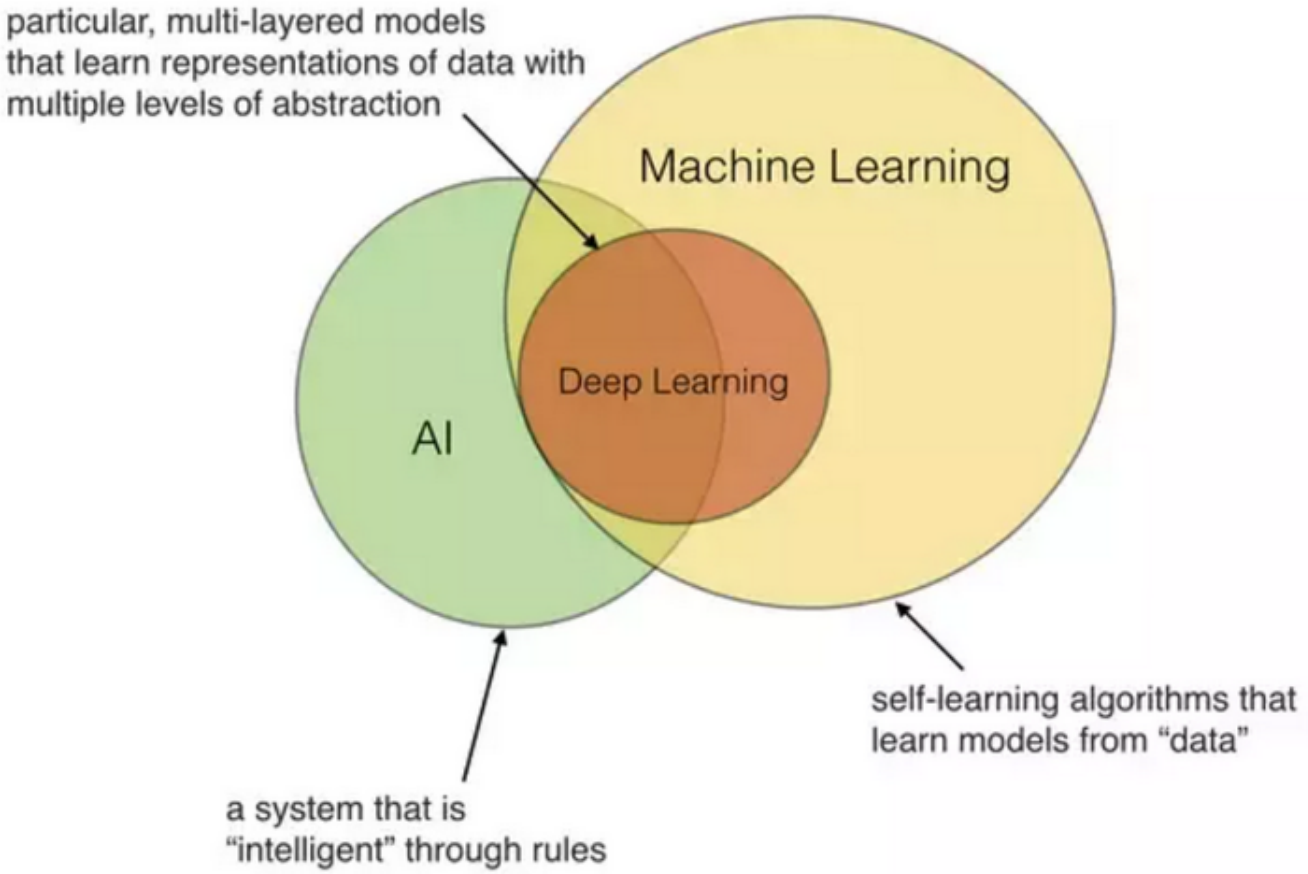
Uczenie Maszynowe

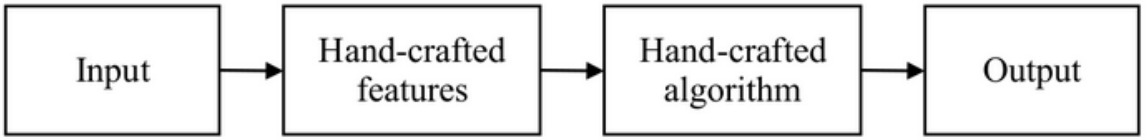
Uczenie głębokie - wykład 8

Adam Szmigielski
aszmigie@pjwstk.edu.pl
materiały: $ftp(public) : //aszmigie/WM$

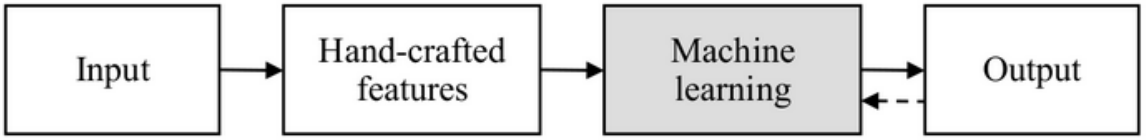
Uczenie głębokie



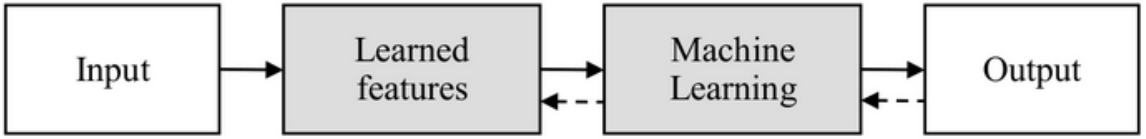
Uczenie głębokie w wizji komputerowej



(a) Traditional vision pipeline



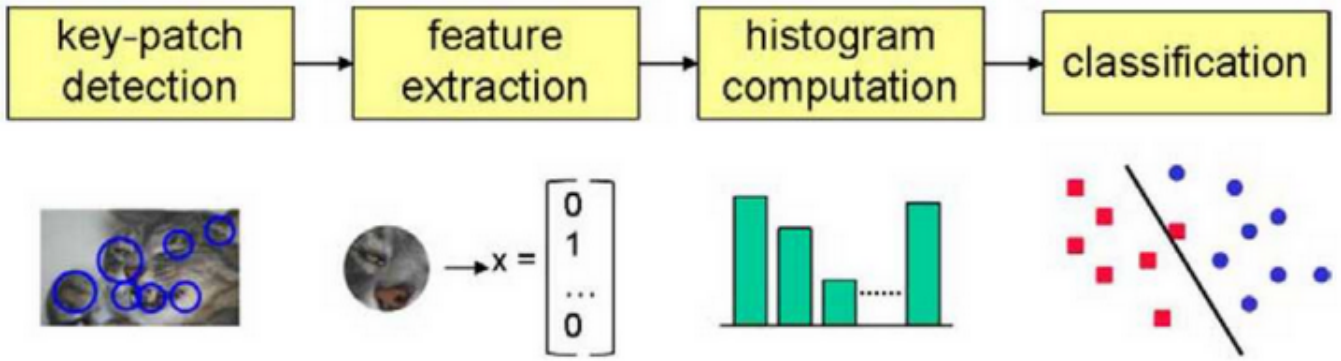
(b) Classic machine learning pipeline



(c) Deep learning pipeline

Obecnie głębokie sieci neuronowe są najpopularniejszymi i szeroko stosowanymi modelami uczenia maszynowego w wizji komputerowej, nie tylko do klasyfikacji semantycznej i segmentacji, ale nawet do zadań niższego poziomu, takich jak ulepszanie obrazu, szacowanie ruchu i odzyskiwanie głębi.

