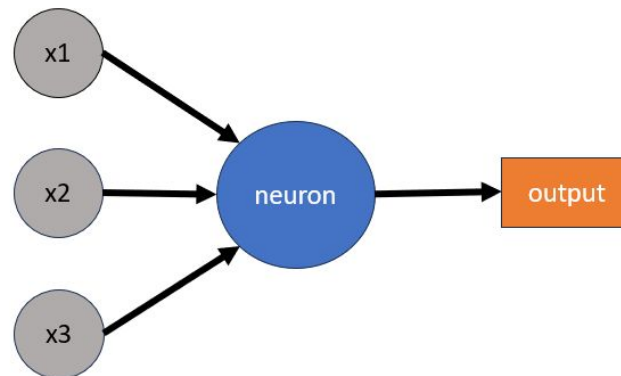
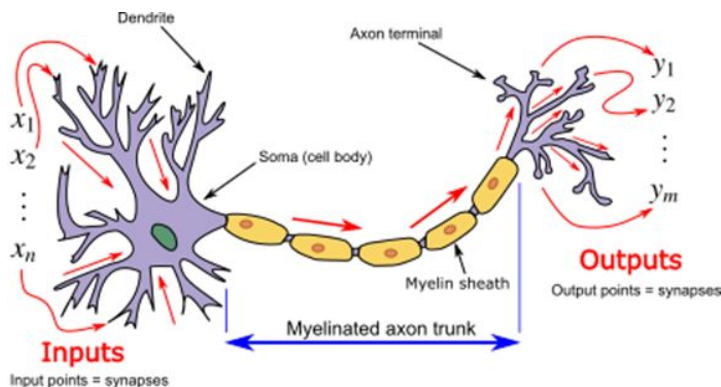


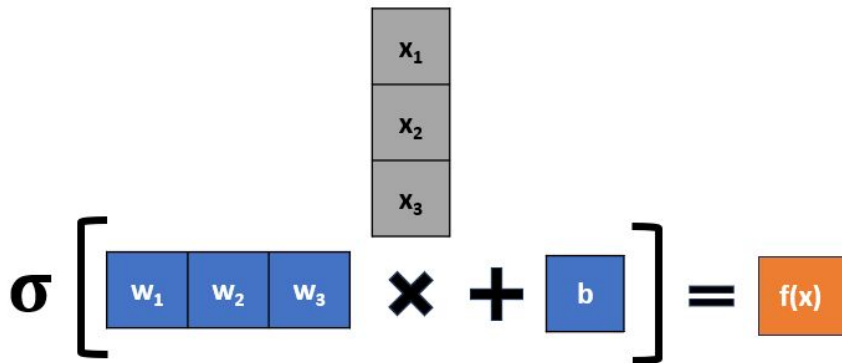
Neuron

- neuron w sieci neuronowej cechuje odległe podobieństwo do neuronu w mózgu
- mózg operuje na impulsach elektrycznych, sieć zaś na liczbach rzeczywistych
- neuron w sieci przyjmuje n wartości i zwraca jedną
- funkcja neuronu w sieci jest podobna do klasyfikatora binarnego; jego zadaniem jest podjęcie pewnej (bardzo drobnej) decyzji



Neuron: implementacja

- neuron jest implementowany jako przekształcenie liniowe otoczone pewną prostą funkcją nieliniową (tzw. funkcją aktywacyjną)
- $f: \mathbb{R}^n \rightarrow \mathbb{R}, f(x) = \sigma(wx + b)$



$f: \mathbb{R}^n \rightarrow \mathbb{R}$

neuron

$x \in \mathbb{R}^n$

wejście

$f(x) \in \mathbb{R}$

wyjście

$w \in \mathbb{R}^n$

wagi (*weights*)

$b \in \mathbb{R}$

bias

$\sigma: \mathbb{R} \rightarrow \mathbb{R}$

funkcja
aktywacyjna

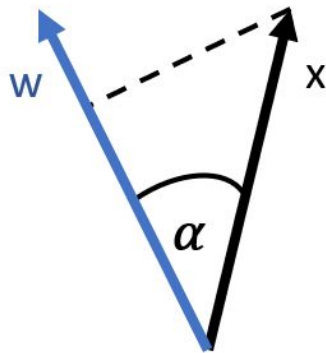
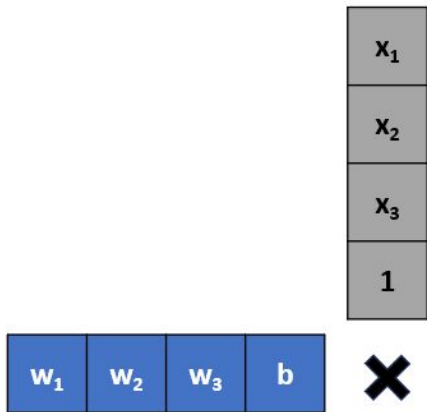
Neuron: wagi i bias

- neuron ma $n+1$ parametrów, które będą zmieniać się w trakcie uczenia: jedną wagę w_i przypadającą na każde z n wejść oraz bias b
- do celów teoretycznych bias można interpretować jako wagę przy dodatkowym wejściu zawsze równym jeden

$$\sigma \left[\begin{bmatrix} w_1 & w_2 & w_3 & b \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix} \right] = f(x)$$

Neuron: dlaczego przekształcenie liniowe?

- iloczyn skalarny wx szybko się liczy i bardzo dobrze się zrównolegla
- mnożenie wag w przez wejście x można interpretować jako wykrywanie, jak duży komponent w kierunku w jest obecny w x . Pojedynczy neuron jest więc detektorem pewnego konkretnego kierunku w przestrzeni \mathbb{R}^n



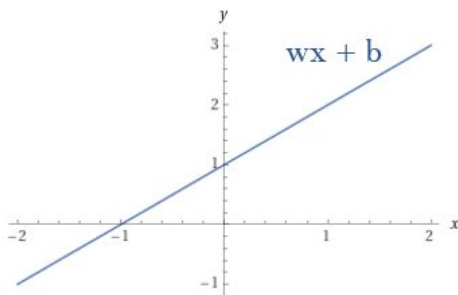
$$wx = |w| |x| \cos(\alpha)$$

Neuron: funkcje aktywacyjne

- funkcja aktywacyjna jest zwykle pozbawiona parametrów uczących się
- $\sigma(wx + b)$ ma przypominać decyzję, np. funkcję schodkową i być różniczkowalne
- dawniej najbardziej popularną funkcją aktywacyjną był sigmoid. Współcześnie używane jest głównie ReLU i jego drobne modyfikacje

