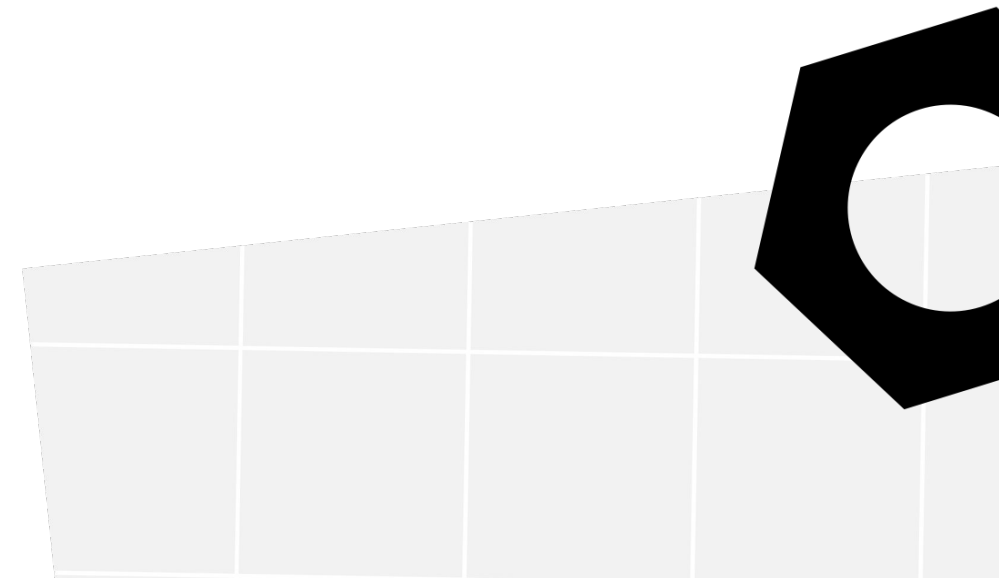




Uczenie głębokich sieci neuronowych

Kurs Data Science





Pobierz materiały do zajęć

Wprowadzenie do deep learning'u:

- [link 1](#)
- [link 2](#)

Deep learning a sztuczna inteligencja

AI (Artificial Intelligence)

Maszyny potrafią rozwiązywać zadania zwykle kojarzone z ludzką inteligencją.

ML (Machine Learning)

Wnioskowanie z doświadczenia zamiast wyłącznego bazowania na programowaniu.

DL (Deep Learning)

Machine Learning wykorzystujące warstwy sieci neuronowych.



Głębina uczenia maszynowego



Głębina uczenia głębokiego nie odwołuje się do żadnego głębokiego rozumienia osiąganego przez tę technikę - głęboką jest tworzenie wielowarstwowej reprezentacji danych.

Głębokością modelu określamy liczbę warstw modelu danych.

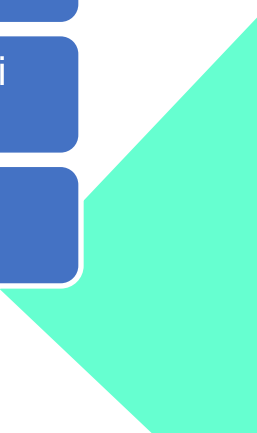
Uczenie głębokie mogłoby również nazywać się uczeniem reprezentacji warstwowych i uczeniem reprezentacji hierarchicznych.

Współcześnie uczenie głębokie.

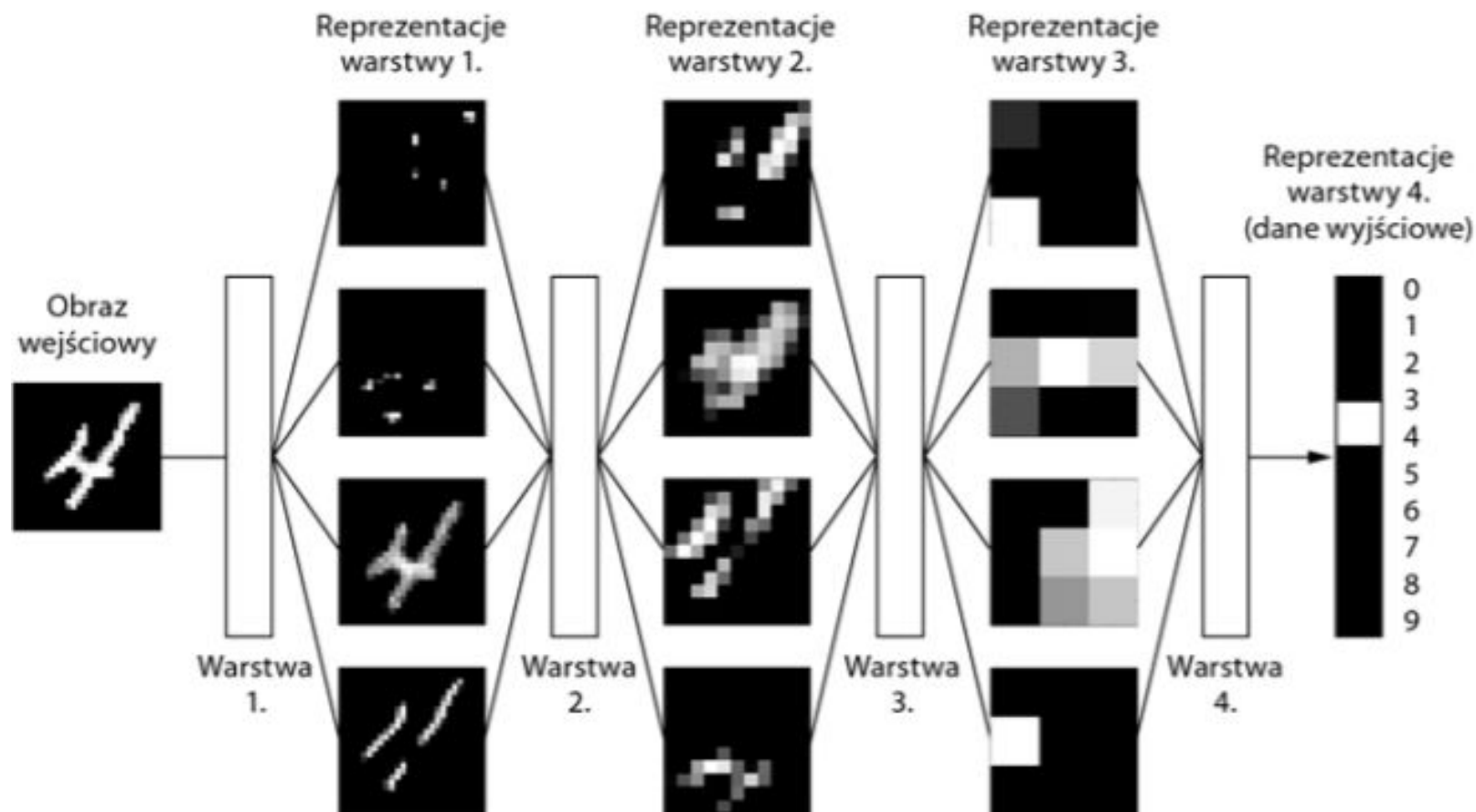
Polega to na tworzeniu dziesiątek, a nawet setek kolejnych warstw reprezentacji - wszystkie warstwy są uczone na podstawie danych treningowych.

