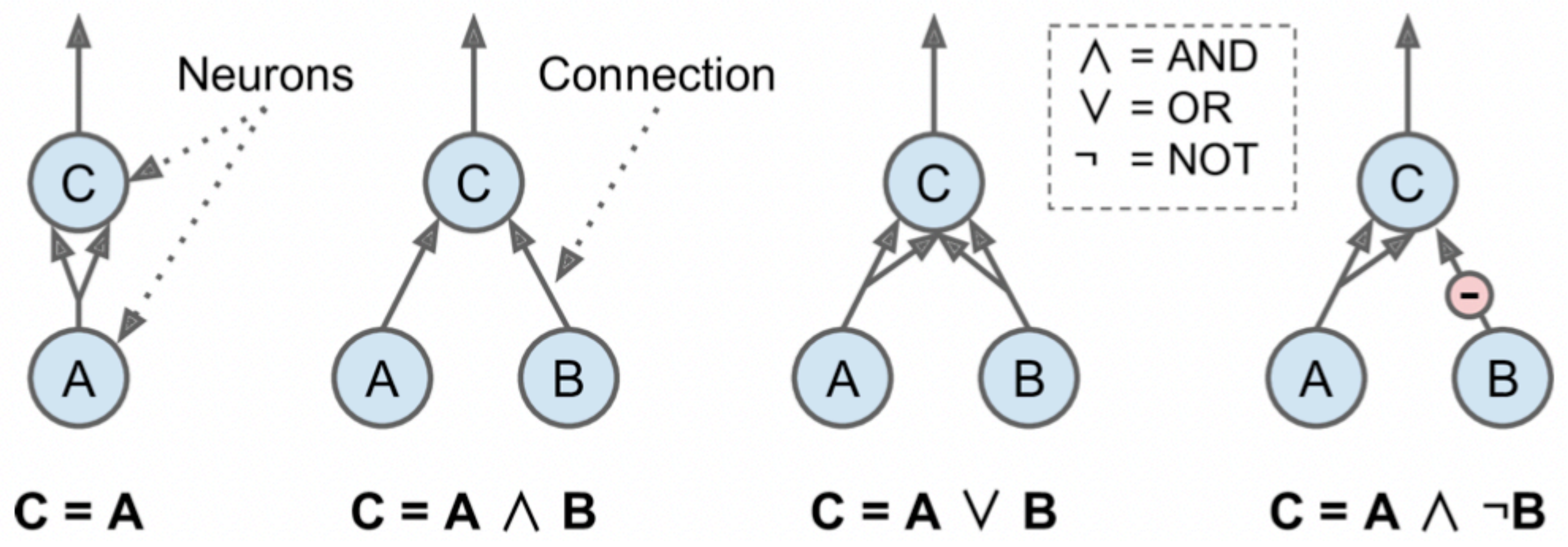
Sztuczne sieci neuronowe

Sztuczny Neuron

"*A Logical Calculus of Ideas Immanent in Nervous Activity*", Warren McCulloch & Walter Pitts (1943)

Neuron zostaje uaktywniony, gdy przynajmniej dwa wejścia będą aktywne.



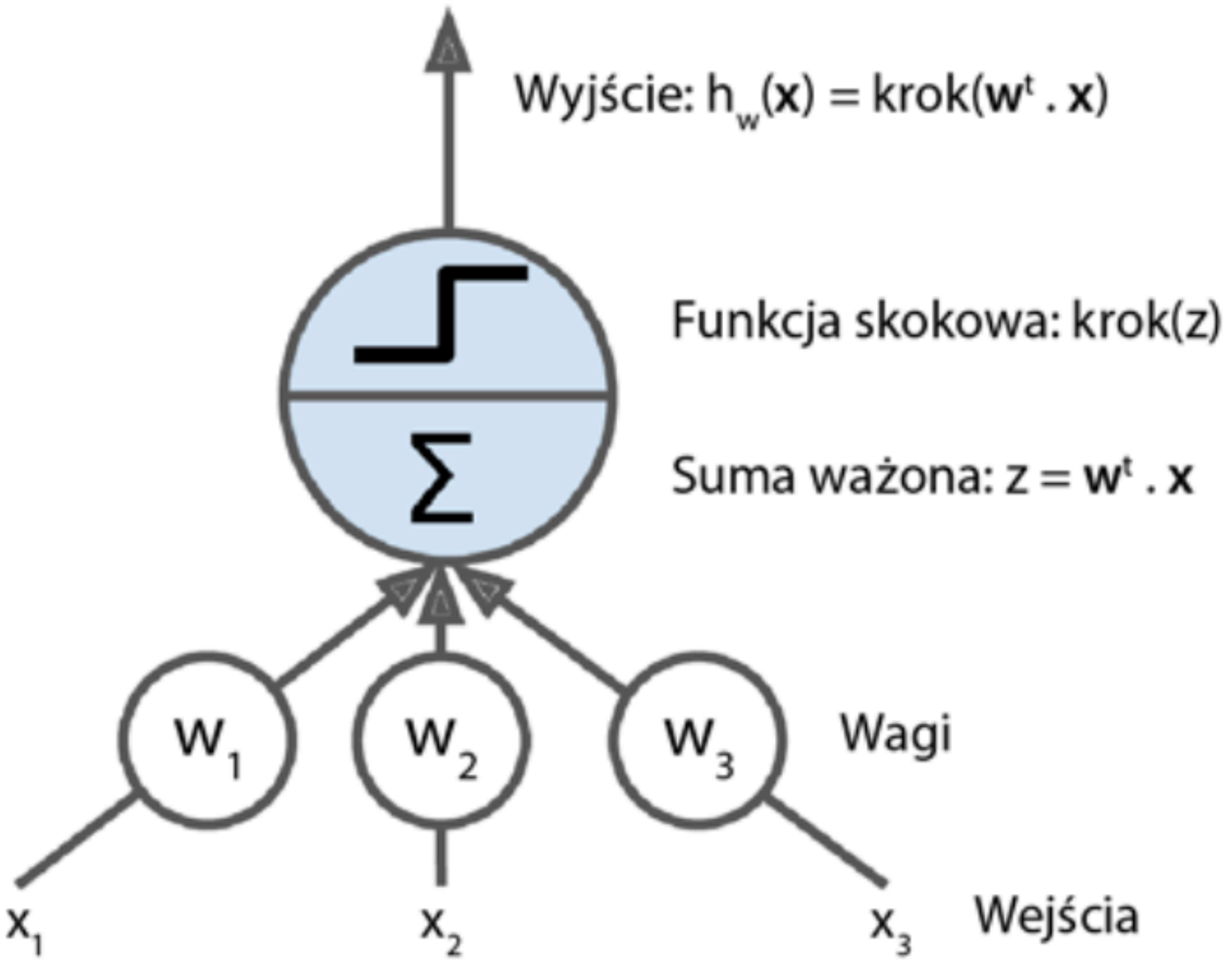
- Pierwsza sieć po lewej to po prostu funkcja tożsamościowa: jeśli neuron A zostanie uaktywniony, aktywacji ulegnie także neuron C (gdyż odbiera dwa sygnały wejściowe z neuronu A), natomiast jeśli neuron A będzie wyłączony, to nie będzie również aktywny neuron C.
- Druga sieć wykonuje operację logiczną I: neuron C zostaje aktywowany jedynie wtedy, gdy obydwa neurony — A i B — będą włączone (pojedynczy sygnał wejściowy nie wystarczy do aktywacji neuronu C).
- Trzecia sieć przeprowadza operację logiczną LUB: neuron C jest aktywny, jeśli któryś z pozostałych neuronów (albo obydwa) zostanie aktywowany.
- W ostatnim przypadku przy założeniu, że sygnał może hamować aktywność neuronu (tak jak to się dzieje w neuronach biologicznych), omawiana sieć przeprowadza nieco bardziej skomplikowaną operację logiczną. Neuron C zostaje aktywowany wyłącznie wtedy, gdy otrzyma sygnał z neuronu A, natomiast neuron B musi być wyłączony. Jeśli neuron A będzie cały czas włączony, to otrzymamy bramkę negacji: neuron C jest aktywny przy wyłączonym neuronie B i odwrotnie.

