Wydanie II

Deep learning z TensorFlow 2 i Keras dla zaawansowanych

Sieci GAN i VAE, deep RL, uczenie nienadzorowane, wykrywanie i segmentacja obiektów i nie tylko

Rowel Atienza

Helion

Packt

Tytuł oryginału: Advanced Deep Learning with TensorFlow 2 and Keras: Apply DL, GANs, VAEs, deep RL, unsupervised learning, object detection and segmentation, and more, 2nd Edition

Tłumaczenie: Magdalena Tkacz

ISBN: 978-83-283-8883-3

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/delet2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 2

Głębokie sieci neuronowe

W tym rozdziale zajmiemy się głębokimi sieciami neuronowymi. Sieci te wykazują doskonałą wydajnością, jeśli chodzi o dokładność klasyfikacji w bardziej wymagających zbiorach danych, takich jak ImageNet, CIFAR10 (*https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf*) i CIFAR100. Skupimy się tutaj tylko na dwóch sieciach: **ResNet** [2] [4] i **DenseNet** [5]. Zanim omówimy je szczegółowo, ważne jest, aby poświęcić chwilę na zapoznanie się z informacjami wprowadzającymi dotyczącymi sieci tego typu.

Przy okazji omawiania sieci ResNet wprowadzimy koncepcję **uczenia resztkowego** (ang. *residual learning*), umożliwiającego konstruowanie bardzo głębokich sieci dzięki rozwiązaniu problemu **znikających gradientów** (ang. *vanishing gradients*). Tematyka ta jest omówiona w podrozdziale drugim dotyczącym głębokich sieci splotowych.

DenseNet jest ulepszoną wersją ResNet — każda operacja splotu ma bezpośredni dostęp do sygnałów wejściowych i do mapy cech niższych warstw. Jest ona również zaprojektowana tak, by zredukować liczbę parametrów w sieciach głębokich — do tego używane są zarówno **warstwy zwężające** (ang. *bottleneck layer*), jak i **warstwy przekształcające** (ang. *transition layer*).

Dlaczego właśnie te dwa modele, a nie inne? Cóż, od czasu ich wprowadzenia pojawiły się niezliczone modele — takie jak **ResNeXt** [6] i **WideResNet** [7]— zainspirowane technikami wykorzystanymi w tych dwóch sieciach. Jednak dzięki zrozumieniu zarówno *ResNet*, jak i *DenseNet* będziemy mogli wykorzystać zasady stosowane przy ich projektowaniu do budowy własnych modeli. Z kolei dzięki wykorzystaniu techniki **uczenia transferowego** (ang. *transfer learning*) będziemy mogli również wykorzystywać wstępnie wytrenowane modele sieci ResNet i DenseNet do własnych celów, takich jak wykrywanie obiektów i segmentacja. Już te dwa powody (plus zgodność z Keras) sprawiają, że modele te są idealne do odkrywania i uzupełniania wiedzy dotyczącej zaawansowanego uczenia głębokiego, zawartej w tej książce.

Tematyka tego rozdziału dotyczy co prawda głębokich sieci neuronowych, ale zaczniemy od omówienia ważnej własności biblioteki Keras zwanej **funkcyjnym API** (ang. *functional API*). Korzystając z niej, możemy konstruować bardziej złożone modele sieci w *tf.keras*, takie, dla których sekwencyjne API nie jest wystarczające. Powodem, dla którego zwracamy szczególną uwagę na ten interfejs API, jest to, że jest on bardzo przydatnym narzędziem do budowania głębokich sieci — takich jak te, które będziemy omawiać w tym rozdziale. Zalecamy zapoznanie się z treścią rozdziału 1. „Wprowadzenie do zaawansowanego uczenia głębokiego z Keras" przed lekturą tego rozdziału, ponieważ w tym rozdziale będziemy odwoływać się do omówionych już podstawowych koncepcji oraz rozwijać przedstawione wcześniej kody źródłowe, tworząc bardziej zaawansowane modele.

Celem tego rozdziału jest zaznajomienie Czytelnika z:

- Funkcyjnym interfejsem API w Keras, a także analizą przykładowych sieci zbudowanych z jego użyciem.
- Implementacją **głębokich sieci resztkowych** (ang. *Deep Residual Networks*) w *tf.keras* (ResNet wersje 1 i 2).
- Wdrażaniem **gęsto połączonych** (w skrócie: gęstych) **sieci splotowych** (DenseNet) w *tf.keras.*
- Dwoma popularnymi modelami sieci neuronowych wykorzystywanych w uczeniu głębokim: ResNet i DenseNet.

Zacznijmy od omówienia funkcyjnego API.

2.1. Funkcyjne API Keras

Tworząc model w sekwencyjnym interfejsie API (który po raz pierwszy wprowadziliśmy w rozdziale 1. „Wprowadzenie do uczenia głębokiego z Keras"), nakładamy na siebie kolejne warstwy. Generalnie dostęp do modelu uzyskuje się poprzez warstwy wejściowe i wyjściowe. Dowiedzieliśmy się też, że nie ma prostego mechanizmu, jeśli chcemy dodać dodatkowe wejście w środku sieci lub wyjście przed ostatnią warstwą.

Model warstwowy ma też inne wady, na przykład nie obsługuje modeli grafowych ani modeli, które zachowują się jak funkcje języka Python. Ponadto trudno jest „przenieść" gotową warstwę do innego modelu. Tych ograniczeń nie ma API funkcyjne i z tego powodu jest ono ważnym narzędziem dla każdego, kto chce pracować z modelami głębokiego uczenia.

Podstawowe założenia funkcyjnego API są następujące:

- Warstwa jest instancją, która przyjmuje tensor jako argument. Wyjściem warstwy jest inny tensor. Aby zbudować model, łączymy instancje warstw (będące obiektami) ze sobą za pomocą tensorów wejściowych i wyjściowych. Końcowy efekt jest podobny do układania wielu warstw w modelu sekwencyjnym, jednak użycie instancji warstw jest ułatwieniem — w modelu mamy dostęp do danych pomiędzy

warstwami. Mamy więc możliwość użycia pomocniczych wejść i wyjść, korzystania z wielu źródeł danych, uzyskiwania wielu danych wyjściowych, ponieważ dane wejściowe/wyjściowe każdej warstwy są łatwo dostępne.

- Model jest funkcją odwzorowującą jeden lub większą liczbę tensorów wejściowych na tensory wyjściowe. Pomiędzy wejściem i wyjściem modelu tensory łączą instancje poszczególnych połączonych ze sobą kolejnych warstw. Dlatego model możemy uważać za funkcję jednej (lub wielu) warstw wejściowych i jednej (lub wielu) warstw wyjściowych. Instancja modelu formalizuje diagram przedstawiający przepływ danych z wejścia do wyjścia.

Po zakończeniu tworzenia modelu za pomocą funkcyjnego API uczenie i ocena są wykonywane przez te same funkcje, które są używane w API sekwencyjnym. Na przykład, korzystając z interfejsu funkcyjnego API, dwuwymiarową warstwę splotową `Conv2D` z 32 filtrami, *x* jako wejściowym tensorem warstwy i *y* jako tensorem wyjściowym warstwy zapiszemy jako:

```
y = Conv2D(32)(x)
```

Możemy również ułożyć wiele warstw w stos, budując swój model. Na przykład możemy przepisać *implementację sieci CNN* do klasyfikacji cyfr MNIST (patrz listing 1.3), używając funkcyjnego API tak, jak przedstawiono w listingu 2.1.

Listing 2.1. cnn-functional-2.1.py

```
import numpy as np
from tensorflow.keras.layers import Dense, Dropout, Input
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical


# załadowanie zbioru MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# zmiana etykiet z rzadkich na kategorie
num_labels = len(np.unique(y_train))
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# zmiana postaci (ang. shape) i normalizacja obrazów wejściowych
image_size = x_train.shape[1]
x_train = np.reshape(x_train,[-1, image_size, image_size, 1])
x_test = np.reshape(x_test,[-1, image_size, image_size, 1])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# parametry sieci
input_shape = (image_size, image_size, 1)
batch_size = 128
kernel_size = 3
filters = 64
```

```
dropout = 0.3

# konstruowanie warstw CNN z użyciem funkcyjnego API
inputs = Input(shape=input_shape)
y = Conv2D(filters=filters,
           kernel_size=kernel_size,
           activation='relu')(inputs)
y = MaxPooling2D()(y)
y = Conv2D(filters=filters,
           kernel_size=kernel_size,
           activation='relu')(y)
y = MaxPooling2D()(y)
y = Conv2D(filters=filters,
           kernel_size=kernel_size,
           activation='relu')(y)
# przekształcenie obrazu na wektor przed połączeniem do warstwy gęstej — spłaszczenie
y = Flatten()(y)
# regularyzacja — pomijanie
y = Dropout(dropout)(y)
outputs = Dense(num_labels, activation='softmax')(y)

# konstruowanie modelu przez podanie wejść/wyjść
model = Model(inputs=inputs, outputs=outputs)
# model w formie tekstowej
model.summary()

# funkcja straty klasyfikatora, optymalizacja Adam, dokładność
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

# uczenie modelu z obrazami wejściowymi i etykietami
model.fit(x_train,
          y_train,
          validation_data=(x_test, y_test),
          epochs=20,
          batch_size=batch_size)

# dokładność modelu na zbiorze testowym
score = model.evaluate(x_test,
                       y_test,
                       batch_size=batch_size,
                       verbose=0)
print("\nDokładność na zbiorze testowym: %.1f%%" % (100.0 * score[1]))
```

Domyślnie `MaxPooling2D` używa `pool_size=2`, więc argument nie został jawnie podany.

