



# Deep Learning od podszewki

iDash Workshop  
Warsaw



# Deep Learning od podszewki

iDash Workshop  
Warsaw



[idash.pl](http://idash.pl)

w jaki  
sposób  
prowadziemy  
szkolenia?

# Tematy szkoleń

- Machine Learning (*R, Python*)
- Deep Learning (*R, Python*)
- Przetwarzanie danych (*R, Python*)
- Wprowadzenie (*R, Python*)

i wiele innych. Bazowe programy dostępne na naszej [stronie](#).

# Trenerzy

# Zasady

# Plan na dzisiaj

- Do czego służy propagacja w przód (forward propagation)
- Na czym polega propagacja wsteczna (back propagation)
- Czym jest Gradient Descent
- Do czego służy learning rate

# Technikalia

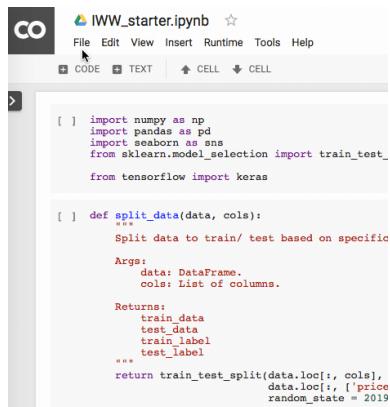
Tą **prezentację** znajdziecie pod następującym adresem:

[http://bit.do/iww\\_dl\\_s](http://bit.do/iww_dl_s)

Punktem wyjściowym jest **notebook** na platformie Google Colab.

[http://bit.do/iww\\_dl\\_c](http://bit.do/iww_dl_c)

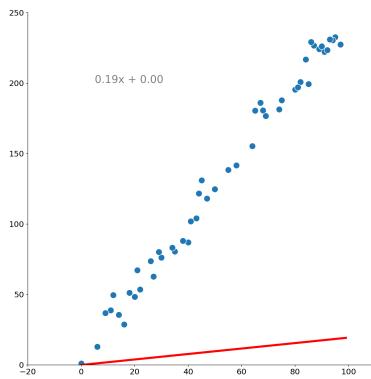
# Google Colab - tworzenie kopii



```
IWW_starter.ipynb ☆
File Edit View Insert Runtime Tools Help
CODE TEXT ⌄ CELL ⌅ CELL
[ ] import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.model_selection import train_test_
from tensorflow import keras

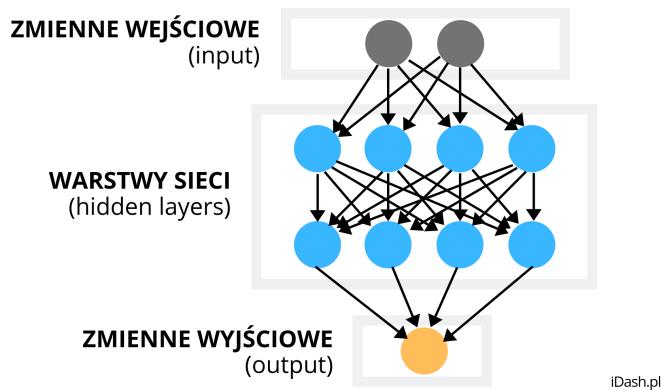
[ ] def split_data(data, cols):
    """Split data to train/ test based on specific
    Args:
        data: DataFrame.
        cols: List of columns.
    Returns:
        train_data
        test_data
        train_label
        test_label
    """
    return train_test_split(data.loc[:, cols],
                           data.loc[:, ['price']],
                           random_state = 2019
```

