LABORATORIUM nr13

Głębokie uczenie

Będziemy budować modele przy użyciu biblioteki Keras, udostępniającej przyjazny interfejs do jednej z najpopularniejszych bibliotek głębokiego uczenia — TensorFlow autorstwa Google.

Zastosowania głębokiego uczenia

Spośród szerokiej gamy zastosowań głębokiego uczenia na szczególną uwagę zasługują następujące:

■ rozgrywanie gier,

■ widzenie komputerowe: rozpoznawanie obiektów, rozpoznawanie wzorców,

■ pojazdy autonomiczne (samochody bez kierowców),

■ roboty,

■ polepszenie doświadczenia klientów,

■ chatboty,

■ diagnostyka medyczna,

■ wyszukiwarka Google,

■ rozpoznawanie twarzy,

■ automatyczne opisywanie obrazów, automatyczne generowanie napisów do filmów,

■ zwiększanie rozdzielczości obrazów,

■ rozpoznawanie mowy,

■ tłumaczenie między językami naturalnymi,

■ przewidywanie wyników wyborów,

■ *Google Sunroof* — kalkulator pomagający w oszacowaniu ewentualnych oszczędności zużycia energii wskutek zainstalowania paneli słonecznych na dachu,

■ aplikacje-generatory: generowanie oryginalnych obrazów, przetwarzanie istniejących

obrazów, kolorowanie obrazów i filmów czarno-białych, analizowanie obrazów pod kątem stylu określonego artysty, komponowanie muzyki, komponowanie poezji i prozy.

Zasoby biblioteki „Keras”

Biblioteka *Keras* także oferuje kilka cennych zasobów, które mogą okazać się pomocne w studiach nad głębokim uczeniem:

■ na kanale *Kerasteam slack channel*, pod adresem *https://kerasteam.slack.com/*, znaleźć można odpowiedzi na wiele interesujących pytań;

■ wiele ciekawych artykułów i tutoriali dostępnych jest pod adresem *http://blog.keras.io/*;

■ dokumentacja biblioteki *Keras* dostępna jest pod adresem *http://keras.io*;

Pliki danych wbudowane w bibliotekę „Keras”

Moduł tensorflow.keras.datasets udostępnia kilka zbiorów danych z biblioteki *Keras* na potrzeby studiów nad głębokim uczeniem:

■ bazę danych **MNIST database of handwritten digits**, używaną do klasyfikowania odręcznie napisanych cyfr dziesiętnych w dziesięciu klasach odpowiadających poszczególnym cyfrom. Baza ta zawiera 60 000 etykietowanych próbek treningowych i 10 000 testowych, każda w postaci obrazu 28×28 pikseli w odcieniach szarości.

■ bazę danych **Fashion-MNIST database of fashion articles**, używaną do klasyfikowania fotografii przedstawiających ubrania i galanterię modową w 10 kategoriach. Zawiera ona 60 000 etykietowanych próbek treningowych i 10 000 testowych, każda w postaci obrazu 28×28 pikseli w odcieniach szarości.

■ zbiór danych **IMDb Movie reviews**, wykorzystywany do analizy sentymentu. Zawiera on próbki w postaci recenzji filmowych, etykietowane jako pozytywne (1) albo negatywne (0), w liczbie 25 000 treningowych i 25 000 testowych.

■ zbiór danych **CIFAR10 small image classification**, zawierający 50 000 próbek treningowych i 100 000 testowych, etykietowanych 10 kategoriami, każda próbka jest kolorowym obrazkiem 32×32 piksele;

■ zbiór danych **CIFAR100 small image classification**, zawierający 50 000 próbek treningowych i 100 000 testowych, etykietowanych 100 kategoriami, każda próbka jest kolorowym obrazkiem 32×32 piksele.

Alternatywne środowiska Anacondy

W zadanie laboratorium korzystać będziemy z biblioteki Keras w wersji wbudowanej w bibliotekę TensorFlow. Fakt ten stanowi dobrą okazję do zaprezentowania użytecznego mechanizmu dystrybucji Anaconda, jakim jest instalowanie alternatywnych konfiguracji zwanych środowiskami (ang. environments).

Po co alternatywne środowiska?

