

Konwolucyjne sieci neuronowe

dr inż. Sebastian Ernst

Przedmiot: Uczenie Maszynowe

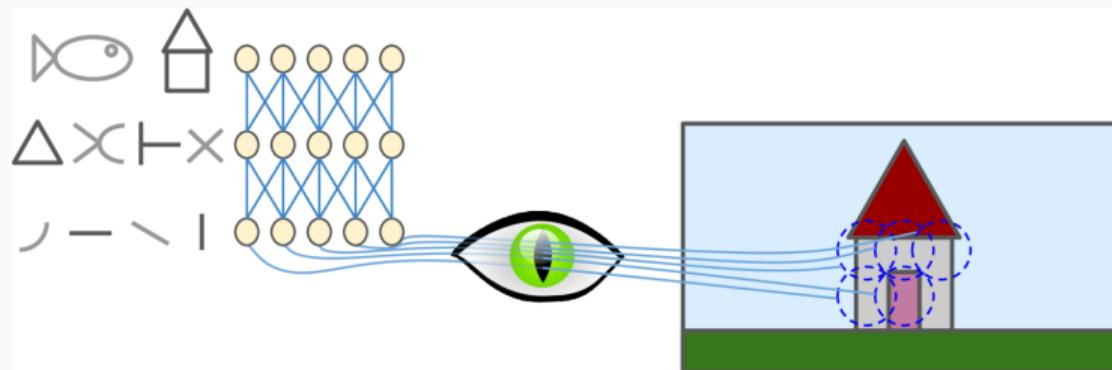
Przetwarzanie obrazów przez sieci konwolucyjne

Zastosowania rozpoznawania obrazów

- Rozwiążuję praktyczne problemy:
 - wyszukiwanie podobnych obrazów
 - pojazdy autonomiczne
 - klasyfikacja zdjęć/video
- Różne zadania:
 - wykrywanie obiektów
 - segmentacja semantyczna

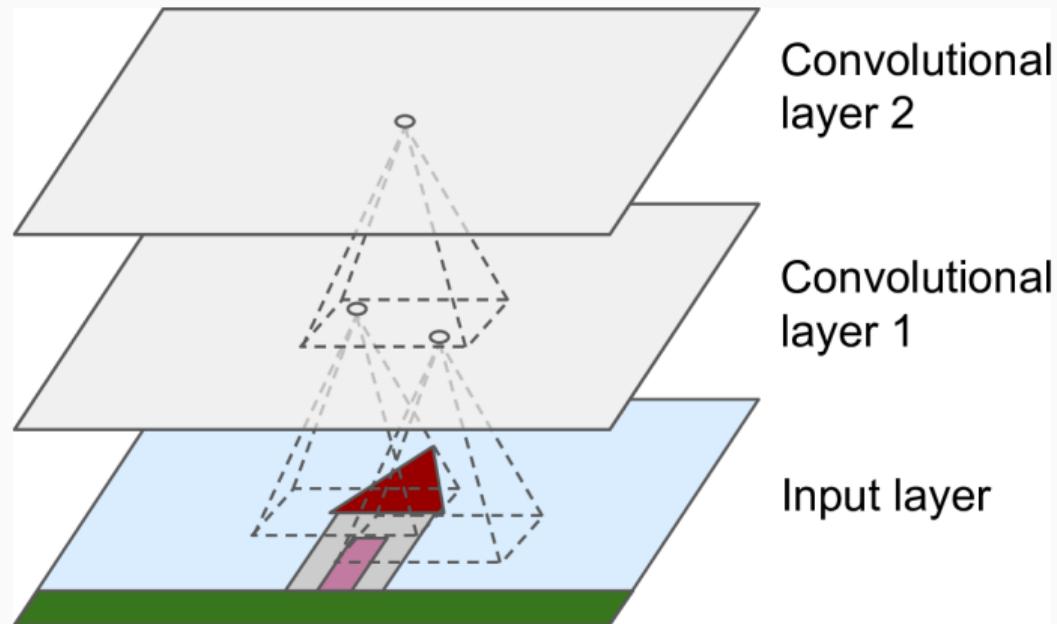
Jak działa kora wzrokowa (w uproszczeniu)

- neurony mają ograniczone pole odbiorcze (*receptive field*)
- wiele neuronów tworzy pełne pole widzenia
- neurony są specjalizowane – mają rozdzielne zadania (np. rozpoznawanie linii poziomych/pionowych)
- warstwy *nie* są gęste
- dlaczego? bo dla obrazów 100x100 (10000 pikseli) warstwa z 1000 neuronów wymagałaby *10 milionów połączeń*



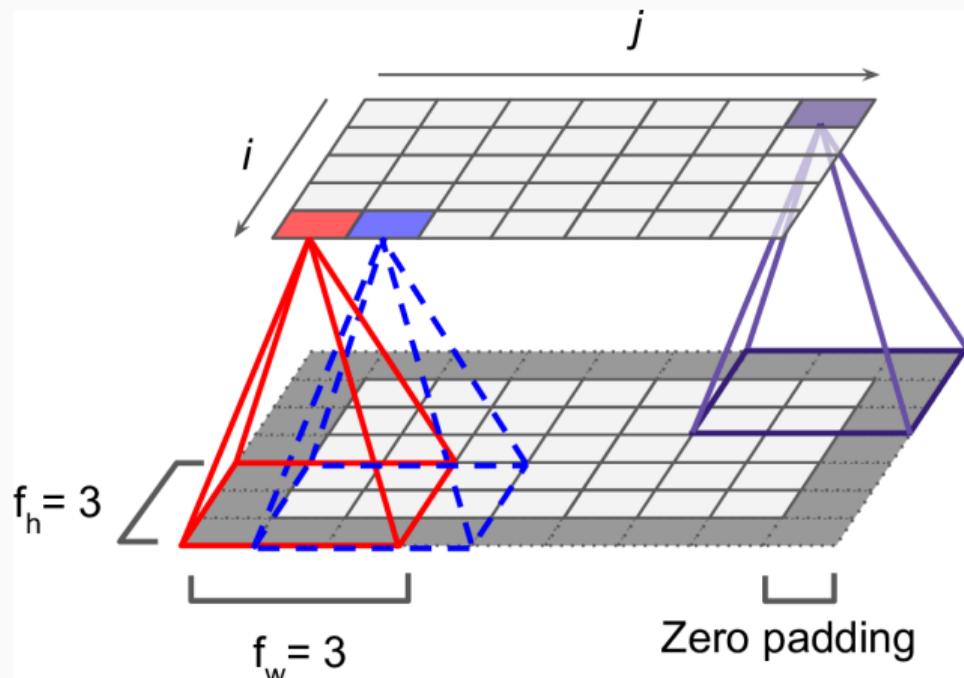
Warstwy konwolucyjne

- warstwy reprezentowane jako macierze (2D), nie wektory (1D)
- neurony pierwszej warstwy „patrzą” na „swoje” obszary w obrazie
- ten schemat powtarzany jest w wyższych warstwach