W przypadku uczenia głębokiego wielowarstwowe reprezentacje danych są prawie zawsze tworzone za pomocą modeli określanych mianem sieci neuronowych.

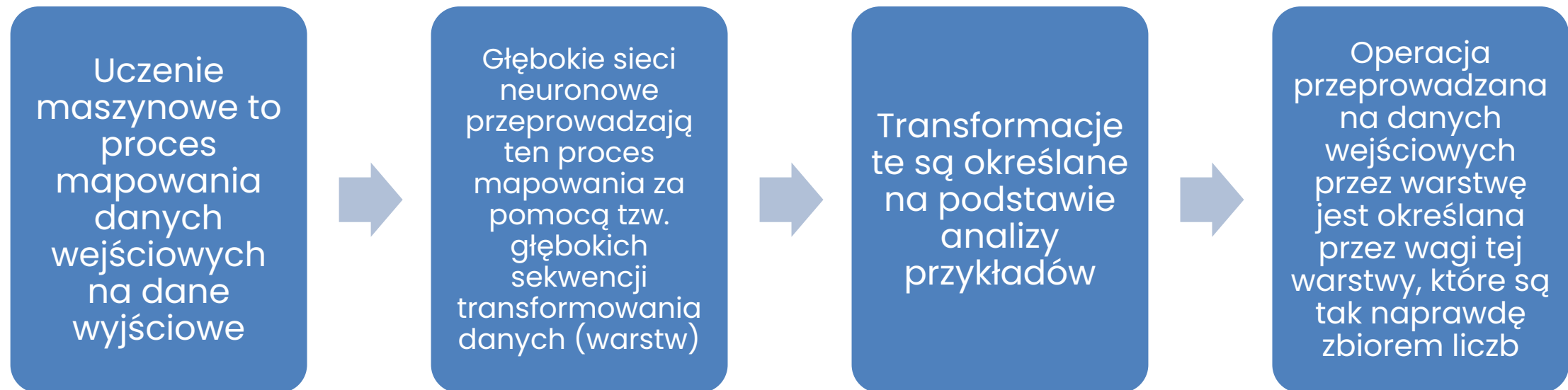
Sieci te składają się z wielu warstw neuronów.



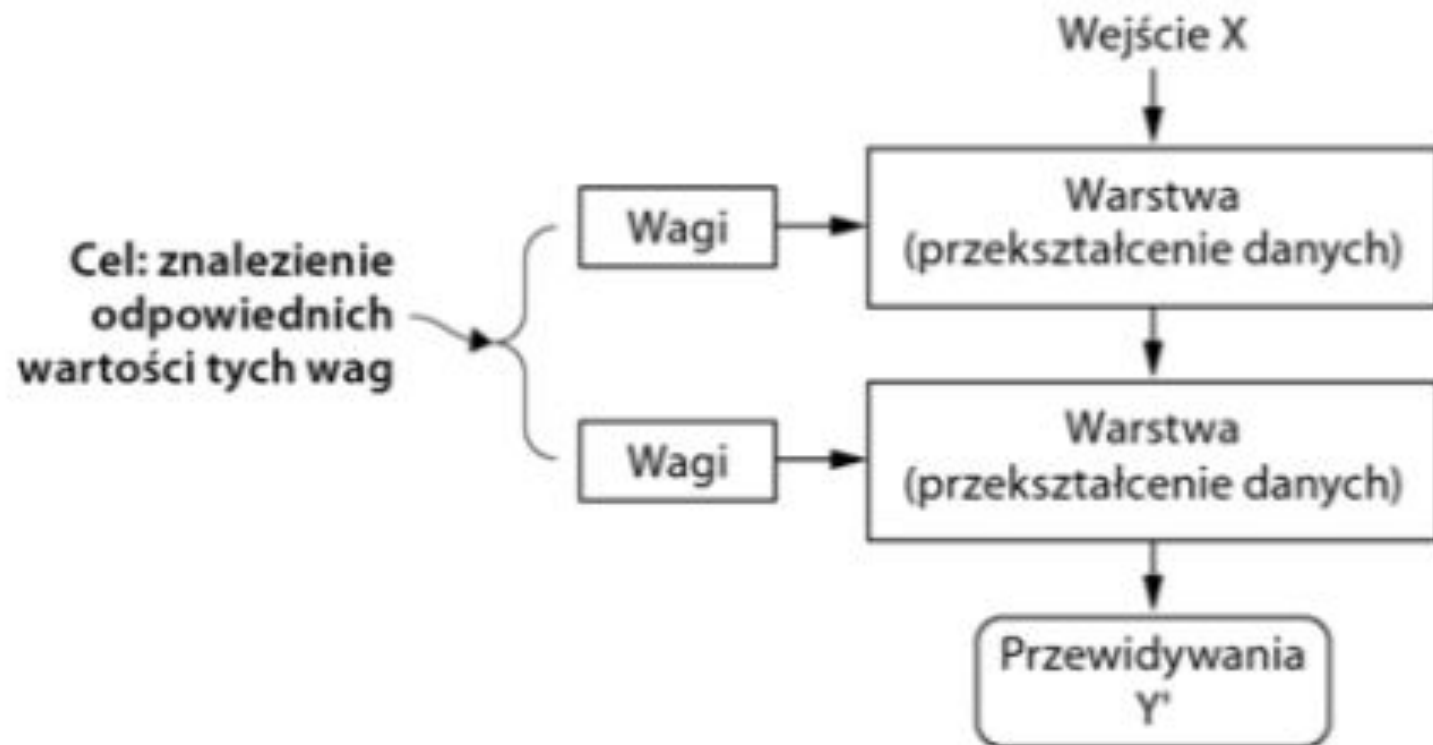
Głębokie reprezentacje utworzone przez klasyfikatora cyfr



Działanie uczenia głębokiego



Parametryzowanie warstwy przez wagi



W takim kontekście uczenie polega na znajdowaniu zbioru wartości wag wszystkich warstw sieci tak, aby sieć poprawnie mapowała dane wejściowe na dane wyjściowe.