Typowa sekwencja przetwarzania danych



Uczenie głębokiej sieci neuronowej za pomocą standardowego interfejsu TensorFlow

Uczenie składa się z dwóch faz:

- **faza konstrukcyjna** - należy określić liczbę wejść i wyjść, a także wyznaczyć liczbę neuronów w każdej warstwie,
- **faza Faza wykonawcza** - otwiera sesję i uruchamia węzeł init inicjujący wszystkie zmienne. Następnie wkraczamy w główną pętlę - uczenie sieci neuronowej.

Faza konstrukcyjna

- Węzły zastępcze reprezentują dane uczące (wejściowe) X i docelowe (wyjściowe) y - węzeł X jako $(None, n_inputs)$, y jednowymiarowy tensor,

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

- Warstwa sieci opisana będzie przez parametry wejścia, liczbę neuronów, funkcję aktywacji i nazwę warstwy,

```
def neuron_layer(X, n_neurons, name, activation=None):
with tf.name_scope(name):
n_inputs = int(X.get_shape()[1])
stddev = 2 / np.sqrt(n_inputs)
init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
W = tf.Variable(init, name="jadro")
b = tf.Variable(tf.zeros([n_neurons]), name="obciazenie")
Z = tf.matmul(X, W) + b
if activation is not None:
return activation(Z)
else:
return Z
```

- określamy konstruktor sieci

```
with tf.name_scope("gsn"):
    hidden1 = tf.layers.dense(X, n_hidden1, name="ukryta1",
    activation=tf.nn.relu)
    hidden2 = tf.layers.dense(hidden1, n_hidden2, name="ukryta2",
    activation=tf.nn.relu)
    logits = tf.layers.dense(hidden2, n_outputs, name="wyjscia")
```

- wyliczenie stratę informacji (poprzez wyliczenie *entropii krzyżowej*)

```
with tf.name_scope("strata"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
    logits=logits)
    loss = tf.reduce_mean(xentropy, name="strata")
```

- liczenie gradientu wraz z optymalizatorem:

```
learning_rate = 0.01
with tf.name_scope("uczenie"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

- Ocenę działania sieci:

```
with tf.name_scope("ocena"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

- węzeł inicjalizujący sieć:

```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Faza wykonawcza

Importujemy dane, otwieramy sesję *TensorFlow* i uruchamia węzeł init inicjujący wszystkie zmienne.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/dane/")
n_epochs = 40
batch_size = 50

init = tf.global_variables_initializer()
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.validation.images,y:
        mnist.validation.labels})
        print(epoch, "Dokladnosc uczenia:", acc_train,
        "Dokladnosc testowania:", acc_test)
    save_path = saver.save(sess, "./moj_model_ostateczny.ckpt"
```

Siec możemy ponownie odczytać i z niej korzystać.

Strojenie hiperparametrów sieci neuronowej

- **Liczba ukrytych warstw** - mogą one modelować skomplikowane funkcje przy wykładniczo mniejszej liczbie neuronów niż w przypadku "płytkich" sieci (znacznie przyśpiesza ich naukę),
- **Liczba neuronów tworzących warstwę ukrytą** - jest uzależniona od danych wejściowych i wyjściowych wymaganych przez określone zadanie (zazwyczaj kolejne warstwy mają mniejszą liczbę neuronów),
- **Funkcje aktywacji** - W większości przypadków wstawiamy w warstwach ukrytych funkcję ReLU, wa warstwie wyjściowej funkcja aktywacji zależy od celu stosowania sieci (softmax dla klasyfikacji, liniowa dla regresji) .

Problemy nauki sieci głębokich

- Problem zanikających gradientów (lub ściśle powiązanym z nim problemem eksplodujących gradientów), który dotyczy głębokich sieci (znacznie utrudnia uczenie niższych warstw sieci),
- Uczenie rozbudowanej sieci może być bardzo czasochłonne.
- Model zawierający miliony parametrów jest w znacznym stopniu narażony na przetrenowanie.

Problemy zanikających/eksplodujących gradientów

- Algorytm propagacji wstecznej przebiega od warstwy wyjściowej do wejściowej i rozprowadza po drodze wartość gradientu błędu.
- Wartości gradientów często maleją wraz z przebiegiem algorytmu do niższych warstw sieci - problem zanikających gradientów (ang. vanishing gradients).
- Czasami możemy natrafić na przeciwne zjawisko: gradienty stale rosną, przez co w wielu warstwach wagi są aktualizowane w szybkim tempie - problemem eksplodujących gradientów (ang. exploding gradients),
- Sieci neuronowe wykazują syndrom niestabilnych gradientów; poszczególne warstwy mogą uczyć się z diametralnie różnymi szybkościami.

