Klasyfikacja symboli amerykańskiego języka migowego w czasie rzeczywistym

20 czerwca 2022

[1]:
```
from IPython.display import Image
```

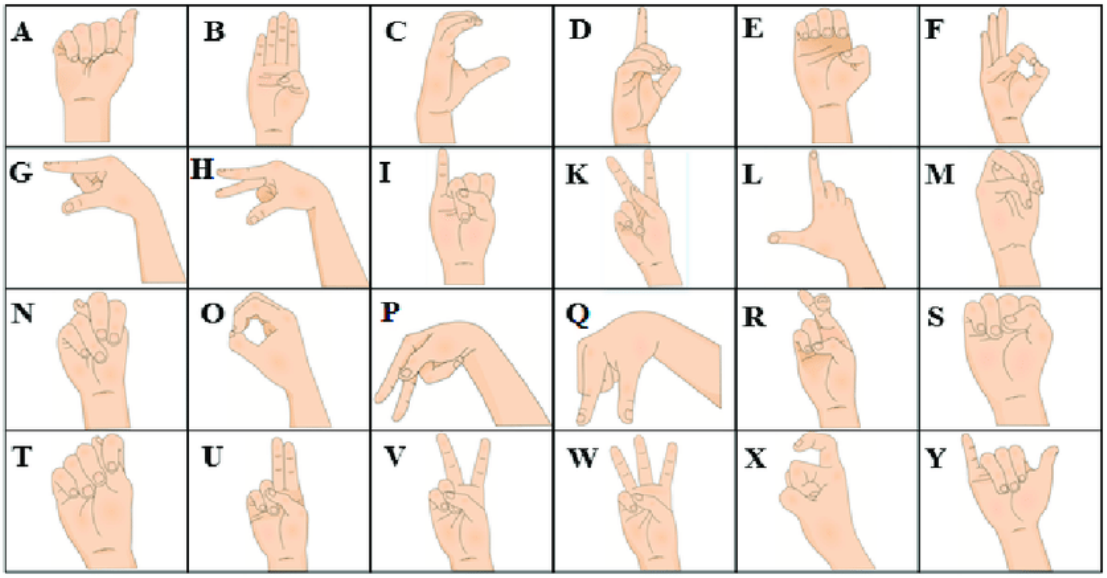
1 Projekt na zajęcia Algorytmy Uczenia Maszynowego, prowadzący dr. inż Piotr Ciskowski

2 Twórcy: Adam Kubiak i Piotr Gorzelnik

W ramach projektu wykorzystaliśmy zbiór danych, złożony z zdjęć dłoni, która reprezentuje jedną z 24 liter języka migowego(W alfabecie znajduje się 26 liter jednak litera "J" oraz "Z" jest reprezentowana poprzez ruch dłonią co wykracza poza założenia projektu). Zbiór danych zostaje wykorzystany do uczenia modelu Convolutional Neural Networks (CNN), Sequential, który jest dostępny w pakiecie o nazwie "keras". Dane uzykane zostały ze strony kaggle.com. Są one udostępnione w formie pliku .csv gdzie każdy pixel jest reprezentowany przez liczbe z zakresu 0-255, każde zdjęcie składa się z 784 pixeli(wymiary 28x28). Dodatkowo dołożyliśmy kilka dodatkowych znaków do wykorzystywanego zbioru danych. Projekt spełnia następujące zadania:

- Przygotowanie zbioru danych
- Przetworzenie zbioru aby spełniał wymagania modelu Sequential
- Przygotowanie modelu i ustawienie odpowiednich wartości jego parametrów
- Nauka modelu
- Sprawdzenie dokładności modelu z wykrzystaniem zbioru testowego danych
- Wykorzystanie biblioteki CV2 w celu zbierania zdjęć z kamery
- Przetworznie pobranego obrazu z kamery
- Estymacja pokazywanego znaku przez użytkownika z wykorzystaniem wyuczonego modelu CNN

3



4 Importowanie pakietów wykorzystanych w projekcie

[2]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import cv2
import tensorflow as tf
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import scipy
import cv2
import os
import keras
from keras.models import Sequential
from keras.layers import Dense,Flatten,Conv2D,MaxPool2D,Dropout
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelBinarizer
from tensorflow.keras.preprocessing.image import ImageDataGenerator