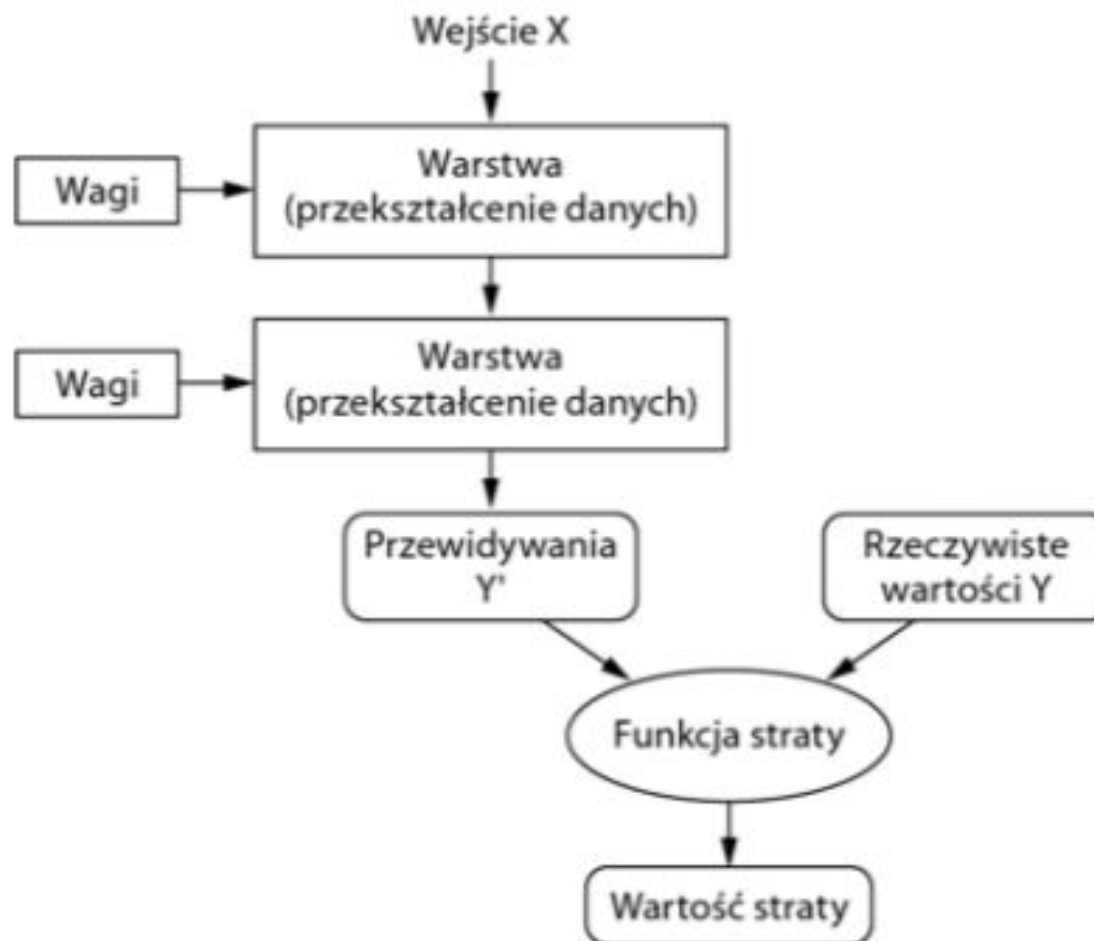
Główny problem uczenia głębokiego

Głęboka sieć neuronowa może zawierać dziesiątki milionów parametrów.

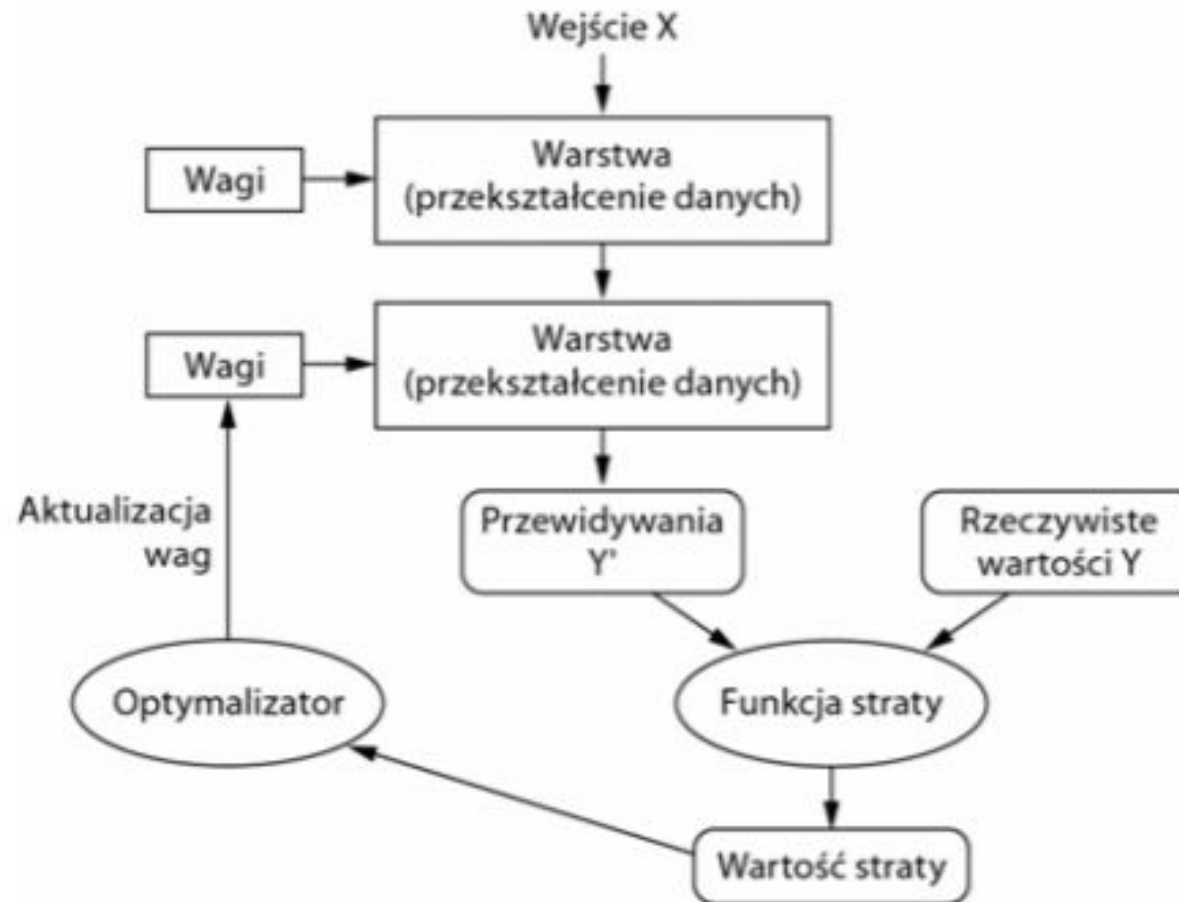
- *Jak sobie z tym radzić?*

W celu kontrolowania działania wyjść sieci neuronowej należy dysponować sposobem pomiaru tego, jak aktualnie generowane dane odbiegają od oczekiwań.

Jakość działania sieci



Informacja zwrotna umożliwiająca korektę wag





Co wyróżnia uczenia głębokie?



Dobrze radzą sobie z dużymi zbiorami danych (możliwość uczenia z wykorzystaniem procesorów graficznych).

Uczenie głębokie sprawia również, że rozwiązywanie problemów staje się o wiele łatwiejsze z powodu pełnej automatyzacji najważniejszego etapu uczenia maszynowego: **obróbki cech**.