Inicjacje wag Xaviera i He

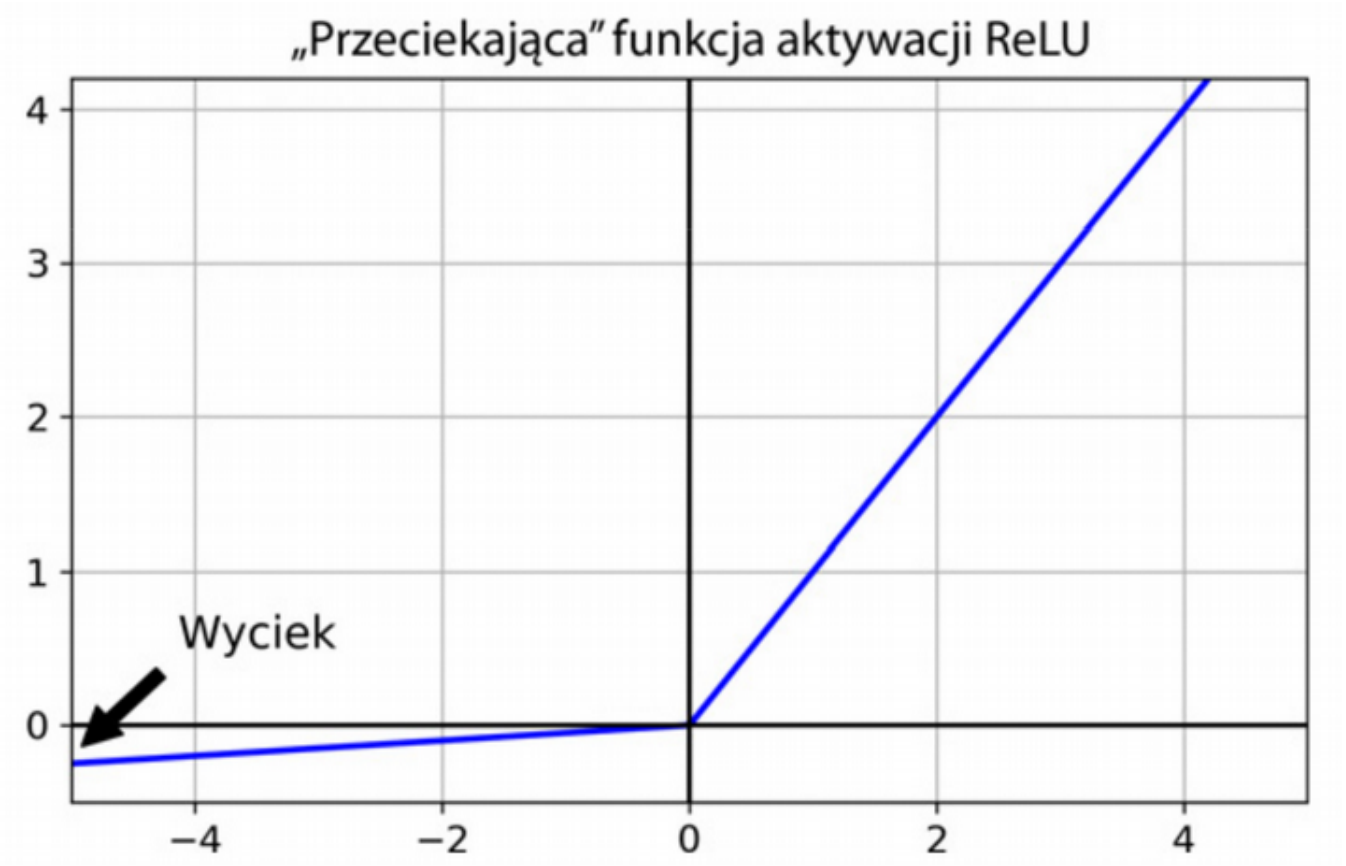
Funkcja aktywacji	Rozkład jednorodny <-r, r>	Rozkład normalny
logistyczna	$r = \sqrt{\frac{6}{n_{inputs}+n_{outputs}}}$	$\sigma = \sqrt{\frac{6}{n_{inputs}+n_{outputs}}}$
tang. hiperboliczny	$r = 4\sqrt{\frac{6}{n_{inputs}+n_{outputs}}}$	$\sigma = 4\sqrt{\frac{6}{n_{inputs}+n_{outputs}}}$
typu RELU	$r = \sqrt{2}\sqrt{\frac{6}{n_{inputs}+n_{outputs}}}$	$\sigma = \sqrt{2}\sqrt{\frac{6}{n_{inputs}+n_{outputs}}}$

- W celu właściwego sygnalizowania kierunku przepływu informacji wariancja wyjść w danej warstwie musi być równa wariancji jej wejść,
- Gradienty muszą mieć taką samą wariancję przed przejściem i po przejściu przez warstwę w odwrotnym kierunku.

Nienasycające funkcje aktywacji

- W przypadku głębokich sieci neuronowych funkcja ReLU sprawuje się lepiej od sigmoidalnych, gdyż nie ulega nasyceniu dla wartości dodatnich,
- Funkcja ReLU nie jest idealna - problem śmierci ReLU (ang. dying ReLUs). W czasie uczenia niektóre neurony trwale "giną", tj. jedynym przesyłanym przez nie sygnałem jest 0,
- Trudno jest ponownie "wprowadzić" neuron do sieci - gradient funkcji ReLU wynosi 0 przy ujemnych wartościach na wejściu.

Przeciekająca funkcja ReLU



Warianty funkcji ReLU

- **Przeciekającą funkcją ReLU (ang. leaky ReLU)**:

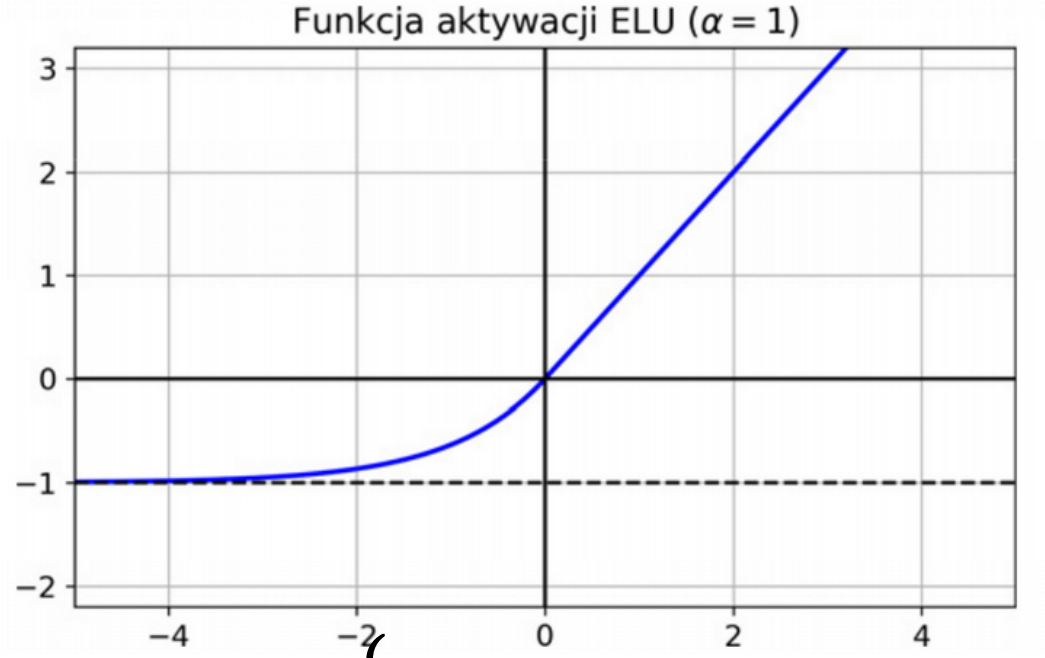
$$ReLU_\alpha(z) = \max\{\alpha_z, z\}$$

  Hiperparametr α określa stopień "przeciekania" funkcji: wyznacza on nachylenie funkcji dla $z < 0$ i zazwyczaj przyjmuje wartość ok. $0,01$.

- **Losowa, przeciekająca funkcja ReLU** (ang. randomized leaky ReLU — RReLU) - wartość hiperparametru α jest podczas nauki losowo dobierana w określonym zakresie,

- **Parametryczna, przeciekająca funkcja ReLU** (ang. parametric leaky ReLU — PReLU) - parametr α "uczy się" wraz z całym modelem (podobnie jak pozostałe parametry, może być modyfikowany w ramach propagacji wstecznej).

Jednostka wykładniczo-liniową (ang. Exponential Linear Unit)



$$ELU_\alpha(z) = \begin{cases} \alpha(e^z - 1) & gdy \quad z < 0 \\ z & gdy \quad z \geq 0 \end{cases}$$

- W przeprowadzonych doświadczeniach okazała się lepsza od wszystkich odmian funkcji ReLU: czas nauki był krótszy, a sama sieć neuronowa sprawowała się lepiej wobec zbioru testowego.