Środowiska Anacondy to zróżnicowane konfiguracje obejmujące różne zestawy różnych wersji bibliotek i innych komponentów. Możliwość przełączania się między środowiskiem podstawowym (ang. base environment) a środowiskami alternatywnymi (ang. custom environments) okazuje się nieoceniona w sytuacji, gdy projekt zależny jest od konkretnej wersji danej biblioteki czy nawet samego języka Python. Środowiska te zapewniają bowiem powtarzalność warunków realizacji tego projektu.

Środowisko podstawowe tworzone jest przy instalowaniu Anacondy, instalowane są w nim także wszystkie biblioteki Pythona oraz wszystkie dodatkowe biblioteki (chyba że jawnie zdecydujemy inaczej). Środowiska alternatywne dają natomiast użytkownikowi kontrolę nad specyficznymi bibliotekami instalowanymi na potrzeby specyficznych zadań.

Tworzenie środowiska

Utworzenie nowego środowiska alternatywnego następuje w wyniku polecenia

```
conda create
```

Na potrzeby dalszych zadań utworzymy takie środowisko o nazwie tf_env (to skrót od TensorFlow environment; jeśli chcesz, możesz wybrać inną nazwę). W tym celu uruchom

■ w Windows: Anaconda Prompt jako administrator,

■ w Linuksie: Terminal lub shell,

■ w MacOS: Terminal

i wydaj polecenie

```
conda create -n tf_env tensorflow anaconda ipython jupyterlab scikit-learn matplotlib
↪seaborn h5py pydot graphviz
```

Instalacja odbywa się w pełni automatycznie: po sprawdzeniu zależności między komponentami instalator wyświetli pytanie

```
Proceed ([y]/n)?
```

należy odpowiedzieć twierdząco, naciskając Enter, po czym uzbroić się w cierpliwość i poczekać na zakończenie instalacji.

Aktywowanie alternatywnego środowiska

Anaconda startuje zawsze z konfiguracją środowiska podstawowego. Aby przełączyć się na inne środowisko, należy je **aktywować**, wydając polecenie

```
conda activate <nazwa środowiska>
```

w naszym przypadku

```
conda activate tf_env
```

Aktywacja skuteczna jest do zakończenia sesji konsoli, wszystkie instalowane w tym czasie biblioteki stają się częścią środowiska alternatywnego, nie środowiska podstawowego. Uruchomienie jednak w tym czasie kolejnej instancji konsoli spowoduje wystartowanie Anacondy w środowisku podstawowym — w ten sposób możemy utrzymywać dwie lub więcej sesji w różnych środowiskach.

Dezaktywacja alternatywnego środowiska

Przełączenie sesji na środowisko alternatywne jest skuteczne tylko do końca tejże sesji, nowa sesja zawsze rozpoczyna się w środowisku podstawowym. Powrót do środowiska podstawowego można osiągnąć także bez kończenia sesji, poprzez **dezaktywację** aktualnego środowiska, wykonywaną za pomocą polecenia

```
conda deactivate
```

Sieci neuronowe

Głębokie uczenie to forma uczenia maszynowego wykorzystująca do nauki **sztuczne sieci neuronowe** (ang. *artificial neural networks*), zwane po prostu **sieciami neuronowymi** (ang. *neural networks*). Sieć neuronowa to konstrukcja programistyczna działająca w taki sam sposób, jak naukowcy wyobrażają sobie działanie ludzkiego mózgu. System nerwowy człowieka to komórki zwane *neuronami*, połączone ścieżkami zwanymi *synapsami*. Specyficzne neurony,