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

5 Ładowanie i wstępne przetwarzanie zbioru danych

Format zestawu danych jest wzorowany, aby ściśle odpowiadał klasycznemu MNIST. Każdy przypadek treningowy i testowy reprezentuje etykietę (0-25) jako mapę jeden do jednego dla każdej litery alfabetu A-Z (nie ma przypadków dla 9=J lub 25=Z ze względu na ruchy gestów). Dane

treningowe (27 455 przypadków) i dane testowe (7172 przypadki). Wiersz danych w pliku .csv zawiera etykiete oraz pixel1,pixel2. . . .pixel784, które w całości reprezentują pojedynczy obraz 28x28 pikseli z wartości w skali szarości z zakresu 0-255. Dodatkowe etykiety wybranych przez nas znaków są przetwarzane za pomocą napisanej funkcji, która bada wszystkie zdjęcia znajdujące się w podanym przez użytkownika folderze, a następnie przetwarza je do odpowiedniego formatu narzuconego przez podstawowy zbiór danych wraz z etykietą przypisaną przez użytkownika.

[3]:
```
train_df=pd.read_csv('kaggle\sign_mnist_train.csv')
test_df=pd.read_csv('kaggle\sign_mnist_test.csv')
```

Prezentacja formatu pobranych danych z pliku .csv do struktury danych DataFrame z pakietu Pandas

[4]:
```
train_df.tail()
```

[4]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	\
27450	13	189	189	190	190	192	193	193	193	
27451	23	151	154	157	158	160	161	163	164	
27452	18	174	174	174	174	174	175	175	174	
27453	17	177	181	184	185	187	189	190	191	
27454	23	179	180	180	180	182	181	182	183	

	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	\
27450	193	...	132	165	99	77	52	
27451	166	...	198	198	198	198	198	
27452	173	...	121	196	209	208	206	
27453	191	...	119	56	27	58	102	
27454	182	...	108	132	170	194	214	

	pixel780	pixel781	pixel782	pixel783	pixel784
27450	200	234	200	222	225
27451	196	195	195	195	194
27452	204	203	202	200	200
27453	79	47	64	87	93
27454	203	197	205	209	215

[5 rows x 785 columns]

Etykiety znajdujące się w DataFrame przed dodaniem własnych znaków do zbioru danych

[5]:
```
df = train_df.iloc[:,0]
print(train_df.iloc[:,0].unique())
```

```
[ 3  6  2 13 16  8 22 18 10 20 17 19 21 23 24  1 12 11 15  4  0  5  7 14]
```

Funkcja mająca na celu przetworzenie zdjęć z dodatkowymi znakami języka migowego do formatu istniejącego DataFrame

[6]:
```
def addDataToSet(path_folder, label_):
    i = 0
    path = os.path.join(path_folder)
    rows = len(os.listdir(path))
    finalImageList = np.zeros(shape=(rows,785))
    while i < len(os.listdir(path)):
        for img in os.listdir(path):
            img_array = cv2.imread(os.path.join(path, img))
            img_array = np.mean(img_array,-1)
            #print(img.shape)
            img_array = scipy.ndimage.zoom(img_array,1/124)
            #plt.imshow(img_array, cmap='gray')
            img_array = img_array.reshape((1,784))
            img_array = np.insert(img_array, 0, int(label_), axis=1) # label
            img_array = np.around(img_array,decimals=0)
            img_array = img_array.astype(int)
            finalImageList[i] = (img_array)
            i+=1
    return finalImageList
```

Dołączenie dodatkowych znaków do zbioru uczącego

[7]:
```
Datadirectory = "FAK"
label = 9
df = pd.DataFrame(addDataToSet(Datadirectory,label), columns = ['label','pixel1','pixel2','pixe
```

```
C:\Users\Adam\AppData\Local\Temp\ipykernel_11508\4258792020.py:5: FutureWarning:
The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.
  train_df = train_df.append(df)
```

[8]:
```
Datadirectory = "ILY"
label = 25
df = pd.DataFrame(addDataToSet(Datadirectory,label), columns = ['label','pixel1','pixel2','pixe
```

```
C:\Users\Adam\AppData\Local\Temp\ipykernel_11508\3238422897.py:5: FutureWarning:
The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.
  train_df = train_df.append(df)
```

