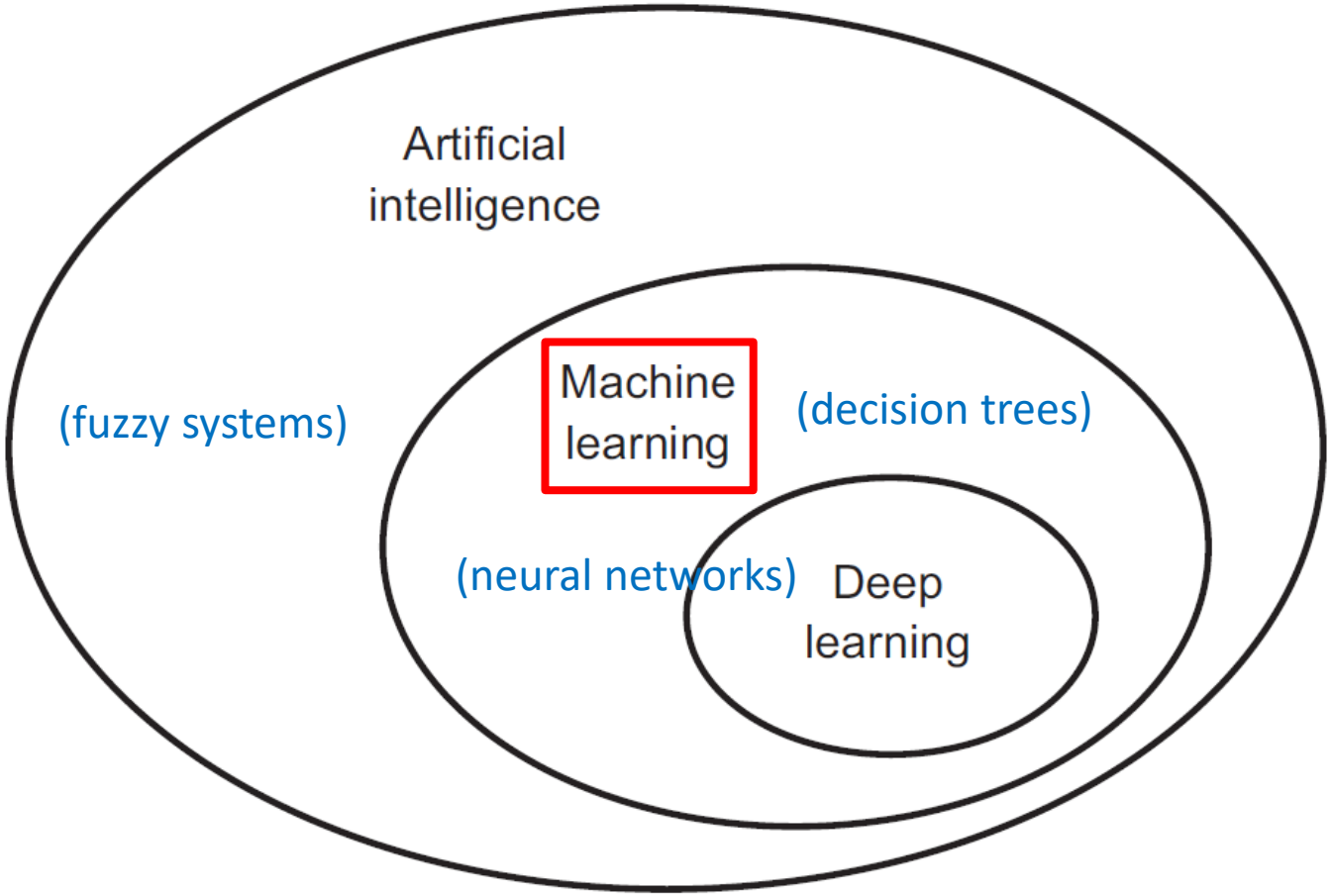
Systemy AI i budowa systemów decyzyjnych

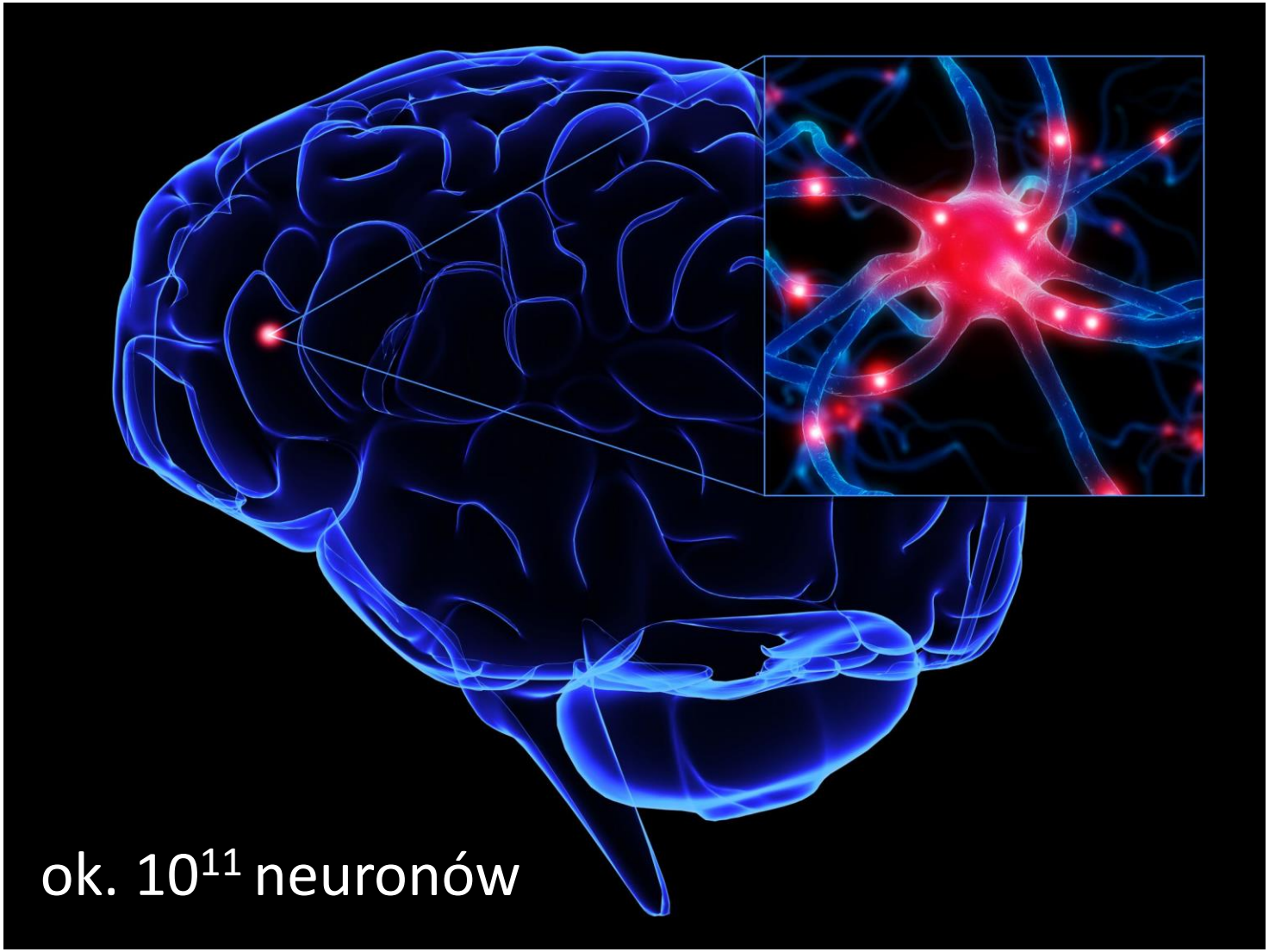
Sieci neuronowe

część 1

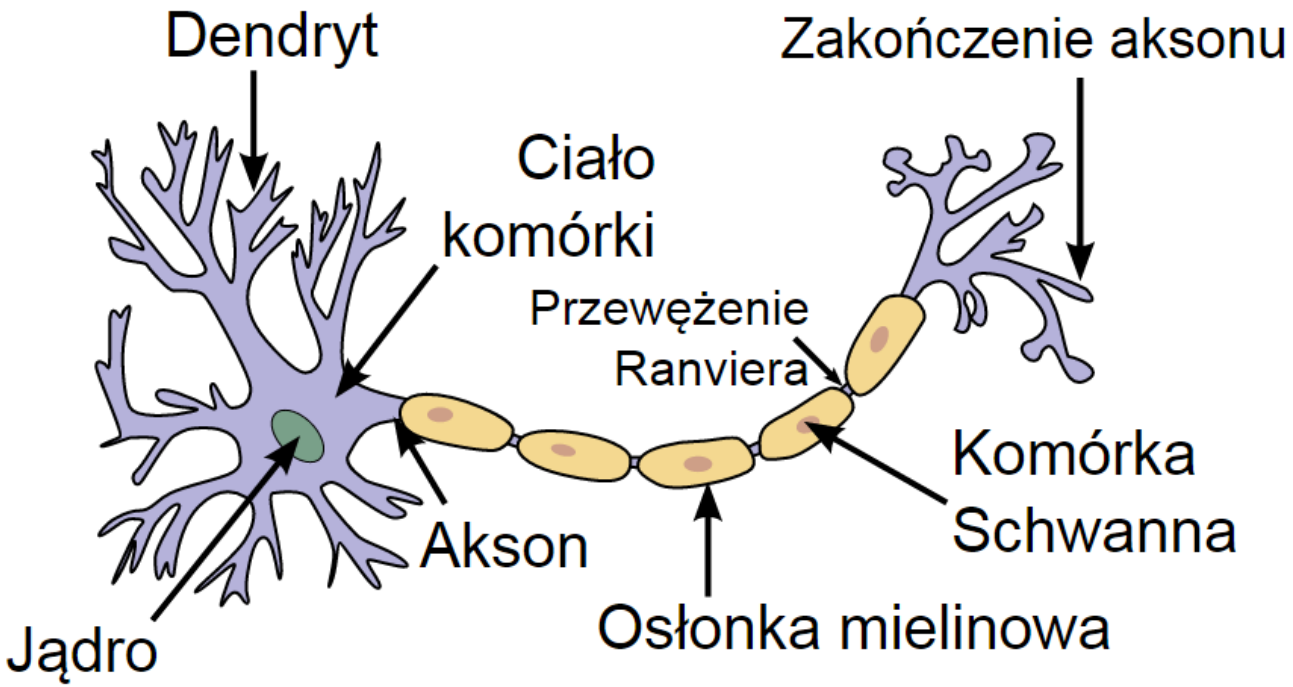
Sztuczna inteligencja

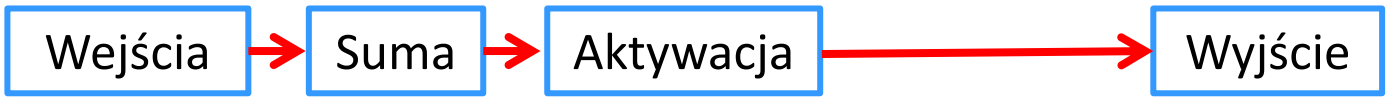


Neurony



Neuron


Dendryt
Zakończenie aksonu
Ciało komórki
Przewężenie Ranviera
Komórka Schwanna
Akson
Jądro
Osłonka mielinowa


Wejścia
Suma
Aktywacja
Wyjście

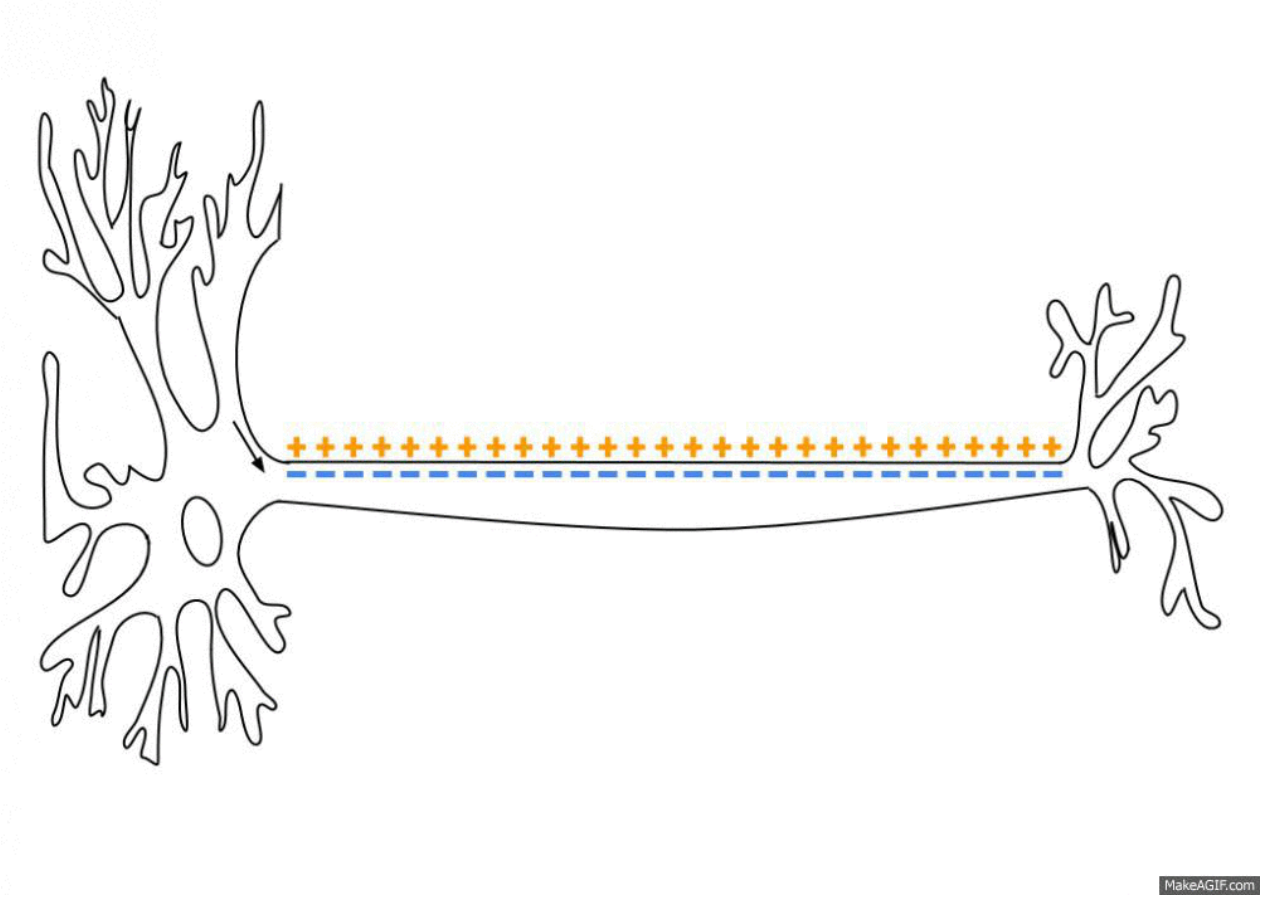
Neuron

- Neurony przesyłają między sobą sygnały (pobudzenie). Pojedynczy neuron może przyjmować sygnały od 1000 innych neuronów.
- Neuron przekazuje pobudzenie innym neuronom przez złącza nerwowe zwane synapsami. Transmisja sygnałów to skomplikowany proces chemiczno-elektryczny.
- Synapsy to przekaźniki informacji. Mogą wzmocnić pobudzenie bądź osłabić.

Neuron

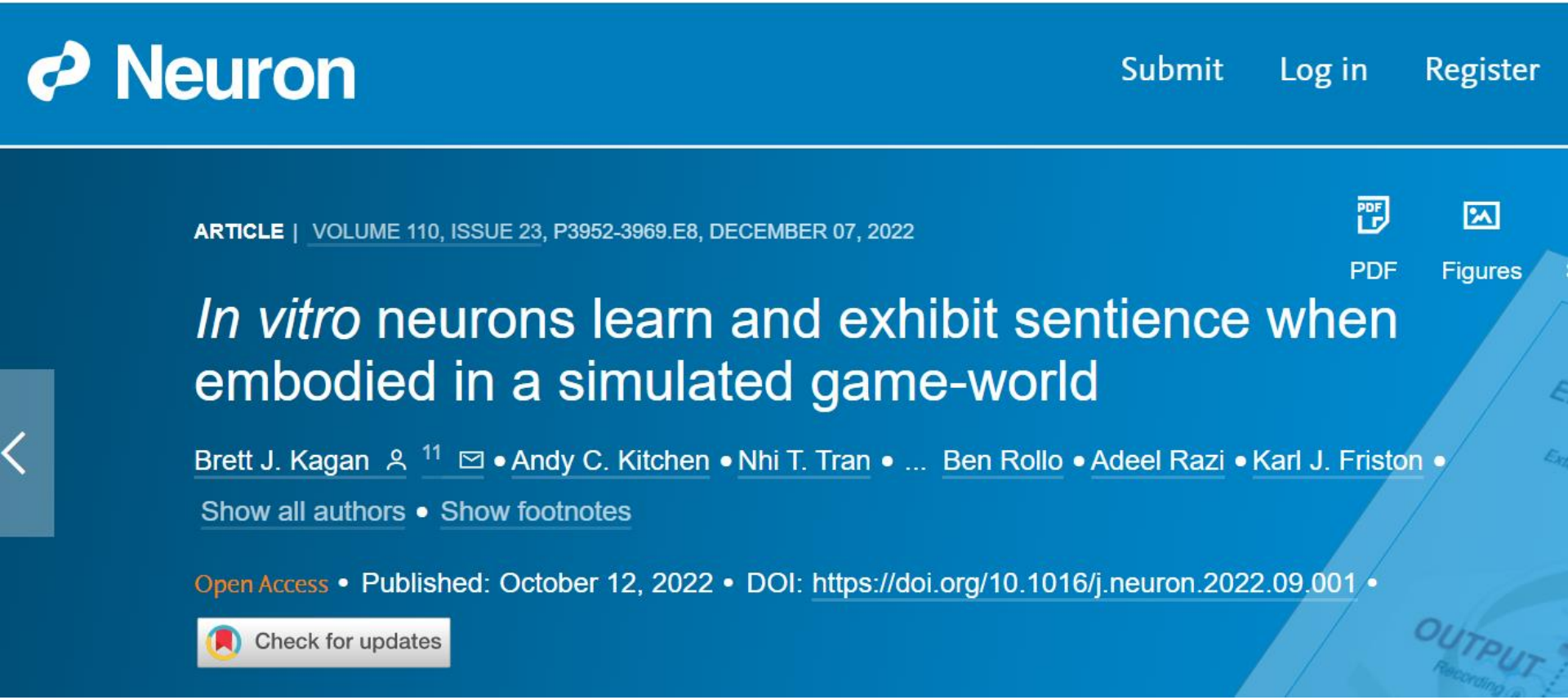
- Cześć sygnałów docierających do neuronu wywiera wpływ pobudający na neuron, a część hamujący.
- Neuron sumuje impulsy pobudzające i hamujące.
- Jeżeli suma ta przekracza pewną wartość progową wówczas sygnał na wyjściu, przez akson, jest przesyłany do innych neuronów.

Neuron



https://pl.wikipedia.org/wiki/Neuron#/media/File:Action_Potential.gif

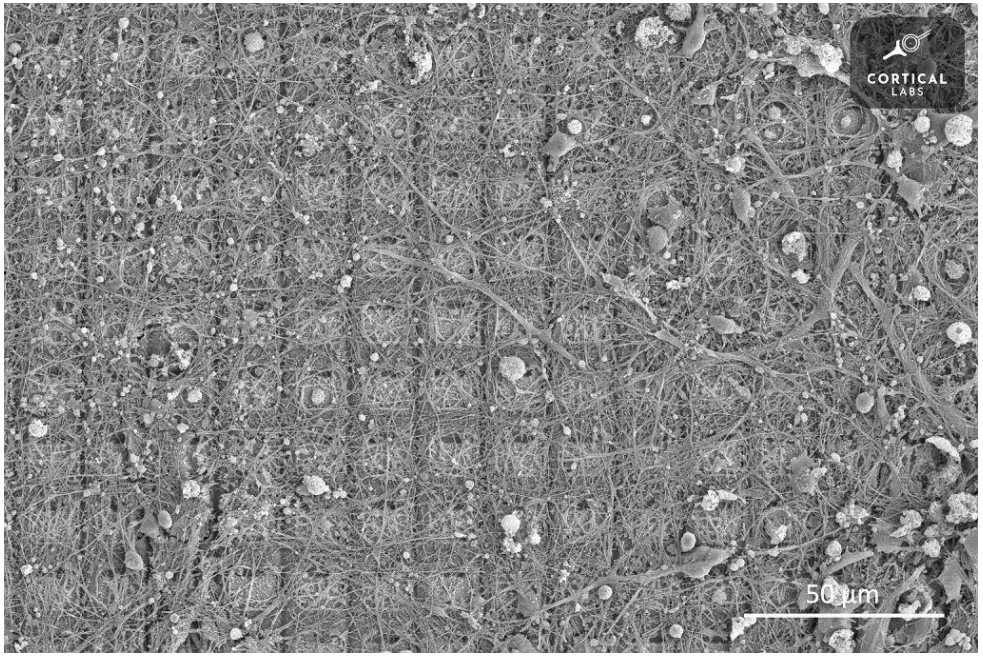
DishBrain



https://www.cell.com/neuron/fulltext/S0896-6273(22)00806-6

DishBrain

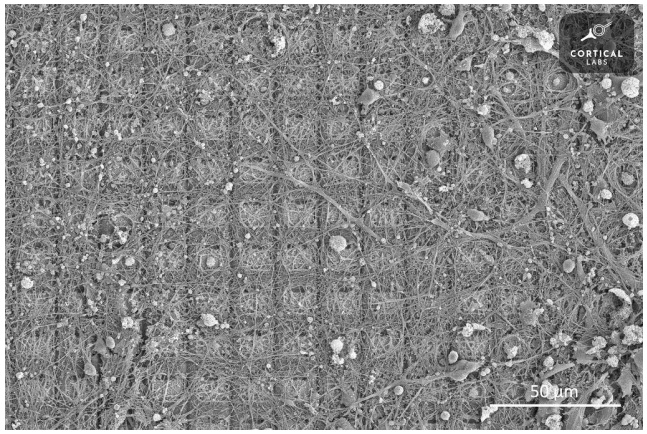
Kilkaset tysięcy mysich i ludzkich neuronów na siatce elektrod. Po miesiącu neurony utworzyły sieć (wetware):



Elektrony komunikowały się nie tylko między sobą, ale także z komputerem za pomocą siatki elektrod.