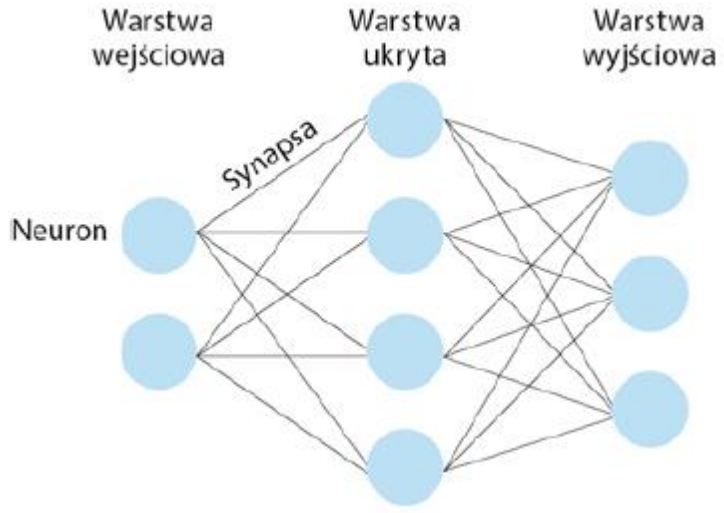
umożliwiające organizmowi wykonywanie określonych czynności — na przykład chodzenia— komunikują się między sobą i aktywowane są zawsze, gdy chcemy pospacerować.

Sztuczne neurony

W sieci neuronowej połączone ze sobą **sztuczne neurony** (ang. *artificial neurons*), symulując działanie połączonych neuronów mózgu, realizują **uczenie się tej sieci**. W procesie tego uczenia, zmierzającym do osiągnięcia określonego celu, wzmacniane są połączenia między wybranymi neuronami. W ramach **nadzorowanego głębokiego uczenia** (ang. *supervised deep learning*) — tę odmianę opisywać będziemy w tym rozdziale — dążymy do przewidywania etykiet próbek. W tym celu trenujemy ogólny model sieci neuronowej, po czym wykorzystujemy go do prognoz dotyczących nieznanych danych.

Diagram sieci neuronowej

Pokazany na rysunku diagram sieci neuronowej uwidacznia jej trzy **warstwy** (ang. *layer*). Każde kółko reprezentuje jeden neuron, a linie między kółkami symulują synapsy. Wyjście jednego neuronu staje się wejściem innego, stąd rzeczownik „sieć”. Diagram z rysunku przedstawia sieć **w pełni połączoną** (ang. *fully connected*) — każdy neuron danej warstwy połączony jest ze *wszystkimi* neuronami warstwy następnej.



Sieć neuronowa w bibliotece „Keras”

Implementacja sieci neuronowej w bibliotece *Keras* obejmuje następujące komponenty:

■ **Sieć** (*network*) zwana także **modelem** to sekwencja **warstw** zawierających neurony realizujące uczenie się na podstawie próbek. Neurony każdej warstwy odbierają informacje wejściowe, przetwarzają je (w ramach *funkcji aktywacji*) i

produkują informacje na wyjściu. Dane trafiają do sieci poprzez **warstwę wejściową** (*input layer*) określającą wymiary próbek danych. Po niej następują wewnętrzne **warstwy ukryte** (*hidden layers*) implementujące proces uczenia, a końcową warstwą jest **warstwa wyjściowa** (*output layer*), której zadaniem jest formułowanie prognoz. Im więcej warstw obejmuje ów *stos warstw*, tym głębsza sieć, stąd przymiotnik „głębokie”.

■ **Funkcja straty** (*loss function*) obliczająca miarę jakości (a raczej — miarę mankamentów) prognoz produkowanych przez sieć. Im mniejsza wartość zwracana przez tę funkcję, tym lepsza prognoza.

■ **Optymalizator** (*optimizer*) dążący do minimalizowania wartości zwracanych przez funkcję straty — czyli dostrajania sieci tak, by produkowała lepsze prognozy.

Wczytanie zbioru „MNIST”

Zaczynamy oczywiście od zaimportowania zbioru danych:

```
[1]: from tensorflow.keras.datasets import mnist
```

Za pomocą funkcji load_data z modułu mnist ładujemy następnie kolekcje próbek — treningowych i testowych:

```
[2]: (X_train, y_train), (X_test, y_test) = mnist.load_data()
```

Eksploracja danych

Spójrzmy najpierw ogólnie na nasze dane, zanim przystąpimy do ich przetwarzania. Zacznijmy od zbadania struktury obiektów zwracanych przez funkcję load_data():

```
[3]: X_train.shape
[3]: (60000, 28, 28)
[4]: y train.shape
[4]: (60000,)
[5]: X test.shape
[5]: (10000, 28, 28)
[6]: y test.shape
[6]: (10000,)
```

Z urywków [3] i [5] można wyczytać, że obrazy-próbki mają większa rozdzielczość (28×28 pikseli) niż w zbiorze *Digits* (jak pamiętamy — 8×8).