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, Aurélien Géron

Perceptron

linear threshold unit (LTU)

Frank Rosenblatt (1957)



Najpowszechniejsze funkcje skokowe wykorzystywane w perceptronach

$$\text{Heaviside}(z) = \begin{cases} 0 \text{ jeśli } z < 0 \\ 1 \text{ jeśli } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 \text{ jeśli } z < 0 \\ 0 \text{ jeśli } z = 0 \\ +1 \text{ jeśli } z > 0 \end{cases}$$

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, Aurélien Géron

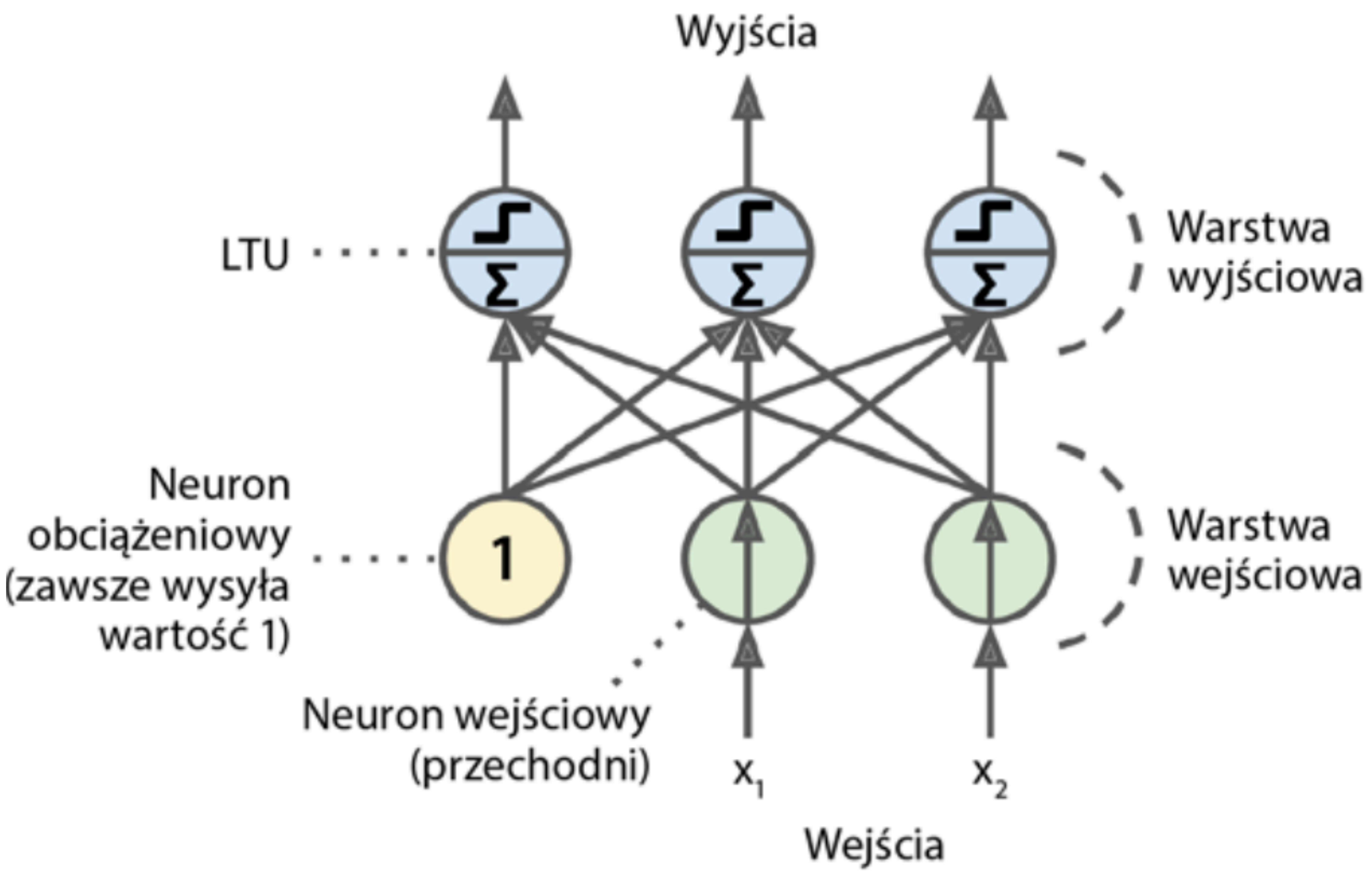
Perceptron

Uczenie

Cells that fire together, wire together

Donald Hebb w swojej książce The Organization of Behavior z 1949 roku zasugerował, że gdy biologiczny neuron często pobudza inną komórkę nerwową, to połączenie pomiędzy tymi dwoma neuronami staje się silniejsze.