DishBrain

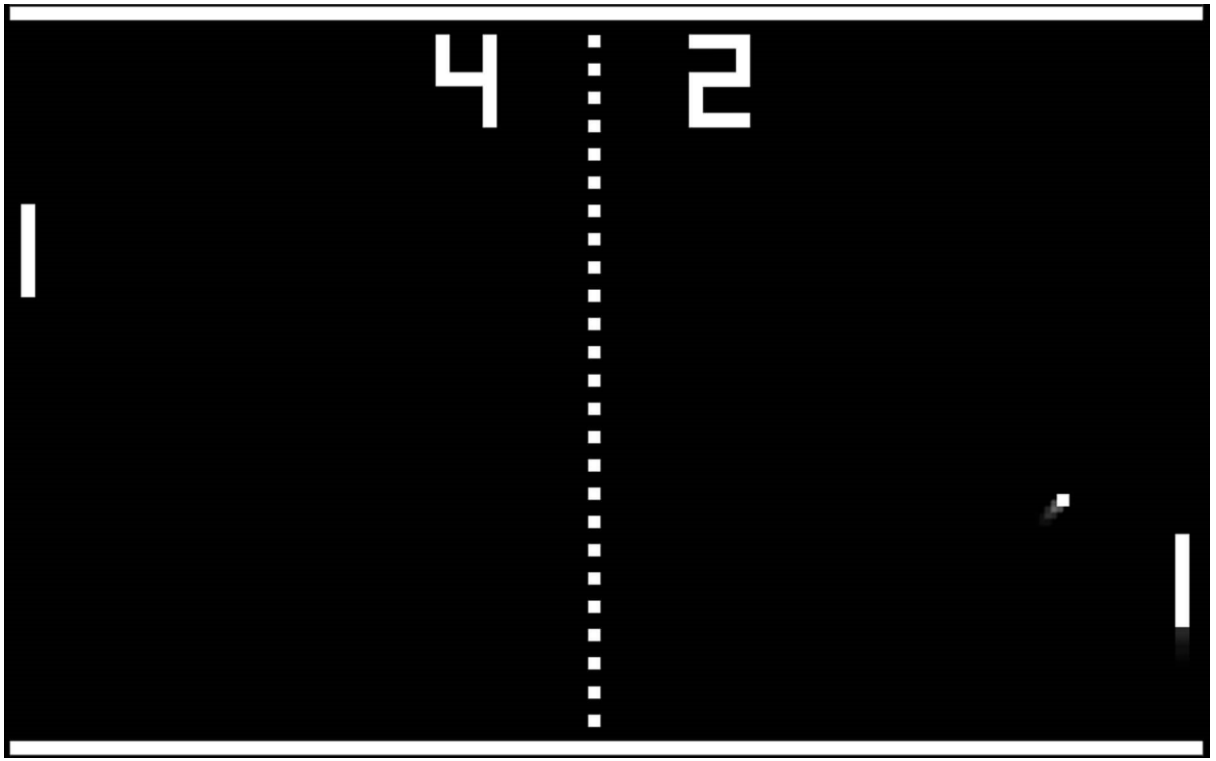
Między siatką elektrod i komputerem możliwe było przesyłanie sygnałów elektrycznych:



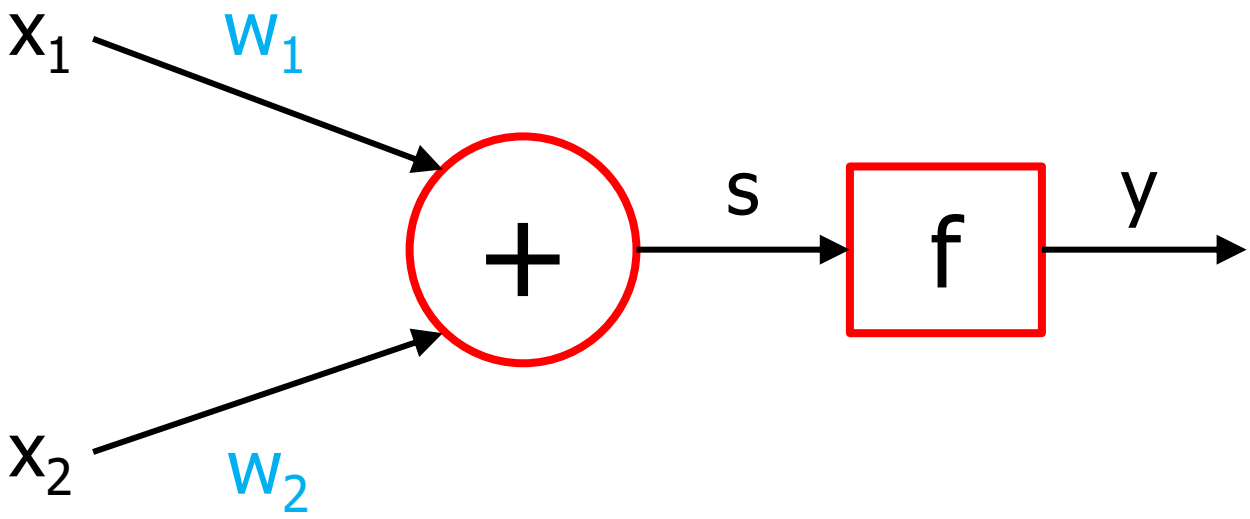
Neurony w ciągu 5 minut (!!!) nauczyły się grać w grę Pong. Było to możliwe dzięki **informacji zwrotnej**, którą otrzymywały od komputera (trafienie w rakietkę lub nie).

DishBrain

… czyli sterować rakietką:



Model neuronu - perceptron

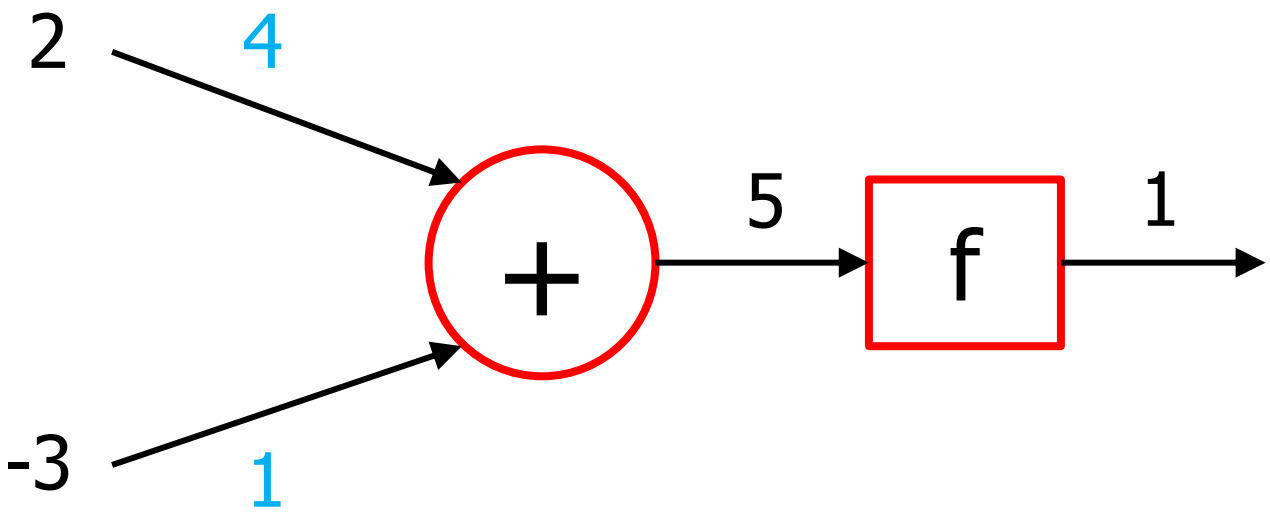


$s = x_1 w_1 + x_2 w_2$

w_1 i w_2 to wagi perceptronu

$$y = f(s) = \begin{cases} 1 \; jeżeli \; s > 0 \\ 0 \; jeżeli \; s \leq 0 \end{cases}$$

Model neuronu - perceptron

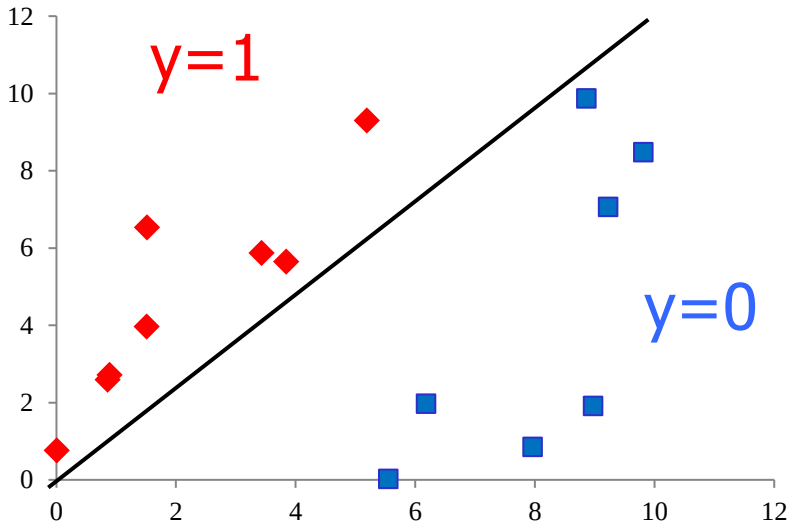


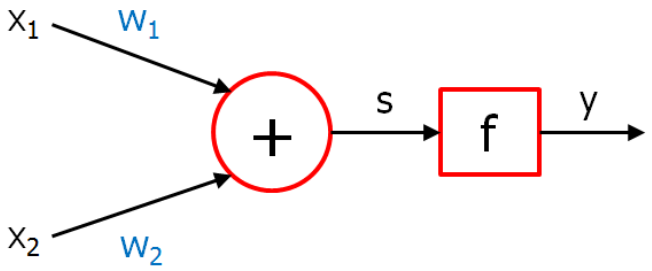
$$s=2\cdot 4 + (-3)\cdot 1=5$$

$$1 = \mathrm{f}(5) = \begin{cases} 1 \; jeżeli \; 5 > 0 \\ 0 \; jeżeli \; 5 \leq 0 \end{cases}$$

Model neuronu - perceptron

Perceptron może nam poklasyfikować punkty na płaszczyźnie:





$s = x_1 w_1 + x_2 w_2$

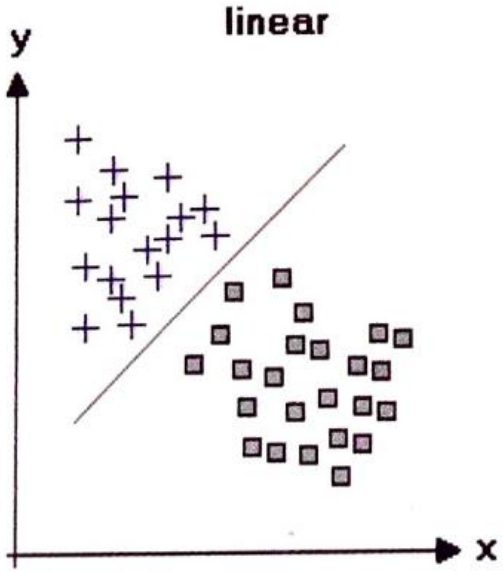
w_1 i w_2 to wagi perceptronu

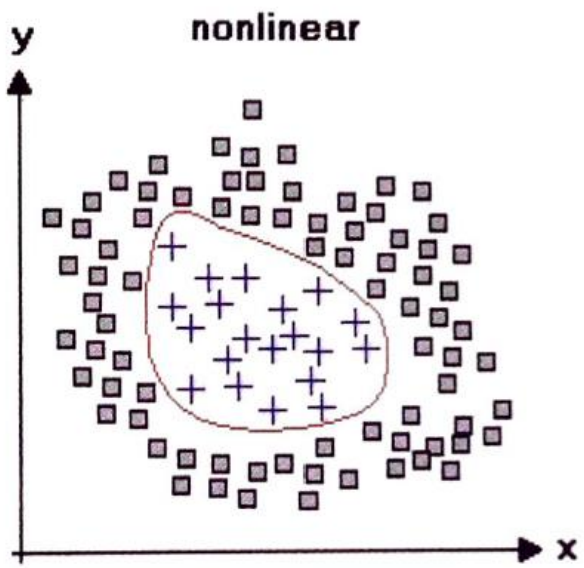
$$y = f(s) = \begin{cases} 1 \; jeżeli \; s > 0 \\ 0 \; jeżeli \; s \leq 0 \end{cases}$$

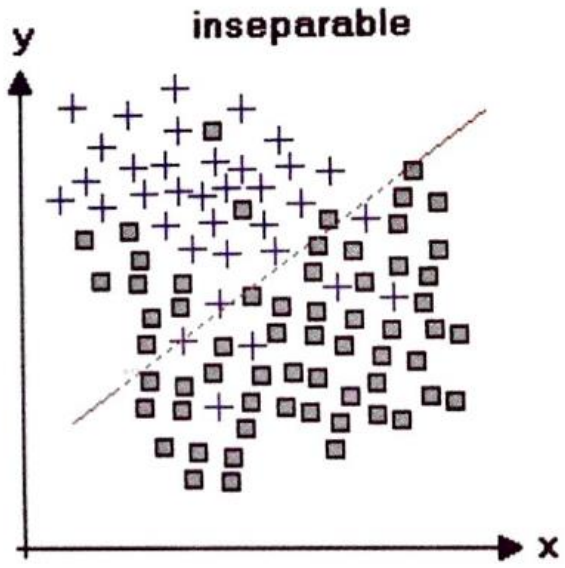
Musimy go tylko tego nauczyć (→ UCZENIE MASZYNOWE)

Model neuronu - perceptron

Jakie możliwości ma perceptron?







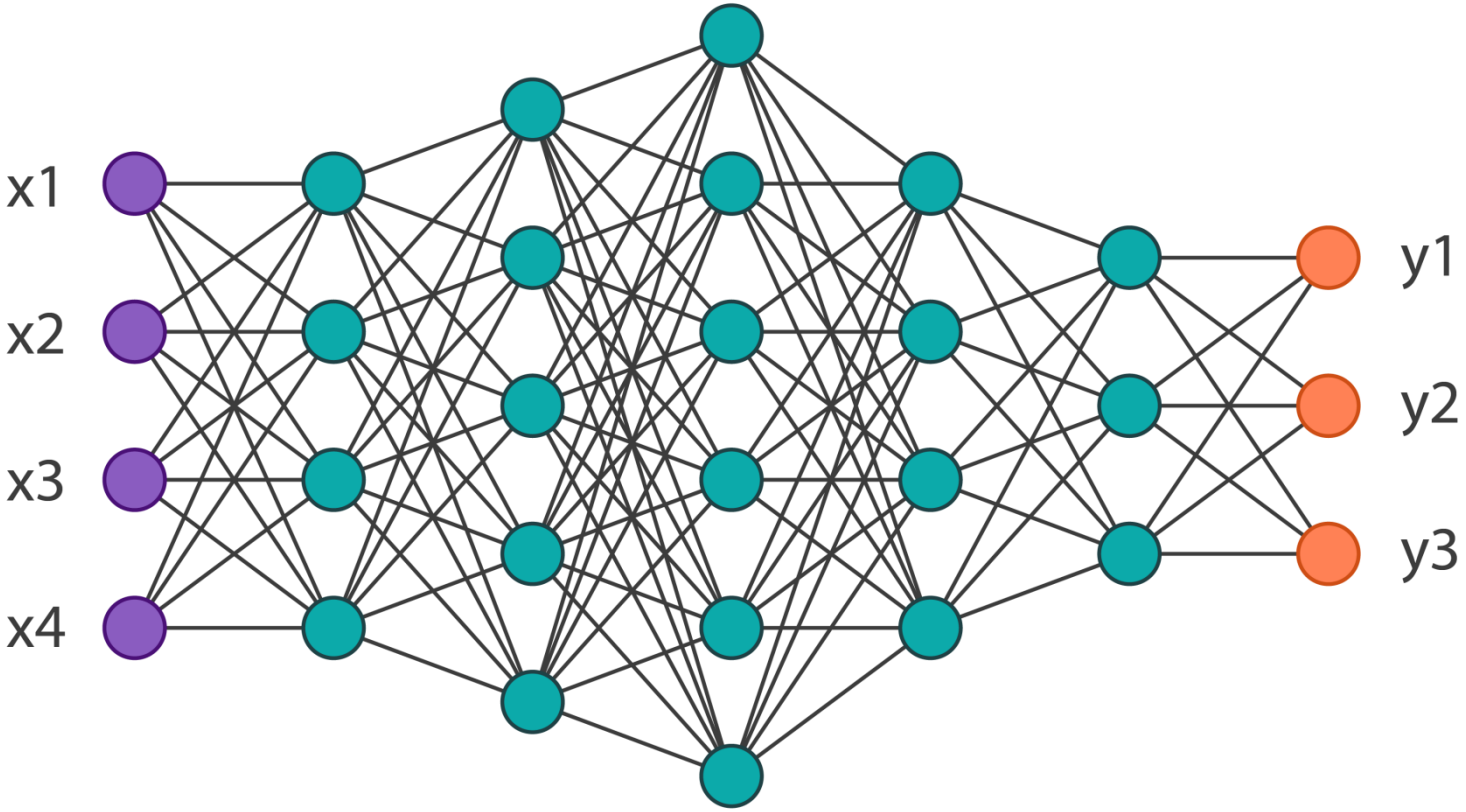
Model neuronu - perceptron

Algorytm uczenia perceptronu:

1. Ustalamy wartości początkowe wag w_1, w_2 perceptronu.
2. Podajemy na wejścia współrzędne punktu (x_1, x_2).
3. Wyliczamy s i następnie y.
4. Jeżeli y nie jest takie jak oczekiwane (1 dla punktów czerwonych, 0 dla punktów niebieskich) wówczas zmieniamy wagi w_1, w_2 perceptronu.
5. Jeżeli dla wszystkich punktów ze zbioru uczącego perceptron zwraca oczekiwane wartości wówczas kończymy naukę. W przeciwnym razie wracamy do punktu 1.

Uczenie maszynowe (ML)

Neurony możemy łączyć w sieci:



Uczenie maszynowe (ML)

kobieta (1)
czy
mężczyzna (0)

Uczenie maszynowe (ML)

Musimy sieć neuronową nauczyć rozpoznawać:

kobieta

mężczyzna

kobieta

mężczyzna

kobieta

mężczyzna

Na czym polega uczenie maszynowe?

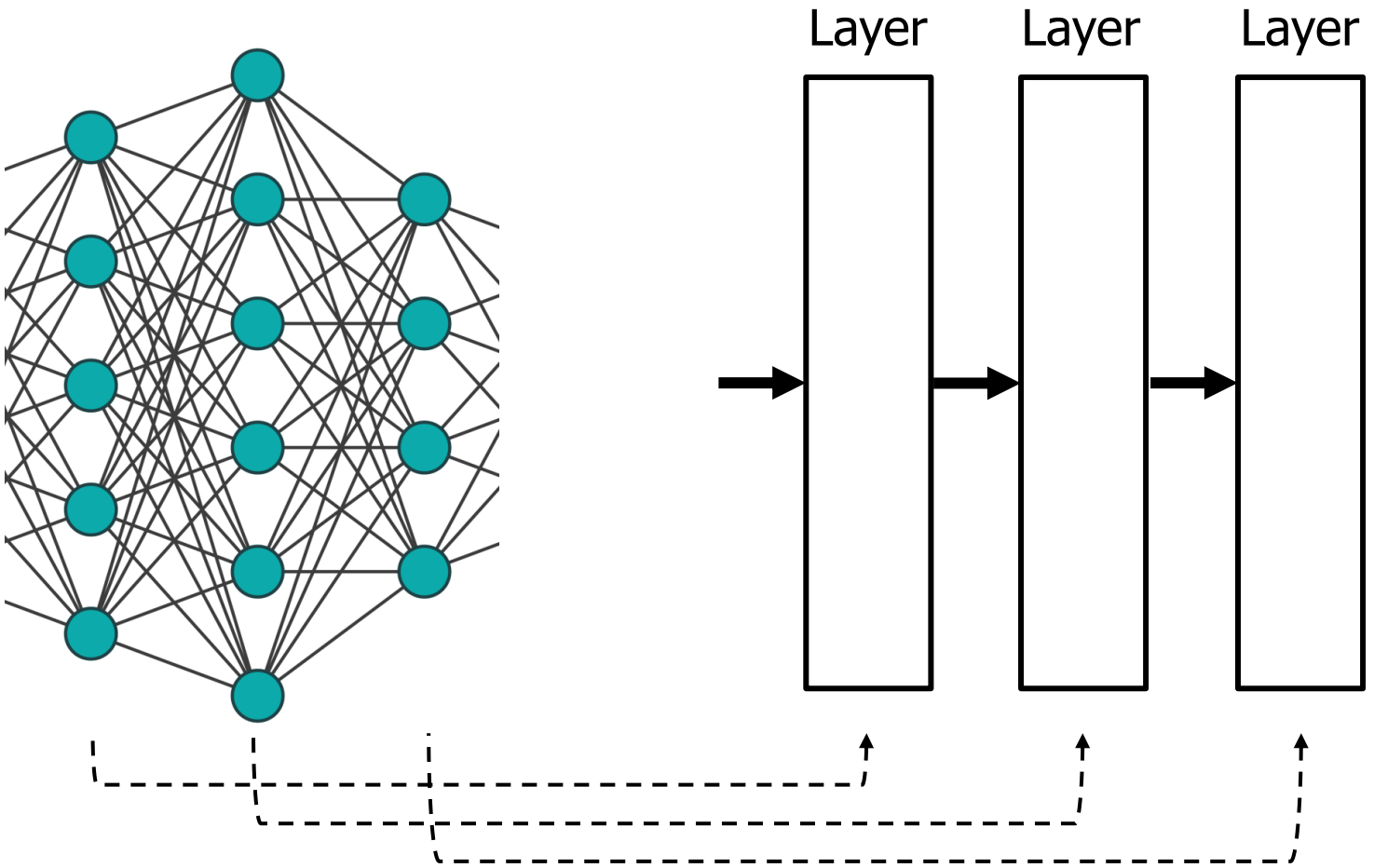
Potrzebujemy 3 rzeczy:

- Dane wejściowe – np. pliki audio, obrazy etc.
- Przykłady oczekiwanych wyjść – np. zapis tekstowy plików audio, podpisy obrazów etc.
- Sposób pomiaru czy algorytm dobrze działa. Ocena działania algorytmu pozwala modyfikować działanie algorytmu → uczenie!