Własności funkcji aktywacji ELU

- Dla $z < 0$ przyjmuje ona ujemne wartości, dzięki czemu średnia wartość na wyjściu zbliżona do zera - rozwiązuje to problem zanikających gradientów,
- Dla $z < 0$ ma ona niezerowy gradient, co stanowi rozwiązanie problemu "umierających" neuronów.
- Funkcja jest gładka w każdym punkcie, również w $z = 0$, co pozwala przyśpieszyć metodę gradientu prostego,
- Jest ona wolniej wyliczana od standardowej funkcji ReLU i jej odmian, a w trakcie uczenia jest to rekompensowane lepszym współczynnikiem zbieżności. Sieć z ELU będzie wolniejsza od sieci z ReLU.

Normalizacja wsadowa

- W uczeniu głębokim istnieje problemu rozkładu zmian wejść w każdej warstwie podczas uczenia, wynikających ze zmian parametrów we wcześniejszej warstwie,
- Przed funkcją aktywacji w każdej warstwie, normalizuje się dane wejściowych, a następnie przeskalowuje i przesuwa wynik za pomocą dwóch nowych parametrów:
  - jeden odpowiada za skalowanie
  - drugi za przesunięcie
- Operacja ta pozwala modelowi określić optymalną skalę i średnią danych wejściowych dla każdej warstwy.

Normalizacja wsadowa za pomocą modułu TensorFlow

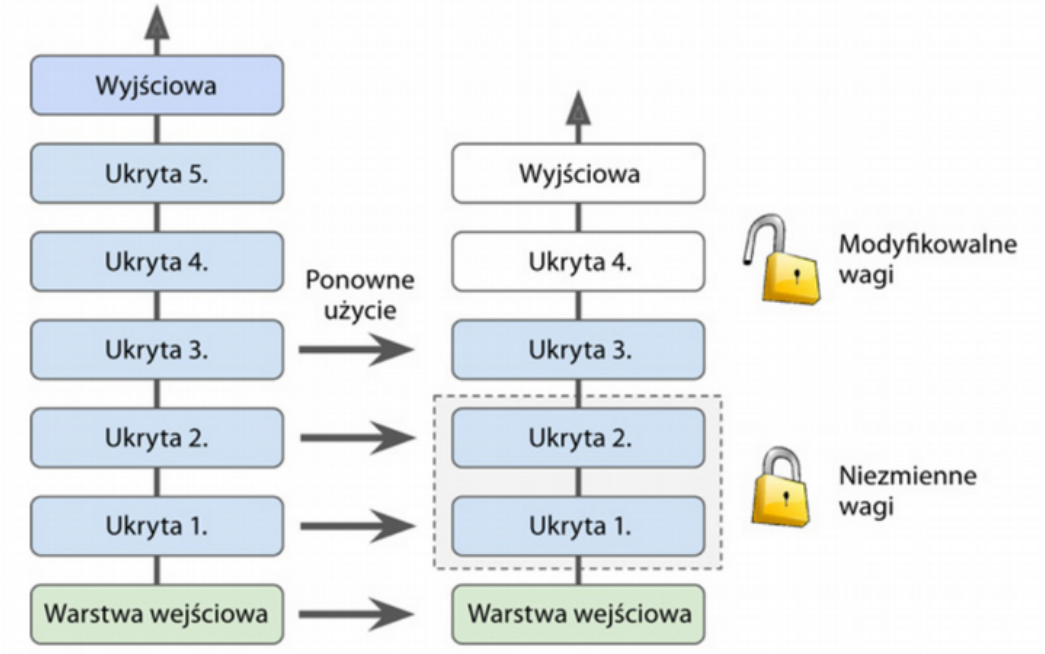
```
hidden1 = tf.layers.dense(X, n_hidden1, name="ukryta1")
bn1 = tf.layers.batch_normalization(hidden1, training=training, momentum=0.9)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="ukryta2")
bn2 = tf.layers.batch_normalization(hidden2, training=training, momentum=0.9)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="wyjscia")
logits = tf.layers.batch_normalization(logits_before_bn,
training=training, momentum=0.9)
```

- Moduł TensorFlow zawiera funkcję *tf.nn.batch_normalization*(), wyśrodkowującą i normalizującą dane wejściowe.
- Funkcja *tf.layers.batch_normalization*() sama oblicza średnią i odchylenie standardowe (za pomocą minigrup danych uczących podczas trenowania) i przekazuje je funkcji w postaci parametrów.
- Przy normalizacji wsadowej nie wyznacza się żadnej funkcji aktywacji - tą stosuje się po każdej normalizacji.

Obcinanie gradientu

- Popularną techniką służącą ograniczaniu problemu eksplodujących gradientów jest obcinanie gradientów na etapie propagacji wstecznej w taki sposób, żeby nigdy nie przekraczały określonego progu,

Wielokrotne stosowanie gotowych warstw



- Uczenie bardzo dużych sieci jest kosztowne.
- Można wykorzystać istniejącą sieć dostosowanej do podobnego zadania i wykorzystać jej niższe warstwy - **uczenie transferowe** (ang. transfer learning),
- Znacząco przyśpiesza ono proces uczenia, lecz także wymaga znacznie mniej danych uczących.

Zamrażanie niższych warstw

```
with tf.name_scope("gsn"):
        hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
        name="ukryta1") # ponownie uzyta, zamrozona
        hidden2 = tf.layers.dense(hidden1, n_hidden2, activation=tf.nn.relu,
        name="ukryta2") # ponownie uzyta, zamrozona
        hidden2_stop = tf.stop_gradient(hidden2)
        hidden3 = tf.layers.dense(hidden2_stop, n_hidden3, activation=tf.nn.relu,
        name="ukryta3") # ponownie uzyta, niezamrozona
        hidden4 = tf.layers.dense(hidden3, n_hidden4, activation=tf.nn.relu,
        name="ukryta4") # nowa!
        logits = tf.layers.dense(hidden4, n_outputs, name="wyjscia") # nowa!
```

- Poprzez wstawienie warstwy *stop_gradient*() do grafu. Wszelkie warstwy znajdujące się pod nią zostaną zamrożone:

Repozytoria modeli

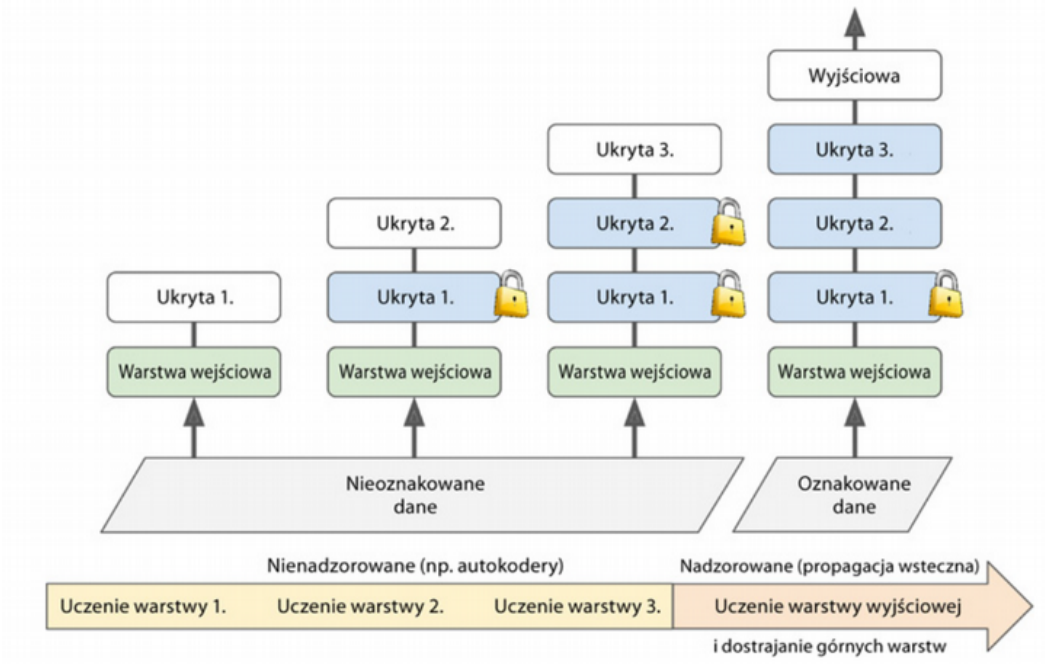
- Moduł $TensorFlow$ ma własne repozytorium na stronie $https://github.com/tensorflow/models$

  Znajdują się tam sieci przeznaczone do klasyfikacji obrazów, takie jak VGG, Inception czy ResNet, gotowe modele, a także narzędzia pozwalające na pobranie najpopularniejszych zbiorów danych uczących

- Repozytorium Model Zoo firmy Caffe:
  $https://github.com/BVLC/caffe/wiki/Model-Zoo$

  Dostępnych jest tu wiele modeli cyfrowego rozpoznawania obrazu (np. LeNet, AlexNet, ZFNet, GoogLeNet, VGGNet, Inception) wytrenowanych na różnorodnych zbiorach danych (takich jak ImageNet, Places Database, CIFAR10 itd.).

Nienadzorowane uczenie wstępne



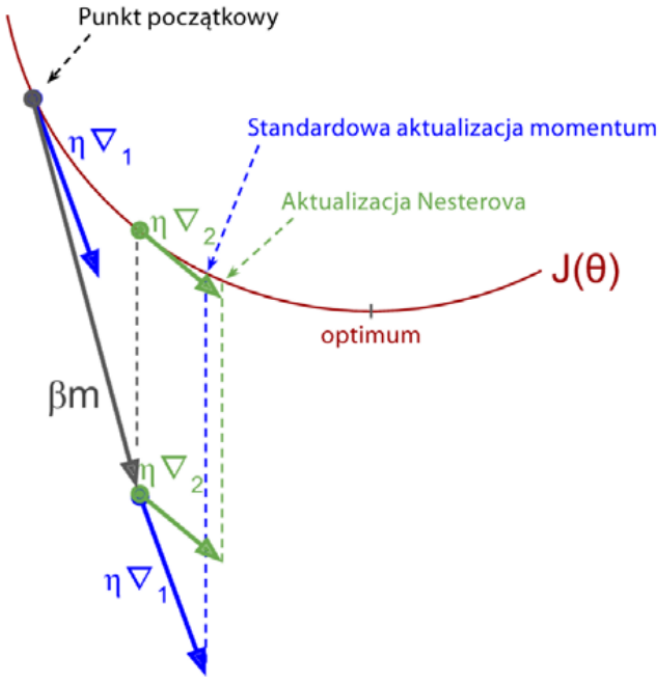
- Dla nieoznakowanych danych uczących, można spróbować trenować model warstwa po warstwie, przy użyciu algorytmu nienadzorowanego wykrywania cech (np. maszyn Boltzmanna lub autokoderów).
- Każda następna warstwa jest uczona przy użyciu wyników uzyskanych za pomocą wcześniejszych warstw,
- Po wyuczeniu wszystkich warstw sieć stroi się propagacja wsteczna.

Algorytm optymalizacji momentum

$$m \leftarrow \beta m - \eta \nabla_\Theta J(\Theta)$$
$$\Theta \leftarrow \Theta + m$$

- W metodzie *gradientu prostego* wagi Θ są aktualizowane poprzez odejmowanie gradientu funkcji kosztu $J(\Theta)$ od wag - $\nabla_\Theta J(\Theta)$ pomnożonych przez współczynnik uczenia η. Wcześniejsze gradienty nie są tu brane pod uwagę,
- Optymalizacja *momentum* uwzględnia wcześniejsze gradienty: w każdym przebiegu lokalny gradient jest odejmowany od **wektora momentu** m (pomnożonego przez współczynnik uczenia η).
- Parametr β zwany po prostu momentem (albo pędem) zapobiega nadmiernemu wzrostowi pędu.

Przyśpieszony spadek wzdłuż gradientu (algorytm Nesterova)

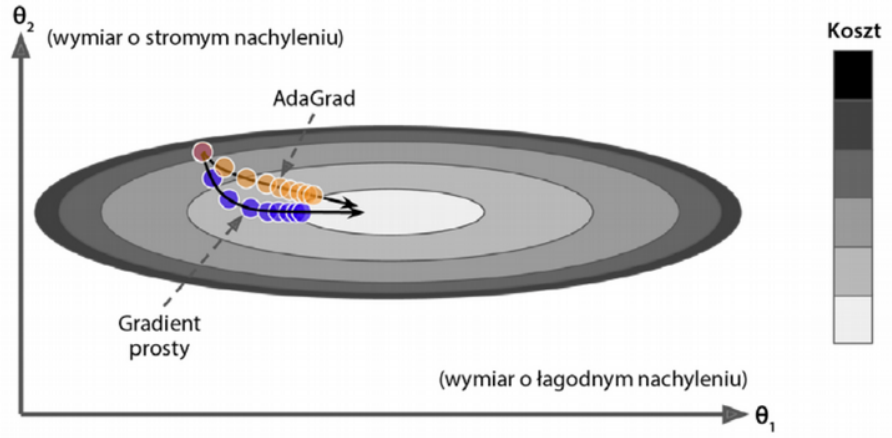


$$m \leftarrow \beta m - \eta \nabla_{\Theta} J(\Theta + \beta m)$$

$$\Theta \leftarrow \Theta + m$$

Jedyną różnicą w porównaniu do optymalizacji momentum jest pomiar gradientu za pomocą wyrażenia $\Theta + \beta m$, a nie Θ.

