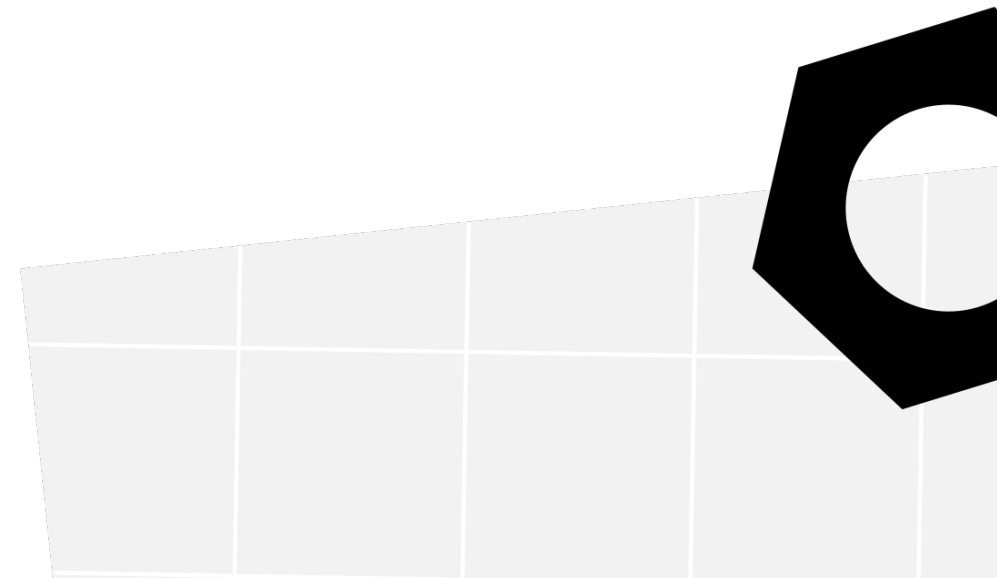




Wprowadzenie do sztucznych sieci neuronowych

Kurs Data Science





Pliki do pobrania



- Pakiety, z których będziemy korzystać podczas zajęć:

```
jupyter==1.0.0  
pandas==1.1.4  
scikit-learn==0.23.2
```

- Będziemy korzystać z notebooka z przykładami na zajęcia. Możesz go pobrać z sekcji “Dodatkowe materiały do bloku” [tutaj](#).



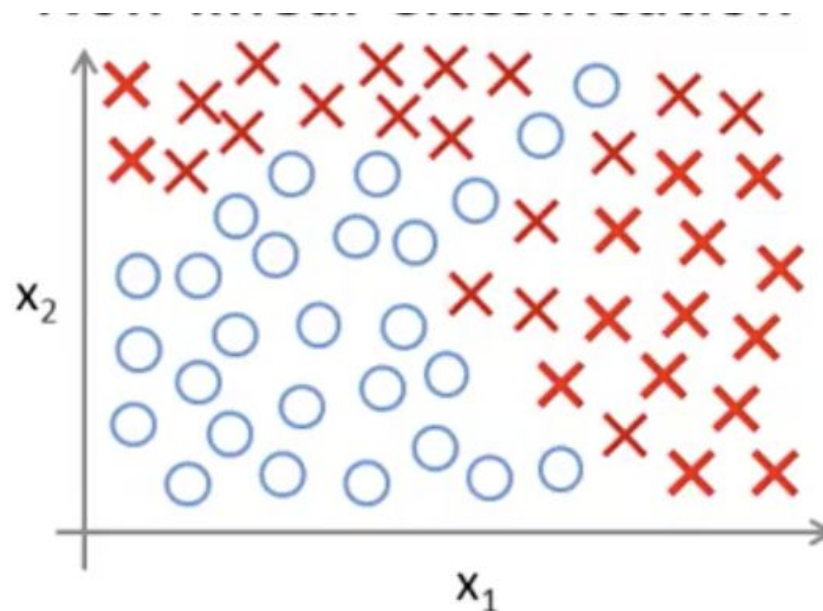
Złożone problemy nieliniowe



Znamy już kilka algorytmów (regresja liniowa, drzewa decyzyjne, SVM), po co mamy więc poznawać kolejne, sztuczne sieci neuronowe?

Złożone problemy nieliniowe

Niektóre problemy nieliniowe nie dadzą się rozwiązać za pomocą poznanych już wcześniej metod – do rozwiązania zadania klasyfikacji poniżej model musiałby korzystać z wielu zmiennych wielomianowych, co byłoby bardzo kosztowne pamięciowo, czasowo i mogłoby prowadzić do zjawiska przeuczenia.





Złożone problemy nieliniowe

Do rozwiązywania nieliniowych problemów z dużą liczbą cech używanych przy predykcji, dużo lepszym wyborem będą właśnie **sztuczne sieci neuronowe**, które zostały stworzone, by naśladować sposób, w jaki działa ludzki mózg.

Sieci neuronowe w wielu skomplikowanych zastosowaniach (choćby w problemach przetwarzania języka naturalnego czy widzenia komputerowego) osiągają najlepsze wyniki.



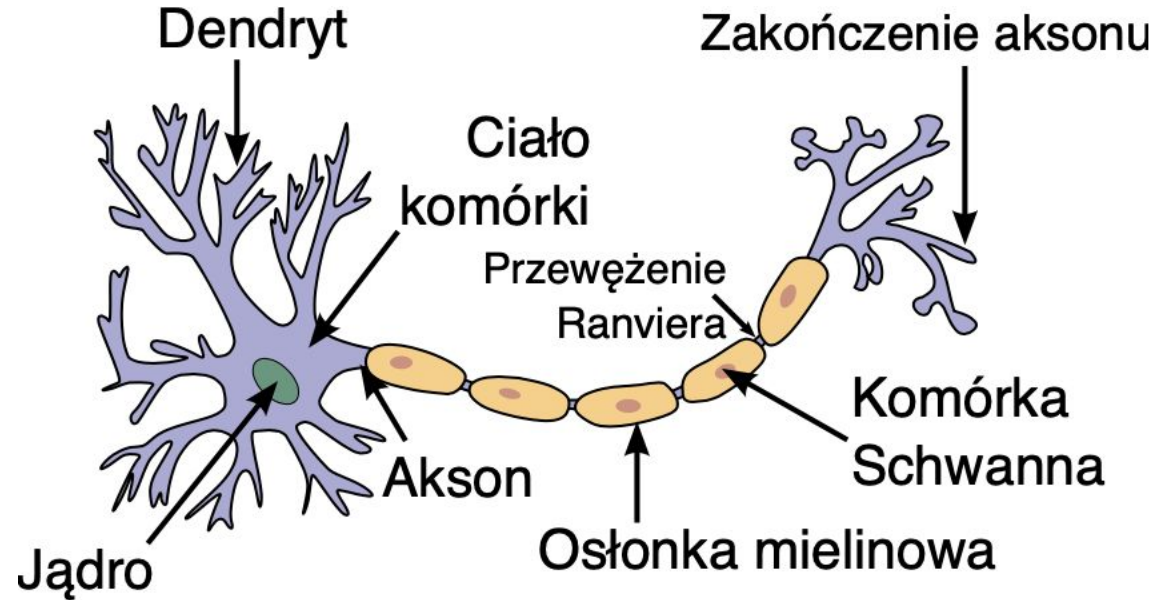
Sieci neuronowe



Stworzone w latach 80., jednak swój rozkwit przeżyły w późnych latach 90. i trwa on do dziś, ze względu na wciąż rosnącą moc obliczeniową komputerów.

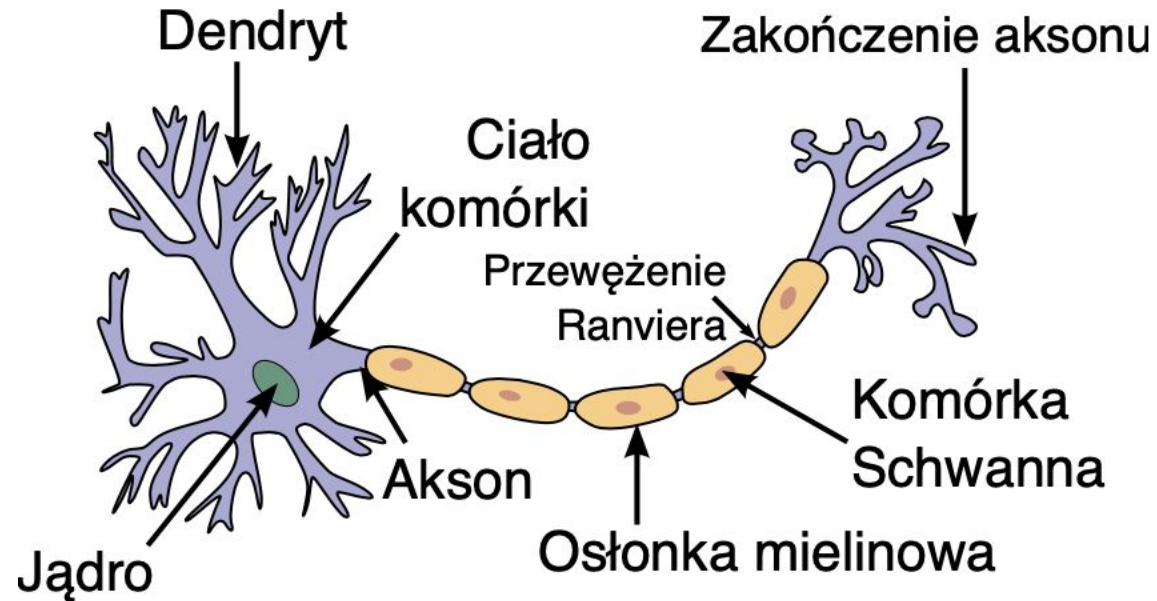
Wzorowane na działaniu ludzkiego mózgu, który zawiera wiele połączonych ze sobą neuronów i przekazujących impulsy elektryczne, na podstawie których wypracowuje odpowiednie reakcje na bodźce.