# Co dzisiaj zrobimy



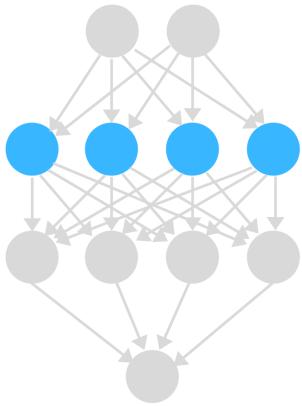
# Powtórka

# Struktura sieci neuronowej



iDash.pl

# Warstwa

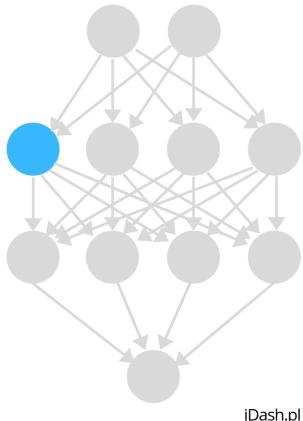


iDash.pl

Warstwa sieci  
składa się z  
pojedynczych  
**neuronów.**

Sieć może  
mieć **wiele**  
**warstw.**

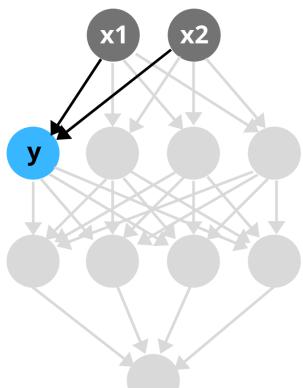
# Neuron



iDash.pl

Neuron to  
**najmniejszy  
element  
sieci.**

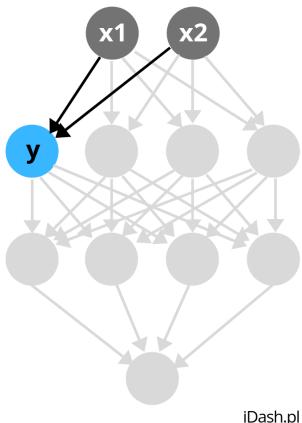
# Neuron



Neuron to  
**najmniejszy  
element  
sieci.**

Jest to w  
najprostszym  
wariancie  
**funkcja  
liniowa** wielu  
zmiennych.

# Neuron



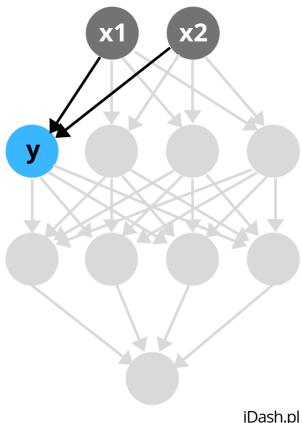
iDash.pl

Neuron to  
**najmniejszy**  
**element**  
**sieci.**

Jest to w  
najprostszym  
wariantie  
**funkcja**  
**liniowa** wielu  
zmiennych.

Bierze ona  
wartości z  
poprzedniej  
warstwy,  
mnoży je  
przez pewne  
**wagi** i dodaje 16 / 99

# Neuron



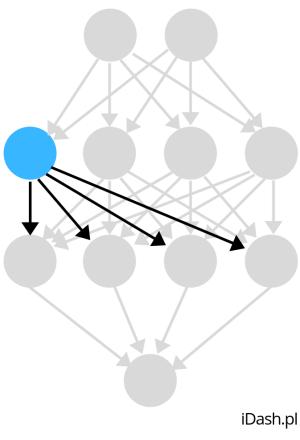
iDash.pl

Neuron to  
**najmniejszy**  
**element**  
**sieci.**

Jest to w  
najprostszym  
wariantie  
**funkcja**  
**liniowa** wielu  
zmiennych.

Bierze ona  
wartości z  
poprzedniej  
warstwy,  
mnoży je  
przez pewne  
**wagi** i dodaje 17 / 99

# Neuron



Obliczona wartość jest wykorzystywana jako wartość wejściowa (input) przez neurony znajdujące się w kolejnej warstwie.

# Do czego zmierzamy?

Sieć dąży do tego, aby ustalić takie wagi, aby generowane predykcje były najbardziej zbliżone do faktycznych wartości widocznych w danych.

**Trenowanie sieci**, to zatem nic innego jak proces dostosowywania wag.

# Trenowanie sieci

Trenowanie sieci jest  
**procesem iteracyjnym:**

1. Dane wejściowe pozwalają wyliczyć funkcję straty.
2. W oparciu o funkcję straty, wyznaczany jest kierunek zmian wag parametrów modelu.
3. Optimizer aktualizuje wagi parametrów modelu.
4. Zaktualizowane wagi + te same dane wejściowe

# Intro to Colab Notebook

Zbudujmy  
sieć od zera!

**Bez użycia  
dedykowanych  
frameworków!!!**

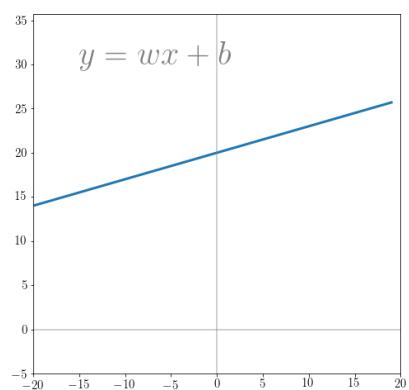
A komu to  
potrzebne?

# Potrzebne aby...

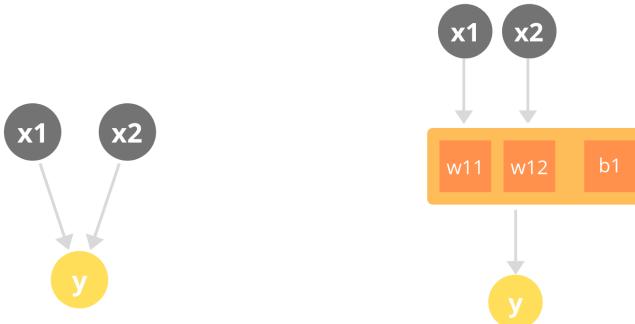
- aby móc zrozumieć, jak sieci neuronowe działają *under the hood*.
- przy kolejnym nieudanym treningu Twojej sieci, wiedzieć co może być przyczyną złych wyników.
- móc pochwalić się przed kolegami/koleżankami :)

Od czego  
zacząć?