Algorytm AdaGrad



$$s \leftarrow s + \nabla_\Theta J(\Theta) \otimes \nabla_\Theta J(\Theta)$$

$$\Theta \leftarrow \Theta - \eta \nabla_\Theta J(\Theta) \oslash \sqrt{s + \in}$$

- Gdzie symbol $\otimes$ oznacza mnożenie poszczególnych elementów, symbol $\oslash$ oznacza dzielenie przez poszczególne elementy, a parametr $\in$ jest członem wygładzającym, służącym do unikania operacji dzielenia przez 0,
- Dokonuje tego poprzez stopniowe zmniejszanie wektora gradientów wzdłuż najbardziej stromych kierunków

Algorytm RMSProp

$$s \leftarrow \beta s + (1 - \beta)\nabla_\Theta J(\Theta) \otimes \nabla_\Theta J(\Theta)$$
$$\Theta \leftarrow \Theta - \eta \nabla_\Theta J(\Theta) \oslash \sqrt{s + \in}$$

- Algorytm AdaGrad nigdy nie osiąga zbieżności z minimum globalnym,
- Problem ten zostaje rozwiązany przez algorytm RMSProp, poprzez gromadzenie wyłącznie gradientów z najbardziej aktualnych przebiegów,
- Współczynnik rozkładu β zazwyczaj ma wartość $0,9$.

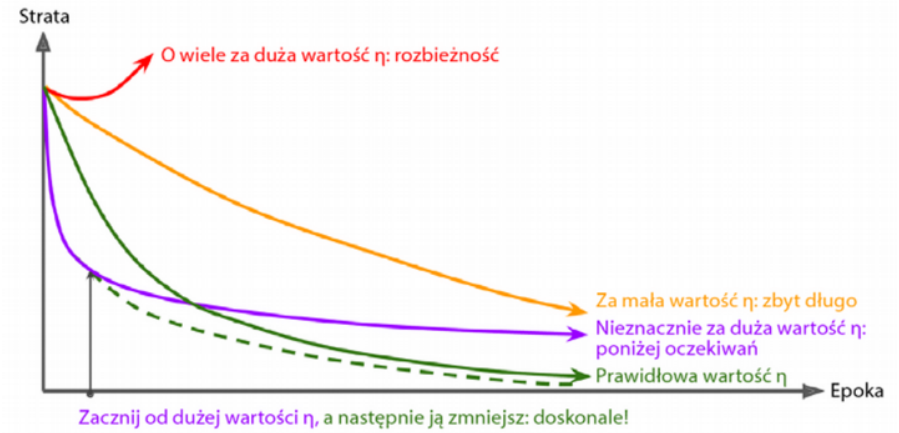
Optymalizacja Adam (ang. adaptive moment estimation)

$$m \leftarrow \beta_1 m - (1 - \beta_1)\nabla_\Theta J(\Theta)$$
$$s \leftarrow \beta_2 s + (1 - \beta_2)\nabla_\Theta J(\Theta) \otimes \nabla_\Theta J(\Theta)$$
$$m \leftarrow \frac{m}{1-\beta_1^t}$$
$$s \leftarrow \frac{s}{1-\beta_2^t}$$
$$\Theta \leftarrow \Theta + \eta m \oslash \sqrt{s + \in}$$

gdzie t - numer przebiegu (począwszy od 1).

- Algorytm Adam łączy koncepcję optymalizacji momentum i optymalizatora RMSProp,
- z optymalizacji momentum bierze śledzenie rozkładu wykładniczego średniej wcześniejszych gradientów,
- z optymalizatora RMSProp śledzenie rozkładu wykładniczego średniej wcześniejszych kwadratów gradientów

Harmonogramowanie współczynnika uczenia

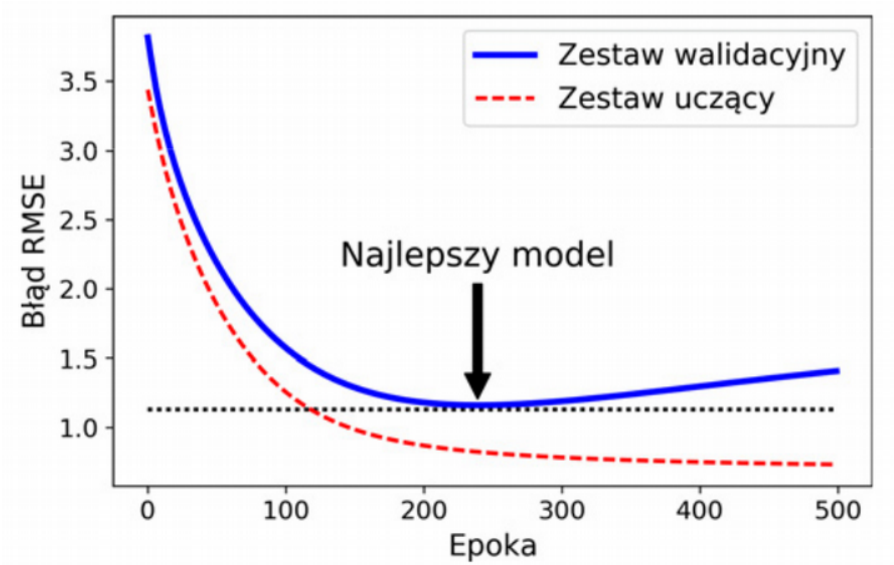


- **Odgórny, przedziałowy, stały współczynnik uczenia** - często wymaga dodatkowego ustalania wartości oraz momentu ich zmiany.
- **Harmonogramowanie wydajnościowe** - mierzymy błąd walidacji co N przebiegów, i w momencie, gdy wartość błędu przestaje maleć, redukujemy współczynnik uczenia
- **Harmonogramowanie wykładnicze** - wyznaczamy współczynnik uczenia na podstawie funkcji liczby przebiegów t
- **Harmonogramowanie potęgowe** - podobne do wykładniczgo, ale wartość współczynnika uczenia maleje znacznie wolniej.

Metody ochrony przed nadmiernym dopasowaniem

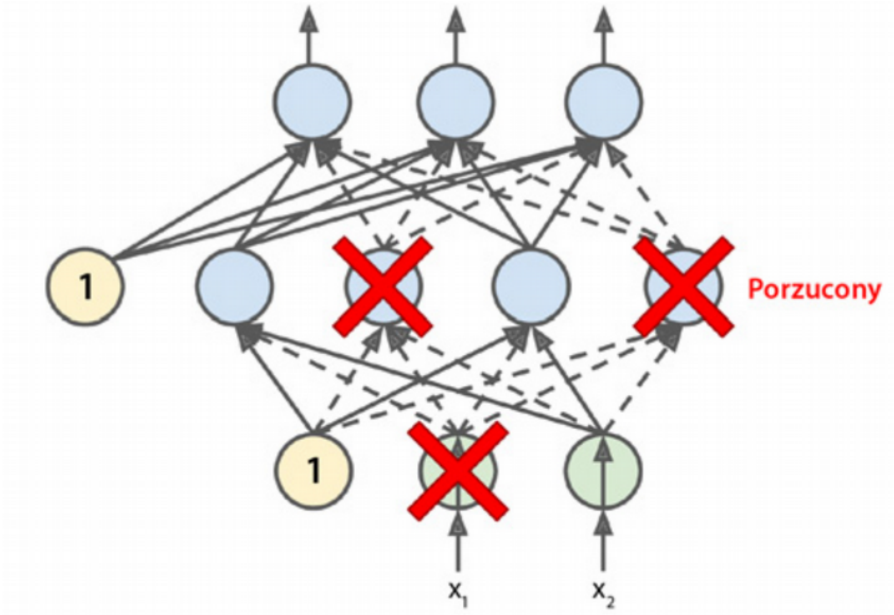
- **Wczesne zatrzymanie** - w momencie spadku wydajności modelu wobec zbioru walidacyjnego,
- **Regularyzacja L_1 i L_2** - można używać w sieciach neuronowych do ograniczania wag połączeń,
- **Porzucanie** - w poszczególnych przebiegach uczenia neuron może zostać tymczasowo "porzucony" tj. całkowicie pominięty,
- **Augmentacja danych** - polega na tworzeniu nowych przykładów uczących z próbek już istniejących.