Neurony



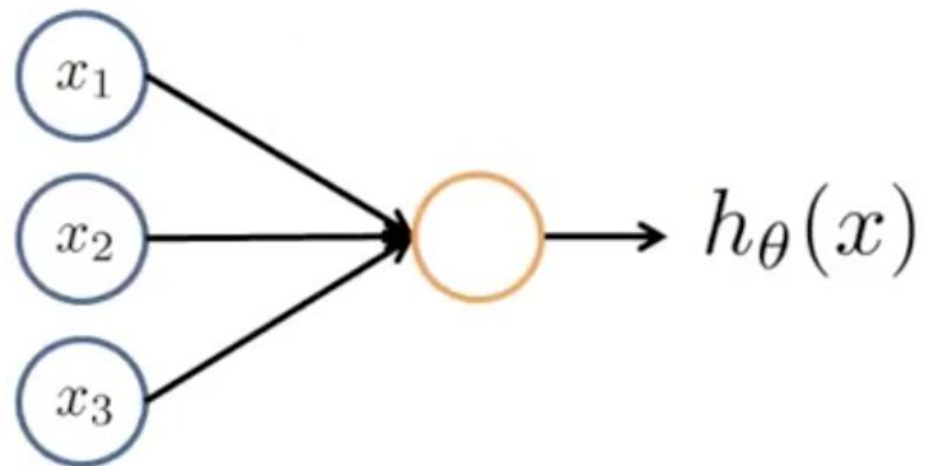
Na powyższej grafice zaprezentowano pojedynczy neuron. Każdy z nich przyjmuje sygnały elektryczne poprzez dendryty, przepuszcza przez swoje ciało, a następnie przekazuje dalej przez akson.

Neurony



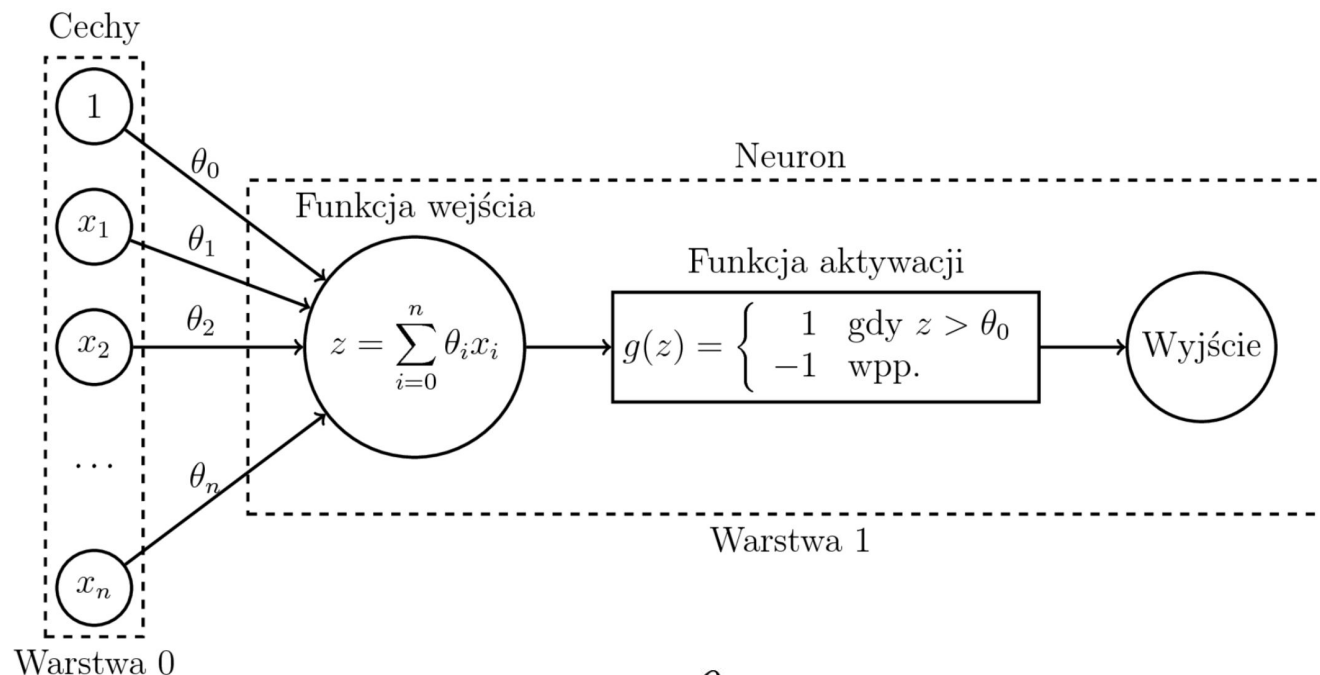
W uproszczeniu możemy więc rozumieć, że każdy neuron przyjmuje sygnał wejściowy, procesuje go dokonując pewnych operacji, a następnie przekazuje dalej poprzez akson, do kolejnego neuronu, którego dendryty są z nim połączone.

Sztuczny neuron



Ideę neuronu możemy przełożyć na model matematyczny - mamy tutaj dane wejściowe, które są procesowane przez sztuczny neuron, dokonywane są pewne przekształcenia przy pomocy funkcji h i wyjście jest przekazywane dalej.

Sztuczny neuron



Każdej cenie wejściowej odpowiada waga θ_i , przez którą dana cecha x będzie pomnożona. Następnie wszystkie te iloczyny są sumowane, a ta suma przepuszczana przez funkcję aktywacji, której rezultat jest wyjściem sieci, przekazywanym dalej.

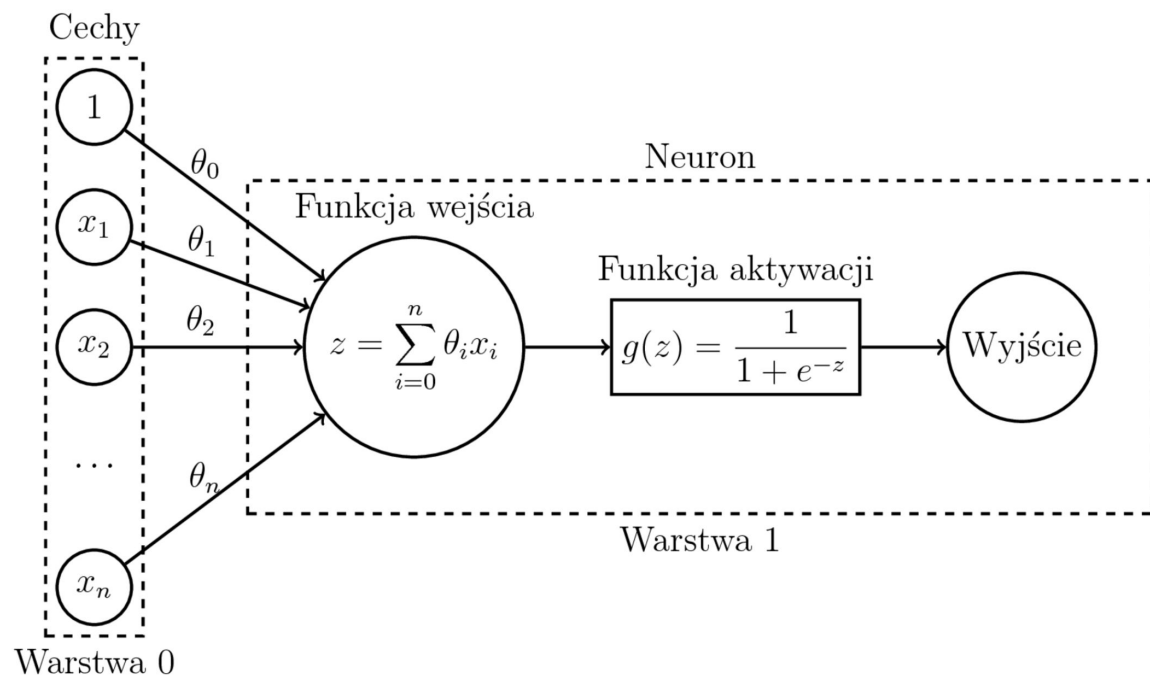


Perceptron



Możliwości pojedynczych sztucznych neuronów są bardzo ograniczone, jednak poprzez połączenie sztucznych neuronów w sieci ich możliwości bardzo gwałtownie wzrastają i ujawnia się rzeczywista moc obliczeń neuronowych. Jest ona wynikiem jednoczesnej (równoległej) pracy wielu neuronów połączonych w sieci tworzące różnorodne struktury (architektury). **Perceptronem** nazywamy najprostszą sieć neuronową, złożoną z jednego lub kilku neuronów przedstawionych na poprzednich slajdach.