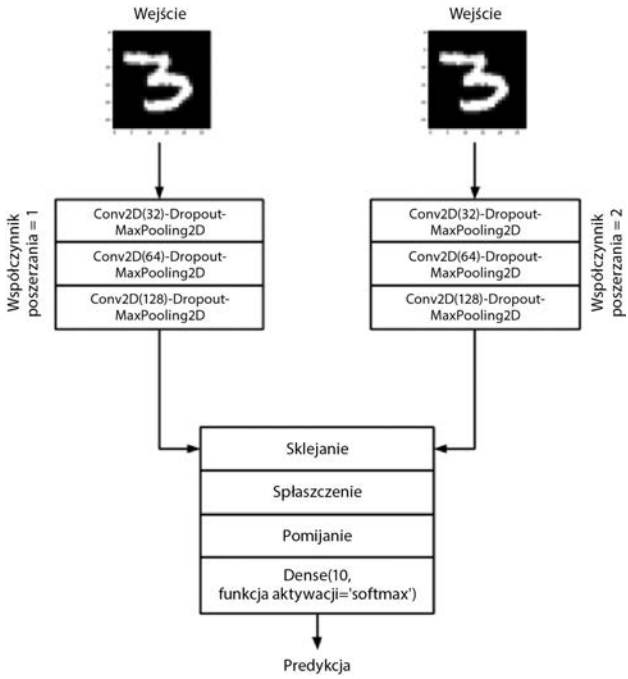
W listingu 2.1 każda warstwa jest funkcją tensora. Każda warstwa generuje tensor jako wyjście, które staje się sygnałem wejściowym dla kolejnej warstwy. Aby utworzyć ten model, możemy wywołać funkcję `Model()` i podać zarówno tensory wejściowe, jak i wyjściowe (lub alternatywnie — listę tensorów). Cała reszta kodu pozostaje bez zmian.

Uczenie i ocena modelu z listingu 2.1 może być wykonane za pomocą funkcji `fit()` i `Evaluation()`, podobnie jak to miało miejsce w modelu sekwencyjnym (klasa `Sequential` jest *de facto* podklasą klasy `Model`). Musimy pamiętać, że w funkcji `fit()` użyliśmy argumentu `validation_data`, aby zobaczyć postęp dokładności walidacji podczas uczenia. Dokładność waha się od 99,3% do 99,4% w 20 epokach.

Tworzenie modelu o dwóch wejściach i jednym wyjściu

Teraz zrobimy coś naprawdę ekscytującego: stworzymy zaawansowany model z dwoma wejściami i jednym wyjściem. Zanim zaczniemy, warto zapamiętać, że sekwencyjny interfejs API jest przeznaczony *wyłącznie* do tworzenia modeli o jednym wejściu i jednym wyjściu.

Załóżmy, że wymyślono nowy model do klasyfikacji cyfr MNIST. Nazwijmy go siecią „Y". Jego schemat pokazano na rysunku 2.1. Model sieci Y używa tych samych danych wejściowych na dwóch wejściach, zarówno w lewej, jak i prawej gałęzi CNN. Następnie model sieci Y łączy wyniki za pomocą **warstwy sklejającej** (ang. *concatenate layer*). Operacja sklejania jest podobna do złożenia dwóch tensorów o tym samym kształcie w stos wzdłuż osi łączenia w celu utworzenia jednego tensora. Na przykład sklejenie dwóch tensorów postaci (3, 3, 16) wzdłuż ostatniej osi da w wyniku tensor postaci (3, 3, 32).

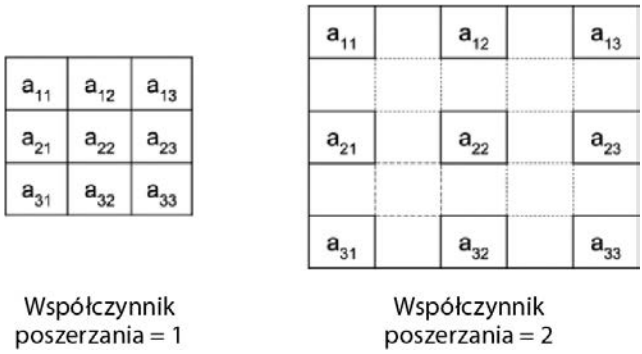


Rysunek 2.1. Sieć Y ma dwa wejścia, na które podawane są te same dane wejściowe, ale przetwarza je w dwóch odrębnych gałęziach sieci splotowych. Wyjścia gałęzi są łączone za pomocą warstwy scalającej. Przewidywanie ostatniej warstwy będzie podobne do modelu klasyfikatora CNN MNIST z poprzedniego rozdziału

Cała reszta modelu po warstwie łączącej pozostanie taka sama jak w modelu klasyfikatora CNN MNIST z poprzedniego rozdziału: najpierw spłaszczenie, potem pomijanie, a następnie warstwa gęsta.

Aby poprawić wydajność modelu z listingu 2.1, możemy wprowadzić kilka zmian. Po pierwsze gałęzie sieci Y podwajają liczbę filtrów, aby zrekompensować zmniejszenie o połowę rozmiaru mapy cech po zastosowaniu funkcji `MaxPooling2D()`. Na przykład, jeśli dane wyjściowe pierwszego splotu to (28, 28, 32), po `MaxPooling2D()` zostanie (14, 14, 32). Filtr następnego splotu będzie miał rozmiar 64 i wymiary wyjściowe (14, 14, 64).

Po drugie, chociaż obie gałęzie mają ten sam rozmiar jądra równy 3, prawa gałąź używa **współczynnika poszerzania** (ang. *dilation rate*) równego 2. Na rysunku 2.2 pokazano wpływ różnych współczynników poszerzania na jądro o rozmiarze 3. Pomysł polega na tym, że dzięki zwiększeniu (przy użyciu współczynnika poszerzania) efektywnej wielkości **pola receptywnego** (ang. *receptive field*) jądra CNN w prawej gałęzi nauczy się innej mapy cech. Użycie współczynnika poszerzania większego niż 1 jest wydajną obliczeniowo, przybliżoną metodą zwiększania rozmiaru pola receptywnego (przybliżoną, ponieważ jądro nie jest w rzeczywistości prawdziwym jądrem). Taka modyfikacja jest efektywna, ponieważ wykonujemy tyle samo operacji, co przy współczynniku poszerzania równym 1.



Rysunek 2.2. Zwiększając współczynnik poszerzania powyżej 1, zwiększamy również efektywny rozmiar pola receptywnego jądra

Aby docenić ideę pola receptywnego, warto zauważyć, że gdy jądro oblicza każdy punkt mapy cech, jego danymi wejściowymi jest łatka w mapie cech poprzedniej warstwy, która jest również zależna od mapy cech z jeszcze poprzedniej warstwy. Jeśli będziemy kontynuować śledzenie tej zależności aż do obrazu wejściowego, okaże się, że jądro będzie zależne od łatki obrazu zwanej polem receptywnym.

Będziemy używać opcji `padding='same'`, by upewnić się że nie będziemy mieć ujemnych wymiarów tensora w momencie używania poszerzania w CNN. Korzystając z opcji `padding='same'`, zachowujemy taki sam wymiar danych wejściowych, jak i mapy cech. Jest to osiągane przez uzupełnianie zerami wejścia (w miarę potrzeby), by upewnić się, że wejście będzie zawsze miało *taki sam* rozmiar.

Listing 2.2 dla *cnn-y-network-2.2.py* przedstawia implementację sieci Y przy użyciu API funkcyjnego. Dwie gałęzie są tworzone w dwóch pętlach `for`. Obie gałęzie oczekują danych wejściowych tej samej postaci. Dwie pętle `for` tworzą dwa 3-warstwowe bliźniacze stosy warstw *Conv2D-Dropout-MaxPooling2D*. Chociaż użyliśmy warstwy łączącej do połączenia danych wyjściowych lewej i prawej gałęzi, moglibyśmy wykorzystać równie dobrze inne funkcje umożliwiające łączenie w *tf.keras*, takie jak `add`, `dot` i `multiply`. Wybór funkcji łączenia nie jest całkowicie arbitralny, ale musi wynikać z przemyślanej decyzji podejmowanej podczas projektowania modelu.