Koncepcja ta została później błyskotliwie podsumowana przez Siegrida Löwela: „*Cells that fire together, wire together*”, czyli w wolnym tłumaczeniu:
„***Komórki pobudzające się wzajemnie wiążą się ze sobą***”.



$$w_{i,j}^{(\text{następny krok})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

$w_{i,j}$ — waga połączenia pomiędzy i-tym neuronem wejściowym i j-tym neuronem wyjściowym,

x_i — i-ta wartość wejściowa bieżącego przykładu uczącego,

$\hat{y}_j$ — wynik j-tego neuronu wyjściowego dla bieżącego przykładu uczącego,

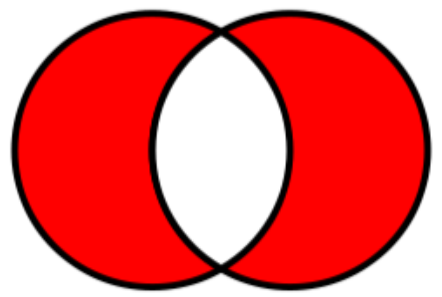
y_j — docelowy wynik j-tego neuronu wyjściowego dla bieżącego przykładu uczącego,

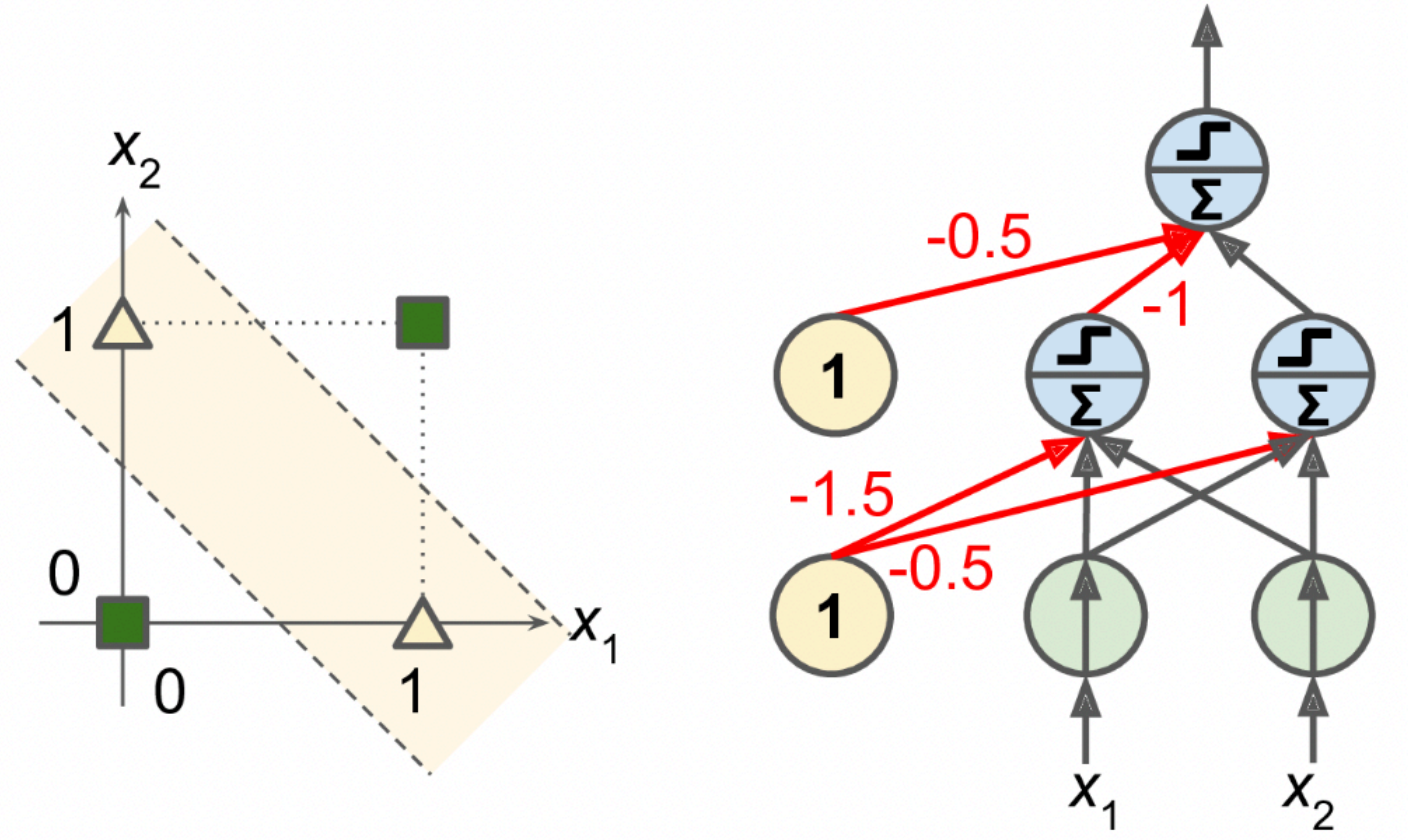
η — współczynnik uczenia.

