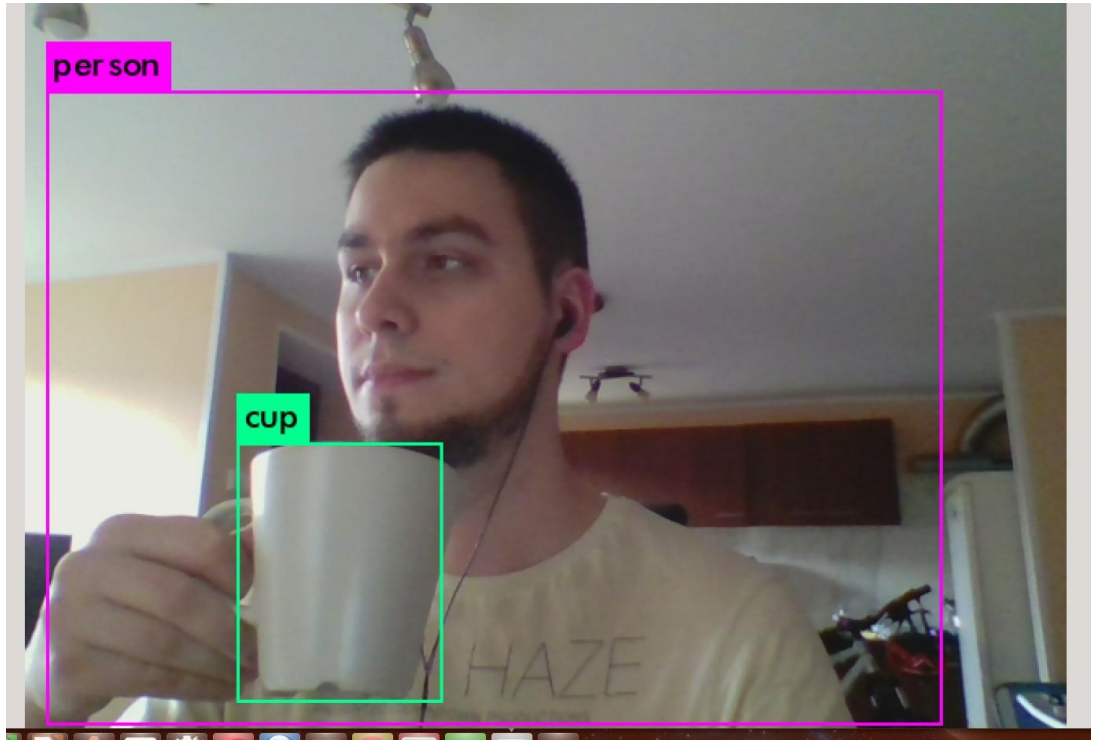
Deep Learning z pakietem Keras w R

Michał Maj

13.04.2020

Kim jestem



Nazywam się Michał Maj:

- Linkedin https://www.linkedin.com/in/michal-maj116/
- Twitter @MichalMaj116

Na co dzień pracuję jako Data Scientist w Billennium.

Absolwent Matematyki Finansowej na Politechnice Gdańskiej. Data Scientist z 5 letnim doświadczeniem. Na co dzień zajmuje się tworzeniem rozwiązań Machine i Deep Learningowych z wykorzystaniem technologii R oraz Python. Do głównych zainteresowań należą zastosowania Deep Learningu w dziedzinie Computer Vision. Współorganizator Trójmiejskiej Grupy Entuzjastów R.

Hobby: fizyka kwantowa i kosmologia, AI, pływanie, origami

Agenda

1. Podstawowe definicje i idea **Deep Learningu**
2. Definicja **Perceptronu**
3. **MLP** – Multilayer perceptron
4. Najważniejsze funkcje aktywacji - **sigmoid**, **tanh**, **ReLU**
5. Czego uczą się **warstwy ukryte** ?
6. Optymalizacja wag metodą **SGD** – implementacja na przykładzie regresji liniowej/logistycznej, a także jednowarstwowym perceptronie.
7. Bardziej zaawansowane metody optymalizacji – **ADAM**, **RMSProp**, **AdaDelta**
8. **Under – overfitting** w sieciach neuronowych
9. Regularyzacja **L1/L2** oraz regularyzacja **dropout**
10. Czym jest **tensorflow** oraz **tensory** ?
11. Czym jest **Keras**?
12. Metody budowy modeli w Keras
13. Budowa MLP w Keras
14. Optymalizacja hiperparametrów z pakietem **tfruns**

Agenda

15. **Sieci konwolucyjne** – warstwa konwolucyjna, poolingu i głęboka
16. **Batch Normalization**
17. Budowa sieci konwolucyjnej w Keras
18. **Callbacks** w Keras – **Early Stopping**, **Model checkpoint**, **LR decay**
19. **Tensorboard**
20. **Generatory danych** w keras
21. **Fine-tuning**
22. **Augmentacja danych**
23. Zaawansowane zadania **Computer Vision** – **segmentacja** i **detekcja**
24. Wizualizacja **filtrów** i **map aktywacji** sieci konwolucyjnej
25. Pakiet/metoda **LIME** z przykładem dla sieci konwolucyjnej
26. **Autoenkodery**
27. Sieci rekurencyjne – **LSTM**, **GRU**
29. Budowa sieci rekurencyjnej w keras

Wymagania

Wymagania:

- R i RStudio
- Python3
- Podstawowa wiedza z zakresu ML i programowania w R
- zainstalowanie pakietów z pliku PACKAGES.R
- instalacja tensorflow i keras (funkcja **install_keras()**)
- (opcjonalnie) instalacja tensorflow/keras w wersji z supportem GPU: https://tensorflow.rstudio.com/installation/gpu/local_gpu/

Część I - Keras, Tensorflow i MLP

O czym będziemy rozmawiać:

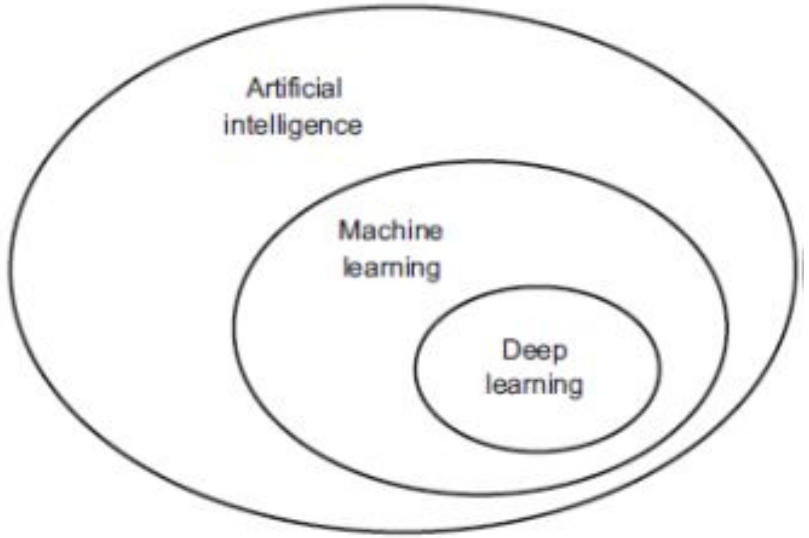
- Deep learning - podstawowe pojęcia, Perceptron, MLP, Funkcje aktywacji, dropout
- Tensorflow, Keras, pakiety pomocnicze, Tensory i graf operacji
- pakiet **reticulate** - komunikacja R z Pythonem
- Budowa sieci neuronowych w Keras [4 zadania]
- Callbacks - model checkpoint, early stopping

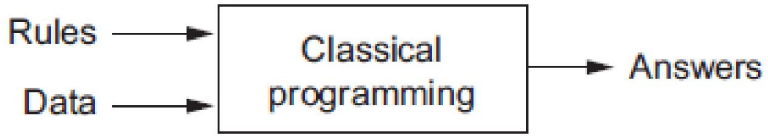
O czym nie będziemy rozmawiać:

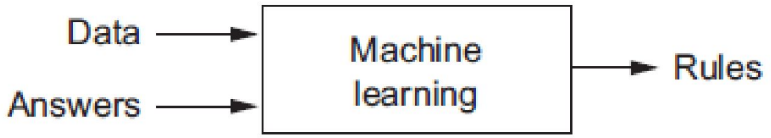
- SGD i inne optymalizatory, batch size i liczba epok [Wykład 3]

Czym jest Deep Learning ?

- **Artificial Intelligence** - zajmuje się modelowaniem i symulowaniem zachowań inteligentnych. W jej skład wchodzą ML, DL, modelowanie ewolucyjne i wiele więcej.
- **Machine learning** - nauka algorytmów modelowania, które uczą się w sposób automatyczny (na podstawie danych i odpowiedzi, a nie ręcznie zdefiniowanych reguł).
- **Deep learning** - subdziedzina ML, skupiająca się na uczeniu warstwowym (warstwowej ekstrakcji danych).

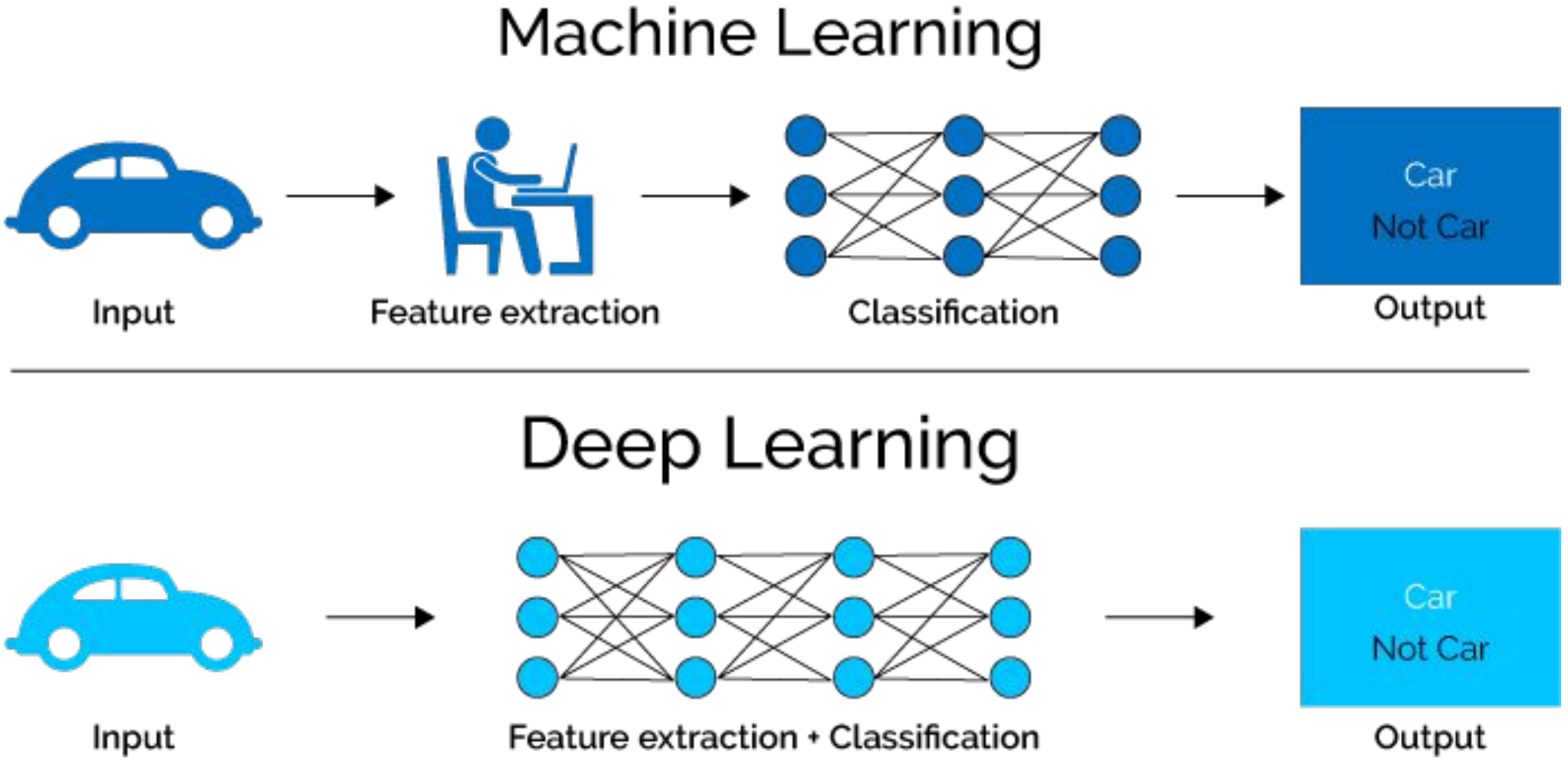






Źródło: Allaire J. J., Chollet F., *Deep learning with R,* Manning 2018

Uczenie warstwowe



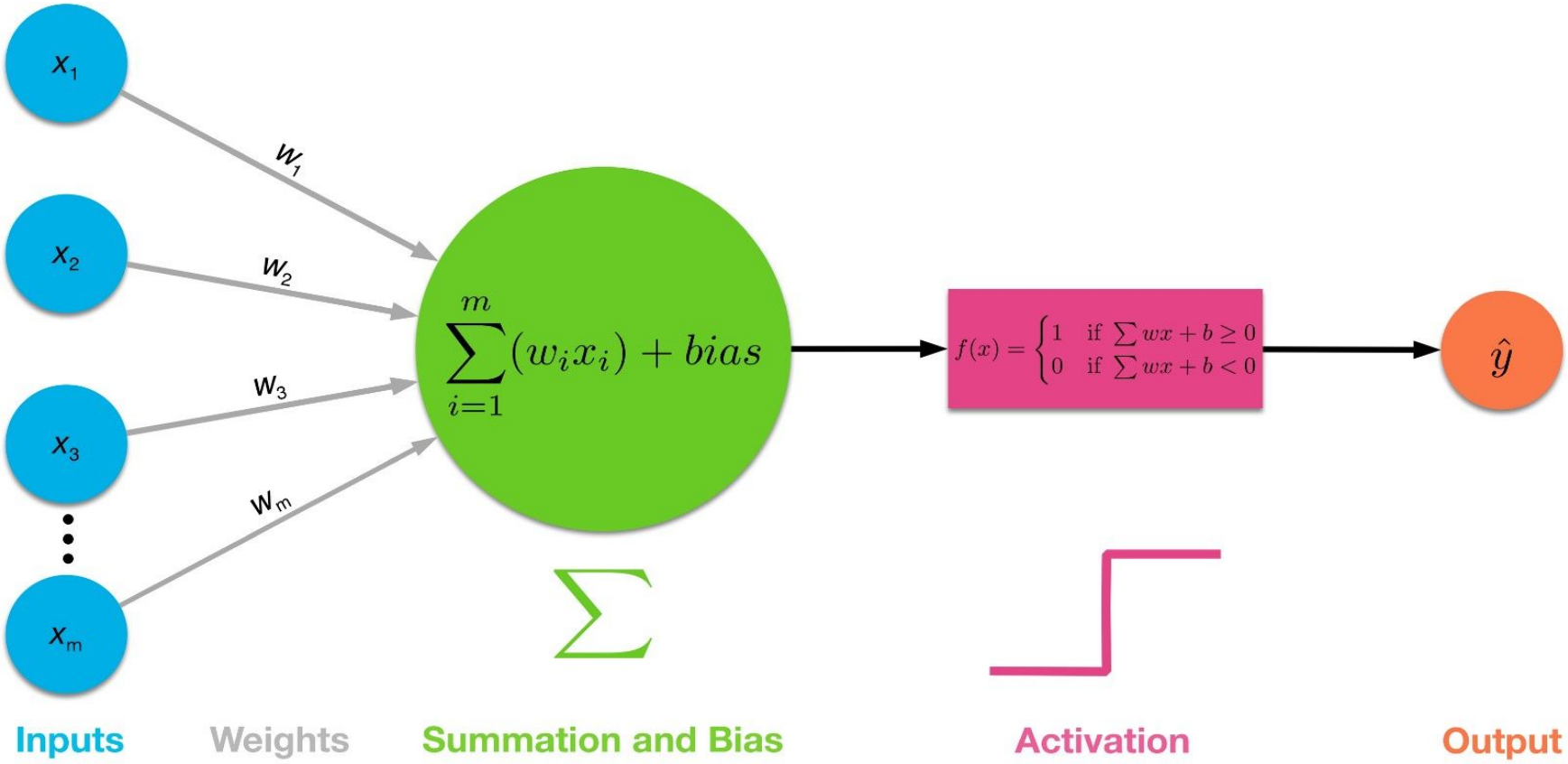
Źródło: https://www.wilfultriumph.com/Blog/Technology/difference-between-deep-learning-and-usual-machine-learning?id=38

Dlaczego sieci neuronowe są tak popularne ?

- **Prostota** – sieci neuronowe zastępują dotychczasowe, skomplikowane procesy analityczne ze skomplikowanym feature engineering,
- **Skalowalność** – sieci neuronowe są łatwe do zrównoleglenia i przetwarzania przez GPU (procesory graficzne) i TPU (Tensor Processor Unit)
- **Wszechstronność** – możliwość użycia w każdym zadaniu
- **Możliwość ponownego użycia** – wytrenowane sieci neuronowe można używać do innych zadań lub modyfikować o nowe dane bez reestymacji całego modelu na pełnej próbie - **Transfer learning**

Perceptron (neuron)

W **perceptronie**, wynikiem jest **liniowa kombinacja inputów** (oraz wyrazu wolnego - **biasu**) przekształcona pewną (zazwyczaj) nieliniową **funkcja aktywacji**.



Funkcje aktywacji

Wybór odpowiedniej funkcji straty może decydować o wyniku modelu. Istnieje wiele możliwych funkcji aktywacji, przykłady tych najbardziej popularnych to **ReLU** i **Sigmoid**.

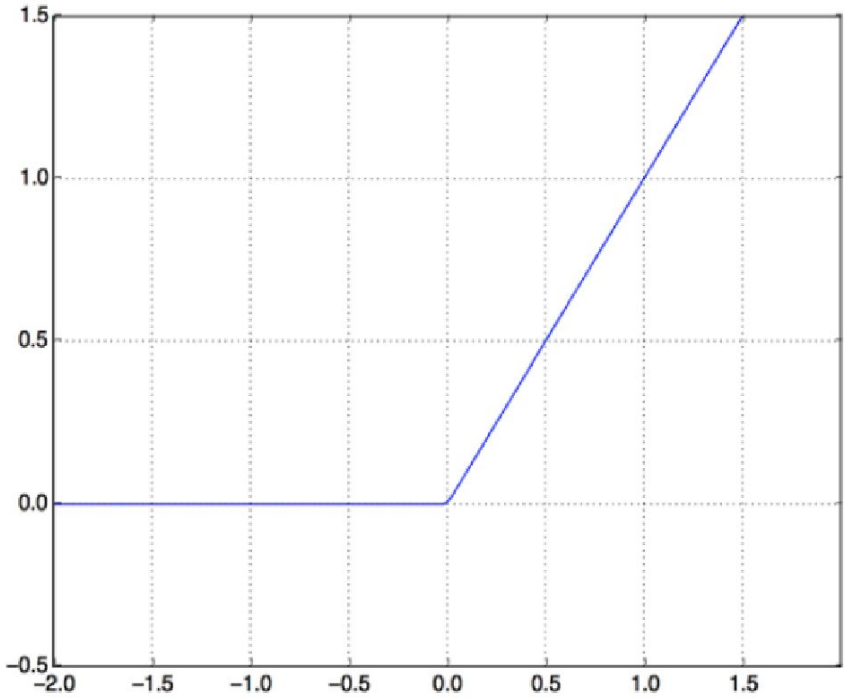


Figure 3.4 The rectified linear unit function

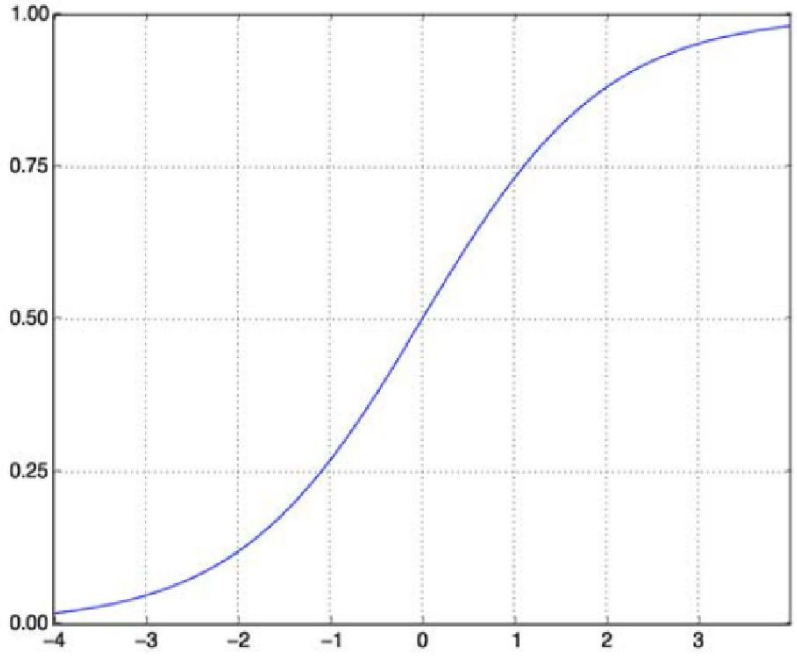


Figure 3.5 The sigmoid function

Funkcje aktywacji

Zalety ReLU jako funkcji aktywacji:

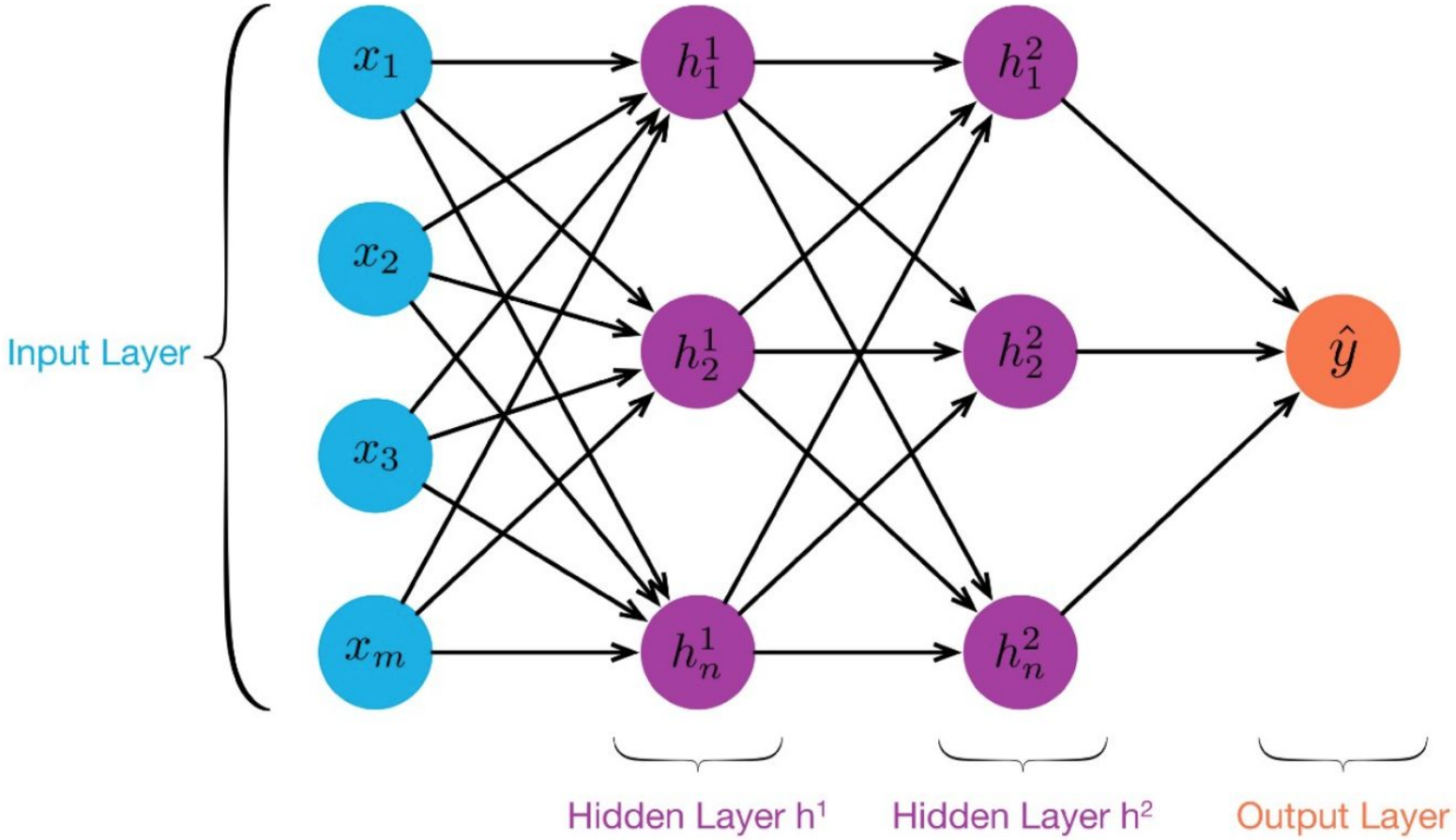
- prostsze i co za tym idzie szybsze obliczenia (proste do obliczenia wartości funkcji jak i wartości funkcji pochodnej)
- brak zanikających gradientów (częsty problem z funkcją sigmoid)
- w praktyce sieci z ReLU szybciej zbiegają (algorytmy typu SGD itd. potrzebują mniej czasu)
- "wygłuszenie" niepotrzebnych sygnałów (nie pojawiają się wartości < 0, co jest korzystne w niektórych przypadkach)

Wady ReLU jako funkcji aktywacji:

- problem martwych neuronów (losowa inicjalizacja wag przy tworzeniu sieci może spowodować w przypadku ReLU otrzymanie wartości 0 już na starcie przez co neuron ten będzie "martwy" w procesie uczenia)

Multilayer perceptron - MLP

MLP zbudowany jest z co najmniej 3 **warstw** (input, warstwy ukryte i output). Możemy myśleć o neuronach w warstwach ukrytych jako o "**nowych zmiennych**" utworzonych w sposób automatyczny z naszych oryginalnych predyktorów



Architektura sieci

Najbardziej widocznym elementem modelu deep learningowego jest architektura sieci, najważniejsze elementy:

Liczba warstw – powinna rosnąć ze stopniem skomplikowania problemu. Pierwsza warstwa uczy się ogólnych i prostych reguł. Wraz ze wzrostem specjalizacji kolejne warstwy uczą się coraz bardziej subtelnych szczegółów – można to porównać do liniowych i nieliniowych klasyfikatorów – pierwsza warstwa wychwytuje liniowe zależności, następne szukają nieliniowych.