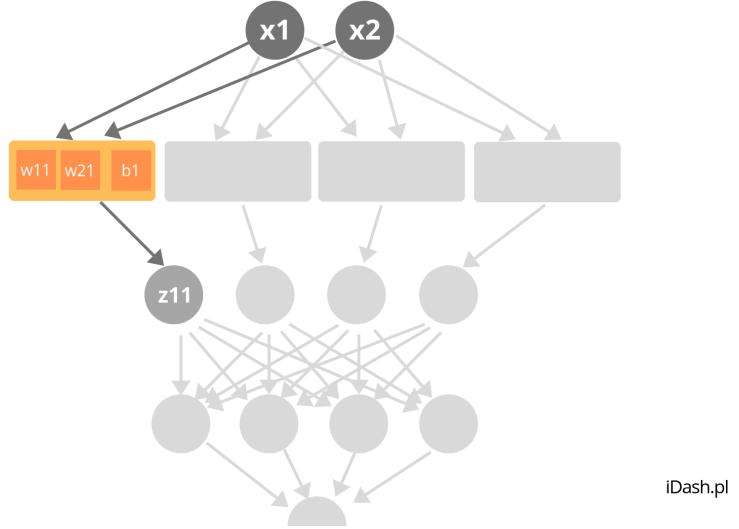
Od funkcji  
liniowej!



# Funkcja liniowa pod postacią sieci

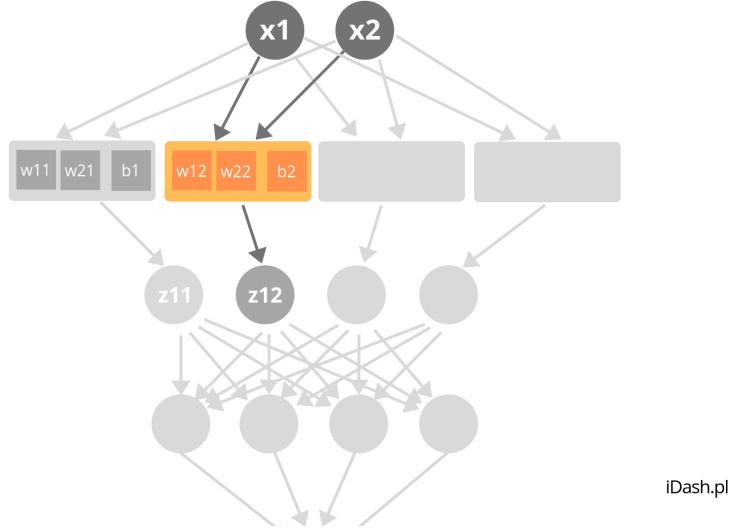


iDash.pl



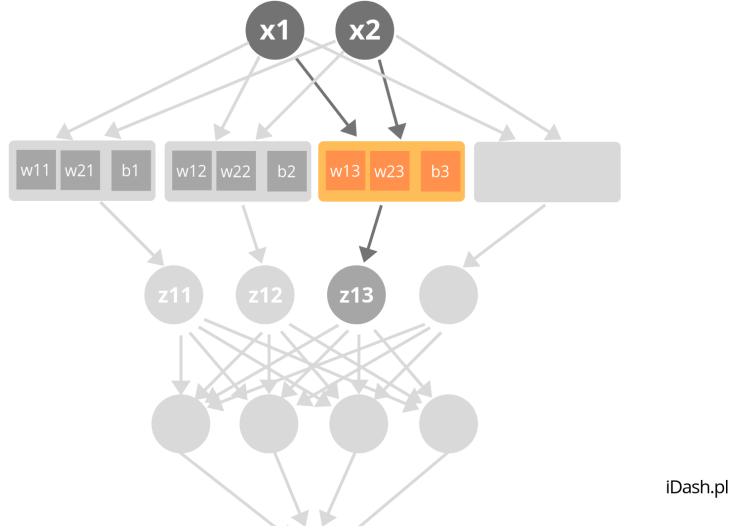
iDash.pl

$$z_{11} = x_1 w_{11} + x_2 w_{21} + b_1$$



$$z_{11} = x_1 w_{11} + x_2 w_{21} + b_1$$

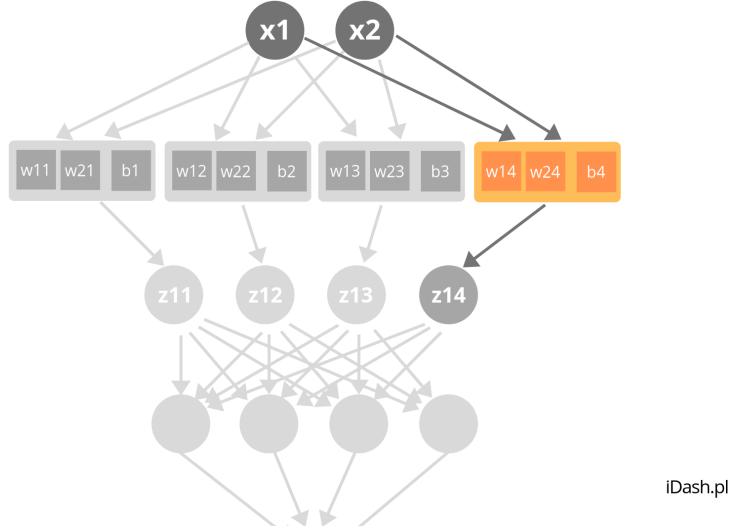
$$z_{12} = x_1 w_{12} + x_2 w_{22} + b_2$$