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, Aurélien Géron

Multi-Layer Perceptron (MLP)

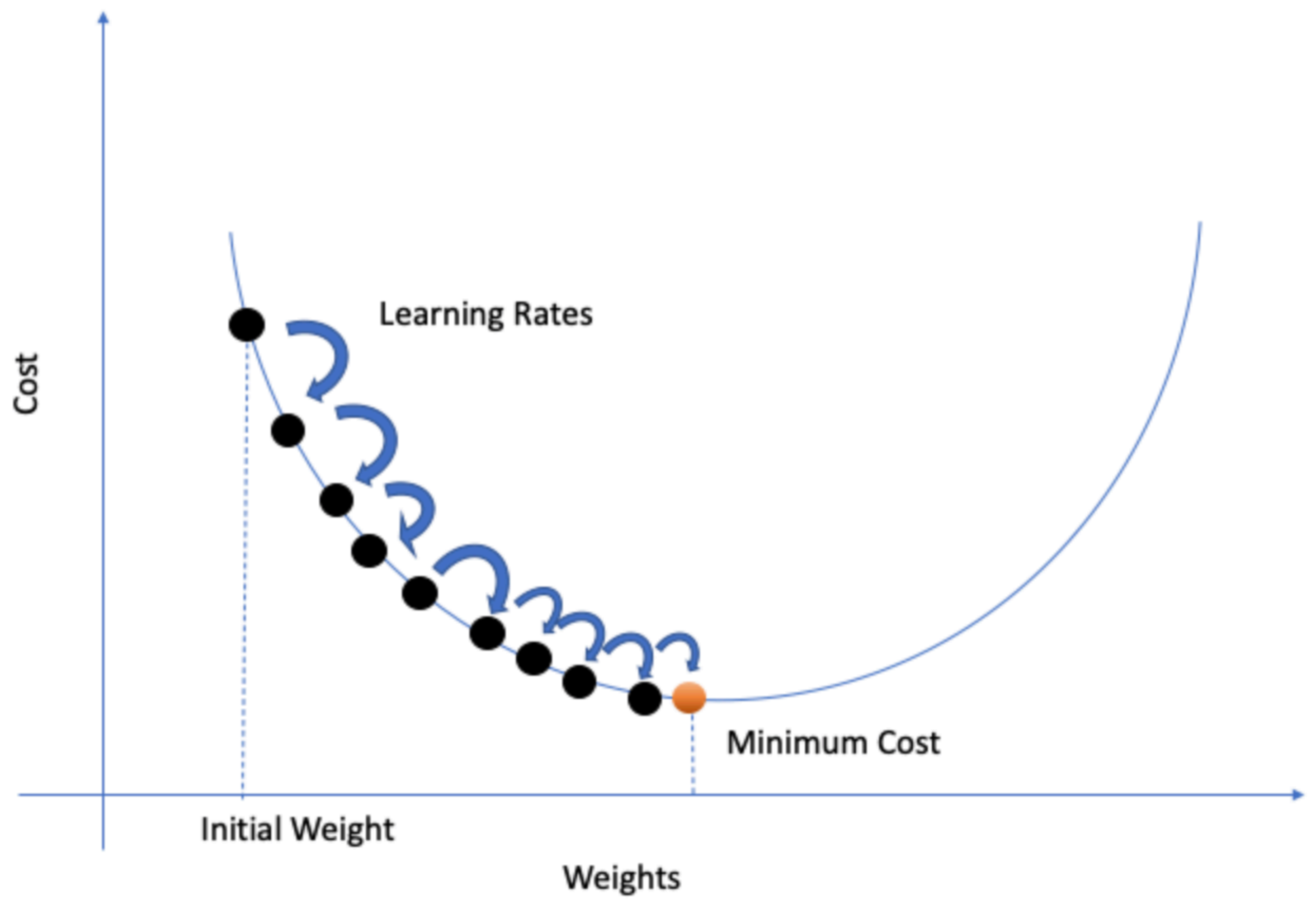
XOR - Exclusive or (alternatywa rozłączeń)





Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, Aurélien Géron

Gradient z minigrupami *Mini-batch Gradient Descent*



$$\theta_j := \theta_j - \eta \frac{\partial}{\partial \theta_j} J_I(\theta_0, \theta_1)$$

$\eta - learning\ rate\ (0.01)$

$$X = \begin{bmatrix} x_{1,1} & \cdots & x_{1,n} \\ x_{2,1} & & x_{2,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{bmatrix}$$

$$\boldsymbol{x_i} = [x_{i,1} \quad x_{i,2} \quad \cdots \quad x_{i,n}]$$

$I - ustalone, losowy\ podzbiór\ X$

$$J(\theta_0, \theta_1) = \frac{1}{2|I|} \sum_{i \in I} (h_\theta(x_i) - y_i)^2$$

$$h_\theta(x_i) = \boldsymbol{\theta^T x_i}$$

$$J(\theta_0, \theta_1) = \frac{1}{2|I|} \sum_{i \in I} (\boldsymbol{\theta^T x_i} - y_i)^2$$