Nie ma konieczności ręcznej obróbki cech.

Uczenie głębokie umożliwia modelowi jednoczesne łączne przetwarzanie wszystkich warstw reprezentacji.

Modyfikacja jednej wewnętrznej cechy modelu powoduje automatyczną adaptację do tej zmiany pozostałych cech bez potrzeby wykonywania operacji przez użytkownika.

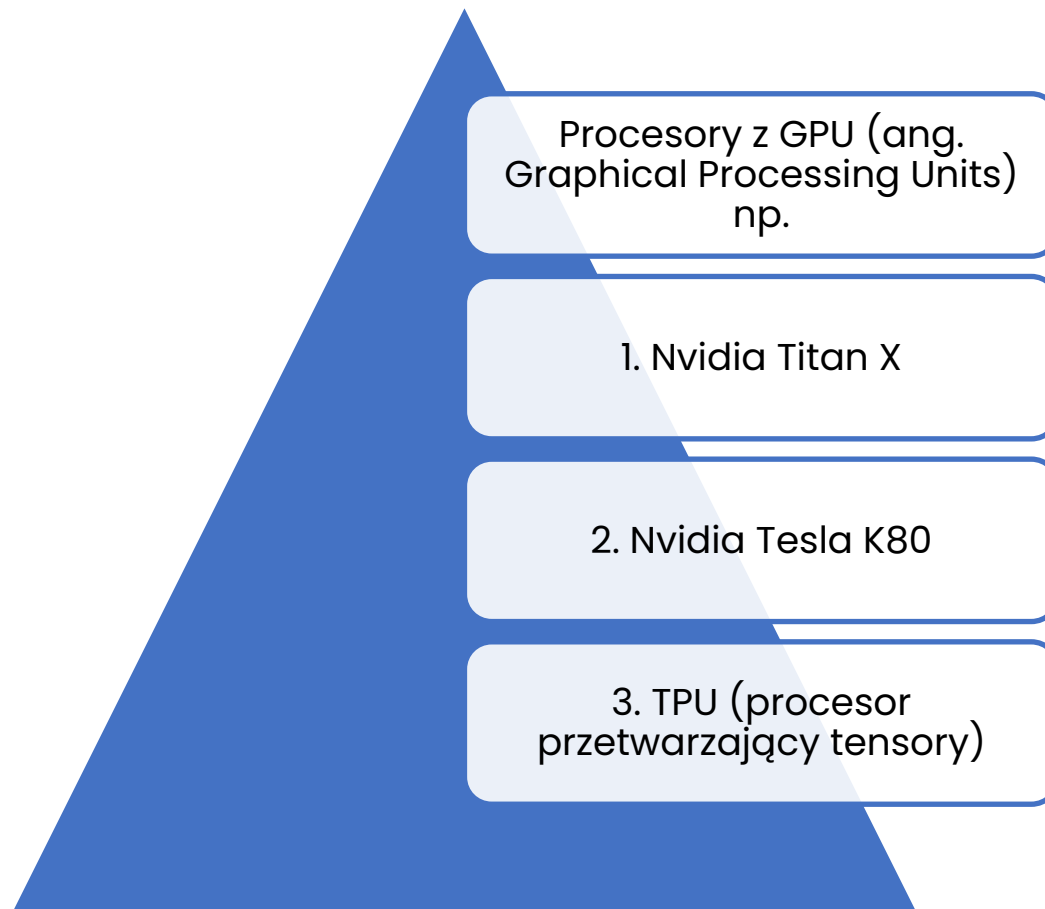


Uczenie głębokie – dlaczego teraz?

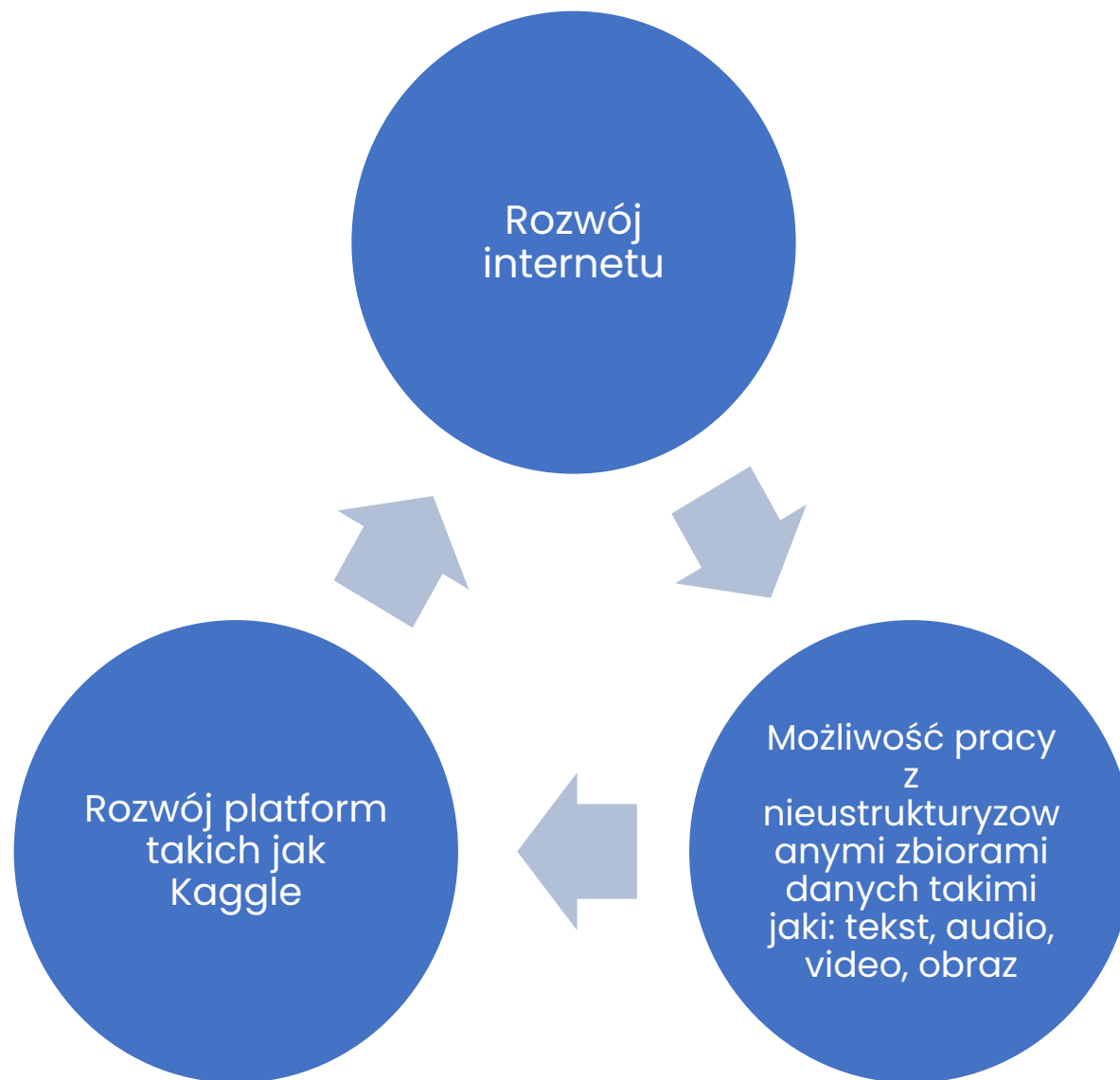
Uczenie głębokie
jest napędzane
przez rozwój
technologiczny:

- Lepszy sprzęt
- Rozwój dużych zbiorów danych
- Lepsze algorytmy

Sprzęt



Dane





Algorytmy



- lepsze funkcje aktywacji warstw neuronów,
- lepsze schematy inicjacji wag rozpoczynające się od wstępnego trenowania poszczególnych warstw (rozwiązanie to jest rzadko stosowane w praktyce),
- lepsze schematy optymalizacji, takie jak RMSProp i Adam.