$$z_{11} = x_1 w_{11} + x_2 w_{21} + b_1$$

$$z_{12} = x_1 w_{12} + x_2 w_{22} + b_2$$

$$z_{13} = x_1 w_{13} + x_2 w_{23} + b_3$$

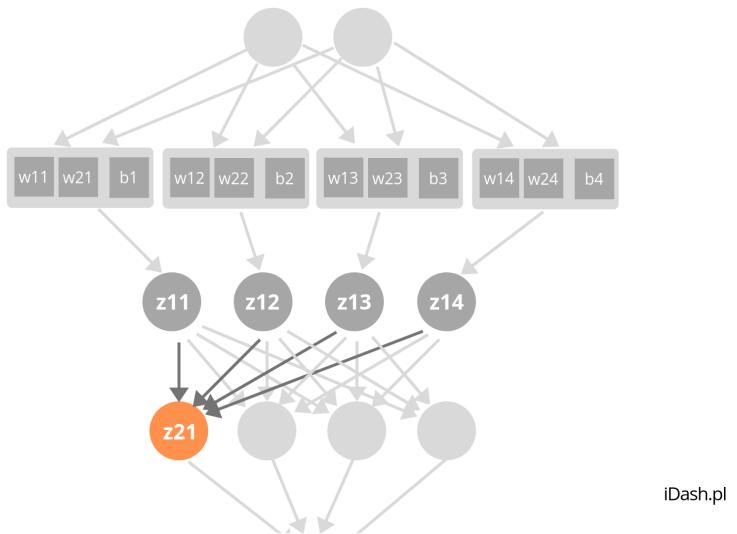


$$z_{11} = x_1 w_{11} + x_2 w_{21} + b_1$$

$$z_{12} = x_1 w_{12} + x_2 w_{22} + b_2$$

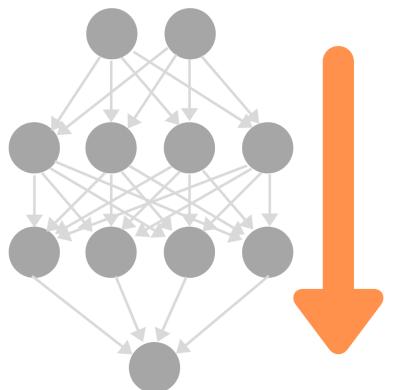
$$z_{13} = x_1 w_{13} + x_2 w_{23} + b_3$$

$$z_{14} = x_1 w_{14} + x_2 w_{24} + b_4$$



# Czym jest zatem forward propagation?

**Forward propagation** jest terminem, który oznacza proces przebiegu informacji od warstwy wejściowej (*input*) do warstwy wyjściowej (*output*).



iDash.pl

Jak policzyć jak  
bardzo nasza  
predykcja  
odbiega od  
wartości  
prawdziwej?

Za pomocą  
funkcji  
straty!

# Mean Square Error

$$\text{square error} = (\hat{y} - y)^2$$

\* w przypadku liczenia błędu na całym zbiorze danych należy policzyć średnią błędów (**mean square error**).

# Iinicjalizowanie wag parametrów

- Aby móc rozpoczęć trenowanie sieci musimy najpierw zainicjalizować wartości wag.
- Zwykle przyjmuje się, że wartości powinny być generowane z wybranego rozkładu (np. rozkładu jednostajnego (*uniform*) albo normalnego ze średnią 0).
- Generalnie, zainicjalizowane wartości

# Zadanie 1a (5 min)

W oparciu o teorię propagacji w przód (forward propagation) stwórz funkcję `get_linear_forward()`, która będzie zwracać wartość funkcji liniowej.

# Zadanie 1a (3 min)

Wykorzystaj testy jednostkowe aby sprawdzić, że wszystko zostało dobrze zaimplementowane.

```
x = np.array([1, 2, 3])
y = np.array([ 3.9,  1.6,

y_hat = get_linear_forward
assert np.mean(y_hat) == 8
assert y_hat.shape == (3, )
```

# Zadanie 1b (5 min)

Następnie stwórz mechanizm liczenia funkcji straty mean square error (MSE) (nazwa funkcji: `compute_mse_loss`).