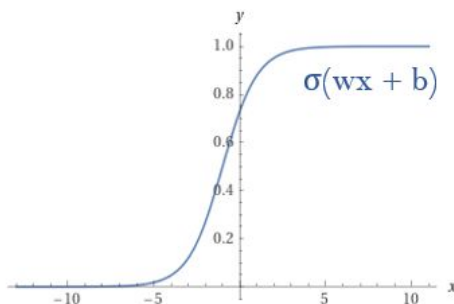
tożsamość

$$\sigma(z) = z$$



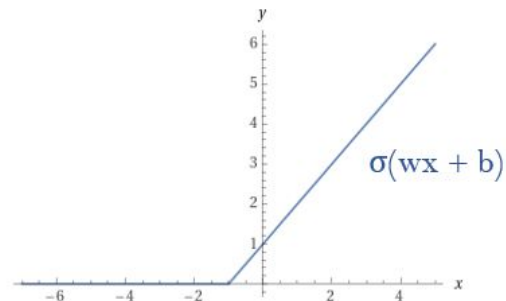
sigmoid

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



ReLU

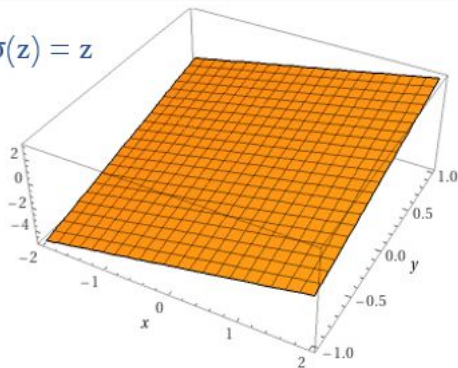
$$\sigma(z) = \max(0, z)$$



Neuron: funkcje aktywacyjne

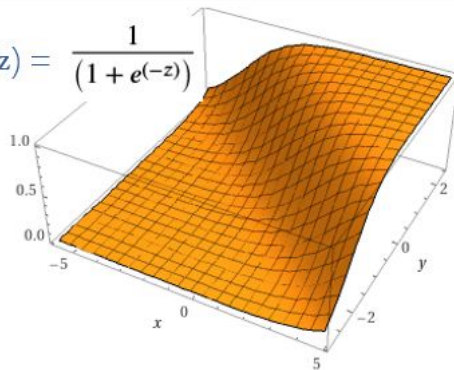
3D plot

$$\sigma(z) = z$$



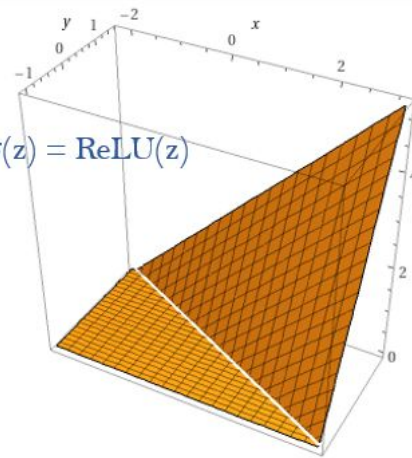
3D plot

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

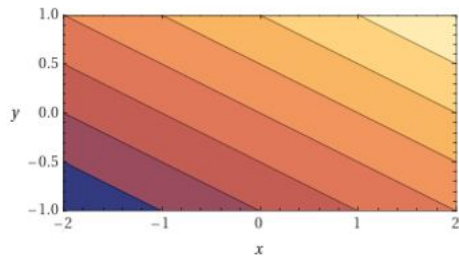


3D plot

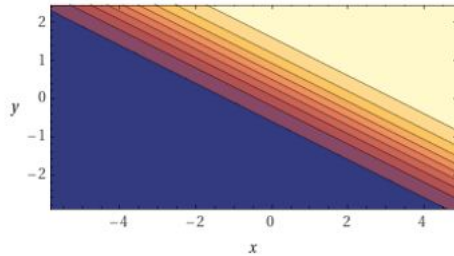
$$\sigma(z) = \text{ReLU}(z)$$



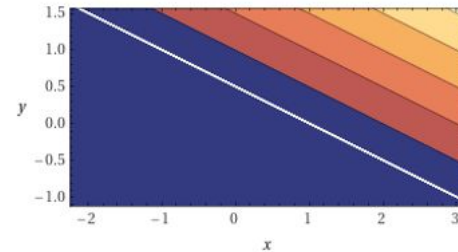
Contour plot



Contour plot

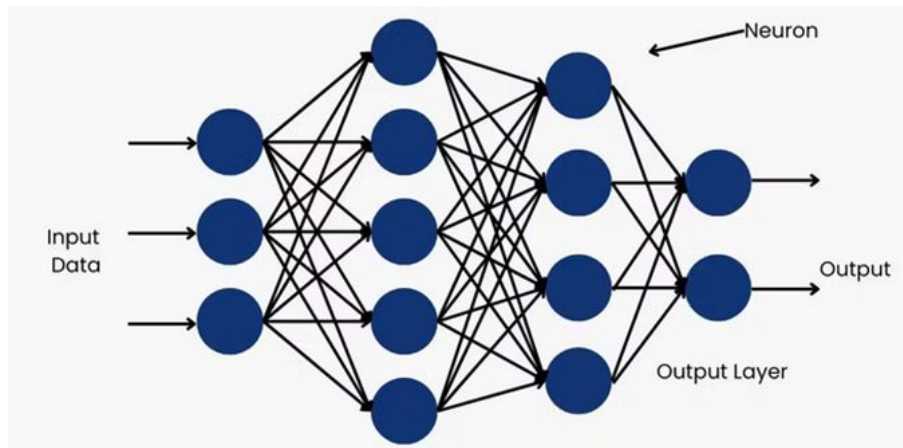


Contour plot



Multilayer perceptron

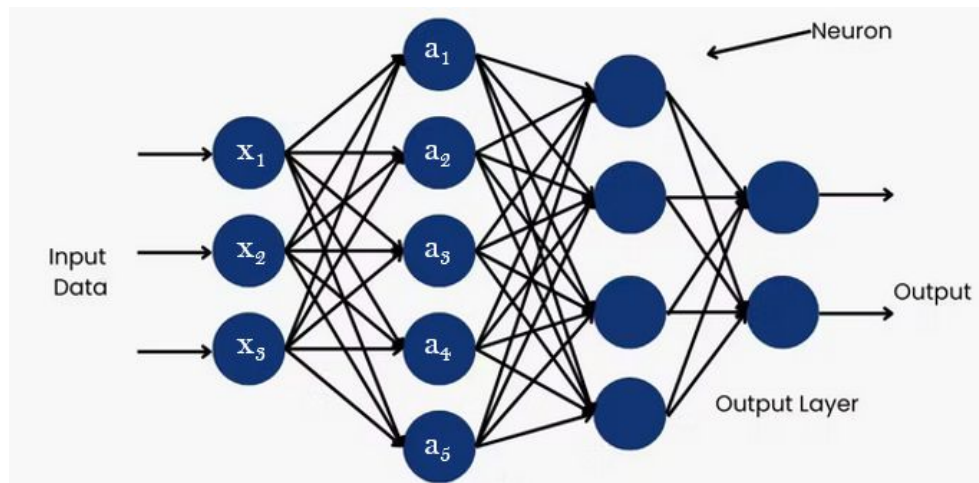
- najprostsza sieć neuronowa
- neurony zostają połączone w warstwy
- wyjście poprzedniej warstwy staje się wejściem następnej