Osiągnięcia uczenia głębokiego



- Klasyfikacja obrazu zbliżona do poziomu człowieka
- Rozpoznawanie mowy
- Pojawienia się asystentów takich jak Google Now czy Amazon Alexa
- Poprawa dobierania reklam w usługach Google
- Pokonanie człowieka w grę Go



Architektura sieci

```
from keras import models  
from keras import layers  
network = models.Sequential()  
network.add(layers.Dense(512, activation='relu',  
input_shape=(28 * 28,)))  
network.add(layers.Dense(10, activation='softmax'))
```



Architektura sieci



- Główny blok składowy sieci neuronowej to **warstwa** – moduł przetwarzania danych, który można traktować jako filtr danych.
- Dane wychodzące z filtra mają bardziej przydatną formę od danych do niego wchodzących.
- Niektóre warstwy dokonują ekstrakcji reprezentacji kierowanych do nich danych – reprezentacje te powinny ułatwiać rozwiązanie problemu, z którym się zmagamy.



Elementy niezbędne na etapie przygotowania sieci do trenowania

- **Funkcja straty** – funkcja ta definiuje sposób pomiaru wydajności sieci podczas przetwarzania treningowego zbioru danych, a więc pozwala na dostrajanie parametrów sieci we właściwym kierunku.
- **Optymalizator** – mechanizm dostrajania sieci na podstawie danych zwracanych przez funkcję straty.
- Metryki monitorowane podczas trenowania i testowania – tutaj interesują nas metryki dokładności(jakościowe).



Etap kompilacji

```
network.compile(optimizer='rmsprop', loss='categorical_crossentropy',  
metrics=['accuracy'])
```

Zanim rozpoczniemy trenowanie, zmienimy kształt danych tak, aby przyjęły kształt oczekiwany przez sieć, i przeskalujemy je do wartości z zakresu $[0, 1]$. Początkowo nasze obrazy treningowe były zapisywane w postaci macierzy o wymiarach $(60000, 28, 28)$, zawierającej wartości z zakresu $[0, 255]$, i typie `uint8`. Przekształcamy je w tablicę typu `float32` o wymiarach $(60000, 28 * 28)$, zawierającą wartości od 0 do 1.



Etap kompilacji

```
train_images = train_images.reshape((60000, 28 * 28))  
train_images = train_images.astype('float32') / 255  
  
test_images = test_images.reshape((10000, 28 * 28))  
test_images = test_images.astype('float32') / 255
```



Przygotowywanie etykiet

```
from keras.utils import to_categorical  
train_labels = to_categorical(train_labels)  
test_labels = to_categorical(test_labels)
```



Reprezentacja danych sieci neuronowych



- Skalary
- Wektory
- Macierze
- Tensory trójwymiarowe
- Tensory wyższych rzędów

Główne atrybuty tensorów

Liczba osi (ranga) — trójwymiarowy tensor ma trzy osie, a macierz ma dwie osie. W bibliotece Numpy atrybut ten określa parametr `ndim`.

Kształt — krótka wartości całkowitoliczbowych określających liczbę wymiarów poszczególnych osi tensora.

Typ danych (w większości bibliotek Pythona jest to parametr `dtype`) - typ danych umieszczonych w tensorze. Tensor może zawierać dane typu `float32`, `uint8`, `float64` itd.



Prawdziwe przykłady tensorów danych



dane wektorowe – tensory dwuwymiarowe mające kształt (próbki, cechy);

dane szeregu czasowego lub dane sekwencyjne – tensory trójwymiarowe mające kształt (próbki, czas, cechy);

obrazy – tensory czterowymiarowe mające kształt (próbki, wysokość, szerokość, kanały) lub (próbki, kanały, wysokość, szerokość);

materiały wideo – tensory pięciowymiarowe mające kształt (próbki, klatki, wysokość, szerokość, kanały) lub (próbki, klatki, kanały, wysokość, szerokość)

Trenowanie algorytmu

- Każda warstwa sieci neuronowej wykonuje poniższą operację przekształcającą dane:

$$\text{output} = \text{relu}(\text{dot}(W, \text{input}) + b)$$

W tym wypadku W i b to parametry algorytmu lub inaczej mówiąc wagi. Inaczej nazywane kernel (jądro) i bias (próg).

- Na początku zachodzi zjawisko tzw. *losowej inicjalizacji* (wypełniamy wartościami bliskimi zeru).
- Potem wagi są stopniowo dostrajane na podstawie sygnału zwrotnego (*trenowanie modelu*).

Pętla trenowania

1. Wygeneruj wsad składający się z próbek treningowych x i odpowiadających im wartości docelowych y .

3. Oblicz stratę sieci na próbce danych i zmierz różnicę między wartościami y_{pred} i y .

2. Uruchom sieć na próbkach x (proces ten nazywamy przebiegiem w przód) w celu uzyskania przewidywanych wartości y_{pred} .