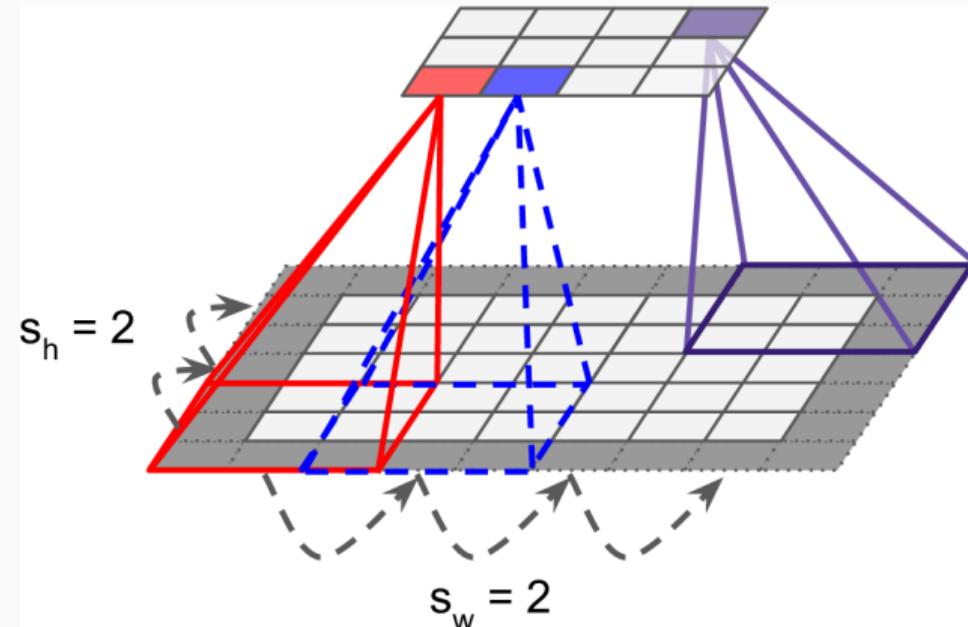
Budowa warstwy konwolucyjnej

- neurony w wierszu i i kolumnie j połączone są z neuronami w wierszach $(i, i + f_h - 1)$ i kolumnach $(j, j + f_w - 1)$ w niższej warstwie
- f_h i f_w to szerokość i wysokość pola odbiorczego
- dodajemy „ramkę” z zer aby utrzymać stały rozmiar warstw



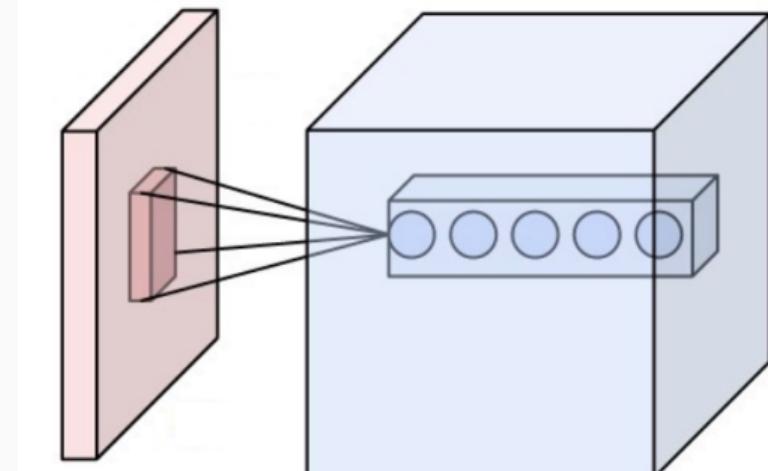
Rozsuwanie pól odbiorczych

- pozwala na połączenie dużej warstwy ze znacznie mniejszą warstwą powyżej
- przesunięcie pomiędzy polami odbiorczymi nazywamy krokiem (*stride*)
- zmodyfikowane wzory:
 - $i \rightarrow (i \times s_h, i \times s_h + f_h - 1)$
 - $j \rightarrow (j \times s_w, j \times s_w + f_w - 1)$



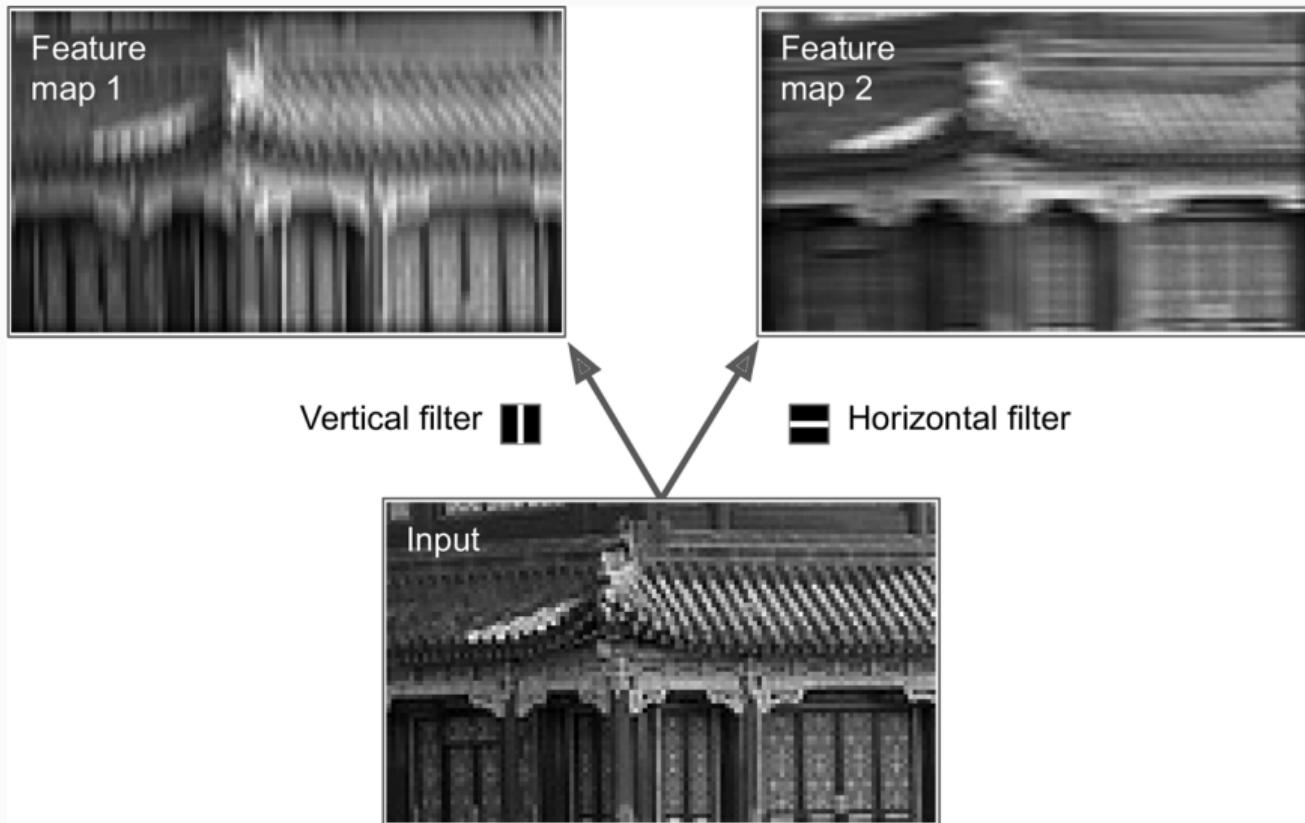
Filtry

- in. jądra konwolucyjne (*convolution kernels*)
- warstwa z neuronami o tym samym filtrze zwraca mapę cech (*feature map*)
- warstwa konwolucyjna sama nauczy się wybierać stosowne filtry