- sieć na obrazku ma dwie “ukryte” warstwy, które razem z warstwą “wyjściową” zawierają 54 parametry
- używane w praktyce sieci zawierają miliony, a nawet miliardy parametrów
- głębsze sieci są w stanie wykrywać bardziej złożone cechy
- multilayer perceptron o jedynie dwóch warstwach ukrytych jest w stanie przybliżyć każdą funkcję z dowolną dokładnością, ale w praktyce kiepsko się uczy

Obliczanie wielu wyjść naraz

- równania dla wszystkich neuronów pojedynczej warstwy można zapisać przy użyciu mnożenia macierzy



$$a_1 = \sigma(w_{1,1}x_1 + w_{2,1}x_2 + w_{3,1}x_3 + b_1)$$

$$a_2 = \sigma(w_{1,2}x_1 + w_{2,2}x_2 + w_{3,2}x_3 + b_2)$$

$$a_3 = \sigma(w_{1,3}x_1 + w_{2,3}x_2 + w_{3,3}x_3 + b_3)$$

$$a_4 = \sigma(w_{1,4}x_1 + w_{2,4}x_2 + w_{3,4}x_3 + b_4)$$

$$a_5 = \sigma(w_{1,5}x_1 + w_{2,5}x_2 + w_{3,5}x_3 + b_5)$$



$$a = \sigma(Wx + b)$$

The matrix equation represents the calculation of the hidden layer outputs a from the input data x and weights W and bias b . The input data x is a column vector with elements x_1 , x_2 , and x_3 . The weight matrix W is a 5x3 matrix with elements w_{ij} . The bias vector b is a column vector with elements b_1 , b_2 , b_3 , b_4 , and b_5 . The output vector a is a column vector with elements a_1 , a_2 , a_3 , a_4 , and a_5 . The equation is written as $\sigma \left[Wx + b \right] = a$.

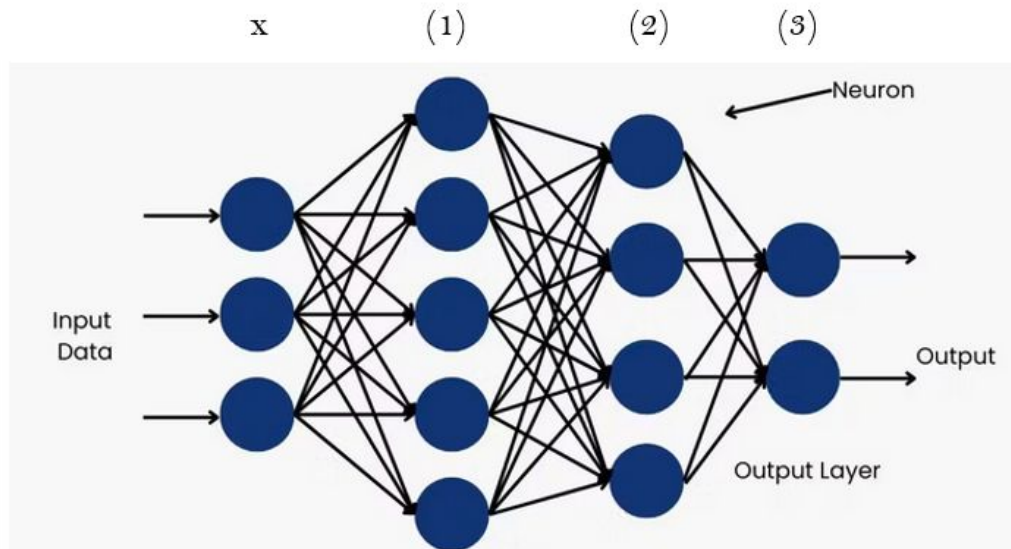
Obliczanie wielu wejść naraz

- równania dla wielu wejść jednej warstwy również można skrócić przy użyciu mnożenia macierzy
- wydajność mnożenia macierzy na GPU to główna przyczyna popularności iloczynu skalarnego jako implementacji neuronu

$$\sigma \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{bmatrix} \times \begin{bmatrix} x_{11} & x_{21} & & & & & \\ x_{12} & x_{22} & & & & & \\ x_{13} & x_{23} & & & & & \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} = \begin{bmatrix} a_{11} & & & & & & \\ a_{12} & & & & & & \\ a_{13} & & & & & & \\ a_{14} & & & & & & \\ a_{15} & & & & & & \end{bmatrix}$$

Sieć neuronowa to ciąg kilku prostych operacji

$$x \rightarrow \sigma(W^{(1)} \cdot +b^{(1)}) \rightarrow \sigma(W^{(2)} \cdot +b^{(2)}) \rightarrow W^{(3)} \cdot +b^{(3)}$$



```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
class SmolMultilayerPerceptron(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(3, 5)
        self.fc2 = nn.Linear(5, 4)
        self.fc3 = nn.Linear(4, 2)

    def forward(self, x):
        x = F.sigmoid(self.fc1(x))
        x = F.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return x
```

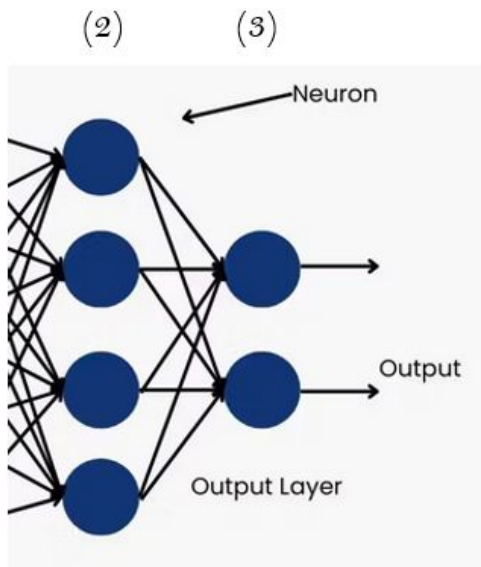
```
model = SmolMultilayerPerceptron()
x = torch.FloatTensor([0.1, 0.2, -0.3])
print(model(x).detach().numpy())
```

✓ 3.1s

[-0.12244508 -0.19386089]

Wyjście sieci neuronowej

$$\rightarrow \sigma(W^{(2)} \cdot +b^{(2)}) \rightarrow W^{(3)} \cdot +b^{(3)}$$