**Hint:** Funkcja `np.power` służy do podnoszenia wartości do potęgi.

# Zadanie 1b (3 min)

Wykorzystaj testy jednostkowe aby sprawdzić, że wszystko zostało dobrze zaimplementowane.

```
x = np.array([1, 2, 3])
y = np.array([ 3.9,  1.6,

y_hat = get_linear_forward
assert np.mean(y_hat) == 8
assert y_hat.shape == (3,)
assert compute_mse_loss(y_
```

## Zadanie 2 (10 min)

1. W oparciu o zbudowane funkcje z zadania 1a i 1b, stwórz funkcję `train`, która zwracać będzie błąd funkcji straty (MSE). Niech funkcja pozwala zdefiniować startowe wartości parametrów modelu (`weight` oraz `bias`).

2. Zainicjuj wagi parametrów:

```
INIT_WEIGHT = 0.5  
INIT_BIAS = 0
```

3. Wykonaj funkcję `train` na danych `generated_data` (dostępne w Google Colab).

## Zadanie 2 (10 min)

Wykorzystaj testy jednostkowe aby sprawdzić, że wszystko zostało dobrze zaimplementowane.

```
first_step = train(x_train  
                    y_train  
                    init_w=  
                    init_b=  
  
assert type(first_step) ==  
assert first_step == 14596
```

Niezadowalające  
wyniki?

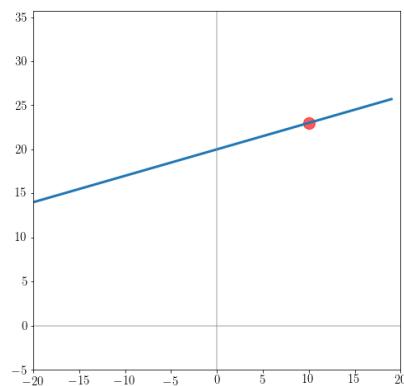
Musimy  
rozpocząć  
trenowanie!

# Trenowanie

- Trenowanie sieci to proces dostosowywania wag.
- Trenowanie ma charakter iteracyjny - dążymy do tego aby **minimalizować** funkcję straty.
- Zatem, naszym głównym zadaniem jest wyznaczenie **kierunku zmian**.

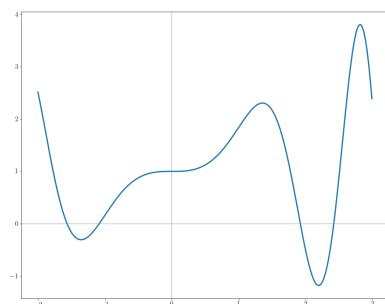
Jak  
wyznaczyć  
kierunek  
zmian?

# Quiz - jak określić wartość zmian?

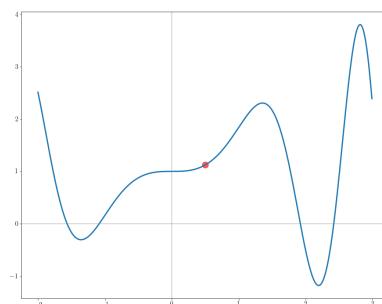


Co jeśli ...

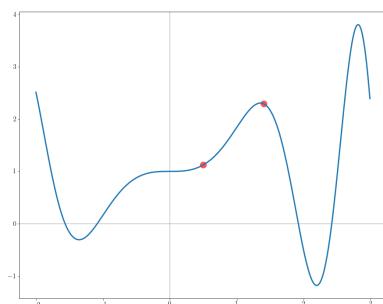
... mamy do czynienia z funkcją nieliniową?



... mamy do czynienia z funkcją nieliniową?



# ... mamy do czynienia z funkcją nieliniową?

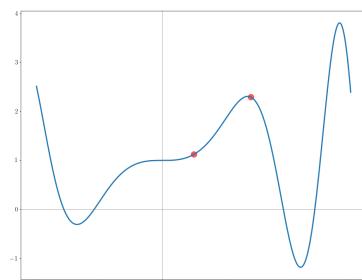


Jak wtedy wyznaczyć zmianę  
w danym punkcie?

Istnieje  
jedno  
magiczne  
narzędzie!

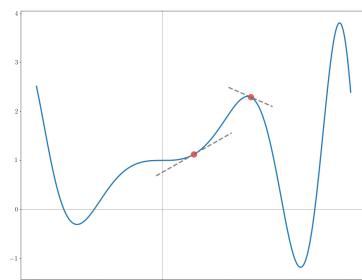
# Zmiana w punkcie

- Narzędzie, które pozwala nam wyznaczyć kąt nachylenia funkcji w danym punkcie, tym samym kierunek zmian.



# Zmiana w punkcie

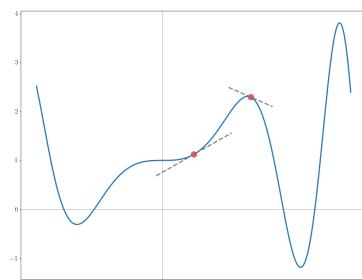
- Narzędzie, które pozwala nam wyznaczyć kąt nachylenia funkcji w danym punkcie, tym samym kierunek zmian.