Listing 2.2. cnn-y-network-2.2.py

```
import numpy as np

from tensorflow.keras.layers import Dense, Dropout, Input
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.layers import Flatten, concatenate
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.utils import plot_model

# załadowanie zbioru MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# zmiana etykiet z rzadkich na kategorie
num_labels = len(np.unique(y_train))
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# zmiana postaci (ang. shape) i normalizacja obrazów wejściowych
image_size = x_train.shape[1]
x_train = np.reshape(x_train,[-1, image_size, image_size, 1])
x_test = np.reshape(x_test,[-1, image_size, image_size, 1])
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

# parametry sieci
input_shape = (image_size, image_size, 1)
batch_size = 32
kernel_size = 3
dropout = 0.4
n_filters = 32

# lewa gałąź sieci Y
left_inputs = Input(shape=input_shape)
x = left_inputs
filters = n_filters
# 3 zestawy warstw Conv2D-Dropout-MaxPooling2D
# liczba filtrów podwaja się po każdym zestawie warstw (32-64-128)
for i in range(3):
    x = Conv2D(filters=filters,
               kernel_size=kernel_size,
```

```
                padding='same',
                activation='relu')(x)
    x = Dropout(dropout)(x)
    x = MaxPooling2D()(x)
    filters *= 2

# prawa gałąź sieci Y
right_inputs = Input(shape=input_shape)
y = right_inputs
filters = n_filters
# 3 zestawy warstw Conv2D-Dropout-MaxPooling2D
# liczba filtrów podwaja się po każdym zestawie warstw (32-64-128)
for i in range(3):
    y = Conv2D(filters=filters,
                kernel_size=kernel_size,
                padding='same',
                activation='relu',
                dilation_rate=2)(y)
    y = Dropout(dropout)(y)
    y = MaxPooling2D()(y)
    filters *= 2

# łączymy wyjścia obu gałęzi
y = concatenate([x, y])
# przekształcenie mapy cech na wektor przed połączeniem do warstwy gęstej — spłaszczenie
y = Flatten()(y)
y = Dropout(dropout)(y)
outputs = Dense(num_labels, activation='softmax')(y)

# konstruowanie modelu przez podanie wejść/wyjść przy wykorzystaniu funkcyjnego API
model = Model([left_inputs, right_inputs], outputs)
# weryfikowanie modelu w formie graficznej
plot_model(model, to_file='cnn-y-network.png', show_shapes=True)
# weryfikacja budowy modelu z użyciem opisu warstw w formie tekstu
model.summary()

# funkcja straty klasyfikatora, optymalizacja Adam, dokładność
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

# uczenie modelu z obrazami wejściowymi i etykietami
model.fit([x_train, x_train],
          y_train,
          validation_data=([x_test, x_test], y_test),
          epochs=20,
          batch_size=batch_size)

# dokładność modelu na zbiorze testowym
score = model.evaluate([x_test, x_test],
                       y_test,
                       batch_size=batch_size,
                       verbose=0)
print("\nDokładność na zbiorze testowym: %.1f%%" % (100.0 * score[1]))
```

Łączenie w modelu sieci Y nie spowoduje odrzucenia żadnej części mapy cech. Zamiast tego pozwolimy warstwie gęstej odkryć, co zrobić z połączonymi mapami cech.

Cofając się o krok, możemy zauważyć, że sieć Y oczekuje dwóch serii danych wejściowych: do uczenia i do walidacji. Dane wejściowe są identyczne, stąd w kodzie `[x_train, x_train]`.

Po 20 epokach uczenia dokładność sieci Y waha się od 99,4% do 99,5%. Jest to niewielka poprawa w stosunku do stosu trzech CNN, który osiągnął dokładność w zakresie pomiędzy 99,3% a 99,4%. Odbyło się to jednak kosztem zarówno większej złożoności, jak i ponad dwa razy większej liczby parametrów.

Na rysunku 2.3 przedstawiono architekturę sieci Y tak, jak ją „widzi" Keras — wygenerowaną przez funkcję `plot_model()`.

To kończy nasze wprowadzenie do funkcyjnego API. Warto pamiętać, że celem tego rozdziału jest omówienie budowania głębokich sieci neuronowych, w szczególności sieci ResNet i DenseNet. Dlatego ograniczamy się do opisania funkcyjnego interfejsu API tylko w takim zakresie, jaki jest niezbędny do ich zbudowania — omówienie całego interfejsu API wykracza poza zakres tej książki. Wyjaśniwszy to, przejdźmy do omówienia ResNet.

> W celu uzyskania dalszych informacji na temat funkcyjnego API zajrzyj na *https://keras.io/*.

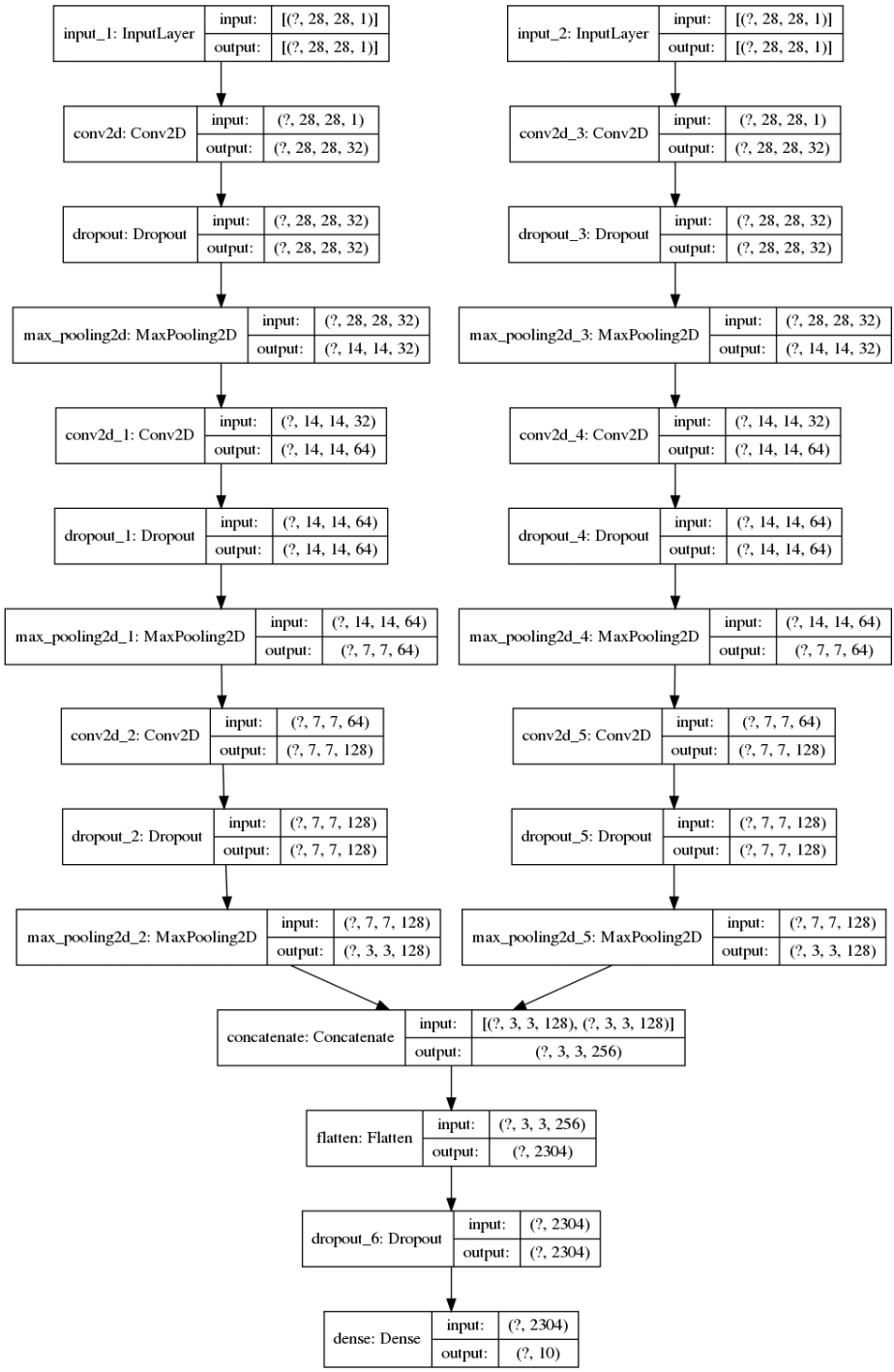
2.2. Głęboka sieć resztkowa (ResNet)

Jedną z kluczowych zalet głębokich sieci jest to, że mają świetną zdolność uczenia się różnych poziomów reprezentacji zarówno na podstawie danych wejściowych, jak i map cech. W klasyfikacji, segmentacji, wykrywaniu i wielu innych problemach z widzeniem komputerowym uczenie się różnych map cech prowadzi generalnie do lepszej wydajności.

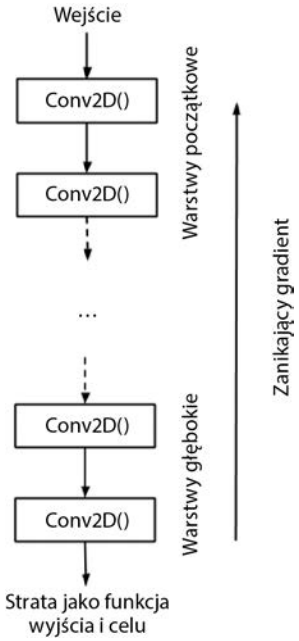
Przekonasz się jednak, że uczenie głębokich sieci nie jest łatwe, ponieważ podczas wstecznej propagacji gradient może zanikać (ang. *vanish*) lub eksplodować (ang. *explode*) w miarę zbliżania się do początkowych warstw. Rysunek 2.4 ilustruje problem zanikania gradientu. Parametry sieci są aktualizowane przez wsteczną propagację z warstwy wyjściowej do wszystkich poprzednich warstw, a ponieważ propagacja wsteczna działa na zasadzie łańcucha, istnieje tendencja do stałego zmniejszania się wartości gradientu w miarę docierania do początkowych warstw. Jest to efektem mnożenia przez siebie małych liczb (zwłaszcza dla małych wartości funkcji straty i małych wartości parametrów).

Liczba operacji mnożenia będzie proporcjonalna do głębokości sieci. Warto również zauważyć, że jeśli wartości gradientu się pogorszą, parametry nie będą odpowiednio aktualizowane.

W konsekwencji nie uzyskamy poprawy wydajności sieci.