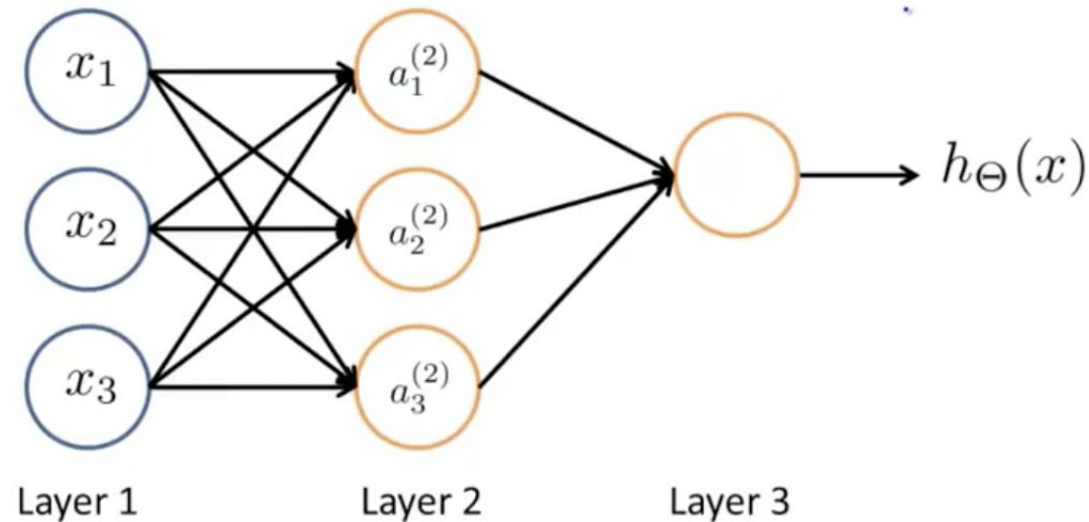
Perceptron a regresja logistyczna



Najprostszy perceptron z sigmoidalną funkcją aktywacji możemy utożsamiać z binarną regresją logistyczną: sumujemy iloczyny cech i wag, następnie obliczamy wartość funkcji logistycznej, a na podstawie ustalonej wartości odcięcia (threshold) ustalamy przypisaną klasę.

Mówiąc "sieci neuronowe" będziemy jednak mieli na myśli trochę bardziej skomplikowane architektury.

Sieć neuronowa



Mamy tu przykład kilku neuronów połączonych ze sobą i tworzących sieć. Wyróżniamy 3 warstwy: wejściową (której przekazujemy dane), złożoną z 3 neuronów, ukrytą (dlaczego nie środkowa? Bo możemy mieć wiele warstw między wejściową, a wyjściową), złożoną z 3 neuronów oraz wyjściową (która jest rezultatem działania sieci) – z jednego.



Sieć neuronowa

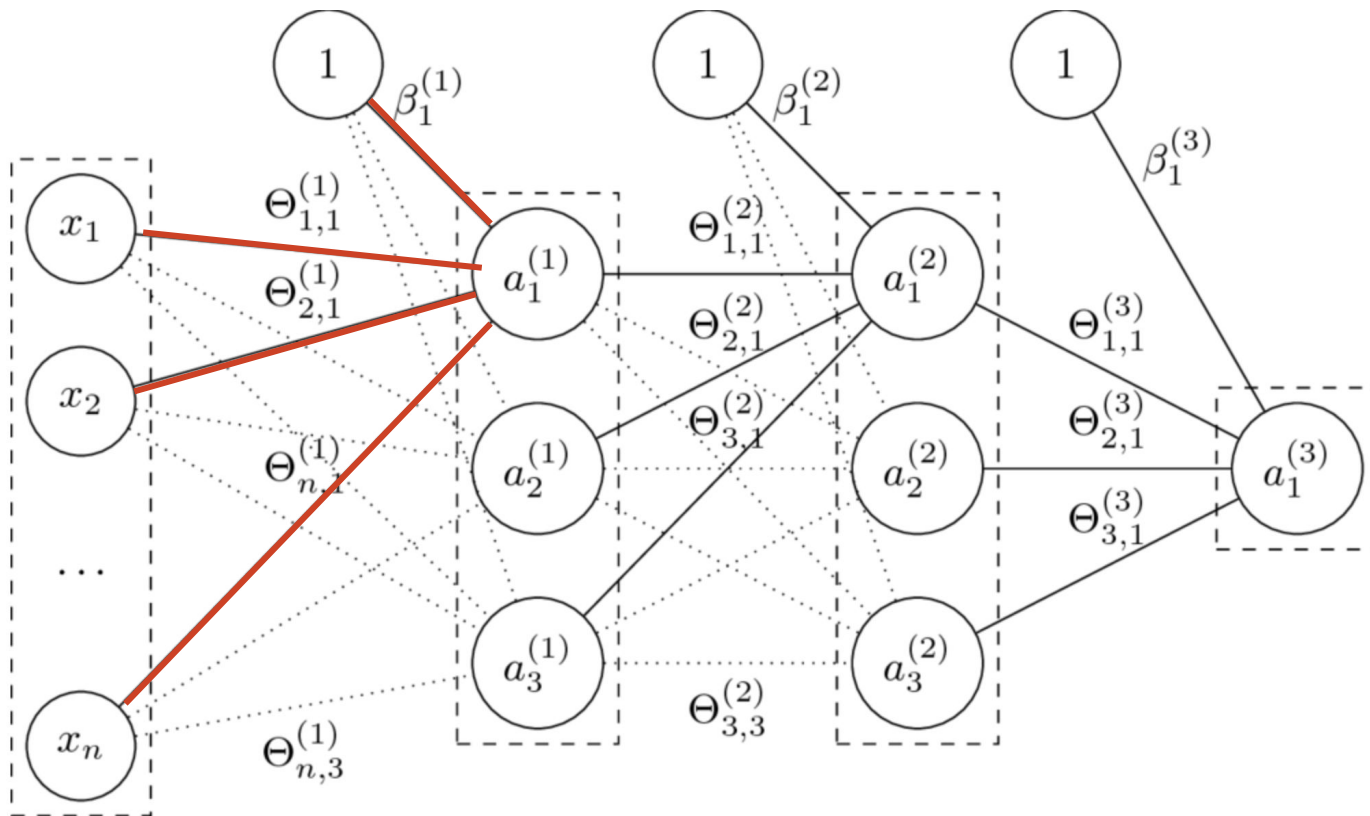


Liczba neuronów w każdej z warstw może być różna, przy czym w danej warstwie wszystkie neurony mają taką samą funkcję aktywacji.

Połączenia występują tylko pomiędzy neuronami z sąsiednich warstw, wg. zasady „każdy z każdym” i mają one charakter asymetryczny.

Sygnały przesyłane są od warstwy wejściowej poprzez warstwy ukryte (jeśli występują) do warstwy wyjściowej (w jednym kierunku). Zadaniem neuronów z warstwy wejściowej jest wstępna obróbka sygnału (np.: normalizacja, kodowanie itp.). Z kolei za przetwarzanie decyzyjne odpowiedzialne są neurony warstw ukrytych i warstwy wyjściowej, a odpowiedź udzielana jest przez neurony warstwy wyjściowej.

Sieć neuronowa



Ta sieć z kolei zawiera 4 warstwy: wejściową, dwie ukryte oraz wyjściową.

jest macierzą wag (parametrów) modelu, θ
a g - funkcją aktywacji. Dla każdego kolejnego neuronu aplikujemy wzór widoczny po lewej, tak wygląda przejście w przód (forward pass).

$$a_1^{(1)} = g(x_1 * \theta_{1,1}^{(1)} + x_2 * \theta_{2,1}^{(1)} + \dots + x_n * \theta_{n,1}^{(1)} + \beta_1^{(1)})$$

Liczba neuronów w warstwach ukrytych w scikit-learn

Wybór liczby neuronów w warstwach ukrytych wpływa na skuteczność modelu:

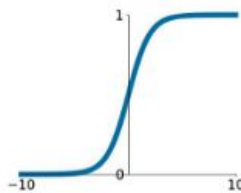
```
for hidden_size in [(10,), (100,), (10, 5), (10, 5, 3)]:  
    model = MLPClassifier(hidden_layer_sizes=hidden_size)  
    model.fit(X_train_sc, y_train)  
    y_pred = model.predict(X_test_sc)  
    print(f"Dla {hidden_size}: {f1_score(y_test, y_pred, average='macro'}}")
```

```
Dla (10,): 0.8974358974358975  
Dla (100,): 0.9709618874773142  
Dla (10, 5): 0.9709618874773142  
Dla (10, 5, 3): 0.9453132832080202
```


Funkcja aktywacji

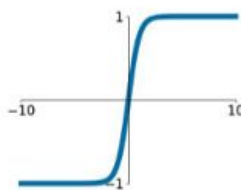
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



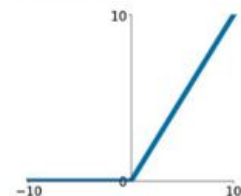
tanh

$$\tanh(x)$$



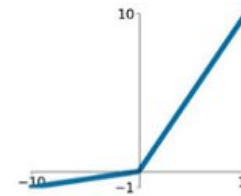
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