- Kąt ten jest

# Zmiana w punkcie

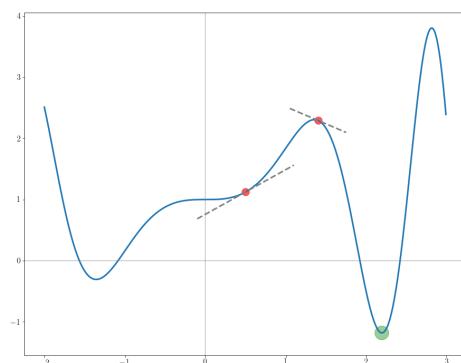
- Narzędzie, które pozwala nam wyznaczyć kąt nachylenia funkcji w danym punkcie, tym samym kierunek zmian.



- Kąt ten jest

Jak  
wykorzystać  
wiedzę o  
pochodnej w  
kontekście  
trenowania  
sieci?

# Poszukujemy punktu minimum



# Pochodne w sieciach neuronowych

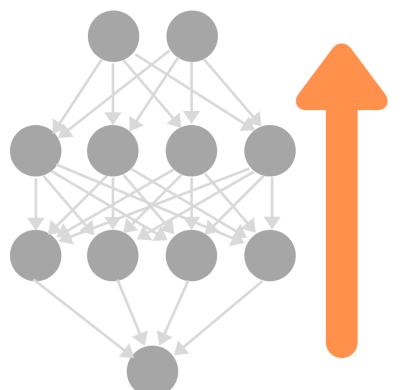
# Pochodna funkcji straty

$$Loss = (\hat{y} - y)^2 = ((wx + b) - y)^2$$

- Dane są  $y$  (przewidywana wartość) jak również  $x$  (zmienne w modelu).
- Dla parametrów  $w$  i  $b$  musimy policzyć pochodne.
- Pochodną ze względu na konkretny parametr definiujemy zapisem  $\frac{df}{d(\text{parametr})}$ , np.  $\frac{df}{dw}$ .

# Back propagation

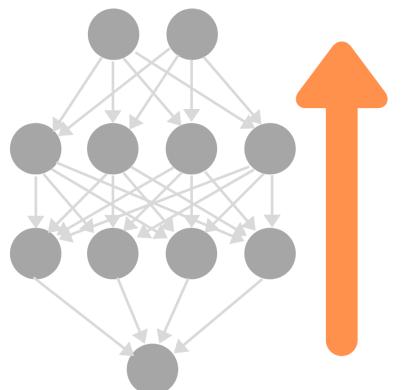
Cały proces wyliczania pochodnych dla wszystkich parametrów sieci nazywamy propagacją wsteczną (*back propagation*).



iDash.pl

# Back propagation

Cały proces wyliczania pochodnych dla wszystkich parametrów sieci nazywamy propagacją wsteczną (*back propagation*).



iDash.pl

Wektor wyliczonych pochodnych wszystkich

Jak policzyć  
pochodną?

# Zadanie 3a. (10 min)

1. Policzmy pochodne!
2. Stwórz funkcję `get_backprop`, która posłuży do wyliczenia pochodnych parametrów modelu liniowego.
3. Do wyliczenia pochodnych skorzystaj ze wzoru na funkcję straty *Loss* (slajd 62) oraz z linku z poprzedniego slajdu.

# Zadanie 3a. (10 min)

Wykorzystaj testy jednostkowe aby sprawdzić, że wszystko zostało dobrze zaimplementowane.

```
x = np.array([1, 2, 3])
y = np.array([ 3.9,  1.6,
output = get_linear_forwar

back_step = get_backprop(o
assert len(back_step) == 2
assert len(back_step[0]) =
assert len(back_step[1]) =
assert np.sum(back_step) =
```

# Gradienty zostały policzone ale...

- Dla każdej obserwacji w zbiorze funkcja `get_backprop` policzy gradient (czyli wektor pochodnych).
- Mając  $N$  obserwacji, mamy  $N$  gradientów. A naszym zadaniem jest wyznaczenie **jednego** kierunku zmian.
- Co zrobić?

## Zadanie 3b. (5 min)

Dokonaj odpowiednich przeształceń funkcji `get_backprop`, aby zwracała jeden gradient z pochodnymi dla parametrów sieci.

## Zadanie 3b. (5 min)

Wykorzystaj testy jednostkowe.

```
x = np.array([1, 2, 3])
y = np.array([ 3.9, 1.6,
output = get_linear_forwar

back_step = get_backprop(o
assert len(back_step) == 2
assert type(back_step[0])
assert type(back_step[1])
assert np.sum(back_step) =
```

# Under the hood\*

$$\text{Loss} = ((wx + b) - y)^2$$

- Gdy funkcja jest zależna od więcej niż jednej zmiennej to wtedy do wyliczenia pochodnej musimy wykorzystać tzw. **pochodne cząstkowe**.
- Pozwala nam ocenić, jak szybko zmienia się wartość funkcji gdy manipujemy wartością tylko jednej zmiennej, a wartości pozostałych