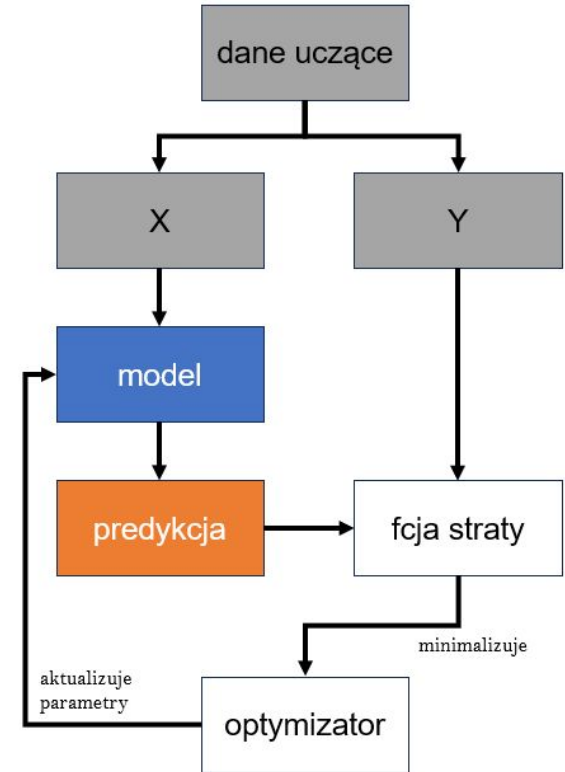


- ostatnia warstwa zwykle nie zawiera funkcji aktywacyjnej i zwraca wartości od $-\infty$ do ∞
- jeśli sieć jest wykorzystywana do regresji, liczba neuronów wyjściowych powinna być taka sama jak liczba przewidywanych wartości
- w klasyfikacji binarnej jeden neuron wystarcza (dodatnia wartość wskazuje na klasę pozytywną)
- w klasyfikacji niebinarnej liczba neuronów odpowiada liczbie klas, a do otrzymania prawdopodobieństw klas używana jest funkcja softmax

$$\frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$$

Uczenie sieci neuronowej

- sieć operuje na liczbach: wszystkie dane muszą zostać przekształcone na liczby
- zbiór uczący stanowi zapis pewnych przykładów zależności pomiędzy wyjściem a wejściem, których ogólną postać ma opanować sieć
- funkcja straty mierzy różnicę pomiędzy wartością zwróconą a wartością oczekiwaną
- procedura optymalizacyjna szuka (zwykle iteracyjnie) parametrów, które pozwalają zminimalizować funkcję straty



Uczenie sieci neuronowej: funkcja straty

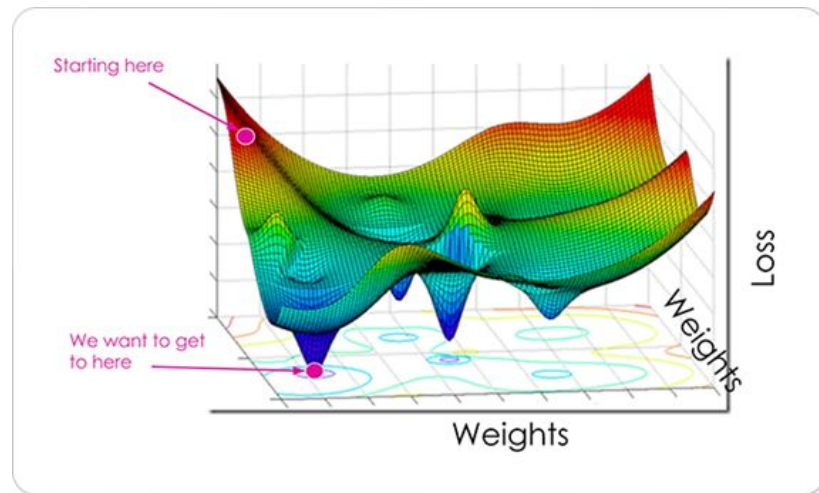
- funkcja straty wyraża różnicę pomiędzy predykcją modelu a spodziewanym wynikiem, tworząc tym samym cel optymalizacji
- różne rodzaje funkcji straty są używane do różnych problemów

regresja		
<u>Mean Absolute Error (MAE)</u>	<u>Mean Squared Error (MSE)</u>	<u>Root Mean Squared Error (RMSE)</u>
$\frac{1}{n} \sum Y - \hat{Y} $	$\frac{1}{n} \sum (Y - \hat{Y})^2$	$\sqrt{\frac{1}{n} \sum (Y - \hat{Y})^2}$

<u>klasyfikacja</u>
Entropia krzyżowa (<u>cross-entropy</u>) (in. <u>negative log-likelihood</u> lub <u>log loss</u>)
$H(y, \hat{y}) = - \sum_{c \in \text{Classes}} y_c \log \hat{y}_c$ <p>gdzie \hat{y} to prawdopodobieństwa klas</p>

Uczenie sieci to problem optymalizacyjny

- celem uczenia jest znalezienie parametrów w^* , które minimalizują funkcję straty L
- problemy: w jest wielowymiarowe, funkcja straty L jest niewypukła, nie znamy analitycznego wzoru na minimum globalne
- nawet prosty *grid search* (sprawdzenie k wartości każdego parametru) prowadzi do zbyt wielu kombinacji
- rozwiązanie: zaczynamy z losowego punktu w_0 i iteracyjnie dążymy do niższej funkcji straty



Optymalizacja: skąd wiemy, dokąd iść?

- w kroku i znajdujemy się w punkcie w_i
- potrafimy policzyć pochodne funkcji straty w punkcie w_i . Wartości tych pochodnych pozwalają nam przybliżyć
 - ... funkcję straty w sąsiedztwie w_i za pomocą rozwinięcia Taylora
 - wykonujemy krok s_{i+1} , który zmniejsza przybliżoną funkcję straty
 - metody “drugiego rzędu” używające drugiej pochodnej (Hesjana) były popularne w 2010
 - obecnie dużo bardziej popularne jest używanie wyłącznie gradientu

$$\mathcal{L}(w_i + s) = \sum_{n=0}^{\infty} \frac{\mathcal{L}^{(n)}(w_i)}{n!} s^n$$

$$\mathcal{L}(w_i + s) \approx \mathcal{L}(w_i) + \nabla \mathcal{L}(w_i) s$$

$$\mathcal{L}(w_i + s) \approx \mathcal{L}(w_i) + \nabla \mathcal{L}(w_i) s + \nabla^2 \mathcal{L}(w_i) \frac{s^2}{2}$$

Optymalizacja: obliczanie gradientu

- sieć jest ciągiem prostych operacji. Możemy obliczyć ich pochodne...

$$x \rightarrow \sigma(W^{(1)} \cdot + b^{(1)}) \rightarrow \sigma(W^{(2)} \cdot + b^{(2)}) \rightarrow W^{(3)} \cdot + b^{(3)} \rightarrow \frac{1}{n} \sum (Y - \cdot)^2$$