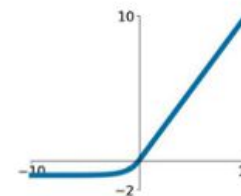


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



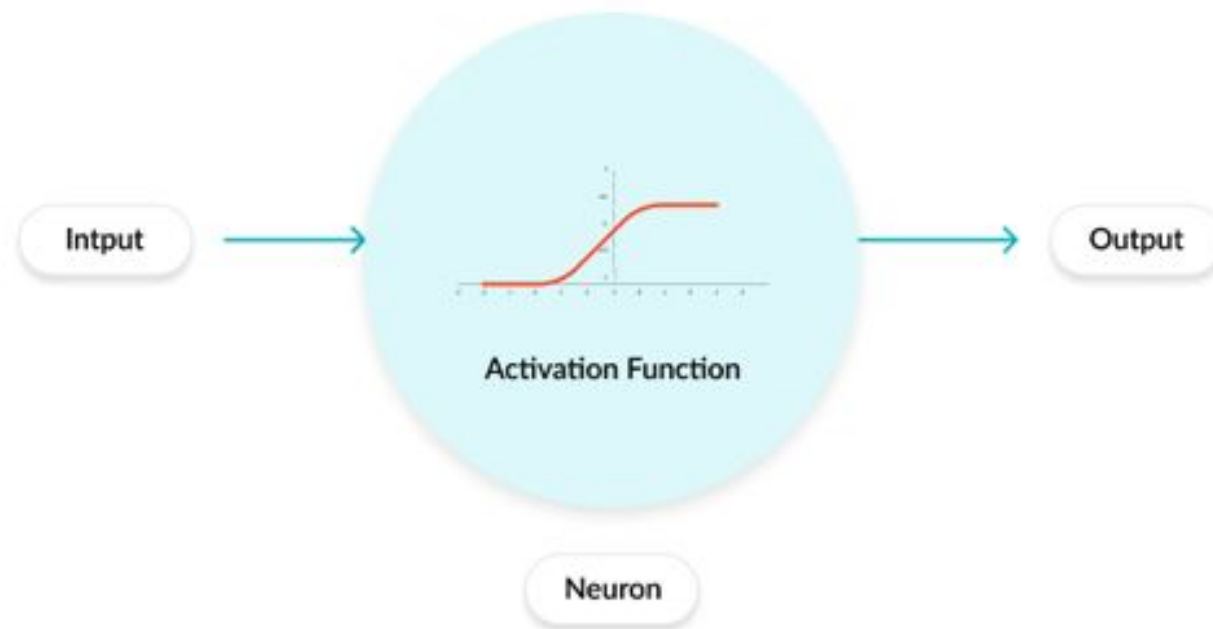
Funkcja aktywacji to matematyczne równanie, które określa wyjście neuronu, czy powinien być on aktywowany bądź nie, w oparciu o to, czy wejście każdego neuronu jest istotne dla przewidywania modelu. Funkcje aktywacji pomagają również znormalizować (czyli przekształcić) wyjście każdego neuronu do zakresu od 0 do 1 lub od -1 do 1.



Funkcja aktywacji

Dodatkowym aspektem funkcji aktywacji jest to, że muszą one być wydajne obliczeniowo, ponieważ są obliczane na tysiącach, a nawet milionach neuronów dla każdej próbki danych. Nowoczesne sieci neuronowe wykorzystują technikę zwaną propagacją wsteczną do trenowania modelu, która powoduje zwiększone obciążenie obliczeniowe funkcji aktywacji i jej funkcji pochodnej.

Funkcja aktywacji



Funkcja aktywacji to matematyczna „bramka” pomiędzy wejściem zasilającym aktualny neuron a jego wyjściem przechodzącym do następnej warstwy. Może być prosta, jak funkcja krokowa, która włącza i wyłącza wyjście neuronu, w zależności od reguły lub progu lub może to być bardziej skomplikowana transformacja.

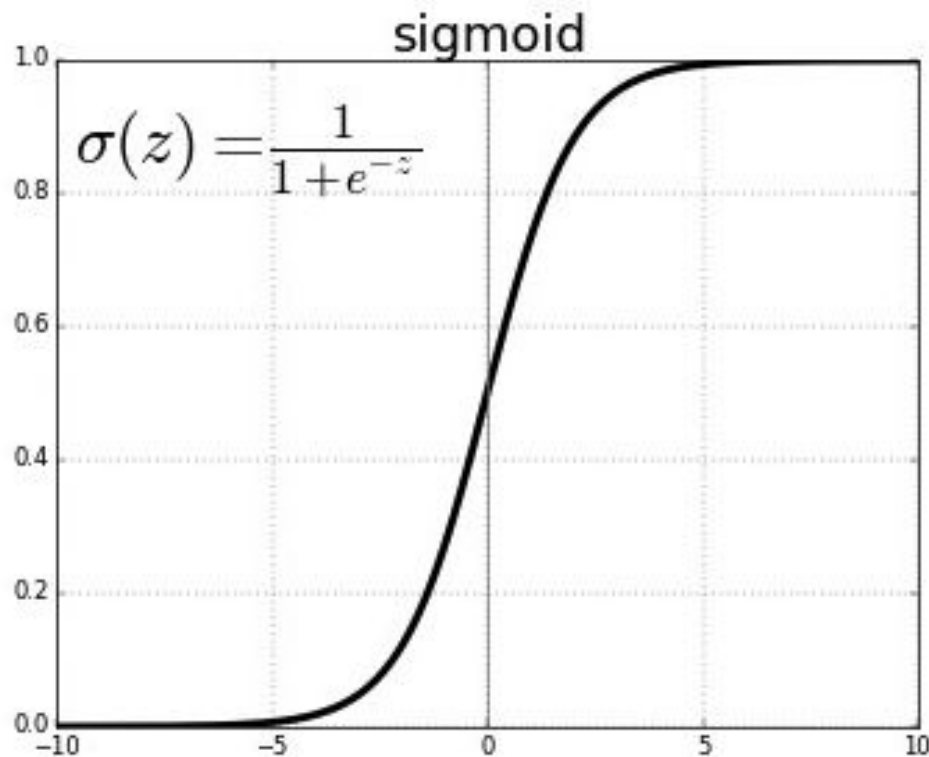


Funkcja aktywacji

Nowoczesne modele sieci neuronowych wykorzystują nieliniowe funkcje aktywacji. Umożliwiają one modelowi tworzenie złożonych mapowań między wejściami i wyjściami sieci, które są niezbędne do uczenia się i modelowania złożonych danych, takich jak obrazy, wideo, audio i zestawy danych, które są nieliniowe lub mają duże wymiary.

Funkcje nieliniowe rozwiązują problemy liniowej funkcji aktywacji: umożliwiają one propagację wsteczną, ponieważ mają funkcję pochodną, która jest powiązana z danymi wejściowymi. Pozwalają na „układanie” wielu warstw neuronów w celu stworzenia głębokiej sieci neuronowej. Aby nauczyć się złożonych zestawów danych z dużą dokładnością, potrzeba wielu ukrytych warstw neuronów.

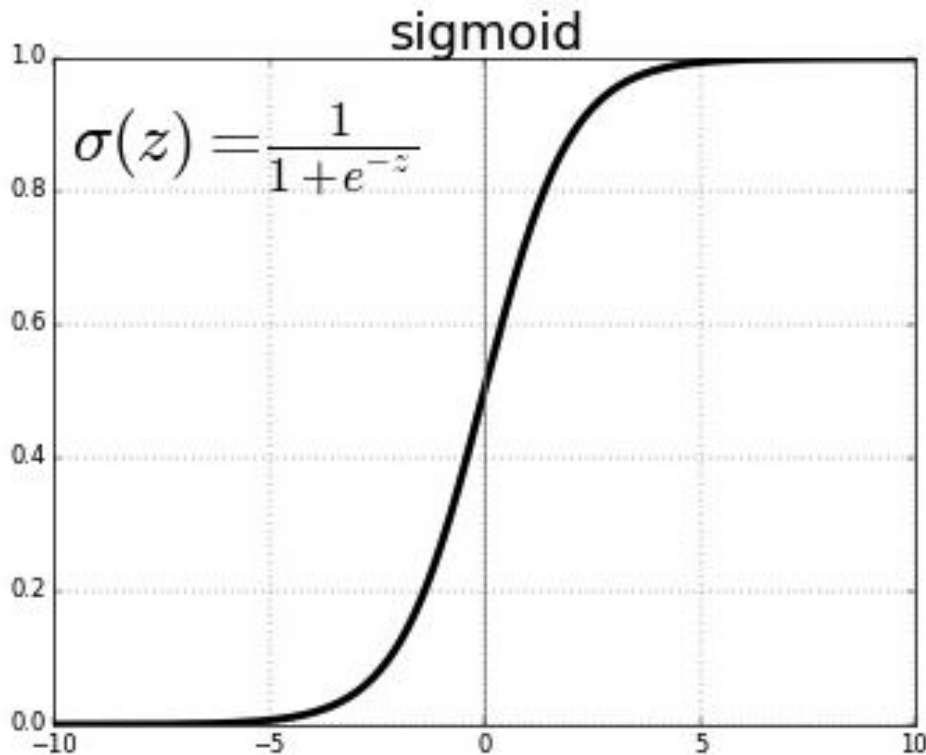
Funkcja sigmoidalna



Zalety:

- **jest gładka**, więc różniczkowalna (ma pochodną w każdym punkcie swej dziedziny) dowolnie wiele razy – zapobiega to skokom w wartościach wyjściowych,
- **wartości wyjściowe są ograniczone** od 0 do 1, normalizując wyjście każdego neuronu,
- **jasne prognozy** – dla x powyżej 5 lub poniżej -5, ma tendencję do zbliżania wartości y (prognozy) do krawędzi krzywej, bardzo blisko 1 lub 0. Umożliwia to jasne przewidywania.

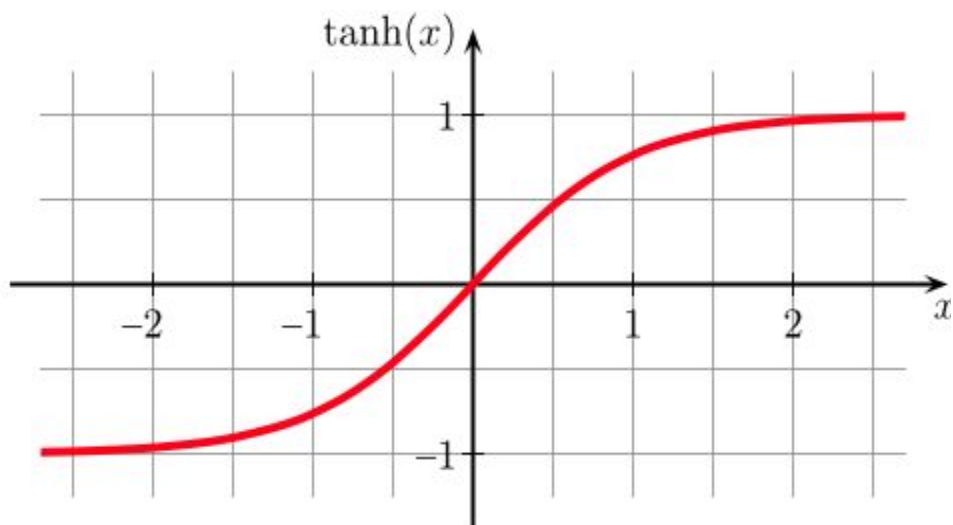
Funkcja sigmoidalna



Wady:

- **znikający gradient** - dla bardzo wysokich lub bardzo niskich wartości x prawie nie ma zmiany w przewidywaniu, powodując problem ze znikającym gradientem. Może to spowodować, że sieć będzie miała problem, by się nauczyć,
- **wyniki nie są wyśrodkowane** na zero,
- **kosztowne** obliczeniowo.