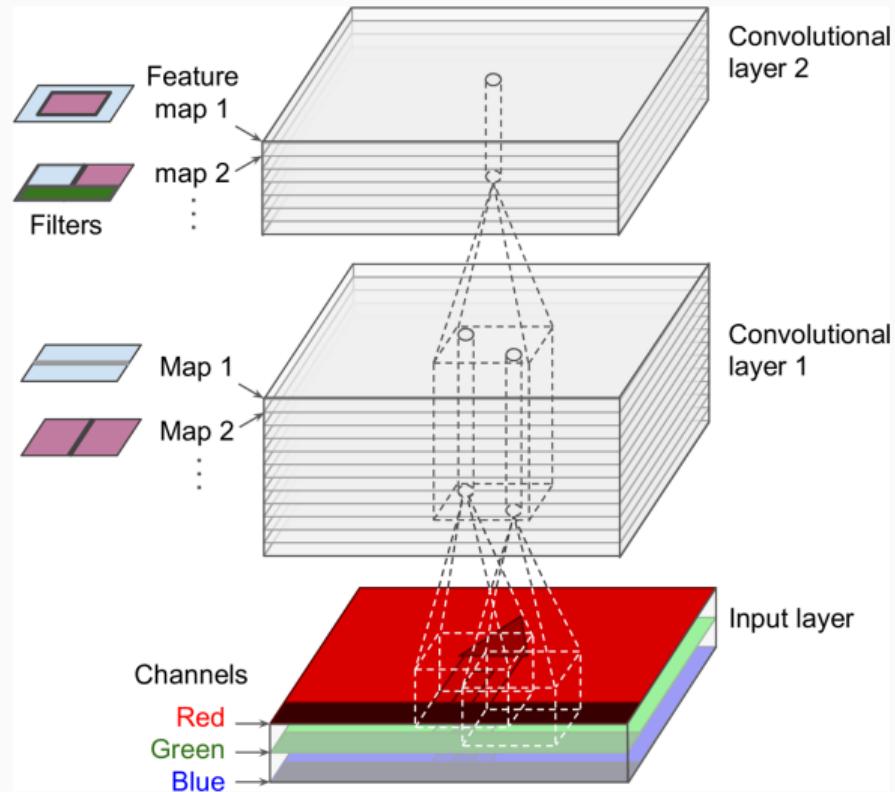
źródło: Wikipedia

Filtры: przykład



Łączenie map cech

- warstwa konwolucyjna ma wiele filtrów, więc *de facto* tworzy strukturę 3D, nie 2D
- każda warstwa składa się z kilku podwarstw (jedna na filtr/mapę cech)
- wszystkie neurony dla danej mapy cech współdzielą te same parametry (wagi i biasy)
- neurony danej mapy cech warstwy wyższej połączone są z odpowiednimi neuronami *wszystkich* map cech warstwy niższej



Funkcja konwolucyjna w TF

```
import numpy as np
from sklearn.datasets import load_sample_image

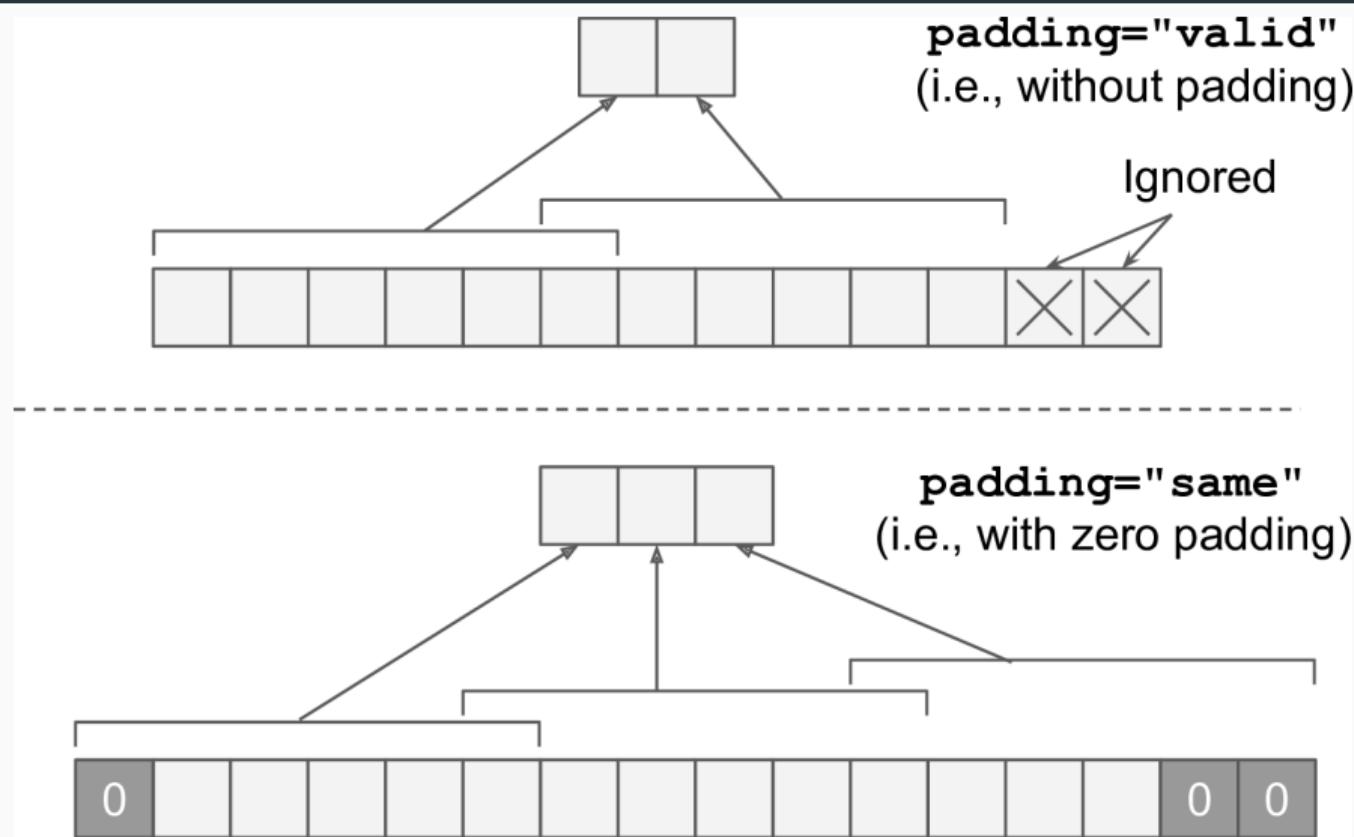
# Load sample images
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1 # vertical line
filters[3, :, :, 1] = 1 # horizontal line

outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")

plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
plt.axis("off") # Not shown in the book
plt.show()
```

Rodzaje wypełnienia (*valid* vs. *same*)



Warstwa konwolucyjna

W praktyce filtry również podlegają uczeniu, a ich liczba i rozmiar stanowią hiperparametry:

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,  
                           padding="same", activation="relu")
```

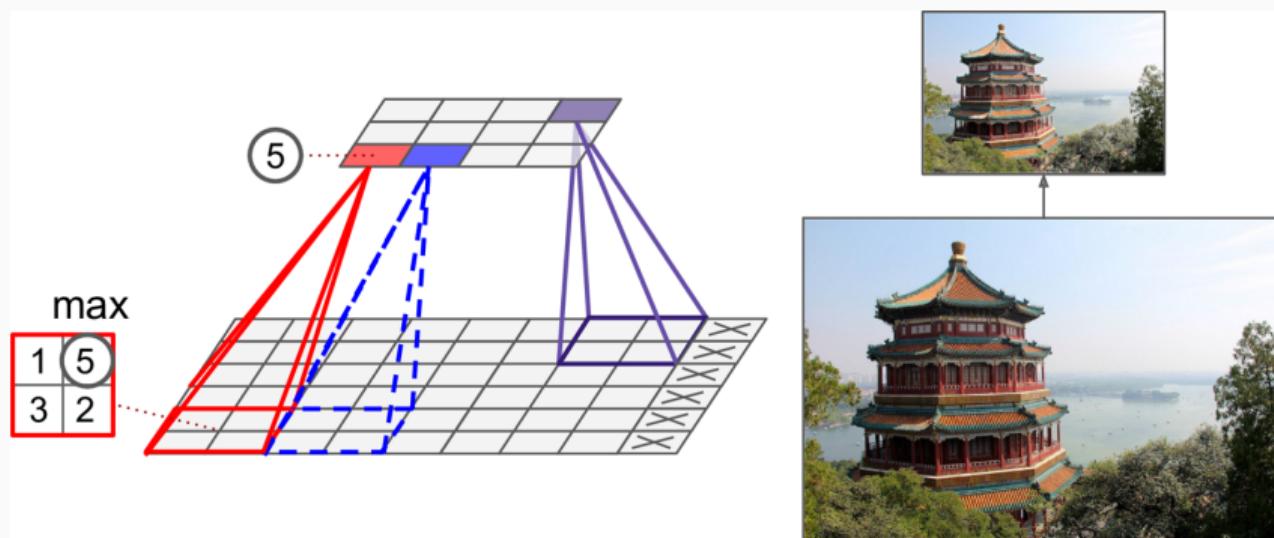
- 32 filtry
- każdy 3×3
- krok = 1, więc nie skalujemy obrazu
- z wypełnieniem (padding="same"), więc obraz nie zostanie też obcięty

Zarządzanie zasobami

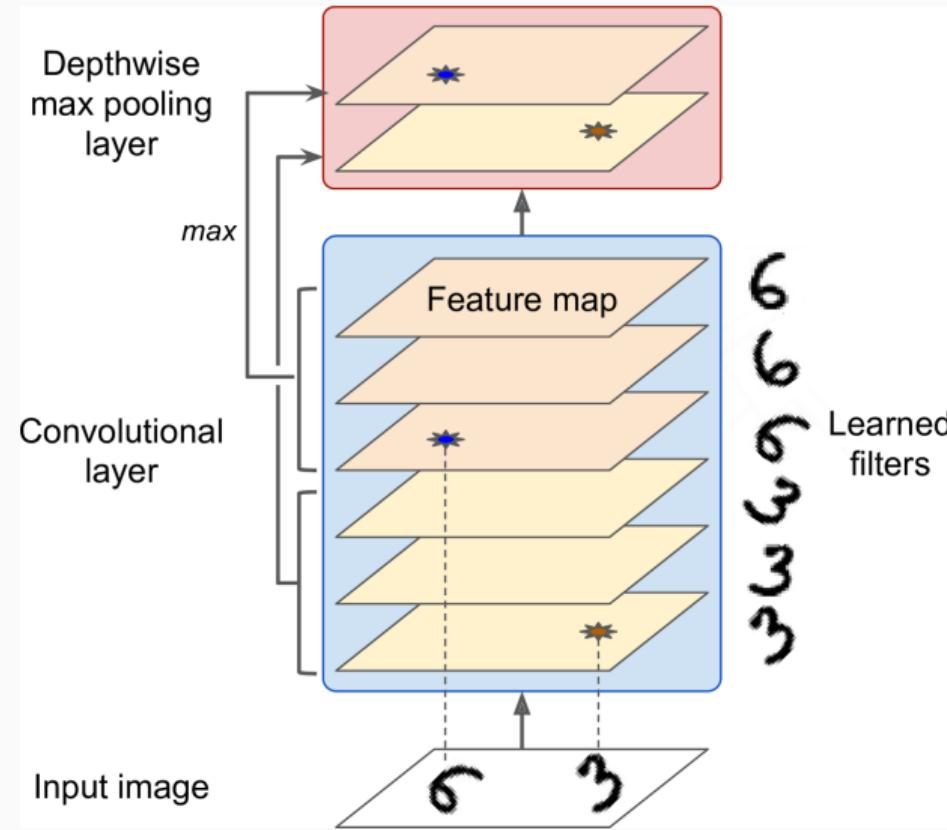
- obrazy RGB 150×100 , 200 filtrów 5×5 , krok=1, z paddingiem
- liczba parametrów: $(5 \times 5 \times 3 + 1) \times 200 = 15200$
- każda z 200 map cech zawiera $150 \times 100 = 15000$ neuronów, z których każdy musi obliczyć ważoną sumę $5 \times 5 \times 3 = 75$ wejść
- razem 225 000 000 operacji mnożenia!
- mapy cech dla jednej instancji potrzebują $200 \times 150 \times 100 \times 32 = 96000000$ bitów
 $= 12$ MB
- $12 \text{ MB} \times 100 \text{ instancji} = 1,2 \text{ GB RAM}$

Warstwy zbierające (*pooling layers*)

- mają na celu zmniejszenie obrazu w celu obniżenia nakładów obliczeniowych, użycia pamięci i liczby parametrów (przeuczenie!)
- podobne do warstw konwolucyjnych, ale *nie mają wag*
- korzystają z funkcji agregacji takich jak średnia czy maksimum



Warstwy zbierające: również wgłąb



Warstwy zbierające w Keras

- prosta implementacja:

```
max_pool = tf.keras.layers.MaxPool2D(pool_size=2)
```

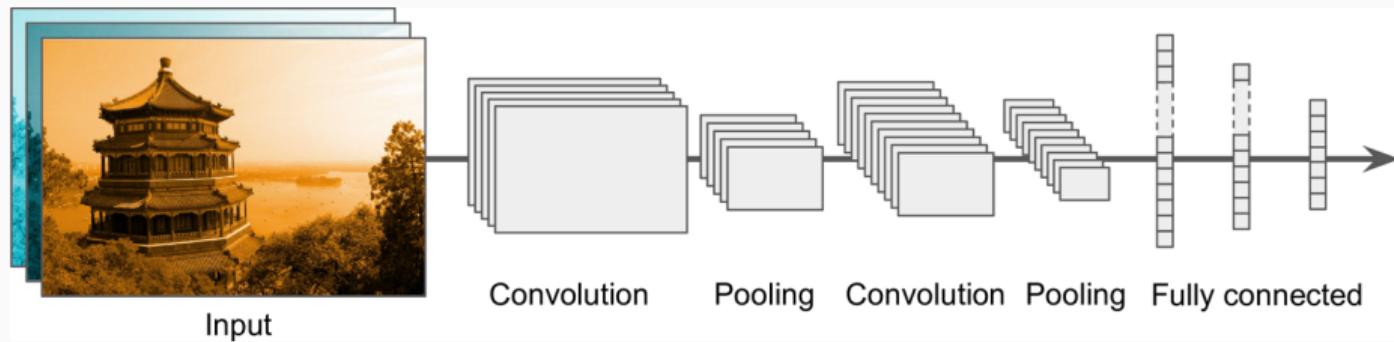
- zbieranie wgłąb – implementacja przy pomocy niskopoziomowej funkcji `max_pool`:

```
tf.nn.max_pool(inputs,  
                ksize=(1, 1, 1, self.pool_size),  
                strides=(1, 1, 1, self.pool_size),  
                padding=self.padding)
```

- funkcję możemy opakować w warstwę typu `Lambda`

Architektury CNN

Architektury CNN



Przykład: Fashion MNIST

- wejście bez kroku, bo obrazy są małe
- im większy obraz, tym więcej struktur C-C-P
- zwiększamy liczbę filtrów, bo w kolejnych warstwach powstają kombinacje prostych cech

```
from functools import partial

DefaultConv2D = partial(keras.layers.Conv2D,
                      kernel_size=3,
                      activation='relu',])
                      padding="SAME")
```