Wczesne zatrzymanie



- Zakończenie uczenia w momencie osiągnięcia minimum błędu walidacyjnego - wczesne zatrzymywanie (ang. early stopping),
- W miarę upływu kolejnych epok algorytm uczy się, a jego błąd predykcji wobec zestawu uczącego maleje w naturalny sposób, tak samo jak błąd prognozowania wobec danych walidacyjnych.
- Po pewnym czasie błąd prognozowania przestaje maleć i zaczyna znów rosnąć (moment przetrenowania).

Porzucanie



- W poszczególnych przebiegach uczenia każdy neuron (oprócz warstwy wyjściowej) może zostać tymczasowo pominięty w procesie uczenia,
- W każdym następnym przebiegu może znów być aktywny,
- Hiperparametr p nosi nazwę współczynnika porzucenia (ang. dropout rate) i zazwyczaj ma przyporządkowaną wartość 50%.
- Po zakończeniu nauki neurony przestają być porzucane.

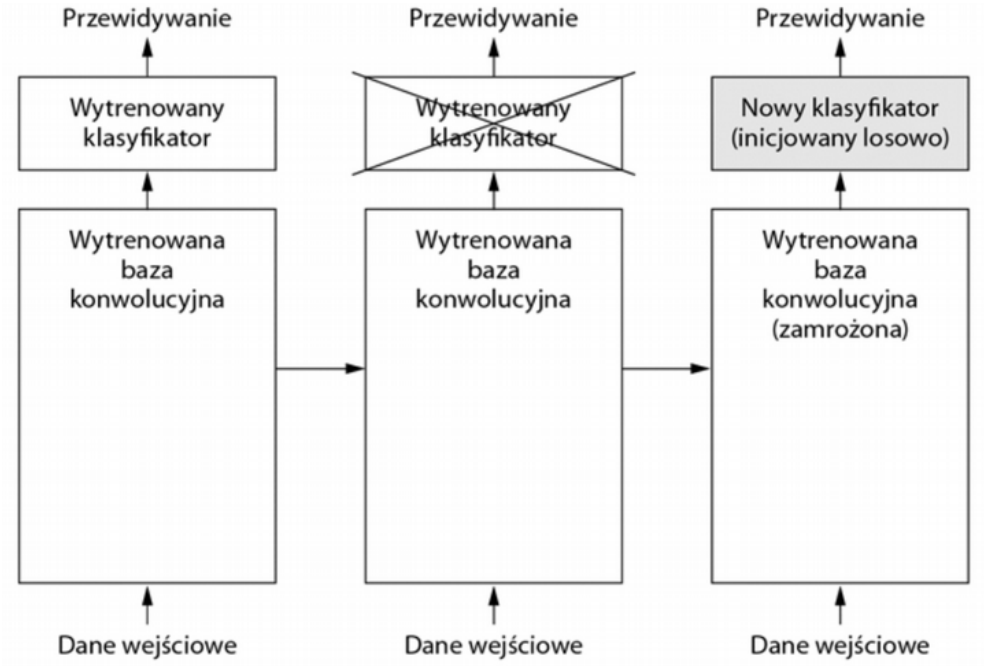
Trenowanie konwolucyjnej sieci neuronowej na małym zbiorze danych

- Często spotykana sytuacja podczas prywatnej pracy nad problemami analizy obrazu,
- *Mała liczba próbek* może oznaczać różną liczbę — od kilkuset do kilkudziesięciu,
- Uczenie głębokie działa głównie wtedy, gdy możliwe jest uzyskanie dostępu do dużej ilości danych,
- Algorytmy tego uczenia mogą samodzielnie wybrać przydatne cechy z treningowego zbioru danych, ale wymagają do tego licznego treningowego zbioru danych.
- Zbiór kilkuset przykładów może okazać się wystarczający, jeżeli model będzie mały i poddany regularyzacji, a zadanie będzie proste.

Technika augmentacji danych

- Nadmierne dopasowanie wynika ze zbyt małej liczby próbek, na których model może się uczyć,
- Augmentacja danych to technika generowania większej liczby elementów treningowego zbioru danych poprzez augmentację próbek na drodze losowych przekształceń zwracających obrazy,
- W Keras augmentacja wspomagana jest przez instancję *ImageDataGenerator*.

Ekstrakcja cech



- Ekstrakcja cech polega na korzystaniu z reprezentacji wyuczonej przez sieć wcześniej w celu dokonania ekstrakcji interesujących nas cech z nowych próbek.
- Cechy te są następnie przepuszczane przez nowy klasyfikator trenowany od podstaw.

Zamrażanie warstwy lub zestawu warstw

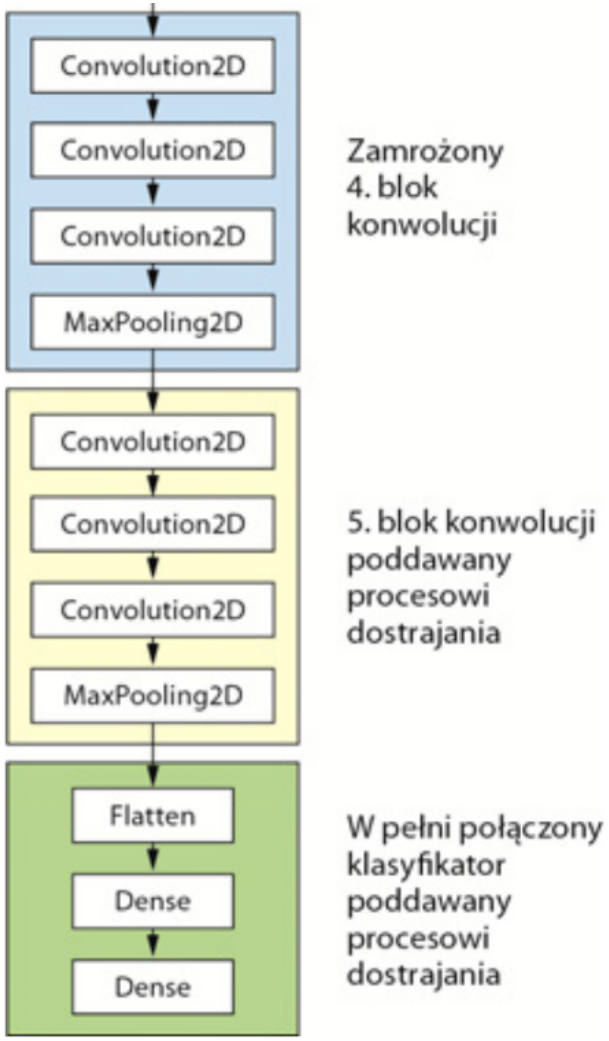
- Zamrażanie warstwy lub zestawu warstw polega na zapobieganiu aktualizacji ich wag w procesie trenowania.
- Be zamrażania reprezentacje wyuczone wcześniej przez bazę konwolucyjną zostaną zmodyfikowane podczas trenowania.
- Warstwy Dense znajdujące się u góry są inicjowane w sposób losowy, co sprawia, że dochodziłoby do dużych zmian wszystkich parametrów sieci, a to skutecznie zniszczyłoby wyuczone wcześniej reprezentacje.
- W pakiecie Keras sieć zamraża się, przypisując wartość *False* atrybutowi *trainable*