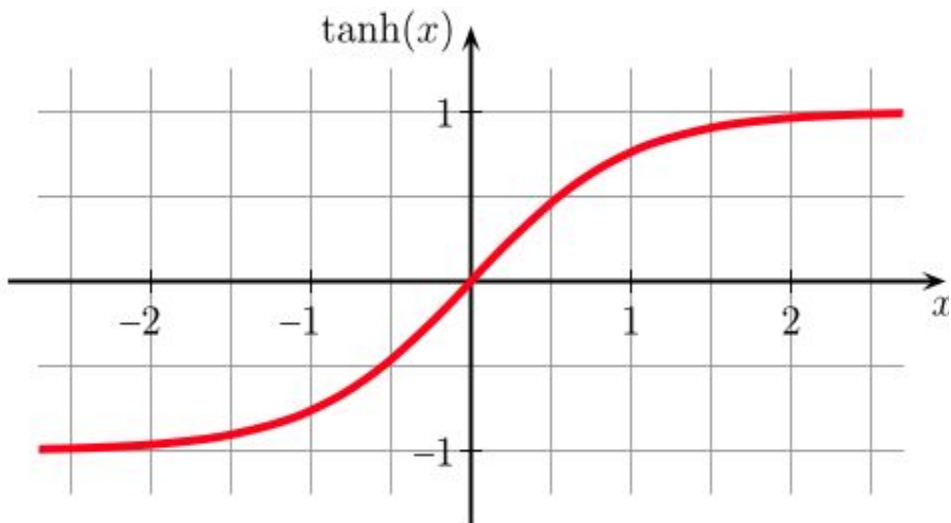
Funkcja hiperboliczna (np. tanh)



Zalety:

- **jest gładka**, więc różniczkowalna dowolnie wiele razy – zapobiega to skokom w wartościach wyjściowych,
- **wartości wyjściowe są ograniczone** od -1 do 1, normalizując wyjście każdego neuronu,
- **jasne prognozy** – dla x powyżej 2 lub poniżej -2, ma tendencję do zbliżania wartości y (prognozy) do krawędzi krzywej, bardzo blisko 1 lub -1. Umożliwia to jasne przewidywania,
- **wyniki są wyśrodkowane na zero** – ułatwia modelowanie danych wejściowych, które mają wartości silnie ujemne, neutralne i silnie dodatnie.

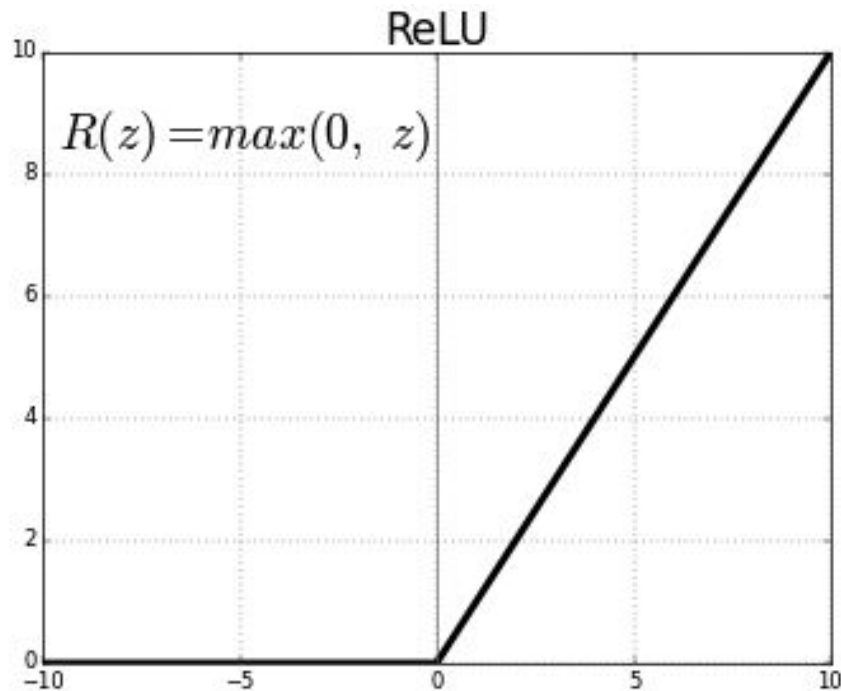
Funkcja hiperboliczna (np. tanh)



Wady:

- **znikający gradient** - dla bardzo wysokich lub bardzo niskich wartości x prawie nie ma zmiany w przewidywaniu, powodując problem ze znikającym gradientem. Może to spowodować, że sieć będzie miała problem, by się nauczyć,
- **kosztowne** obliczeniowo.

ReLU (Rectified Linear Unit)



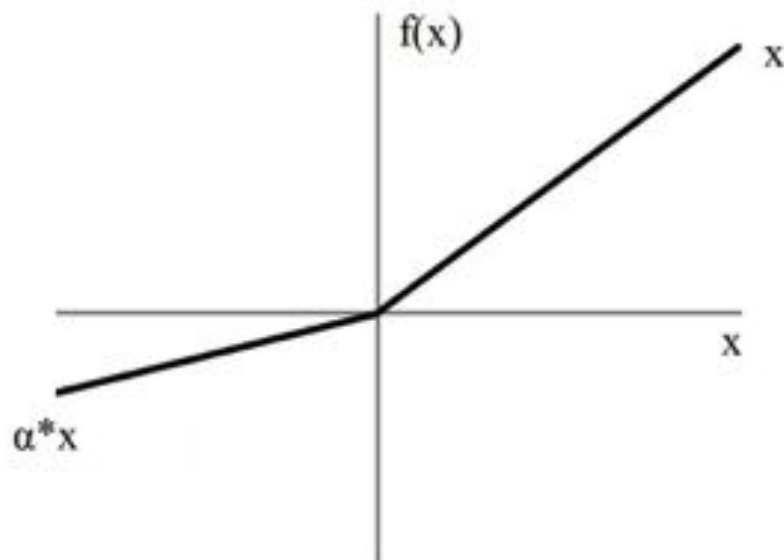
Zalety:

- **wydajna obliczeniowo** - umożliwia bardzo szybką zbieżność sieci,
- **nieliniowa** - chociaż wygląda jak funkcja liniowa, ReLU ma funkcję pochodną i umożliwia wsteczną propagację.

Wady:

- **martwe ReLU** - kiedy dane wejściowe zbliżają się do zera lub są ujemne, gradient funkcji staje się zerowy, sieć nie może wykonać propagacji wstecznej i nie może się uczyć.

Leaky ReLU



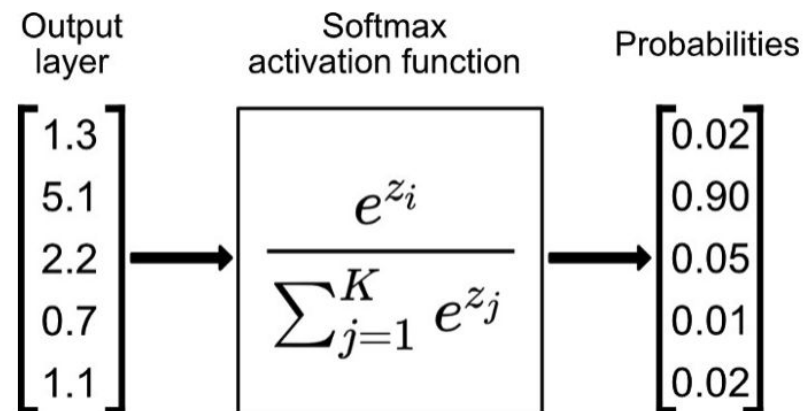
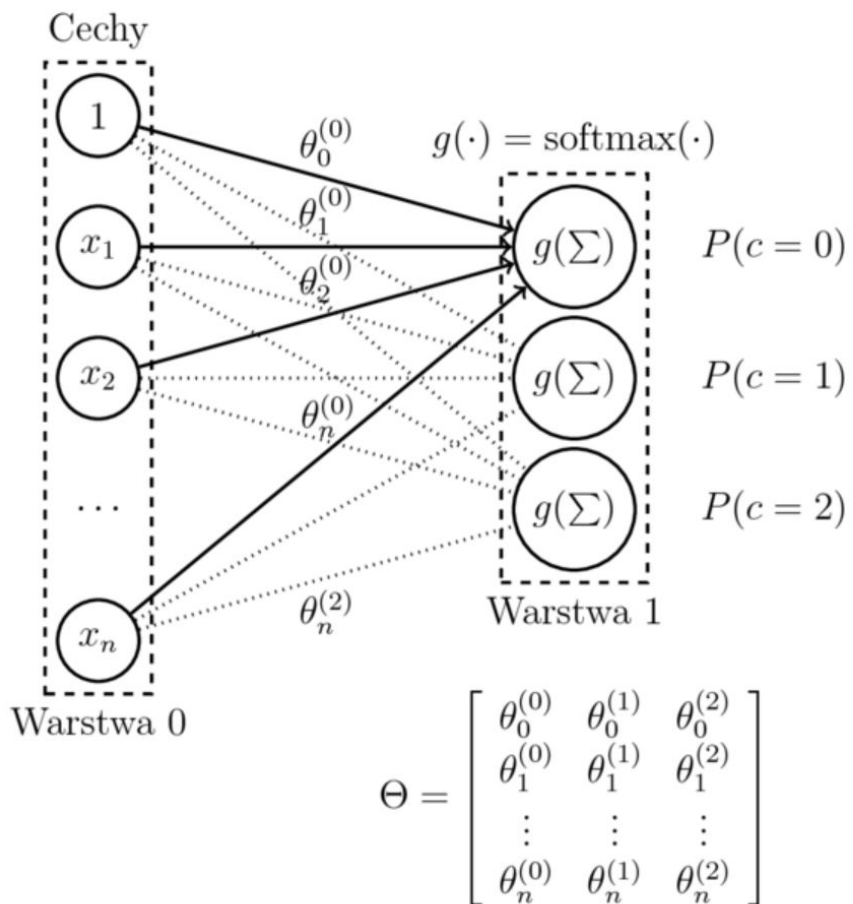
Zalety:

- **wydajna obliczeniowo** - umożliwia bardzo szybką zbieżność sieci,
- **nieliniowa,**
- **zapobiega problemowi umierania ReLU** - ta odmiana ReLU ma małe dodatnie nachylenie w obszarze ujemnym, więc umożliwia wsteczną propagację, nawet dla ujemnych wartości wejściowych.

Wady:

- **wyniki nie są spójne** - Leaky ReLU nie zapewnia spójnych prognoz dla ujemnych wartości wejściowych.

Softmax



W przeciwieństwie do pozostałych funkcji potrafi obsłużyć wiele klas – normalizuje wyniki dla każdej klasy od 0 do 1 i dzieli przez ich sumę, dając prawdopodobieństwo, że wartość wejściowa będzie w określonej klasie.

Funkcja aktywacji w scikit-learn

Wybór funkcji aktywacji wpływa na skuteczność modelu:

```
▶ for activation_function in ["identity", "logistic", "tanh", "relu"]:  
    model = MLPClassifier(activation=activation_function)  
    model.fit(X_train_sc, y_train)  
    y_pred = model.predict(X_test_sc)  
    print(f"Dla {activation_function}: {f1_score(y_test, y_pred, average='macro')}")
```

```
↳ Dla identity: 0.9709618874773142  
   Dla logistic: 1.0  
   Dla tanh: 0.9709618874773142  
   Dla relu: 0.9709618874773142
```



Warstwa wyjściowa

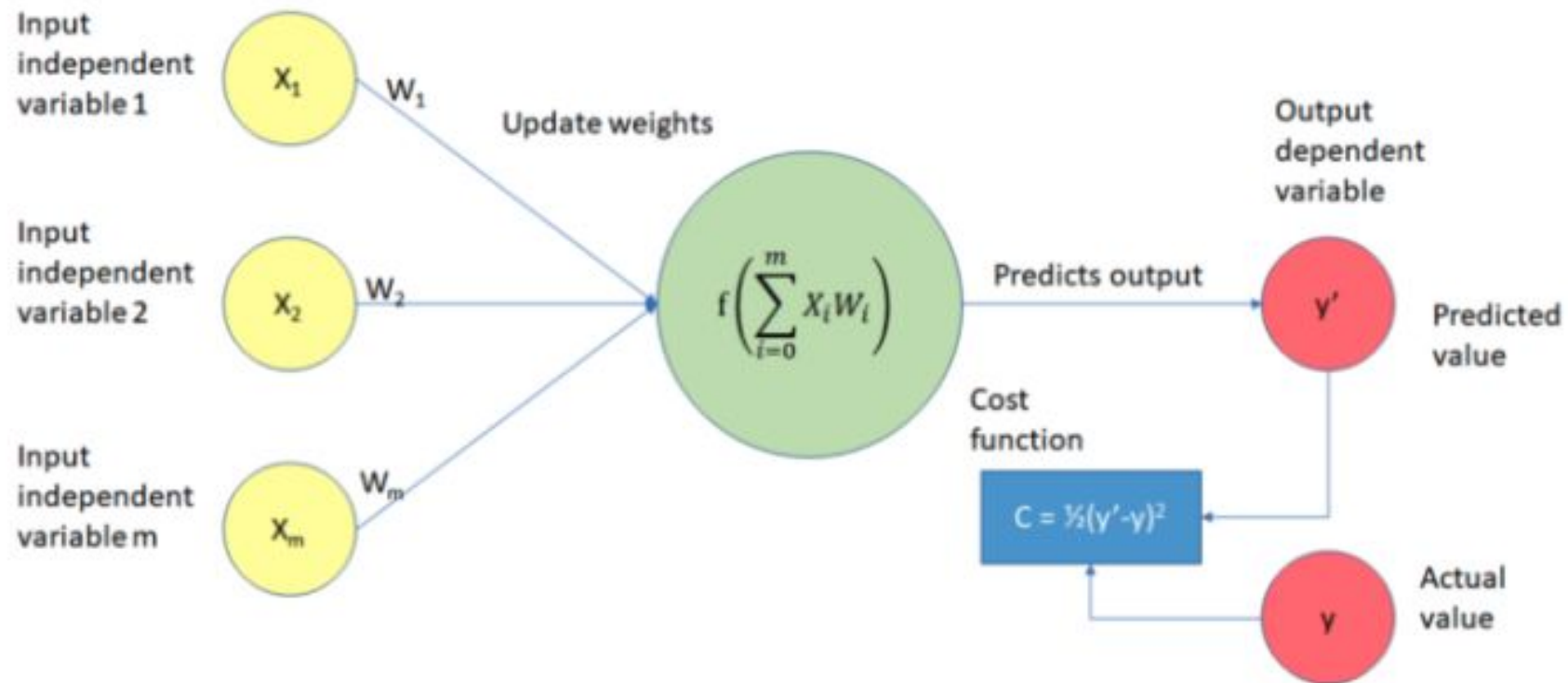


Typ funkcji aktywacji neuronów warstwy wyjściowej zależy od przeznaczenia sieci. Jeśli mierzymy się z problemem:

- regresji: wyjściem będzie pojedynczy neuron, a jego funkcją aktywacji może być np. funkcja identycznościowa $f(x) = x$,
- klasyfikacji binarnej (dwuklasowej): wyjściem może być pojedynczy neuron, a funkcją aktywacji **sigmoid** – w zależności od ustalonej wartości progowej będziemy określali, czy dane wejściowe wskazują na pierwszą klasę, czy na drugą,
- klasyfikacji wieloklasowej – wyjściem będzie warstwa złożona z n neuronów (gdzie n to liczba klas), a funkcją aktywacji – **softmax**.

Funkcja kosztu

Tak jak w przypadku pozostałych algorytmów, tak i tutaj potrzebujemy **funkcji kosztu**, która pozwoli nam określić, jak daleko nasza predykcja na wyjściu odbiega od prawidłowej (ground truth).



Funkcja kosztu

W przypadku regresji możemy w tym celu wykorzystać MAE lub MSE

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n \underbrace{|y_i - \hat{y}_i|}_{\substack{\text{predicted value} \\ \text{actual value}}}$$

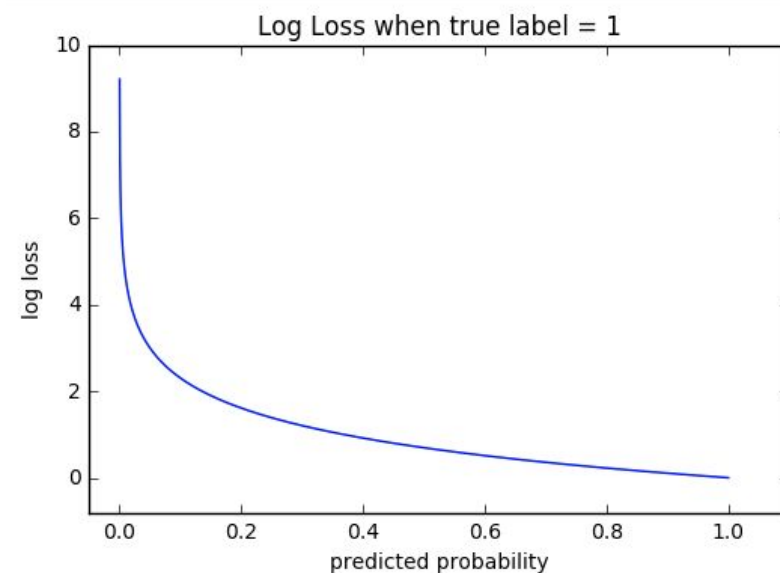
test set

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n \underbrace{(y_i - \hat{y}_i)^2}_{\substack{\text{predicted value} \\ \text{actual value}}}$$

test set

Natomiast w przypadku klasyfikacji – log-loss.

$$-\sum_{c=1}^M y_{o,c} \log(p_{o,c})$$





Wsteczna propagacja błędu



Wiemy więc jak wyznaczyć błąd dla warstwy wyjściowej, ale jak dokonać tego dla warstw ukrytych?

Problem ten przez wiele lat stawiał pod znakiem zapytania możliwość efektywnego uczenia sieci wielowarstwowych. Dopiero opracowanie metody, która pozwalała matematycznie wyznaczyć błąd popełniany przez neurony warstw ukrytych – na podstawie błędu warstwy wyjściowej – i wykorzystanie go do korekty wag neuronów tychże warstw umożliwił efektywne wykorzystanie reguł uczenia nadzorowanego do treningu sieci wielowarstwowych. Metoda ta nosi nazwę **metody wstecznej propagacji błędu** (*backpropagation*) i jej idea jest powszechnie stosowana do uczenia sieci wielowarstwowych.