W uczeniu maszynowym dane wejściowe są przekształcane na dane wyjściowe. Jak przekształcać maszyna uczy się korzystając z przykładowych oczekiwanych wyjść.

Uczenie maszynowe (ML)

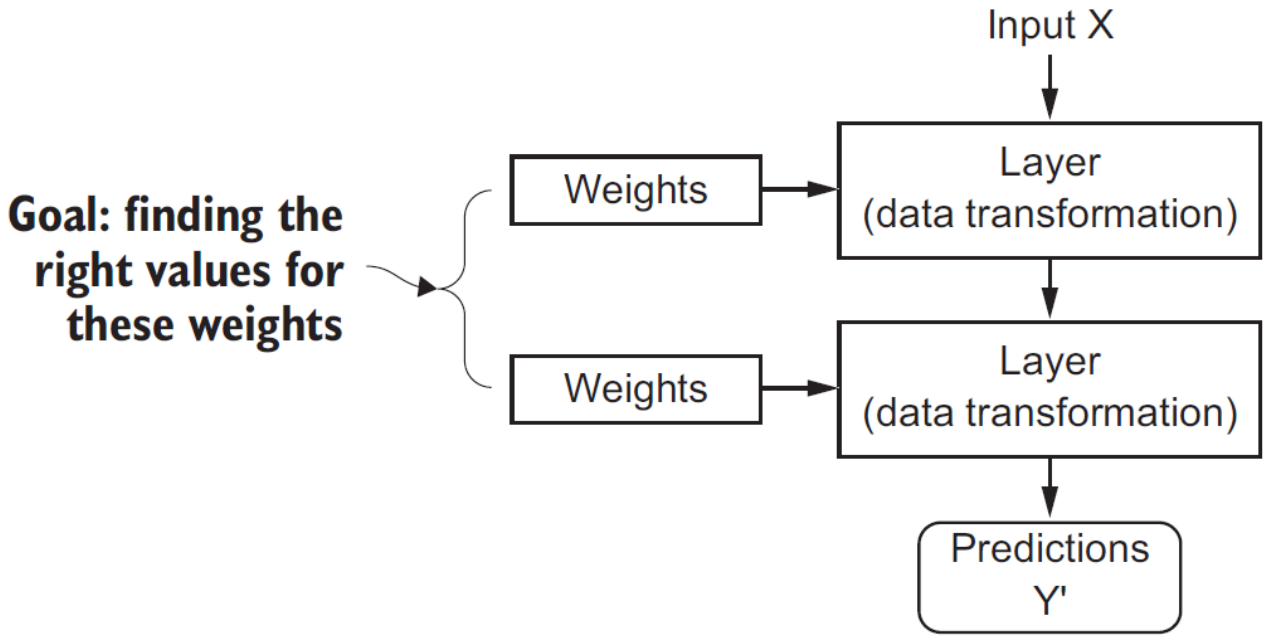
Wprowadźmy następujące oznaczenie warstw sieci neuronowej:



Uczenie maszynowe (ML)

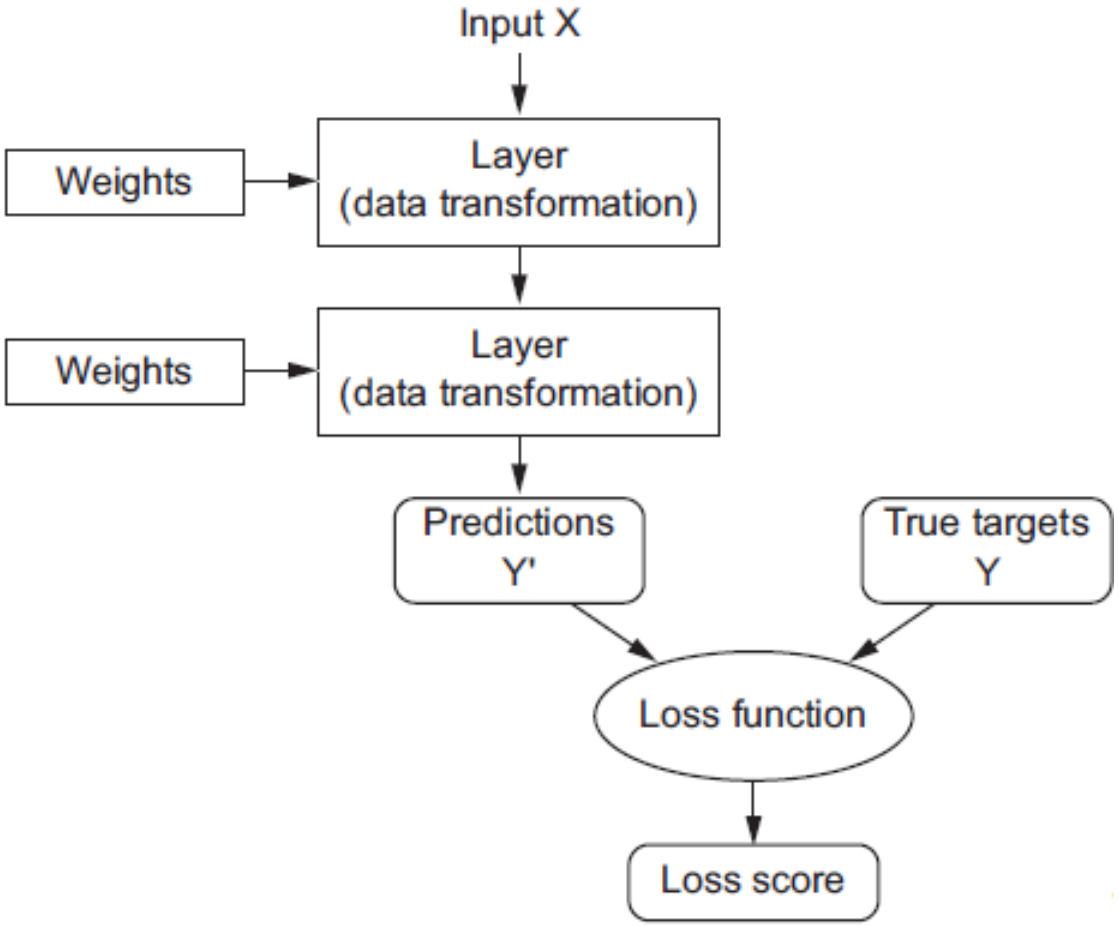
Przekształcania danych (data transformations) możemy układać w warstwy (layers).

Przekształcania danych są parametryzowane przez wagi. Celem jest znalezienie odpowiednich wartości wag.



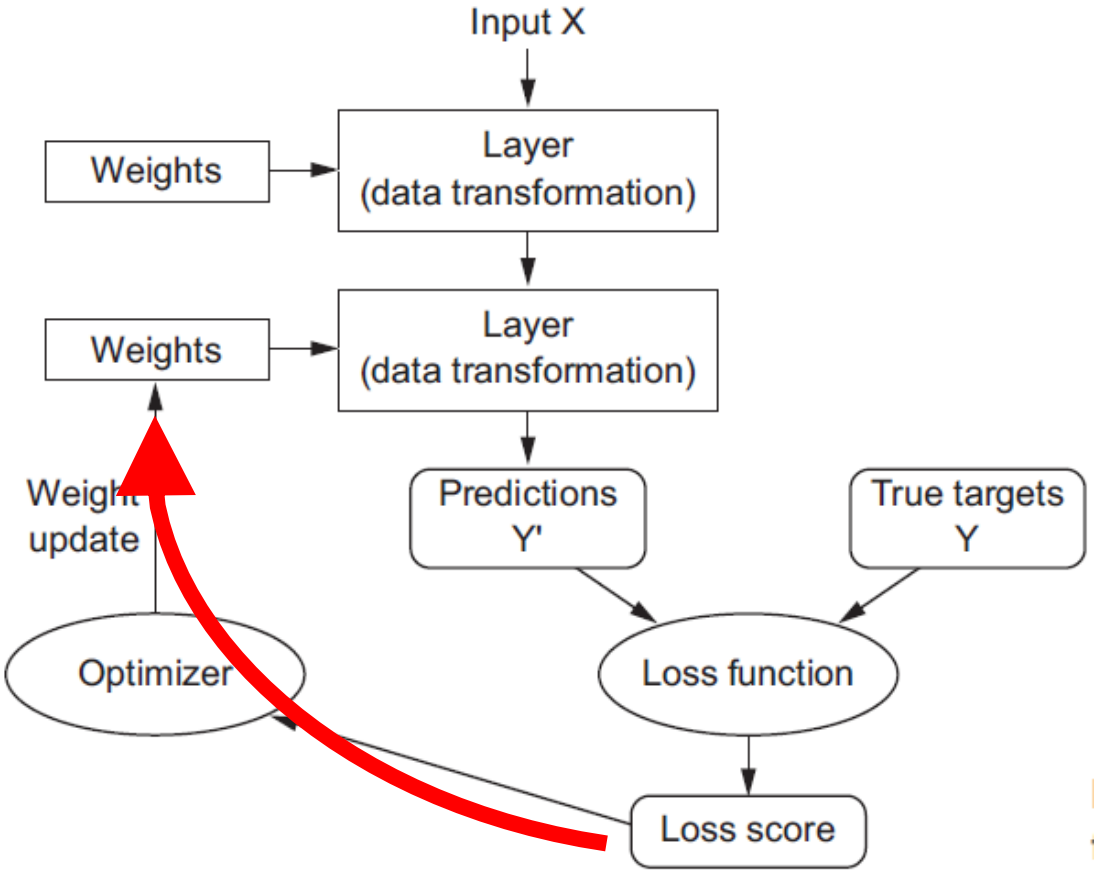
Uczenie maszynowe (ML)

Sprawdzenie poprawności przekształcania danych możliwe jest dzięki funkcji straty (loss function).



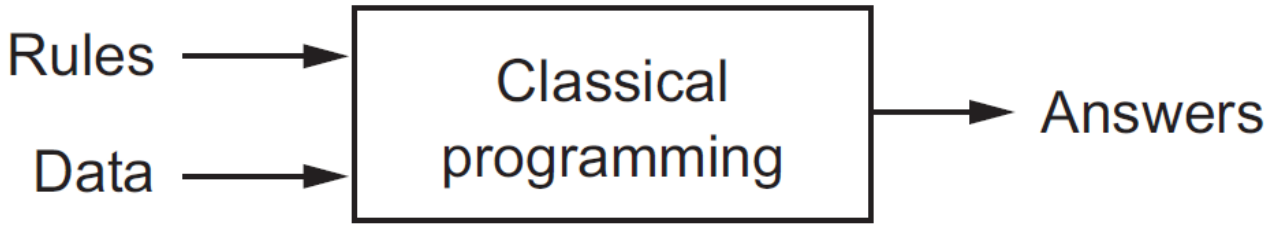
Uczenie maszynowe (ML)

W oparciu o wartość policzonego błędu (loss score) modyfikowane są wagi.

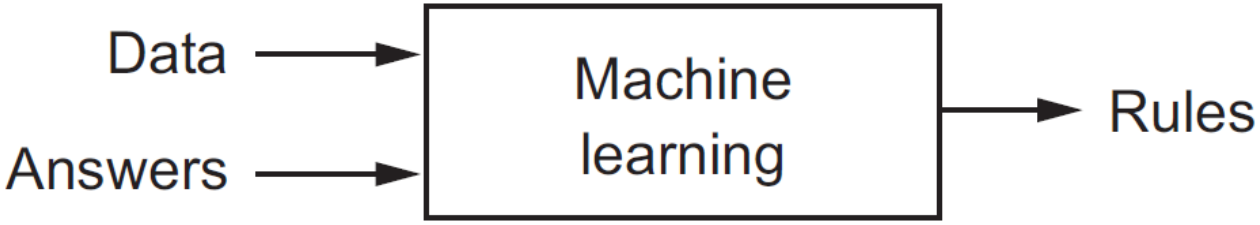


Uczenie maszynowe (ML)

Stary paradygmat programowania:



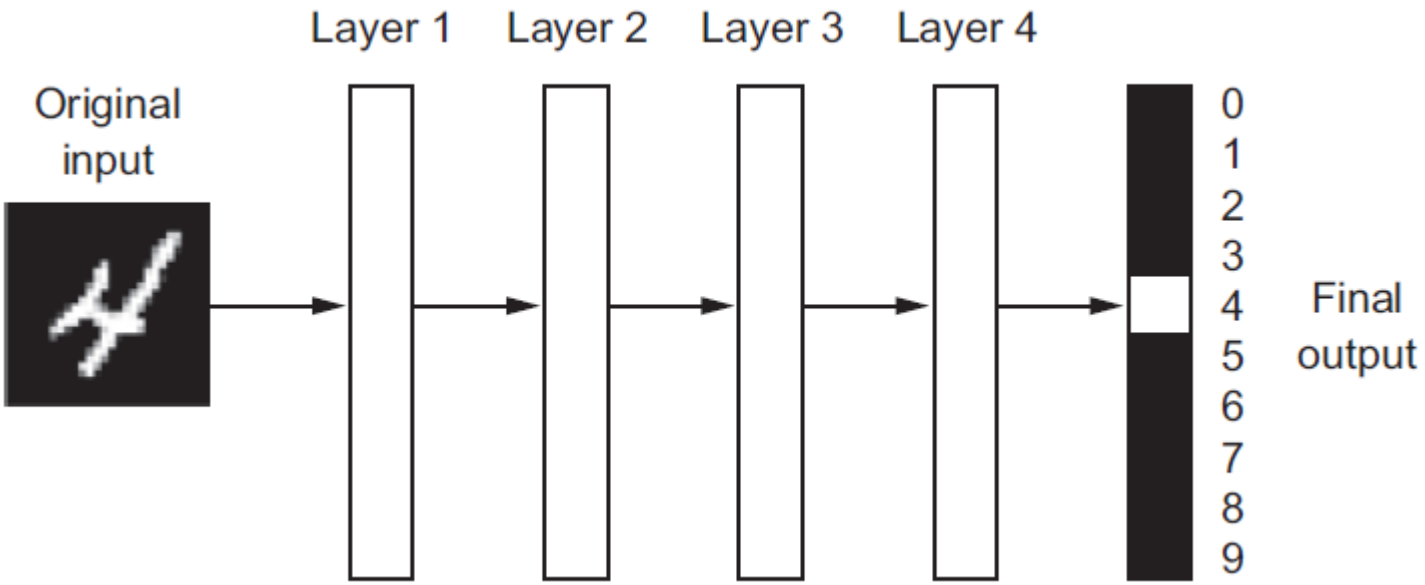
Nowy paradygmat!



System działający w oparciu o uczenie maszynowe jest raczej trenowany (uczony) niż programowany!

Uczenie głębokie

W systemach uczenia głębokiego (deep learning) liczba warstw jest (bardzo) duża.



Uczenie maszynowe (ML)

Musimy pokazać sieci neuronowej tysiące przykładowych twarzy kobiet i mężczyzn na podstawie, których będzie się uczyła:

Sieci neuronowe potrafią **uogólniać wiedzę** – po nauczeniu sieć będzie poprawnie klasyfikowała zdjęcia (kobieta czy mężczyzna), których wcześniej nie widziała.

Rozpoznawanie wzorców

Sieć w czasie uczenia znajdzie **wzorzec kobiety** i **wzorzec mężczyzny**.

Rozważmy następujące dwie liczby:

0, 1, ?, ?, ?, ?, …

Jakie będą następne liczby?

Liczb jest zbyt mało i trudno zauważyć tutaj jakąś zależność (wzorzec).

Rozpoznawanie wzorców

Rozważmy więcej liczb:

0, 1, 1, 2, ?, ?, …

Jakie będą następne liczby?

Możemy przyjąć, że następną jest 2.

Rozpoznawanie wzorców

Rozważmy jeszcze więcej liczb:

0, 1, 1, 2, 3, 5, 8, 13, 21, ?, ?, ?, …

Jakie będą następne liczby?

Rozpoznajemy, że jest to tzw. ciąg Fibonacciego tzn.

$$F_n := \begin{cases} 0 & \text{dla } n = 0, \\ 1 & \text{dla } n = 1, \\ F_{n-1} + F_{n-2} & \text{dla } n > 1. \end{cases}$$

Rozpoznawanie wzorców

Im więcej danych tym łatwiej znaleźć wzorce!

Regresja liniowa

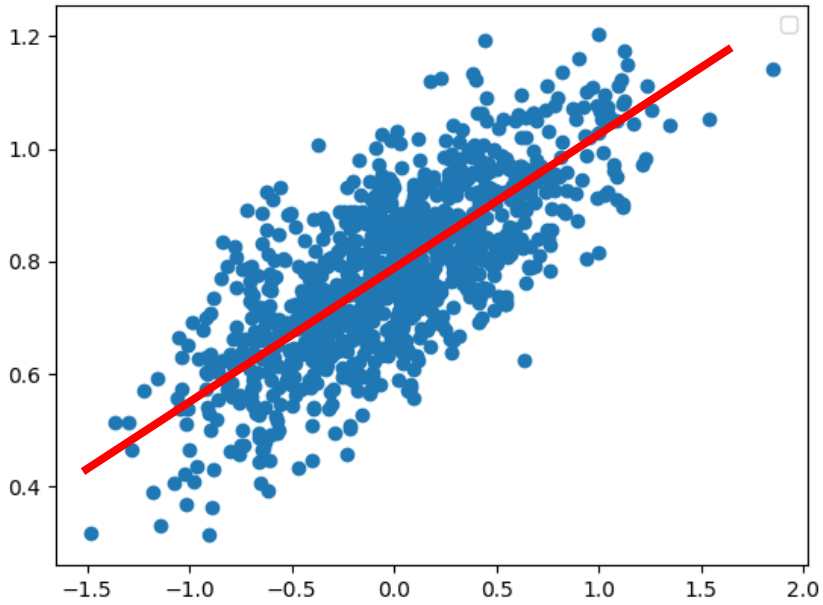
Regresja liniowa – metoda estymowania wartości oczekiwanej zmiennej **y** przy znanych wartościach innej zmiennej lub zmiennych **x**.

Zmienna **y** jest nazywana zmienną objaśnianą lub zależną. Zmienne **x** nazywane są zmiennymi objaśniającymi lub niezależnymi.

Zarówno zmienne objaśniane i objaśniające mogą być wielkościami skalarnymi lub tensorami.

Regresja liniowa

Regresja liniowa jest nazywana liniową, gdyż zakładanym modelem zależności między zmiennymi zależnymi a niezależnymi, jest funkcja liniowa bądź przekształcenie liniowe (afiniczne) reprezentowane przez macierz (tensor!) w przypadku wielowymiarowym.



Jaka prosta?

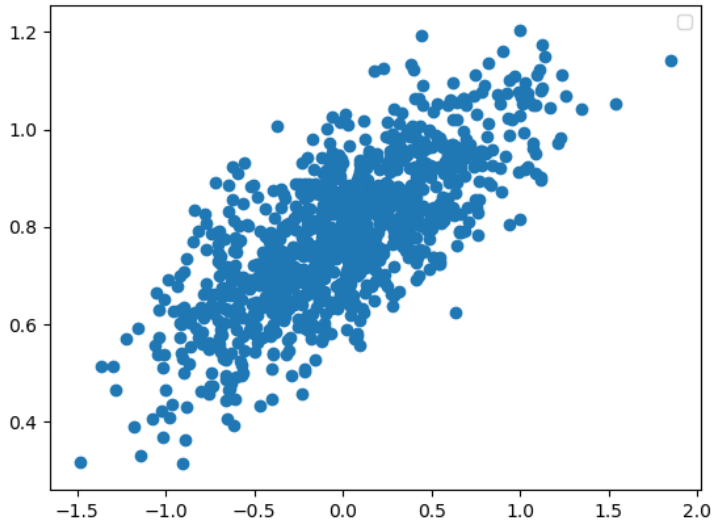
$y=ax+b$

$a = ?$ $b = ?$

Regresja liniowa

Dla danych $\{(x_1,y_1), \ldots, (x_N,y_N)\}$ zdefiniujmy błąd:

$$E(a,b) = \sum_{n=1}^{N} (y_n - (ax_n + b))^2$$



Nasz cel to znalezienie wartości ***a*** i ***b*** dla których błąd jest najmniejszy.

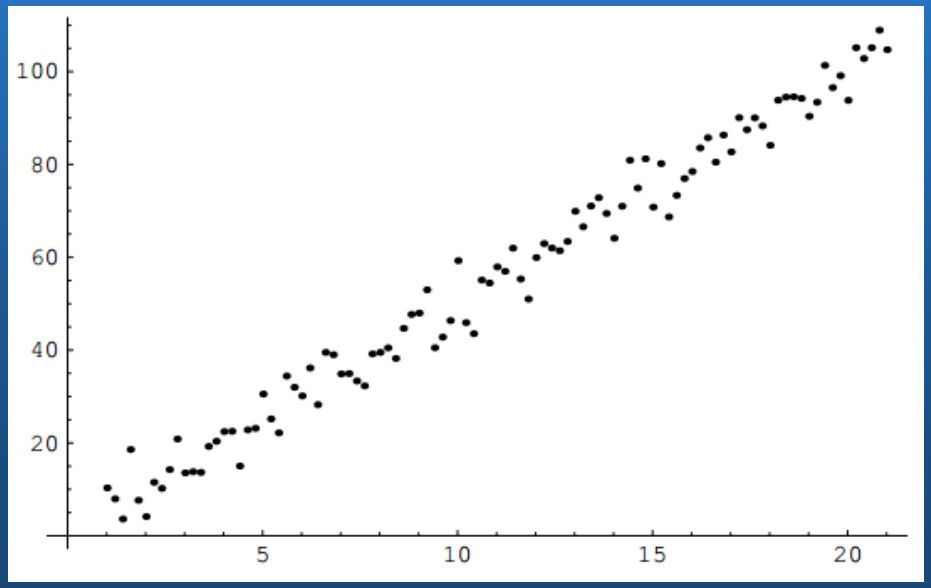
Metoda najmniejszych kwadratów

Metoda najmniejszych kwadratów służy do znalezienia krzywej najlepiej pasującej do danych.