Wizualizacja cyfr

Do podglądu próbek użyjemy bibliotek *Matplotlib* i *Seaborn*. Najpierw je zaimportujemy i ustawimy skalę czcionki:

```
[7]: %matplotlib inline
[8]: import matplotlib.pyplot as plt
[9]: import seaborn as sns
[10]: sns.set(font_scale=2)
```

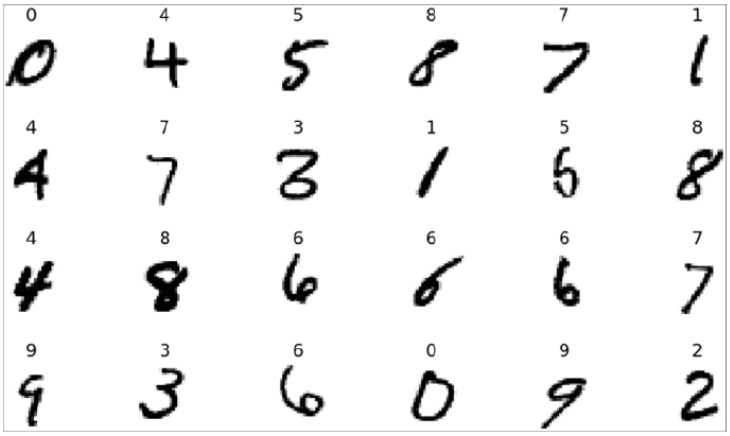
„Magiczne” polecenie

```
%matplotlib inline
```

powoduje, że grafika produkowana przez bibliotekę *Matplotlib* wyświetlana będzie *wewnątrz notatnika*, nie w osobnym oknie. Informację na temat innych „magicznych” poleceń interpretera IPython związanych z notatnikiem znaleźć można pod adresem
https://ipython.readthedocs.io/en/stable/interactive/magics.html

Następnie wyświetlimy 24 losowo wybrane próbki treningowe.

```
[11]: import numpy as np
index = np.random.choice(np.arange(len(X train)), 24, replace=False)
figure, axes = plt.subplots(nrows=4, ncols=6, figsize=(16, 9))
for item in zip(axes.ravel(), X train[index], y train[index]):
axes, image, target = item
axes.imshow(image, cmap=plt.cm.gray r)
axes.set xticks([]) # usunięcie znaczników z osi x
axes.set yticks([]) # usunięcie znaczników z osi y
axes.set title(target)
plt.tight_layout()
```



Zmiana struktury danych

Sieci neuronowe implementowane przez bibliotekę *Keras* wymagają, by każda próbka miała strukturę postaci

```
(szerokość, wysokość, kanały)
```

Dla próbek zbioru *MNIST szerokość* i *wysokość* równe są 28 (pikseli), zaś wygląd każdego piksela opisany jest przez jeden kanał — liczbę całkowitą z przedziału 0 – 255, wyrażającą intensywność w skali szarości: (28, 28, 1)

```
[12]: X_train = X_train.reshape((60000, 28, 28, 1))

[13]: X train.shape
[13]: (60000, 28, 28, 1)

[14]: X test = X test.reshape((10000, 28, 28, 1))

[15]: X test.shape
[15]: (10000, 28, 28, 1)
```

Normalizacja danych

```
[16]: X train = X train.astype('float32') / 255
[17]: X_test = X_test.astype('float32') / 255
```

Kodowanie „z gorącą jedynką” (ang. *one-hot encoding*)

```
[18]: from tensorflow.keras.utils import to_categorical

[19]: y train = to categorical(y train)

[20]: y train.shape
[20]: (60000, 10)

[21]: y train[0]
[21]: array([ 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
dtype=float32)

[22]: y test = to categorical(y test)

[23]: y test.shape
[23]: (10000, 10)
```

Budowanie sieci neuronowej

```
[24]: from tensorflow.keras.models import Sequential

[25]: cnn = Sequential()
```

Dodawanie warstw do sieci

```
[26]: from tensorflow.keras.layers import Conv2D, Dense, Flatten,MaxPooling2D
```