```
model = keras.models.Sequential([
    DefaultConv2D(filters=64, kernel_size=7,
                  input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=128),
    DefaultConv2D(filters=128),
    keras.layers.MaxPooling2D(pool_size=2),
    DefaultConv2D(filters=256),
    DefaultConv2D(filters=256),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Flatten(),
    keras.layers.Dense(units=128, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units=64, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units=10, activation='softmax'),
```

LeNet-5 (1998)

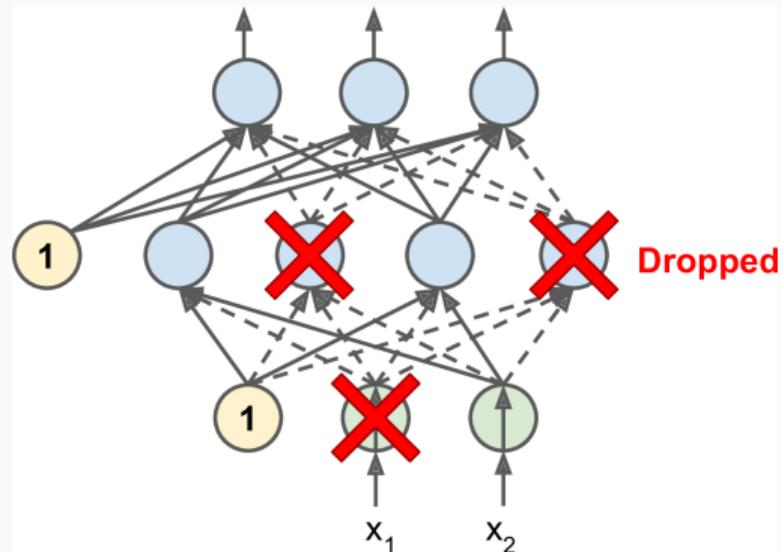
- szeroko stosowana do rozpoznawania odręcznie pisanych cyfr (MNIST)
- padding tylko na pierwszej warstwie, więc potem obraz się zmniejsza
- warstwy zbierające mnożą przez współczynnik i dodają bias (po 1 na mapę, trenowane)
- neurony w C3 połączone z 3–4 warstwami S2
- warstwa wyjściowa mierzy odległość między wektorem wejściowym a wektorem wag
- więcej szczegółów w [artykule źródłowym](#)

AlexNet (2012)

- podobna do LeNet-5, ale większa i głębsza
- po raz pierwszy warstwy konwolucyjne jedna na drugiej
- regularyzacja:
 - dropout 50% na warstwach F8 i F9
 - augmentacja danych
- LRN (*local response normalization*)
 - najmocniej aktywowane neurony „wyłączają” neurony na swojej pozycji w sąsiednich mapach
 - dywersyfikacja rozpoznawanych cech
- artykuł źródłowy

Dropout

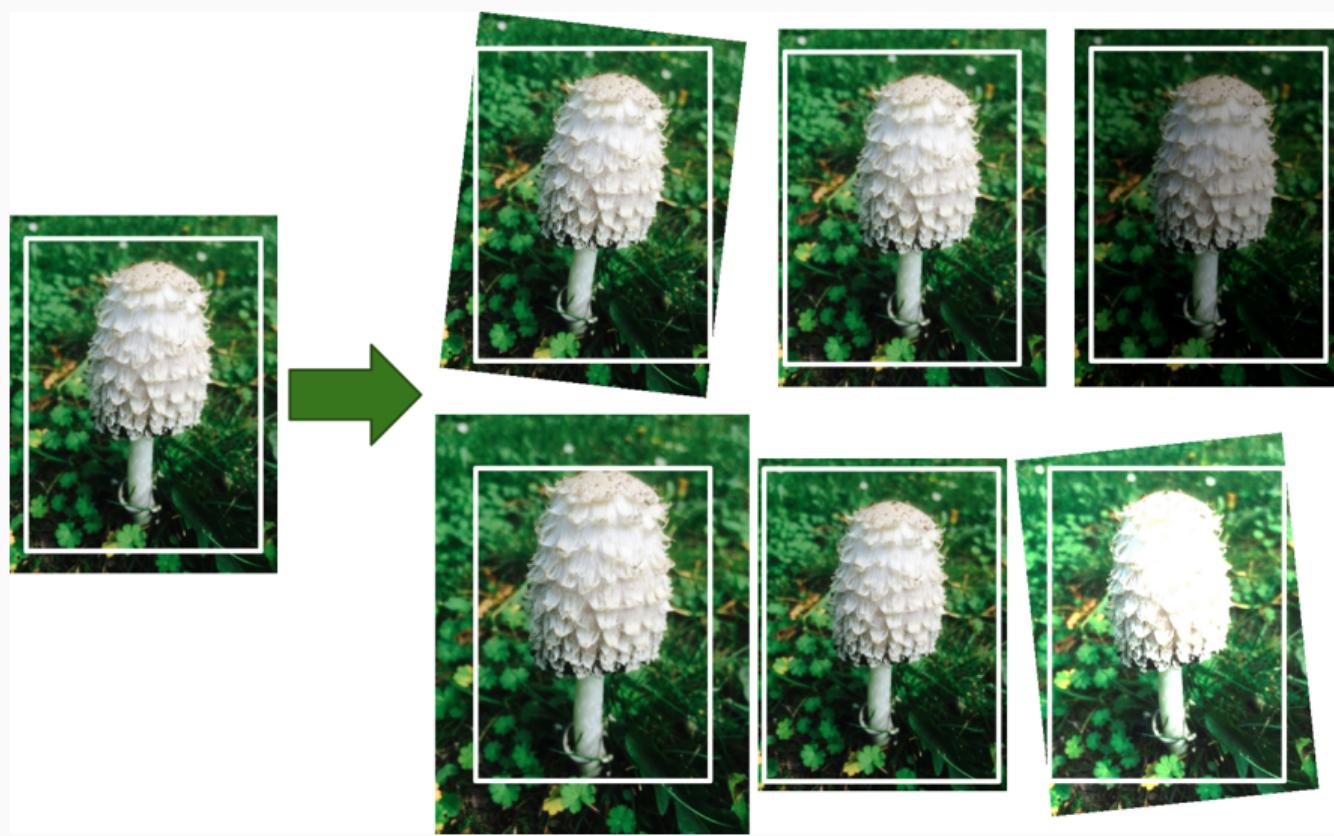
- Metoda zaproponowana i rozwinięta w 2012 i 2014 roku
- W każdym kroku, każdy neuron z prawdopodobieństwem p zostanie opuszczony (ang. *dropped out*)
- Opuszczony neuron jest ignorowany, ale tylko w bieżącym kroku uczenia
- Hiperparametr p (współczynnik opuszczenia – *dropout rate*) ustawiamy na 10–50% (zakres większy w sieciach rekurencyjnych i konwolucyjnych)



Dropout w Keras

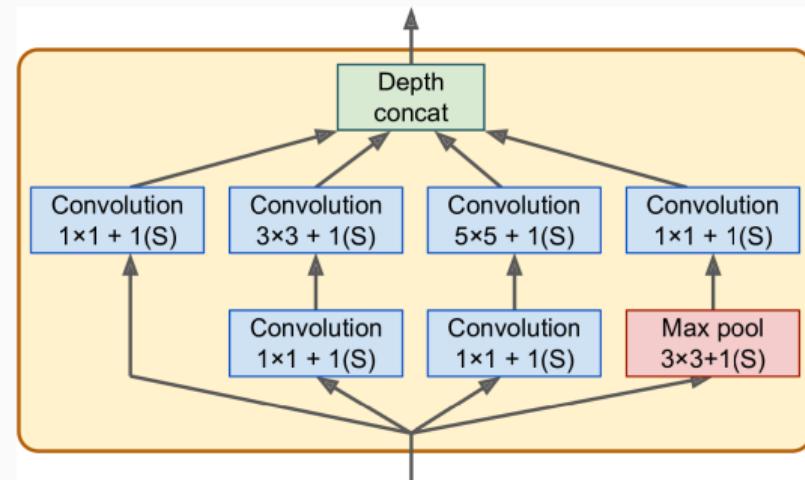
```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu",
                      kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu",
                      kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

Powiększanie danych (*data augmentation*)



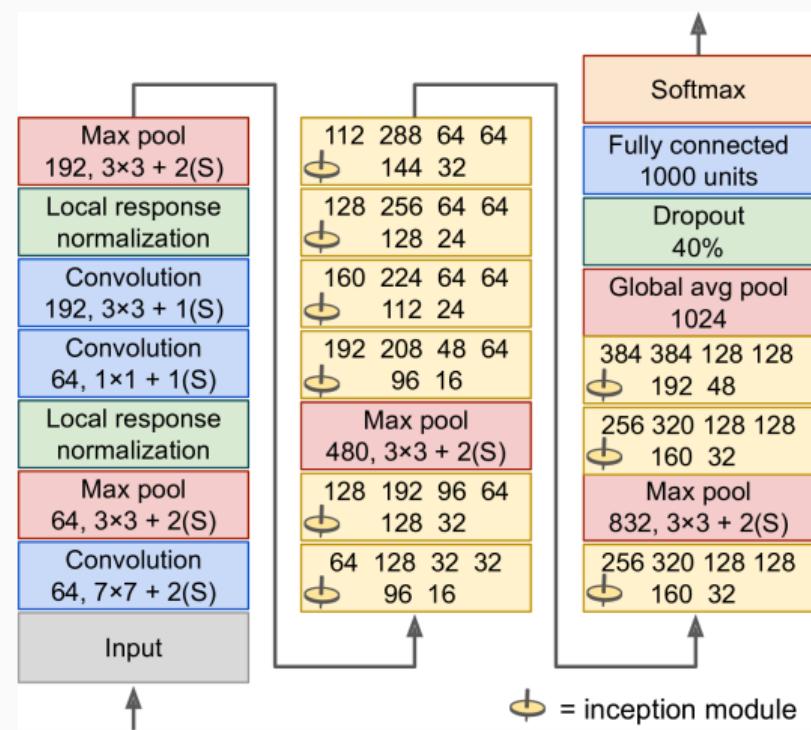
GoogLeNet (2014)