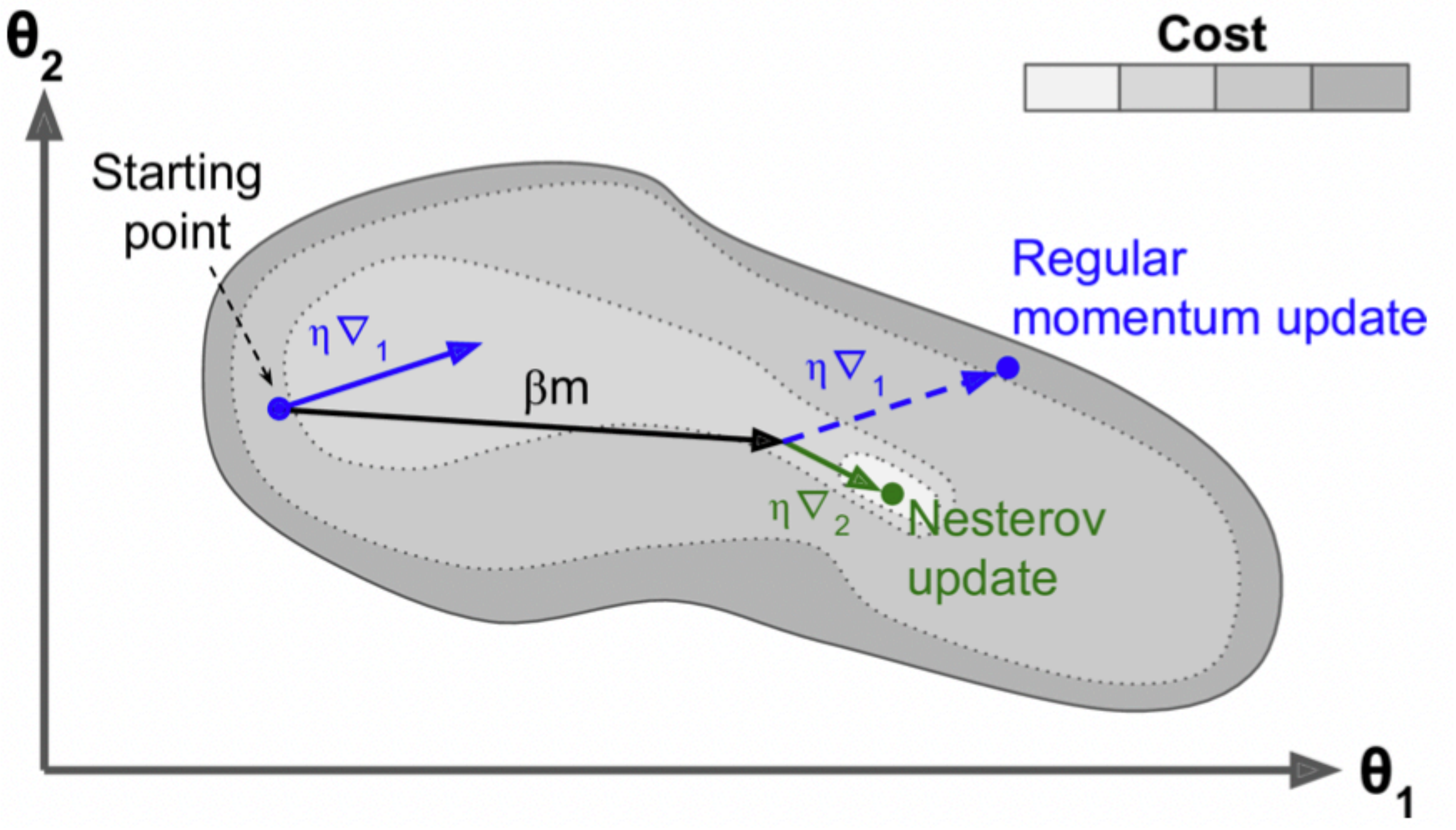
$$\frac{\partial}{\partial \theta_j} J_I(\theta_0, \theta_1) = \frac{1}{2|I|} \sum_{i \in I} 2(\boldsymbol{\theta^T x_i} - y_i) x_{i,j} = \frac{1}{2|I|} \sum_{i \in I} (\boldsymbol{\theta^T x_i} - y_i) x_{i,j}$$

Optymalizatory

Momentum Optimization (optymalizacja momentu)

$$\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$$

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```



Nesterov Accelerated Gradient (NAG) (algorytm Nestrova)

$$\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta\mathbf{m})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$$

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, Aurélien Géron

Optymalizatory

RMSProp

$$\mathbf{s} \leftarrow \beta \mathbf{s} + (1-\beta)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\, \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \epsilon}$$

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Adam (adaptive momentum estimation)

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} - \left(1-\beta_1\right)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\mathbf{s} \leftarrow \beta_2 \mathbf{s} + \left(1-\beta_2\right)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1-{\beta_1}^t}$$

$$\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-{\beta_2}^t}$$

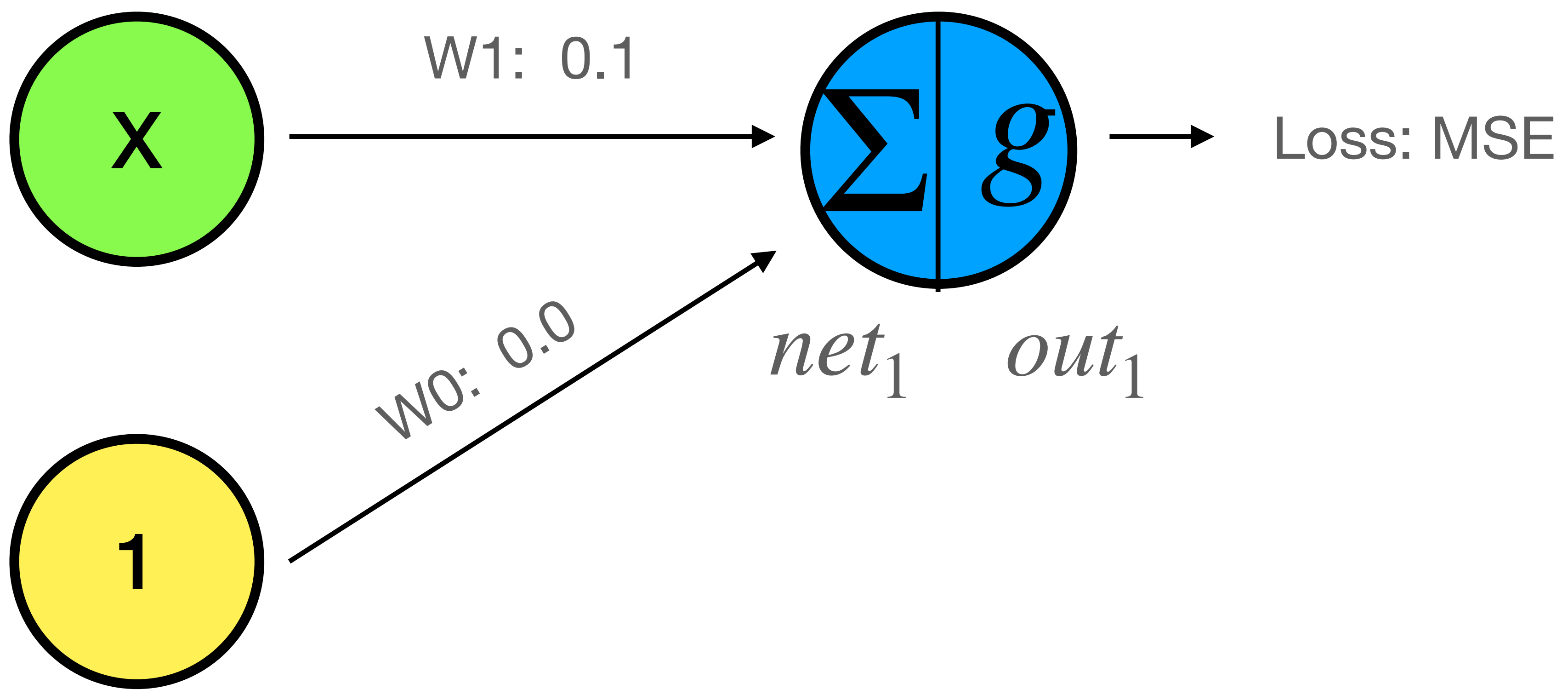
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\, \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \epsilon}$$