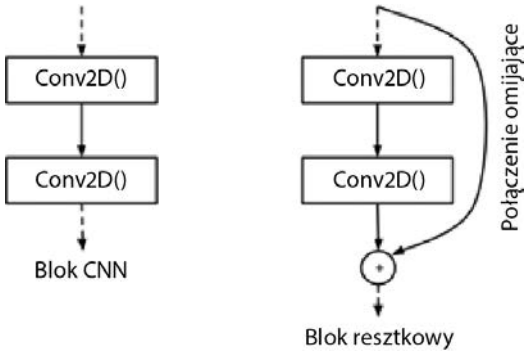
Rysunek 2.3. Architektura sieci Y — CNN zaimplementowana w listingu 2.2



Rysunek 2.4. Częstym problemem w głębokich sieciach jest to, że gradient zanika, gdy dociera do płytkich (początkowych) warstw podczas wstecznej propagacji

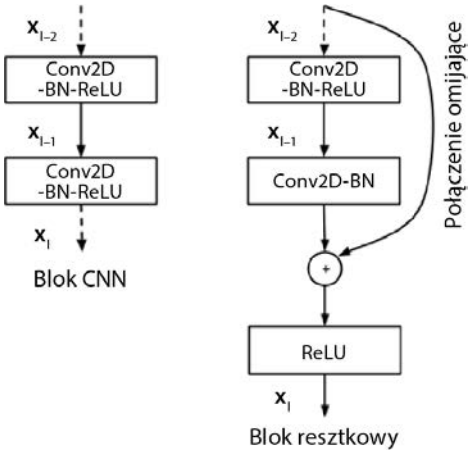
Aby złagodzić degradację gradientu w głębokich sieciach, w sieciach ResNet wprowadzono koncepcję głębokiego **uczenia resztkowego** (ang. *residual learning*). Przeanalizujmy blok: mały segment naszej głębokiej sieci.

Rysunek 2.5 przedstawia porównanie między typowym blokiem CNN a blokiem resztkowym ResNet. Ideą ResNet jest to, że aby zapobiec degradacji gradientu, pozwalamy na przepływ informacji przez **połączenia omijające** (ang. *shortcut connections*), tak aby mogła ona dotrzeć do początkowych warstw.



Rysunek 2.5. Porównanie bloku w typowej sieci CNN z blokiem w ResNet. Aby zapobiec zanikowi gradientu podczas wstecznej propagacji, wprowadzono połączenie skrótowe

Przyjrzyjmy się bardziej szczegółowo różnicom między tymi dwoma blokami. Rysunek 2.6 pokazuje więcej szczegółów bloku CNN innej powszechnie używanej głębokiej sieci VGG [3] i bloku ResNet. Mapy cech warstwy będziemy reprezentować jako x. Mapy cech na warstwie l to x_l. Operacje na warstwie CNN to *Conv2D – Batch Normalization(BN) – ReLU*.



Rysunek 2.6. Szczegółowe operacje na warstwach dla zwykłego bloku CNN i bloku resztkowego

Załóżmy, że reprezentujemy ten zestaw operacji w postaci $H() = Conv2D – Batch\ Normalization(BN) – ReLU$; następnie:

$$x_{l-1} = H(x_{l-2}) \qquad \text{(równanie 2.1)}$$

$$x_l = H(x_{l-1}) \qquad \text{(równanie 2.2)}$$

Innymi słowy, mapy cech w warstwie l-2 są przekształcane do x_{l-1} przez $H() = Conv2D – Batch\ Normalization(BN) – ReLU$. Ten sam zestaw operacji jest stosowany do przekształcenia x_{l-1} w x_l. Innymi słowy, jeśli sieć VGG ma 18 warstw, to mamy 18 operacji $H()$, zanim obraz wejściowy zostanie przetransformowany na 18-warstwową mapę cech.

Ogólnie rzecz biorąc, możemy zauważyć, że na wyjściowe mapy cech warstwy l bezpośrednio wpływają tylko poprzednie mapy cech. Tymczasem dla ResNet:

$$x_{l-1} = H(x_{l-2}) \qquad \text{(równanie 2.3)}$$

$$x_l = ReLU(F(x_{l-1}) + x_{l-2}) \qquad \text{(równanie 2.4)}$$

$F(x_{l-1})$ składa się z *Conv2D-BN*, znanego również jako **mapowanie resztkowe** (ang. *residual mapping*). Znak + oznacza dodawanie tensorów między połączeniem omijającym a wyjściem $F(x_{l-1})$. Połączenie omijające nie powoduje dodania kolejnych parametrów ani nie zwiększa złożoności obliczeniowej.

Operację dodawania można zaimplementować w *tf.keras* za pomocą funkcji scalania `add()`. Jednak zarówno $F(x_{l-1})$, jak i x_{l-2} powinny mieć te same wymiary.

Jeśli wymiary są różne, na przykład przy zmianie rozmiaru mapy cech, należy wykonać rzutowanie liniowe na x_{l-2}, aby dopasować rozmiar $F(x_{l-1})$. W oryginalnym artykule rzutowanie liniowe dla przypadku, gdy rozmiar mapy cech jest zmniejszony o połowę, jest wykonywane przez *Conv2D* z jądrem 1×1 i liczbą kroków `strides = 2`.

W rozdziale 1. „Wprowadzenie do uczenia głębokiego z Keras" powiedziano, że parametr `strides>1` jest równoważny pomijaniu pikseli podczas splotu. Na przykład, jeśli `strides=2`, to możemy pominąć co drugi piksel w trakcie przesuwania jądra podczas procesu splotu.

Zarówno równanie 2.3, jak i 2.4 opisują operacje w blokach resztkowych ResNet. Wynika z nich, że jeśli można wytrenować głębsze warstwy tak, aby miały mniej błędów, to nie ma powodu, dla którego te będące wcześniejszymi miałyby mieć ich więcej.

Znając podstawowe elementy składowe ResNet, jesteśmy w stanie zaprojektować głęboką sieć resztkową do klasyfikacji obrazów. Jednak tym razem zajmiemy się bardziej wymagającym zbiorem danych.

Naszym zbiorem danych będzie tym razem CIFAR10, jeden ze zbiorów, na których przeprowadzano walidację w oryginalnym artykule. *tf.keras* zapewnia wygodny dostęp do zestawu danych CIFAR10, wystarczy dopisać w kodzie:

```
from tensorflow.keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

Podobnie jak w przypadku zbioru MNIST, obrazy w zbiorze CIFAR10 mają 10 kategorii. Dane te to zbiór małych (32×32) zdjęć rzeczywistych obiektów w skali koloru RGB. Obiekty na zdjęciach to: samolot, samochód, ptak, kot, jeleń, pies, żaba, koń, statek i ciężarówka; odpowiadają one każdej z 10 kategorii. Na rysunku 2.7 pokazano przykładowe obrazy ze zbioru CIFAR10.

W zbiorze znajduje się 50 000 obrazów z przypisanymi etykietami w zbiorze uczącym i 10 000 obrazów z przypisanymi etykietami w zbiorze testowym — do walidacji.

W przypadku danych CIFAR10 sieć ResNet można zbudować przy użyciu różnych architektur sieciowych, zestawiono je w tabeli 2.1. Tabela 2.1 oznacza, że mamy trzy zestawy bloków resztkowych. Każdy zestaw ma $2n$ warstw odpowiadających n pozostałym blokom. Dodatkowa warstwa w rozmiarze 32×32 to pierwsza warstwa dla obrazu wejściowego.

Rozmiar jądra wynosi 3, z wyjątkiem przejścia między dwiema mapami cech o różnych rozmiarach, realizującego mapowanie liniowe (na przykład Conv2D o rozmiarze jądra równym 1 i `strides = 2`). Ze względu na spójność z DenseNet będziemy używać określenia „warstwa przekształcająca" (ang. *transition layer*), gdy łączymy dwa bloki resztkowe o różnych rozmiarach.

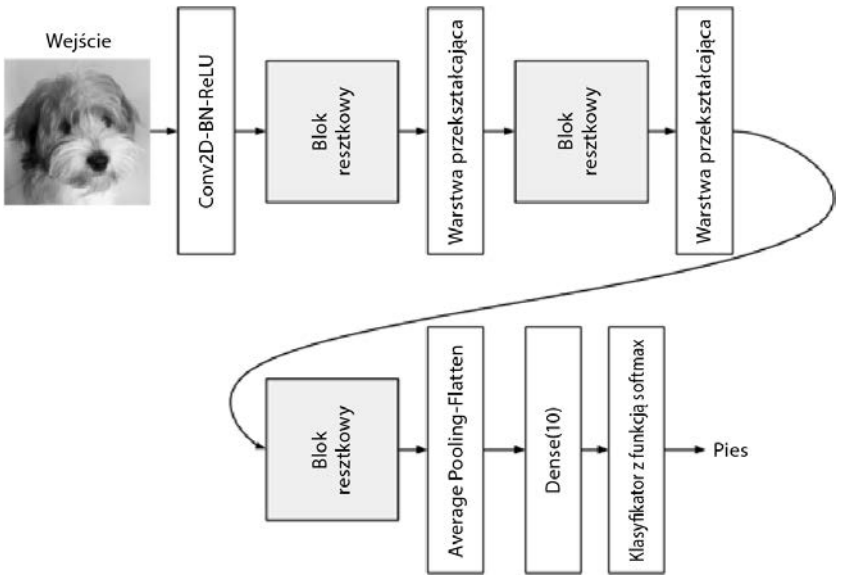
Rysunek 2.7. Przykładowe obrazy ze zbioru danych CIFAR10. Pełny zestaw danych zawiera w zbiorze uczącym 50 000 obrazów z przypisanymi etykietami i 10 000 obrazów w zbiorze testowym do przeprowadzenia walidacji

ResNet używa parametru `kernel_initializer='he_normal'` w celu wspomagania zbieżności podczas wstecznej propagacji błędów [1]. Ostatnia część jest złożona z zestawu warstw *AveragePooling2D – Flatten – Dense*. Warto w tym miejscu zauważyć, że ResNet nie korzysta z pomijania. Wydaje się również, że operacja *dodawanie-scalanie* i *konwolucja 1×1* wykazują efekt samoregulacji. Rysunek 2.8 przedstawia architekturę modelu sieci ResNet dla zbioru danych CIFAR10, zgodnie z opisem w tabeli 2.1.

Poniższy fragment kodu przedstawia częściową implementację ResNet w *tf.keras*. Kod został umieszczony w oficjalnym repozytorium Keras GitHub. Z tabeli 2.2 (zamieszczonej w dalszej części podrozdziału) widać również, że modyfikując wartość n, jesteśmy w stanie zwiększyć głębokość sieci.

Na przykład dla $n = 18$ mamy już ResNet110, sieć głęboką z 110 warstwami. Do budowy ResNet20 używamy $n = 3$:

```
n = 3
# wersja modelu
# oryginalny artykuł: wersja 1 = 1 (ResNet v1),
# ulepszony ResNet: wersja = 2 (ResNet v2)
version = 1
```



Rysunek 2.8. Architektura modelu sieci ResNet dla klasyfikacji zbioru CIFAR10

Tabela 2.1. Konfiguracja architektury modelu sieci ResNet

Warstwy	Rozmiar wyjścia	Rozmiar filtra	Działania
Splot	32×32	16	3×3 *Conv2D*
Blok resztkowy (1)	32×32		$\begin{Bmatrix} 3\times 3 & Conv2D \\ 3\times 3 & Conv2D \end{Bmatrix} \cdot n$
Warstwa przekształcająca (1)	32×32 16×16		{1×1 *Conv2D*, `strides=2`}
Blok resztkowy (2)	16×16	32	$\begin{Bmatrix} 3\times 3 & Conv2D, \text{strides} = 2, \text{jeśli pierwsza } Conv2D \\ 3\times 3 & Conv2D \end{Bmatrix} \cdot n$
Warstwa przekształcająca (2)	16×16 8×8		{1×1 *Conv2D*, `strides = 2`}
Blok resztkowy (3)	8×8	64	$\begin{Bmatrix} 3\times 3 & Conv2D, \text{strides} = 2, \text{jeśli pierwsza } Conv2D \\ 3\times 3 & Conv2D \end{Bmatrix} \cdot n$
Warstwa łączenie — średnia	1×1		8×8 *AveragePooling2D*

```
# obliczanie głębokości z parametru n
if version == 1:
    depth = n * 6 + 2
elif version == 2:
    depth = n * 9 + 2

if version == 2:
    model = resnet_v2(input_shape=input_shape, depth=depth)
else:
    model = resnet_v1(input_shape=input_shape, depth=depth)
```

Polecenie `resnet_v1()` buduje model sieci ResNet. Używana jest funkcja narzędziowa, `resnet_layer()`, aby pomóc zbudować stos warstw *Conv2D-BN-ReLU*.

Odnosi się to do wersji pierwszej. Jak zobaczymy w następnej sekcji, zaproponowano ulepszenia sieci ResNet i tę zmodyfikowaną wersję nazwano ResNet w wersji 2 lub v2. W porównaniu z ResNet ResNet v2 ma ulepszoną konstrukcję bloków resztkowych, co przekłada się na lepszą wydajność.

Listing 2.3 przedstawia część kodu *resnet-cifar10-2.3.py*, który odpowiada implementacji modelu sieci ResNet v1 w *tf.keras*.

Listing 2.3. resnet-cifar10-2.3.py

```
def resnet_v1(input_shape, depth, num_classes=10):
    """Konstruktor sieci ResNet w wersji 1 [a]

    Stos 2 x (3x3) Conv2D-BN-ReLU
    Ostatnia warstwa ReLU jest po połączeniu skrótowym.
    Na początku każdego etapu rozmiar mapy cech jest zmniejszany o połowę
    (ang. downsampled) przez warstwę splotową z krokiem 2 tak długo, jak długo
    jest spełniony warunek. Na każdym etapie warstwy mają taką samą liczbę filtrów.
    Rozmiar mapy cech:
    etap 0: 32x32, 16
    etap 1: 16x16, 32
    etap 2: 8x8, 64
    Liczba parametrów jest w przybliżeniu równa pokazanym w tabeli 6 dla [a]:
    ResNet20 0.27M
    ResNet32 0.46M
    ResNet44 0.66M
    ResNet56 0.85M
    ResNet110 1.7M

    Argumenty:
        input_shape (tensor): postać tensora obrazu wejściowego
        depth (int): liczba warstw splotowego jądra bazowego
        num_classes (int): liczba klas (CIFAR10 ma 10 klas)

    Zwraca:
        model (Model): instancję modelu Keras
    """
    if (depth - 2) % 6 != 0:
        raise ValueError('głębokość powinna wynosić 6n+2 (tzn. 20, 32, w [a])')
    # początek definicji modelu
    num_filters = 16
```

```
    num_res_blocks = int((depth - 2) / 6)

    inputs = Input(shape=input_shape)
    x = resnet_layer(inputs=inputs)
    # stworzenie instancji stosu jednostek resztkowych
    for stack in range(3):
        for res_block in range(num_res_blocks):
            strides = 1
            # pierwsza warstwa (ale nie stos)
            if stack > 0 and res_block == 0:
                strides = 2  # zmniejszanie rozmiaru (ang. downsample)
            y = resnet_layer(inputs=x,
                             num_filters=num_filters,
                             strides=strides)
            y = resnet_layer(inputs=y,
                             num_filters=num_filters,
                             activation=None)
            # pierwsza warstwa (ale nie stos)
            if stack > 0 and res_block == 0:
                # projekcja liniowa skrótu resztkowego
                # połączenie — dopasowanie zmienionych wymiarów
                x = resnet_layer(inputs=x,
                                 num_filters=num_filters,
                                 kernel_size=1,
                                 strides=strides,
                                 activation=None,
                                 batch_normalization=False)
            x = add([x, y])
            x = Activation('relu')(x)
        num_filters *= 2

    # dodanie klasyfikatora na górze
    # v1 nie używa BN po ostatnim połączeniu skrótowym - ReLU
    x = AveragePooling2D(pool_size=8)(x)
    y = Flatten()(x)
    outputs = Dense(num_classes,
                    activation='softmax',
                    kernel_initializer='he_normal')(y)

    # stworzenie instancji modelu
    model = Model(inputs=inputs, outputs=outputs)
    return model
```

Wydajność ResNet w zależności od różnych wartości *n* przedstawiono w tabeli 2.2.

Istnieje kilka drobnych różnic w stosunku do oryginalnej implementacji ResNet. W szczególności nie używamy SGD, zamiast tego użyliśmy *Adam*. Dzieje się tak, ponieważ łatwiej osiągniemy zbieżność ResNet z *Adam*. Użyjemy również harmonogramu dla współczynnika uczenia (*lr*), `lr_ schedule()`, aby zaplanować jego spadek w epokach nr 80, 120, 160 i 180 (z domyślnego *1e-3*). Funkcja `lr_schedule()` będzie wywoływana po każdej epoce podczas uczenia jako część zmiennej w wywołaniach zwrotnych.

Tabela 2.2. Walidacja architektur ResNet na zbiorze CIFAR10 dla różnych wartości n

Warstwy	n	% dokładność na CIFAR10 (oryginalny artykuł)	% dokładność na CIFAR10 (ta książka)
ResNet20	*3*	91,25	92,16
ResNet32	*5*	92,49	92,46
ResNet44	*7*	92,83	92,50
ResNet56	*9*	93,03	92,71
ResNet110	*18*	93,57	92,65

Drugie wywołanie zwrotne zapisuje punkt kontrolny za każdym razem, gdy nastąpi postęp w dokładności walidacji. Podczas trenowania głębokich sieci dobrą praktyką jest zapisywanie modelu (lub punktu kontrolnego dla wag), ponieważ trenowanie głębokich sieci zajmuje dużo czasu.

Gdy zechcesz ponownie skorzystać z sieci, wystarczy po prostu załadować odpowiedni punkt kontrolny, a wytrenowany model zostanie przywrócony. Można to osiągnąć przez użycie *tf.keras* `load_model()`. Dołączona jest też funkcja `lr_reducer()`. W przypadku gdy wartości wybranej metryki przestają zmieniać się przed zaplanowaną redukcją współczynnika, to wywołanie zwrotne zmniejszy szybkość uczenia o pewien czynnik podany w argumencie, jeśli utrata walidacji nie poprawi się po upłynięciu 5 epok (`patience = 5`).

Zmienna *wywołań zwrotnych* jest przekazywana jako argument, gdy wywoływana jest metoda `model.fit()`. Podobnie jak w oryginalnym artykule, implementacja w *tf.keras* wykorzystuje przekształcone dane (ang. *data augmentation*), `ImageDataGenerator()` w celu pozyskania dodatkowych danych uczących jako część procesu regularyzacji. Wraz ze wzrostem dostępnych danych uczących zdolność sieci do uogólniania ulegnie poprawie.

Prostym przekształceniem danych jest na przykład odbicie lustrzane zdjęcia psa, co pokazano na rysunku 2.9 (`horizontal_flip = True`). Jeśli zdjęcie przedstawiało psa, to po jego odbiciu w lustrze nadal mamy zdjęcie psa. Można także wykonać inne przekształcenia, takie jak skalowanie, rotacja, wybielanie itd., a etykieta obrazu nadal pozostanie taka sama.

Rysunek 2.9. Proste dogenerowanie danych przez odbicie lustrzane w pionie oryginalnego obrazu

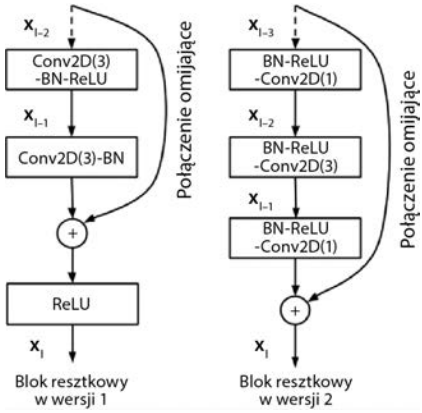
Pełny kod jest dostępny w archiwum pod adresem *https://ftp.helion.pl/przyklady/delet2.zip*.

Często trudno jest dokładnie odtworzyć implementację z oryginalnej publikacji. W tej książce zastosowaliśmy inny optymalizator i inne przekształcenia danych. Może to skutkować niewielkimi różnicami w wydajności ResNet zaimplementowanej w tej książce z użyciem *tf.keras* i modelu w oryginalnym artykule.

Po pojawieniu się drugiego artykułu na temat sieci ResNet [4] przyjęto nazywać oryginalny model przedstawiony w tej sekcji jako ResNet v1. Poprawiony ResNet jest powszechnie nazywany ResNet v2 — omówimy go w następnym podrozdziale.

2.3. ResNet v2

Udoskonalenia zastosowane w ResNet v2 dotyczą głównie sposobu rozmieszczenia warstw w bloku resztkowym, jak pokazano na rysunku 2.10.



Rysunek 2.10. Porównanie bloków resztkowych pomiędzy sieciami ResNet v1 i ResNet v2

Najważniejsze zmiany w ResNet v2 to:

- Użycie stosu warstw *BN-ReLU-Conv2D* w postaci $1\times1 - 3\times3 - 1\times1$.
- Normalizacja wsadowa (BN) i aktywacja ReLU pojawiają się przed dwuwymiarowym splotem.

ResNet v2 jest również zaimplementowany w tym samym kodzie *resnet-cifar10-2.3.py*, co widać na listingu 2.4:

Listing 2.4. resnet-cifar10-2.3.py