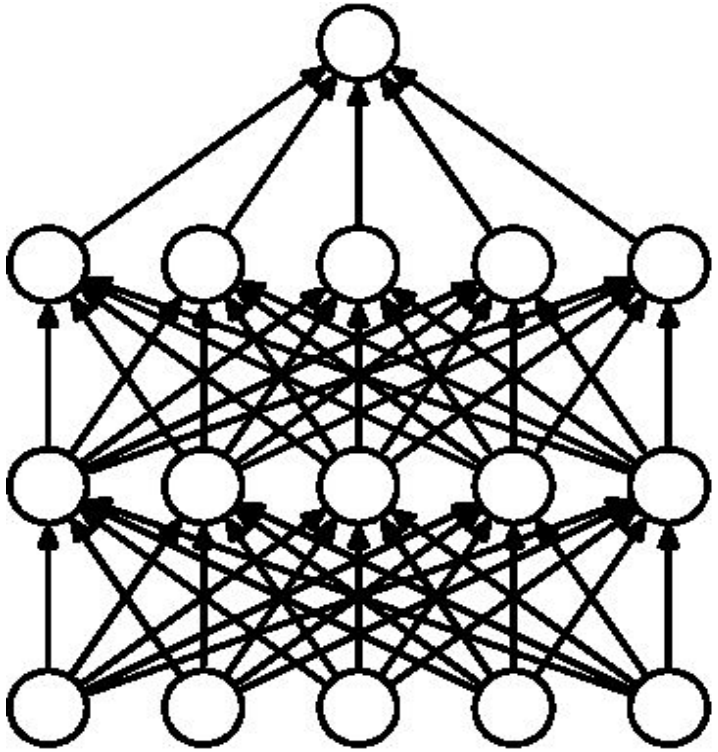
Liczba jednostek w warstwach – im więcej jednostek w warstwach, tym większe dopasowanie do zbioru uczącego (którego przy odpowiednio dużej sieci można się nauczyć na pamięć) i większe ryzyko przetrenowania. W praktyce pokazano, że nawet sieci o małej liczbie parametrów, ale dostatecznie dużej liczbie iteracji i warstw są w stanie nauczyć się skomplikowanych wzorców.

Rodzaj funkcji aktywacyjnych –funkcje aktywacyjnych wybieramy dla każdej warstwy. Bardzo często jest to ReLU.

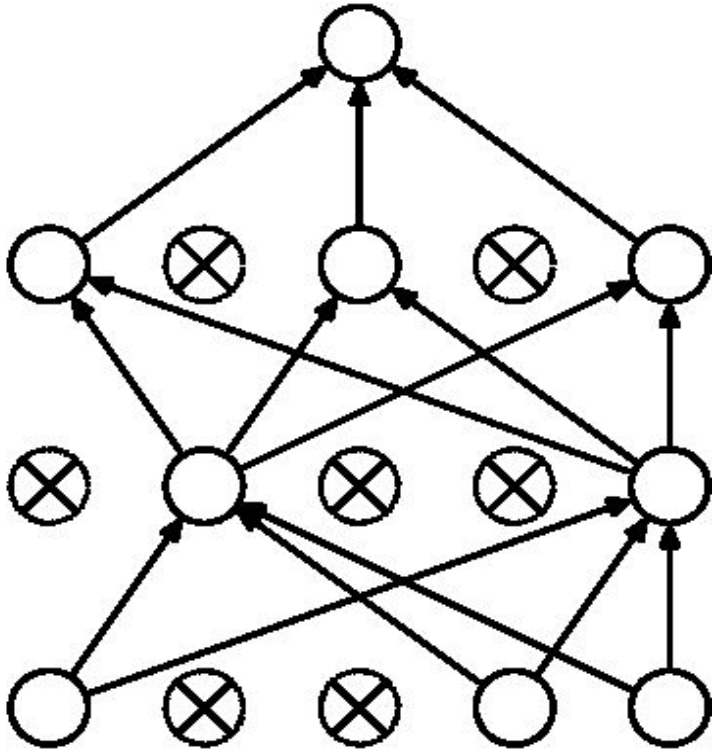
Funkcje regularyzujące (L1, L2, L1/L2) – (ewentualny) wybór formy regularyzacji i jego parametru. W praktyce **dropout** jest skuteczniejszy.

Dropout

Warstwy dropout – ciekawy zabieg, polegający na losowej dezaktywacji części neuronów w każdej iteracji. Inne neurony muszą wtedy przejąć rolę tych wyłączonych przez co ich funkcjonalności zaczynają się dublować – zapobiega przetrenowaniu modelu.



(a) Standard Neural Net



(b) After applying dropout.

Najważniejsze hiperparametry w sieciach klasy MLP

Konstrukcja sieci:

- Liczba warstw
- Liczba unitów (neuronów w warstwie)
- Typ funkcji aktywacyjnej
- Dropout
- Regularyzacja L1 i L2

Parametry dopasowania:

- Liczba iteracji (epoch)
- Liczba obserwacji w batchu
- Wybór optymalizatora
- Wybór parametrów optymalizatora:

1. learning rate (jak w boostingu)
2. momentum (średnia ważona dla gradientów z różnych iteracji)
3. decay – stopniowe zmniejszanie learning rate pomiędzy iteracjami

Czym jest TensorFlow ?

TensorFlow:

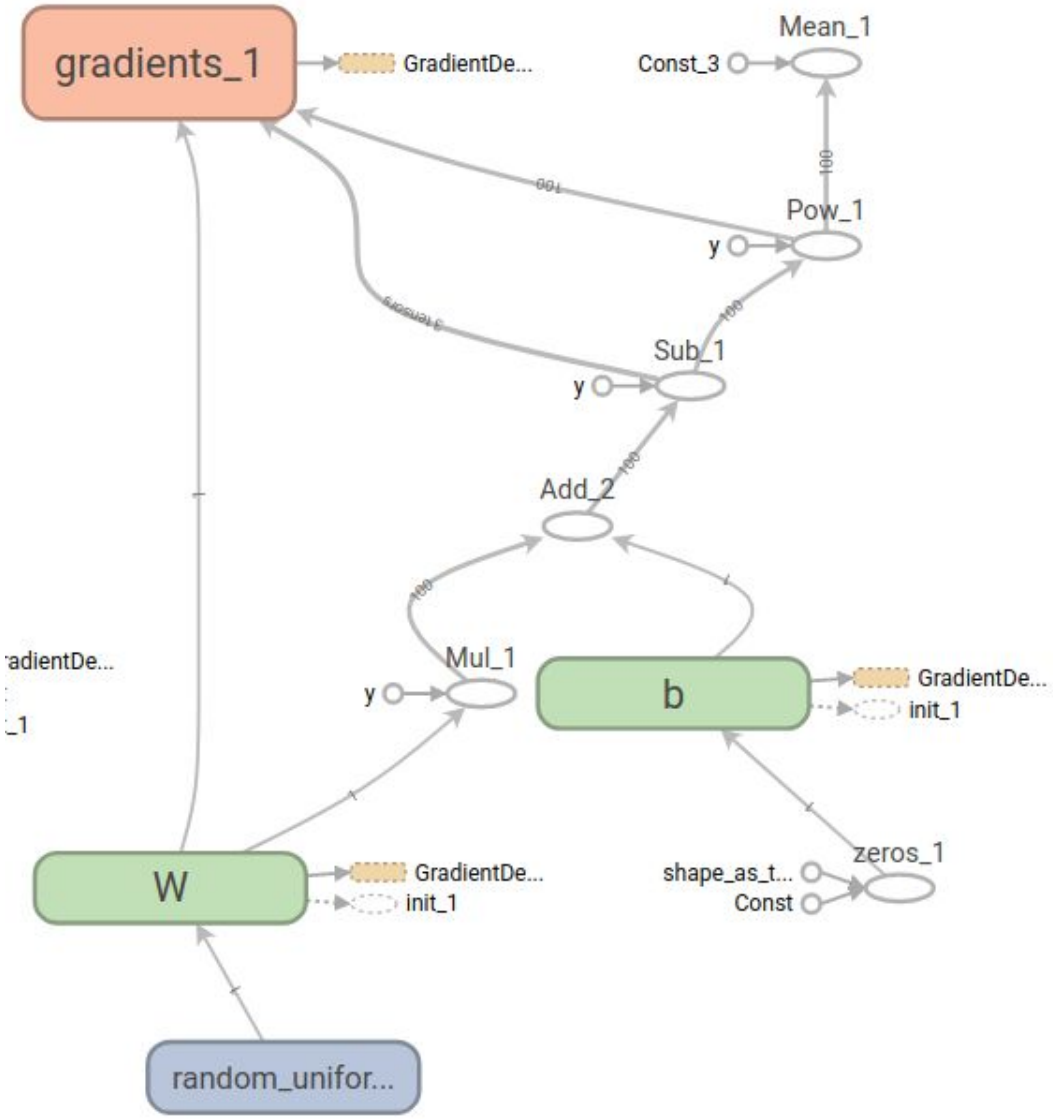
- jest biblioteką obliczeń **ogólnego przeznaczenia** (nie tylko Deep Learning!) napisana w C++.
- jest oprogramowaniem **open-source**.
- pozwala na obliczenia na **wielu CPU, GPU i TPU**.
- został opracowany przez naukowców i inżynierów z **Google Brain Team.**

Czym są tensory ?

Możesz myśleć o **tensorze** jako o tablicy wielowymiarowej:

Data	Tensor
Vector data	2D tensor: (samples, features)
Time Series data	3D tensor: (samples, timestep, features)
Image	4D tensor: (samples, height, width, channels)
Video	5D tensor: (samples, frames, height, width, channels)

“flow” tensorów



Graf opisuje operacje przeprowadzone na **tensorach**. Każdy wierzchołek grafu przyjmuje jeden lub więcej tensorów, przeprowadza na nich pewne **operacje** (obliczenia) i oddaje jeden lub więcej wektorów.

```
1  library(tensorflow)
2
3  x_data <- runif(100, min=0, max=1)
4  y_data <- x_data * 0.1 + 0.3
5
6  W <- tf$Variable(tf$random_uniform(shape(1L), -1.0, 1.0), name = "W")
7  b <- tf$Variable(tf$zeros(shape(1L)), name = "b")
8  y <- W * x_data + b
9
10 loss <- tf$reduce_mean((y - y_data) ^ 2)
11 optimizer <- tf$train$GradientDescentOptimizer(0.5)
12 train <- optimizer$minimize(loss)
13
14 sess = tf$Session()
15 sess$run(tf$global_variables_initializer())
16
17 sess$close()
```

Czym jest Keras ?

Keras:

Keras jest **wysokopoziomowym interfejsem** pozwalającym budować sieci neuronowe z wykorzystaniem różnych backendów takich jak: **TensorFlow**, **CNTK**, lub **Theano**. Jedną z jego największych zalet jest jego "łatwość obsługi" - w bardzo prosty sposób możemy tworzyć zaawansowane modele jak **sieci konwolucyjne** czy **sieci rekurencyjne**.

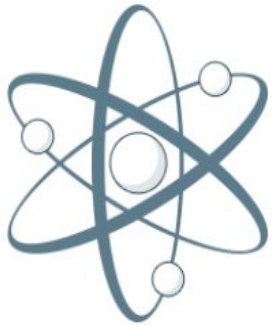
Keras i Tensorflow w R

Keras API

The Keras API for TensorFlow provides a high-level interface for neural networks, with a focus on enabling fast experimentation.

Estimator API

The Estimator API for TensorFlow provides high-level implementations of common model types such as regressors and classifiers.



Core API

The Core TensorFlow API is a lower-level interface that provides full access to the TensorFlow computational graph.

Narzędzia dodatkowe

- tfruns—Trackowanie, wizualizacja i zarządzanie. Tuning hiperparametrów.
- tfdeploy—Narzędzia zaprojektowane tak, aby eksportowanie i obsługa modeli TensorFlow była prosta.
- cloudml—interfejs R dla Google Cloud Machine Learning Engine.

Typy modeli w Keras

W Keras modele możemy budować na 3 sposoby:

1. Używając **modelu sekwencyjnego** (warstwa po warstwie):
   - MLP, CNN, RNN

2. Używając **API funkcyjnego** (sieci z wieloma inputami/outputami,):
   - Object detection (f.e. Faster R-CNN)
   - Generative adversarial networks (GANs)

3. Stosując **wstępnie wyuczone modele (pre-trained models)**

Budowa modelu w Keras

Aby zbudować najprostszy model w Keras musimy wykonać następujące kroki:

1. **Zdefiniować architekturę modelu** - wybrać typ modelu, dodać warstwy ukryte, regularyzacje, inputy i outputy.

2. **Skompilować model** - wybrać funkcję straty i metodę optymalizacji.

3. **Wytrenować model**

4. **Dokonać ewaluacji / predykcji**

Funkcje straty w Keras

Najważniejsze funkcje straty:

Loss	Task
“mse”, “mae”	regression
“binary_crossentropy”	binary classification
“categorical_crossentropy”	multiclass classification
“hinge”	classification

Istnieje możliwość stworzenia własnej funkcji straty.

Optymalizatory w Keras

Najważniejsze optymalizatory:

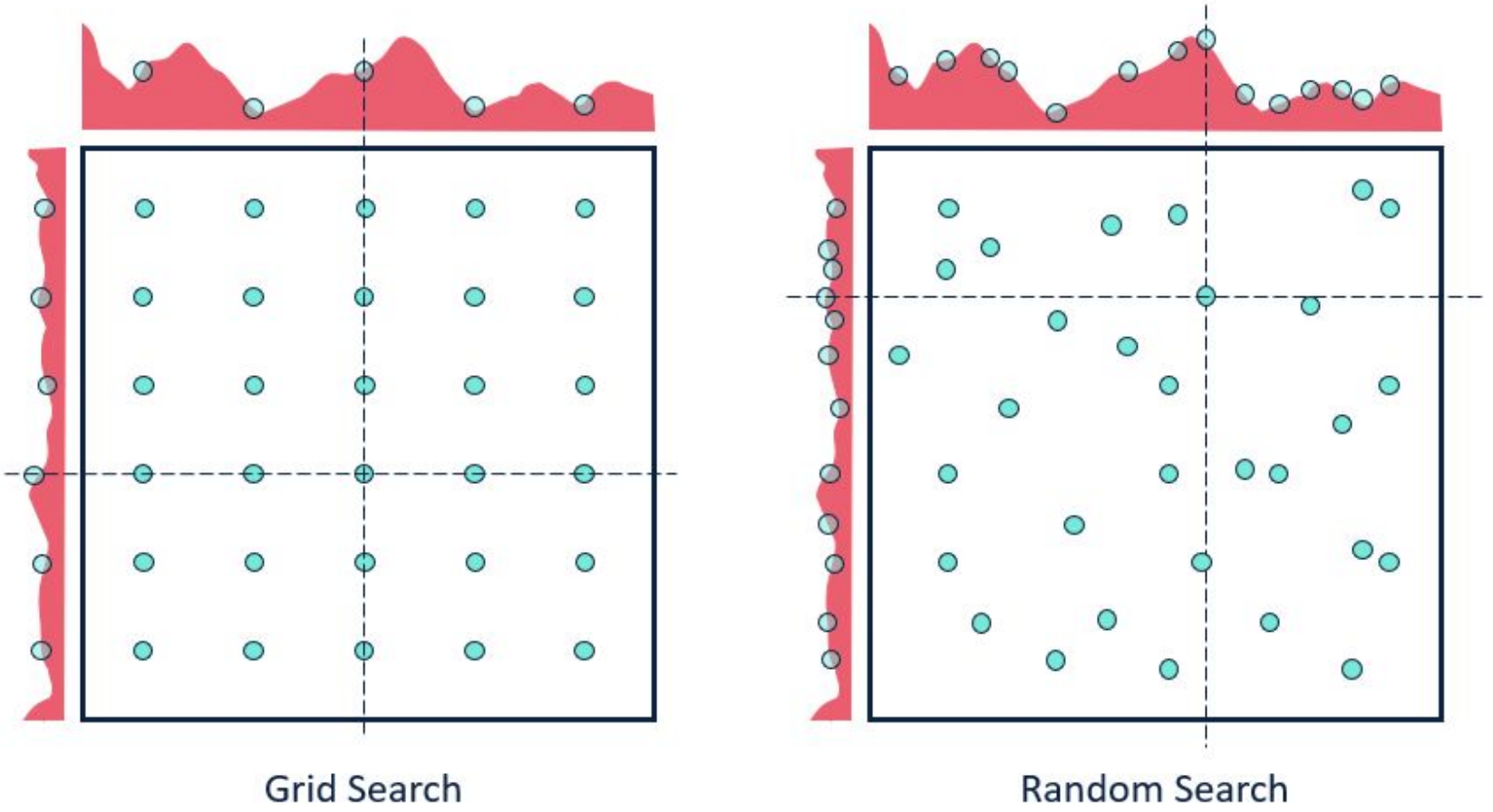
Loss	Function
“sgd”	optimizer_sgd()
“rmsprop”	optimizer_rmsprop()
“adam”	optimizer_adam()
“adadelta”	optimizer_adadelta()

Część II - optymalizacja hiperparametrów z tfruns

O czym będziemy rozmawiać:

- grid search, random search
- pakiet tfruns

Grid i random search



Źródło: https://community.alteryx.com/t5/Data-Science-Blog/Hyperparameter-Tuning-Black-Magic/ba-p/449289

Część III - SGD i backpropagation

O czym będziemy rozmawiać:

- Pochodne i znajdowanie minimum funkcji
- Liczba epok i batch size
- GD / SGD:
  - dla zwykłej funkcji
  - dla regresji liniowej
  - dla regresji logistycznej
  - dla MLP (backpropagation)

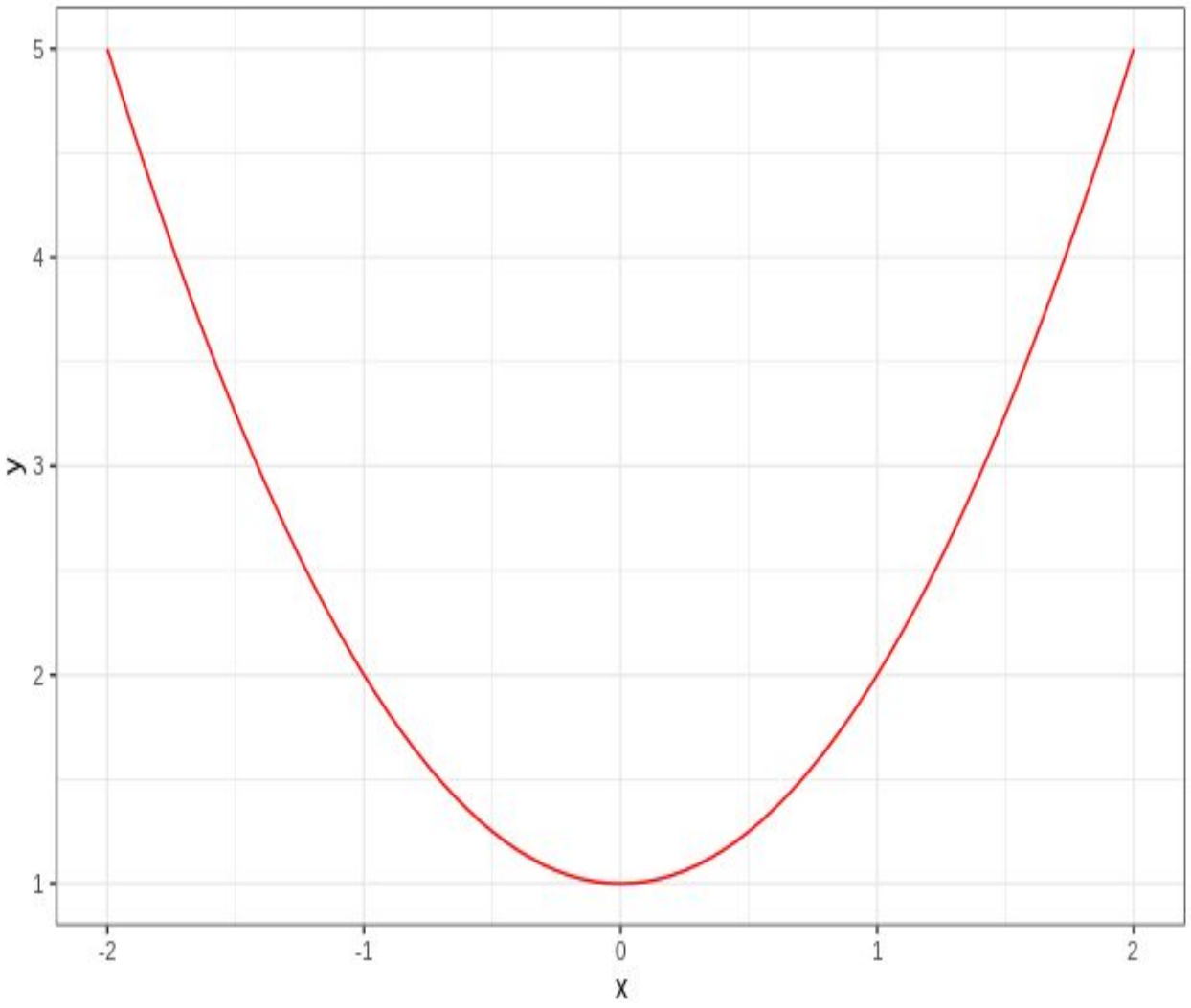
Minimalizacja funkcji straty

W większości modeli ML/DL mamy określony pewien **model**, w którym optymalizujemy pewne **parametry** poprzez **minimalizację** pewnej **funkcji straty**.

Minimum owej funkcji możemy znaleźć na następujące sposoby:

1. Bezpośrednio (tylko dla prostych funkcji, w ML/DL niewykonalne)
2. Znajdując wartości parametrów, w których **pochodna** danej funkcji jest równa 0:
   - Bezpośrednio
   - Stosując algorytm iteracyjny np. **GD** / **SGD**

Pochodna

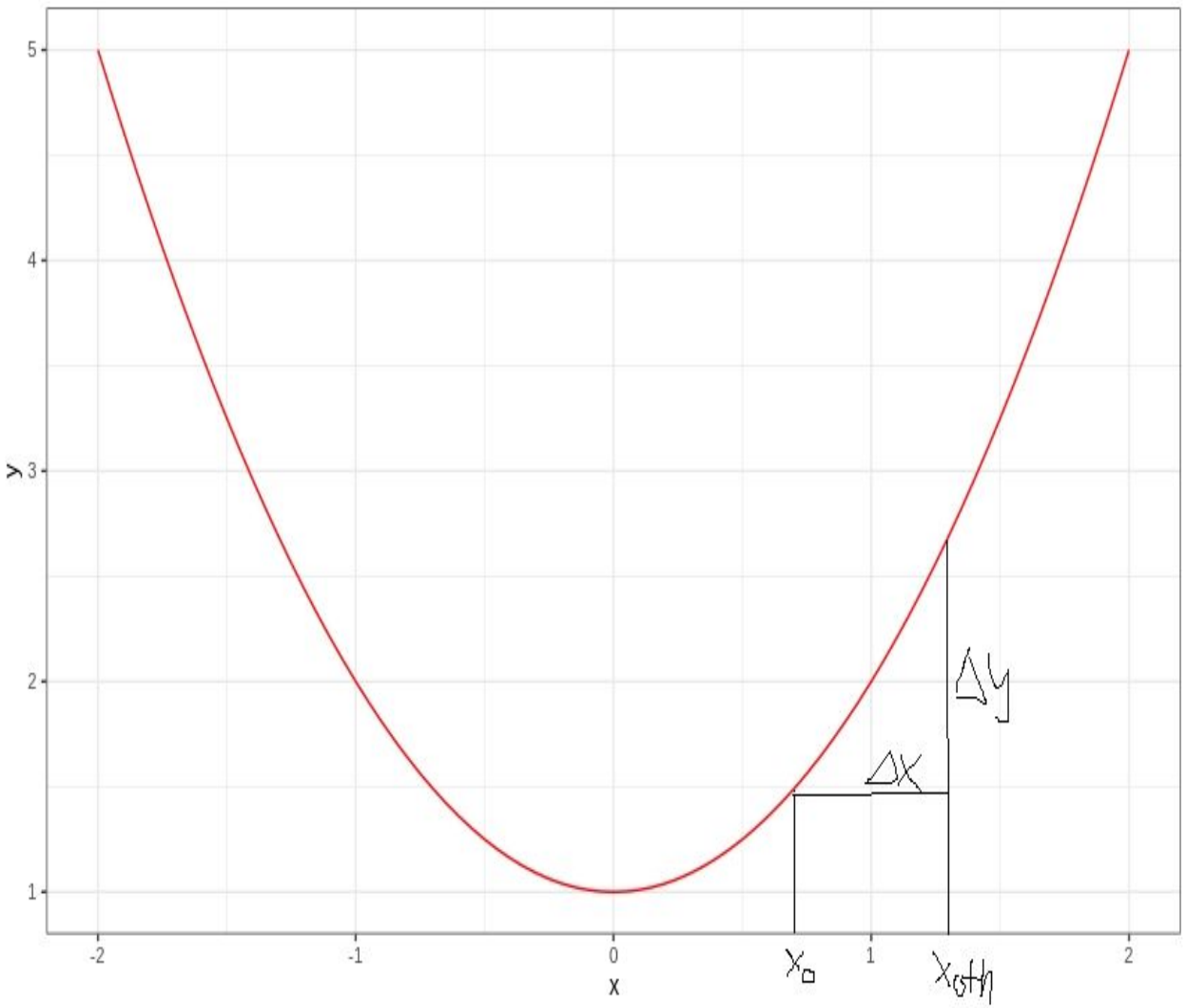


Zacznijmy od bardzo prostej funkcji **$f(x) = x^2+1$** i zastanówmy się gdzie znajduje się jej minimum.