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, Aurélien Géron

Trenowanie

epoka: 1



$$X = \begin{bmatrix} 0.06 \\ 0.87 \\ 0.6 \\ 0.71 \\ 0.02 \end{bmatrix}$$

$$y = \begin{bmatrix} 3.17 \\ 4.73 \\ 4.26 \\ 4.57 \\ 3.02 \end{bmatrix}$$

$$g \rightarrow g(x) = x$$

I. Forward propagation

1. Wejście neuronu (średnia ważona)

$$\sum = W^T X = \sum_i w_i x_i = w_1 x_1 + w_0 = \begin{bmatrix} 0.006 \\ 0.087 \\ 0.060 \\ 0.071 \\ 0.002 \end{bmatrix}$$

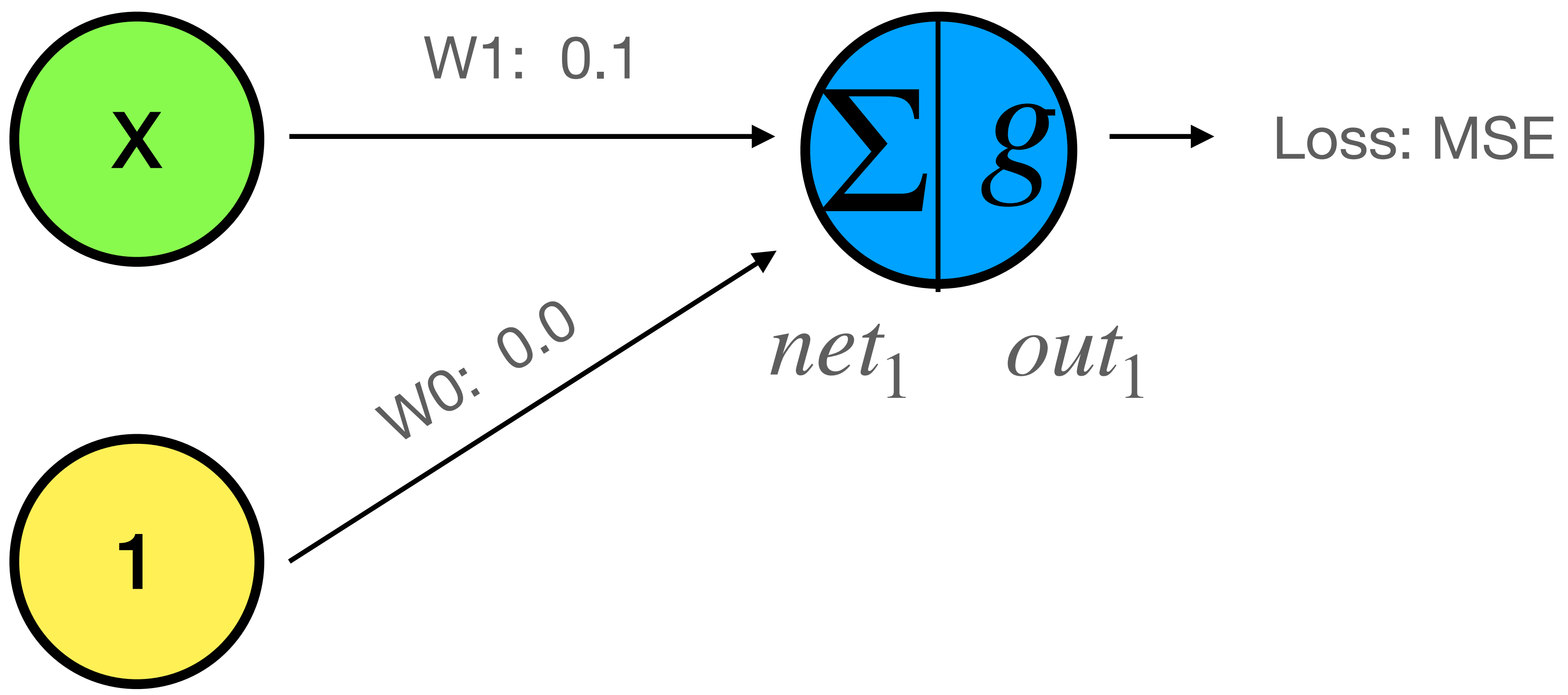
2. Funkcja aktywacji (wyjście z sieci/predykcje)

$$g(x) = x = \begin{bmatrix} 0.006 \\ 0.087 \\ 0.060 \\ 0.071 \\ 0.002 \end{bmatrix}$$

3. Strata

$$MSE = \frac{1}{n} \sum (y_{true} - y)^2 = 15.71$$

Trenowanie



$$net_1 \quad out_1$$

$$X = \begin{bmatrix} 0.06 \\ 0.87 \\ 0.6 \\ 0.71 \\ 0.02 \end{bmatrix}$$

$$y = \begin{bmatrix} 3.17 \\ 4.73 \\ 4.26 \\ 4.57 \\ 3.02 \end{bmatrix}$$

$$g \rightarrow g(x) = x$$

II. Backward propagation

$\frac{\partial Loss}{\partial w_1}$ - jaki wpływ na stratę ma waga w_1

$$\frac{\partial Loss}{\partial w1} = \frac{\partial Loss}{\partial out_1} \times \frac{\partial out_1}{\partial net_1} \times \frac{\partial net_1}{\partial w_1}$$