4. Aktualizuj wartości wszystkich wag sieci tak, aby nieco zredukować stratę uzyskaną na tej próbce.

Optymalizacja gradientowa – SGD

1. Wygeneruj wsad składający się z próbek treningowych x i odpowiadających im wartości docelowych y .

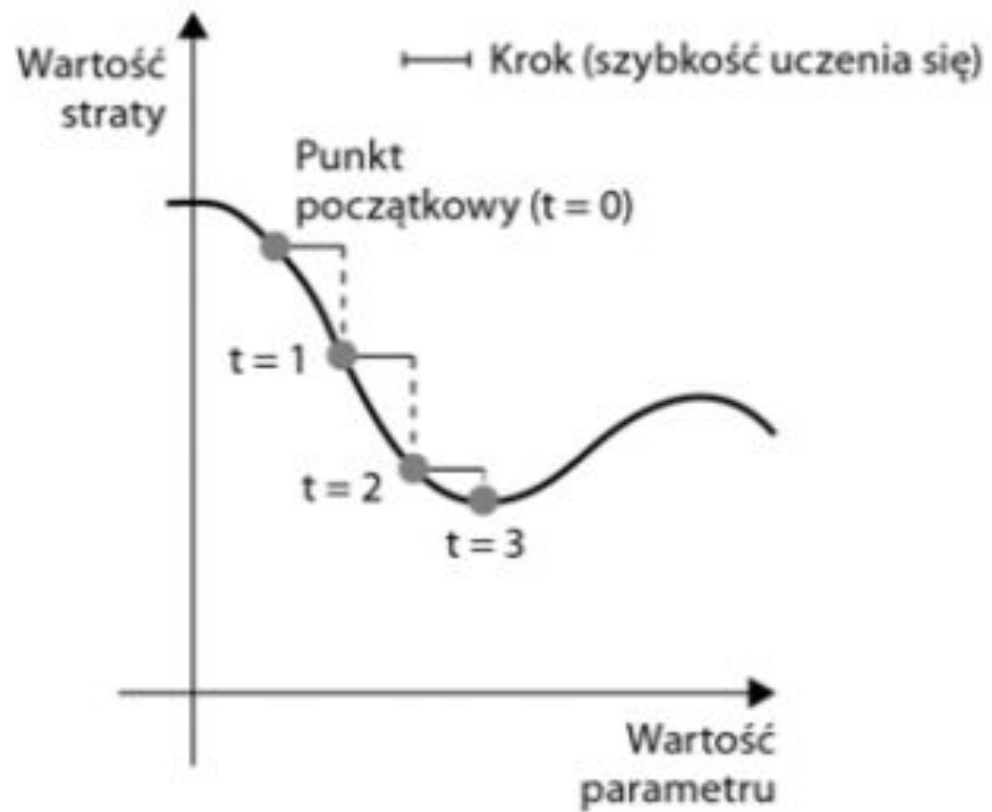
3. Oblicz stratę sieci na próbce danych i zmierz różnicę między wartościami y_{pred} i y .

5. Przesuń wartości parametrów w kierunku przeciwnym do gradientu – np. korzystając z równania $W -= step * gradient$, co powinno nieco zredukować stratę uzyskaną na bieżącej próbce danych treningowych.

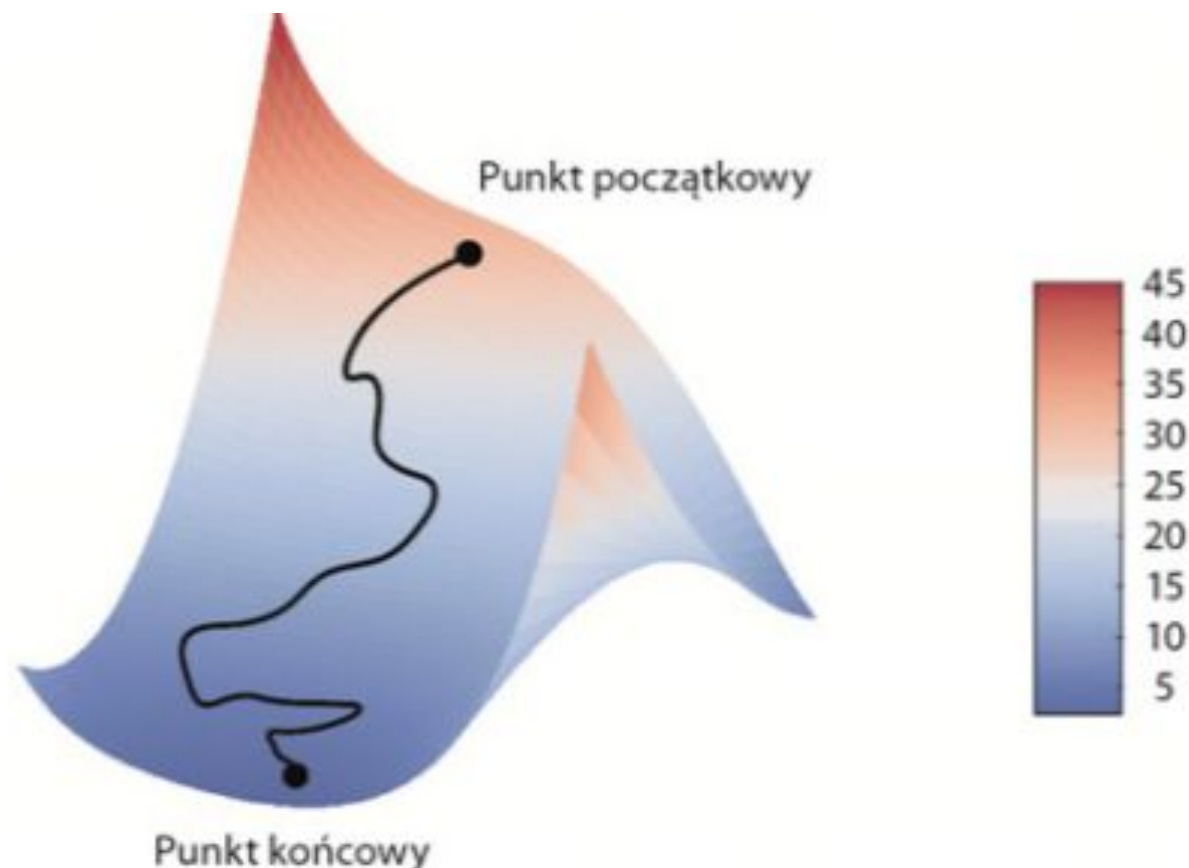
2. Uruchom sieć na próbkach x w celu uzyskania przewidywanych wartości y_{pred} .

4. Oblicz gradient straty w odniesieniu do parametrów sieci (przejdźcie w tył).

Algorytm SGD



Spadek gradientowy w przestrzeni 2D



Aby zwizualizować to zagadnienie wyobraź sobie, że w jednym garnku gotujesz zupę dla 300 osób. Aby zupa się zmieściła garnek musi być olbrzymi. We wnętrzu tak wielkiego naczynia temperatura rozkłada się nierównomiernie – gdzieś będzie cieplej a gdzieś zimniej. Na szczęście masz pod ręką funkcję $T(x, y, z)$, która mówi o temperaturze w garnku. Zmienne x, y, z to konkretny punkt w przestrzeni, w którym ta funkcja zwraca nam temperaturę. Wówczas gradient funkcji T wskazywałby nam kierunek, gdzie temperatura jest najmniejsza.