- głębsza od AlexNet, ale 10x mniej parametrów (6M vs 60M) dzięki modułom incepcyjnym (*inception modules*)
- 1×1 , więc nie wykrywają cech przestrzennych, ale mogą wykrywać cechy idąc „wgłąb”
- zmniejszają liczbę map cech, więc redukują wymiarowość (*bottleneck layers*)
- para $[1 \times 1, 3 \times 3]$ lub $[1 \times 1, 5 \times 5]$ może być traktowana jako jedna, silna warstwa konwolucyjna
- artykuł źródłowy



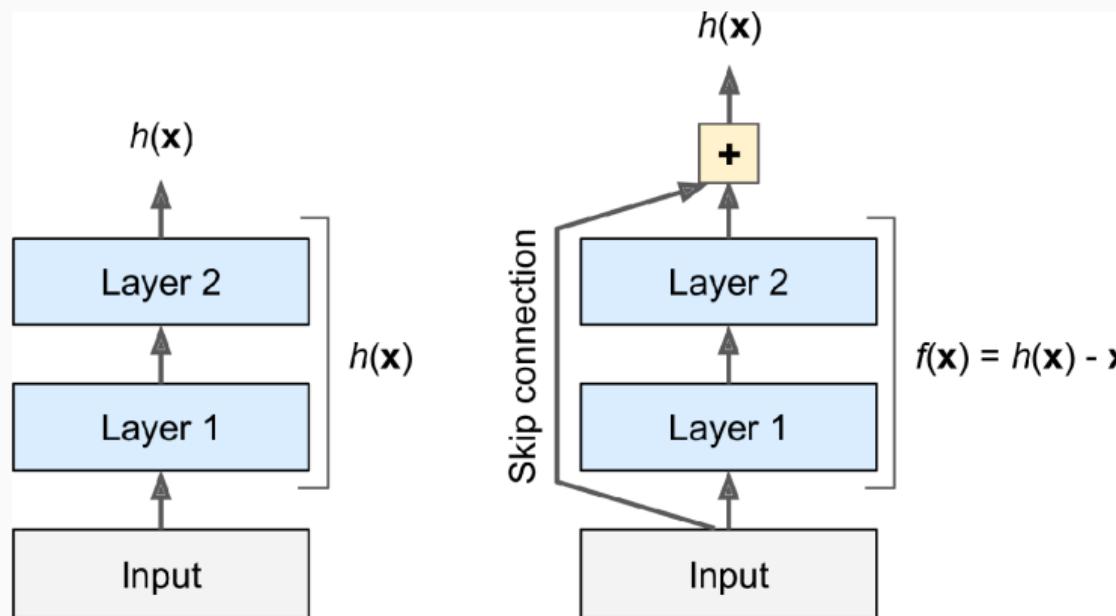
GoogLeNet, c.d.

- pierwsze dwie warstwy: zmniejszenie obrazu, duże filtry aby zachować dużo informacji
- LRN dla dywersyfikacji
- 9 *inception modules*
- *global avg pool* pozbywa się całkiem danych przestrzennych, ale to OK, bo przed nią rozmiar i tak spada z 224×224 do 7×7
- dzięki redukcji rozmiaru nie potrzebujemy warstw gęstych



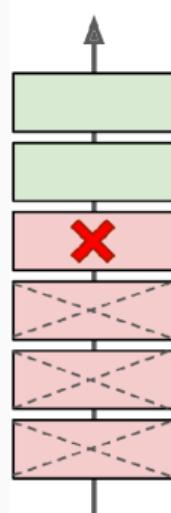
Uczenie rezydualne

- sieci stają się coraz głębsze, ale mają coraz mniej parametrów
- obejście (*skip/shortcut connection*) przyspiesza uczenie sieci – uczenie rezydualne (*residual learning*)



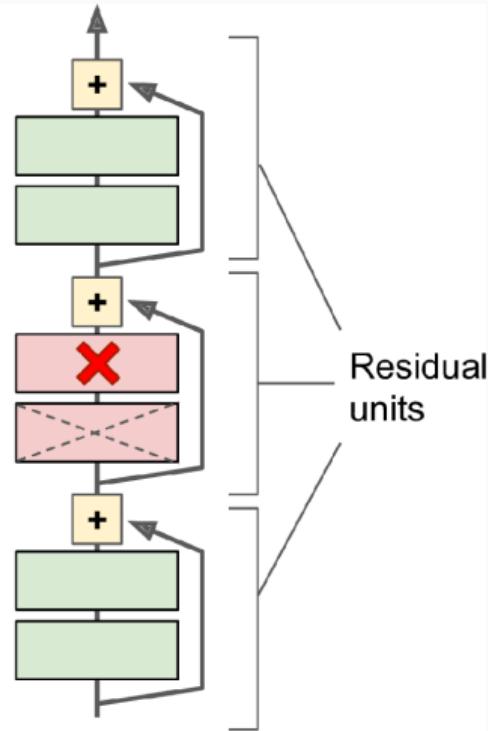
Sieć zwykła i sieć rezydualna

Sieć może robić postępy nawet jeżeli niektóre warstwy nie zaczęły się jeszcze uczyć:



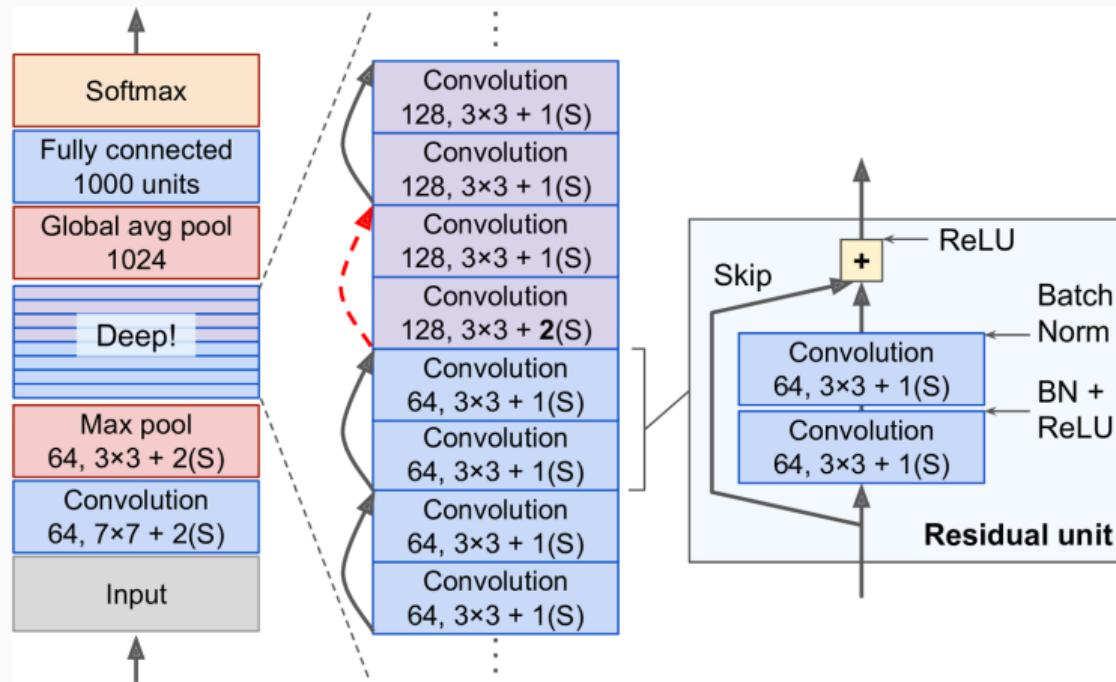
= Layer blocking
backpropagation

= Layer not learning



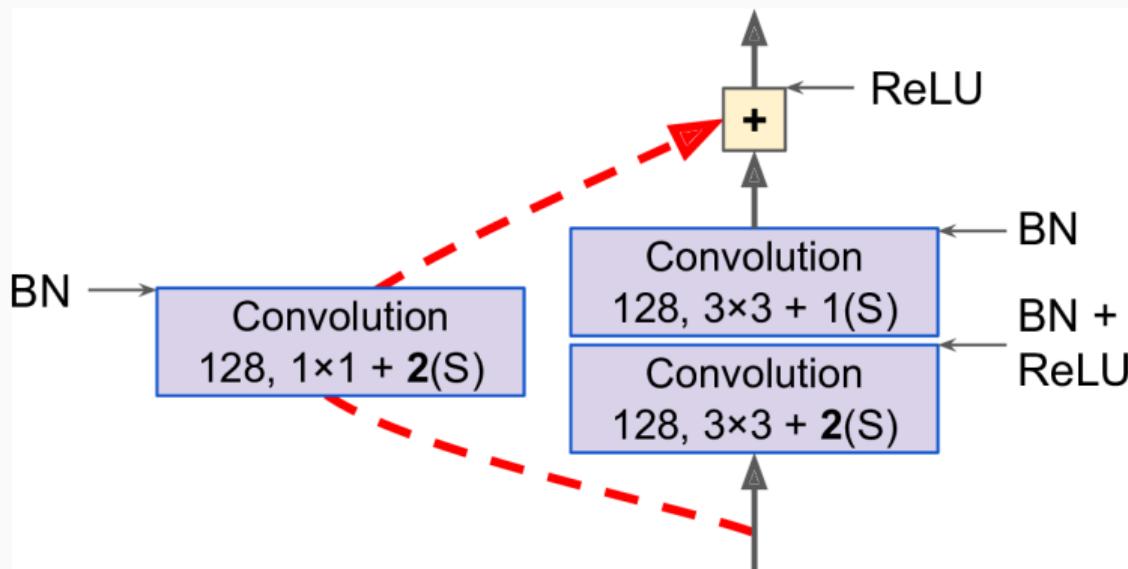
ResNet (2015)

- początek i koniec jak w GoogLeNet, tylko bez dropoutu
- liczba map rośnie co kilka jednostek rezydualnych, ale zarazem zmniejszamy obraz



ResNet: obejście przy kroku > 1

W takich miejscach obejście musi być wyposażone w warstwę konwolucyjną 1×1 :



Warianty ResNet

- ResNet-34: 34 warstwy:
 - 3 RU, 64 map
 - 4 RU, 128 map
 - 6 RU, 256 map,
 - 3 RU, 512 map
- głębsze, jak ResNet-152, zamiast dwóch warstw 3×3 (np. z 256 mapami) używają trzech:
 - 1×1 , 64 mapy (bottleneck)
 - 3×3 , 64 mapy
 - 1×1 , 256 map
- ResNet-152:
 - 3 RU, 256 map
 - 8 RU, 512 map,
 - 36 RU, 1024 mapy
 - 3 RU, 2048 map

ResNet-34 w Keras

```
DefaultConv2D = partial(keras.layers.Conv2D, kernel_size=3, strides=1,
                       padding="SAME", use_bias=False)

class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        (...)

    def call(self, inputs):
        (...)
```

ResNet-34 w Keras

```
class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            DefaultConv2D(filters, strides=strides),
            keras.layers.BatchNormalization(),
            self.activation,
            DefaultConv2D(filters),
            keras.layers.BatchNormalization()]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                DefaultConv2D(filters, kernel_size=1, strides=strides),
                keras.layers.BatchNormalization()]

    def call(self, inputs):
        (...)
```

ResNet-34 w Keras

```
class ResidualUnit(keras.layers.Layer):
    (...)

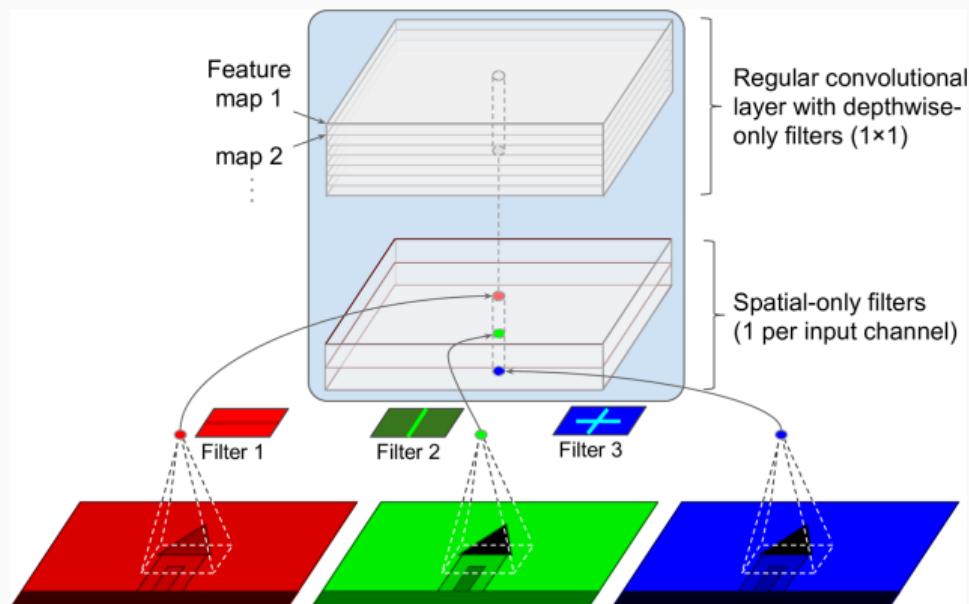
    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)
```

ResNet-34 w Keras

```
model = keras.models.Sequential()
model.add(DefaultConv2D(64, kernel_size=7, strides=2,
                      input_shape=[224, 224, 3]))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation("relu"))
model.add(keras.layers.MaxPool2D(pool_size=3, strides=2, padding="SAME"))
prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters
model.add(keras.layers.GlobalAvgPool2D())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation="softmax"))
```

Xception = Extreme Inception

- wariant GoogLeNet
- oparty też o koncepcje z ResNet
- zastępuje moduły incepcyjne warstwami (*depthwise*) *separable convolution layer*
- zakłada że wozrce przestrzenne i międzykanałowe mogą być modelowane osobno



CNN w praktyce

Wykorzystanie wstępnie przyuczonych sieci

Gotowe sieci dostępne w pakiecie `tf.keras.applications`:

```
model = tf.keras.applications.resnet50.ResNet50(weights="imagenet")
```

Rozmiar obrazów musi być dostosowany do danej sieci

```
images_resized = tf.image.resize(images, [224, 224])
```

Preprocessing dla wybranej sieci i możemy wykonywać predykcje:

```
inputs =
    keras.applications.resnet50.preprocess_input(images_resized * 255)
Y_proba = model.predict(inputs)
```

Uczenie transferowe, tworzenie modelu

Załadowanie modelu bazowego bez górnych warstw i dodanie naszych:

```
base_model = keras.applications.Xception(weights="imagenet",
                                            include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.models.Model(inputs=base_model.input, outputs=output)
```

Uczenie transferowe, wstępne przyuczanie

Blokujemy warstwy modelu bazowego aby ich nie zburzyć w pierwszych krokach uczenia (gdy model daje słabą dokładność):

```
for layer in base_model.layers:  
    layer.trainable = False  
  
optimizer = keras.optimizers.SGD(learning_rate=0.2, momentum=0.9,  
                                 decay=0.01)  
model.compile(loss="sparse_categorical_crossentropy",  
              optimizer=optimizer, metrics=["accuracy"])  
history = model.fit(train_set, validation_data=valid_set,  
                     epochs=5)
```

Model szybko osiąga rozsądную dokładność (75–80%) i przestaje się uczyć.