Nietrudno odgadnąć, że funkcja ta osiąga minimum gdy **x=0**.

Przypuśćmy jednak, że nie jesteśmy w stanie znaleźć minimum bezpośrednio.

Pochodna



Wybierzmy dowolny punkt x_0 i sprawdźmy jak zmienia się wartość naszej funkcji gdy przemieszczamy się o dowolną liczbę jednostek **h**.

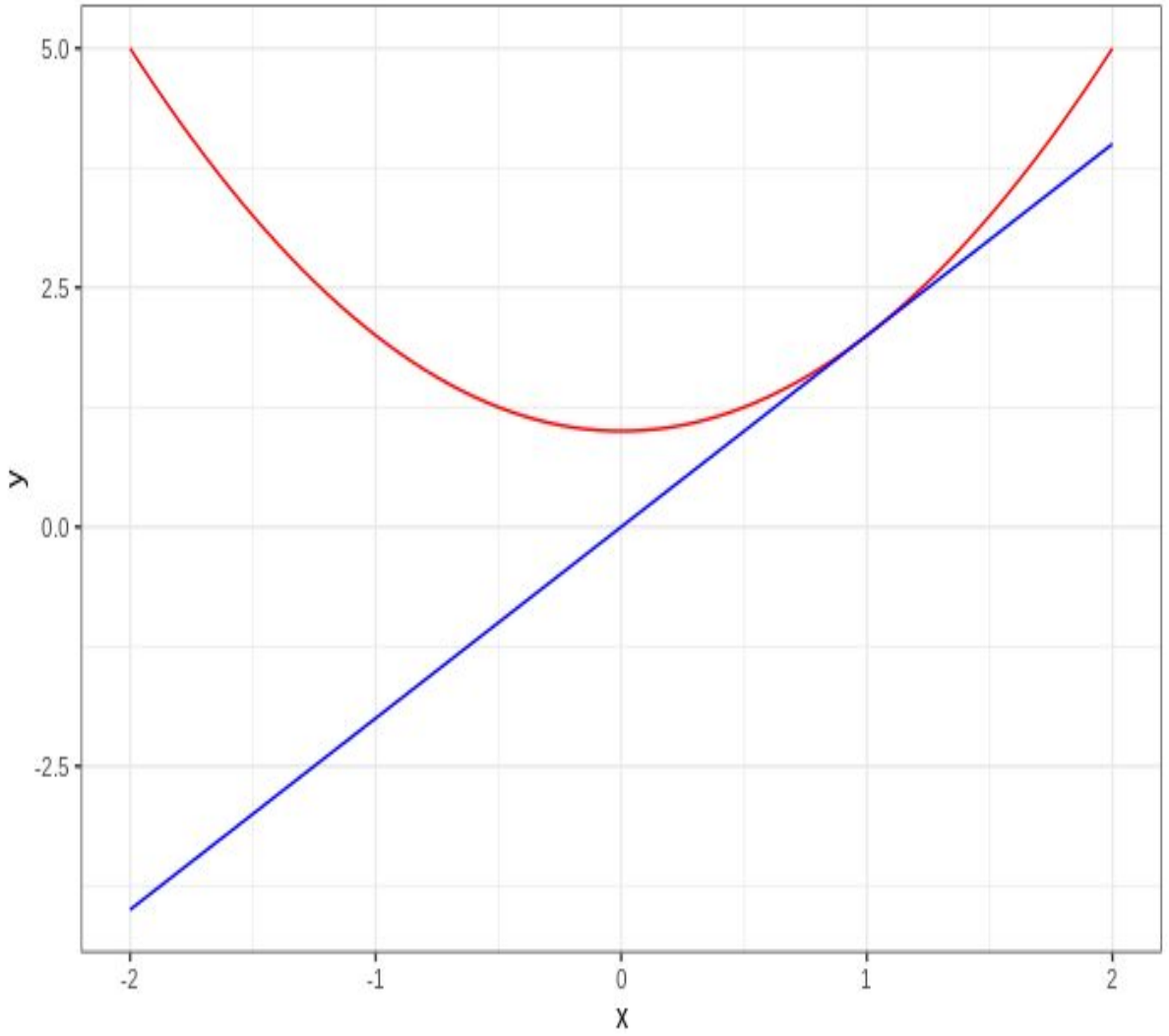
$$\Delta y/\Delta x = ((x_0 + h)^2 + 1 - x_0^2 - 1) / (x_0 + h - x_0) =$$

$$(x_0^2 + 2hx_0 + h^2 + 1 - x_0^2 - 1) / h =$$

$$(2hx_0 + h\hat{}2) / h = 2x_0 + h$$

Jak widzimy wartość **Δy/Δx** zależna jest od kroku **h**, aby temu zaradzić, możemy sprawdzić co stanie się gdy będziemy zmniejszać h (policzymy granicę gdy h zbiega do 0).

Pochodna



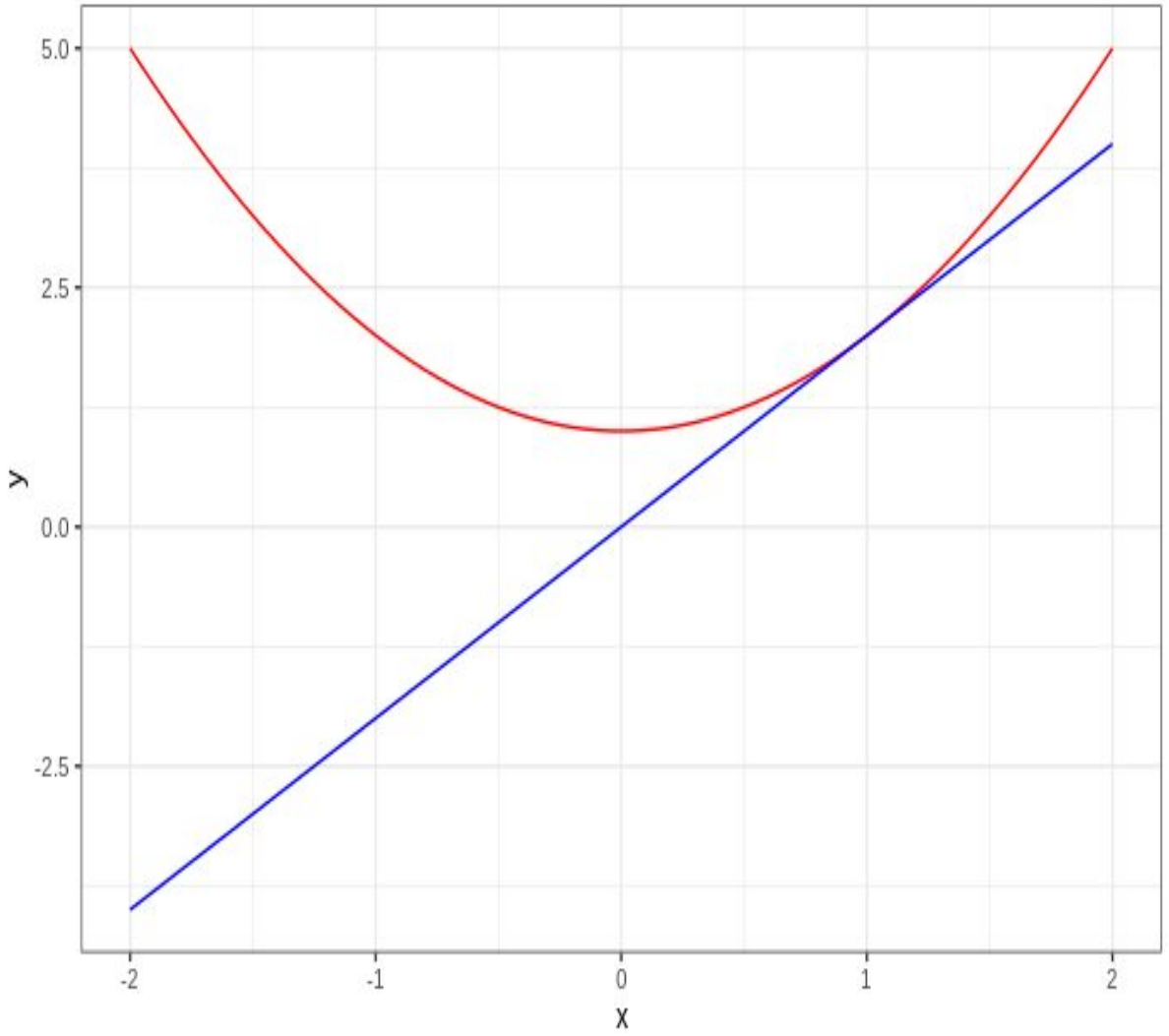
Otrzymamy wtedy: wartość $\mathbf{2x_0}$, uogólniając dla dowolnego **x** możemy zapisać jako funkcję:

f'(x) = 2x,

którą nazywać będziemy **pochodną funkcji f(x)**. Pochodna mówi nam o natychmiastowej zmianie funkcji f(x).

Jeśli o wartościach funkcji f(x) pomyślimy jako o przebytej drodze w pewnym czasie x to o pochodnej możemy myśleć jako o prędkości w danym momencie x.

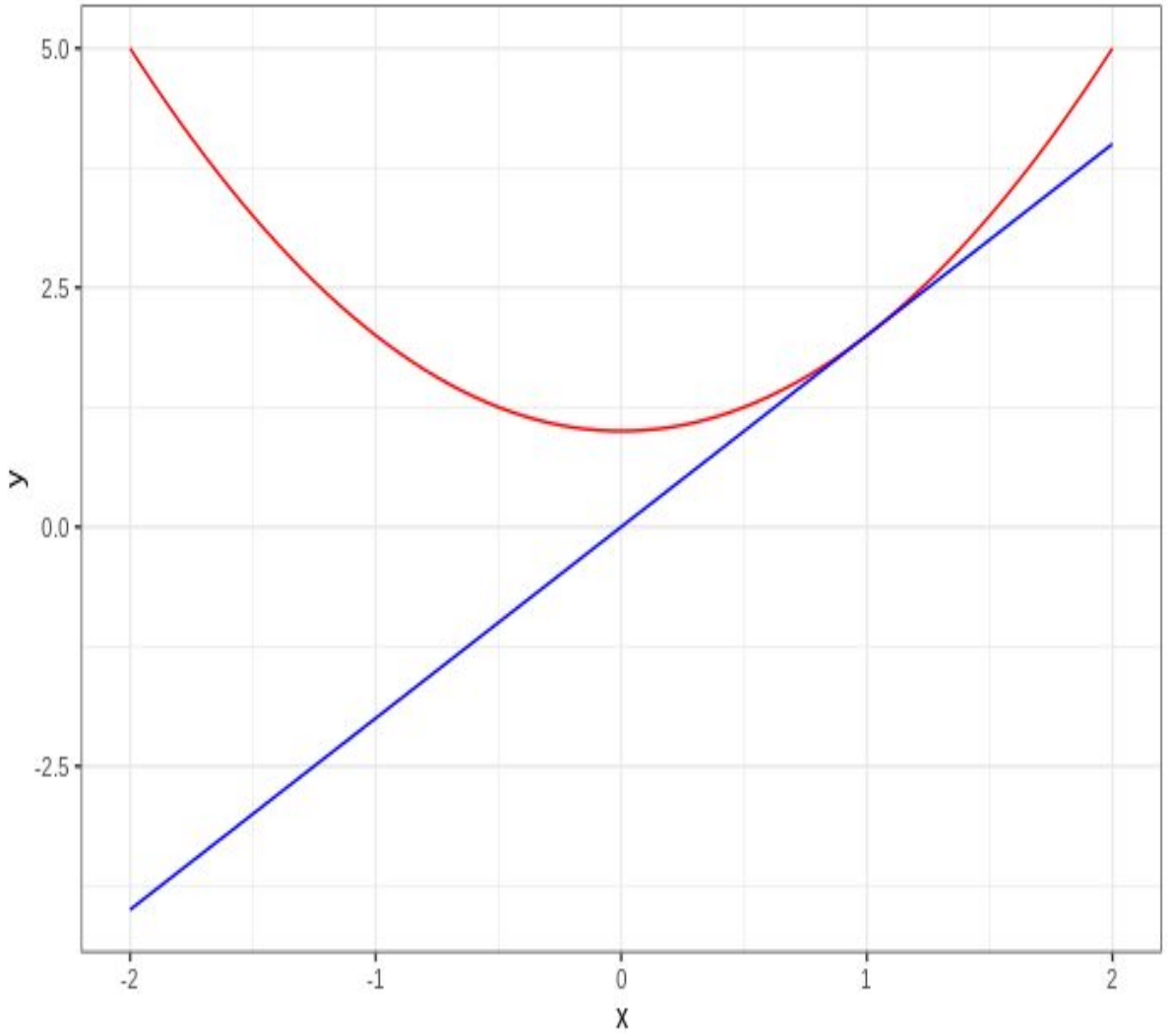
Pochodna



Pochodna funkcji ma kilka użytecznych wartości, które pomogą nam w znalezieniu minimum funkcji:

1. Pochodna przyjmuje wartość 0 dla minimum funkcji f(x) (także dla maximów i punktów siodłowych, ale póki co nie musimy się tym martwić)
2. Przyjmuje wartości dodatnie w przedziałach gdy funkcja jest rosnąca, ujemne w przedziałach gdzie jest malejąca.

Pochodna

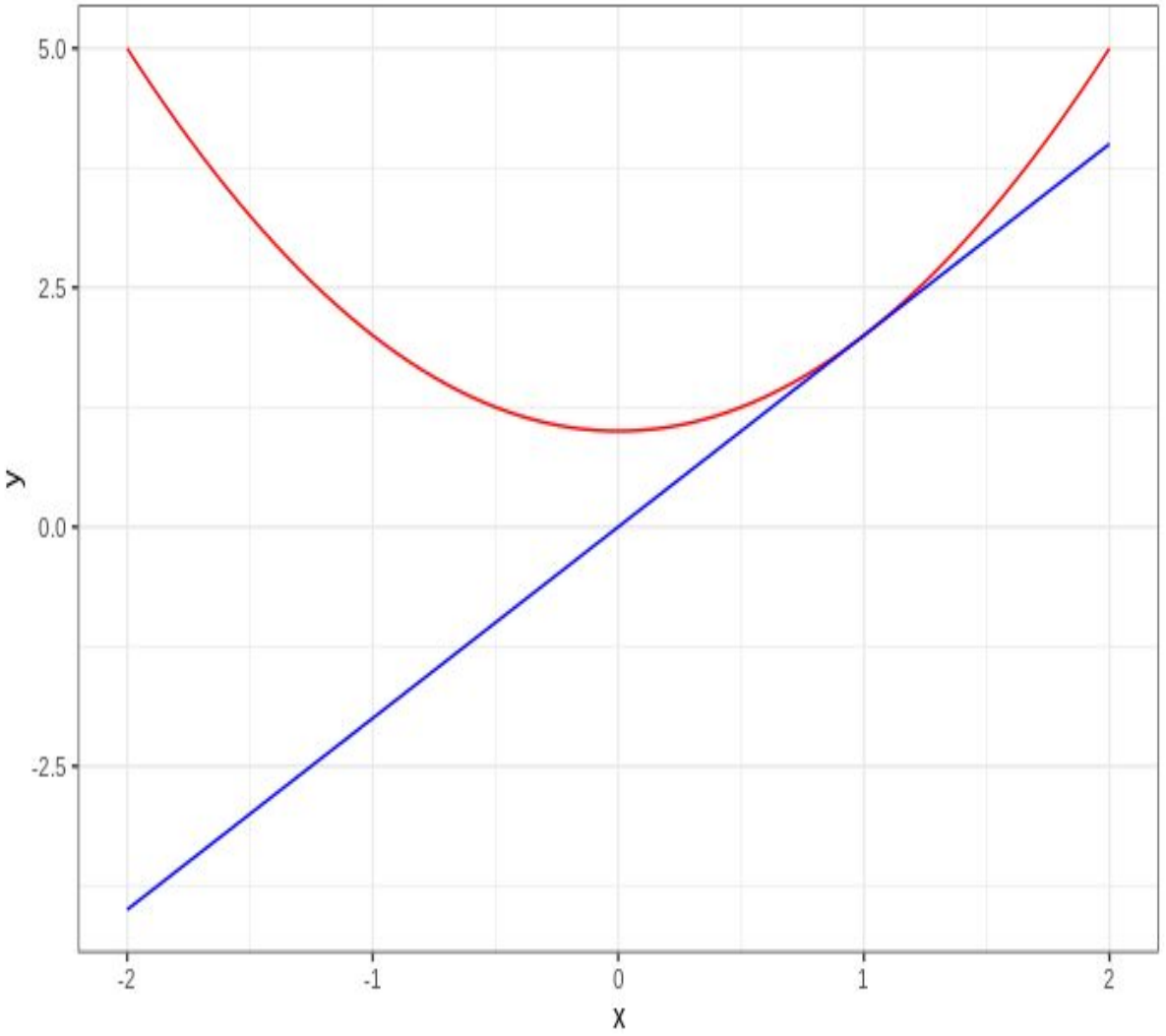


Wracając do przykładu naszej funkcji $f(x) = x^2+1$. Zakładając, że nie jesteśmy w stanie znaleźć jej minimum bezpośrednio możemy jej pochodną przyrównać do 0 i rozwiązać równanie: $2x = 0$.

W tym wypadku dostajemy tę samą odpowiedź co poprzednio: $x = 0$.

Co jednak zrobić, jeśli znalezienie rozwiązania $f'(x) = 0$ nie jest oczywiste ?

Gradient Descent

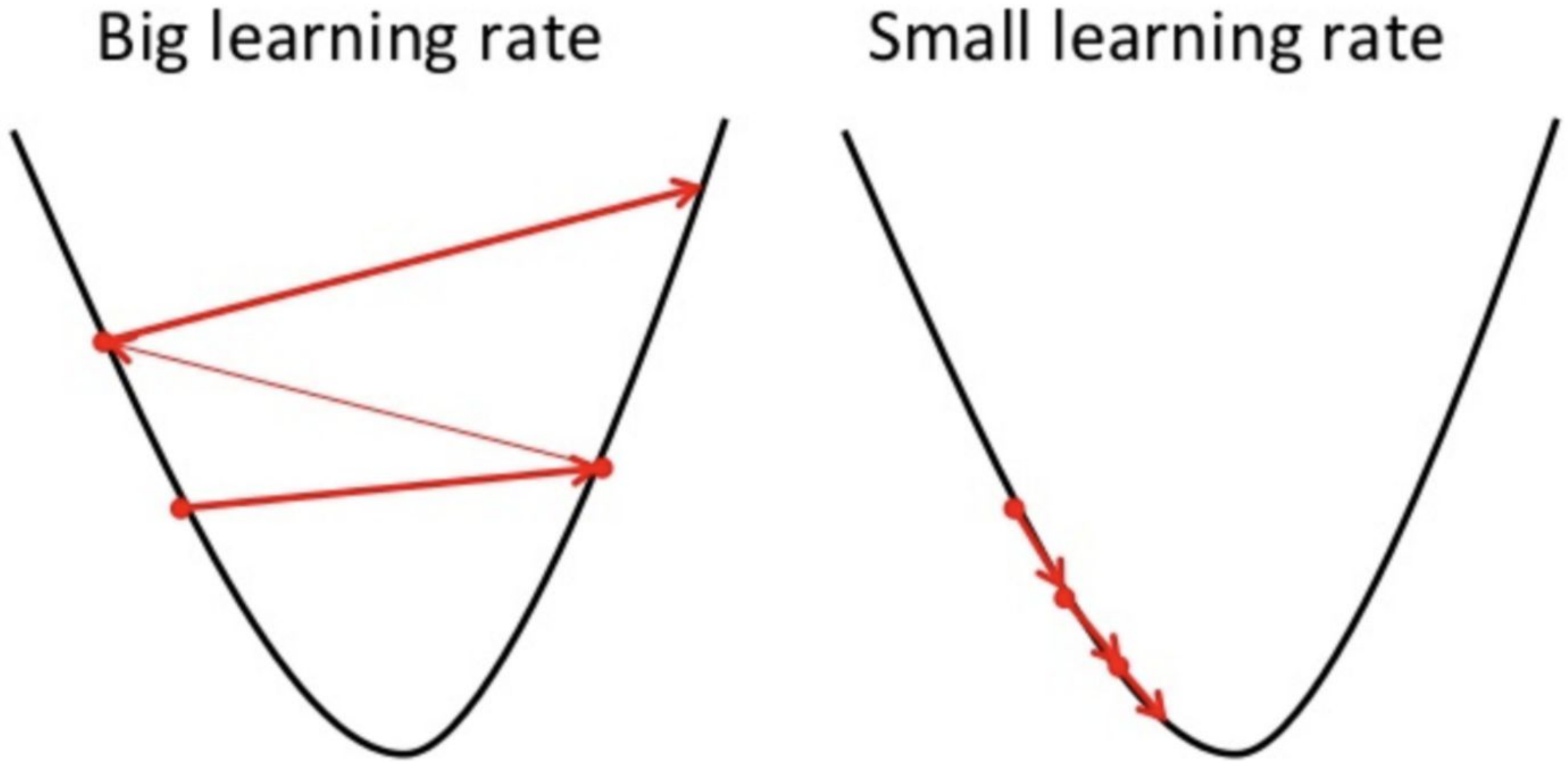


W tym wypadku możemy skorzystać z algorytmu iteracyjnego **Gradient Descent** (metoda spadku wzdłuż gradientu):

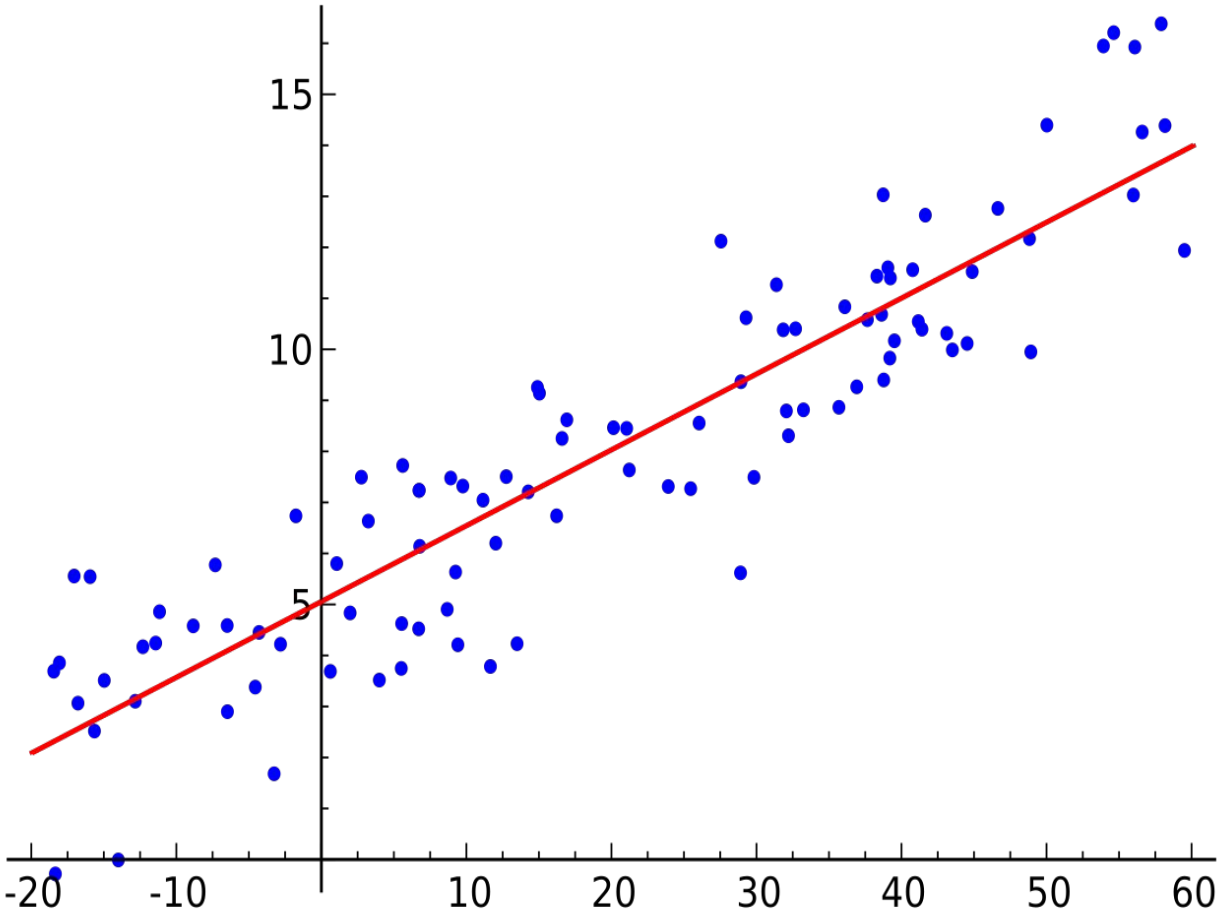
1. Wybieramy dowolny punkt $\mathbf{x_0 = x_{old}}$
2. W pętli aktualizujemy nasz parametr:
   $\mathbf{x_{new} = x_{old} - LR * f'(x_{old})}$

gdzie **LR** jest hipermarameterm zwanym **learning rate**. Ilość aktualizacji w 2. nazywamy liczbą **epok**.