Rozważmy metodę w przypadku linii prostej:

$$y = ax + b$$

Załóżmy, że dla $n \in \{1,\ldots,N\}$ pary (x_n, y_n) wyglądają następująco:



Metoda najmniejszych kwadratów

Dla danych $\{(x_1,y_1), \ldots ,(x_N,y_N)\}$ zdefiniujmy błąd:

$$E(a,b) = \sum_{n=1}^{N} (y_n - (ax_n + b))^2$$

Nasz cel to znalezienie wartości *a* i *b* dla których błąd jest najmniejszy.

Oznacza to, że musimy znaleźć wartości *(a, b)* takie, że:

$$\frac{\partial E}{\partial a} = 0, \quad \frac{\partial E}{\partial b} = 0.$$

Metoda najmniejszych kwadratów

Obliczmy pochodne *E(a,b)*:

$$\frac{\partial E}{\partial a} = \sum_{n=1}^{N} 2\left(y_n - (ax_n + b)\right) \cdot (-x_n)$$

$$\frac{\partial E}{\partial b} = \sum_{n=1}^{N} 2\left(y_n - (ax_n + b)\right) \cdot 1.$$

Zatem otrzymujemy:

$$\sum_{n=1}^{N} \left(y_n - (ax_n + b)\right) \cdot x_n = 0$$

$$\sum_{n=1}^{N} \left(y_n - (ax_n + b)\right) = 0.$$

Metoda najmniejszych kwadratów

Możemy zapisać to inaczej:

$$\begin{aligned}\left(\sum_{n=1}^{N} x_n^2\right) a + \left(\sum_{n=1}^{N} x_n\right) b &= \sum_{n=1}^{N} x_n y_n \\ \left(\sum_{n=1}^{N} x_n\right) a + \left(\sum_{n=1}^{N} 1\right) b &= \sum_{n=1}^{N} y_n.\end{aligned}$$

Otrzymujemy zatem równanie macierzowe:

$$\begin{pmatrix} \sum_{n=1}^{N} x_n^2 & \sum_{n=1}^{N} x_n \\ \sum_{n=1}^{N} x_n & \sum_{n=1}^{N} 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_{n=1}^{N} x_n y_n \\ \sum_{n=1}^{N} y_n \end{pmatrix}$$

Metoda najmniejszych kwadratów

Zakładając, że macierz jest odwracalna otrzymujemy

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_{n=1}^{N} x_n^2 & \sum_{n=1}^{N} x_n \\ \sum_{n=1}^{N} x_n & \sum_{n=1}^{N} 1 \end{pmatrix}^{-1} \begin{pmatrix} \sum_{n=1}^{N} x_n y_n \\ \sum_{n=1}^{N} y_n \end{pmatrix}$$

Oznaczmy tą macierz przez *M*. Policzmy jej wyznacznik:

$$\det M = \sum_{n=1}^{N} x_n^2 \cdot \sum_{n=1}^{N} 1 - \sum_{n=1}^{N} x_n \cdot \sum_{n=1}^{N} x_n.$$

$$\overline{x} = \frac{1}{N} \sum_{n=1}^{N} x_n$$

Wariancja

$$\det M = N \sum_{n=1}^{N} x_n^2 - (N\overline{x})^2 = N^2 \left(\frac{1}{N} \sum_{n=1}^{N} x_n^2 - \overline{x}^2 \right) = N^2 \cdot \frac{1}{N} \sum_{n=1}^{N} (x_n - \overline{x})^2$$

Zatem dopóki wszystkie x_n nie są równe, *detM* będzie różny od 0 i M będzie odwracalna!

Metoda najmniejszych kwadratów

Przykład

Rozważmy następujace dane:

X	1	2	3	4	5	6	7	8	9	10
Y	0.5	5	4.1	8.2	5.9	8.1	12.4	12	12.7	18.3

Wówczas otrzymujemy (N=10):

$\sum_{n=1}^{N} x_n^2 =$ 385 $\sum_{n=1}^{N} x_n =$ 55 $\sum_{n=1}^{N} x_n y_n =$ 613,8

$\sum_{n=1}^{N} x_n =$ 55 $\sum_{n=1}^{N} 1 =$ 10 $\sum_{n=1}^{N} y_n =$ 87,2

Metoda najmniejszych kwadratów

Przykład

Macierz M:

$$\begin{bmatrix} 385 & 55 \\ 55 & 10 \end{bmatrix}$$

Macierz odwrotna M^{-1}:

$$\begin{bmatrix} 0{,}012121 & -0{,}06667 \\ -0{,}06667 & 0{,}466667 \end{bmatrix}$$

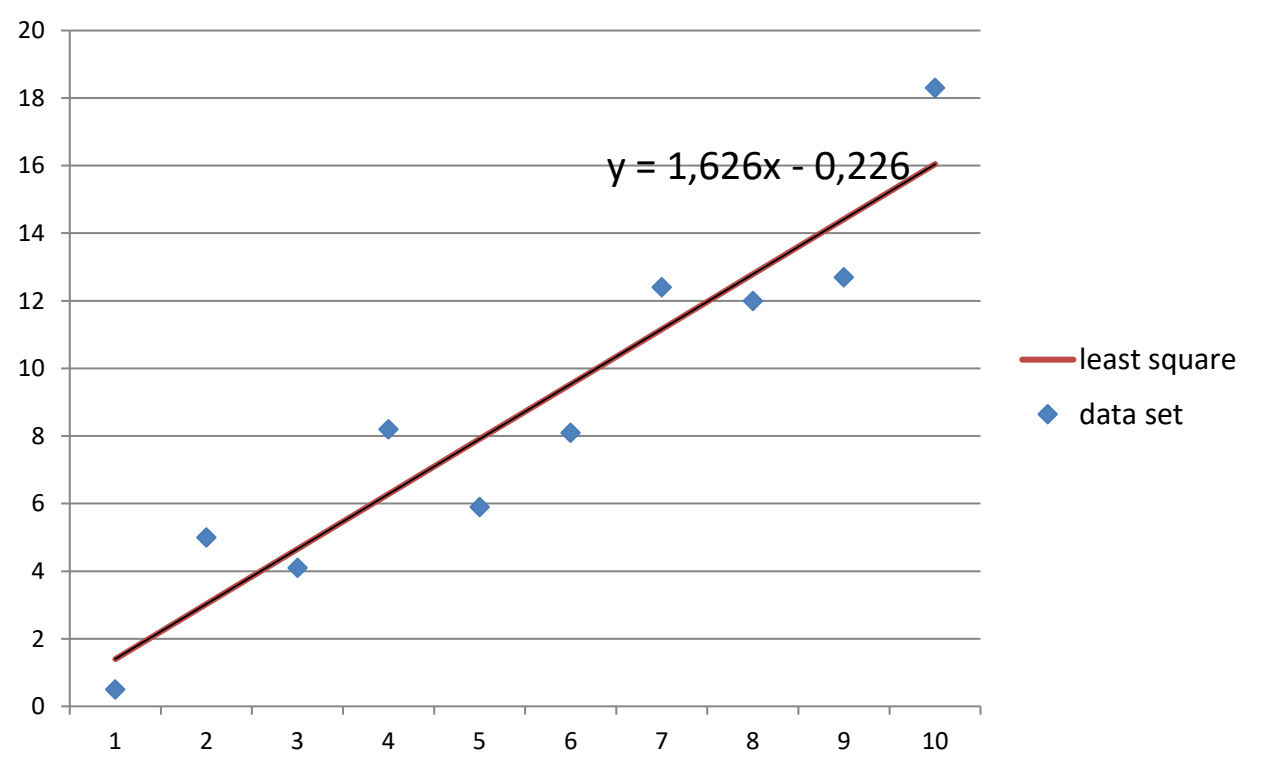
Zatem:

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 0{,}012121 & -0{,}06667 \\ -0{,}06667 & 0{,}466667 \end{bmatrix} \cdot \begin{bmatrix} 613{,}8 \\ 87{,}2 \end{bmatrix} = \begin{bmatrix} 1{,}626667 \\ -0{,}22667 \end{bmatrix}$$

Metoda najmniejszych kwadratów

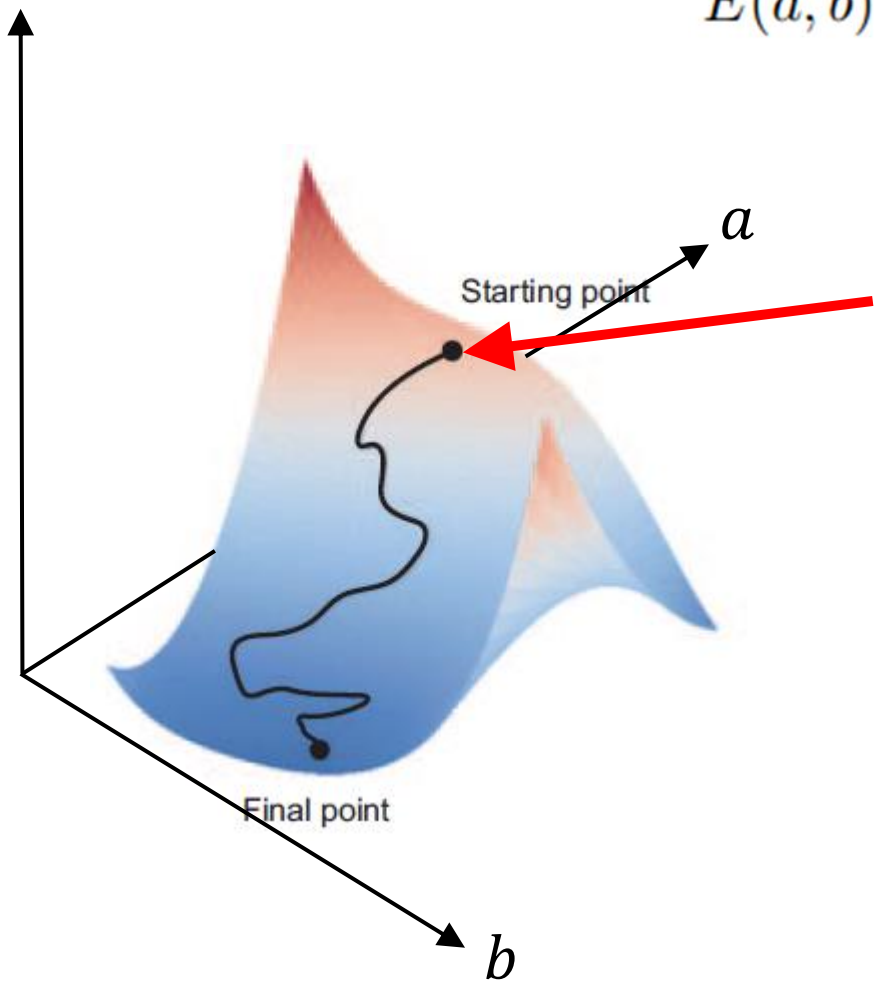
Przykład

Otrzymaliśmy prostą:



Inne rozwiązanie?

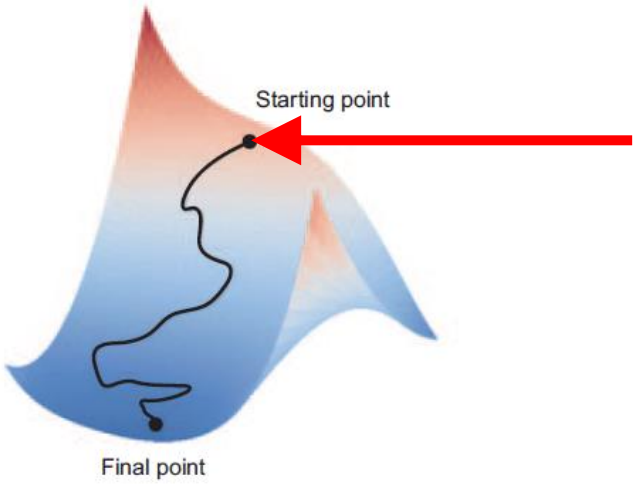
$$E(a,b) = \sum_{n=1}^{N} (y_n - (ax_n + b))^2$$

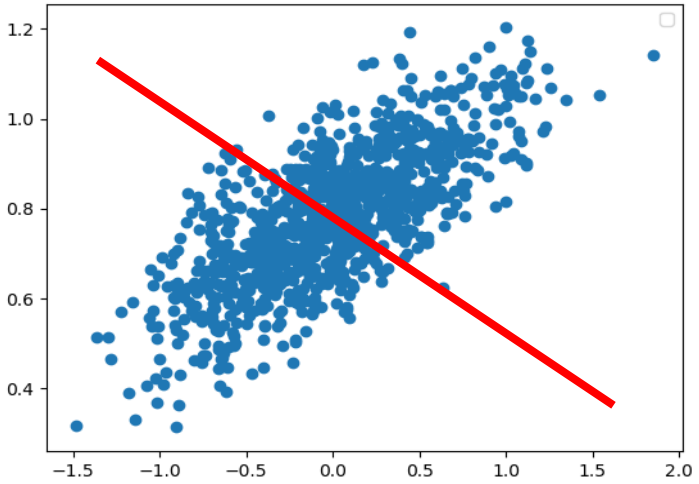


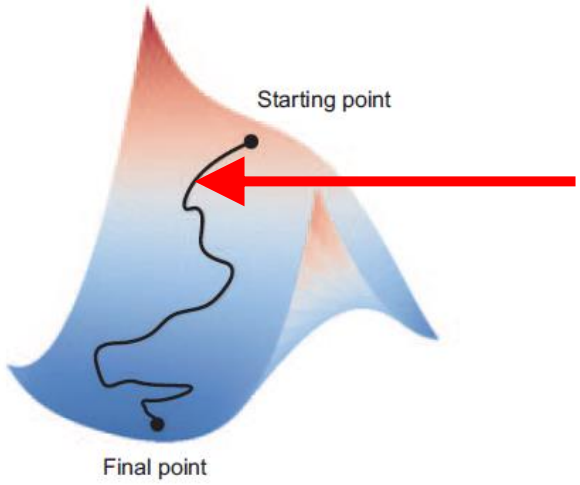
Wartość błędu $E(a,b)$ dla początkowych wartości parametrów a i b.

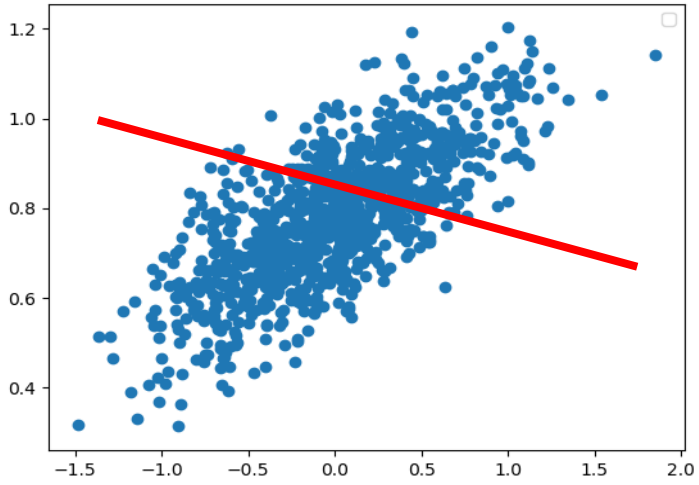
Czy możemy modyfikować parametry a i b w taki sposób, że wartość błędu będzie się przesuwała w kierunku minimum funkcji $E(a,b)$?

Inne rozwiązanie?

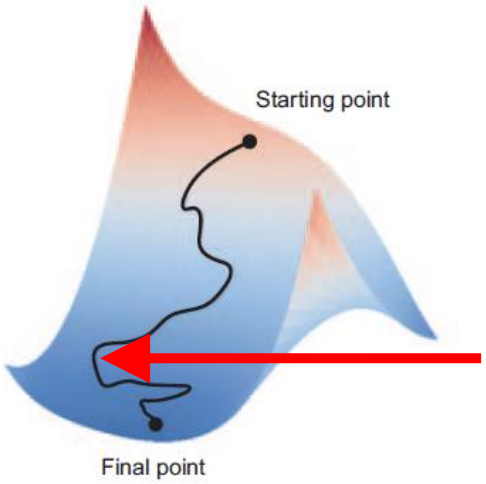


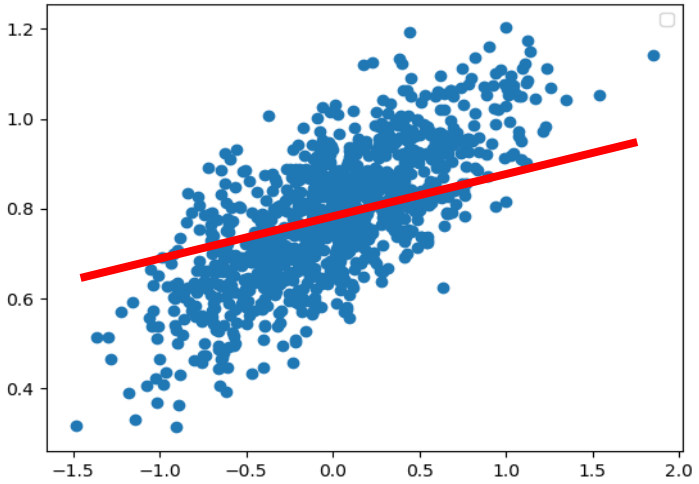


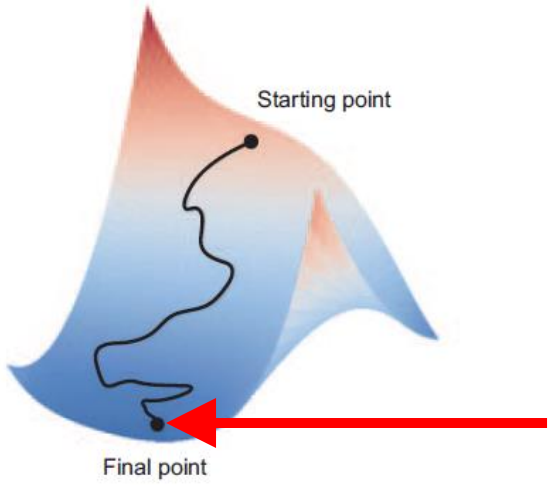


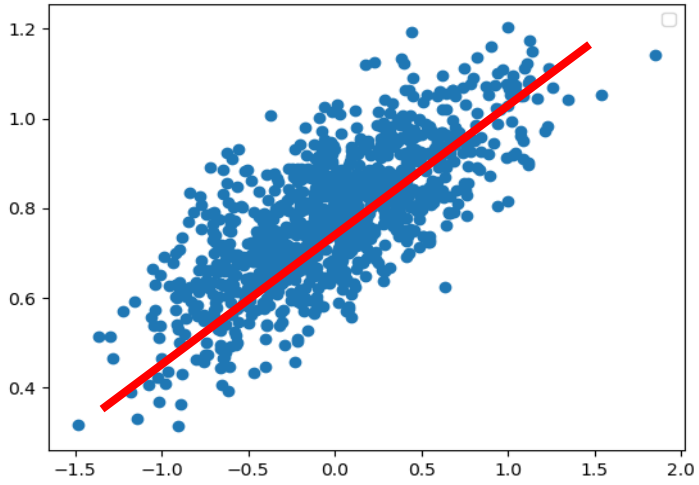


Inne rozwiązanie?



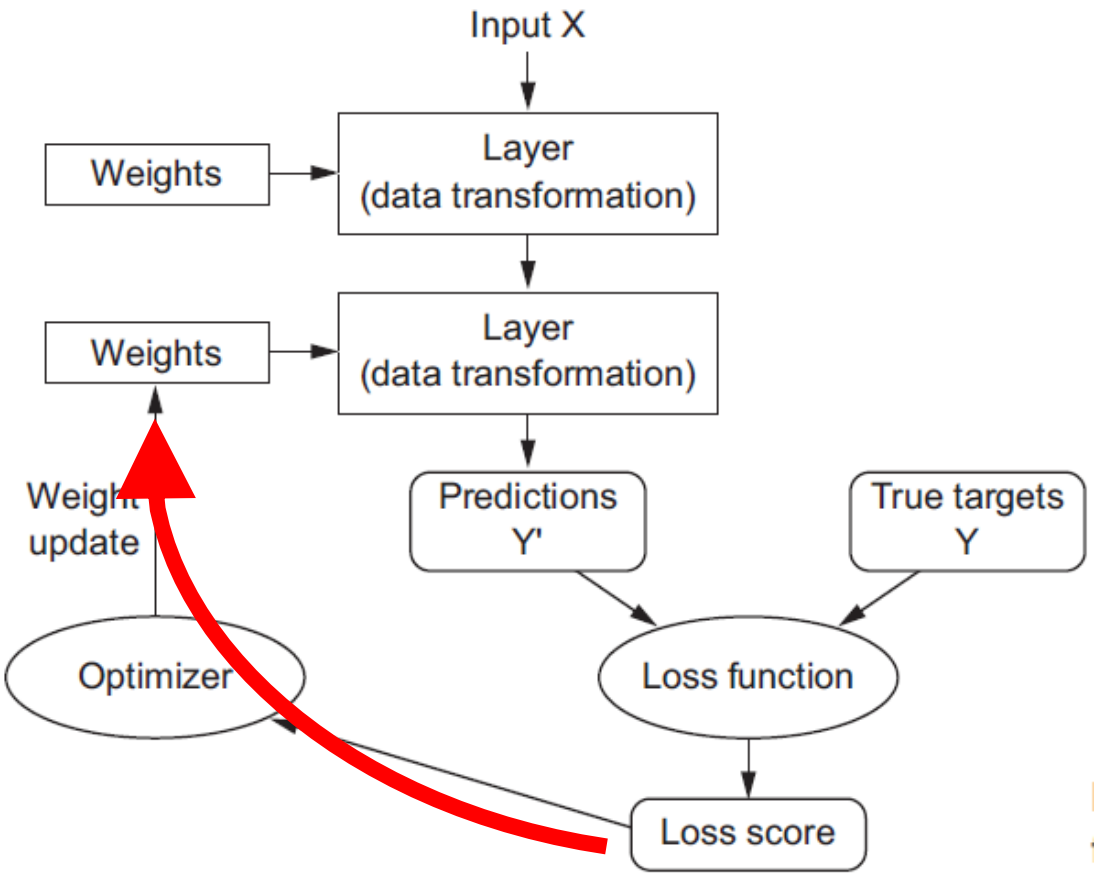






Uczenie maszynowe - idea

W oparciu o wartość policzonego błędu modyfikowane są wagi.