```
def resnet_v2(input_shape, depth, num_classes=10):
    """Konstruktor sieci ResNet w wersji 2 [b]
```

```
    Stos warstw BN-ReLU-Conv2D (1x1)-(3x3)-(1x1)
    również znanych jako warstwa ograniczająca.
    Pierwsze połączenie skrótowe na warstwę to 1x1 Conv2D.
    Drugie i następne połączenia są identycznościowe.
    Na początku każdego etapu rozmiar mapy cech jest zmniejszany o połowę
    (ang. downsampled) przez warstwę splotową z krokiem 2 tak długo, jak długo
    jest spełniony warunek. Na każdym etapie warstwy mają taką samą liczbę filtrów
    i taki sam rozmiar filtrów map.
    Rozmiary map cech:
    conv1  : 32x32,  16
    stage 0: 32x32,  64
    stage 1: 16x16, 128
    stage 2: 8x8,   256

    Argumenty:
        input_shape (tensor): postać tensora obrazu wejściowego
        depth (int): liczba warstw splotowego jądra bazowego
        num_classes (int): liczba klas (CIFAR10 ma 10 klas)

    Zwraca:
        model (Model): instancję modelu Keras
    """
    if (depth - 2) % 9 != 0:
        raise ValueError('depth should be 9n+2 (eg 110 in [b])')
    # początek definicji modelu
    num_filters_in = 16
    num_res_blocks = int((depth - 2) / 9)

    inputs = Input(shape=input_shape)
    # w v2 występuje Conv2D z BN-ReLU na wejściu przed podziałem na dwie ścieżki
    x = resnet_layer(inputs=inputs,
                     num_filters=num_filters_in,
                     conv_first=True)

    # stworzenie instancji stosu jednostek resztkowych
    for stage in range(3):
        for res_block in range(num_res_blocks):
            activation = 'relu'
            batch_normalization = True
            strides = 1
            if stage == 0:
                num_filters_out = num_filters_in * 4
                # pierwsza warstwa i pierwszy stos
                if res_block == 0:
                    activation = None
                    batch_normalization = False
            else:
                num_filters_out = num_filters_in * 2
                # pierwsza warstwa (ale nie stos)
                if res_block == 0:
                    # zmniejszanie rozmiaru/próbkowanie w dół (ang. downsample)
                    strides = 2

            # resztkowa jednostka ograniczająca
            y = resnet_layer(inputs=x,
```

```
                        num_filters=num_filters_in,
                        kernel_size=1,
                        strides=strides,
                        activation=activation,
                        batch_normalization=batch_normalization,
                        conv_first=False)
            y = resnet_layer(inputs=y,
                        num_filters=num_filters_in,
                        conv_first=False)
            y = resnet_layer(inputs=y,
                        num_filters=num_filters_out,
                        kernel_size=1,
                        conv_first=False)
            if res_block == 0:
                # projekcja liniowa skrótu resztkowego, połączenie — dopasowanie zmienionych
                #wymiarów
                x = resnet_layer(inputs=x,
                                 num_filters=num_filters_out,
                                 kernel_size=1,
                                 strides=strides,
                                 activation=None,
                                 batch_normalization=False)
            x = add([x, y])

        num_filters_in = num_filters_out

    # dodanie klasyfikatora na górze, v2 ma BN-ReLU przed warstwą łączącą
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = AveragePooling2D(pool_size=8)(x)
    y = Flatten()(x)
    outputs = Dense(num_classes,
                    activation='softmax',
                    kernel_initializer='he_normal')(y)

    # stworzenie instancji modelu
    model = Model(inputs=inputs, outputs=outputs)
    return model
```

Konstruktor modelu ResNet v2 jest przedstawiony w poniższym kodzie. Na przykład, aby zbudować ResNet110 v2, użyjemy `n = 12` i `version = 2`:

```
n=12
# wersja modelu
# oryginalny artykuł: wersja = 1 (ResNet v1),
# ulepszony ResNet: wersja = 2 (ResNet v2)
version = 2
# obliczanie głębokości z parametru n
if version == 1:
    depth = n * 6 + 2
elif version == 2:
     depth = n * 9 + 2
```