Dodawanie warstwy konwolucji

```
[27]: cnn.add(Conv2D(filters=64, kernel size=(3, 3), activation='relu',
                  input_shape=(28, 28, 1)))
```

Dodawanie warstwy łączącej

```
[28]: cnn.add(MaxPooling2D(pool_size=(2, 2)))
```

Dodawanie kolejnych warstw: konwolucyjnej i łączącej

Sieci CNN mają zwykle kilka warstw konwolucji i kilka warstw łączących.

```
[29]: cnn.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu'))

[30]: cnn.add(MaxPooling2D(pool_size=(2, 2)))
```

Spłaszczanie wyników

```
[31]: cnn.add(Flatten())
```

Dodawanie warstwy gęstej w celu redukcji liczby cech

```
[32]: cnn.add(Dense(units=128, activation='relu'))
```

Dodawanie kolejnej warstwy gęstej produkującej wynik klasyfikacji

```
[33]: cnn.add(Dense(units=10, activation='softmax'))
```

Podsumowanie modelu

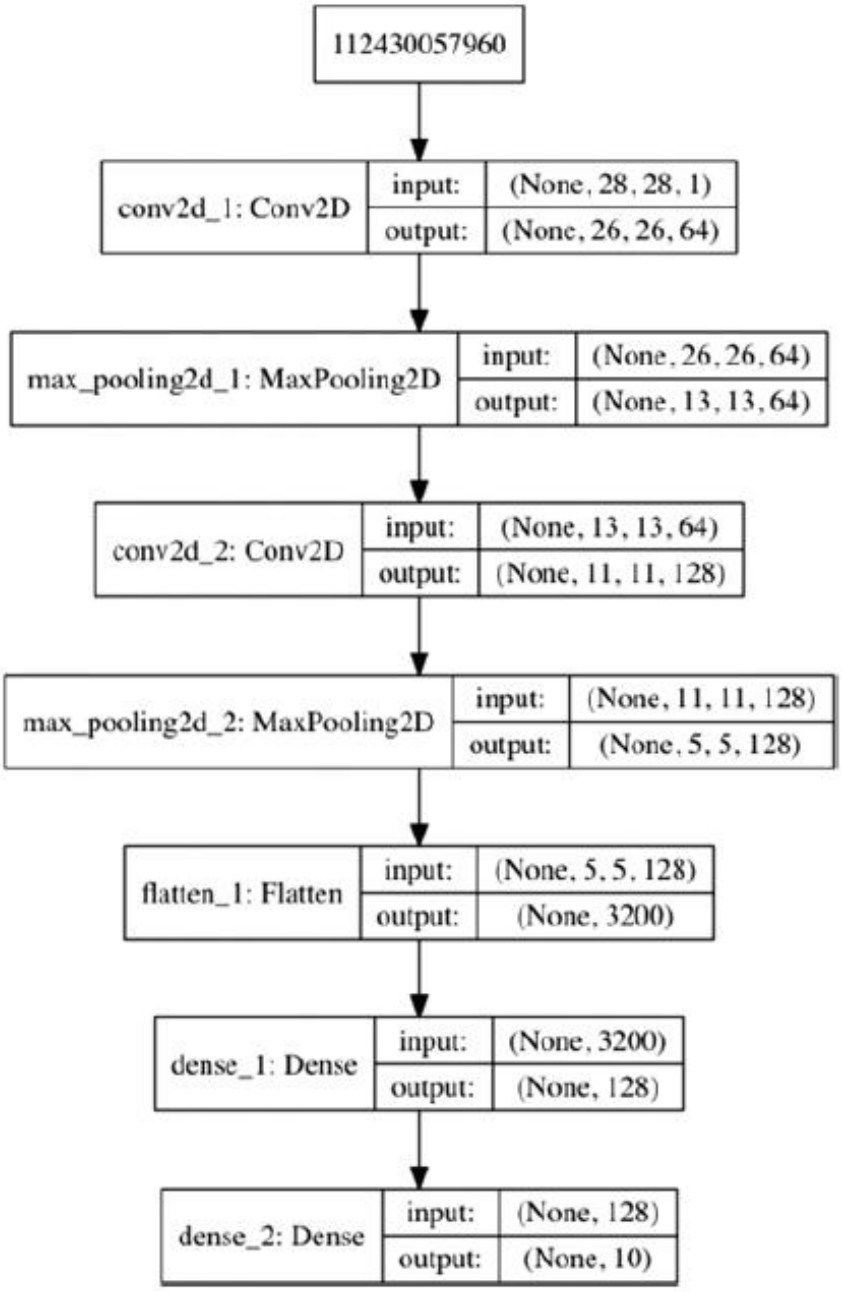
Metoda summary modelu wypisuje informację podsumowującą na temat kolejnych jego warstw. Zauważmy, że *Keras* przyporządkowała im automatycznie wygenerowane nazwy.

```
[34]: cnn.summary()

Layer (type) Output Shape Param #
=================================================================
conv2d 1 (Conv2D) (None, 26, 26, 64) 640

max pooling2d 1 (MaxPooling2 (None, 13, 13, 64) 0

conv2d 2 (Conv2D) (None, 11, 11, 128) 73856

max pooling2d 2 (MaxPooling2 (None, 5, 5, 128) 0

flatten 1 (Flatten) (None, 3200) 0

dense 1 (Dense) (None, 128) 409728

dense 2 (Dense) (None, 10) 1290
=================================================================
Total params: 485,514
Trainable params: 485,514
Non-trainable params: 0
```