# Under the hood\*

Ze względu na to że funkcja straty jest **funkcją złożoną** (tj.  $f(g(x))$ ), możemy skorzystać z reguły łańcuchowej (*chain rule*).

$$\text{Loss} = ((wx + b) - y)^2$$

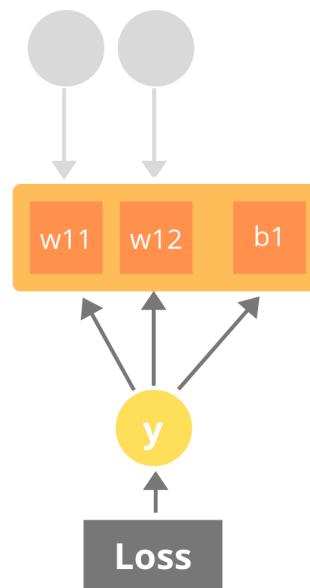
$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w}$$

$$\frac{\partial L}{\partial w} = ((\hat{y} - y)^2)' * (wx + b)'$$

$$\frac{\partial L}{\partial w} = 2(\hat{y} - y) * x, \quad \frac{\partial L}{\partial b} = 2(\hat{y} - y)$$

# Under the hood\*

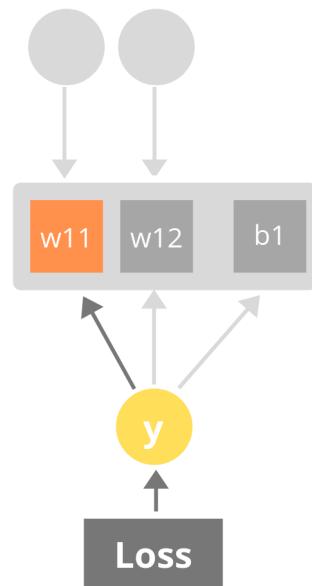
Taka forma wyliczania pochodnych jest niezbędna przy bardziej rozbudowanych sieciach neuronowych.



# Under the hood\*

Taka forma wyliczania pochodnych jest niezbędna przy bardziej rozbudowanych sieciach neuronowych.

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial \hat{y}} *$$



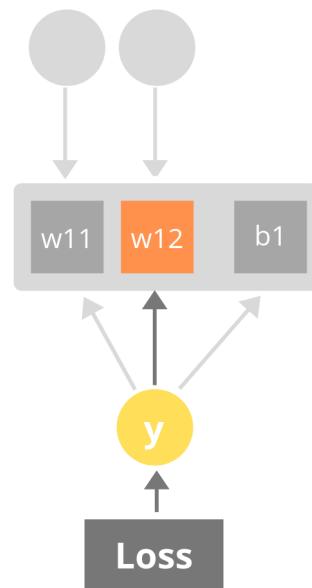
iDash.pl

# Under the hood\*

Taka forma wyliczania pochodnych jest niezbędna przy bardziej rozbudowanych sieciach neuronowych.

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial \hat{y}} *$$

$$\frac{\partial L}{\partial w_{12}} = \frac{\partial L}{\partial \hat{y}} *$$

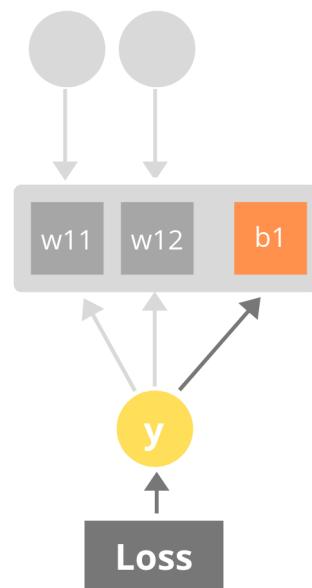


# Under the hood\*

Taka forma wyliczania pochodnych jest niezbędna przy bardziej rozbudowanych sieciach neuronowych.

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial \hat{y}} *$$

$$\frac{\partial L}{\partial w_{12}} = \frac{\partial L}{\partial \hat{y}} *$$



iDash.pl

Pochodne  
wyliczone a  
ciągle stoję w  
miejscu!

# Pomysły?

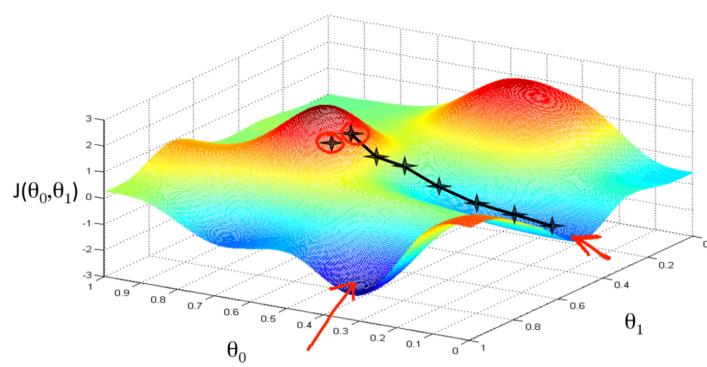
Musimy się  
jakoś ruszyć!