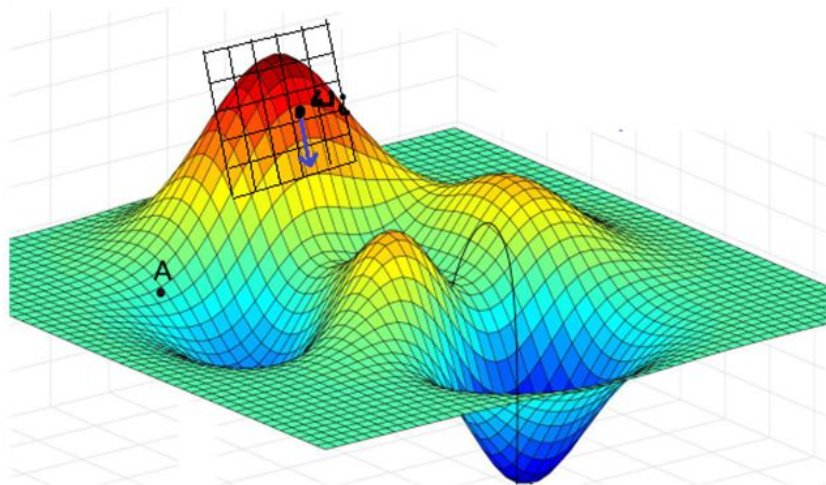
$$\frac{\partial}{\partial \cdot} \quad W^{(1)} \sigma'(W^{(1)} \cdot + b^{(1)}) \rightarrow W^{(2)} \sigma'(W^{(2)} \cdot + b^{(2)}) \rightarrow W^{(3)} \rightarrow -\frac{1}{n} \sum 2 \cdot$$

- ... a następnie użyć wzoru na pochodną funkcji złożonej i otrzymać pochodną cząstkową funkcji straty po dowolnym parametrze
- algorytm, który robi to wydajnie nazywamy *Backpropagation*
- nie będziemy omawiać go w szczegółach; dla nas liczy się to, że obliczenie pierwszej pochodnej funkcji straty L w punkcie w zajmuje mniej więcej tyle samo czasu, co obliczenie $L(w)$ i nie wymaga zbyt wiele dodatkowej pamięci
- obliczenie pochodnych wyższych rzędów zużywa dużo więcej czasu i pamięci

Optymalizacja: Gradient Descent

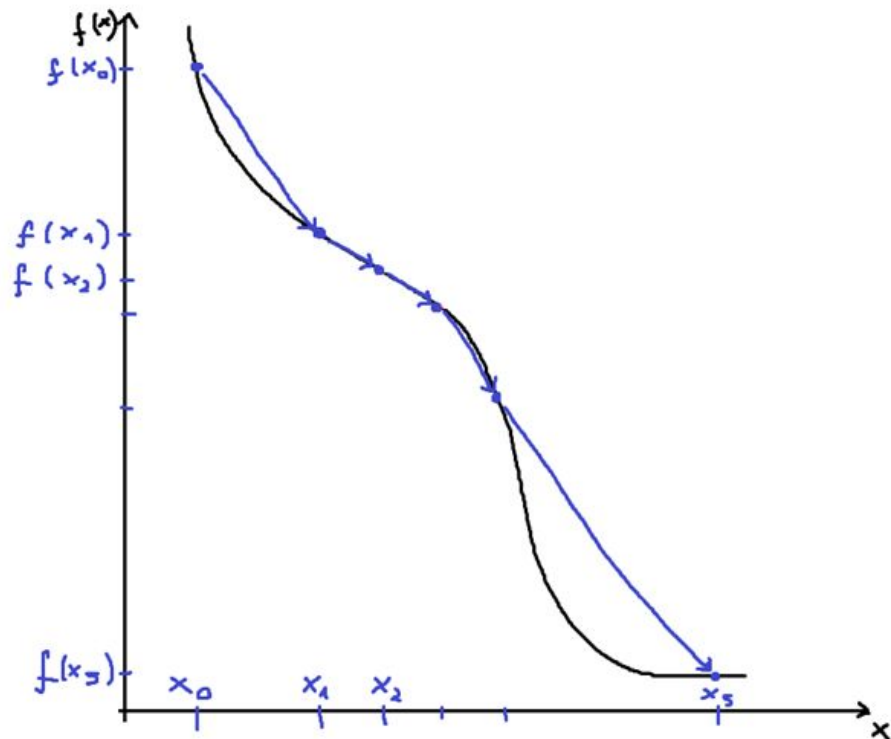
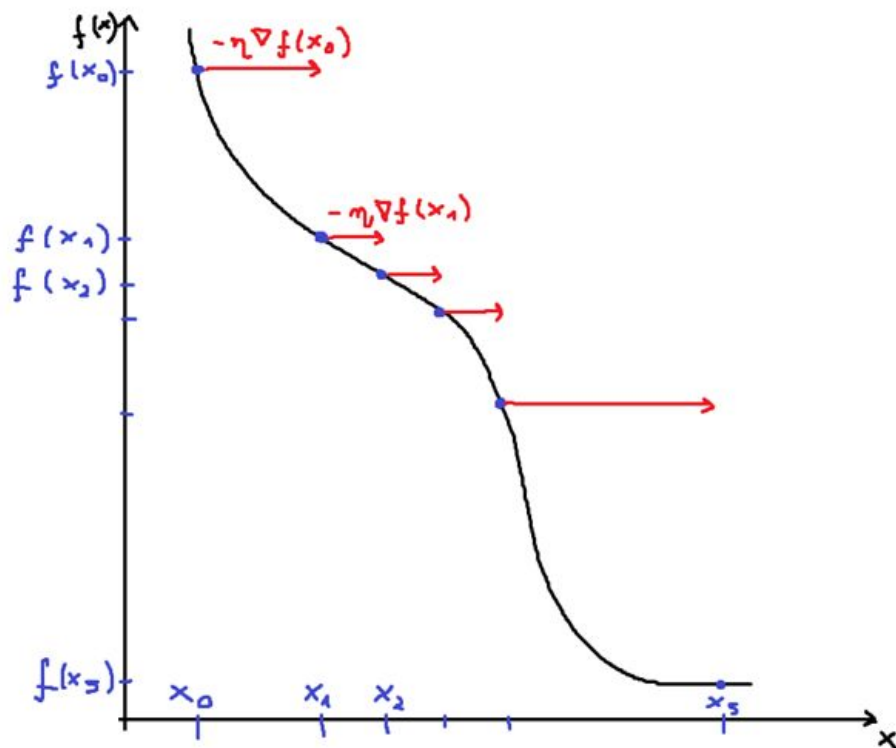
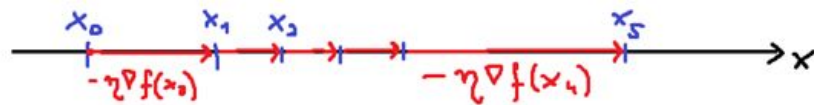
$$\mathcal{L}(w_i + s) \approx \mathcal{L}(w_i) + \nabla \mathcal{L}(w_i)s$$