Uczenie maszynowe - idea

Uczenie odbywa się w następującej pętli treningowej:

1. Wybierz partię próbek x i odpowiednich celów y.
2. Podaj na wejście sieci x (krok nazywany *forward pass*), aby uzyskać prognozy y_{pred}.
3. Oblicz stratę sieci czyli błąd między y_{pred} i y.
4. Zaktualizuj wszystkie wagi sieci w sposób, który (nieznacznie) zmniejsza błąd.
5. Jeżeli to konieczne (błąd jest nadal duży) – wróć do punktu 1.

Uczenie maszynowe - idea

W jaki sposób aktualizować wagi sieci tak aby błąd ulegał zmniejszeniu?

Rozwiązanie naiwne

- Modyfikujemy tylko jedną wybraną wagę.
- Zmieniamy nieznacznie jej wartość i po każdej zmianie sprawdzamy czy błąd uległ zwiększeniu czy zmniejszeniu.

Przykład

Błąd sieci wynosi 0.5. Wybrana waga ma wartość 0.3. Jeżeli zmienimy wartość na 0.35, błąd wyniesie 0.6. Jeżeli zmienimy na 0.2, błąd wyniesie 0.4. A zatem nowa wartość wagi wynosi 0.2.

Uczenie maszynowe - idea

Metoda koszmarnie nieefektywne bo:

- Zmiana jednego parametru sieci wymaga dwukrotnego przepuszczenia x przez sieć.
- Parametrów, które musimy zmieniać są często miliony.

Jakie inne rozwiązanie?

Z pomocą przychodzi nam matematyka!

Uczenie maszynowe - idea

Zauważmy, że:

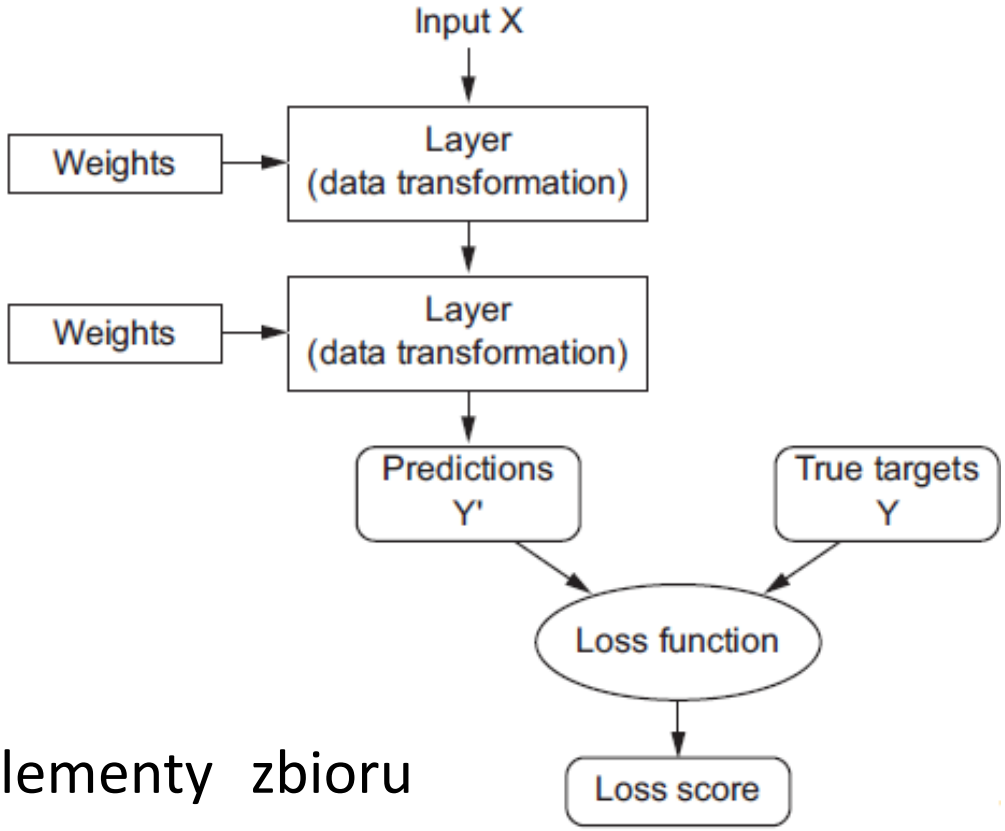
$y_{pred} = f_1(W,x)$

$loss = f_2(y_{pred}, y)$

W jest tensorem wag.

Czyli:

$loss = f_2(f_1(W,x), y)$



Ponieważ x i y jako elementy zbioru treningowego są stałe zatem:

loss = f(W)

$$E(a,b) = \sum_{n=1}^{N} (y_n - (ax_n + b))^2$$

Gradient funkcji

Przypomnijmy sobie zatem definicję gradientu funkcji:

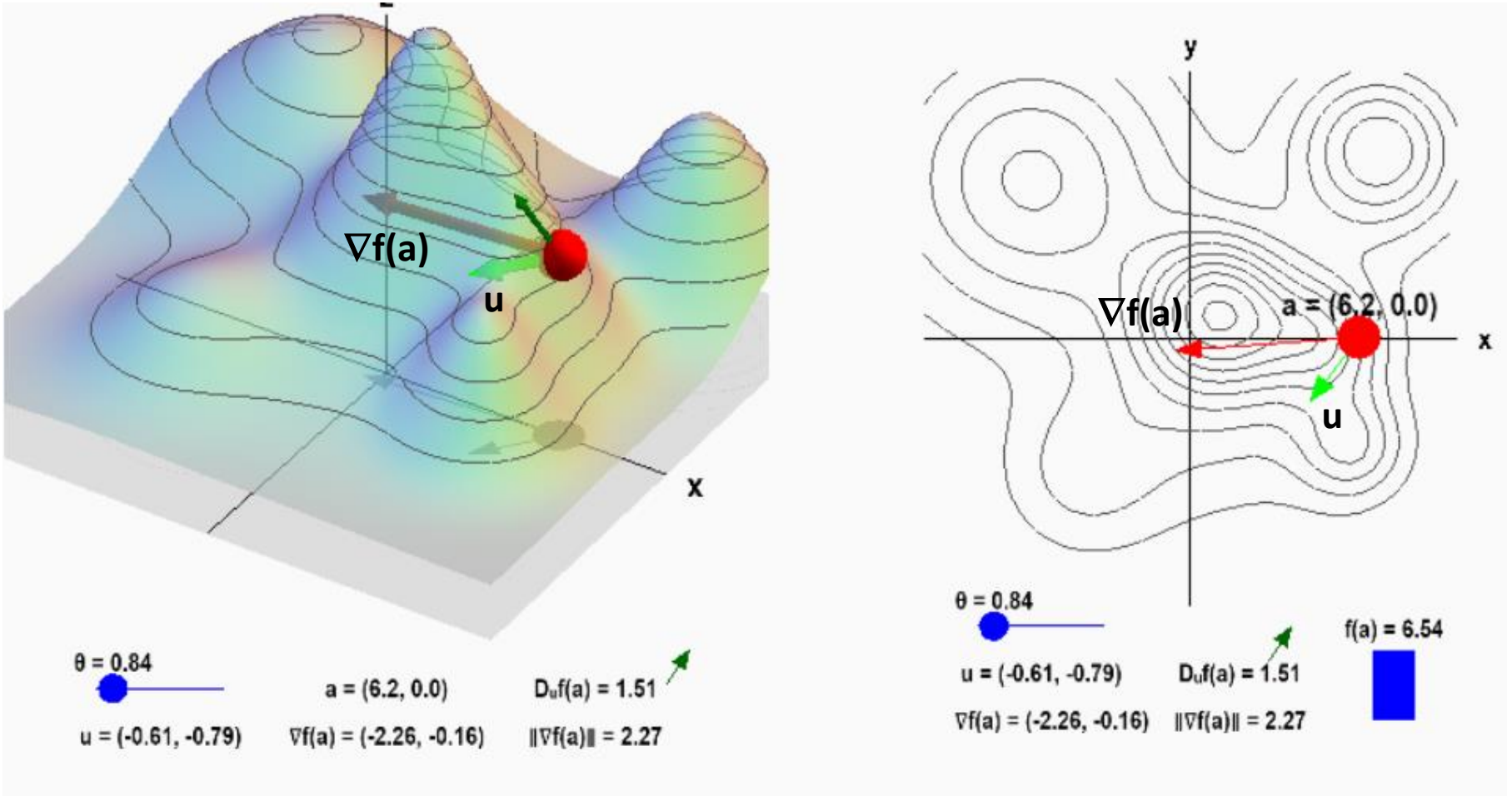
Gradient (lub gradientowe pole wektorowe) funkcji skalarnej $f(x_1, \ldots, x_n)$ oznaczany ∇f, gdzie ∇ (nabla) to wektorowy operator różniczkowy nazywany nabla. Innym oznaczeniem gradientu f jest $\operatorname{grad} f$.

W układzie współrzędnych kartezjańskich gradient jest wektorem, którego składowe są pochodnymi cząstkowymi funkcji f. Gradient definiuje się jako pewne pole wektorowe. W układzie współrzędnych kartezjańskich składowe gradientu funkcji f są pochodnymi cząstkowymi tej funkcji tzn.

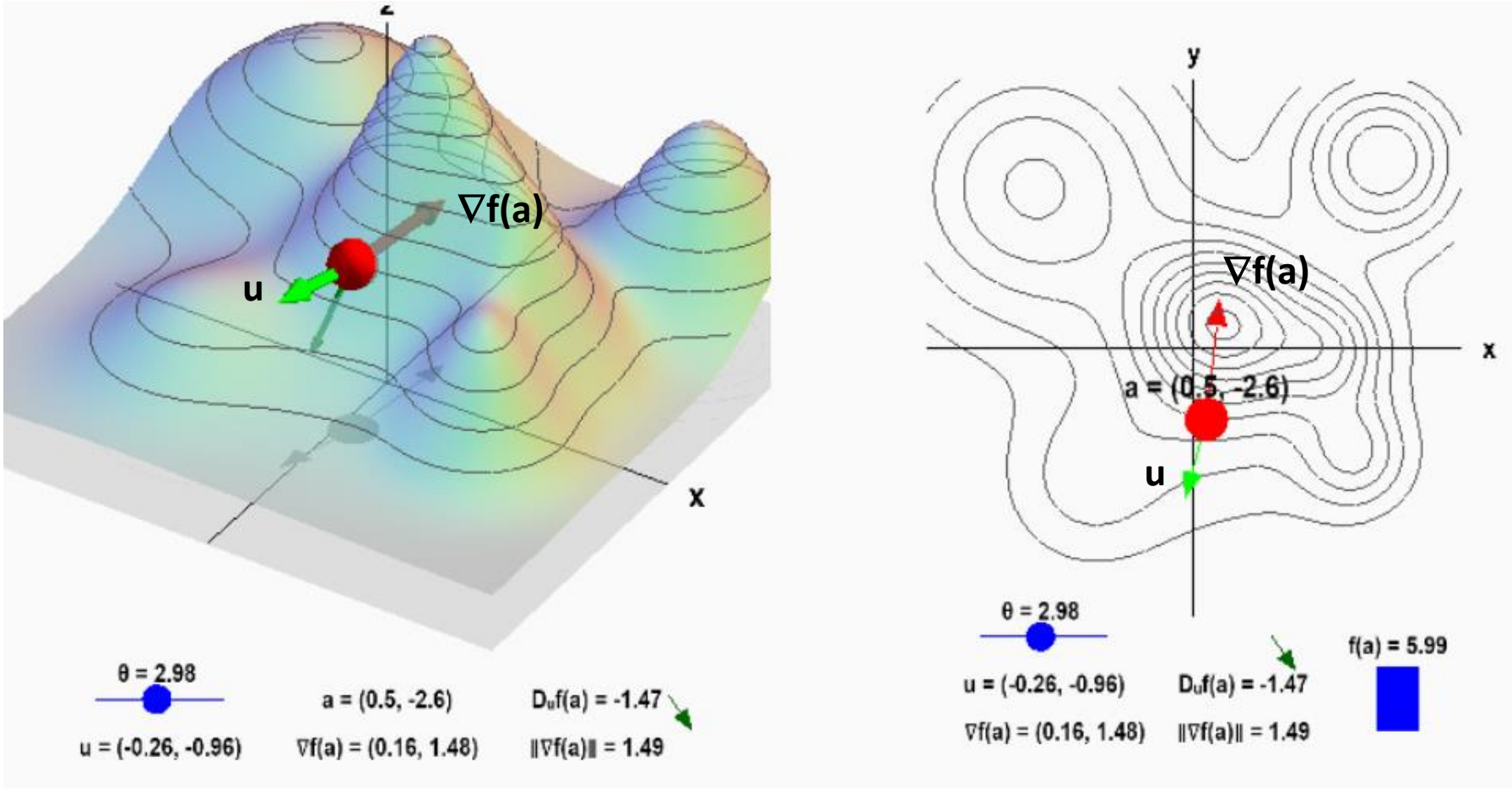
$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right].$$

Wektor gradientu pokazuje kierunek największego wzrostu funkcji w danym punkcie!

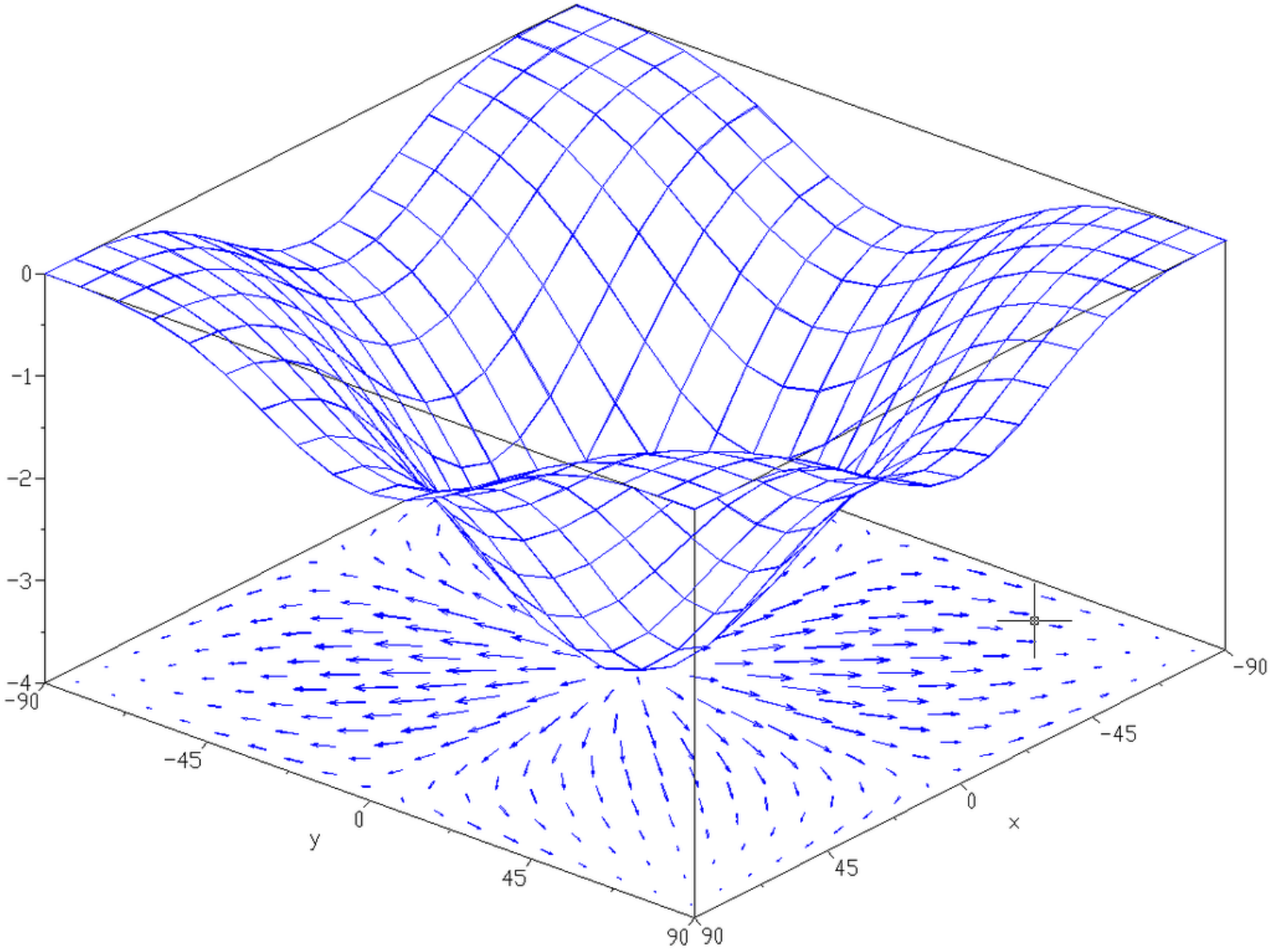
Gradient funkcji



Gradient funkcji



Gradient funkcji



Gradient funkcji

Przykład

Funkcja: $f(x, y) = x^2 y.$

Policzmy gradient: $\nabla f(3, 2)$

Z definicji: $\nabla f = \left[\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right]$

Pochodne cząstkowe:

$$\frac{\partial f}{\partial x}(x, y) = 2xy \qquad \frac{\partial f}{\partial y}(x, y) = x^2$$

$$\frac{\partial f}{\partial x}(3, 2) = 12 \qquad \frac{\partial f}{\partial y}(3, 2) = 9$$

Optymalizacja gradientowa

Wróćmy do funkcji błędu:

$$\text{loss} = f(W) \qquad (*)$$

Funkcja ta jest zwykle funkcją bardzo wielu zmiennych (bo sieć ma bardzo dużo parametrów).

$$\text{loss} = f(w_1, w_2, \ldots, w_n)$$

UWAGA: 'W' w powyższej formule (*) to pewien tensor.

Optymalizacja gradientowa

Funkcji błędu:

$$\text{loss} = f(W)$$

Przyjmijmy, że aktualna wartość W wynosi W_0.

W_0 jest tensorem zawierającym parametry sieci.

UWAGA: Gradient funkcji f w punkcie W_0 czyli:

$$\nabla f(W_0)$$

ma ten sam kształt co W_0.

Optymalizacja gradientowa

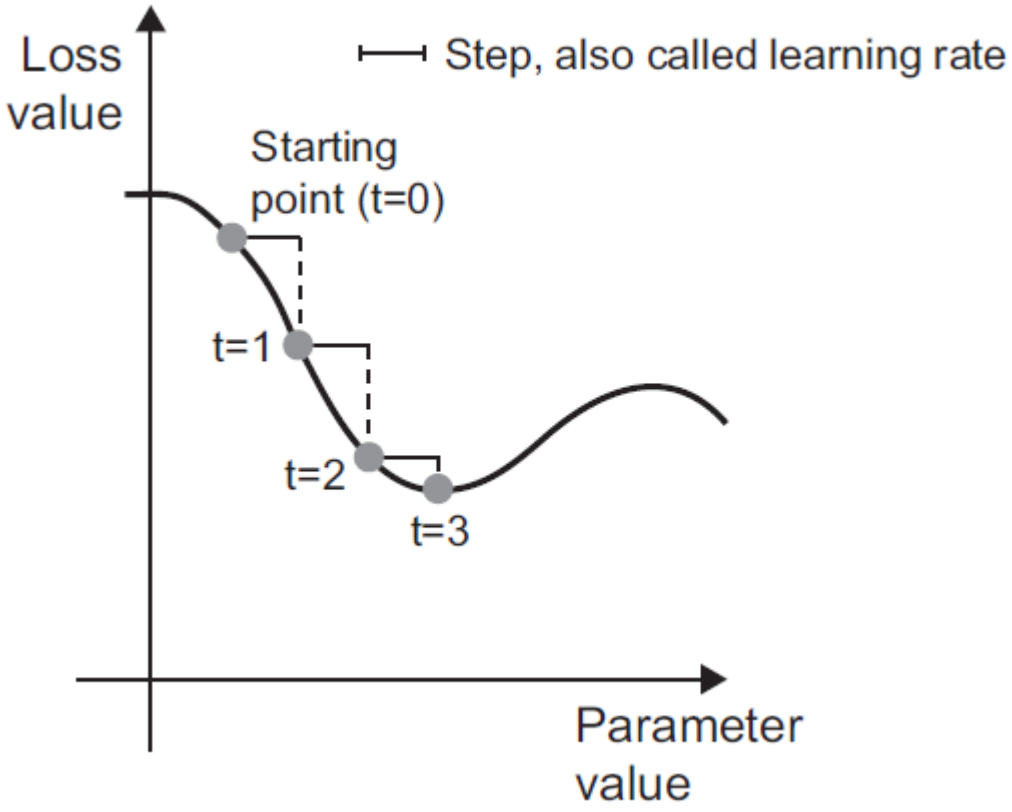
Ponieważ wektor gradientu pokazuje kierunek największego wzrostu funkcji w danym punkcie zatem wartość funkcji f(W) możemy zmniejszyć przesuwając się w kierunku przeciwnym do gradientu np:

$$W_1 = W_0 - \alpha \cdot \nabla f(W_0)$$

Przy czym α jest małym współczynnikiem uczenia. Jest on konieczny bo nie chcemy odejść zbyt mocno od W_0.