Gradient  
Descent na  
ratunek!

# Gradient Descent

Gradient Descent jest algorytmem optymalizacyjnym.

- Jego celem jest znalezienie minimum danej funkcji.
- Jest odpowiedzialny za modyfikowanie wag parametrów modelu w oparciu o wyliczoną pochodną.
- Po aktualizacji wag parametrów, następuje ponowna iteracja forward



# Gradient Descent

$$w_{new} = w_{current} - \alpha * \frac{\partial L}{\partial w}$$

- $w_{new}, w_{current}$  - obecna i zaktualizowana wag parametru  $w$ .
- $\frac{\partial f}{\partial w}$  - pochodna dla parametru  $w$  względem funkcji straty  $L$ .
- Do czego **służy**  $\alpha$ ?

# Parametr $\alpha$

- Parametr  $\alpha$  zwany **learning rate** jest kluczowym elementem algorytmu Gradient Descent.
- $$w_{new} = w_{current} - \alpha * \text{gradient}$$

# Parametr $\alpha$

- Parametr  $\alpha$  zwany **learning rate** jest kluczowym elementem algorytmu Gradient Descent.  
 $w_{new} = w_{current} - \alpha *$
- Learning rate kontroluje tempo, w jakim nasz model będzie trenowany.

# Na co uważać

Podczas ustalania wartości *learning rate* należy być ostrożny.

## Zbyt duża wartość $\alpha$

Trenowanie się nie powiedzie, ponieważ algorytm Gradient Descent w kolejnych iteracjach będzie powodować wzrost błędu funkcji straty, tym samym nie możliwe będzie znalezienie minimum.

## Zbyt mała wartość $\alpha$

Proces trenowania będzie trwać dłużej. Może również

# Playground

# Jak ustawić odpowiednio learning rate?

- Nie ma jednej gotowej reguły. Wszystko zależy od struktury sieci, a także od danych.
- W teorii, wartość learning rate powinna definiowana z przedziału  $\alpha \in (0, 1)$ .
- W praktyce, wartość learning rate powinna raczej oscylować w przedziale  $\alpha \in (0.0001, 0.1)$ .

## Zadanie 4. (5 min)

Stwórz funkcję  
update\_parameters, która  
będzie wyliczać nowe  
zaktualizowane parametry w  
oraz b.

# Zadanie 4. (5 min)

Wykorzystaj testy jednostkowe.

```
w_current = 1
b_current = 0
alpha = 0.1
w_gradient = 0.7
b_gradient = -0.01

new_params = update_params(
    w_current, b_current,
    alpha, w_gradient)

assert len(new_params) == 2
assert new_params[0] == 0.0
assert new_params[1] == 0.0
```

# Zadanie 5. (15 min)

1. Zmodyfikuj funkcję `train`, tak aby umożliwiała znalezienie optymalnych wartości parametrów modelu.

- Dodaj argument `epochs`, które będzie umożliwiał wielokrotne powtarzanie kroków propagacji.
- Niech na wyjściu funkcja zwraca nowe zaktualizowane wagi parametrów `weight` i `bias`.
- \*Niech funkcja wyświetla wyliczony błąd za pomocą funkcji `compute_mse_loss` po każdej zapiszczonej epoce.

2. Wykonaj funkcję `train` na 1000 epokach na zbiorze `generated_data`, manipulując parametrem *learning rate*.

# Zadanie 5. (15 min)

Wykorzystaj testy jednostkowe.

```
x = np.array([1, 2, 3])
y = np.array([ 3.9, 1.6,
init_weight = 0.1
init_bias = 0
alpha = 0.1
epochs = 10

final_params = train(x, y,
rounded_final_params = np.
assert len(final_params) =
assert rounded_final_param
assert rounded_final_param
```

# Learning rate a normalizacja danych

- Normalizacja danych wejściowych może przyspieszyć proces trenowania.
- W trakcie mniejszej liczby epok może osiągnąć minimum funkcji straty.
- Należy jednak wtedy pamiętać, żeby na nowo manipulować parametrem *learning rate*, ponieważ pochodne będą się różniły, aniżeli to

## Zadanie 6\*. (10 min)

1. Znormalizuj dane wejściowe (pomocniczy [link](#)).
2. Wykonaj funkcję `train` na 10 epokach.

Czy sieć szybciej się wytrenuje?

Zrobiliśmy  
to!

# Podsumowanie

# Podsumowanie

- Udało się nam zbudować bardzo prostą sieć od zera.
- Poznaliśmy najważniejsze koncepty, dzięki którym możliwe jest trenowanie sieci neuronowych.
- Musimy jednak pamiętać, że głębokie sieci składają się z **tysięcy parametrów** i mogą posiadać **wiele warstw** i wtedy cały proces trenowania, szczególnie propagacja wsteczna (*back propagation*) jest zadaniem bardziej

Kolejne  
spotkanie już  
24.  
października.

Do zobaczenia!



mb@idash.pl mo@idash.pl