$$s = -\eta \nabla \mathcal{L}(w_i)$$

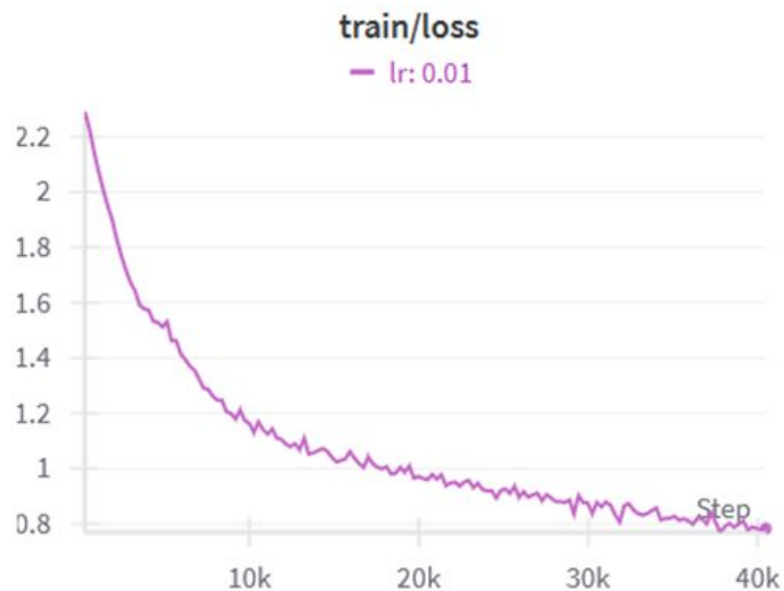
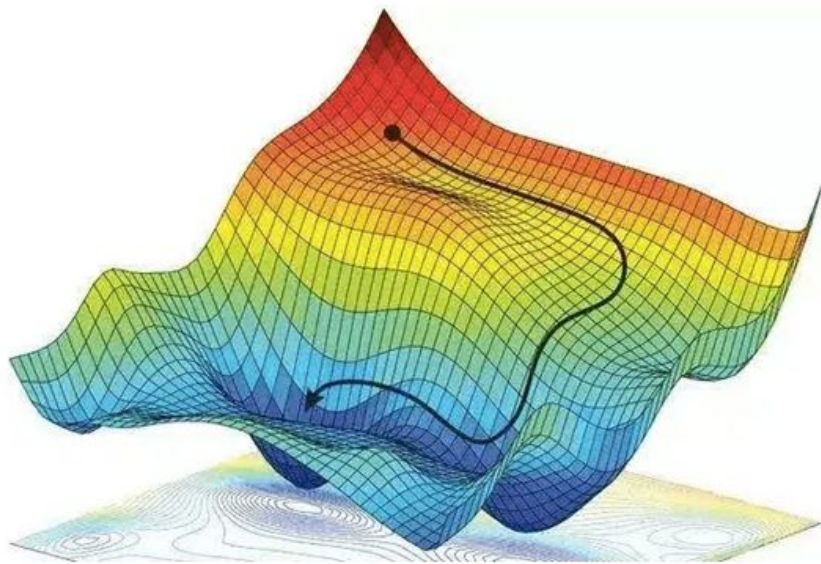


- “najprostsza” metoda optymalizacyjna pierwszego stopnia
- wybiera krok s , w którego kierunku przybliżenie Taylora pierwszego stopnia maleje najszybciej
- ponieważ przybliżenie jest funkcją liniową i nie posiada minimum, długość kroku jest określana za pomocą hiperparametru *learning rate*
- tak się (trochę przypadkiem) składa, że wybrany kierunek to wektor identyczny z gradientem

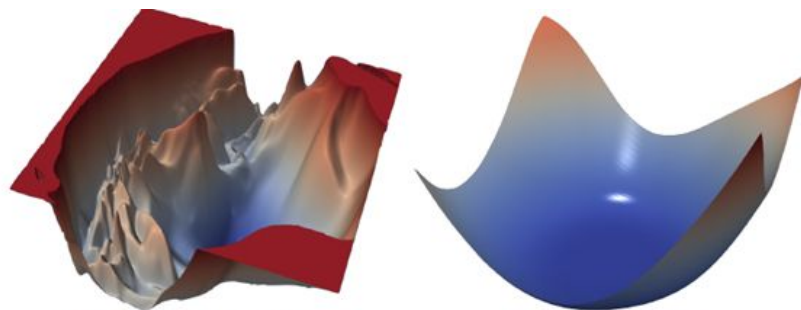
Wizualizacja Gradient Descent



Wizualizacja Gradient Descent



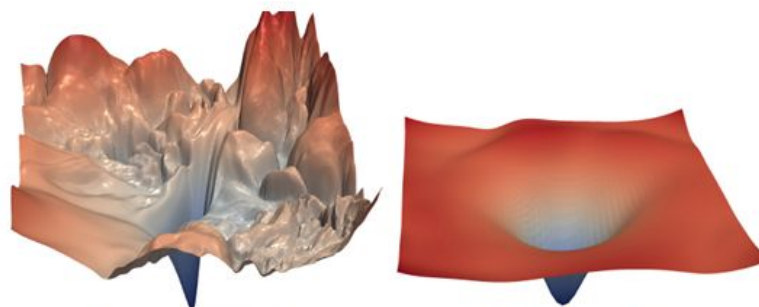
Trudność optymalizacji zależy od ukształtowania funkcji straty, a ono z kolei zależy od architektury



(a) ResNet-110, no skip connections

(b) DenseNet, 121 layers

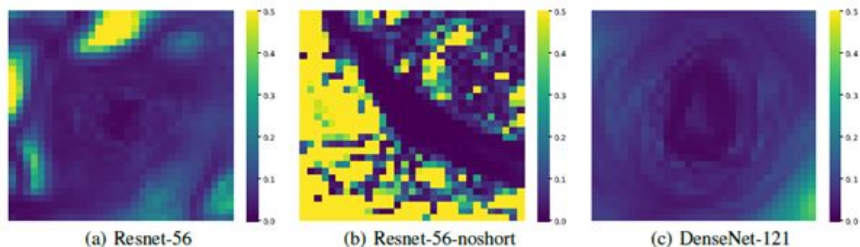
Figure 4: The loss surfaces of ResNet-110-noshort and DenseNet for CIFAR-10.



(a) without skip connections

(b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.