Learning rate



Gradient Descent dla regresji liniowej



W przypadku **regresji liniowej** naszą funkcją straty jest **błąd średniokwadratowy**:

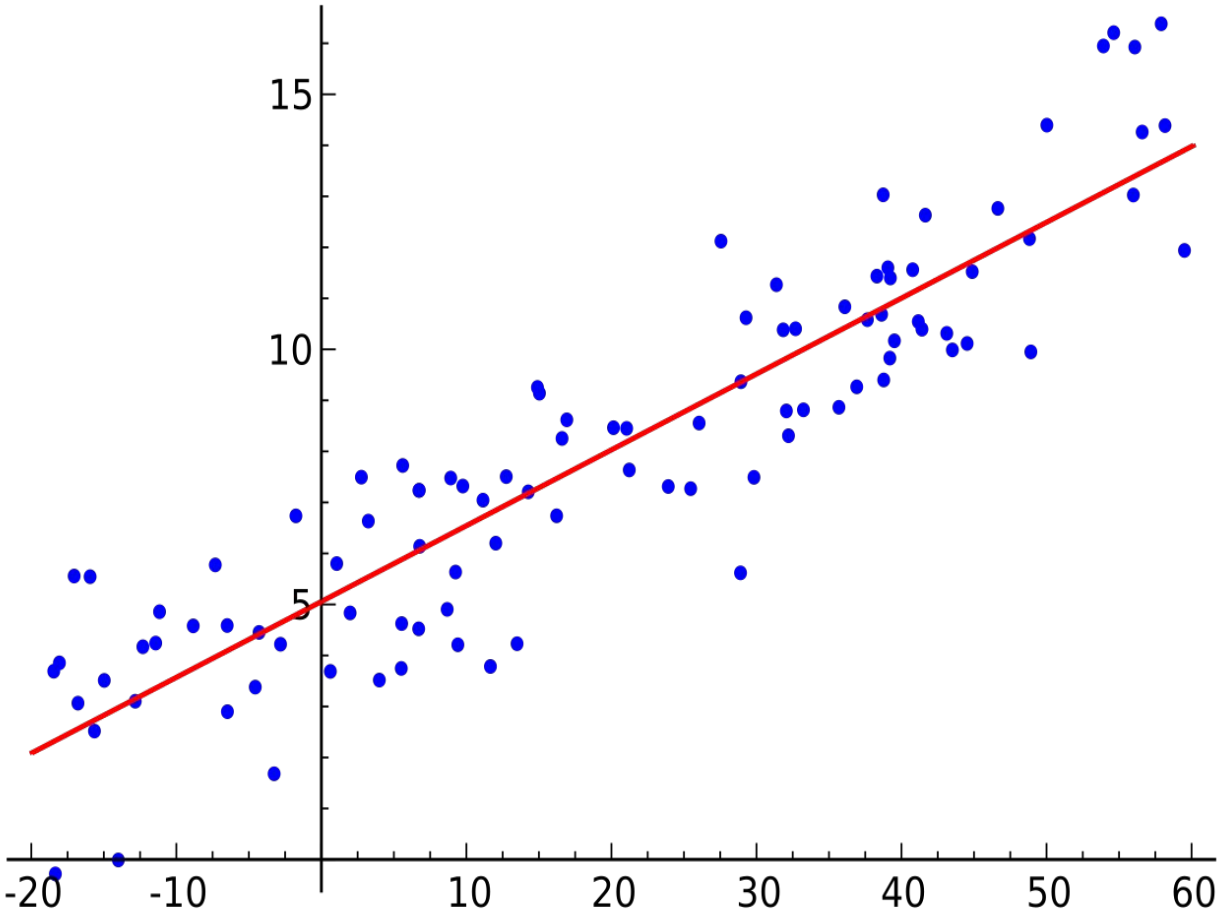
$$\mathbf{MSE\ (\beta) = 1/n * \Sigma(\beta X^T - y)^2}$$

Gradient MSE względem **β** dany jest wzorem:

$$\mathbf{MSE\_grad\ (\beta) = 1/n\ ^*2(\beta X^T - y)X}$$

Źródło: https://en.wikipedia.org/wiki/Generalized_linear_model

Stochastic Gradient Descent dla regresji liniowej

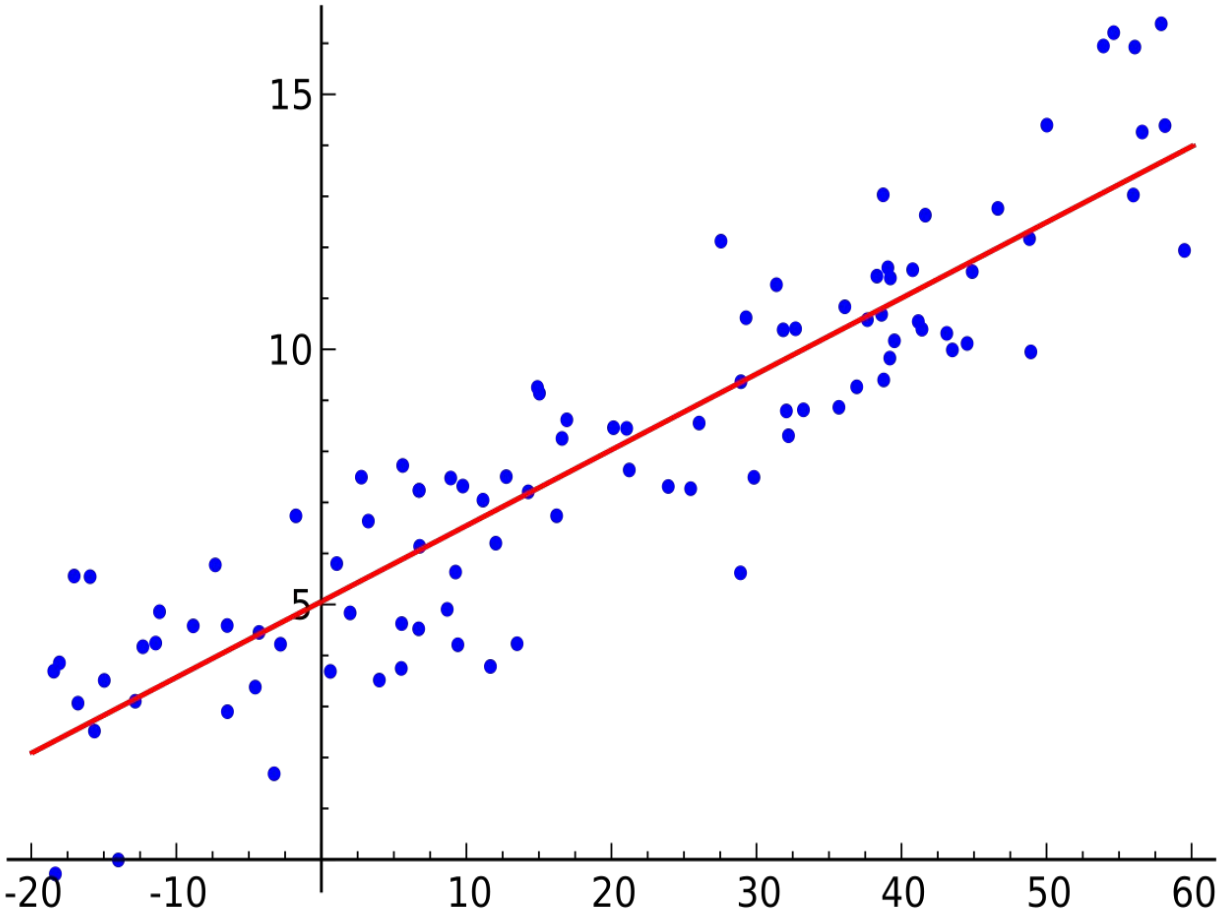


W przypadku stochastycznej odmiany algorytmu GD do aktualizacji naszych wag nie będziemy używać całej macierzy predyktorów **X**, a tylko jej podzbioru, tak zwanego **batcha**.

Załóżmy, że mamy 1 000 000 obserwacji, dzieląc nasz zbiór na batche o wielkości 100 000 (**batch size**) otrzymamy 10 batchy. Gdy wagi naszego modelu zostaną zaktualizowane na każdym z batchy powiemy, że algorytm zakończył jedną epokę.

Źródło: https://en.wikipedia.org/wiki/Generalized_linear_model

Stochastic Gradient Descent dla regresji liniowej



Zalety SGD:

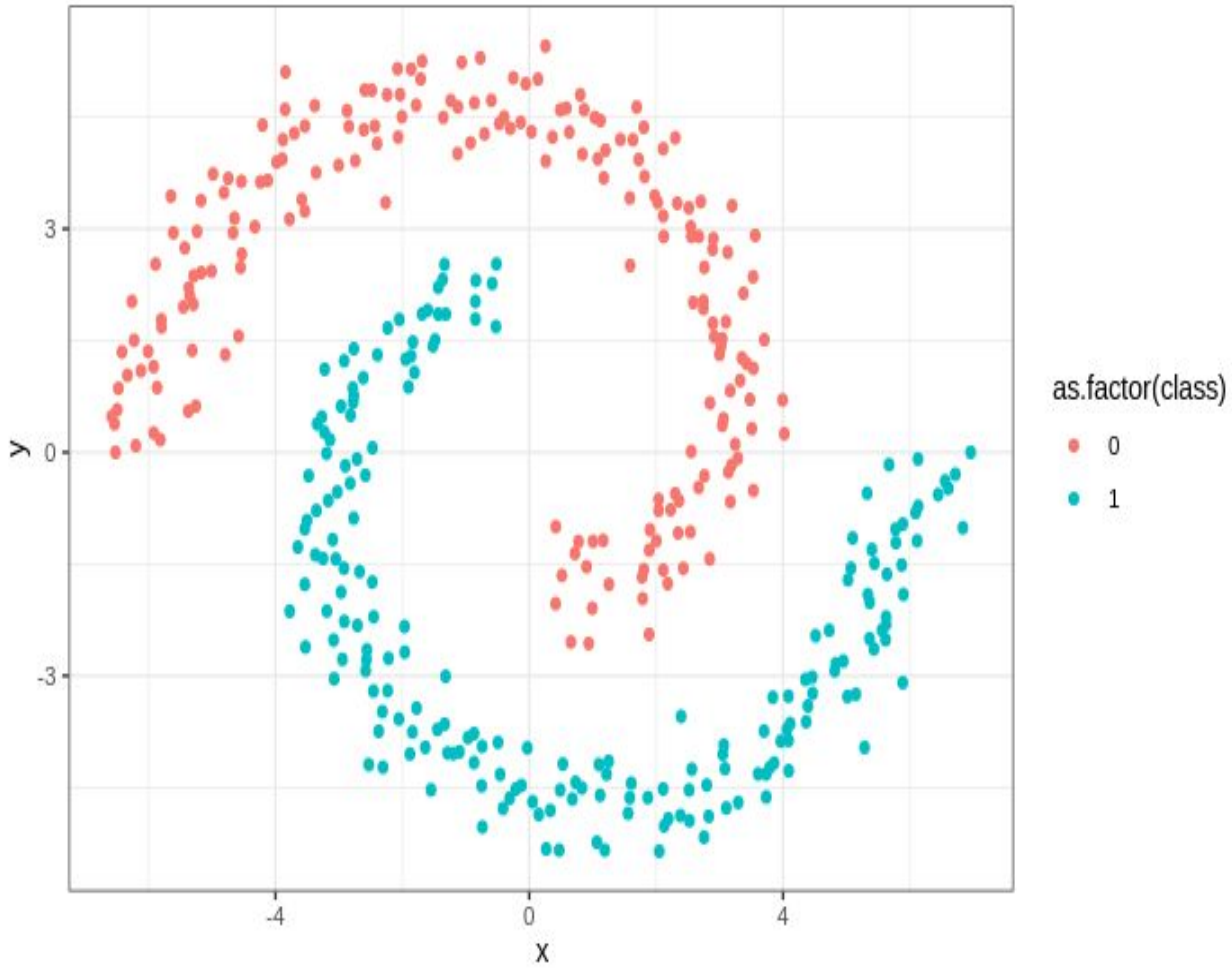
- szybsze obliczenia związane z aktualizacją wag
- potrzeba mniejszej mocy obliczeniowej
- SGD potrafi "wyjść" z **minimum lokalnego** w przeciwieństwie do klasycznego GD

Wady SGD:

- nie podejmujemy optymalnej decyzji (wartość funkcji straty może wzrosnąć)
- potrzeba więcej iteracji (epok) na optymalizację

Źródło: https://en.wikipedia.org/wiki/Generalized_linear_model

Gradient Descent dla regresji logistycznej



W przypadku **regresji logistycznej** naszą funkcją straty jest **binarna entropia krzyżowa**:

$$BC(\beta) = -1/n * \Sigma(Y*\ln(Z) + (1-Y)*\ln(1-Z))$$

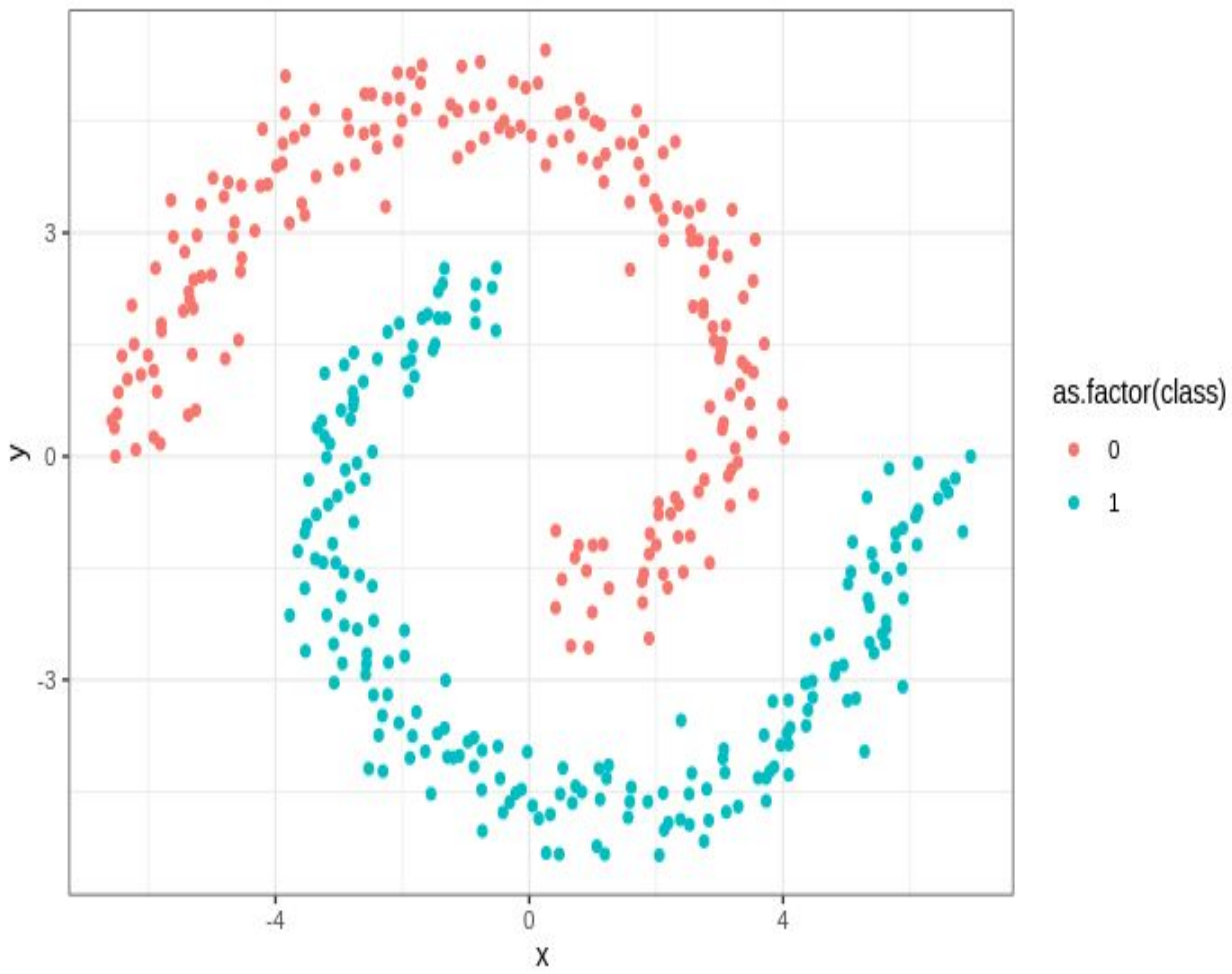
gdzie:

$$Z = \sigma(\beta X^T)$$

oraz:

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

Gradient Descent dla regresji logistycznej



Gradient BC względem **β** dany jest wzorem:

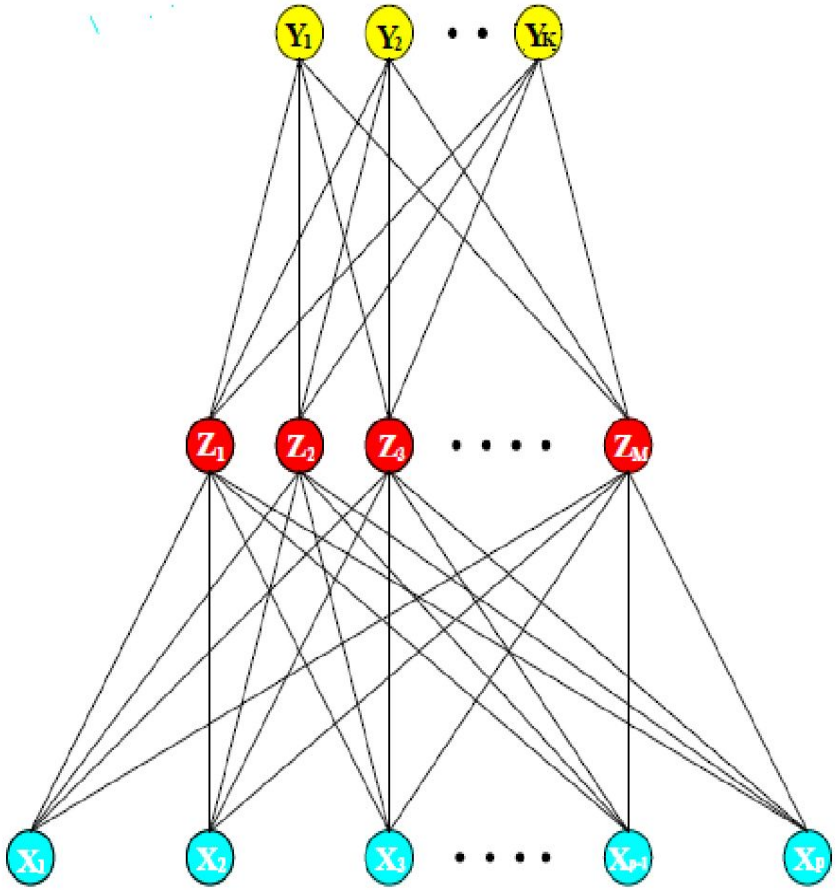
$BC\_grad(\beta) = \partial BC/\partial Z * \partial Z/\partial \sigma * \partial \sigma/\partial \beta$

gdzie:

$\partial BC/\partial Z = 1/n * (-Y/Z - (1-Y)/(Z-1))$
$Z = \sigma(\beta X^T)$
$\partial Z/\partial \sigma = \sigma'(\beta X^T)$
$\sigma'(x) = \sigma(x)(1 - \sigma(x))$
$\sigma(x) = 1/(1 + e^{-x})$
$\partial \sigma/\partial \beta = X$

Stosujemy tu tak zwaną **zasadę łańcuchową dla pochodnych**.

Gradient Descent dla jednowarstwowego perceptronu



Zanim przejdziemy do funkcji straty zastanówmy się jak w danym modelu obliczane są predykcje (**forward propagation**):

W pierwszej kolejności obliczamy wartości neuronów w warstwie ukrytej:

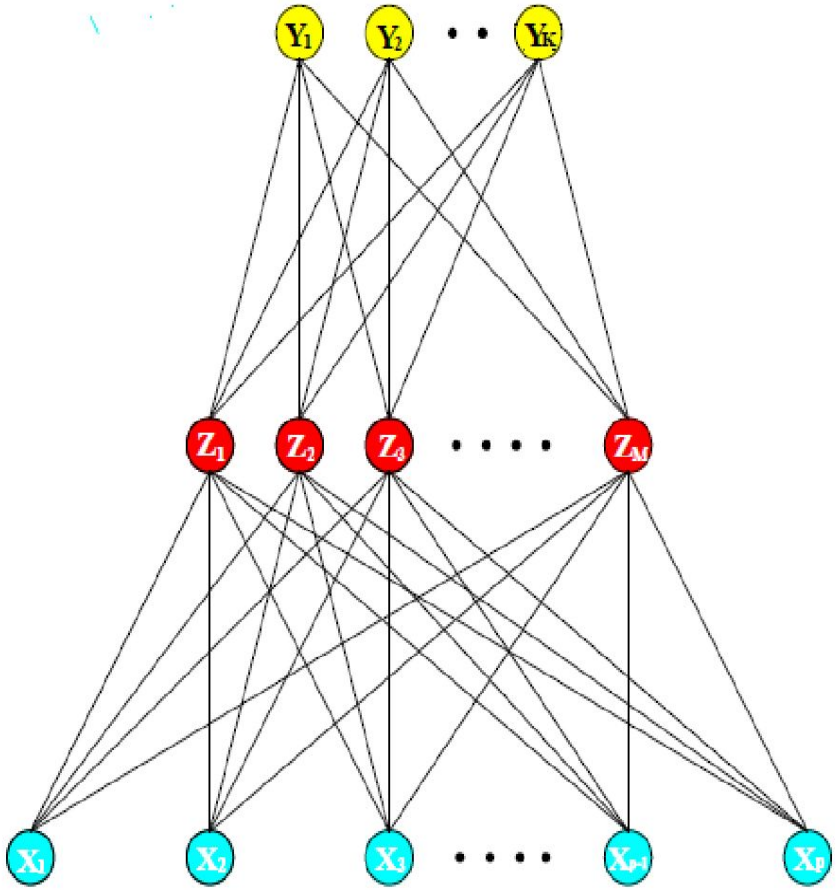
$Z_1 = XW_1$ - kombinacja liniowa
$h = \sigma(Z_1)$ - funkcja aktywacji

Następnie za pomocą tych wartości tworzymy predykcje:

$$Z_2 = hW_2$$
$$y_{pred} = \sigma(Z_2)$$

Źródło: Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. The elements of statistical learning. Vol. 1. New York: Springer series in statistics, 2001.