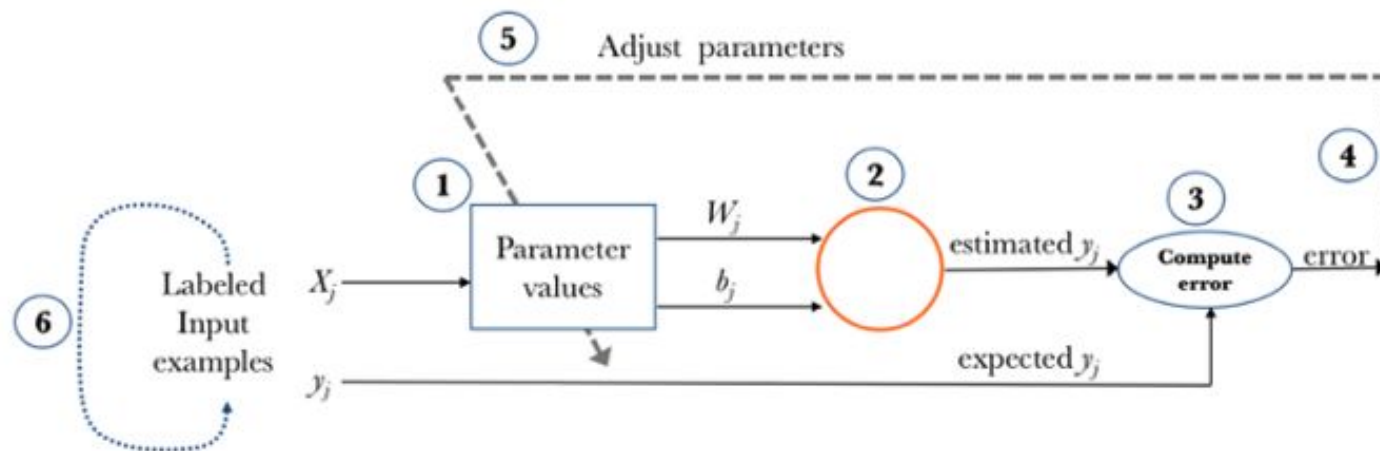
Wsteczna propagacja błędu

Algorytm wstecznej propagacji błędu zdecydowanie dominuje wśród metod uczenia jednokierunkowych sieci wielowarstwowych. Nazwa metody oddaje zasadę jej działania, która polega na “przenoszeniu” błędu, jaki popełniła sieć, w kierunku od warstwy wyjściowej do warstwy wejściowej (a więc wstecz w stosunku do kierunku przepływu informacji).

Wsteczna propagacja błędu

Cykl uczenia metodą wstecznej propagacji błędu (backpropagation) składa się z następujących etapów:

- Wyznaczenie odpowiedzi neuronów warstwy wyjściowej oraz warstw ukrytych na zadany sygnał wejściowy.
- Wyznaczenie błędu popełnianego przez neurony znajdujące się w warstwie wyjściowej i przesłanie go w kierunku warstwy wejściowej.
- Adaptacja wag.



Wsteczna propagacja błędu

Algorytm przebiega następująco:

- Wygeneruj losowo wektory wag
- Podaj dane wejściowe na warstwę wejściową sieci
- Wyznacz odpowiedzi wszystkich neuronów wyjściowych sieci

$$y_k^{wyj} = f\left(\sum_{j=1}^l w_{kj}^{wyj} y_j^{wyj-1}\right)$$

- Oblicz błędy wszystkich neuronów warstwy wyjściowej:

$$\delta_k^{wyj} = z_k - y_k^{wyj}$$

Wsteczna propagacja błędu

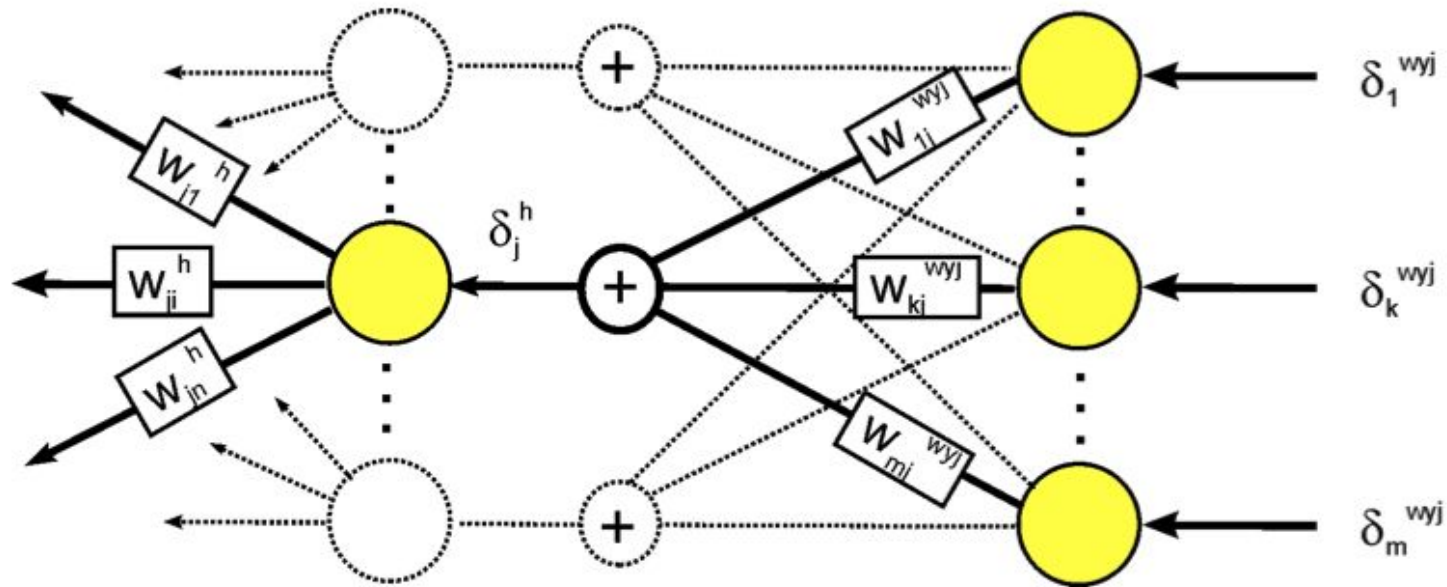
- Oblicz błędy w warstwach ukrytych (pamiętając, że aby wyznaczyć błąd w warstwie $h - 1$, konieczna jest znajomość błędu w warstwie po niej następującej - h),

$$\delta_j^{h-1} = \frac{df(u_j^{h-1})}{du_j^{h-1}} \sum_{k=1}^l \delta_k^h w_{kj}^h$$

- Zmodyfikuj wagi wg zależności:

- Jeżeli wartość funkcji kosztu j $w_{ji}^{h-1} = w_{ji}^{h-1} + \eta \delta_j^{h-1} y_i^{h-1}$ proces.

Wsteczna propagacja błędu



W praktyce metoda wstecznej propagacji błędu okazuje się bardzo skuteczna, niestety, charakteryzuje się ona długim czasem uczenia. Przebieg procesu uczenia sieci backpropagation silnie zależy od wielkości **współczynnika uczenia** η (*learning rate*), zbyt duża jego wartość prowadzi często do rozbieżności tego procesu, a zbyt mała – bardzo go wydłuża. Niestety, nie ma reguł które potrafiłyby określić precyzyjnie jego wartość, jest to jeden z hiperparametrów sieci.



Wsteczna propagacja błędu

Przykład dostępny pod adresem:

<https://mattmazor.com/2015/03/17/a-step-by-step-backpropagation-example/>



Hiperparametry sieci



Jak już wiemy, hiperparametrami sieci, których wielkość trzeba ustalić są: **współczynnik uczenia**, **liczba warstw ukrytych**, **liczba neuronów** w każdej warstwie. Do nich zaliczyć możemy jeszcze **liczbę epok** oraz **rozmiar batchu**.

Co oznaczają te dwa pojęcia?



Hiperparametry sieci



Często zdarza się, że danych mamy tak dużo, że nie jesteśmy w stanie przekazać ich do sieci wszystkich naraz, bo byłoby to zbyt pamięciożerne, a w konsekwencji wiązałoby się z dużym obciążeniem dla komputera. By sobie z tym poradzić, dzielimy dane na mniejsze partie i podajemy do sieci jedna za drugą, aktualizując wagi sieci po każdym kroku, by dopasować model do przekazanych danych.

Każda taka część danych nosi nazwę **batch**.



Hiperparametry sieci



Epoką nazywamy przejście w przód i w tył wszystkich dostępnych danych przez sieć. To znaczy, że w trakcie jednej epoki model “widzi” wszystkie możliwe przykłady ze zbioru.

Dlaczego potrzebujemy więcej niż jednej epoki? Pamiętajmy, że proces uczenia sieci jest iteracyjny i im więcej epok, tym więcej razy dokonujemy aktualizacji wag, więc model lepiej uczy się wzorców w danych. Niestety, nie ma jednoznacznej odpowiedzi na to, ile tych epok być powinno. Zależy to od tego, jak bardzo zadanie jest skomplikowane, czy też od stopnia zróżnicowania danych.




Hiperparametry sieci



Na przykładzie: mamy zbiór danych z 2000 przykładami, określiliśmy rozmiar batchu na 50, a liczbę epok – na 100.

W efekcie zbiór danych zostanie podzielony na 40 batchy, w każdym z nich znajdzie się 50 przykładów. Wagi modelu zostaną zaktualizowane po każdym batchu, a to oznacza, że w każdej epoce będziemy procesowali 40 batchy, więc tyle razy dokonamy aktualizacji wag modelu.