(a) Resnet-56

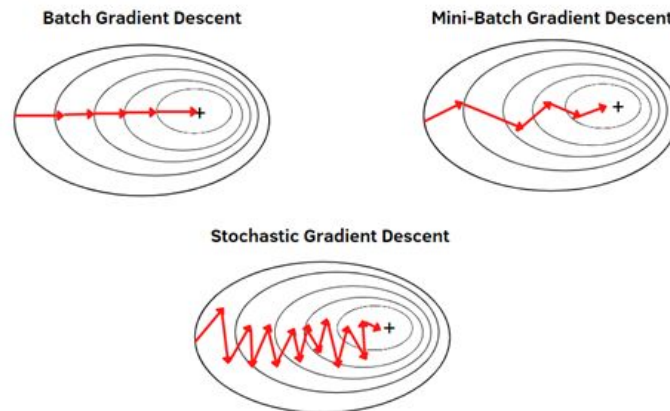
(b) Resnet-56-noshort

(c) DenseNet-121

Figure 7: For each point in the filter-normalized surface plots, we calculate the maximum and minimum eigenvalue of the Hessian, and map the ratio of these two.

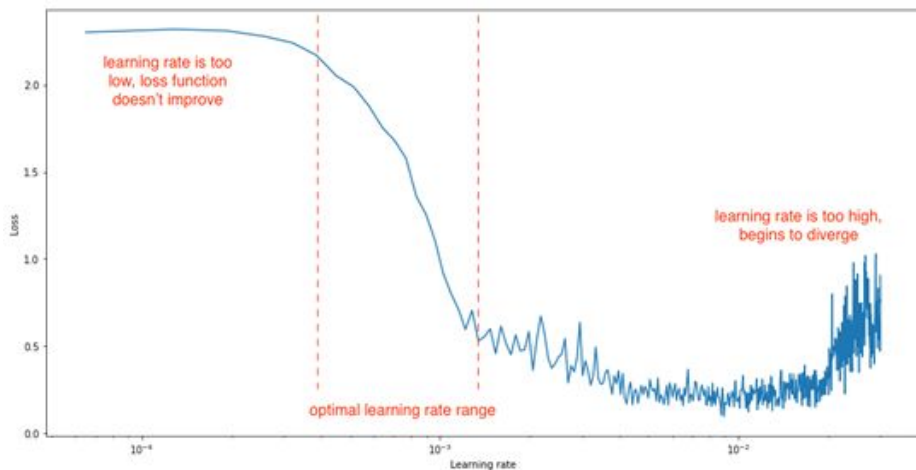
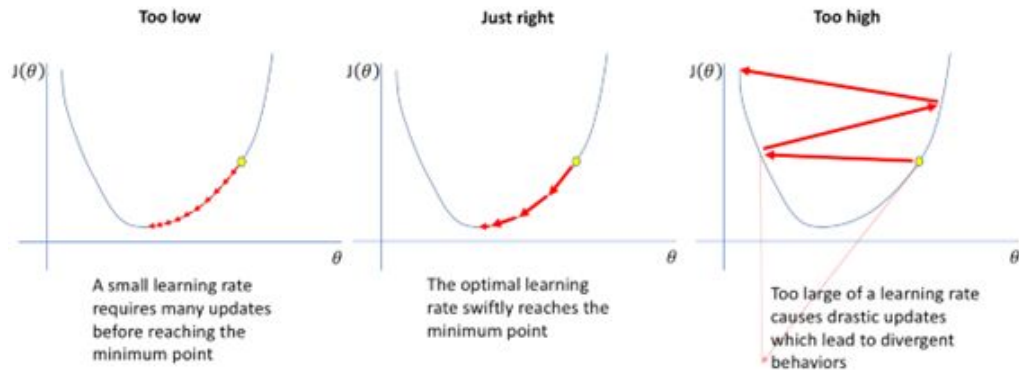
Stochastic Gradient Descent

- aby nie obliczać każdego kroku GD na podstawie całego zbioru danych, używamy podzbiorów (*mini-batches*)
 - zamiast wykonywać bardzo dokładny krok na podstawie 100 elementów, wykonujemy 10 kroków przy użyciu 10 elementów
 - poszczególne kroki mają wtedy większą wariancję, lecz ich średnia jest taka sama (*unbiased estimator*)
 - prowadzi to do szybszej zbieżności i być może pomaga z generalizacją
- dawniej SGD oznaczało ekstremalny przypadek, kiedy do *mini-batch*'a należy tylko jeden element. Współcześnie na większe *mini-batch*'e również mówi się SGD

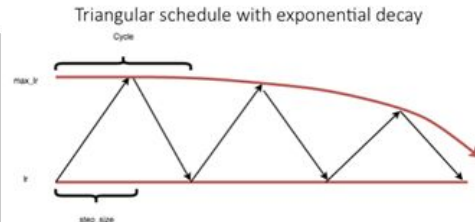
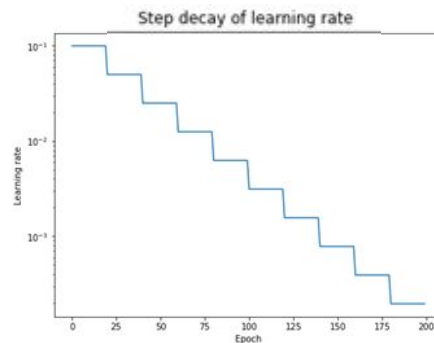


Learning rate

- najważniejszy hiperparametr, kontrolujący rozmiar kroku podczas optymalizacji



- może być modyfikowany w miarę postępów w uczeniu



Przykład: trenowanie Multilayer Perceptron

<https://api.wandb.ai/links/podcast-o-rybach-warsaw-university-of-technology/2qr5y1lj>

<https://github.com/JakubBilski/wztum/blob/main/Code1.ipynb>

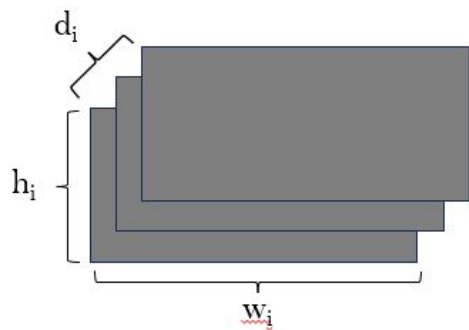
Multilayer perceptron kiepsko radzi sobie z analizą obrazu

- aby stać się wejściem MP, piksele obrazu muszą zostać przekształcone w wektor 1D, co prowadzi do utraty informacji o ich wzajemnym położeniu
- neurony są połączone do wszystkich elementów poprzedniej warstwy, co generuje bardzo wiele parametrów
- różne części obrazu są analizowane w niezależny sposób. Choć ogólnie jest to zaleta, to być może chcielibyśmy wykrywać pewne cechy (np. sylwetkę człowieka) niezależnie od tego, w którym miejscu na obrazie wystąpią
- pomysł: stwórzmy warstwę, której elementami na wejściu są macierze 3D reprezentujące obrazy. Analizujemy tylko piksele blisko siebie (w jakimś niewielkim oknie) i używamy tych samych wag niezależnie od tego, gdzie na obrazie znajduje się piksel, który analizujemy