Gradient Descent dla jednowarstwowego perceptronu



Za funkcję straty przyjmiemy błąd kwadratowy (dla uproszczenia obliczeń!). Tym razem musimy aktualizować wagi $\mathbf{W_1}$ oraz $\mathbf{W_2}$:

$RSS = \Sigma(y_{pred} - y)\hat{}2$

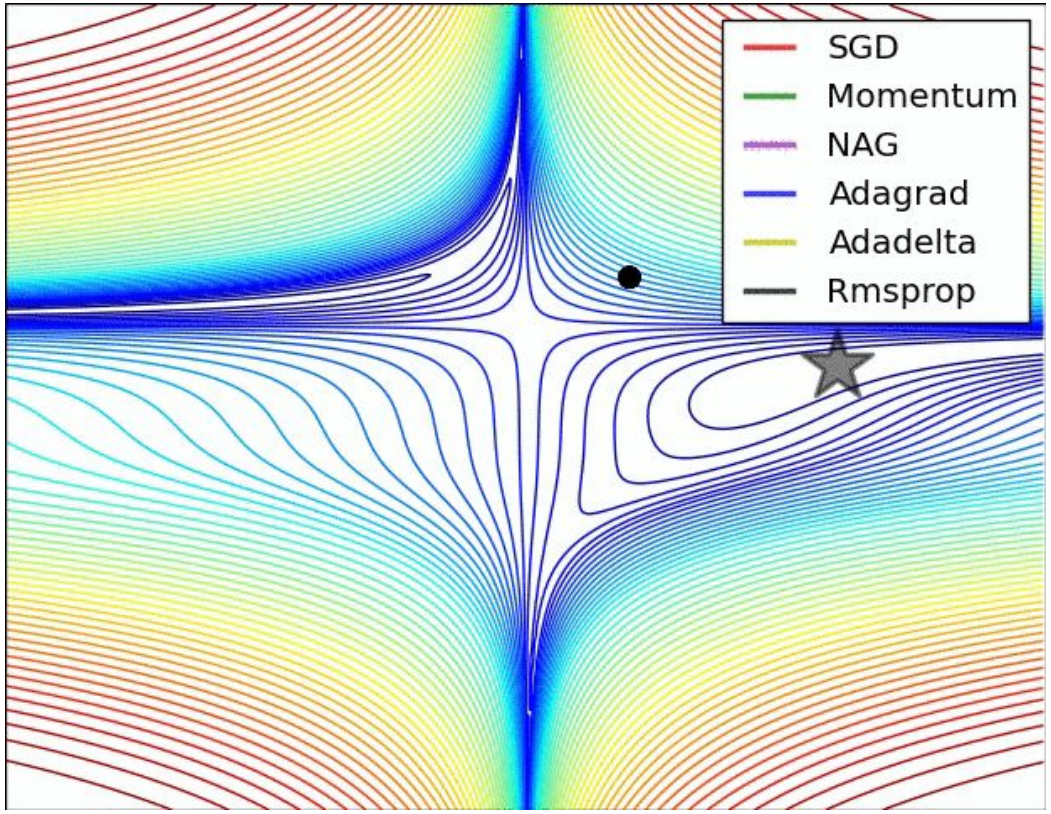
$dW_2 = 2\Sigma(y_{pred} - y)h$ - w rzeczywistości powinniśmy uwzględnić funkcję sigmoid, ale dla uproszczenia obliczeń pomijamy

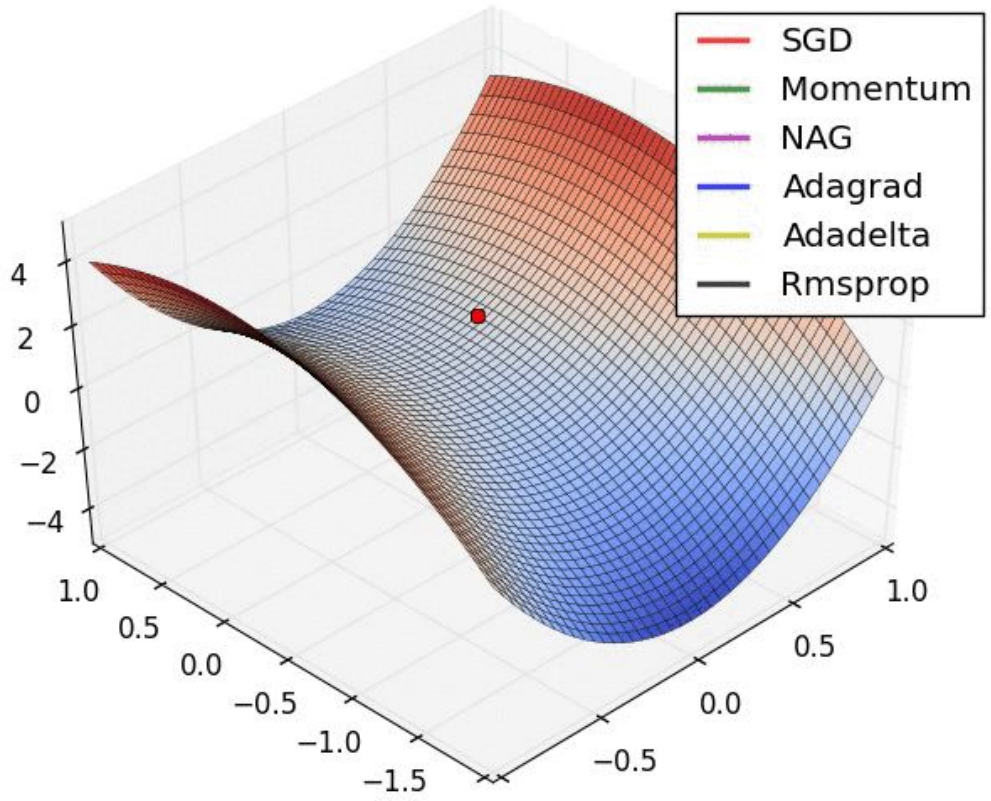
$dW_1 = X^T * h * (1-h) * dh$
$dh = (y_{pred} - y)W_2$ - w rzeczywistości powinniśmy uwzględnić funkcję sigmoid, ale dla uproszczenia obliczeń pomijamy

Źródło: Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. The elements of statistical learning. Vol. 1. New York: Springer series in statistics, 2001.

Inne optymalizatory

W Keras, poza SGD, mamy do wyboru kilka innych optymalizatorów.





Source: http://ruder.io/optimizing-gradient-descent/

Inne optymalizatory

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

Momentum:
w metodzie tej update parametrów przeprowadzany jest nie tylko za pomocą gradientu, ale także przy pomocy wartości tego parametru w poprzednim kroku.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

AdaDelta:
w tej metodzie learning rate jest automatycznie dostosowywany (zmniejszamy lub zwiększamy) w zależności od wartości gradientów w poprzednich krokach

Pozostałe metody jak **AdaGrad**, **AdaMax**, **RMSProp** łączą wymienione powyżej metody tworząc jeszcze dokładniejsze techniki.

Source: http://ruder.io/optimizing-gradient-descent/

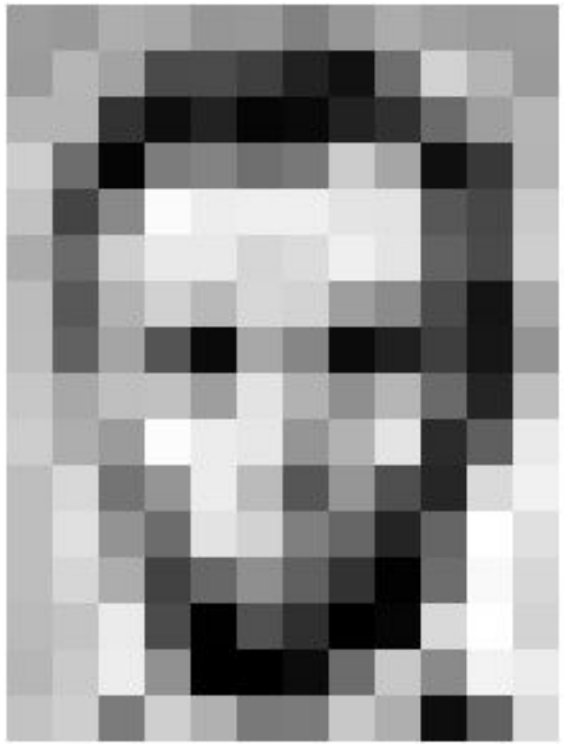
Część IV - Sieci konwolucyjne

O czym będziemy rozmawiać:

- Reprezentacja obrazu
- Idea sieci konwolucyjnych
- Warstwa konwolucyjna i poolingu
- Batch Normalization
- Budowa sieci konwolucyjnej w Keras
- Zaawansowane zastosowania sieci konwolucyjnych:
  - segmentacja obrazu
  - detekcja obrazu

Reprezentacja obrazu

Dla komputera obraz reprezentowany jest jako **3-wymiarowa tablica** (tensor). Pierwsze dwa wymiary odpow
za **wysokość** i **szerokość** obrazu (w pikselach), trzeci wymiar reprezentuje liczbą **kanałów**.
Dla przykładu obraz w odcieniach szarości (**grayscale**) reprezentowany jest jako macierz, w której każdy pikse
przyjmuje wartości od 0 (czerń) do 255 (biel). Mamy tu tylko 1 kanał koloru.



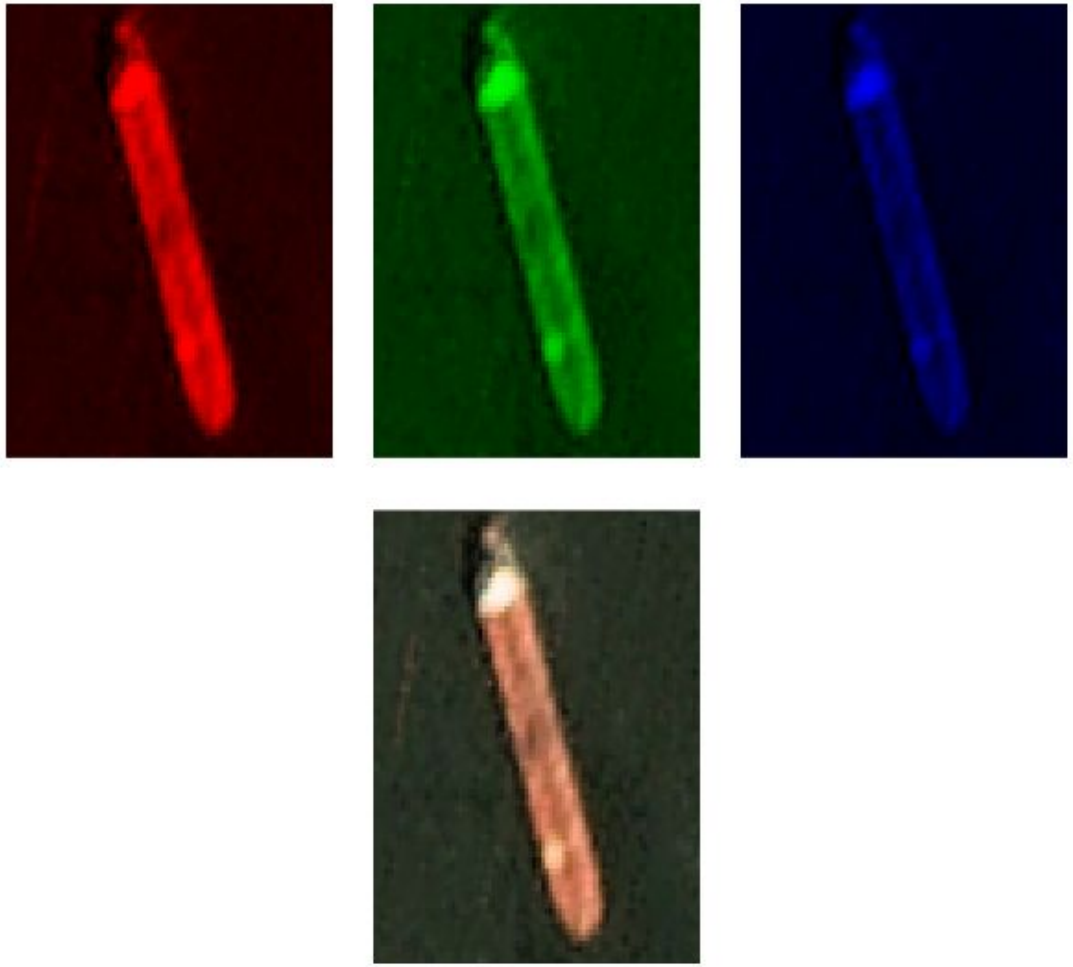
157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Reprezentacja obrazu

Obrazu kolorowe możemy reprezentować za pomocą 3 macierzy. Pierwsza macierz odpowiada za kolor **czerwony**, druga za **zielony**, a trzecia za **niebieski**. Jest to tak zwany format **RGB**. Wartości pikseli w każdej z macierzy przyjmują wartości od 0 do 255.

Poza **RGB** istnieją także inne reprezentacje (color spaces) obrazów kolorowych: CIE, HSV, HSL, ...



Klasyfikacja obrazu przy pomocy MLP

Do tej pory pracowaliśmy nad klasyfikacją dla zbioru Fashion-MNIST przy użyciu MLP.

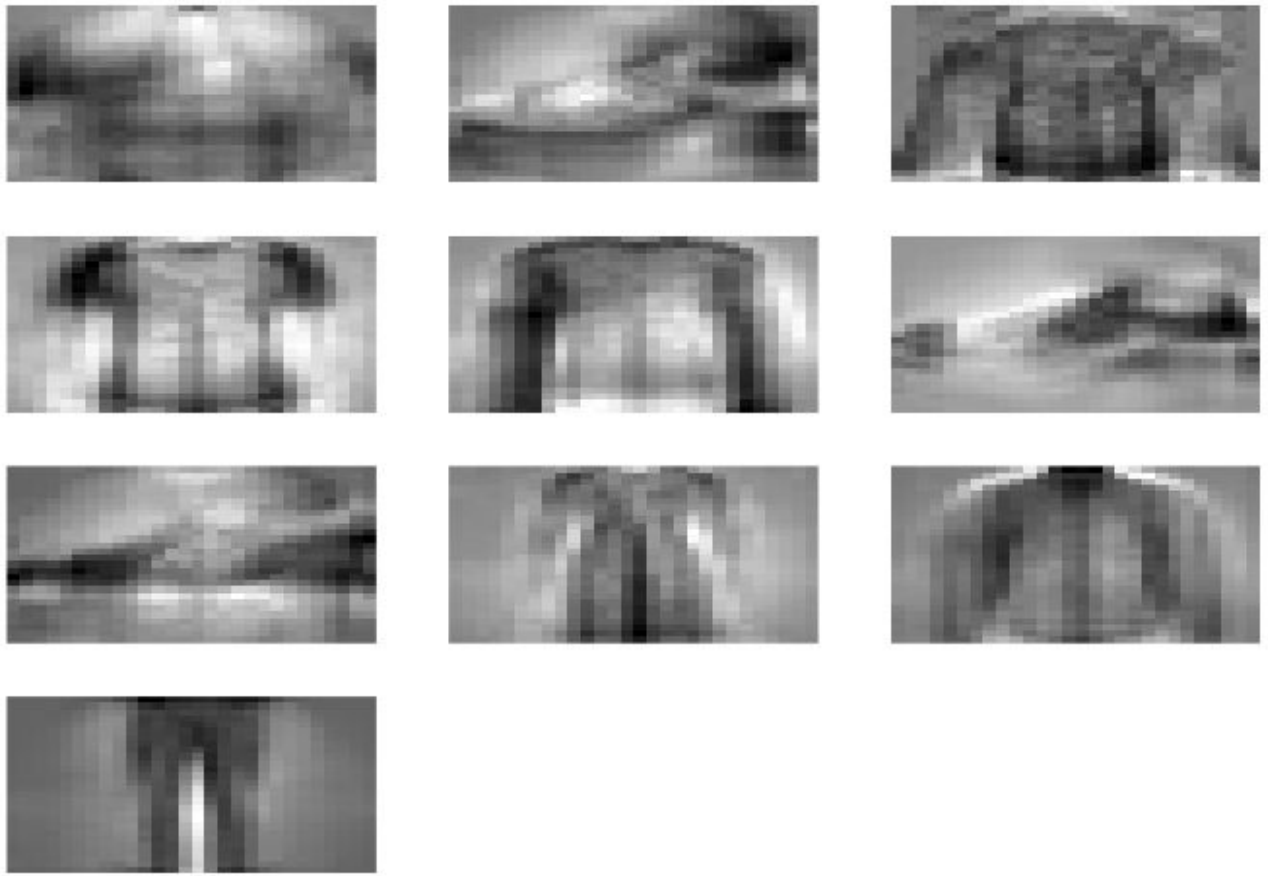
Każdy piksel traktowany był jako **osobna zmienna**. W warstwie ukrytej neurony tworzyły kombinację liniową między wagami a wartościami naszych pikseli. Następnie na obliczoną sumę nakładaliśmy funkcję aktywacji.

Klasyfikacja obrazu przy pomocy MLP

Oznacza to, że nasze “nowe zmienne” (neurony) tworzymy uwzględniając **wzorce globalne** (global patterns).

Takie rozwiązanie zadziała w przypadku Fashion-MNIST, jednakże dla zdjęć, które bardziej odzwierciedlają rzeczywistość już nie.

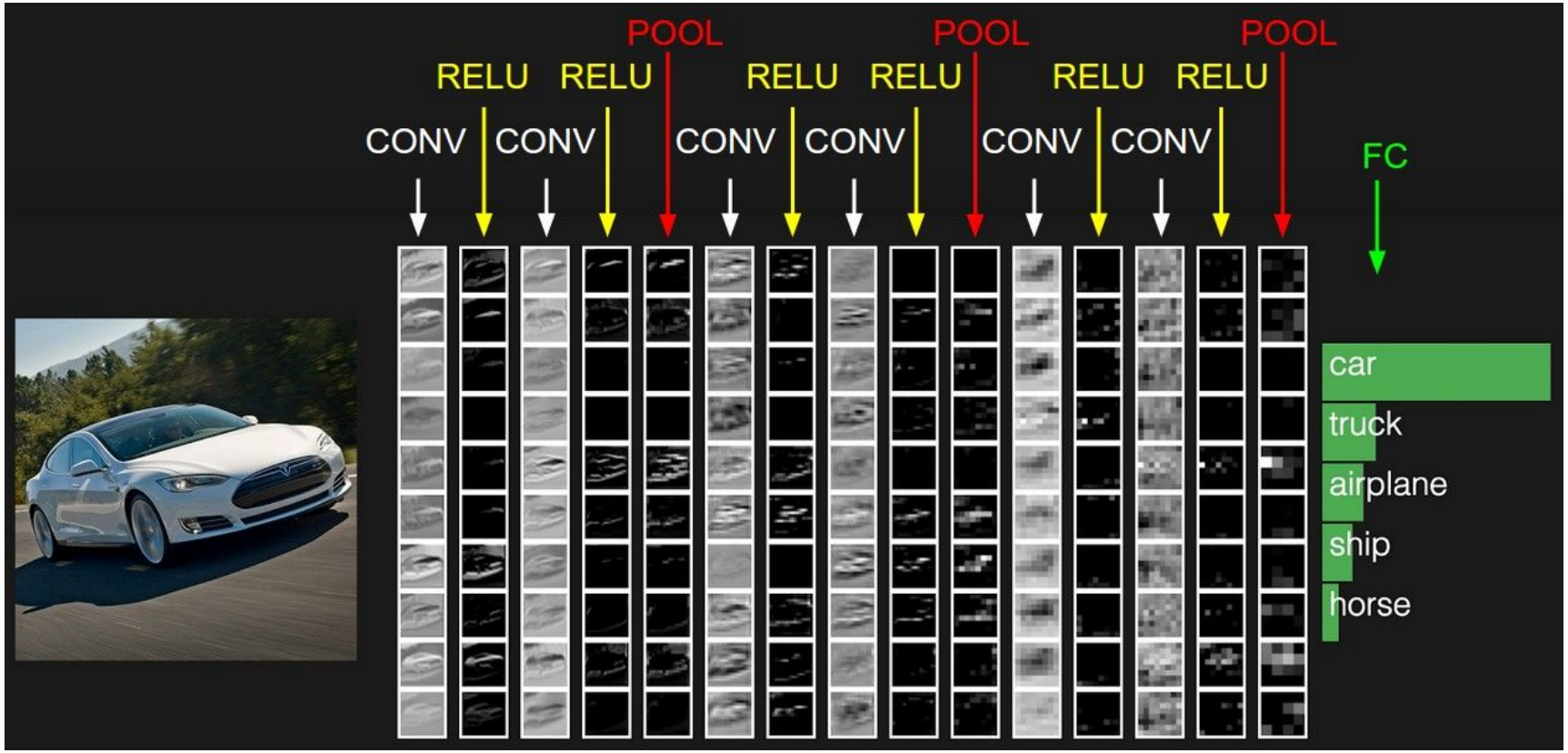
Musimy nauczyć się tworzyć zmienne uwzględniające **wzorce lokalne** (local patterns).



Czym jest sieć konwolucyjna ?

Konwolucyjne sieci neuronowe (CNN) lub w skrócie **ConvNets** to klasa głębokich sztucznych sieci neuronowych zaprojektowanych do rozwiązywania problemów, takich jak rozpoznawanie obrazu / wideo / audio, wykrywanie obiektów itp.

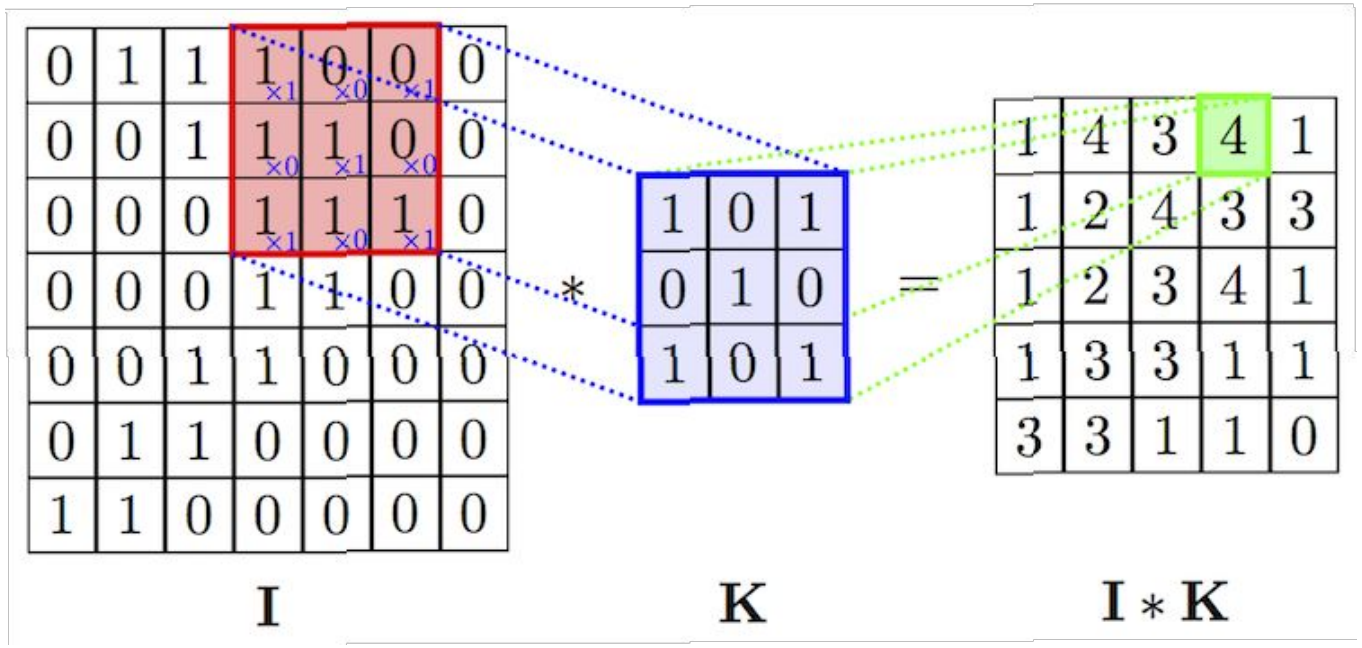
Architektura ConvNets różni się w zależności od problemu, ale istnieje kilka podstawowych wspólnych części.



Warstwa konwolucyjna

Pierwszym i najważniejszym rodzajem warstw w CNN jest **warstwa konwolucyjna**. Mówiąc krótko w warstwach konwolucyjnych, bierzemy zestaw małych **filtrów** (zwanych również **kernelami**) i umieszczamy je na części naszego oryginalnego obrazu, aby uzyskać iloczyn skalarny (**kombinację liniową**) między filtrem a odpowiednią częścią obrazu. Następnie przenosimy nasz filtr do następnego miejsca i powtarzamy tę akcję. Liczba pikseli do przesunięcia filtra nazywana jest **krokiem** (stride). Po wykonaniu operacji konwolucji dla całego obrazu otrzymujemy tak zwaną **mapę aktywacji**.