Wizualizacja struktury modelu

```
[35]: from tensorflow.keras.utils import plot_model
      from IPython.display import Image
      plot model(cnn, to file='convnet.png', show shapes=True,
                 show layer names=True)
      Image(filename='convnet.png')
```

Po zapisaniu obrazu w pliku *convnet.png* wykorzystujemy klasę Image z modułu IPython.display, by ujrzeć wyprodukowany diagram w oknie notatnika



Kompilowanie modelu

```
[36]: cnn.compile(optimizer='adam',
                  loss='categorical crossentropy',
                  metrics=['accuracy'])
```

Trenowanie i ewaluacja modelu

```
[37]: cnn.fit(X_train, y_train, epochs=5, batch_size=64,
              validation_split=0.1)
```

W wyniku wykonania urywka [37] otrzymamy następujący raport — wyróżniliśmy wartości odzwierciedlające dokładność modelu (acc) i dokładność obliczoną w wyniku walidacji (val_acc):

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/5
54000/54000 [==============================] - 68s 1ms/step - loss:
0.1407 - acc: 0.9580 - val_loss: 0.0452 - val_acc: 0.9867
Epoch 2/5
54000/54000 [==============================] - 64s 1ms/step - loss:
0.0426 - acc: 0.9867 - val_loss: 0.0409 - val_acc: 0.9878
Epoch 3/5
54000/54000 [==============================] - 69s 1ms/step - loss:
0.0299 - acc: 0.9902 - val_loss: 0.0325 - val_acc: 0.9912
Epoch 4/5
54000/54000 [==============================] - 70s 1ms/step - loss:
0.0197 - acc: 0.9935 - val_loss: 0.0335 - val_acc: 0.9903
Epoch 5/5
54000/54000 [==============================] - 63s 1ms/step - loss:
0.0155 - acc: 0.9948 - val_loss: 0.0297 - val_acc: 0.9927
[37]: <tensorflow.python.keras.callbacks.History at 0x7f105ba0ada0>
```

Ewaluacja

```
[38]: loss, accuracy = cnn.evaluate(X_test, y_test)
10000/10000 [==============================] - 4s 366us/step

[39]: loss
[39]: 0.026809450998473768

[40]: accuracy
[40]: 0.9917
```

Prognozowanie

```
[41]: przypuszczenia = cnn.predict(X_test)
```

Jak wyjaśnialiśmy, dla każdej próbki prognoza ma postać 10-elementowej tablicy, której elementy zawierają prawdopodobieństwo tego, iż próbka reprezentuje konkretną cyfrę dziesiętną.

Dla każdej próbki testowej sporządziliśmy *oczekiwaną* wartość takiej prognozy, kodując ją według „gorącej” jedynki — przykładowo, dla pierwszej próbki reprezentującej cyfrę 7, ma ona następującą postać:

```
[42]: y_test[0]
[42]: array([0., 0., 0., 0., 0., 0., 0., 1., 0., 0.], dtype=float32)
```

Prognoza wykonana przez model dla tej próbki

```
[43]: for indeks, przypuszczenie in enumerate(przypuszczenia[0]):
          print(f'{indeks}: { przypuszczenie:.10%}')
0: 0.0000000201%
1: 0.0000001355%
2: 0.0000186951%
3: 0.0000015494%
4: 0.0000000003%
5: 0.0000000012%
```

```
6: 0.0000000000%
7: 99.9999761581%
8: 0.0000005577%
9: 0.0000011416%
```

A więc próbka została rozpoznana prawidłowo — prognoza optuje za właściwą cyfrą z prawdopodobieństwem niemal stuprocentowym. Oczywiście nie wszystkie prognozy bywają aż tak wyraźne.

Wyszukiwanie chybionych prognoz

Recz jasna, interesują nas także prognozy mniej lub bardziej chybione.

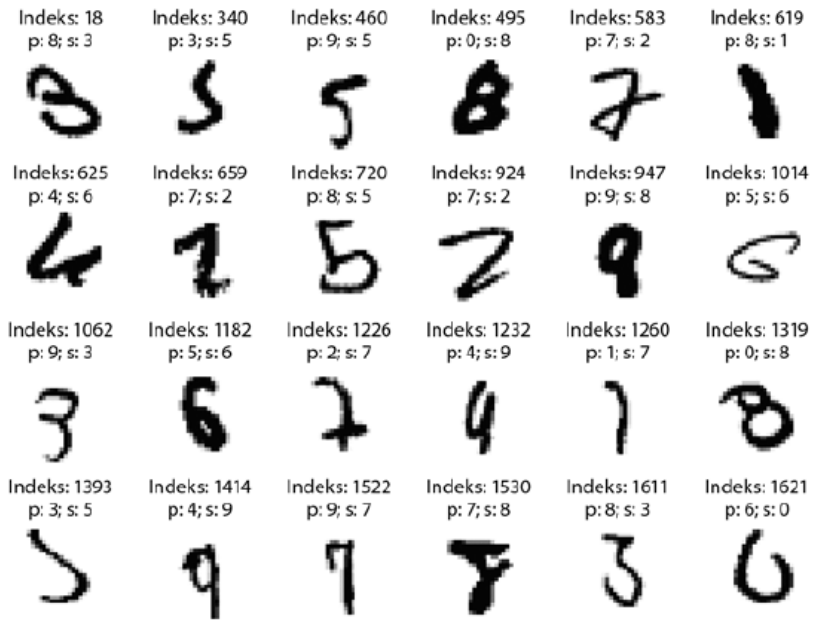
```
[44]: obrazy = X_test.reshape((10000, 28, 28))
      chybione prognozy = []

      for i, (p, e) in enumerate(zip(przypuszczenia, y_test)):
          prognozowany, spodziewany = np.argmax(p), np.argmax(e)

          if prognozowany != spodziewany:
              chybione prognozy.append(
                  (i, obrazy[i], prognozowany, spodziewany))
```

Interesujące jest, ile spośród 10 000 próbek testowych zostało błędnie sklasyfikowanych:

```
[45]: len(chybione_prognozy)
[45]: 83
```

Wizualizacja chybionych prognoz

```
[46]: figure, axes = plt.subplots(nrows=4, ncols=6, figsize=(16, 12))

      for axes, element in zip(axes.ravel(), chybione prognozy):
          indeks, obraz, prognozowany, spodziewany = element
          axes.imshow(obraz, cmap=plt.cm.gray r)
          axes.set xticks([]) # usuń znaczniki z osi x
          axes.set yticks([]) # usuń znaczniki z osi y
          axes.set title(
              f'indeks: {indeks}\np: {prognozowany}; s: {spodziewany}')
      plt.tight_layout()
```



Zadanie

1. Wykonaj zadanie z przykładu i uzyskaj wytrenowany model rozpoznawania odręcznie napisanych cyfr.
2. Utwórz plik z odręcznie napisaną cyfrą w dowolnym edytorze graficznym i rozpoznaj go za pomocą modelu, wytrenowanego w pkt. 1.