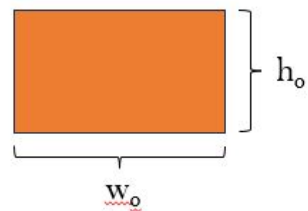
Sieci splotowe (*Convolutional Neural Networks*)

- ... to sieci które zawierają warstwy splotowe, zwykle położone blisko wejścia
- warstwa splotowa przekształca wejście 3D na wyjście 3D. Szerokość i wysokość zwykle ulegają redukcji, podczas gdy liczba kanałów zwykle rośnie
- neurony w warstwie splotowej są połączone do niewielkiej części wejść i dzielą te same wagi z resztą neuronów danej warstwy
- zanim zapiszemy, jak działają wtedy same neurony, wyobraźmy sobie warstwę splotową jako zbiór d_{output} filtrów $(w_{\text{input}}, h_{\text{input}}, d_{\text{input}}) \rightarrow (w_{\text{output}}, h_{\text{output}})$

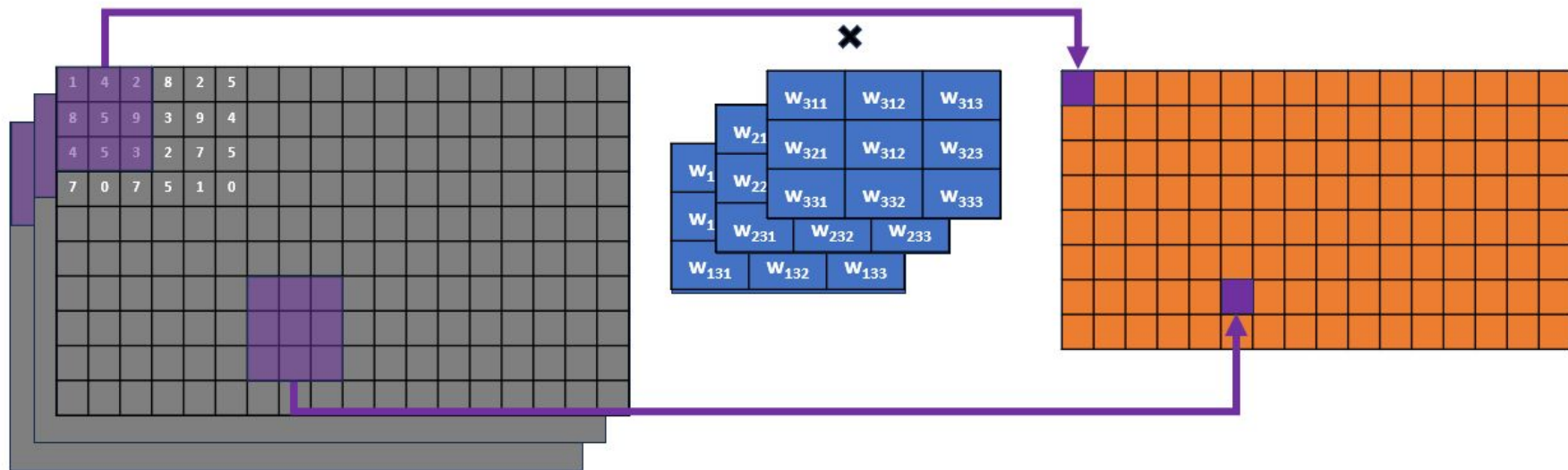




pojedynczy filtr
→

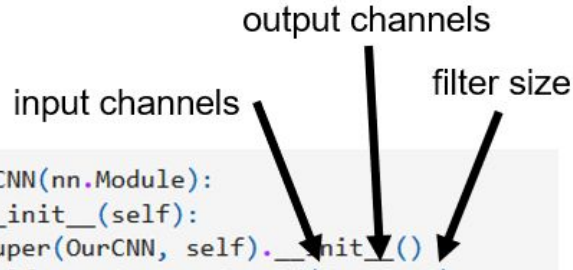


$\sigma [\quad + \quad b \quad]$



Warstwa spłotowa (*convolutional layer*)

- każdy filtr jest stosowany na całym obrazie wejściowym
 - podczas *Backpropagation* gradienty wszystkich wejść zostają zsumowane i razem wpływają na zmianę parametrów
 - ponieważ jeden filtr ma niewiele parametrów, “stać nas” na użycie wielu filtrów w ramach jednej warstwy
 - zachowanie warstwy spłotowej można kontrolować wieloma parametrami.
- Niektóre z nich mają wpływ na kształt wyjścia

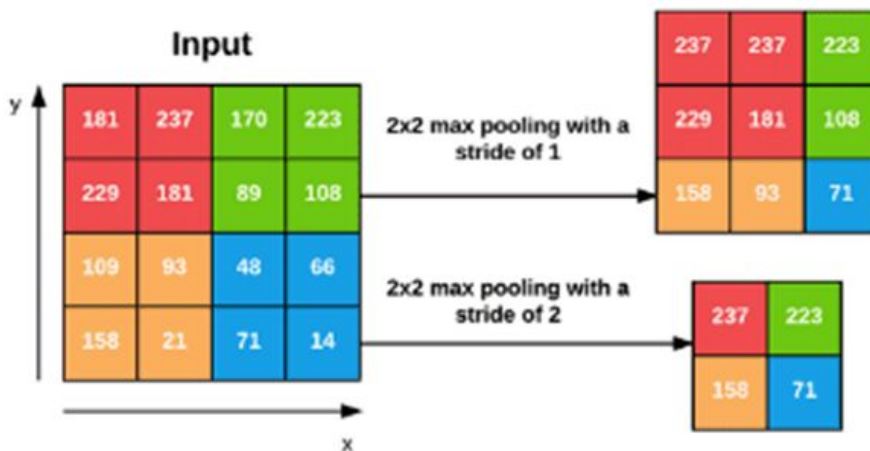


```
class OurCNN(nn.Module):
    def __init__(self):
        super(OurCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 18, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(18, 32, 3)
        self.fc1 = nn.Linear(32 * (INPUT_RESOLUTION//
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, len(CLASS_NAME

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 32 * (INPUT_RESOLUTION//
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Warstwa łącząca (*pooling layer*)

- okazuje się, że w przypadku niewielkich sieci lepsze rezultaty przynosi zastosowanie prostego wygładzenia po warstwie splotowej
- popularne sposoby na wygładzanie to średnia i maksimum
- zmniejsza to wrażliwość małych sieci na szum w danych i *overfitting*



Przykład: trenowanie sieci splotowej

<https://api.wandb.ai/links/podcast-o-rybach-warsaw-university-of-technology/2qr5y1lj>

<https://github.com/JakubBilski/wztum/blob/main/Code1.ipynb>