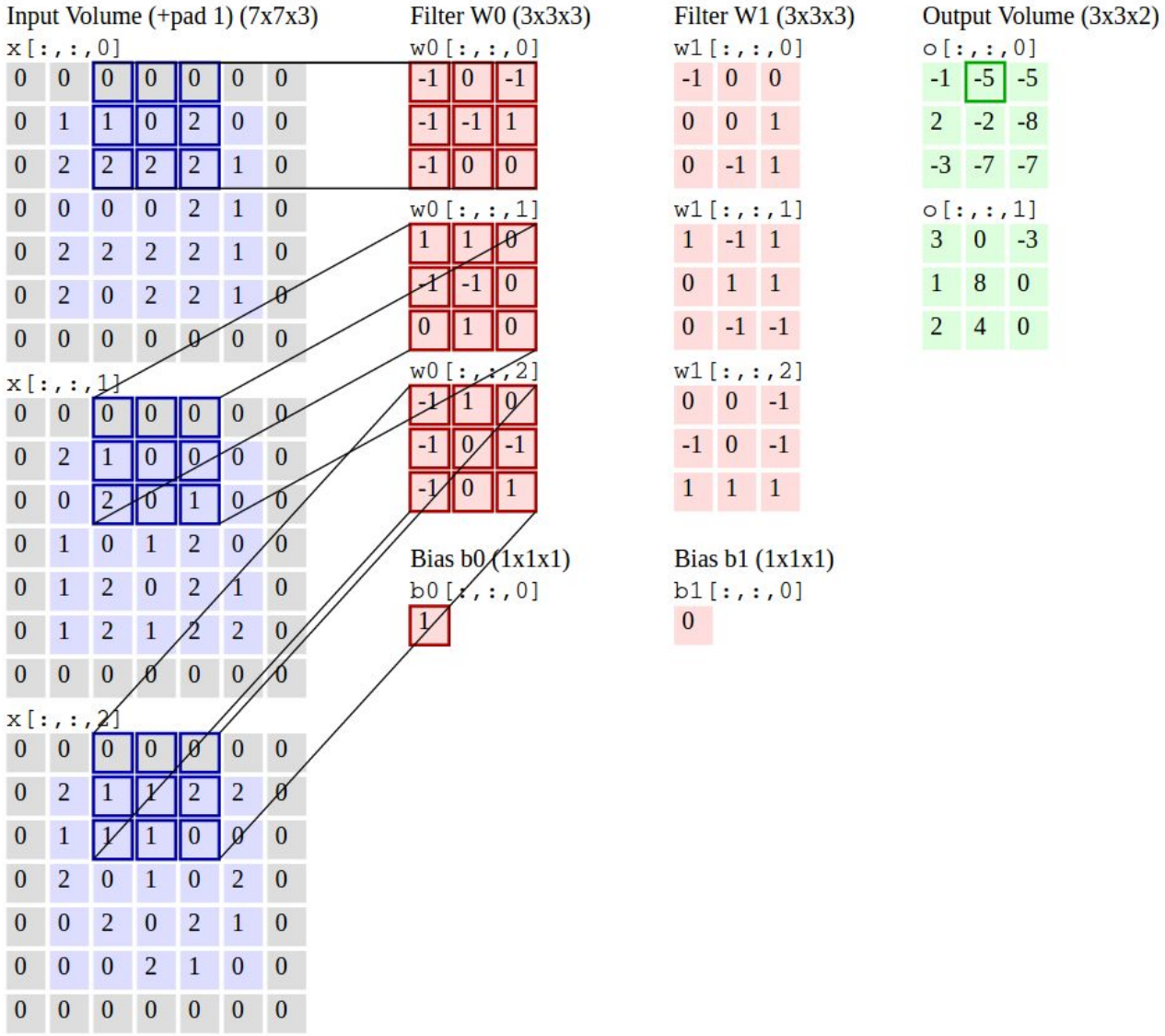
$$O = \frac{(W - K + 2P)}{S} + 1$$



Warstwa konwolucyjna

Jeśli obraz reprezentowany jest przez N kanałów (np. RGB) to filtry również są N-kanałowe - wyjściowa mapa aktywacji jest macierzą (ma tylko 1 kanał).

Szerokość i wysokość mapy aktywacji jest mniejsza od oryginalnego obrazu. Jeśli nie chcemy zmniejszać wysokości i szerokości zbyt szybko, możemy dodać "otoczkę" (**padding**) do naszego oryginalnego obrazu.

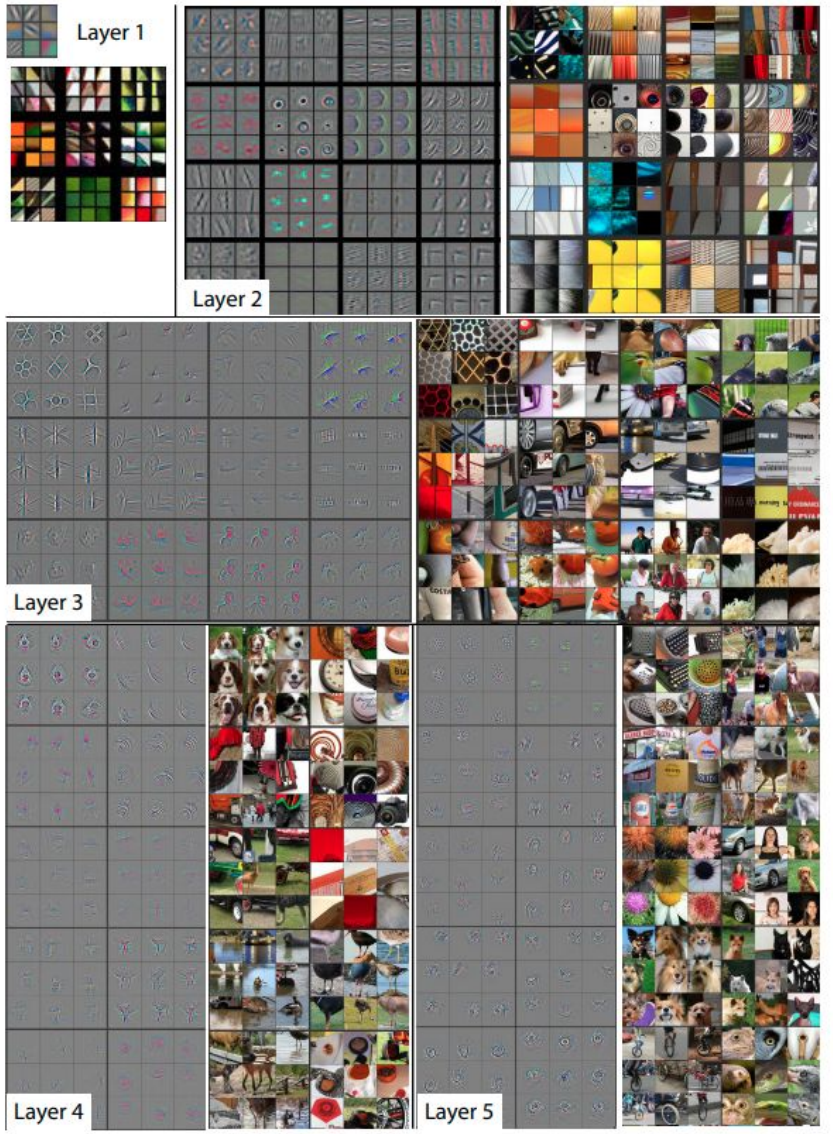


Warstwa konwolucyjna

Wagi filtrów inicjalizowane są losowo i optymalizowane w procesie SGD tak jak w przypadku MLP.

Filtry w pierwszych warstwach konwolucyjnych uczą się prostych ogólnych wzorców takich jak kreski, kropki, krzywe, ...

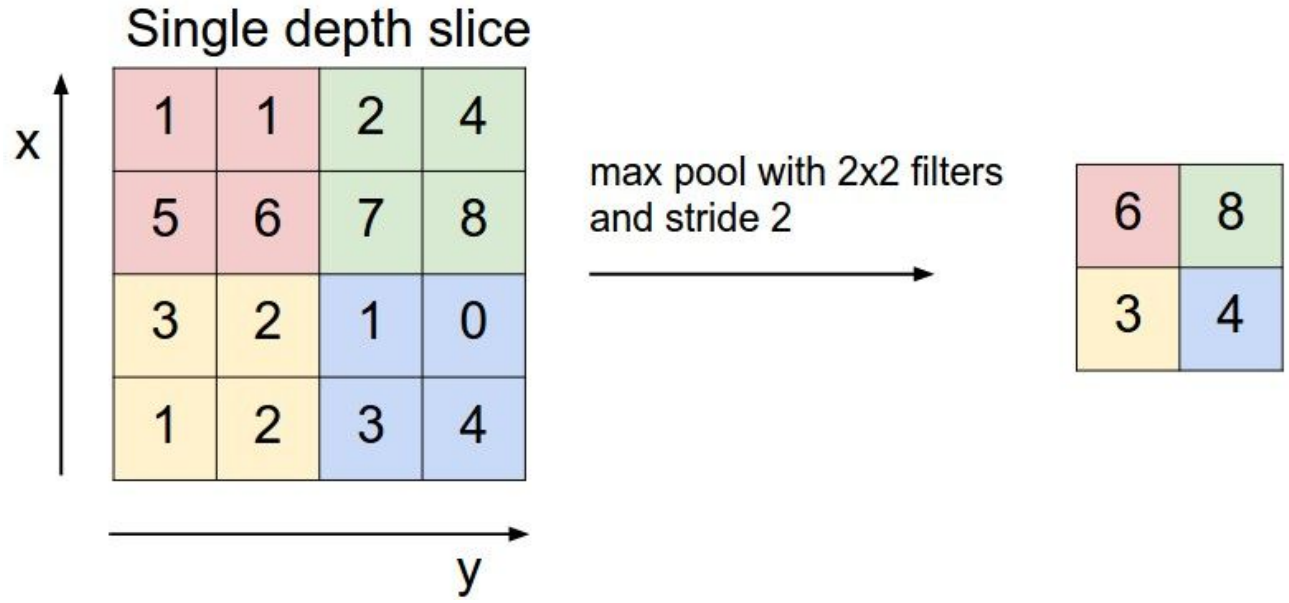
Filtry w dalszych warstwach uczą się bardziej złożonych wzorców na podstawie wzorców z poprzednich warstw.



Source: https://arxiv.org/pdf/1311.2901v3.pdf#page=4

Warstwa poolingu

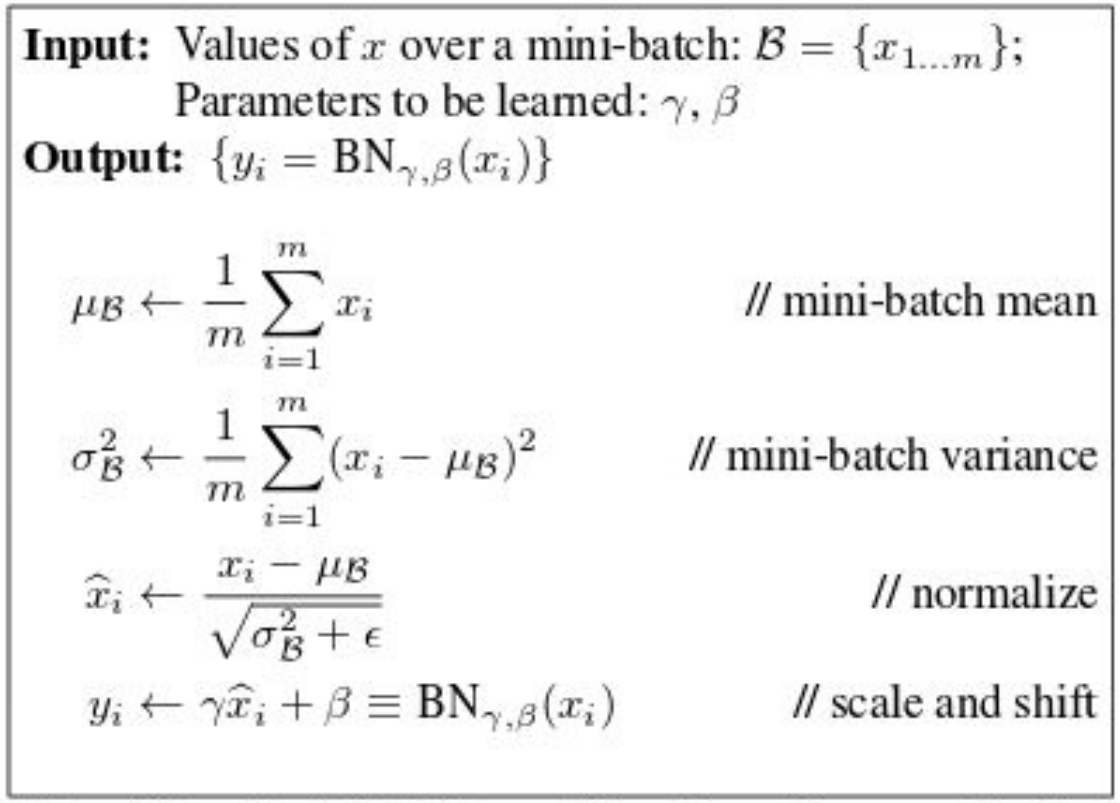
Drugi typ warstw w CNN to **warstwa poolingu**. Ta warstwa odpowiada za **generalizację** map aktywacyjnych. Istnieje kilka rodzajów poolingu, ale max pooling jest jednym z najpopularniejszych. Podobnie jak w przypadku warstw konwolucyjnych mamy pewien **kernel** i **krok**. Po umieszczeniu kernela na części obrazu pobieramy maksymalną wartość z tej części, a następnie przechodzimy do następnego miejsca o liczbę pikseli określoną krokami.



Normalizacja batch'owa

Jak wiadomo standaryzacja zmiennych jest zwykle dobrym pomysłem. W przypadku sieci neuronowych mamy możliwość normalizacji nie tylko oryginalnych predyktorów, ale także zmiennych w warstwach ukrytych.

Normalizacja batch'owa dodaje do każdej warstwy dwa trenowalne parametry - parametr wariancji (**gamma**) i parametr średniej (**beta**).



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

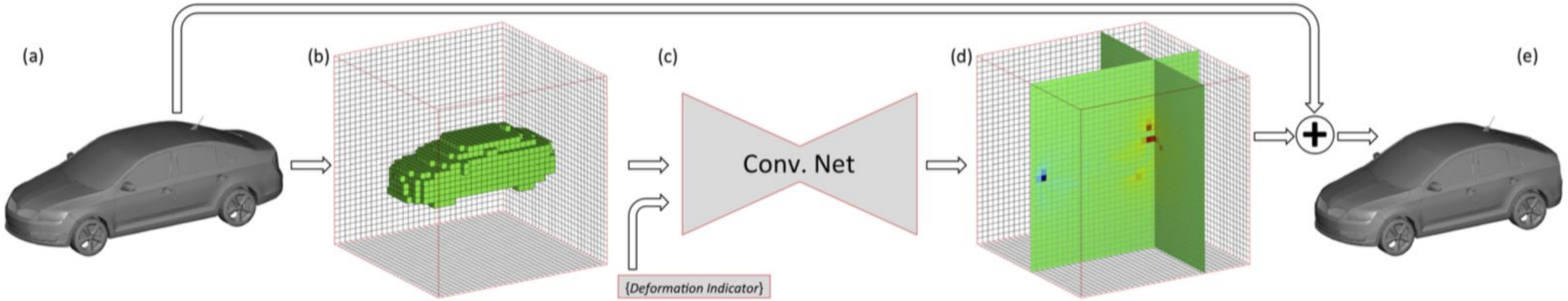
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

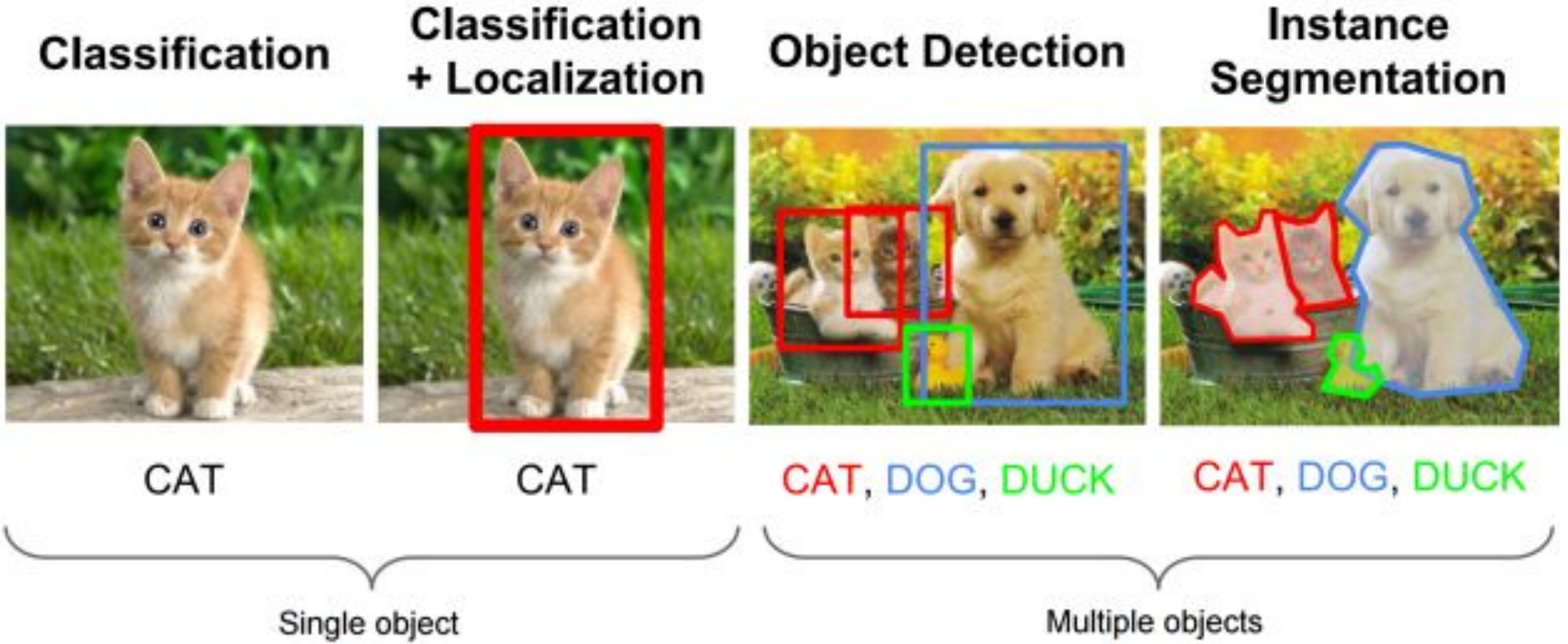
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

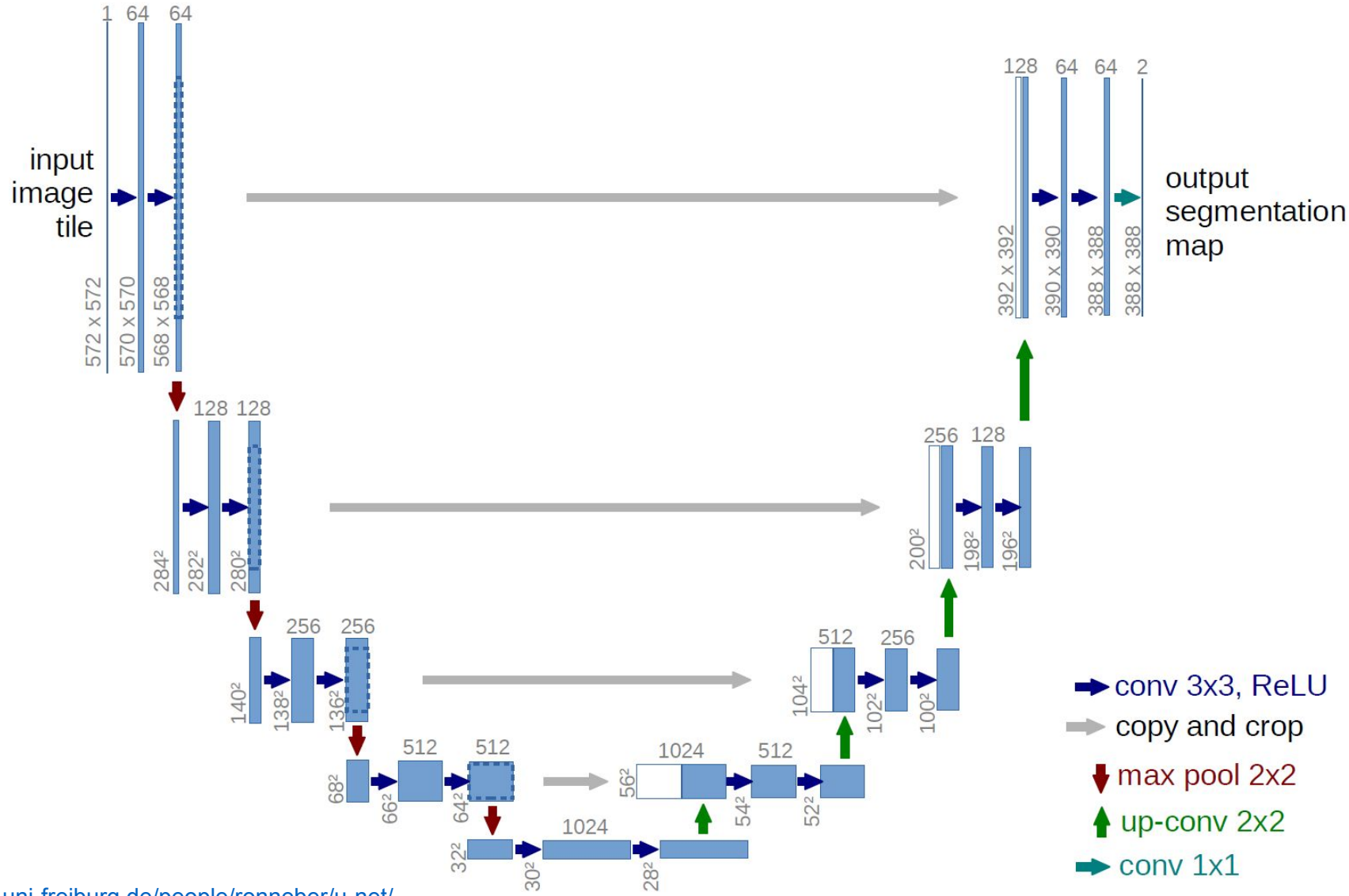
Konwolucja 1D i 3D



Zastosowania CNN



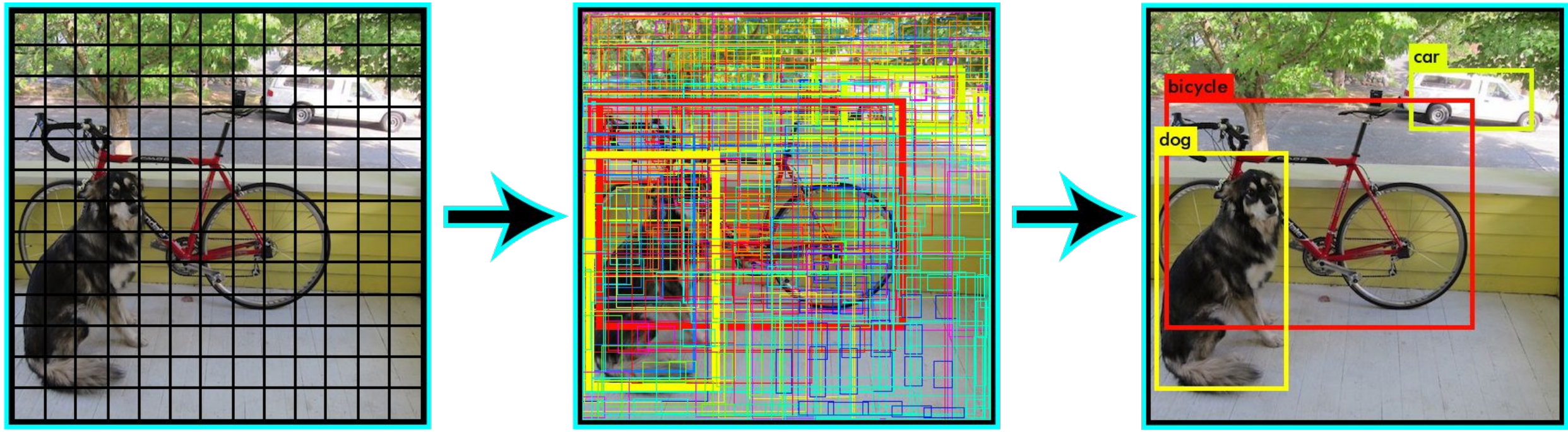
Segmentacja obrazu: U-Net



Source: https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

Detekcja obiektów: YOLO

YOLO – ‘You Only Look Once’ to prosty algorytm detekcji obrazu, który wykonuje setki predykcji bounding boxów i klas obiektu w różnych jego miejscach pozostawiając tylko te najbardziej prawdopodobne.