Łączenie pochodnych – algorytm propagacji wstecznej

Wyobraźmy sobie sieć neuronową składającą się z 3 operacji tensorowych (a, b, c) oraz 3 wag (W1, W2, W3)

$$f(W1, W2, W3) = a(W1, b(W2, c(W3)))$$

Propagacja wsteczna rozpoczyna się od ostatniej wartości straty i przetwarza warstwy odwrotnie (od góry do dołu) z zastosowaniem reguły łańcuchowej w celu obliczenia wpływu każdego parametru na wartość straty

Podsumowanie powyższego

- **Trenowanie** – polega na ustalaniu kombinacji parametrów modelu minimalizujących funkcję straty na podstawie próbek z treningowego zbioru danych i ich etykiet.
- Trenowanie jest przeprowadzane poprzez tworzenie losowego zbioru próbek wraz z etykietami i obliczanie gradientu parametrów sieci z uwzględnieniem straty wynikłej podczas przetwarzania zbioru próbek. Parametry sieci są następnie nieco poprawiane (rozmiar korekty parametrów zależy od zdefiniowanego tempa uczenia się sieci) – ich wartości są zmieniane w kierunku przeciwnym do gradientu.
- Cały proces nauki jest możliwy dzięki temu, że sieci neuronowe są łańcuchami różniczkowalnych operacji tensorowych, a także dzięki temu, że możliwe jest stosowanie reguły łańcuchowej różniczkowania w celu określenia funkcji gradientu przypisującej bieżące parametry i bieżącą próbkę danych do wartości gradientu.
- W kolejnych rozdziałach często będę odwoływał się do koncepcji **straty** i **optymalizatorów**. Muszą być one zdefiniowane przed rozpoczęciem trenowania i sieci.
- **Strata** jest wartością, którą sieć będzie próbowała minimalizować podczas trenowania, a więc powinna być miarą sukcesu w rozwiązaniu danego problemu.
- **Optymalizator** określa dokładnie sposób używania gradientu straty podczas aktualizacji parametrów: w sieci neuronowej możesz zastosować np. algorytm optymalizatora RMSProp czy algorytm stochastycznego spadku wzdłuż gradientu z pędem.



Elementy sieci neuronowej



- warstwy, które po połączeniu ze sobą tworzą sieć (lub model)
- dane wejściowe i odpowiadające im docelowe etykiety
- funkcja straty, która definiuje sygnał zwrotny używany w procesie uczenia
- optymalizator, który określa przebieg trenowania

Warstwy – podstawowe bloki konstrukcyjne uczenia głębokiego

- Proste dane wektorowe przetwarzamy za pomocą sieci gęsto połączonych lub w pełni połączonych (w keras klasa Dense).
- Próbki znaczniki czasu, dane sekwencyjne przetwarza się zazwyczaj za pomocą warstw rekurencyjnych (np. warstwy LSTM).
- Dane obrazu przekazuje się zazwyczaj w tensorach czterowymiarowych za pomocą 2 – wymiarowych warstw konwolucyjnych (popr. Splotowych).
- *Warstwy można porównać do klocków LEGO uczenia głębokiego. Z metafory tej korzystali między innymi twórcy pakietu Keras.*

```
from keras import layers
```

```
layer = layers.Dense(32, input_shape=(784,))
```

← **Warstwa gęsta z 32 jednostkami wyjściowymi.**

Funkcje aktywacji

Czym są funkcje aktywacji i dlaczego musimy z nich korzystać?

Warstwą Dense bez funkcji aktywacyjnej takiej jak funkcje relu (zwana również funkcją nieliniową) składałaby się z dwóch operacji liniowych – iloczynu skalarnego i dodawania: $\text{output} = \text{dot}(W, \text{input}) + b$

W związku z tym warstwa mogłaby uczyć się tylko **przekształceń liniowych** (przekształceń afinicznych) danych wejściowych. **Przestrzeń hipotez** takiej warstwy byłaby zestawem wszystkich możliwych przekształceń liniowych zbioru danych w przestrzeni o 16 wymiarach. Taka przestrzeń hipotez jest zbyt ograniczona i nie wykorzystalibyśmy tego, że dysponujemy wieloma warstwami reprezentacji (głęboki stos warstw liniowych implementowałby tylko operacje liniowe i dodawanie kolejnych warstw nie rozszerzałoby przestrzeni hipotez).

W celu uzyskania dostępu do o wiele bogatszej przestrzeni hipotez i skorzystanie z możliwości oferowanych przez głębokie reprezentacje danych potrzebujemy nieliniowości zapewnionej przez funkcje aktywacji. Funkcją najczęściej stosowaną w uczeniu głębokim tego typu jest funkcja relu, ale istnieje wiele innych funkcji o podobnie dziwnych nazwach: prelu, elu itd.



Nadmierne dopasowanie



- Crossvalidacja
- Więcej danych
- Redukcja rozmiaru sieci
- Regularyzacja?



Regularyzacja



- Regularyzacja L1 – koszt jest dodawany proporcjonalnie do bezwzględnej wartości współczynników wag (normy L1 wag).
- Regularyzacja L2 – koszt jest dodawany proporcjonalnie do kwadratu wartości współczynników wag (normy L2 wag). W kontekście sieci neuronowych regularyzacja L2 jest również określana mianem rozkładu wag. Pomimo innej nazwy jest to ten sam proces, który w matematyce określamy jako regularyzacja L2.

Dodawanie regularyzacji L2 do wag modelu

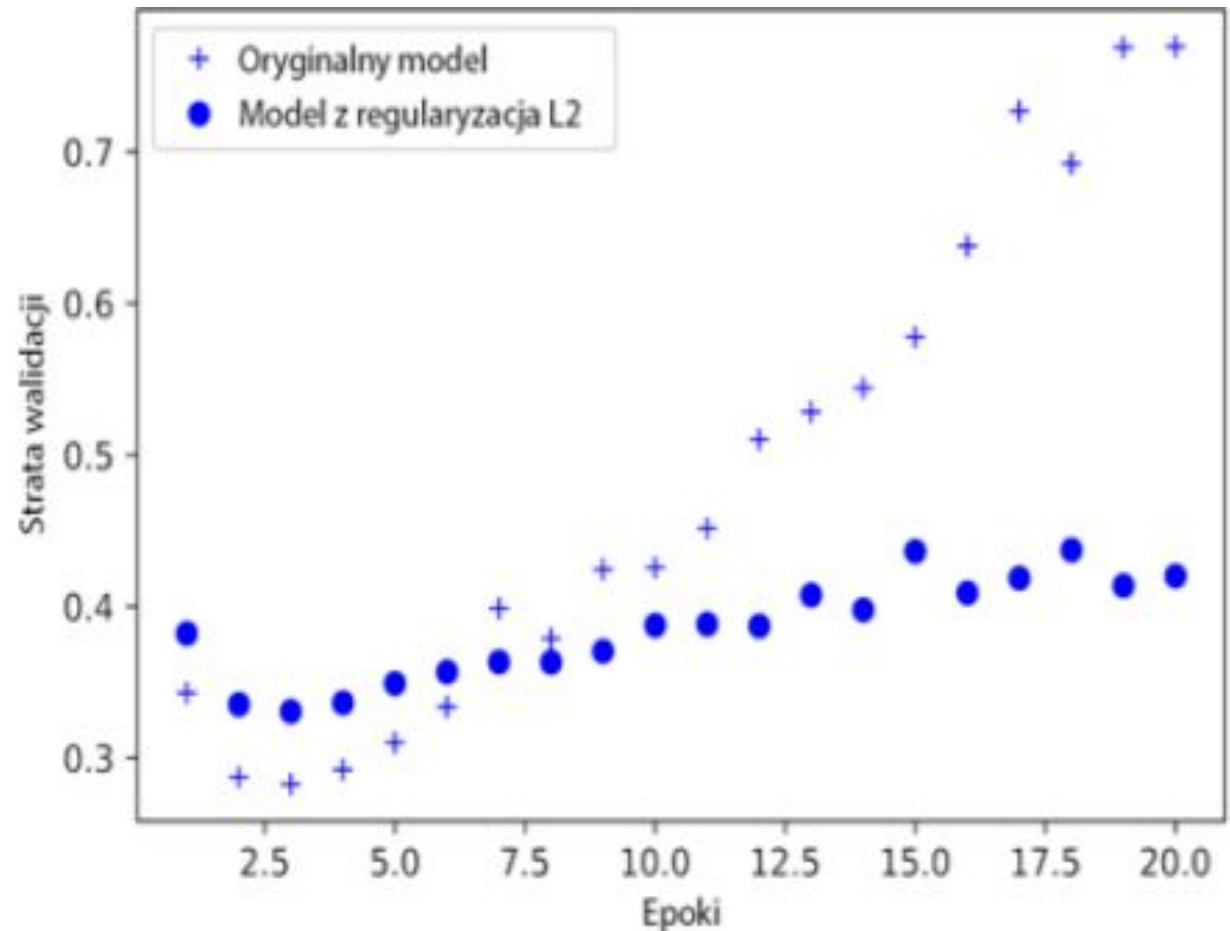
```
from keras import regularizers
12_model = models.Sequential()
12_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                           activation='relu', input_shape=(10000.)))
12_model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                           activation='relu'))
12_model.add(layers.Dense(1, activation='sigmoid'))
```