Dostrajanie

Dostrajanie jest techniką ponownego stosowania modeli uzupełniającą ekstrakcję cech.

- Polega ona na odmrażaniu kilku górnych warstw zamrożonej bazy modelu używanej do ekstrakcji cech i trenowaniu jej łącznie z nową częścią modelu,
- Najczęściej tą częścią modelu jest w pełni połączony klasyfikator,
- Proces ten określamy mianem dostrajania, ponieważ modyfikuje częściowo wcześniej wytrenowane bardziej abstrakcyjne reprezentacje modelu w celu dostosowania ich do bieżącego problemu.

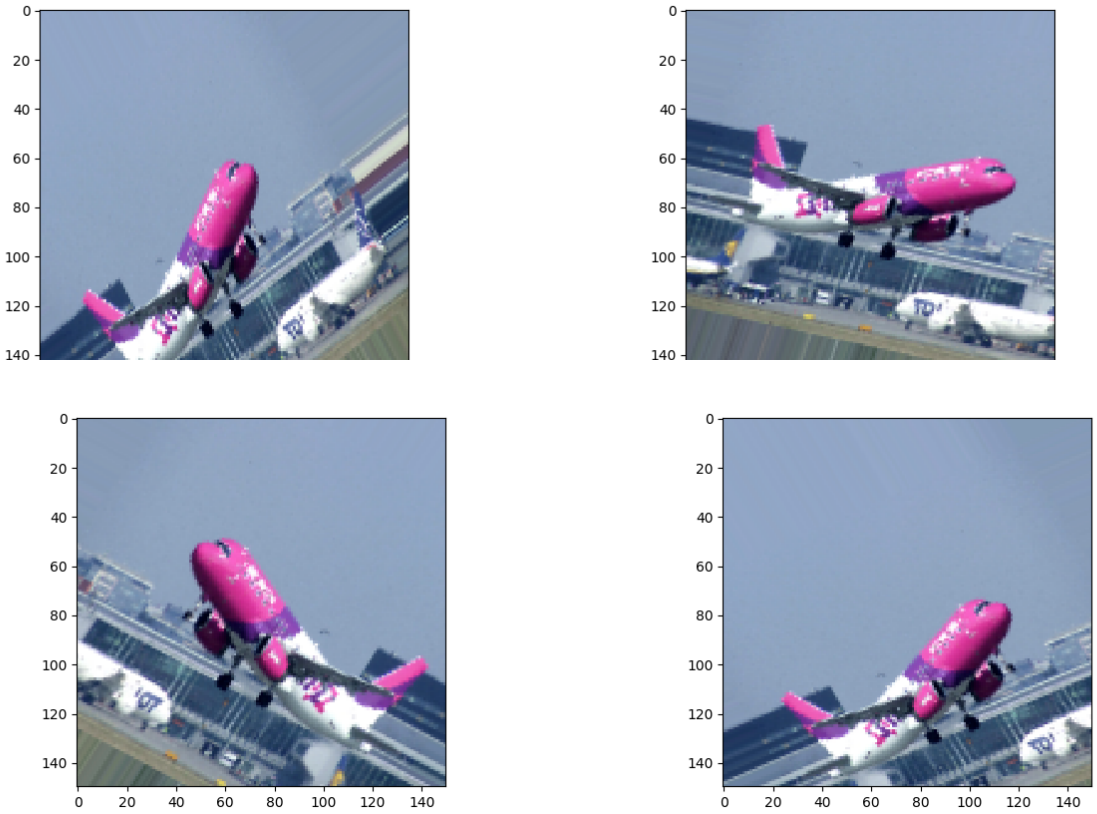
Dostrajanie



Wizualizacja efektów uczenia konwolucyjnych sieci neuronowych

- **Wizualizacja pośrednich danych wyjściowych** - technika przydatna do zrozumienia tego, jak kolejne warstwy sieci konwolucyjnej przekształcają dane wejściowe, i tego, co robią poszczególne filtry sieci.
- **Wizualizacja filtrów sieci konwolucyjnej** - technika umożliwiająca dokładne zrozumienie graficznego wzorca lub graficznej koncepcji, na którą reaguje każdy z filtrów sieci konwolucyjnej.
- **Wizualizacja map ciepła aktywacji klas obrazu** — technika przydatna do określenia części obrazu zidentyfikowanych jako należące do danej klasy

Augmentacja danych



- Polega na tworzeniu nowych przykładów z próbek już istniejących,
- W różnym stopniu nieznacznie przesuwać, obracać i zmieniać rozmiar.

Uczenie głębokie - generowania danych

- Generowanie tekstu za pomocą sieci LSTM,
- DeepDream - technika artystycznego modyfikowania obrazów, która korzysta z reprezentacji wyuczonych przez konwolucyjne sieci neuronowe.

DeepDream

- W algorytmie DeepDream próbuje się maksymalizować aktywację całej warstwy, a nie wybranego filtra, co powoduje jednoczesne mieszanie się wizualizacji wielu cech.
- Generowanie obrazu nie rozpoczyna się od pustego, nieco zaszumianego obrazu wejściowego — na wejściu przetwarzany jest gotowy obraz, co powoduje nanoszenie efektów na utworzony wcześniej obraz,
- Obrazy wejściowe są przetwarzane przy różnych skalach określanych mianem oktaw, co ma na celu poprawienie jakości wizualizacji.

Neuronowy transfer stylu

- Neuronowy transfer stylu polega na zastosowaniu stylu obrazu referencyjnego w celu przetworzenia innego obrazu z zachowaniem jego zawartości.
- Pojęcie **stylu** odnosi się do tekstur, kolorów i sposobu przedstawiania rzeczy widocznych na obrazie.
- **Treścią** określamy wysokopoziomową makrostrukturę obrazu.