Reguła łańcuchowa, pochodne funkcji złożonej (chain rule)

1. Liczymy wpływ wyjścia z neuronu na stratę

$$Loss = \left(y_{true} - out_1\right)$$

$$\frac{\partial Loss}{\partial out_1} = 2 * \left(y_{true} - out_1\right)^2 \times (-1) = -2 \times \begin{bmatrix} -3.164 \\ -4.643 \\ -4.2 \\ -4.499 \\ -3.018 \end{bmatrix} = \begin{bmatrix} -6.328 \\ -9.286 \\ -8.400 \\ -8.998 \\ -6.036 \end{bmatrix}$$

2. Liczymy wpływ wejścia na wyjście

$$g(x) = x$$

$$out_1 = g(net_1) = net_1$$

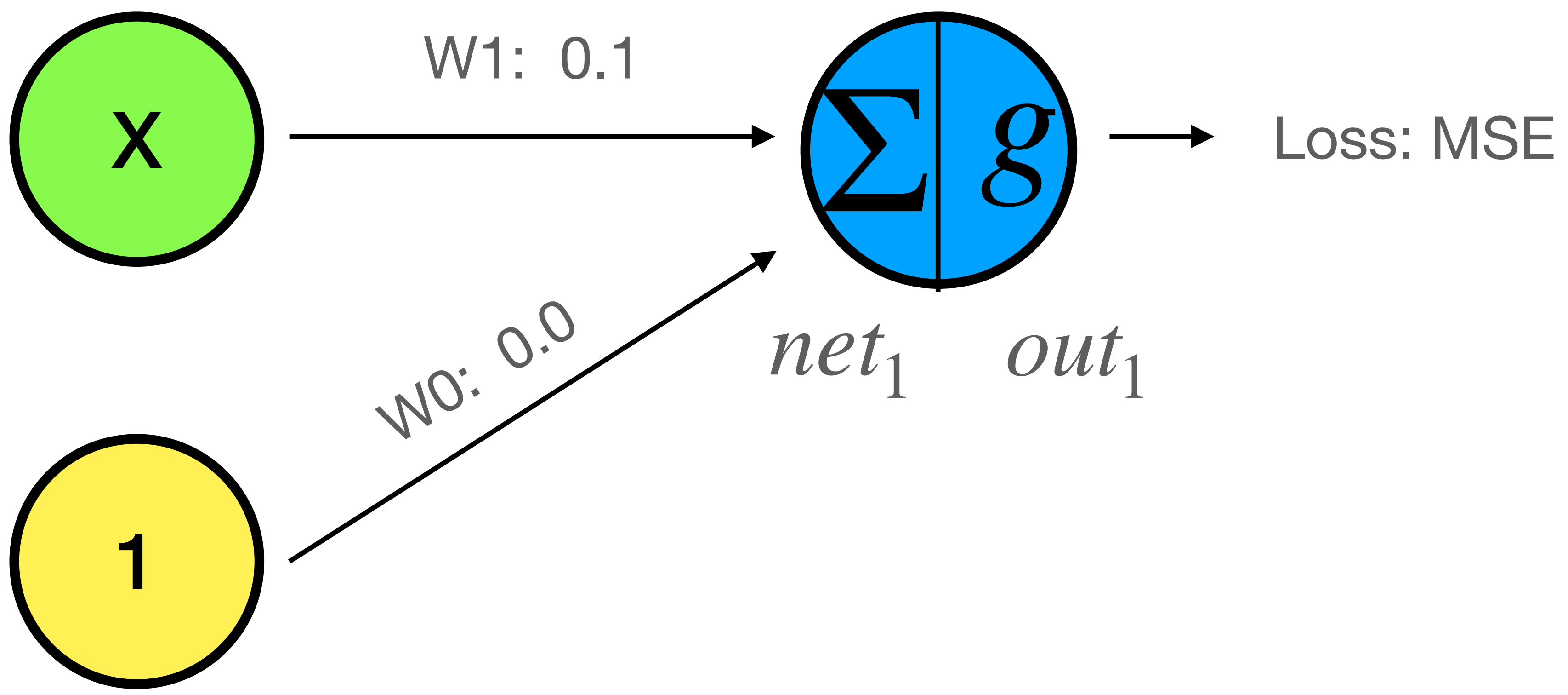
$$\frac{\partial out_1}{\partial net_1} = 1$$

3. Liczymy wpływ wagi na wejście

$$net_1 = w_1 \times X + w_0$$

$$\frac{\partial net_1}{\partial w_1} = X = \begin{bmatrix} 0.06 \\ 0.87 \\ 0.60 \\ 0.71 \\ 0.02 \end{bmatrix}$$

Trenowanie



$$X = \begin{bmatrix} 0.06 \\ 0.87 \\ 0.6 \\ 0.71 \\ 0.02 \end{bmatrix}$$

$$y = \begin{bmatrix} 3.17 \\ 4.73 \\ 4.26 \\ 4.57 \\ 3.02 \end{bmatrix}$$

$$g \rightarrow g(x) = x$$

II. Backward propagation

$\frac{\partial Loss}{\partial w_1}$ - jaki wpływ na stratę ma waga w_1

$$\frac{\partial Loss}{\partial w1} = \frac{\partial Loss}{\partial out_1} \times \frac{\partial out_1}{\partial net_1} \times \frac{\partial net_1}{\partial w_1}$$

Reguła łańcuchowa, pochodne funkcji złożonej (chain rule)