Optymalizacja gradientowa

W przypadku jednego parametru (wagi):



Mini-batch stochastic gradient descent

Uczenie odbywa się zatem w następującej pętli treningowej:

1. Wybierz partię próbek x i odpowiednich celów y.
2. Podaj na wejście sieci x (krok nazywany *forward pass*)), aby uzyskać prognozy y_{pred}.
3. Oblicz stratę sieci (czyli błąd) między y_{pred} i y.
4. Zmodyfikuj wszystkie wagi sieci w sposób, który nieznacznie zmniejsza błąd.
5. Jeżeli to konieczne (błąd jest nadal duży) – wróć do punktu 1.

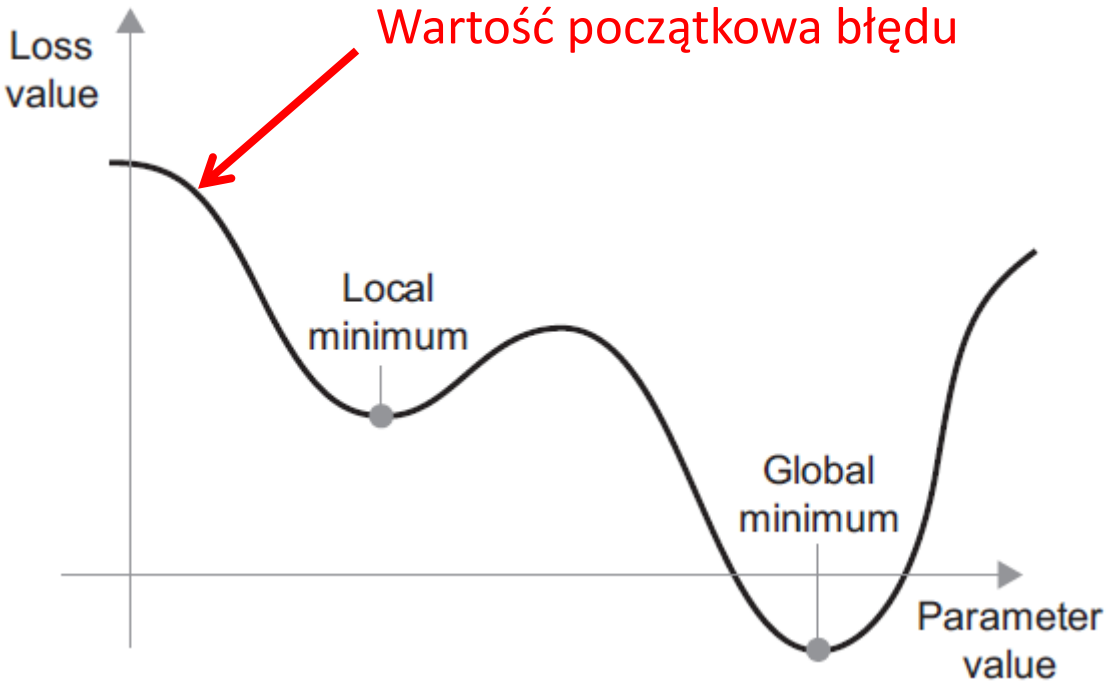
Mini-batch stochastic gradient descent

Uczenie odbywa się zatem w następującej pętli treningowej:

1. Wybierz partię próbek x i odpowiednich celów y (→ batch).
2. Podaj na wejście sieci x (krok nazywany *forward pass*)), aby uzyskać prognozy y_{pred}.
3. Oblicz stratę sieci (czyli błąd) między y_{pred} i y.
4. Oblicz gradient funkcji błędu f(W) i zmodyfikuj wszystkie wagi: $W_1 = W_0 - \alpha \cdot \nabla f(W_0)$
5. Jeżeli to konieczne (błąd jest nadal duży) – wróć do punktu 1.

Optymalizacja gradientowa

A co w takiej sytuacji?



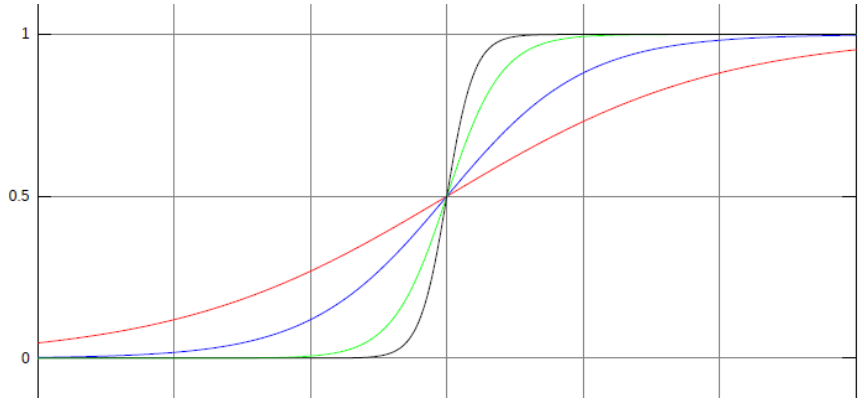
Model neuronu sigmoidalnego

Budowa tego neuronu jest podobna do budowy perceptronu. Różnica sprowadza się do funkcji aktywacji, która w tym przypadku jest funkcją sigmoidalną (unipolarną lub bipolarną).

Model neuronu sigmoidalnego

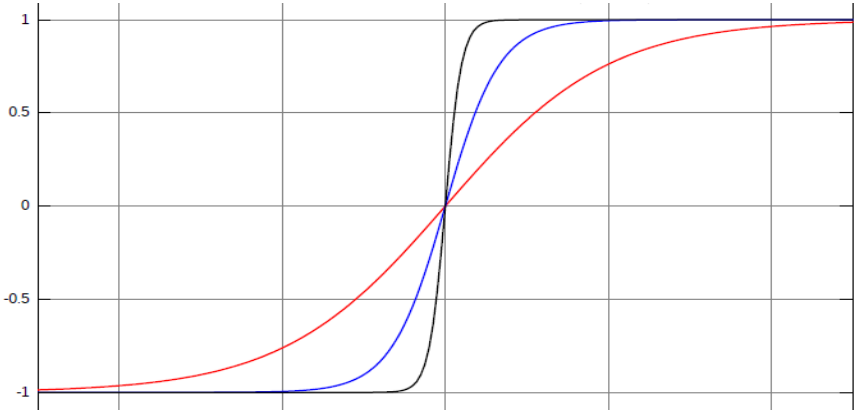
Funkcja unipolarna

$$f(x) = \frac{1}{1 + \exp(-\beta x)}$$

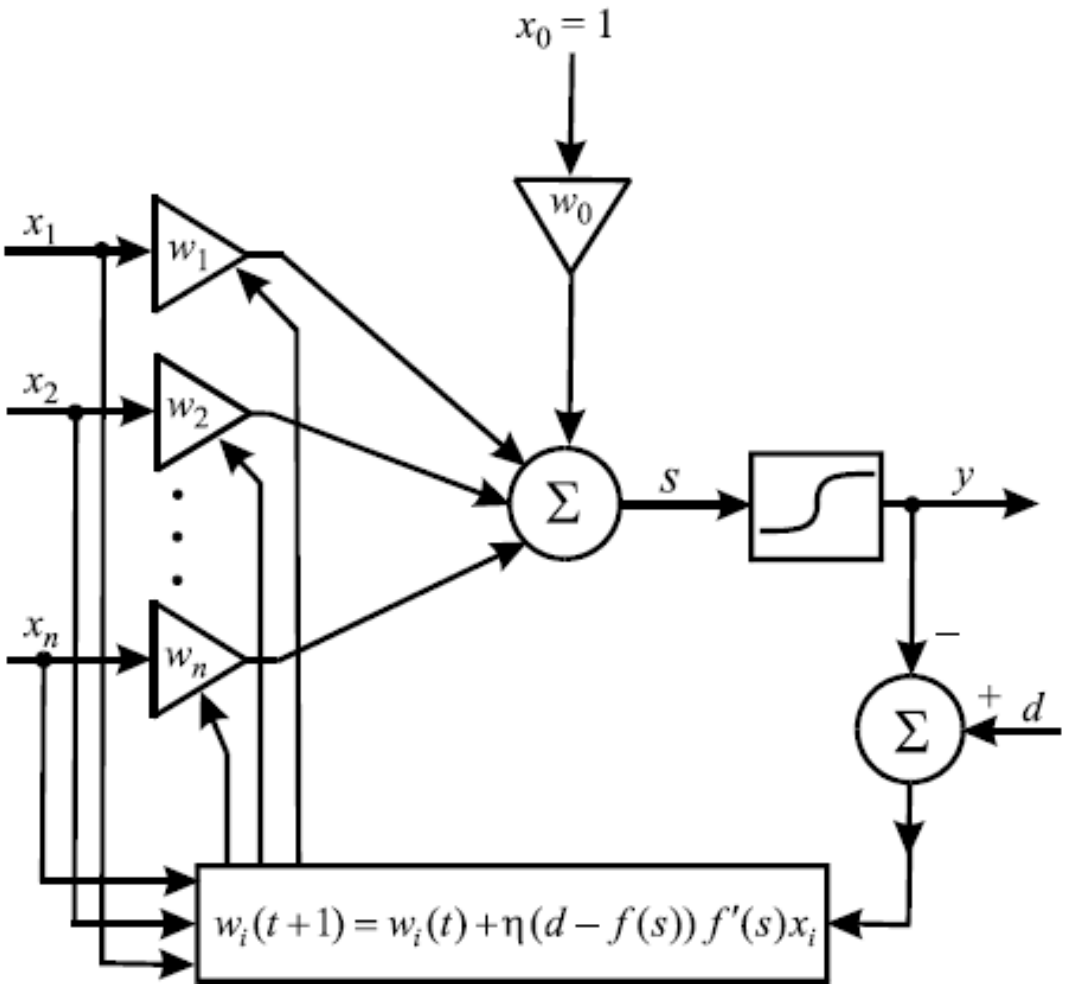


Funkcja bipolarna

$$f(x) = \frac{1 - \exp(-\beta x)}{1 + \exp(-\beta x)}$$



Model neuronu sigmoidalnego



Model neuronu sigmoidalnego

Ważną zaletą funkcji sigmoidalnych jest ich różniczkowalność.

Pochodne te łatwo obliczyć.

Funkcja unipolarna:

$$f'(x) = \beta f(x)(1 - f(x))$$

Funkcja bipolarna:

$$f'(x) = \beta\left(1 - f^2(x)\right)$$

Wyjście neuronu:

$$y(t) = f\left(\sum_{i=0}^{n} w_i(t)\, x_i(t)\right)$$

Model neuronu sigmoidalnego

Uczenie neuronu polega na minimalizacji błędu kwadratowego:

$$Q(\mathbf{w}) = \frac{1}{2}\left[d - f\left(\sum_{i=0}^{n} w_i x_i\right)\right]^2$$

Funkcja $Q(\mathbf{w})$ jest różniczkowalna, ze względu na wagi a zatem możemy tutaj wykorzystać optymalizację gradientową.

Model neuronu sigmoidalnego

Wagi w neuronie sigmoidalnym modyfikujemy według wzoru (wykorzystujemy wektor gradientu) :

$$w_i(t+1) = w_i(t) - \eta \frac{\partial Q(w_i)}{\partial w_i}$$

gdzie η jest tzw. współczynnikiem uczenia.

$$\frac{\partial Q(w_i)}{\partial w_i} = \frac{\partial Q(w_i)}{\partial s} \cdot \frac{\partial s}{\partial w_i}$$

Łatwo obliczyć, że:

$$\frac{\partial s}{\partial w_i} = x_i$$

Model neuronu sigmoidalnego

Zatem:

$$\frac{\partial Q(w_i)}{\partial w_i} = \frac{\partial Q(w_i)}{\partial s} \cdot x_i$$

Łatwo sprawdzić, że:

$$\frac{\partial Q(w_i)}{\partial s} = -(d - f(s)) \cdot f'(s)$$

Wprowadźmy oznaczenie:

$$\delta = -(d - f(s)) \cdot f'(s)$$

Model neuronu sigmoidalnego

Zatem ostatecznie otrzymujemy następującą formułę na modyfikację wag:

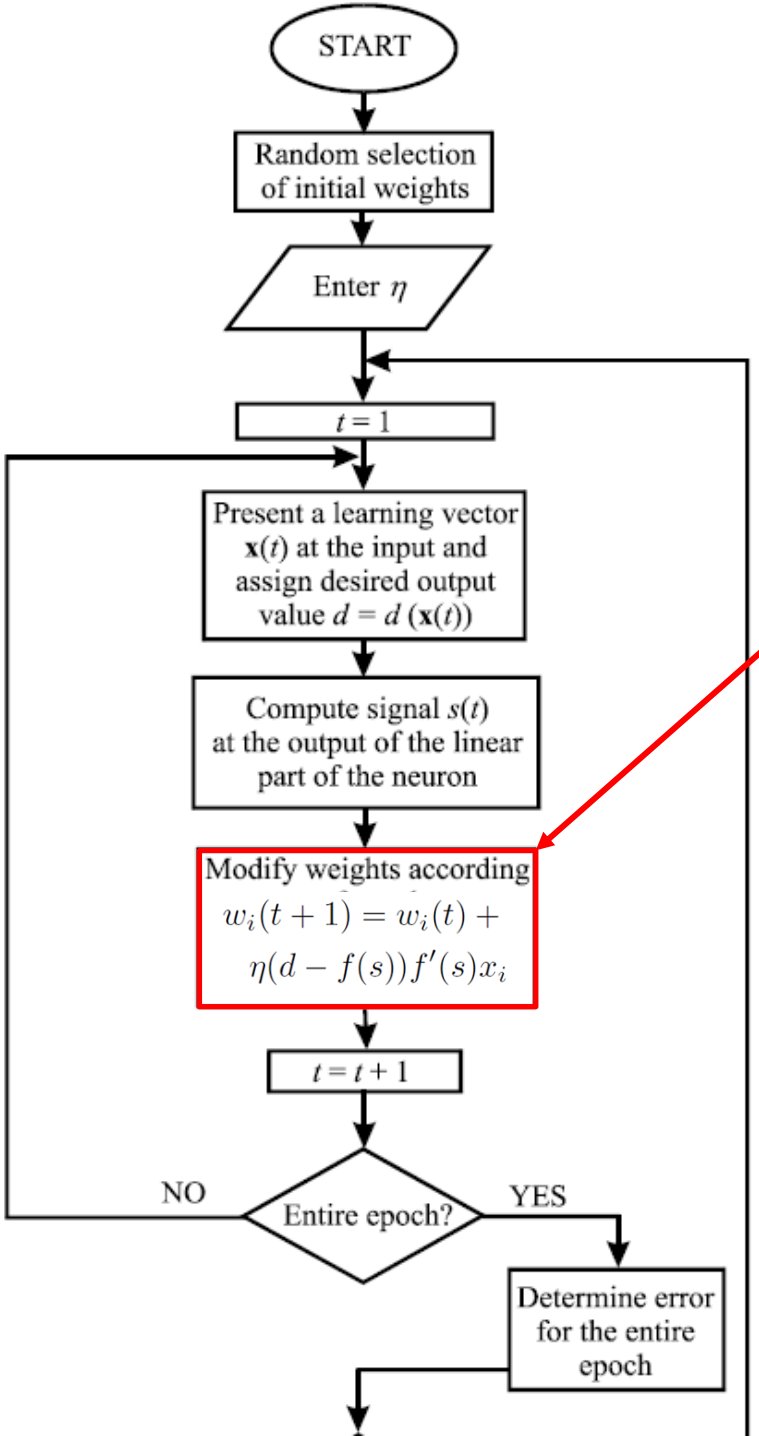
$$w_i(t+1) = w_i(t) - \eta\delta x_i =$$

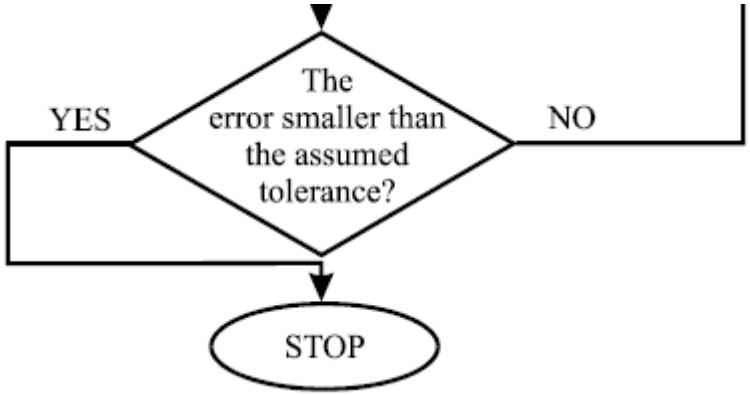
$$= w_i(t) + \eta(d - f(s))f'(s)x_i$$

gdzie η jest tzw. współczynnikiem uczenia.

Neuron sigmoidalny

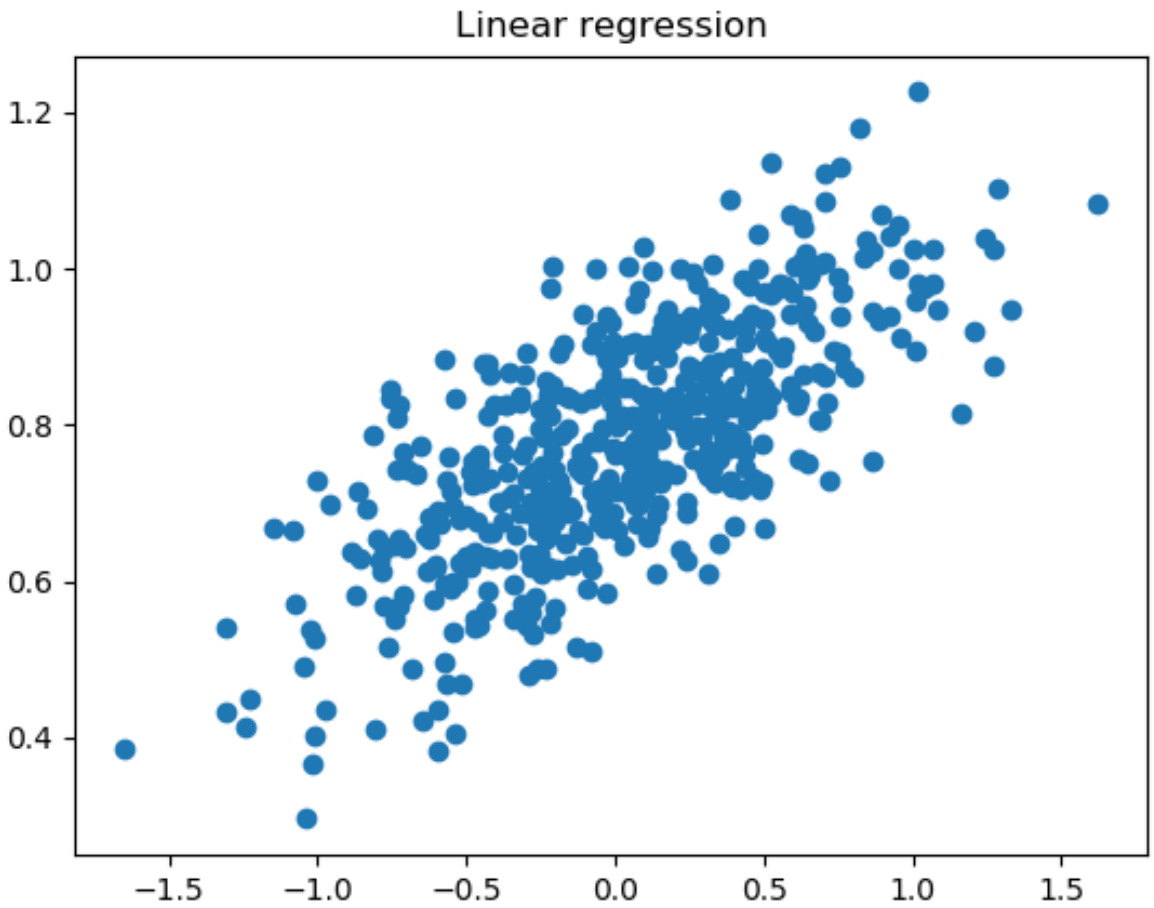
Wagi modyfikowane są w każdym przejściu pętli!!!