Source: https://pjreddie.com/darknet/yolo/

Część V - Fine-tuning i augmentacja danych

O czym będziemy rozmawiać:

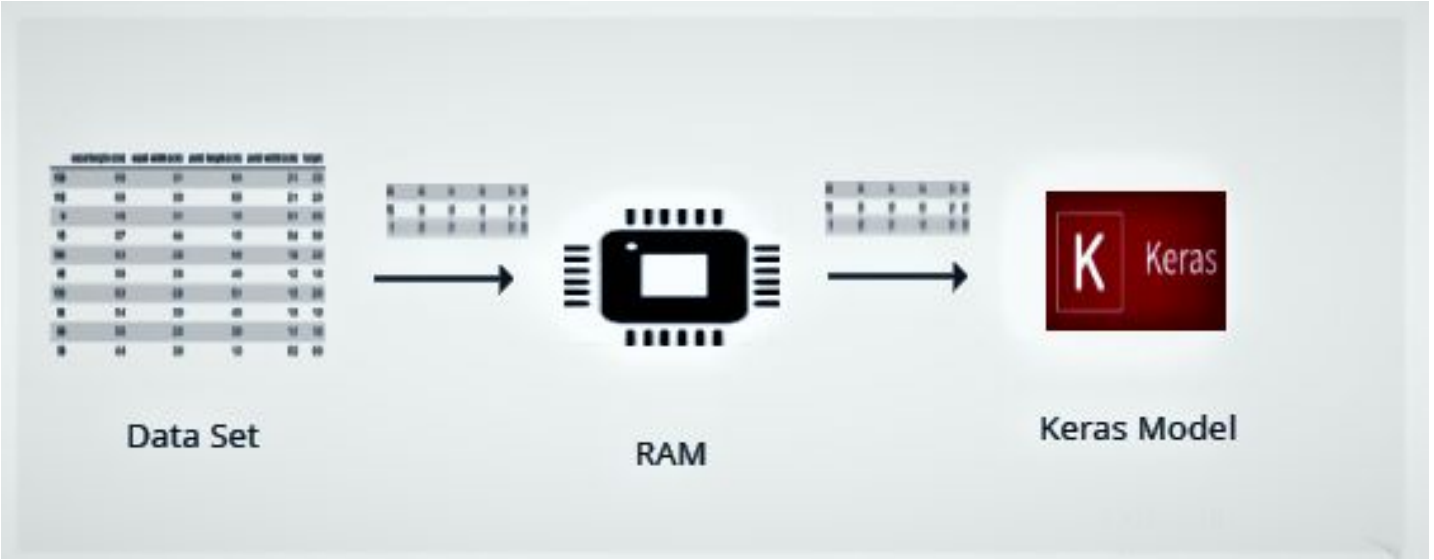
- Generatory danych
- Augmentacja danych
- Fine-tuning

Generatory danych

W przypadku dużych zbiorów danych nasze dotychczasowe podejście, polegające na wczytywaniu całego zbioru do R, a następnie budowanie modelu może być niewykonalne z powodu braku pamięci. Aby obejść ten problem możemy wczytywać dane batchowo używając **generatorów**.

Generatory pozwalają na:

- wczytanie tylko części danych (batch'a) i zaimportowanie ich do modelu w Keras
- natychmiastową normalizację danych
- natychmiastową **augmentację danych**



Source: https://mc.ai/train-keras-model-with-large-dataset-batch-training/

Augmentacja danych

Aby zwiększyć liczbę obserwacji w naszym zbiorze danych możemy dokonać tak zwanej **augmentacji danych**.

W przypadku obrazów możemy wykorzystać: zoom, rotację, zaszumianie, przesunięcia, itp.

W ogólnym przypadku możemy wykorzystać metody takie jak MCMC i bootstrap parametryczny.

Fine-tuning

Fine-tuning jest procesem polegającym na wzięciu modelu, który został już **wyuczony** dla danego zadania i **dostrojeniu** jego wag do innego podobnego zadania.

Kroki:

1. Wczytanie wyuczonej sieci neuronowej (np. przy pomocy funkcji typu **application_NET**()).
2. **Usunięcie wysokopoziomowych warstw** (outputu).
3. **Zamrożenie wag niskopoziomowych** warstw sieci.
4. **Dodaniu nowych warst wysokopoziomowych** (outputu).
5. **Ponowne wytrenowanie** modelu na nowych danych.

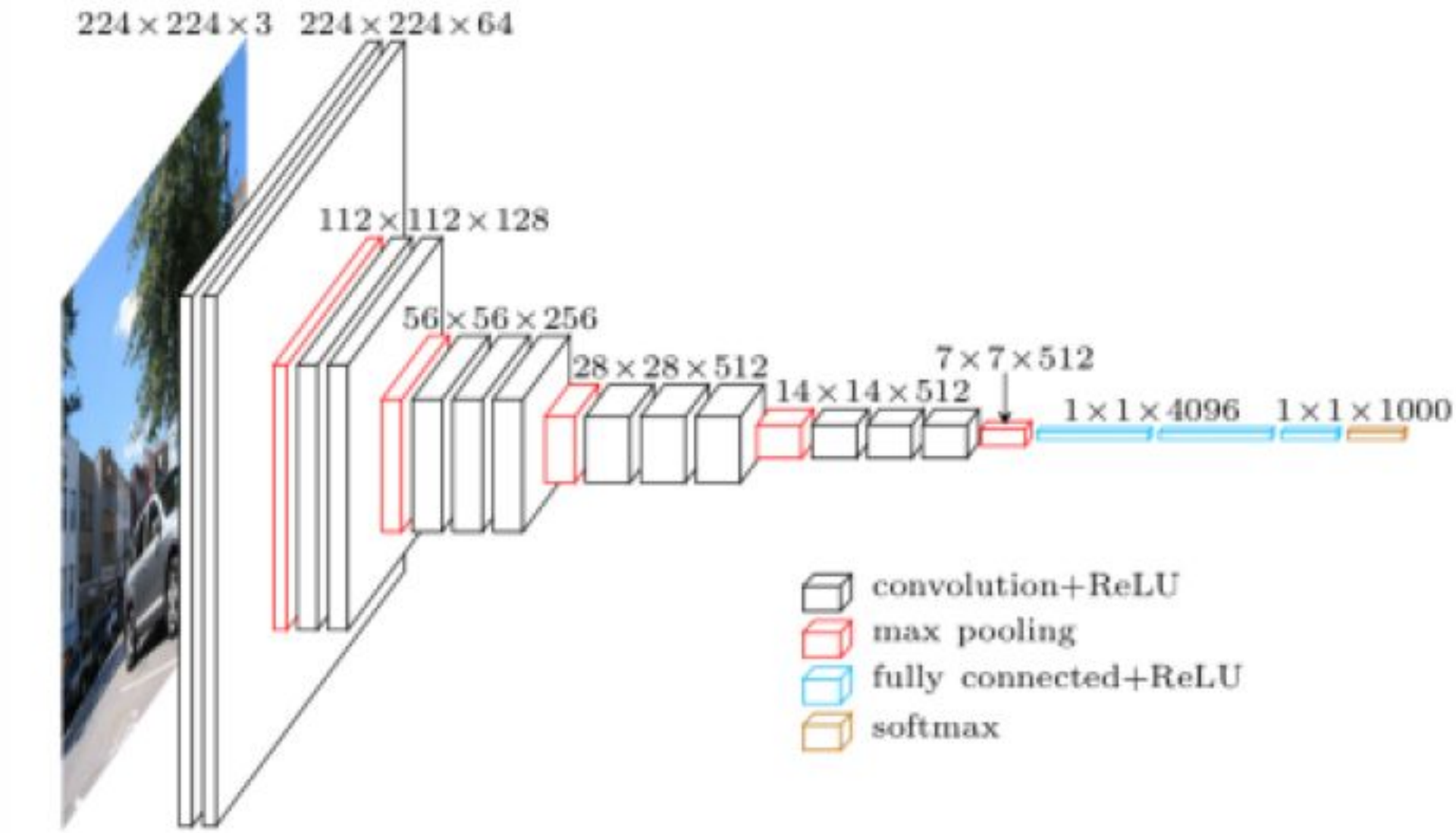
Imagenet

ImageNet Challenge

IMAGENET

- 1,000 object classes (categories).
- Images:
  - 1.2 M train
  - 100k test.

Gotowe architektury - VGG16



Część VI - Autoencodery

O czym będziemy rozmawiać:

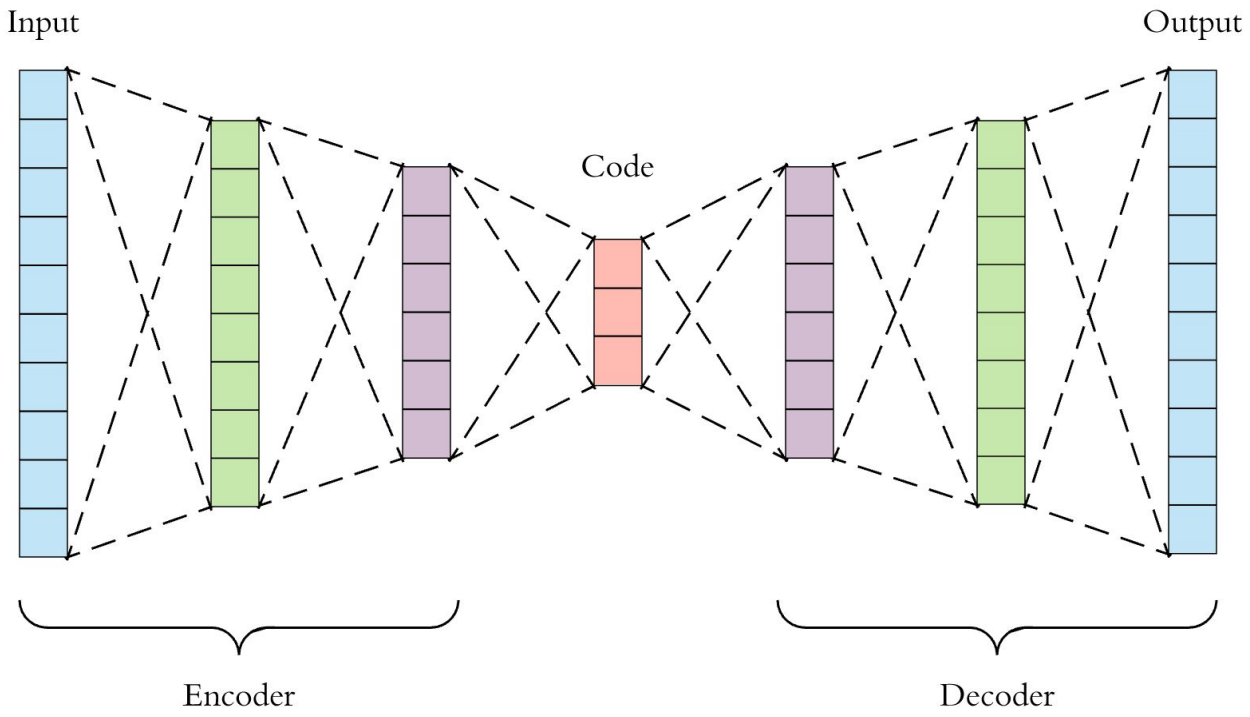
- Autoencodery i ich zastosowania
- Zadania w Keras:
  - odszumianie obrazu
  - redukcja wymiaru
  - detekcja anomalii

Autoencoder

Autoencodery to specjalna architektura sieci neuronowej składająca się z dwóch części. **Encoder**, który **koduje** oryginalną informację oraz **decoder**, który ją **rozkodowuje**. Autoencodery są częstym narzędziem do **redukcji wymiaru**.

W przypadku autoencoderów input i output sieci jest tym samym (zazwyczaj).

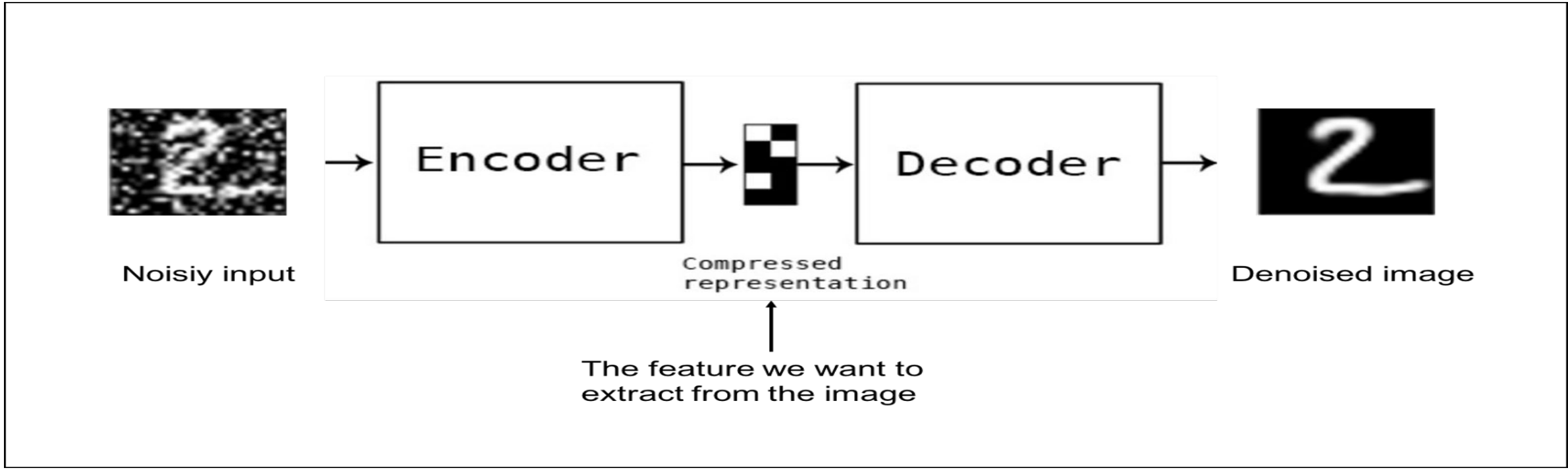
Sieć ma się nauczyć reprezentacji oryginalnych danych w przestrzeni **niskowymiarowej** (zazwyczaj), a następnie zdekodować ją z powrotem do postaci pierwotnej.



Źródło: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

Denoising Autoencoder

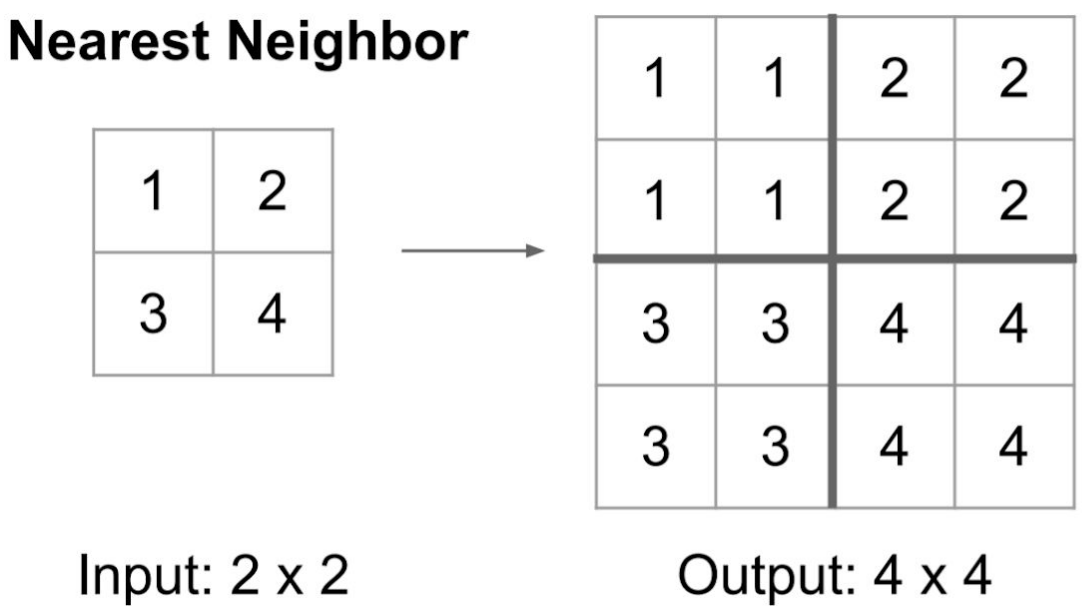
Autoencoder odszumiający (denoising autoencoder) to modyfikacja klasycznego autoencodera, w której sieć uczy się, poza zadaniem redukcji wymiaru, **usuwania szumu** (zakłóceń) z naszych danych. Pomaga to w utworzeniu poprawniejszej reprezentacji niskowymiarowej.

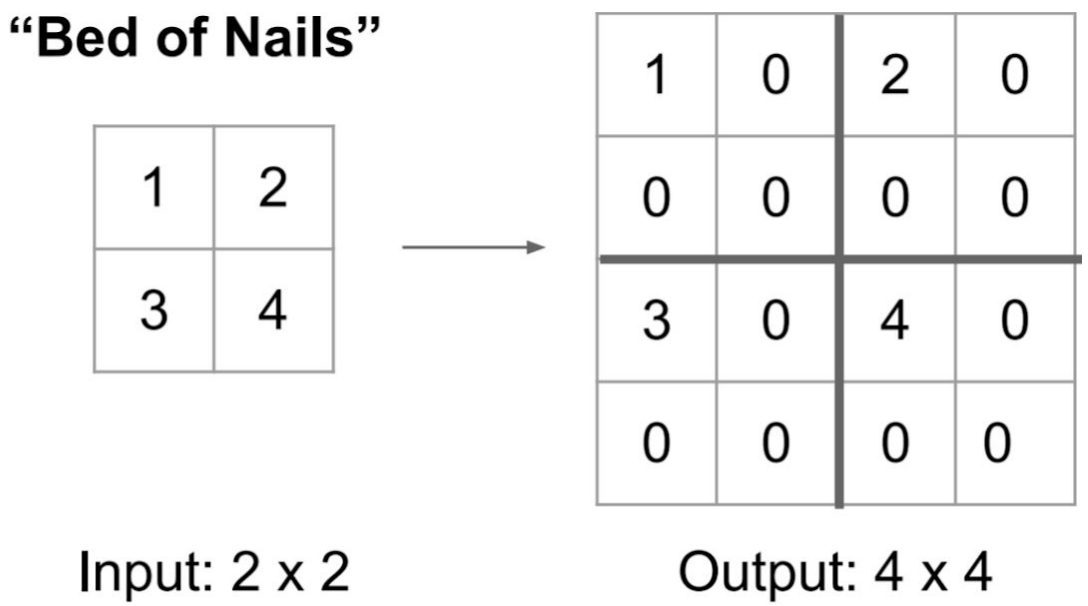


Źródło: https://iq.opengenus.org/types-of-autoencoder/

Upsampling

W sieciach autoencoderach konwolucyjnych podczas dekodowania należy warstwowo powiększyć rozmiar map aktywacji. Można tego dokonać na kilka sposobów: **upsampling**, **unpooling**, **dekonwolucja**.



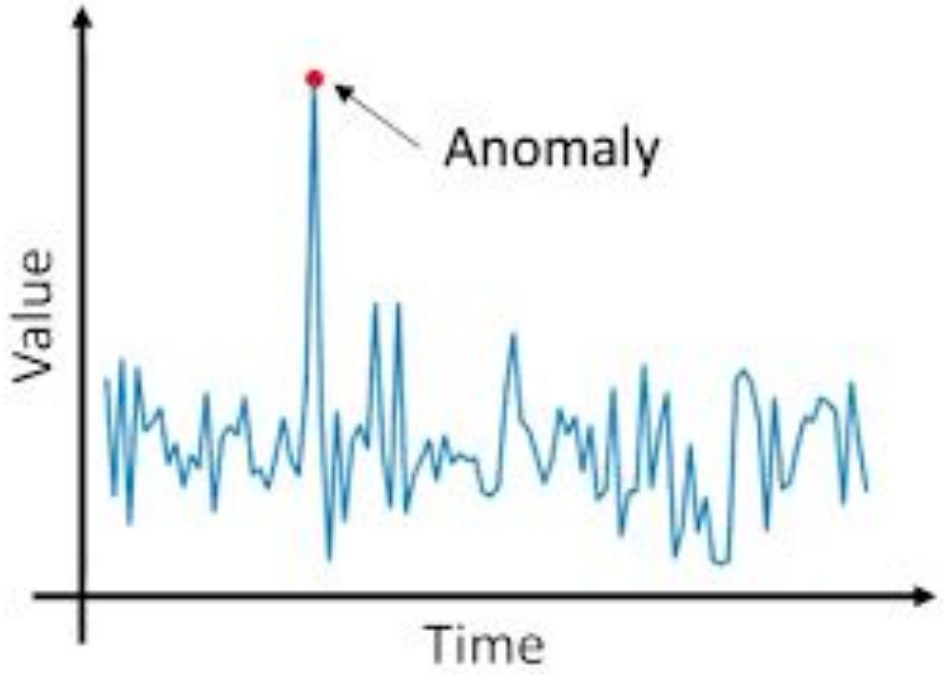


Źródło: https://iq.opengenus.org/types-of-autoencoder/

Detekcja anomalii

Detekcja anomalii to proces identyfikowania nieoczekiwanych pozycji lub zdarzeń w zestawach danych, które różnią się od normy. Wykrywanie anomalii ma dwa podstawowe założenia:

1. Anomalie występują w danych bardzo rzadko.
2. Anomalie różnią się znacznie od normalnych przypadków.



Część VII - Sieci rekurencyjne

O czym będziemy rozmawiać:

- Idea sieci rekurencyjnych
- LSTM i GRU
- Word Embeddings

Sieci rekurencyjne

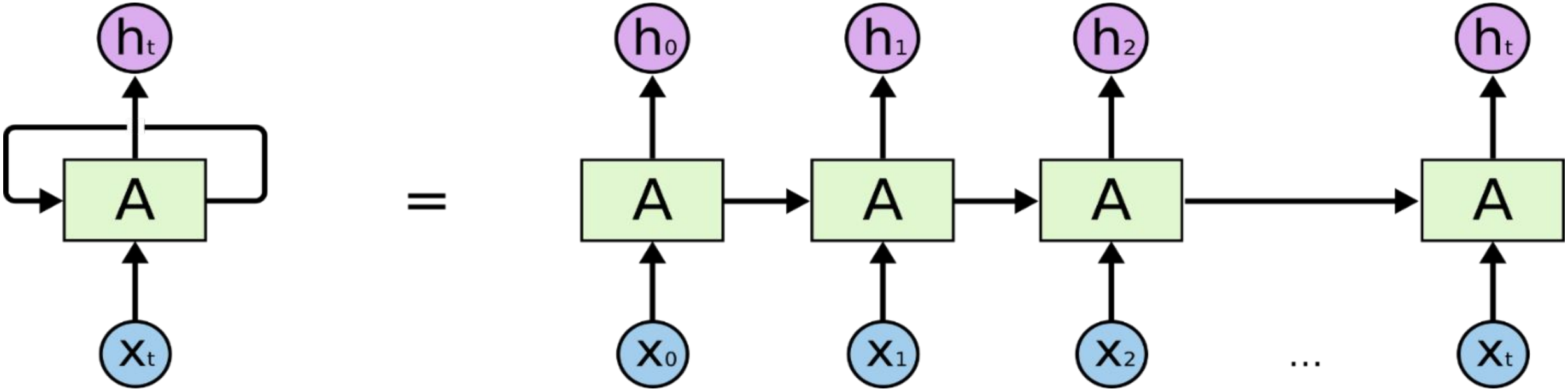
Sieci rekurencyjne (RNN) to specjalny typ sieci neuronowej wykorzystywany do modelowania danych **sekwencyjnych** (tekst, audio, video, ogólne szeregi czasowe).

RNN wykorzystywane są w:

- tłumaczeniu tekstu na mowę i mowy na tekst
- odczytywaniu tekstu ze zdjęć (**optical character recognition**)
- tłumaczeniu między językowym
- tworzenie muzyki i tekstu
- predykcji szeregów czasowych
- detekcji anomalii w szeregach czasowych
- kontrolowaniu zachowań robotów/dronów
- wielu innych

Warstwa rekurencyjna

W pojedynczym neuronie sieci rekurencyjnej modelowane są wszystkie wartości w sekwencji, a do **zamodelowania wartości kolejnej** wykorzystujemy **zamodelowane wartości poprzednie**. Dobrym analogiem klasycznych modeli może być model **ARIMA**.

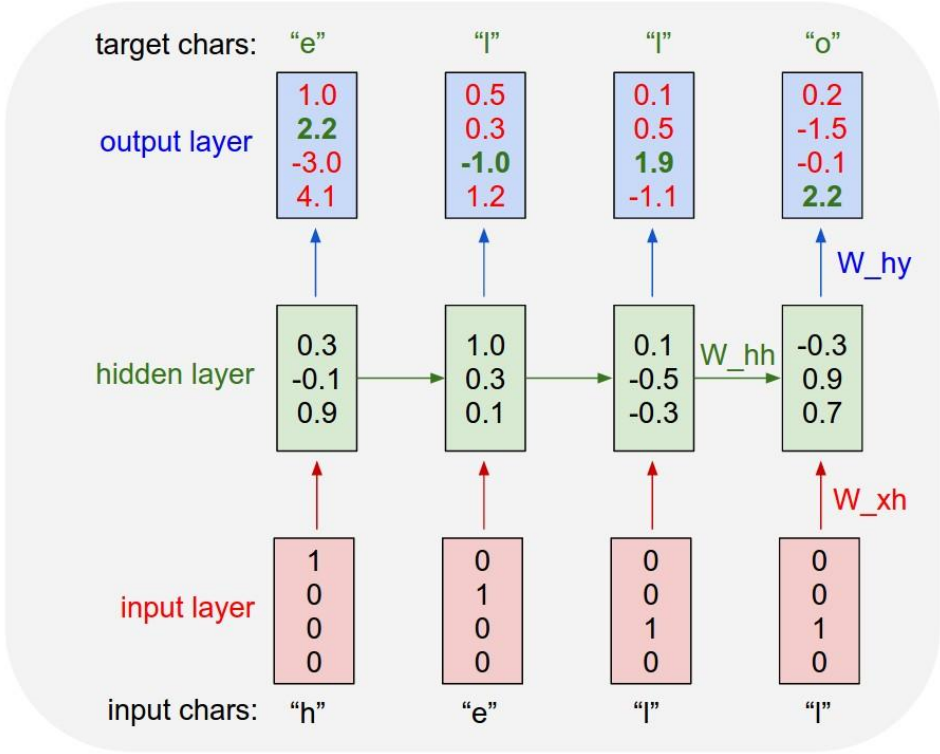


Źródło: http://colah.github.io/posts/2015-08-Understanding-LSTMs

Warstwa rekurencyjna

Przypuśćmy, że chcemy zamodelować kolejne litery w słowie/zdaniu. Dla uproszczenia załóżmy, że w naszym modelu **tokenem** (klasą) będzie pojedyncza litera. Każdy token zakodowany będzie jako wektor one-hot encoding długości N, gdzie N to ilość tokenów.