Argument `l2(0.001)` oznacza, że każdy współczynnik macierzy wag warstwy doda wartość równą $0.001 * \text{weight_coefficient_value}$ (0,001 razy wartość współczynnika wagi) do całkowitej straty sieci. Kara ta jest dodawana tylko podczas trenowania, a więc strata sieci w czasie trenowania będzie o wiele wyższa niż w czasie testowania.

Regularyzacja L2

Na rysunku pokazano wpływ kary w postaci regularyzacji L2.

Jak widać, model z regularyzacją L2 (kropki) stał się o wiele bardziej odporny na nadmierne dopasowanie od modelu referencyjnego (krzyżyki) pomimo tego, że oba modele charakteryzują się identyczną liczbą parametrów.



Metody regularyzacji dostępne w Keras

```
from keras import regularizers
```

```
regularizers.l1(0.001) ← Regularyzacja L1.
```

```
regularizers.l1_l2(l1=0.001, l2=0.001) ← Jednoczesna regularyzacja L1 i L2.
```



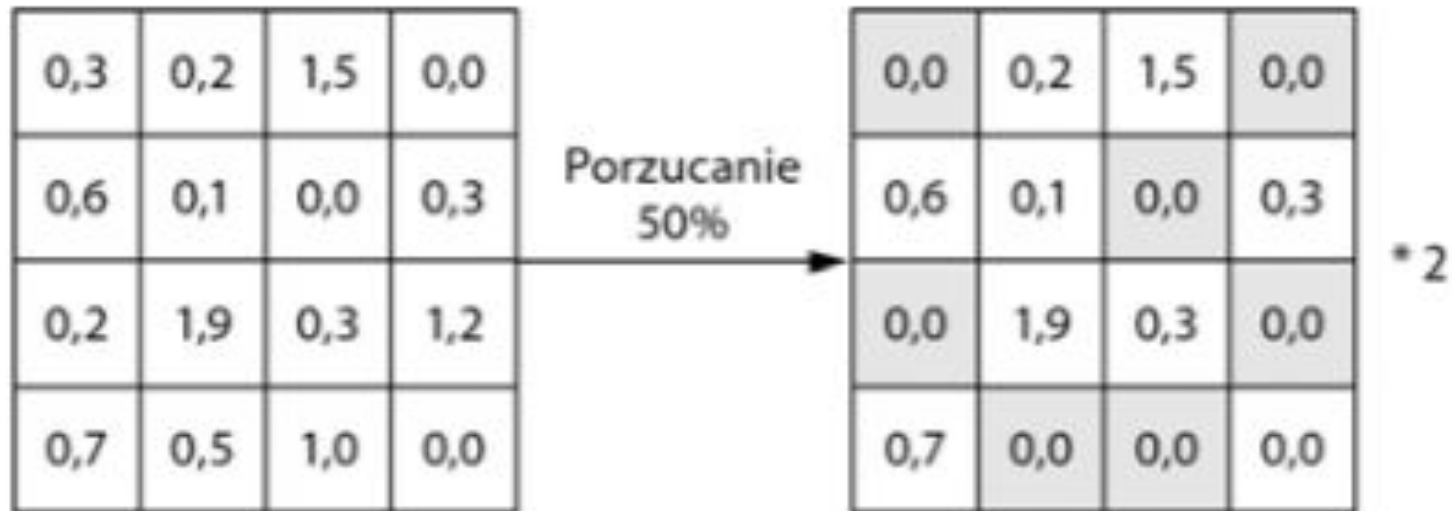

Porzucanie – technika dropout



Technika ta polega na losowym wybieraniu pewnej liczby cech wyjściowych warstwy podczas trenowania (wartości tych warstw są zastępowane zerami).

Podczas testowania żadne jednostki nie są porzucane – wartości wyjściowe warstwy sieci są skalowane o współczynnik równy współczynnikowi porzucania. Równoważy to aktywność większej liczby jednostek podczas testowania niż trenowania.

Porzucanie użyte na macierzy aktywacji





Jak radzić sobie z nadmiernym dopasowaniem?

- Zdobyć większej ilości danych treningowych.
- Redukcja pojemności sieci.
- Dodanie regularyzacji wag.
- Dodanie mechanizmu porzucania.

Wybór odpowiedniej funkcji aktywacji ostatniej warstwy

Rodzaj problemu	Funkcja aktywacji ostatniej warstwy	Funkcja straty
Klasyfikacja binarna	sigmoid	binary_crossentropy
Wieloklasowa klasyfikacja jednoetykietowa	softmax	categorical_crossentropy
Wieloklasowa klasyfikacja wieloetykietowa	sigmoid	binary_crossentropy
Regresja dowolnych wartości	Brak	mse
Regresja wartości z zakresu od 0 do 1	sigmoid	mse lub binary_crossentropy



Regularyzacja modelu i dostrajanie hiperparametrów – podsumowanie

mechanizm porzucania;

zastosowanie różnych architektur poprzez dodawanie lub usuwanie warstw;

dodanie regularyzacji L1 lub L2;

definiowanie różnych hiperparametrów modelu, takich jak liczba jednostek w poszczególnych warstwach i współczynnik uczenia się optymalizatora (pozwoli to na znalezienie optymalnej konfiguracji modelu);

przeprowadzenie dodatkowej inżynierii cech: dodanie nowych cech, usunięcie cech, które wydają się nie zawierać przydatnych informacji.



Polecane materiały związane z tematyką bloku

- <https://keras.io/>
- <https://www.ibm.com/pl-pl/cloud/deep-learning>