```
if version == 2:
    model = resnet_v2(input_shape=input_shape, depth=depth)
else:
    model = resnet_v1(input_shape=input_shape, depth=depth)
```

Dokładność ResNet v2 pokazano w tabeli 2.3.

Tabela 2.3. Architektury sieci ResNet v2 — walidacja w zbiorze danych CIFAR10

Warstwy	n	% dokładność na zbiorze CIFAR10 (oryginalny artykuł)	% dokładność na zbiorze CIFAR10 (ta książka)
ResNet56	9	nie podano	93,01
ResNet110	18	93,63	93,15

W pakiecie aplikacji Keras zaimplementowano niektóre modele ResNet v1 i v2 (na przykład: 50, 101, 152). Istnieją alternatywne implementacje, które są wstępnie wytrenowane, i można je łatwo ponownie wykorzystać w uczeniu transferowym. Modele użyte w tej książce są elastyczne pod względem doboru liczby warstw.

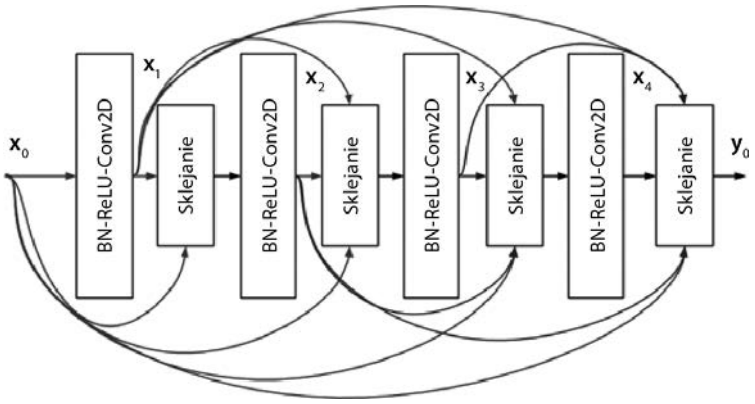
Na tym kończymy omówienie jednej z najczęściej używanych głębokich sieci neuronowych, ResNet v1 i v2. W następnej sekcji omówimy DenseNet, inną popularną architekturę głębokich sieci neuronowych.

2.4. Gęsto połączona sieć splotowa (DenseNet)

W sieci DenseNet rozwiązanie problemu znikającego gradientu jest inne. Zamiast używać omijających połączeń, mapy cech wszystkich poprzednich warstw stają się danymi wejściowymi dla następnej warstwy. Rysunek 2.11 przedstawia przykład połączenia gęstego w jednym bloku *Dense*.

Dla uproszczenia na tym rysunku pokazujemy tylko cztery warstwy. Zwróć uwagę, że dane wejściowe do warstwy l to połączenie wszystkich poprzednich map cech. Jeśli pozwolimy, aby *BN-ReLU-Conv2D* była reprezentowana przez operację $H(x)$, to wyjście warstwy l można opisać równaniem 2.5:

$$x_l = H(x_0, x_1, x_2, \cdots, x_{l-1}) \qquad \text{(równanie 2.5)}$$



Rysunek 2.11. Czterowarstwowy gęsty blok w sieci DenseNet. Wszystkie poprzednie mapy cech są danymi wejściowymi dla wszystkich następujących po niej warstw

Conv2D używa jądra o rozmiarze 3. Liczba map cech generowanych na warstwę nazywana jest **współczynnikiem wzrostu** (ang. *growth rate*) i oznaczana literą k. Zwykle $k = 12$, ale $k = 24$ jest również używane w artykule *Densely Connected Convolutional Networks* autorstwa Huanga i innych (2017) [5]. Dlatego też, jeśli liczba map cech dla sygnału wejściowego x_0 wynosi k_0, to całkowita liczba map cech na końcu czterowarstwowego bloku gęstego na rysunku 2.11 wyniesie $4k+k_0$.

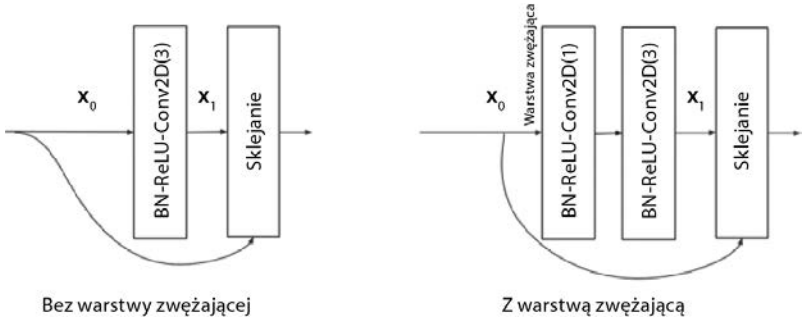
W sieciach DenseNet zalecane jest, aby blok gesty był poprzedzony przez zestaw warstw *BN-ReLU-Conv2D*, z liczbą map cech dwukrotnie większą niż współczynnik wzrostu, $k_0 = 2 \cdot k$. Na końcu bloku gęstego całkowita liczba map cech wyniesie $4 \cdot 12 + 2 \cdot 12 = 72$.

Jeśli chodzi o warstwę wyjściową, w DenseNet sugeruje się zastosowanie warstwy *AvgPool* przed warstwą gęstą z warstwą *softmax*. Jeśli nie zwiększamy ilości danych uczących przez przekształcanie, to warstwa pomijania musi zostać umieszczona za gęstym blokiem *Conv2D*.

W miarę jak sieć staje się coraz głębsza, pojawiają się dwa nowe problemy. Po pierwsze, ponieważ każda warstwa dostarcza k map cech, liczba danych wejściowych w warstwie l wynosi $(l–1) \cdot k+k_0$. Mapy cech mogą szybko się powiększać w głębokich warstwach, spowalniając obliczenia. Na przykład dla sieci 101-warstwowej będzie to $1200+24 = 1224$ dla $k = 12$.

Po drugie, podobnie jak w przypadku ResNet, w miarę zwiększania się głębokości sieci rozmiar map cech będzie zmniejszany, aby zwiększyć rozmiar obszaru pola receptywnego jądra. Jeśli DenseNet używa sklejania (ang. *concatenation*) w operacji łączenia (ang. *merge*), musi uwzględnić (ewentualne) różnice w rozmiarach.

Aby zapobiec zwiększeniu liczby map cech do punktu, w którym stanie się to nieefektywne obliczeniowo, w DenseNet wprowadzono *warstwę zwężającą* (ang. *bottleneck layer*), jak pokazano na rysunku 2.12. Pomysł polega na tym, że po każdym sklejeniu stosuje się splot 1×1 z rozmiarem filtra równym $4 \cdot k$. Ta technika redukcji wymiarów zapobiega gwałtownemu zwiększaniu się liczby map cech, które mają być przetwarzane przez *Conv2D(3)*.



Rysunek 2.12. Warstwa w bloku gęstym sieci DenseNet, z warstwą zwężającą BN-ReLU-Conv2D(1) i bez niej. Argumentem Conv2D jest rozmiar jądra

Warstwa zwężająca modyfikuje następnie warstwę DenseNet jako *BN-ReLU-Conv2D(1)_ BN-ReLU-Conv2D(3)*, zamiast tylko *BN-ReLU-Conv2D(3)*. Aby było to widoczne od razu, podaliśmy rozmiar jądra jako argument *Conv2D*. W przypadku warstwy zwężającej każdy *Conv2D(3)* przetwarza tylko $4 \cdot k$ map cech zamiast $(l-1) \cdot k + k_0$ dla warstwy l. Na przykład dla sieci 101-warstwowej dane wejściowe ostatniego *Conv2D(3)* to nadal 48 map cech dla $k = 12$ zamiast 1224 (jak obliczono wcześniej).

Aby rozwiązać problem niedopasowania rozmiarów map cech, DenseNet dzieli głęboką sieć na wiele gęstych bloków, które są połączone ze sobą warstwami przekształcającymi, jak pokazano na rysunku 2.13. W każdym bloku gęstym rozmiar mapy cech (czyli szerokość i wysokość) pozostaje stały.

Rysunek 2.13. Warstwa przejściowa pomiędzy dwoma gęstymi blokami

Rolą warstwy przekształcającej między dwoma gęstymi blokami jest taka zmiana rozmiarów mapy cech, by była ona równa mniejszemu z rozmiarów mapy cech. Rozmiar zmniejsza się zwykle o połowę, jest to osiągane przez warstwę *AveragePooling2D*. Na przykład *AveragePooling2D* z domyślnym parametrem `pool_size=2` zmniejsza rozmiar mapy cech z (64, 64, 256) do (32, 32, 256). Dane wejściowe warstwy przekształcającej to dane wyjściowe ostatniej warstwy sklejającej w poprzednim gęstym bloku.

Jednak zanim mapy cech zostaną przekazane do uśredniania do warstwy *AvgPooling*, ich liczba zostanie przy użyciu *Conv2D(1)* zmniejszona o pewien współczynnik kompresji, $0 < \theta < 1$. DenseNet wykorzystuje tu wartość $\theta = 0{,}5$. Na przykład, jeśli wynik ostatniego sklejania

poprzedniego bloku gęstego to (64, 64, 512), to po przejściu przez warstwę *Conv2D (1)* nowe wymiary mapy cech będą miały postać (64, 64, 256). By połączyć razem wykonanie operacji kompresji i redukcji wymiarów, warstwa przejściowa jest tworzona z warstw *BNConv2D(1) – AveragePooling2D*. W praktyce normalizacja wsadowa jest przed warstwą splotową.

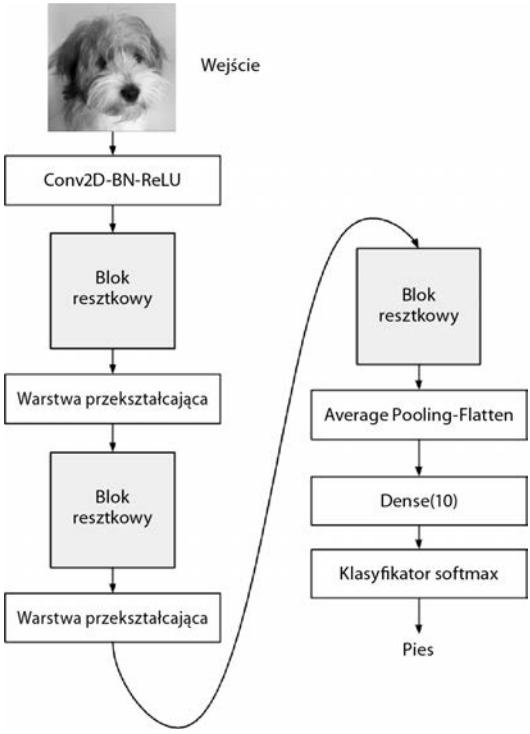
Znamy już ważne koncepcje związane z siecią DenseNet. Pozostaje zbudować ją w *tf.keras* i zweryfikować działanie *DenseNet-BC* dla zbioru danych CIFAR10.

Budowa stuwarstwowej sieci DenseNet-BC dla CIFAR10

Teraz zbudujemy DenseNet-BC (zwężenie-kompresja) o 100 warstwach dla zbioru danych CIFAR10, korzystając ze wskazówek projektowych, które omówiliśmy powyżej.

Tabela 2.4 przedstawia konfigurację modelu, kolejny rysunek przedstawia architekturę modelu, a listing pokazuje częściową implementację Keras dla DenseNet-BC z setką warstw. Musimy pamiętać, że używamy RMSprop, ponieważ jest on bardziej zbieżny niż SGD lub Adam podczas korzystania z DenseNet.

Rysunek 2.14 przedstawia przejście od konfiguracji do architektury.



Rysunek 2.14. Architektura modelu DenseNet-BC z setką warstw do klasyfikacji zdjęć ze zbioru CIFAR10

Poniżej na listingu 2.5 znajduje się częściowa implementacja Keras dla DenseNet-BC ze 100 warstwami zgodnie ze strukturą pokazaną w tabeli 2.4.

Tabela 2.4. Stuwarstwowa DenseNet-BC do klasyfikacji zdjęć ze zbioru CIFAR10

Warstwy	Rozmiar wyjściowy	DenseNet-100BC
Splot	32×32	3×3 *Conv2D*
Gęsty blok (1)	32×32	$\begin{Bmatrix} 1\times 1 & Conv2D \\ 3\times 3 & Conv2D \end{Bmatrix}\cdot 16$
Warstwa przekształcająca(1)	32×32 16×16	$\begin{Bmatrix} 1\times 1 & Conv2D \\ 2\times 2 & \text{AveragePooling2D} \end{Bmatrix}$
Gęsty blok (2)	16×16	$\begin{Bmatrix} 1\times 1 & Conv2D \\ 3\times 3 & Conv2D \end{Bmatrix}\cdot 16$
Warstwa przekształcająca(2)	16×16 8×8	$\begin{Bmatrix} 1\times 1 & Conv2D \\ 2\times 2 & \text{AveragePooling2D} \end{Bmatrix}$
Gęsty blok (3)	8×8	$\begin{Bmatrix} 1\times 1 & Conv2D \\ 3\times 3 & Conv2D \end{Bmatrix}\cdot 16$
Łączenie — średnia	1×1	8×8 *AveragePooling2D*
Warstwa klasyfikująca		*Flatten-Dense(10)-softmax*

Listing 2.5. densenet-cifar10-2.4.py