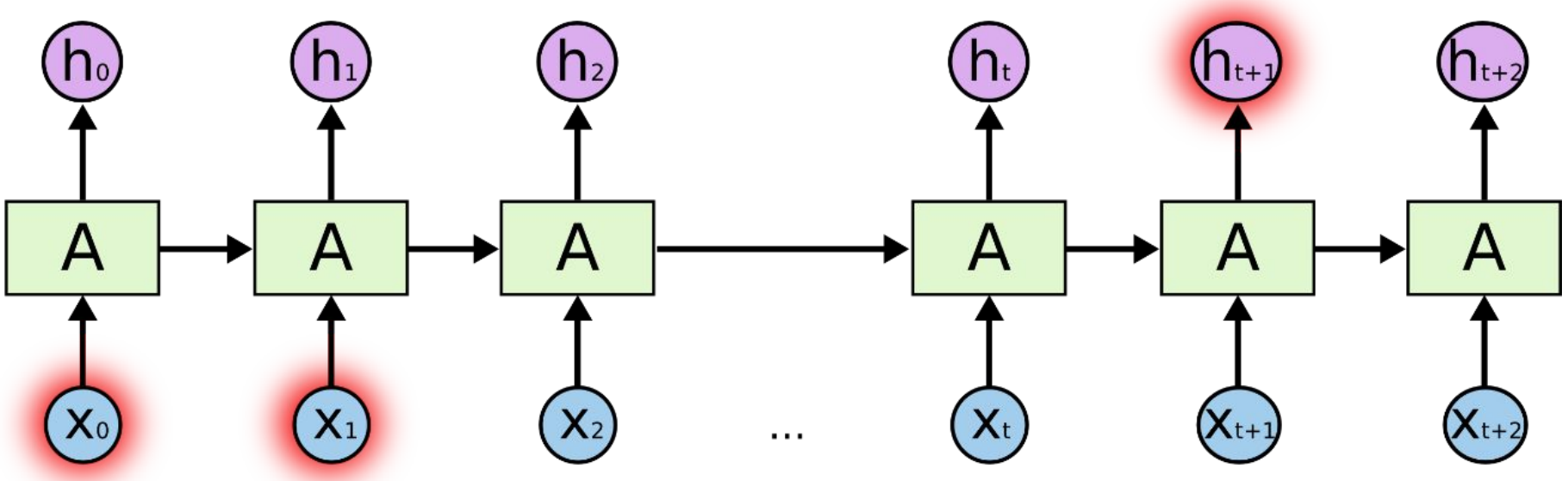
W warstwie rekurencyjnej neurony zasilane są informacją bieżącego tokena (wagi W_hx) oraz dopasowaniami z tokenów poprzednich (wagi W_hh)



Źródło: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Warstwa rekurencyjna

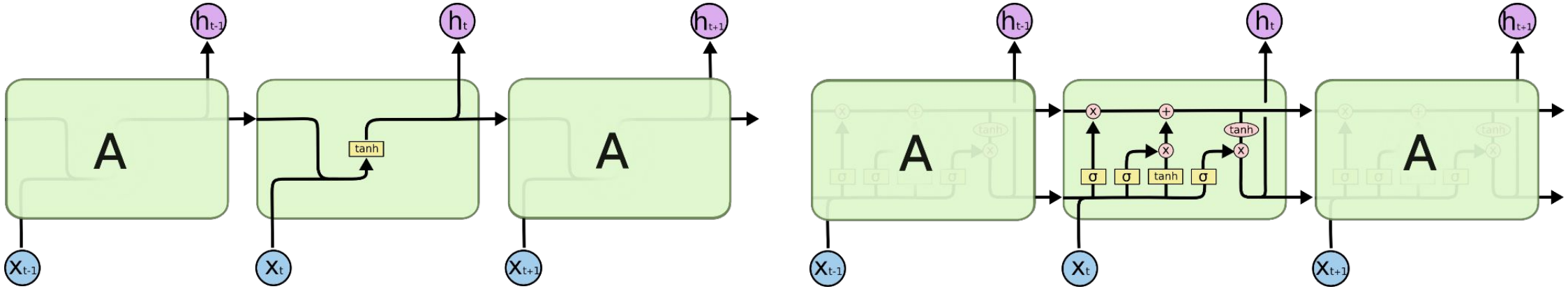
Podstawowym ograniczeniem prostej (klasycznej) warstwy rekurencyjnej jest **"krótka pamięć"**. Jeżeli nasza sekwencja jest bardzo długa w przypadku prostych RNN informacje z przeszłości bardzo szybko **wygasają** (nie są uwzględniane w modelowaniu późniejszych wartości). Rozwiązaniem tego problemu jest zastosowanie jednostek **LSTM** i **GRU**.



Źródło: http://colah.github.io/posts/2015-08-Understanding-LSTMs

LSTM

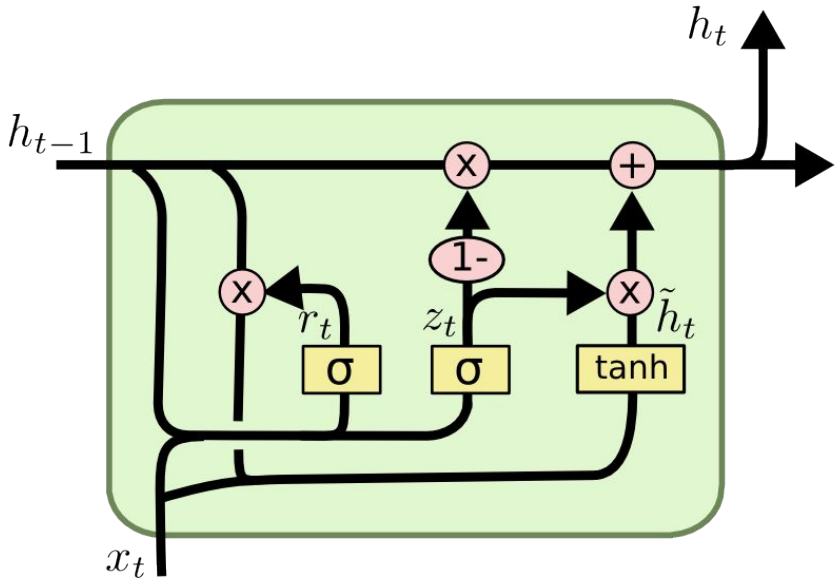
Long short-term memory (LSTM) to specjalna odmiana jednostek rekurencyjnych zaprojektowana z myślą o zachowaniu informacji w przypadku modelowania długich sekwencji. W przypadku LSTM rekurencyjne przekazywanie informacji odbywa się przy pomocy 4 funkcji aktywacji i specyficznej konstrukcji neuronu. Istnieje także kilka różnych wariantów LSTM. Poszczególne operacje (wrota) decydują o tym, jaka informacja w przeszłości powinna być **zapomniana**, a która **zachowana** i **przekazana** w przyszłość.



Źródło: http://colah.github.io/posts/2015-08-Understanding-LSTMs

GRU

Gated Recurrent Unit (GRU) jest kolejną odmianą (bardziej zaawansowaną od LSTM) jednostek rekurencyjnych, która korzysta z architektury wrót, aby zdecydować, który element powinien zostać zapomniany, a który przekazany dalej.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

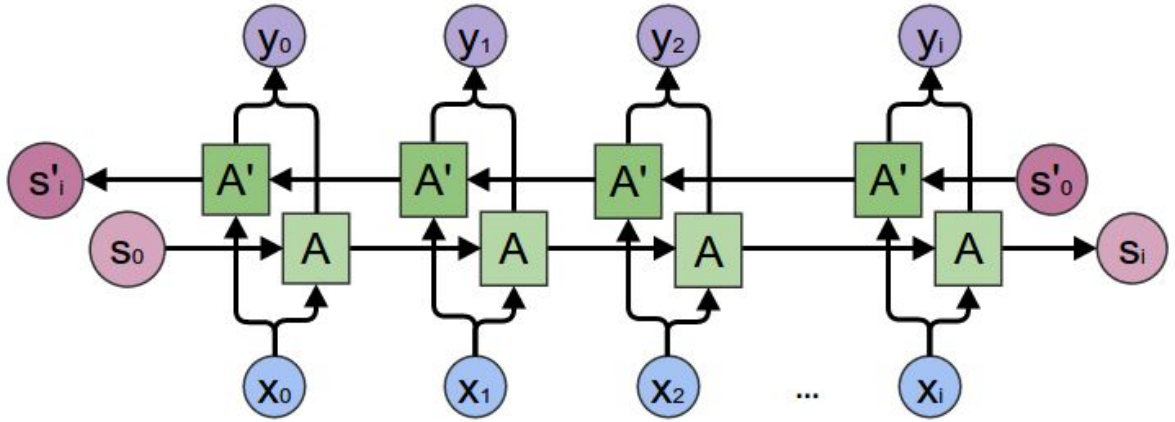
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Źródło: http://colah.github.io/posts/2015-08-Understanding-LSTMs

Rekurencyjne sieci dwukierunkowe

Bidirectional RNN, czyli rekurencyjna sieć dwukierunkowa jest kolejnym rozwinięciem architektury RNN. W sieci dwukierunkowej do modelowania wartości obecnej wykorzystujemy nie tylko wartości z przeszłości, ale także z przyszłości.

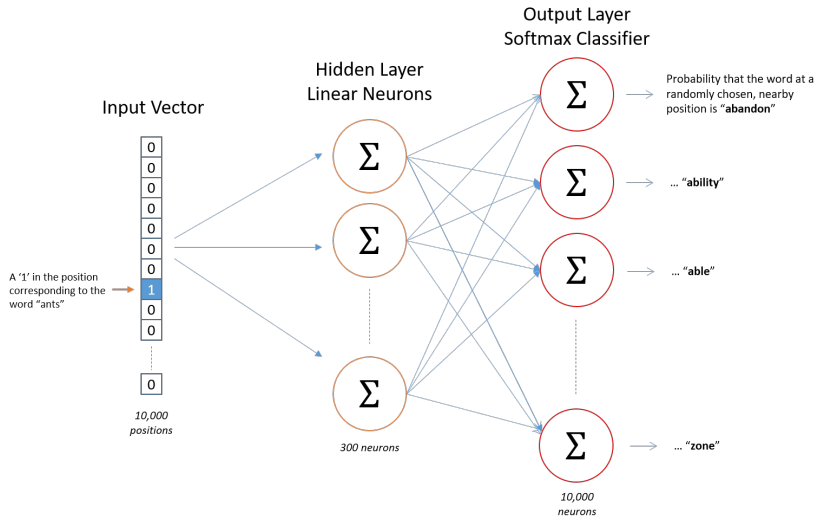
Można o tym myśleć jak o dwóch klasycznych sieciach rekurencyjnych działających w przeciwnych kierunkach.



Źródło: https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66

Word embeddings

W RNN inputem są **sekwencje tokenów** zakodowane w postaci **wektorów one-hot encoding**. W przypadku bardzo dużej liczby tokenów pojawia się jednak problem, ponieważ przetwarzanie tak dużych tensorów może okazać się bardzo czasochłonne bądź niemożliwe. Rozwiązaniem jest zakodowanie naszych tokenów za pomocą **embeddings** w **niskowymiarowej przestrzeni**. Aby tego dokonać do naszej sieci możemy dodać tak zwaną warstwę embeddingową. W warstwie tej tworzymy **jednowarstwowy perceptron**, którego zadaniem jest przewidzenie prawdopodobieństwa czy w pobliżu danego tokena pojawi się inny token ze słownika. Nie jesteśmy zainteresowani samymi prawdopodobieństwami (wynikiem), a wartościami w naszej warstwie ukrytej - embeddingami.



Źródło: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Część VIII - XAI

O czym będziemy rozmawiać:

- XAI
- DALEX
- LIME

XAI

Explainable artificial intelligence - XAI - jest zbiorem metod pozwalających na **wyjaśnienie** i **zrozumienie** zaawansowanych modeli machine / deep learningowych (**black boxów**) oraz decyzji przez te modele podejmowanych.

W obecnych czasach samo posługiwanie się zaawansowanymi black boxami to za mało. Dobrym przykładem może być tutaj sektor bankowy / finansowy. Klienci banków zgłaszających się po kredyt mają tak zwane **"prawo do wyjaśnienia"**, oznacza to, że w przypadku np. nieudzielenia kredytu przez bank, klient ma prawo dowiedzieć się dlaczego taka a nie inna decyzja została podjęta.

Innym przykładem może być sektor medycyny, gdzie lekarzom należy przedstawiać wyniki modelowania pewnych zjawisk w prosty i czytelny sposób.

DALEX

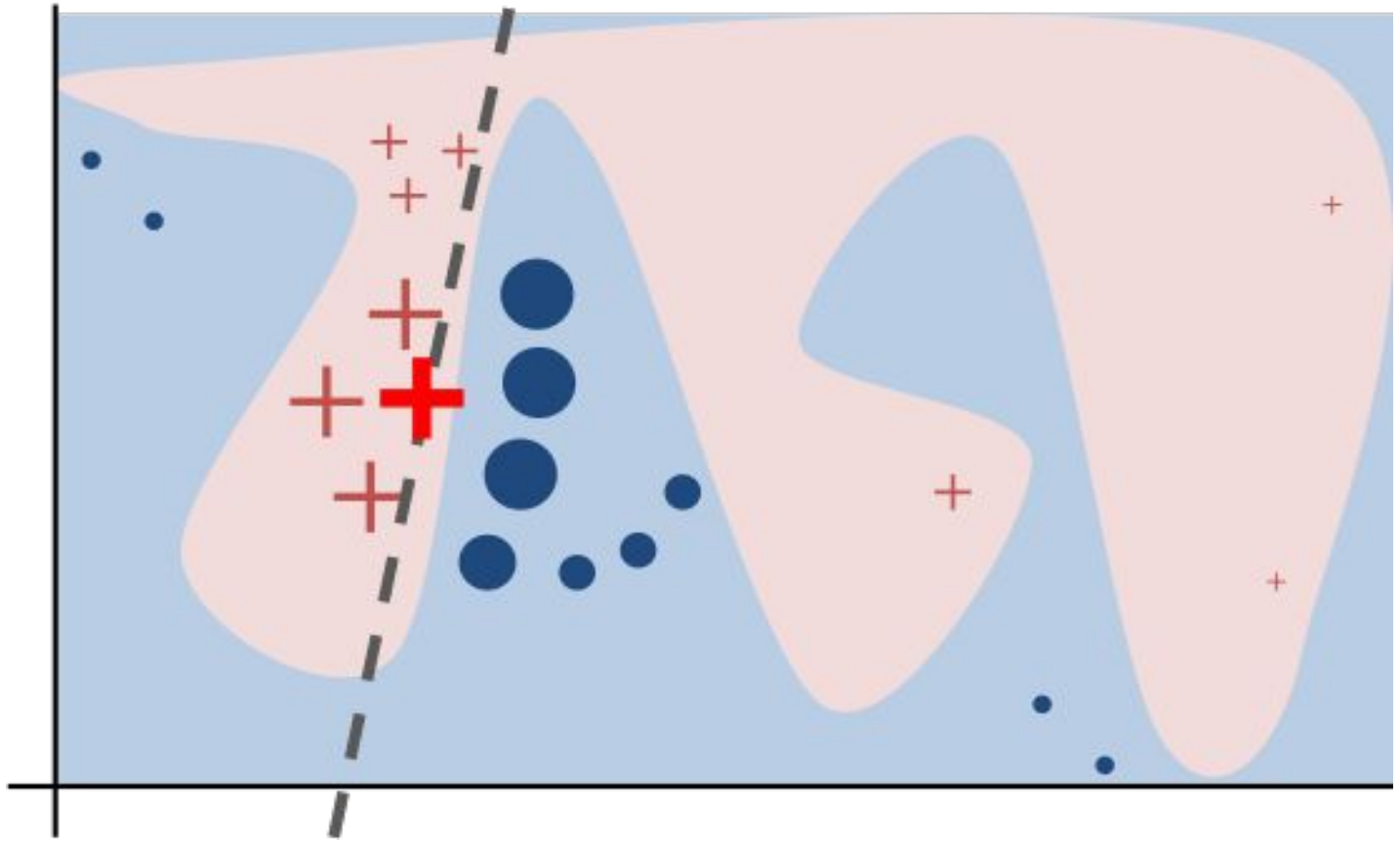
moDel Agnostic Language for Exploration and eXplanation - DALEX - jest pakietem pozwalającym tworzyć i wizualizować wyjaśnienia dla black boxów dowolnego typu (keras, h2o, mlr, i wiele innych).

Do podstawowych funkcjonalności DALEX należą:

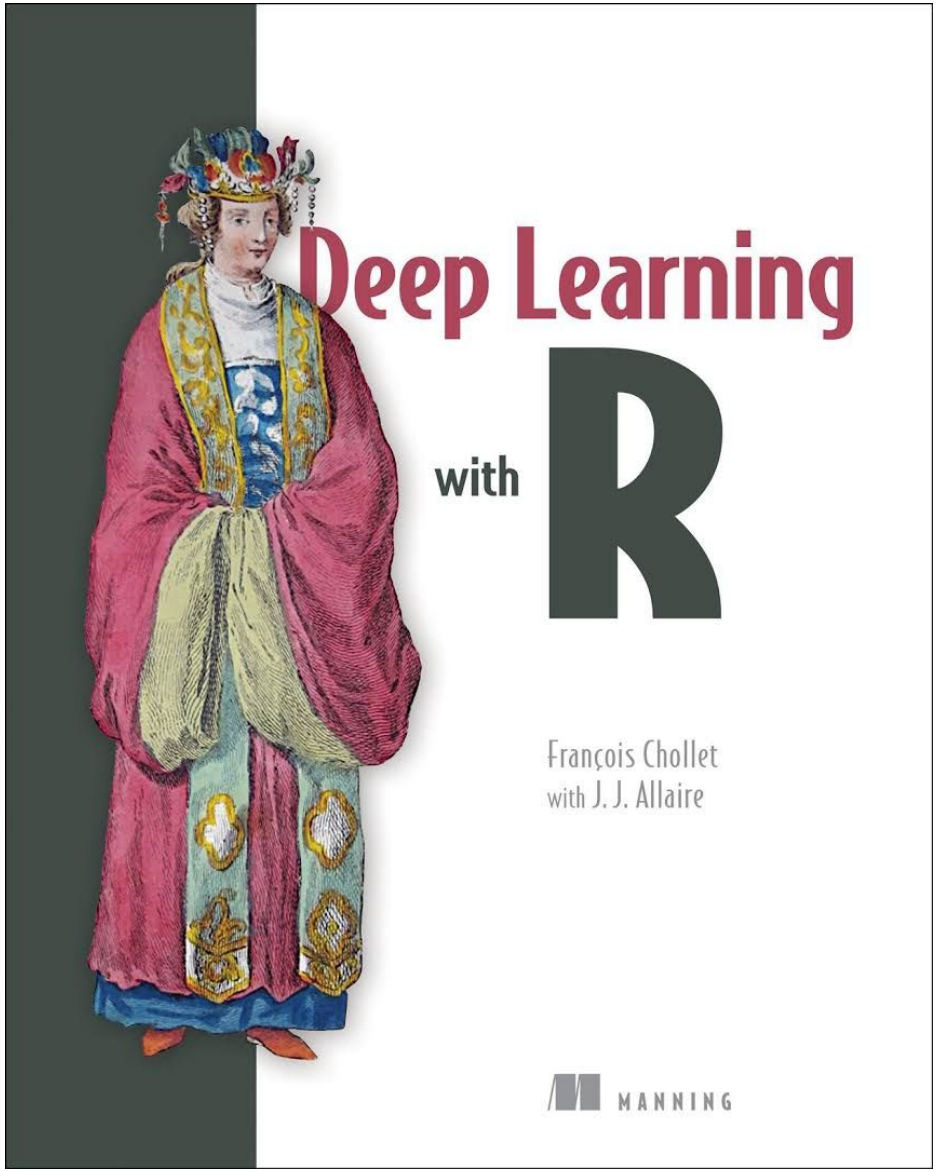
- badanie istotności zmiennych
- tłumaczenie predykcji dla konkretnej obserwacji
- tłumaczenie zależności między zmiennymi, a predykcjami
- tworzenie scenariuszy “co gdyby”

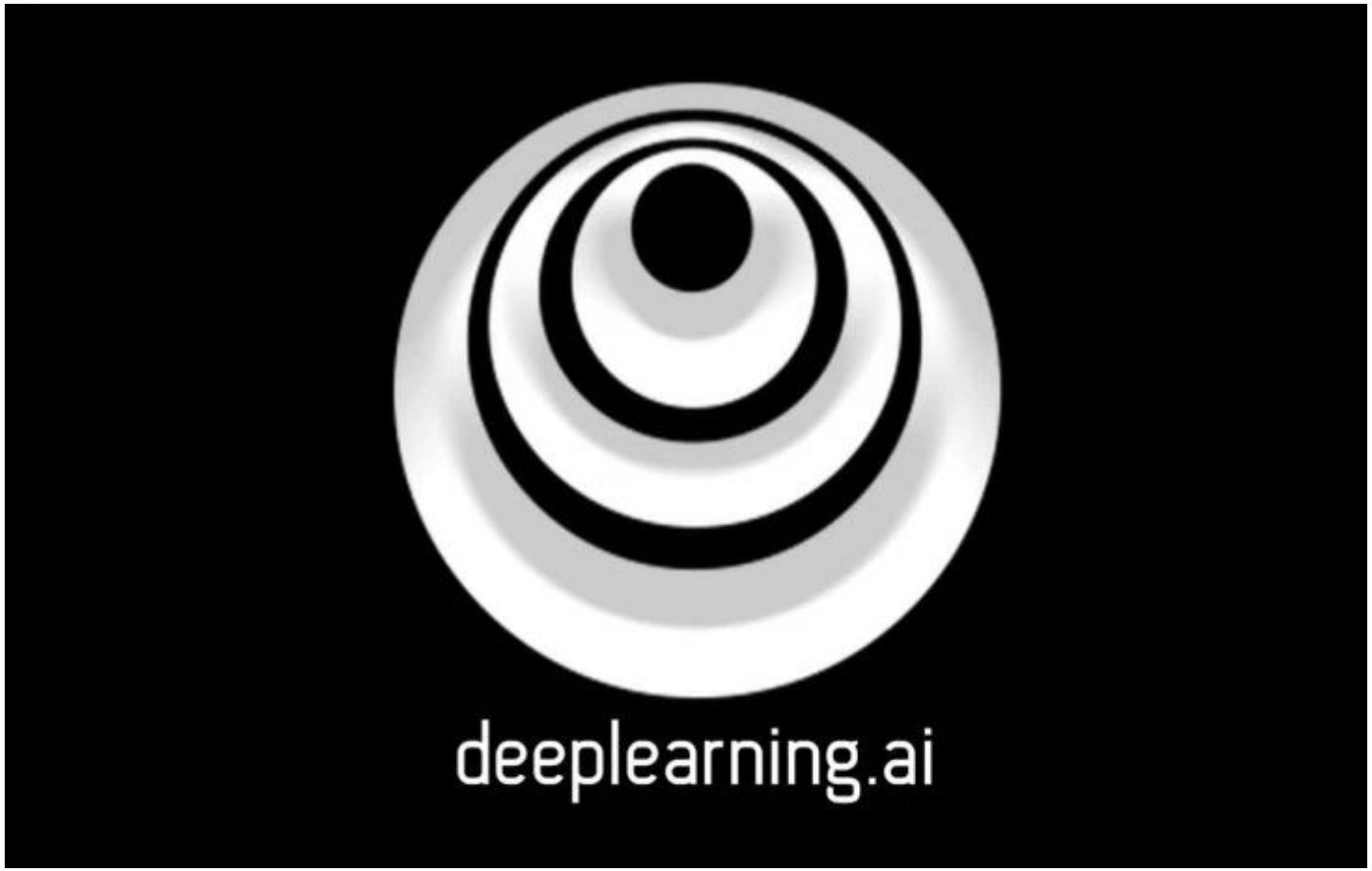
LIME

Local Interpretable Model-Agnostic Explanations - LIME - jest metodą, w której wyjaśniamy predykcje tworząc tak zwany **interpretowalny lokalny model zastępczy** (Surrogate Model)



Next steps


Deep Learning
with R
François Chollet
with J. J. Allaire
MANNING


deeplearning.ai

Pytania ?