Uczenie transferowe, uczenie zasadnicze

Umożliwiamy modyfikację wag warstw bazowych:

```
for layer in base_model.layers:  
    layer.trainable = True  
  
optimizer = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9,  
                                 nesterov=True, decay=0.001)  
model.compile(loss="sparse_categorical_crossentropy",  
              optimizer=optimizer, metrics=["accuracy"])  
history = model.fit(train_set,  
                     validation_data=valid_set,  
                     epochs=40)
```

Klasyfikacja i lokalizacja

Lokalizacja jest zadaniem regresyjnym: *znajdź ramkę wokół obiektu.*

Dodajemy drugą gęstą warstwę wyjściową z czterema neuronami:

```
base_model = keras.applications.Xception(weights="imagenet",
                                             include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = keras.layers.Dense(4)(avg)
model = keras.models.Model(inputs=base_model.input,
                           outputs=[class_output, loc_output])
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2],
              optimizer=optimizer, metrics=["accuracy"])
```

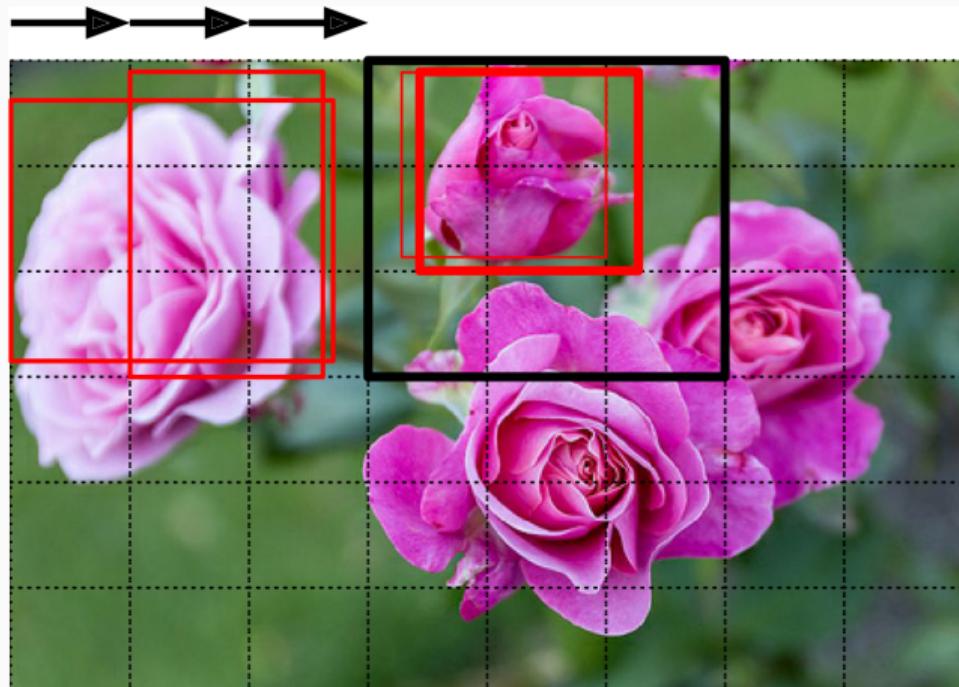
Intersection over Union

- metryka do oceniania jakości predykcji położenia ramki
- stosunek przecięcia ramki właściwej i predykowanej do ich sumy



Wykrywanie obiektów

- tradycyjnie: „przesuwanie” modelu po obrazie
- wymaga stosowania różnych wielkości ramek dla różnych wielkości obiektów
- jeden obiekt może być wykryty kilka razy
- potrzebujemy wyjścia oceniającego czy mamy obiekt w ramce: *objectness*



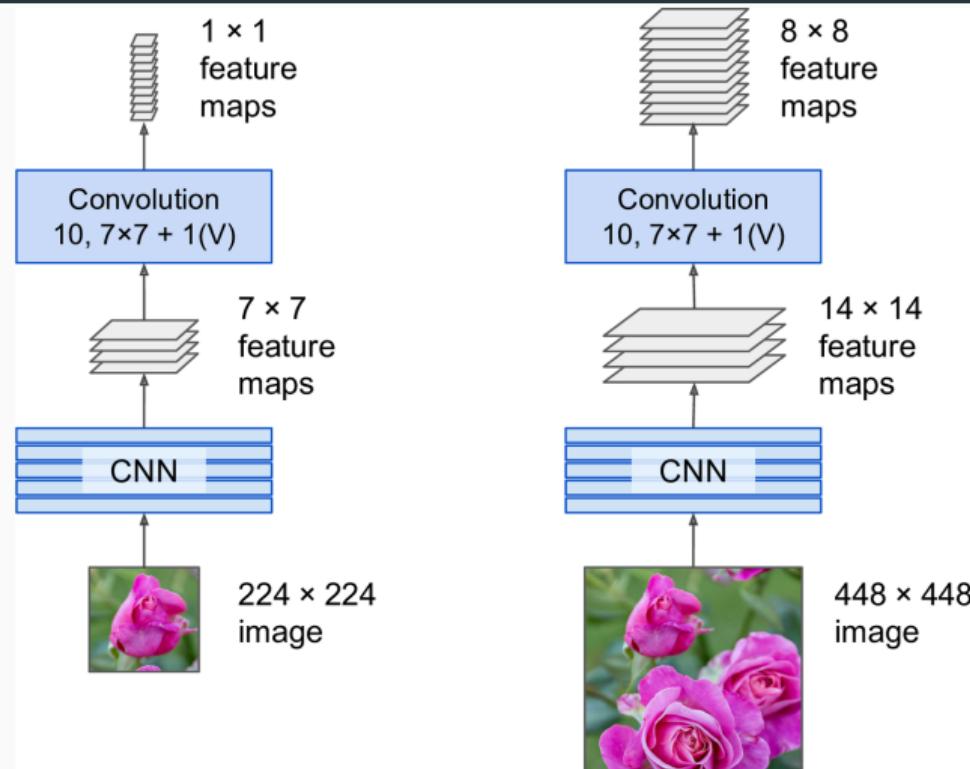
Segmentacja semantyczna

- każdy piksel jest „kolorowany” zgodnie z klasą obiektu, którego jest częścią
- główne wyzwanie: obrazy przechodząc przez CNN stopniowo tracą rozdzielczość



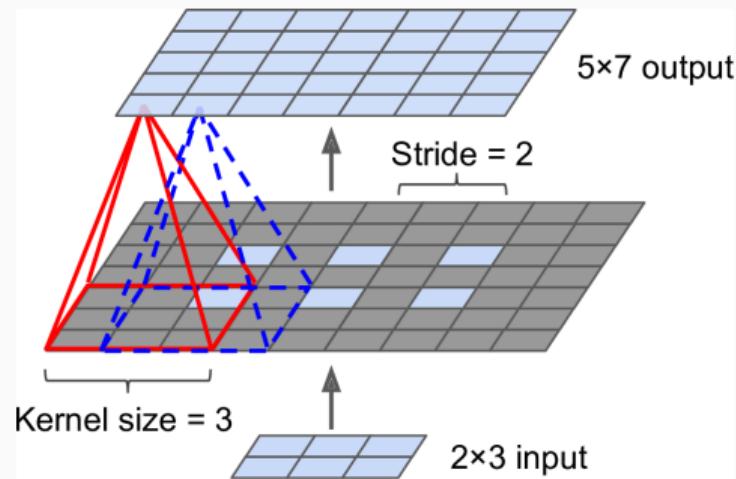
Sieci w pełni konwolucyjne (*fully convolutional networks*)

- warstwy gęste zmieniamy na warstwy konwolucyjne
- nie ma warstw gęstych, więc sieć może przetwarzać obrazy o dowolnych rozmiarach



Segmentacja semantyczna, rozwiążanie

- Jonathan Long et al., 2015
- zaczynamy od CNN którą przerabiamy na FCN
- łączny krok = 32, a więc obrazy 32 razy mniejsze
- powiększamy obraz (*upsampling*) przy pomocy transponowanej warstwy konwolucyjnej



Segmentacja semantyczna, rozwiązań c.d.

- rozwiązań zbyt nieprecyzyjne
- dodajemy obejścia z niższych warstw