```
# początek definicji modelu
# głęboka sieć CNN, kompleks złożony z BN-ReLU-Conv2D
inputs = Input(shape=input_shape)
x = BatchNormalization()(inputs)
x = Activation('relu')(x)
x = Conv2D(num_filters_bef_dense_block,
           kernel_size=3,
           padding='same',
           kernel_initializer='he_normal')(x)
x = concatenate([inputs, x])

# stos bloków gęstych połączonych warstwami przekształcającymi
for i in range(num_dense_blocks):
    # blok gęsty jest stosem warstw zwężających
    for j in range(num_bottleneck_layers):
        y = BatchNormalization()(x)
        y = Activation('relu')(y)
        y = Conv2D(4 * growth_rate,
                   kernel_size=1,
                   padding='same',
                   kernel_initializer='he_normal')(y)
        if not data_augmentation:
            y = Dropout(0.2)(y)
        y = BatchNormalization()(y)
        y = Activation('relu')(y)
        y = Conv2D(growth_rate,
```

```
                kernel_size=3,
                padding='same',
                kernel_initializer='he_normal')(y)
        if not data_augmentation:
            y = Dropout(0.2)(y)
        x = concatenate([x, y])

    # bez warstwy przekształcającej po ostatnim gęstym bloku
    if i == num_dense_blocks - 1:
        continue

    # warstwa przekształcająca kompresuje liczbę map cech i redukuje rozmiar
    num_filters_bef_dense_block += num_bottleneck_layers * growth_rate
    num_filters_bef_dense_block = int(num_filters_bef_dense_block *
    ↪compression_factor)
    y = BatchNormalization()(x)
    y = Conv2D(num_filters_bef_dense_block,
               kernel_size=1,
               padding='same',
               kernel_initializer='he_normal')(y)
    if not data_augmentation:
        y = Dropout(0.2)(y)
    x = AveragePooling2D()(y)

# dodanie klasyfikatora na szczycie
# łączenie-średnia, rozmiar mapy cech 1 x 1
x = AveragePooling2D(pool_size=8)(x)
y = Flatten()(x)
outputs = Dense(num_classes,
                kernel_initializer='he_normal',
                activation='softmax')(y)
# stworzenie instancji i kompilacja modelu
# W oryginalnym artykule użyto SGD, ale RMSprop działa lepiej dla DenseNet.
model = Model(inputs=inputs, outputs=outputs)
model.compile(loss='categorical_crossentropy',
              optimizer=RMSprop(1e-3),
              metrics=['acc'])
model.summary()
```

Uczenie sieci DenseNet zaimplementowanej w *tf.keras* przez 200 epok osiąga dokładność 93,74% w porównaniu z dokładnością 95,49% podaną w artykule. Stosowane jest przekształcanie danych. Użyliśmy tych samych funkcji wywołań zwrotnych w ResNet v1/v2 dla DenseNet.

W przypadku głębszych warstw zmienne `growth_rate` i `depth` muszą być zmienione w tabeli w kodzie języka Python. Jednak wyuczenie sieci na głębokość 190 lub 250 warstw zajmie dużo czasu, co opisano w artykule. Aby dać wyobrażenie o czasie treningu, należy wyjaśnić, że każda epoka trwa około godziny na procesorze graficznym 1060Ti. Podobnie jak w przypadku ResNet, pakiet Keras zawiera wstępnie wytrenowane modele dla DenseNet 121 i wyższych.

DenseNet kończy temat głębokich sieci neuronowych. Sieć ta oraz ResNet są nieocenione same w sobie lub jako sieci do ekstrakcji cech w wielu innych zadaniach.

2.5. Podsumowanie

W tym rozdziale przedstawiliśmy funkcyjne API jako zaawansowaną metodę budowania złożonych modeli głębokich sieci neuronowych przy użyciu *tf.keras*. Zademonstrowaliśmy również, w jaki sposób można wykorzystać funkcyjne API do budowy sieci Y z wieloma wejściami i jednym wyjściem. Ta sieć w porównaniu z nierozgałęzioną siecią CNN osiągnęła lepszą dokładność. W dalszej części książki okaże się, że funkcyjne API jest niezbędne do budowania bardziej złożonych i zaawansowanych modeli. Na przykład w następnym rozdziale funkcyjne API umożliwi nam zbudowanie modułowego kodera, dekodera i sieci autokodującej.

Spędziliśmy również sporo czasu, badając dwie ważne głębokie sieci, ResNet i DenseNet. Obie te sieci są wykorzystywane nie tylko w klasyfikacji, ale także w innych obszarach, takich jak segmentacja, wykrywanie, śledzenie, generowanie i wizualne rozumienie semantyczne. W rozdziale 11. „Wykrywanie obiektów" i rozdziale 12. „Segmentacja semantyczna" użyjemy ResNet do wykrywania i segmentacji obiektów. Musimy pamiętać, że dużo ważniejsze jest dogłębne rozumienie decyzji podejmowanych podczas projektowania modeli w ResNet i DenseNet niż tylko naśladowanie ich pierwotnej implementacji — tylko w ten sposób będziesz w stanie wykorzystać kluczowe koncepcje ResNet i DenseNet do własnych celów.

2.6. Bibliografia

[1] Kaiming He et al., *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, „Proceedings of the IEEE international conference on computer vision", 2015, *https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf?spm=5176.100239.blogcont55892.28.pm8zm1&file=He_Delving_Deep_into_ICCV_2015_paper.pdf*.

[2] Kaiming He et al., *Deep Residual Learning for Image Recognition*, „Proceedings of the IEEE conference on computer vision and pattern recognition", 2016, *http://openaccess.thecvf.com/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf*.

[3] Karen Simonyan, Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Image Recognition*, ICLR, 2015, *https://arxiv.org/pdf/1409.1556/*.

[4] Kaiming He et al., *Identity Mappings in Deep Residual Networks*, European Conference on Computer Vision, Springer International Publishing, 2016, *https://arxiv.org/pdf/1603.05027.pdf*.

[5] Gao Huang et al., *Densely Connected Convolutional Networks*, „Proceedings of the IEEE conference on computer vision and pattern recognition", 2017, *http://openaccess.thecvf.com/content_cvpr_2017/papers/Huang_Densely_Connected_Convolutional_CVPR_2017_paper.pdf*.

[6] Saining Xie et al., *Aggregated Residual Transformations for Deep Neural Networks*, Computer Vision and Pattern Recognition (CVPR, 2017), IEEE Conference on. IEEE, 2017, *http://openaccess.thecvf.com/content_cvpr_2017/papers/Xie_Aggregated_Residual_Transformations_CVPR_2017_paper.pdf*.

[7] Sergey Zagoruyko, Nikos Komodakis, *Wide residual networks*, 2016, *arXiv:1605.07146*.

Oto propozycja dla specjalistów zajmujących się programowaniem sztucznej inteligencji i studentów kształcących się w tej dziedzinie. Autor przybliża tajniki tworzenia sieci neuronowych stosowanych w uczeniu głębokim i pokazuje, w jaki sposób używać w tym celu bibliotek Keras i TensorFlow. Objaśnia zagadnienia dotyczące programowania AI zarówno w teorii, jak i praktyce. Liczne przykłady, czytelna oprawa graficzna i logiczne wywody sprawiają, że to skuteczne narzędzie dla każdego, kto chce się nauczyć budowania sieci neuronowych typu MLP, CNN i RNN.

Książka wprowadza w teoretyczne fundamenty uczenia głębokiego — znalazły się w niej wyjaśnienia podstawowych pojęć związanych z tą dziedziną i różnice pomiędzy poszczególnymi typami sieci neuronowych. Opisano tutaj również metody programowania algorytmów używanych w uczeniu głębokim i sposoby ich wdrażania. Dzięki lekturze lepiej zrozumiesz sieci neuronowe, nauczysz się ich tworzenia i zastosowania w różnych projektach z zakresu AI.

Polecamy tę książkę każdemu, kto:

- chce zrozumieć, jak działają sieci neuronowe i w jaki sposób się je tworzy
- specjalizuje się w uczeniu głębokim lub zamierza lepiej poznać tę dziedzinę
- posługuje się sieciami neuronowymi w programowaniu
- chce się nauczyć stosować biblioteki Keras i TensorFlow w uczeniu głębokim

Rowel Atienza

— profesor w Instytucie Elektrycznym i Inżynierii Elektronicznej Uniwersytetu Filipińskiego w Diliman, kierownik katedry sztucznej inteligencji w Instytucie Dado i Marii Banatao. Ma praktyczne doświadczenie w programowaniu robotów, tworzeniu algorytmów sztucznej inteligencji i widzeniu komputerowym. Autor licznych artykułów i wystąpień na konferencjach dotyczących AI, specjalista w zakresie sieci neuronowych i uczenia głębokiego.

ISBN 978-83-283-8883-3