4. Obliczamy (średni) wpływ wag na stratę

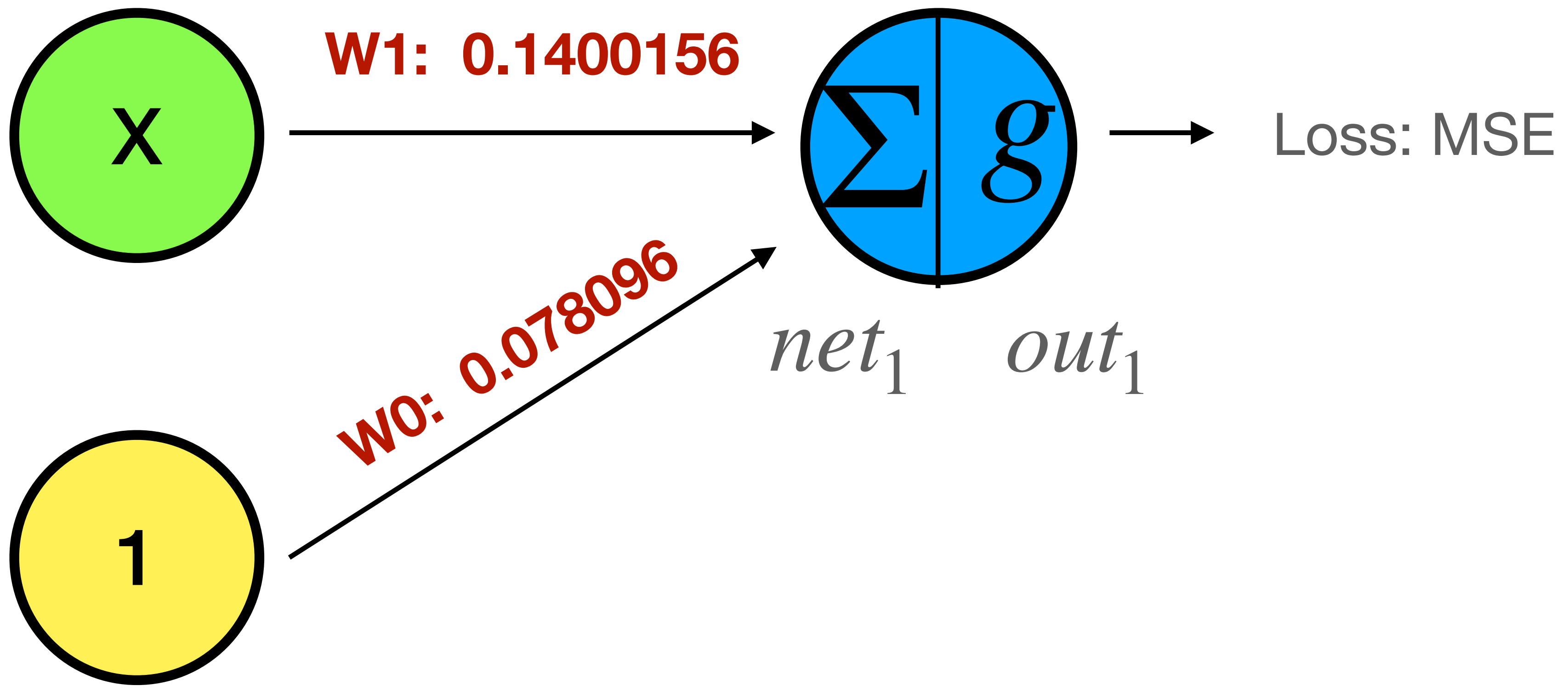
$$\frac{\partial Loss}{\partial w_1} = \begin{bmatrix} -6.328 \\ -9.286 \\ -8.400 \\ -8.998 \\ -6.036 \end{bmatrix} \times 1 \times \begin{bmatrix} 0.06 \\ 0.87 \\ 0.60 \\ 0.71 \\ 0.02 \end{bmatrix} = \begin{bmatrix} -0.38 \\ -8.079 \\ -5.04 \\ -6.389 \\ -0.121 \end{bmatrix}$$ → mean loss: $= -4.00156$

5. Aktualizujemy wagi

$$w_{1:new} = w_1 - \eta \left(\frac{\partial Loss}{\partial w_1}\right)_{\text{mean}} = 0.1400156$$

6. Podobne obliczenia powtarzamy dla wagi W0

Trenowanie



$$X = \begin{bmatrix} 0.06 \\ 0.87 \\ 0.6 \\ 0.71 \\ 0.02 \end{bmatrix}$$

$$y = \begin{bmatrix} 3.17 \\ 4.73 \\ 4.26 \\ 4.57 \\ 3.02 \end{bmatrix}$$

$$g \rightarrow g(x) = x$$

epoka: 2

Powtarzamy Forward propagation I Back propagation
dla nowych zaktualizowanych wag.

epoka: 3

(…)

epoka: n

Kontynuujemy proces dopóki nie skończymy
wszystkich iteracji (epok) lub nie osiągniemy
zamierzonego rezultatu.

Optimizers TF

optimizers

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001)
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9)
optimizer = tf.keras.optimizers.SGD(learning_rate=0.001, momentum=0.9, nesterov=True)

model.compile(optimizer='SGD', loss=loss)

optimizer = tf.keras.optimizers.RMSprop(learning_rate=0.001, rho=0.9)

model.compile(optimizer='RMSprop', loss=loss)

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)

model.compile(optimizer='Adam', loss=loss)
```

Losses TF

losses

```
loss = tf.keras.losses.MeanSquaredError()

model.compile(optimizer=optimizer, loss=loss)
model.compile(optimizer=optimizer, loss='mean_squared_error')

loss = tf.keras.losses.MeanAbsoluteError()

model.compile(optimizer=optimizer, loss=loss)
model.compile(optimizer=optimizer, loss='mean_absolute_error')

loss = tf.keras.losses.SparseCategoricalCrossentropy()

model.compile(optimizer=optimizer, loss=loss)
model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy')

loss = tf.keras.losses.BinaryCrossentropy()

model.compile(optimizer=optimizer, loss=loss)
model.compile(optimizer=optimizer, loss='binary_crossentropy')
```

Activations TF

activations

```
tf.keras.activations.relu
model = tf.keras.Sequential([tf.keras.layers.Dense(..., activation=tf.keras.activations.relu)])

tf.keras.activations.linear
model = tf.keras.Sequential([tf.keras.layers.Dense(..., activation=tf.keras.activations.linear)])


tf.keras.activations.sigmoid
model = tf.keras.Sequential([tf.keras.layers.Dense(..., activation=tf.keras.activations.sigmoid)])

tf.keras.activations.softmax
model = tf.keras.Sequential([tf.keras.layers.Dense(..., activation=tf.keras.activations.softmax)])
```


Me waiting for my neural network to finish training