Regresja liniowa

Wygląd zbioru testowego:



Regresja liniowa

Dane, funkcja błędu i początkowe wartości parametrów:

```
real_x = np.array(x_point)
real_y = np.array(y_point)


def loss_fn(real_y, pred_y):
    return tf.reduce_mean((real_y - pred_y)**2)

import random
a = tf.Variable(random.random())
b = tf.Variable(random.random())
```

Regresja liniowa

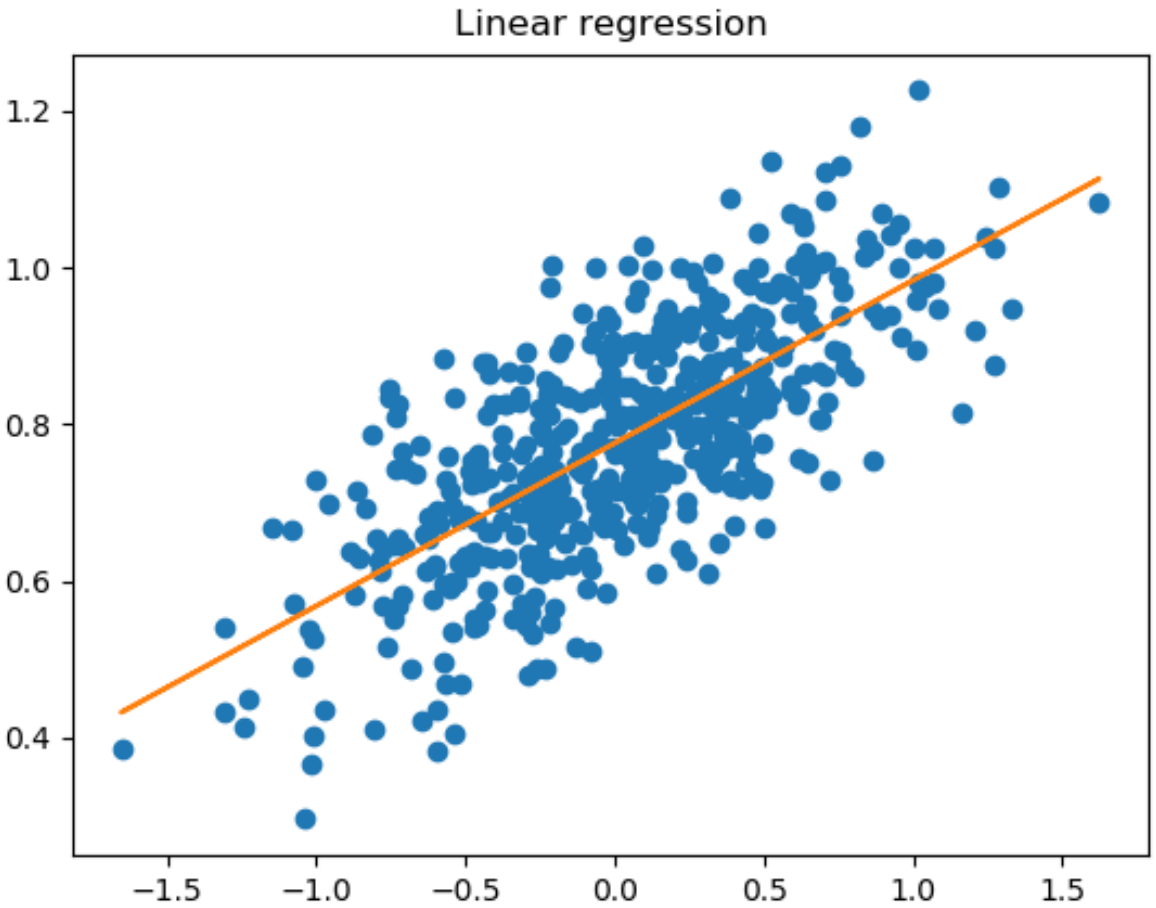
gradient

Modyfikacja parametrów (wag) i sesja: $W_1 = W_0 - \alpha \cdot \nabla f(W_0)$

```
Loss = []
epochs = 50
learning_rate = 0.2

for _ in range(epochs):
  with tf.GradientTape() as tape:
    pred_y = a * real_x + b
    loss = loss_fn(real_y, pred_y)
    Loss.append(loss.numpy())

  grad_a, grad_b = tape.gradient(loss,(a, b))

  a.assign_sub(learning_rate*grad_a)
  b.assign_sub(learning_rate*grad_b)
```

Regresja liniowa

Znaleziona linia prosta:

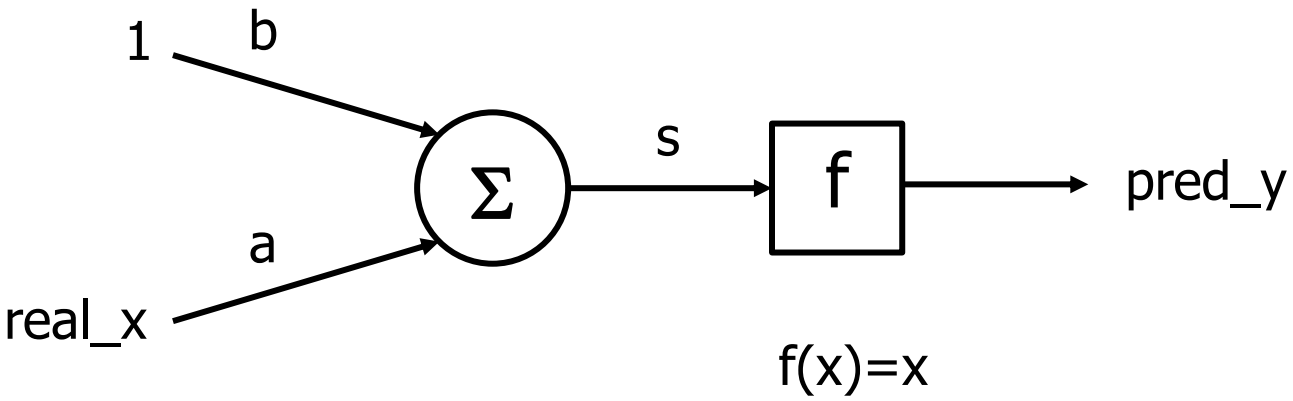


Sieć neuronowa?

Rozwiązujemy problem za pomocą jednego neuronu o jednym wejściu.

```
pred_y = a * real_x + b
```

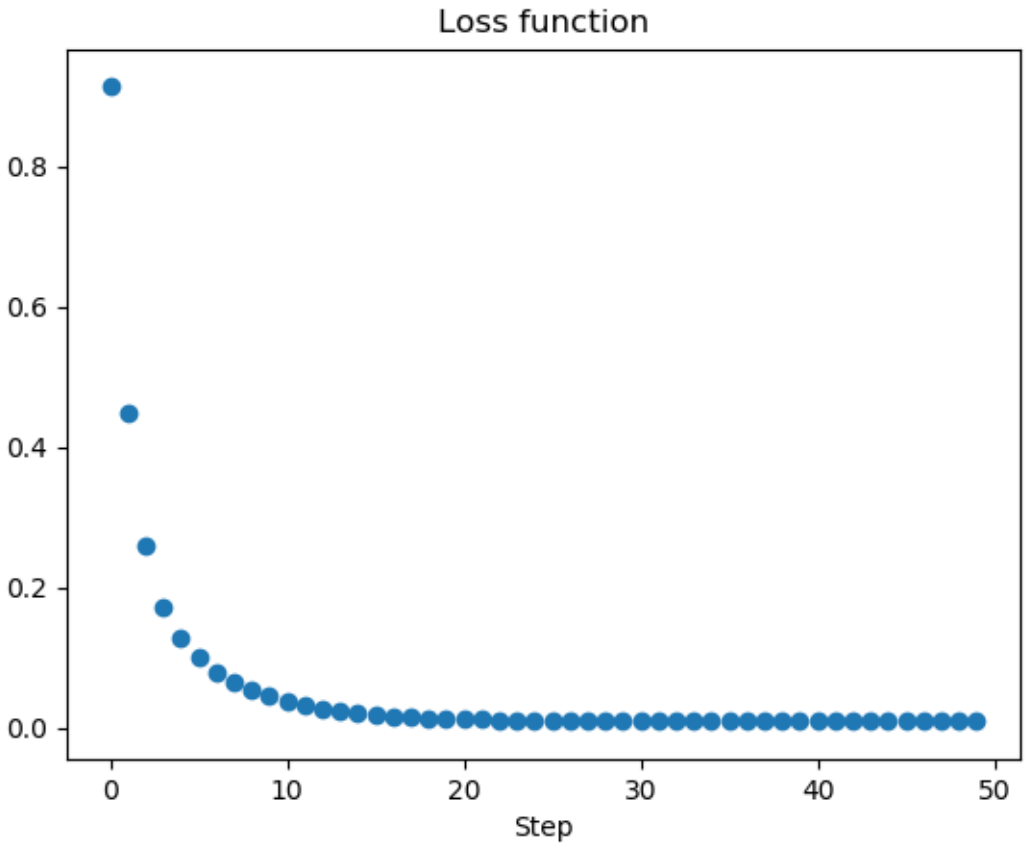
Wagami neuronu są parametry a i b.



Regresja liniowa

Zmiana błędu:

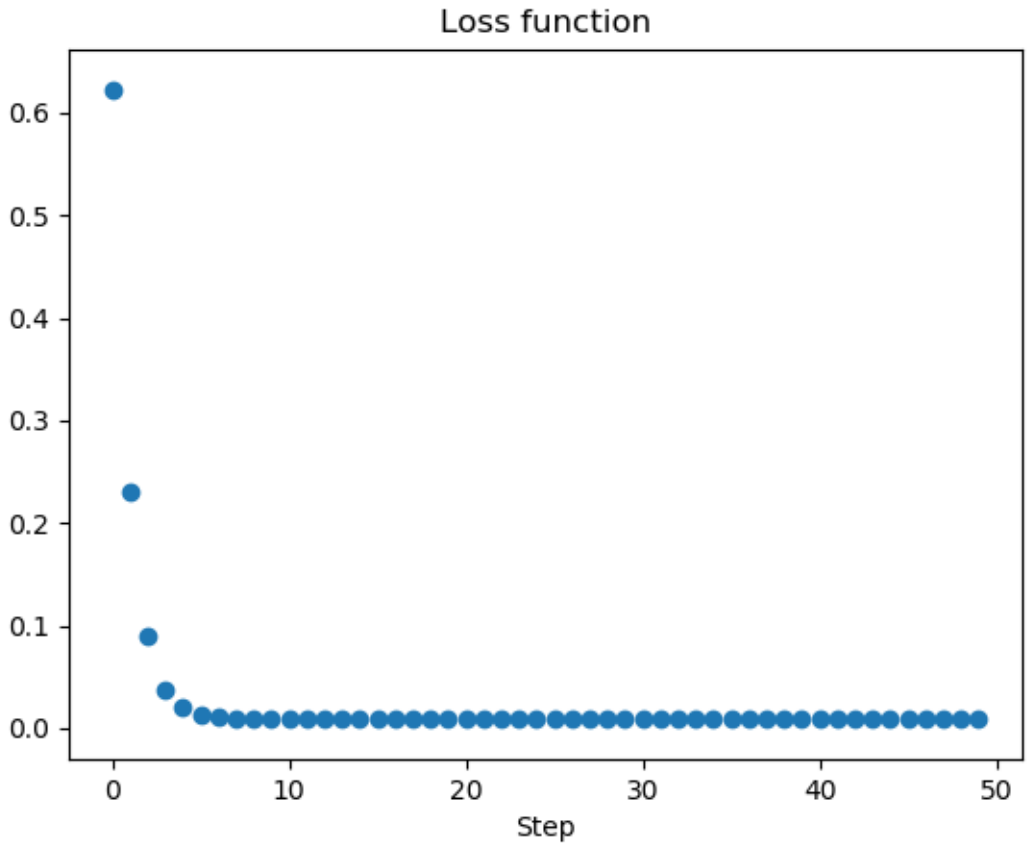
```
learning_rate = 0.2
steps = 50
```



Regresja liniowa

Zmiana błędu:

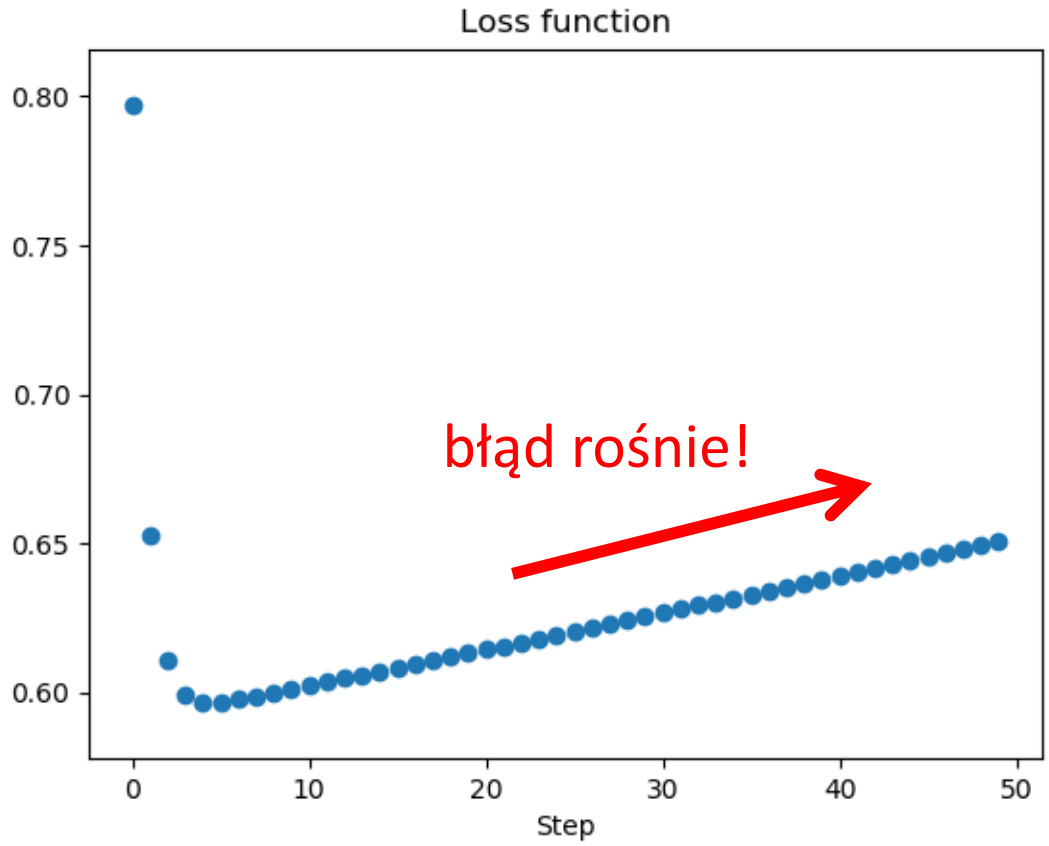
```
learning_rate = 0.8
steps = 50
```



Regresja liniowa

Zmiana błędu:

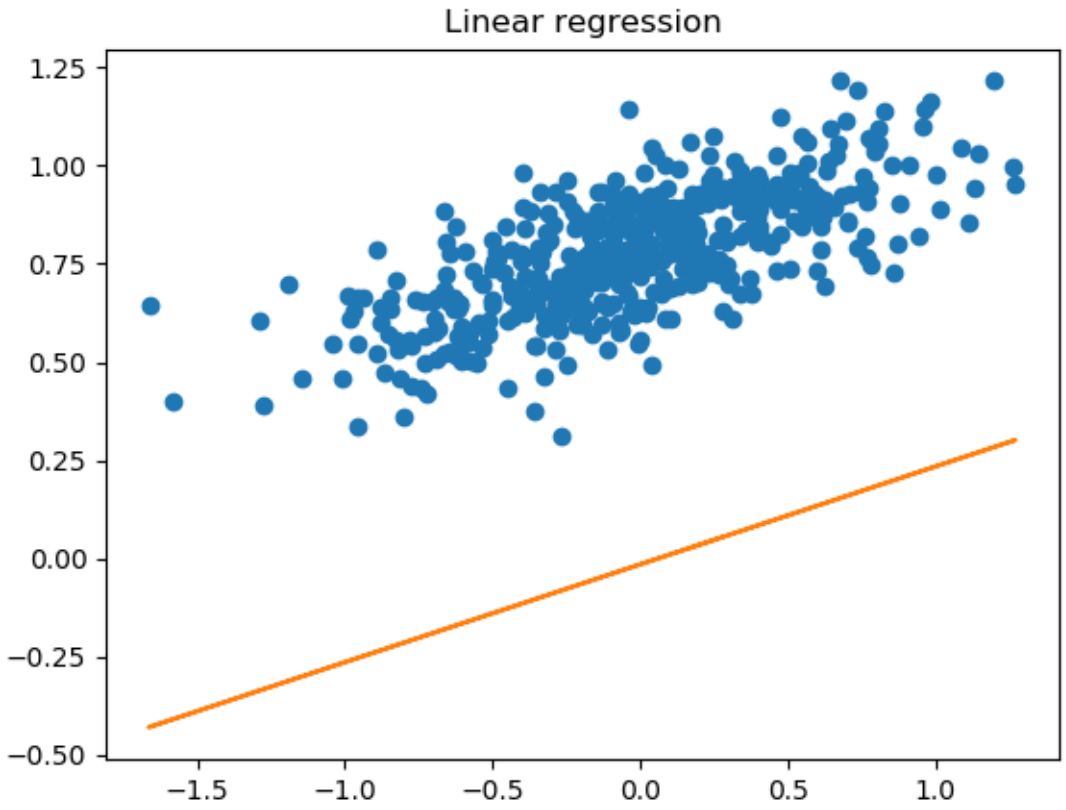
```
learning_rate = 1.0
steps = 50
```



Regresja liniowa

Zmiana błędu:

```
learning_rate = 1.0
steps = 50
```



Regresja liniowa

Obliczenie gradientu:

```
grad_a, grad_b = tape.gradient(loss,(a, b))
```

Modyfikacja wag (parametrów):

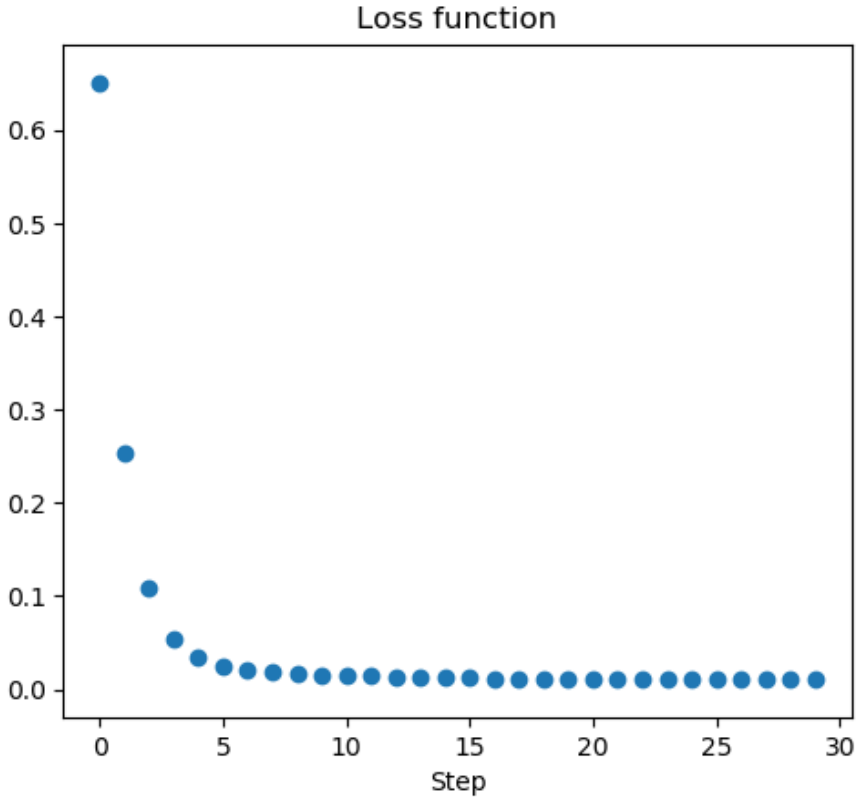
```
learning_rate = 1.0

a.assign_sub(learning_rate*grad_a)
b.assign_sub(learning_rate*grad_b)
```

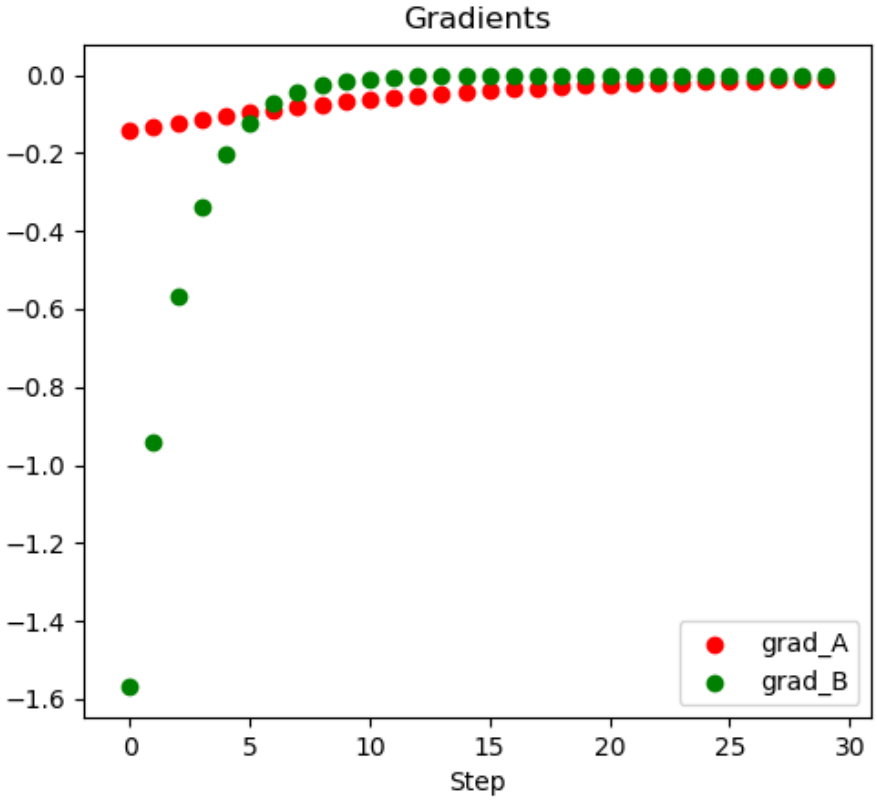
Jak zmieniają się wartości grad_a i grad_b?