W ciągu 100 epok, model stukrotnie “zobaczy” cały zbiór danych, a trakcie całego tego procesu aktualizacja wag zostanie dokonana 4000 razy.



Rozmiar batcha w scikit-learn

Wybór rozmiaru batcha wpływa na skuteczność modelu:

```
▶ for batch in [10, 20, 50, 100]:  
    model = MLPClassifier(batch_size=batch)  
    model.fit(X_train_sc, y_train)  
    y_pred = model.predict(X_test_sc)  
    print(f"Dla {batch}: {f1_score(y_test, y_pred, average='macro'}}")
```

```
↳ Dla 10: 0.9709618874773142  
   Dla 20: 0.9709618874773142  
   Dla 50: 0.9453132832080202  
   Dla 100: 0.9709618874773142
```

Liczba epok w scikit-learn

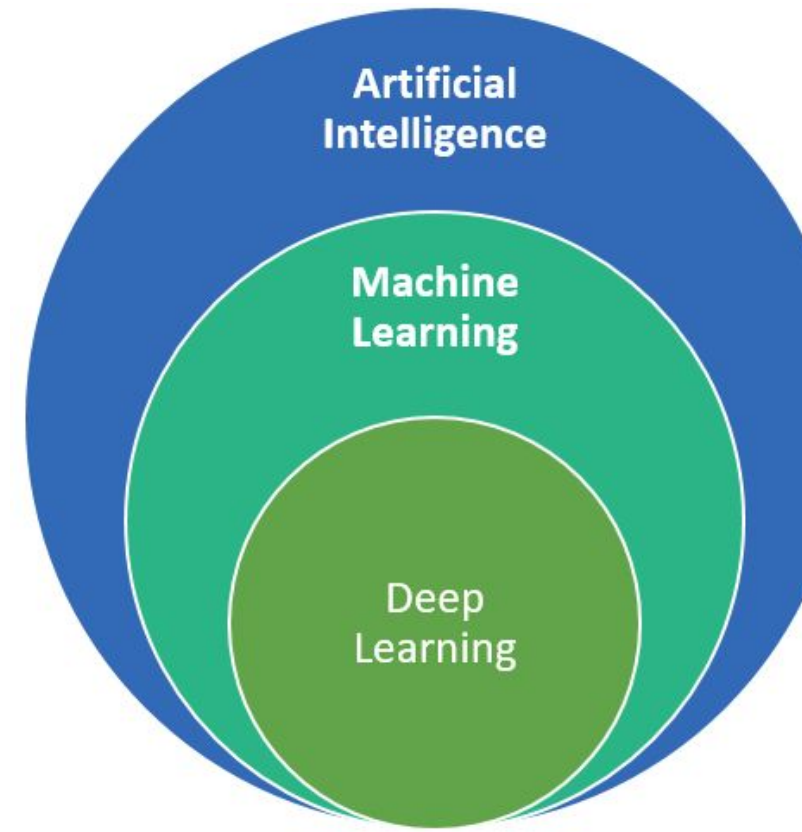
Wybór maksymalnej liczby iteracji wpływa na skuteczność modelu:

```
▶ for iter in [10, 20, 50, 200]:  
    model = MLPClassifier(max_iter=iter)  
    model.fit(X_train_sc, y_train)  
    y_pred = model.predict(X_test_sc)  
    print(f"Dla {iter}: {f1_score(y_test, y_pred, average='macro')}")
```

```
Dla 10: 0.7746913580246914  
Dla 20: 0.8896296296296295  
Dla 50: 0.9743209876543211  
Dla 200: 0.9709618874773142
```

Uczenie głębokie

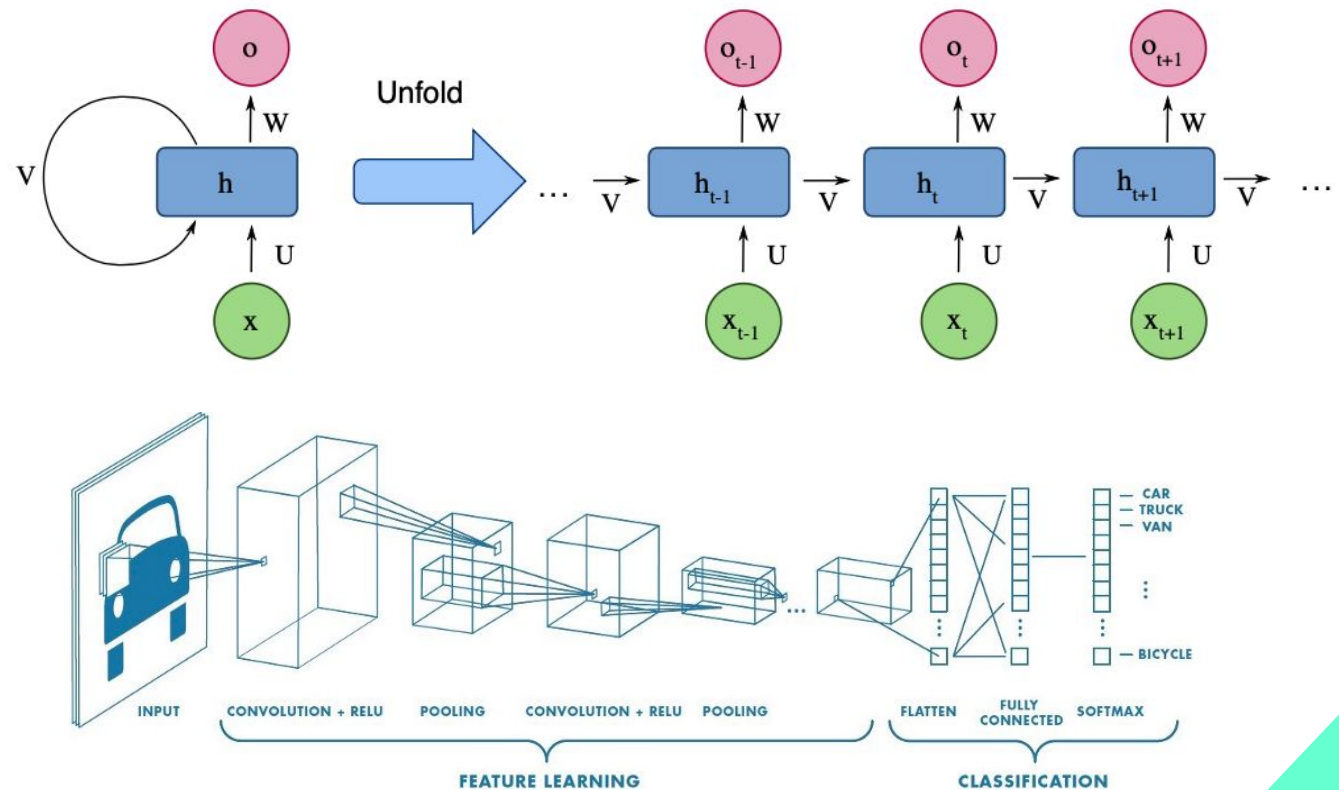
Uczenie głębokie jest podkategorią uczenia maszynowego i zakłada budowanie i trenowanie sieci neuronowych o dużej liczbie warstw lub skomplikowanej architekturze, których można użyć do zadań z zakresu **widzenia komputerowego** (ang. *computer vision*) lub **przetwarzania języka naturalnego** (ang. *natural language processing*).



Uczenie głębokie

Najbardziej upowszechniane architektury sieci stosowanych w uczeniu głębokim:

- sieci rekurencyjne (*recurrent neural networks*),
- sieci splotowe (*convolutional neural networks*).





Implementacja w scikit-learn



By korzystać z dobrodziejstw modeli opartych o sieci neuronowe, możemy skorzystać z dobrze nam znanego modułu scikit-learn.

Dla problemów regresji będziemy używali **MLPRegressor**, dla klasyfikacji - **MLPClassifier**, oba z pakietu **sklearn.neural_network**

MLPRegressor

```
reg_model = MLPRegressor(  
    hidden_layer_sizes=(16, 8, 4),  
    activation='relu',  
    solver='adam',  
    random_state=42,  
    max_iter=1000,  
    batch_size=64)
```

```
reg_model.fit(X_train_sc, y_train)
```

```
MLPRegressor(batch_size=64, hidden_layer_sizes=(16, 8, 4), max_iter=1000,  
             random_state=42)
```

- **hidden_layer_sizes** – ten parametr umożliwia nam zdefiniowanie liczby warstw ukrytych i liczbę neuronów w każdej z nich – w tym przypadku mamy 3 warstwy ukryte, w pierwszej znajduje się 16 neuronów, w drugiej – 8, a w trzeciej – 4,
- **activation** – funkcja aktywacji (ta sama dla każdej warstwy ukrytej),
- **solver** – algorytm używany do optymalizacji wag,
- **max_iter** – liczba epok,
- **batch_size** – rozmiar batchu.

MLPClassifier

```
cls_model = MLPClassifier(  
    hidden_layer_sizes=(8, 4),  
    activation='relu',  
    solver='adam',  
    random_state=42,  
    max_iter=1000,  
    batch_size=32)  
  
cls_model.fit(X_train_sc, y_train)  
  
MLPClassifier(batch_size=32, hidden_layer_sizes=(8, 4), max_iter=1000,  
              random_state=42)
```

Tym razem sieć będzie złożona z 2 warstw ukrytych, w pierwszej z nich znajduje się 8 neuronów, w drugiej – 4.



Polecane materiały związane z tematyką bloku



- [Jako Rzecz Maszyna - #4 O co chodzi z tą czarną skrzynką? - sztuczne sieci neuronowe i tajemnice ludzkiego umysłu](#)
- [Neural Network In 5 Minutes | What Is A Neural Network?](#)
- [Neural Network Backpropagation Basics](#)