Regresja liniowa

Błąd i gradienty:

```
learning_rate = 0.2
steps = 30
```

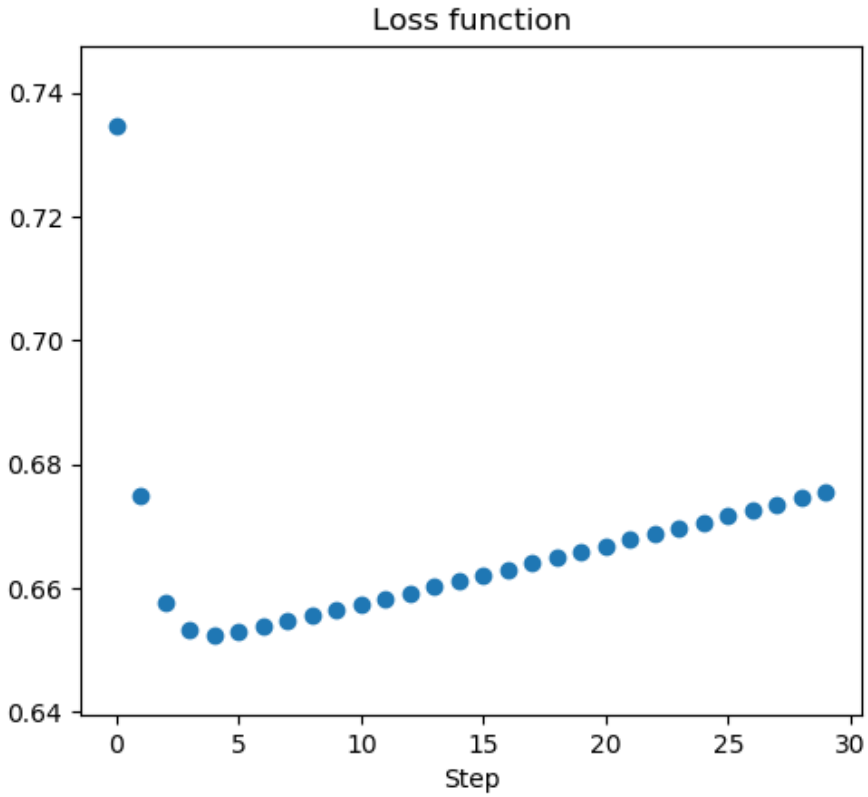


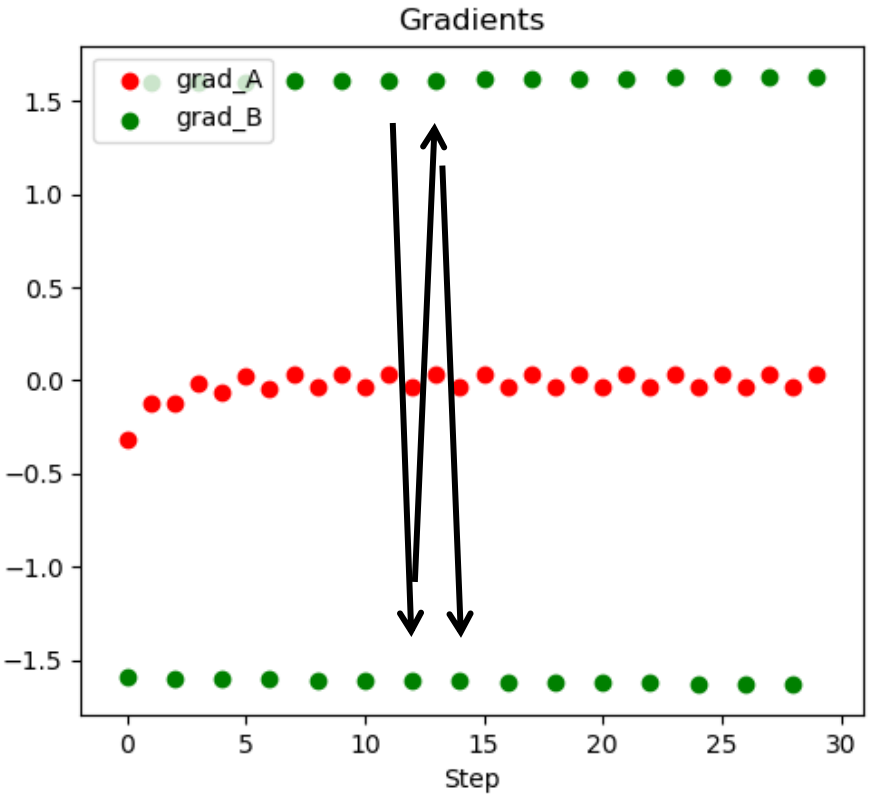


Regresja liniowa

Błąd i gradienty:

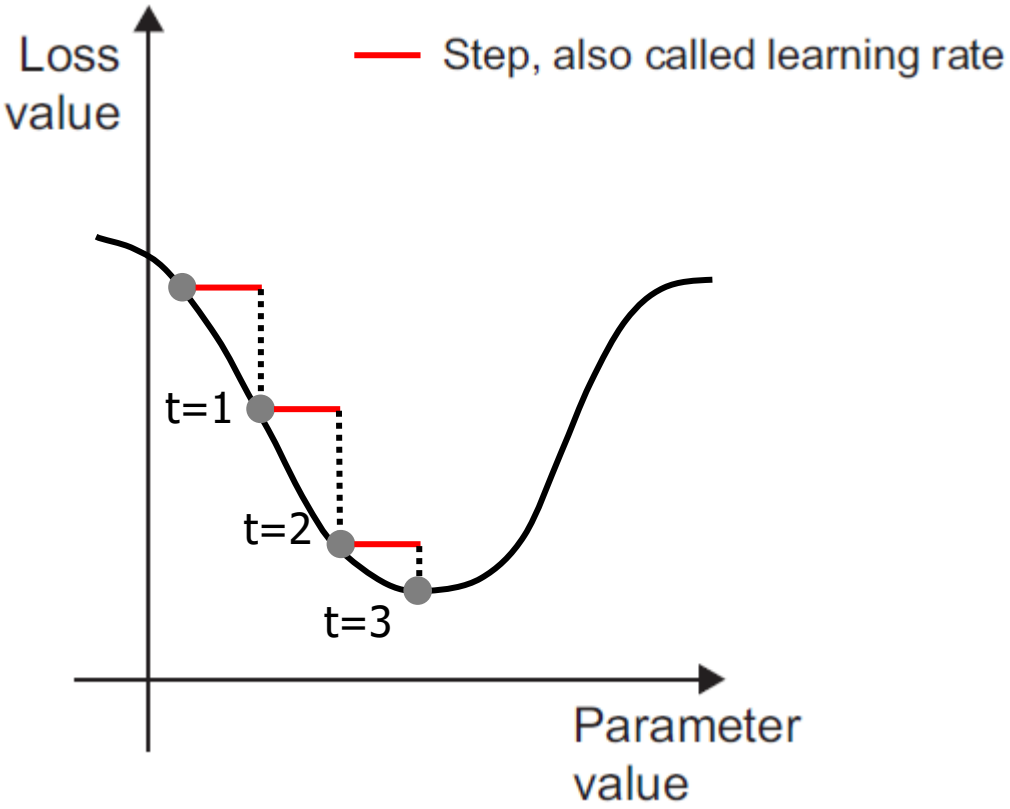
```
learning_rate = 1.0
steps = 30
```





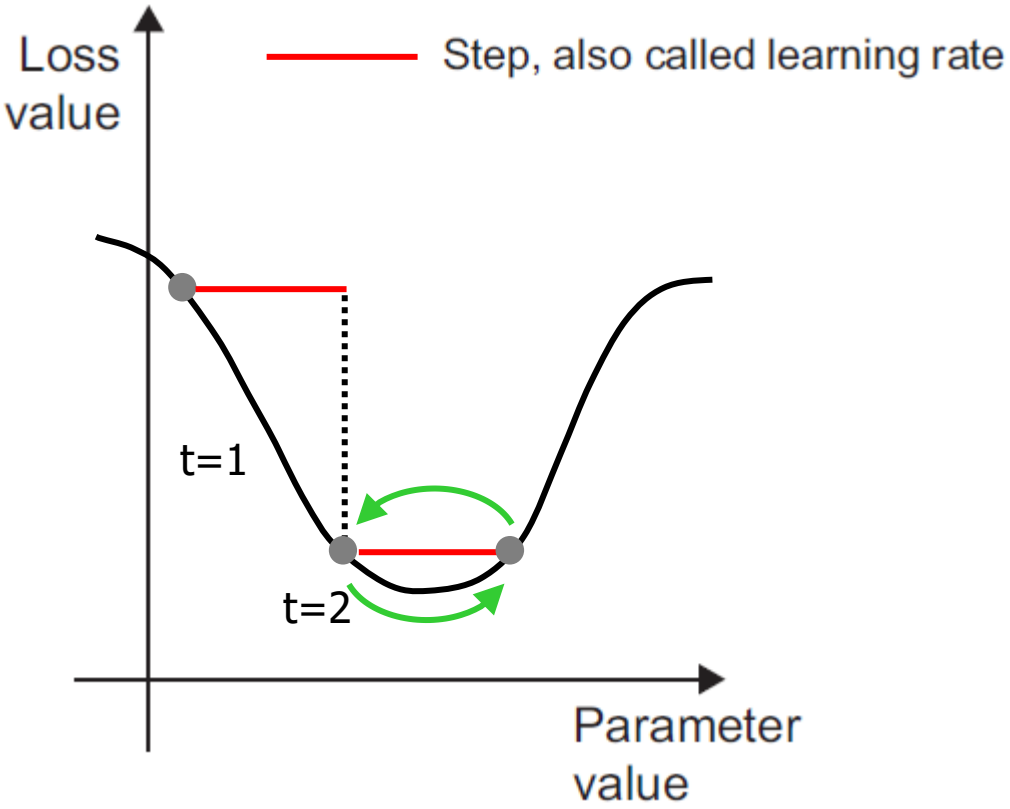
Optymalizacja gradientowa

W przypadku jednego parametru (wagi):



Optymalizacja gradientowa

W przypadku jednego parametru (wagi):



Mini-batch SGD

Uczenie odbywa się w następującej pętli treningowej:

1. Wybierz partię próbek x i odpowiednich celów y.
2. Podaj na wejście sieci x (krok nazywany *forward pass*)), aby uzyskać prognozy y_{pred}.
3. Oblicz stratę sieci czyli błąd między y_{pred} i y.
4. Oblicz gradient funkcji błędu f(W) i zmodyfikuj wszystkie wagi: $W_1 = W_0 - \alpha \cdot \nabla f(W_0)$
5. Jeżeli to konieczne (błąd jest nadal duży) – wróć do punktu 1.

Optymalizacja gradientowa

Uczenie maszynowe polega na aktualizacji współczynników metodą gradientową. Dostępnych jest kilka wariantów tej metody w zależności od wielkości próbki treningowej.

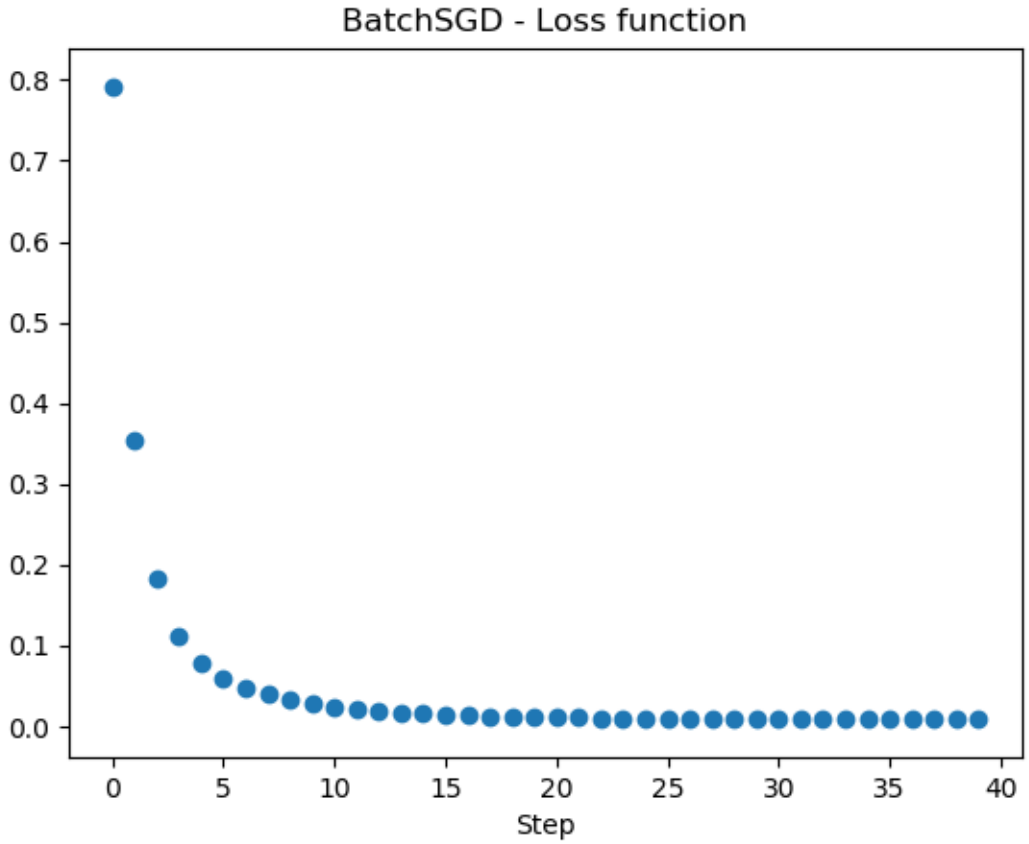
Batch SGD – gradient funkcji straty obliczany jest dla całego zestawu treningowego w każdej epoce.

Minusy takiego rozwiązania:

- Wymaga załadowania całego zestawu danych do pamięci.
- Możliwe utknięcie w minimach lokalnych – mniejsza szansa na znalezienie minimum globalnego.

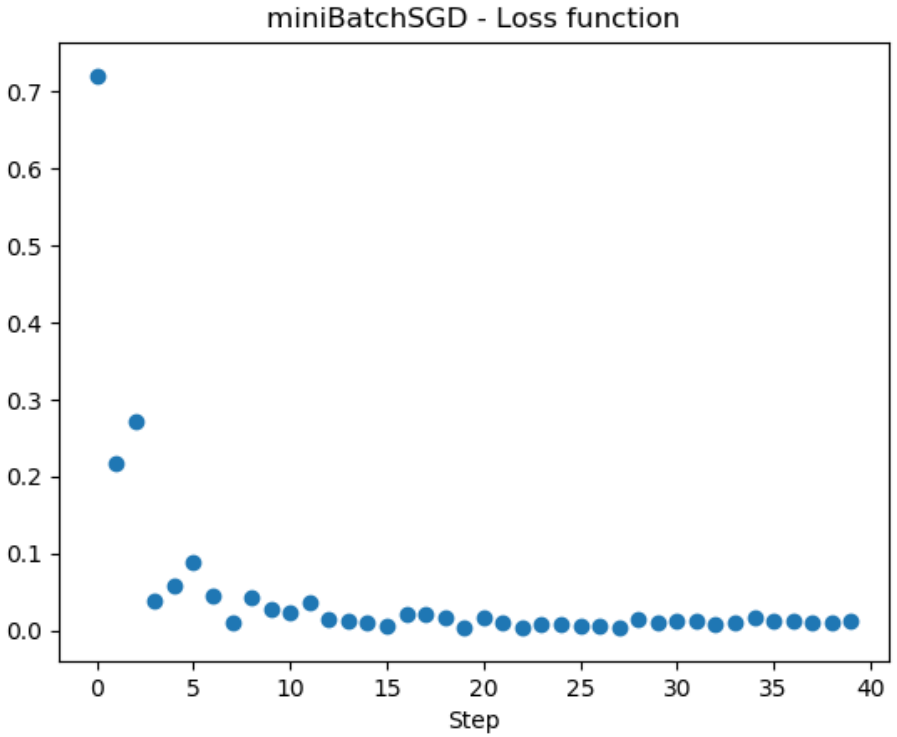
Optymalizacja gradientowa

Batch Gradient Descent – zmiana błędu:



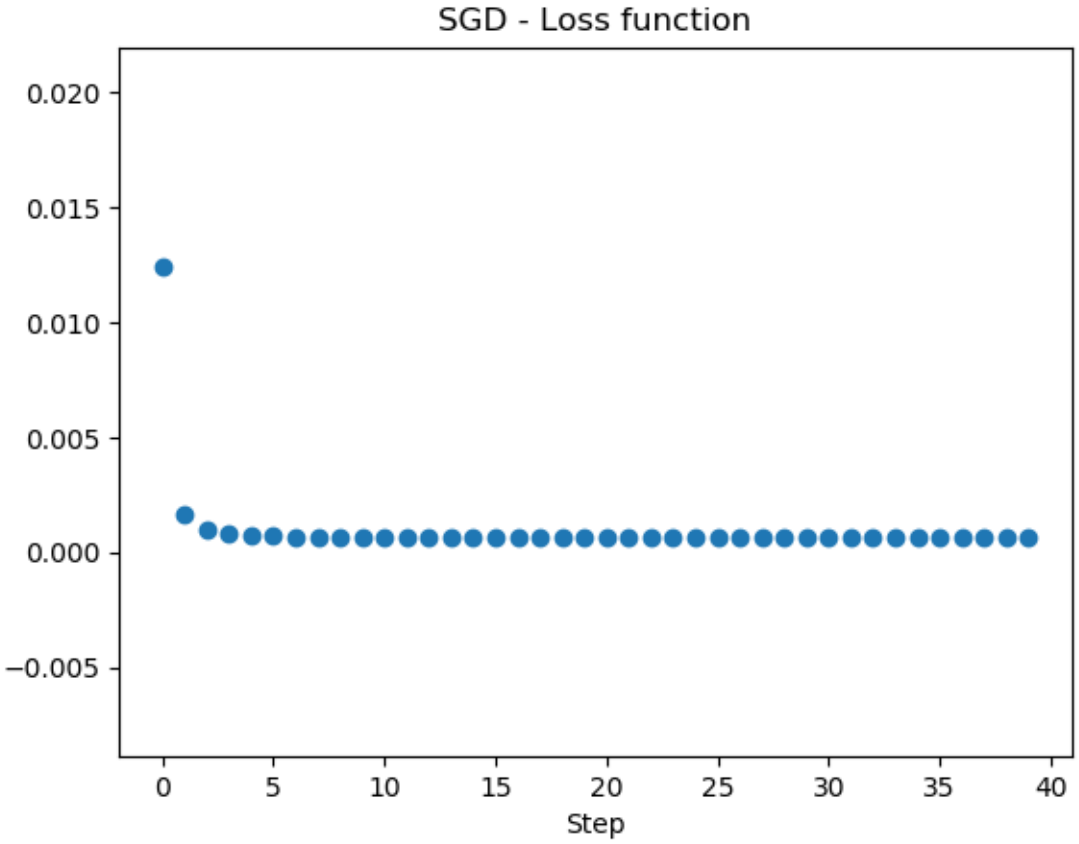
Optymalizacja gradientowa

Mini-batch gradient descent – w tym przypadku parametry są aktualizowane dla pewnej partii danych treningowych (tzw. batch).



Optymalizacja gradientowa

True Stochastic Gradient Descent – gradient funkcji straty obliczany jest dla pojedynczego elementu z zestawu treningowego w każdej epoce.



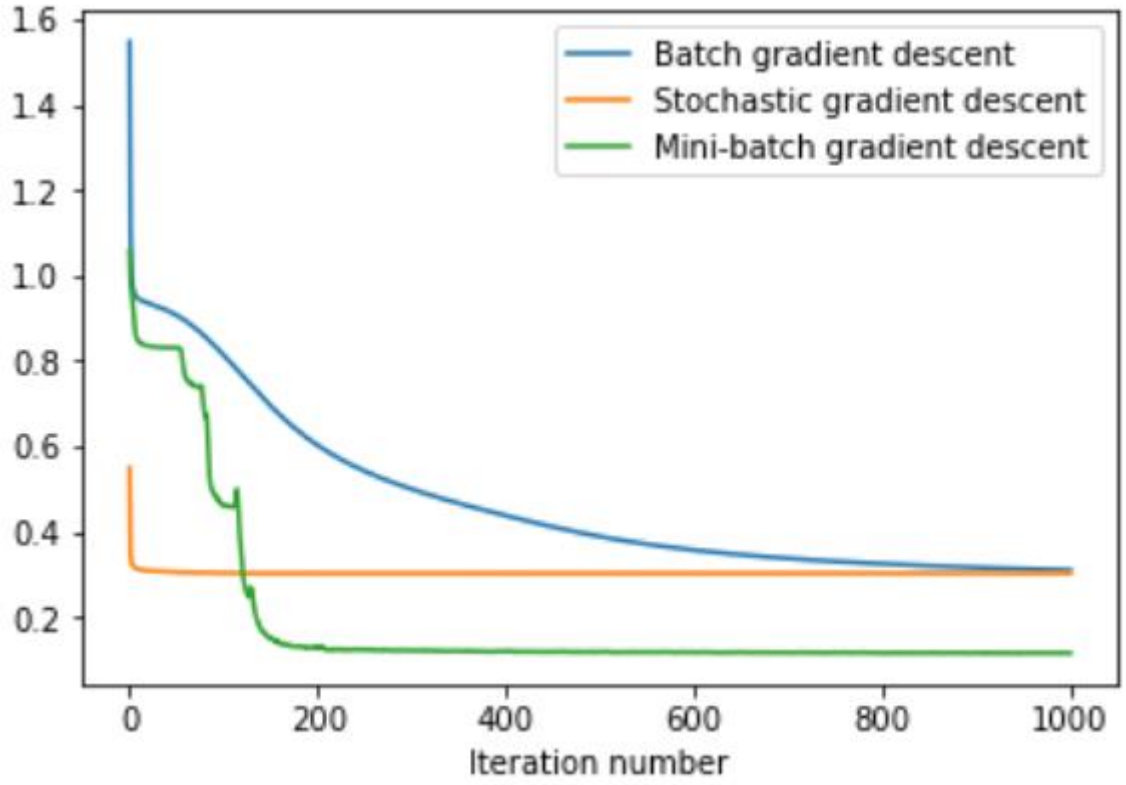
Optymalizacja gradientowa

Funkcja zwracająca partię danych treningowych:

```
def subset_dataset(x_dataset, y_dataset, subset_size):
    arr = np.arange(len(x_dataset))
    np.random.shuffle(arr)
    x_train = [x_dataset[i] for i in arr[0:subset_size]]
    y_train = [y_dataset[i] for i in arr[0:subset_size]]
    return x_train,y_train
```

Optymalizacja gradientowa

Podsumowanie:



Optymalizacja gradientowa

Ponadto istnieje wiele wariantów SGD, które różnią się np. tym, że podczas obliczania następnej aktualizacji parametrów uwzględniają także poprzednie wartości gradientów, a nie tylko ich bieżące wartości.

Jest to np. SGD z członem momentum, a także Adagrad, RMSProp i kilka innych.

Takie warianty nazywane są metodami optymalizacji lub **optymalizatorami**.