Dołączenie dodatkowych znaków do zbioru testowego

[9]:
```
Datadirectory = "FAK_TEST"
label = 9
df = pd.DataFrame(addDataToSet(Datadirectory,label), columns = ['label','pixel1','pixel2','pixe
```

```
])
test_df = test_df.append(df)
```

```
C:\Users\Adam\AppData\Local\Temp\ipykernel_11508\1002997612.py:5: FutureWarning:
The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.
  test_df = test_df.append(df)
```

[10]:
```
Datadirectory = "ILY_TEST"
label = 25
df = pd.DataFrame(addDataToSet(Datadirectory,label), columns = ['label','pixel1','pixel2','pixe
```

```
C:\Users\Adam\AppData\Local\Temp\ipykernel_11508\3401862683.py:5: FutureWarning:
The frame.append method is deprecated and will be removed from pandas in a
future version. Use pandas.concat instead.
  test_df = test_df.append(df)
```

[11]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 27669 entries, 0 to 109
Columns: 785 entries, label to pixel784
dtypes: float64(785)
memory usage: 165.9 MB
```

[12]:
```
test_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 7251 entries, 0 to 39
Columns: 785 entries, label to pixel784
dtypes: float64(785)
memory usage: 43.5 MB
```

[13]:
```
for col in test_df:
    test_df[col] = test_df[col].astype('int')
test_df.tail()
```

[13]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	\
35	25	158	160	164	162	156	157	150	154	
36	25	166	165	161	157	155	150	156	156	
37	25	152	162	158	152	157	152	153	157	
38	25	160	159	159	161	154	154	156	158	
39	25	155	160	161	154	146	156	150	145	

	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	\
35	160	...	151	164	162	172	179	189	
36	155	...	120	168	175	180	186	192	

```
37     155  ...       139       166       170       174       188       197
38     161  ...       162       173       174       180       190       192
39     154  ...       161       164       175       173       190       195

    pixel781  pixel782  pixel783  pixel784
35       198       203       205       200
36       204       203       204       202
37       203       199       202       204
38       202       206       205       204
39       196       194       200       195

[5 rows x 785 columns]
```

[14]:
```
df = train_df.iloc[:,0]
#print(col)
print(train_df.iloc[:,0].unique())
```

```
[ 3.  6.  2. 13. 16.  8. 22. 18. 10. 20. 17. 19. 21. 23. 24.  1. 12. 11.
 15.  4.  0.  5.  7. 14.  9. 25.]
```

[15]:
```
train_df.describe()
```

[15]:
```
              label        pixel1        pixel2        pixel3        pixel4  \
count  27669.000000  27669.000000  27669.000000  27669.000000  27669.000000
mean      12.356753    145.434855    148.526474    151.256424    153.537497
std        7.305953     41.217814     39.803059     38.916585     38.454202
min        0.000000      0.000000      0.000000      0.000000      0.000000
25%        6.000000    122.000000    126.000000    130.000000    133.000000
50%       13.000000    150.000000    153.000000    155.000000    158.000000
75%       19.000000    174.000000    176.000000    178.000000    179.000000
max       25.000000    255.000000    255.000000    255.000000    255.000000

             pixel5        pixel6        pixel7        pixel8        pixel9  \
count  27669.000000  27669.000000  27669.000000  27669.000000  27669.000000
mean     156.169757    158.349272    160.410170    162.265604    163.889262
std       36.976847     35.997933     34.892493     33.553920     32.543516
min        0.000000      0.000000      0.000000      0.000000      0.000000
25%      137.000000    140.000000    142.000000    145.000000    146.000000
50%      160.000000    162.000000    163.000000    165.000000    166.000000
75%      180.000000    182.000000    183.000000    184.000000    185.000000
max      255.000000    255.000000    255.000000    255.000000    255.000000

       ...      pixel775      pixel776      pixel777      pixel778  \
count  ...  27669.000000  27669.000000  27669.000000  27669.000000
mean   ...    141.188550    147.576385    153.393292    159.165095
std    ...     63.614358     65.374612     64.300854     63.603236
min    ...      0.000000      0.000000      0.000000      0.000000
25%    ...     92.000000     96.000000    103.000000    112.000000
```

```
50%    ...    144.000000    163.000000    173.000000    180.000000
75%    ...    196.000000    202.000000    204.000000    207.000000
max    ...    255.000000    255.000000    255.000000    255.000000

           pixel779      pixel780      pixel781      pixel782      pixel783  \
count  27669.000000  27669.000000  27669.000000  27669.000000  27669.000000
mean     162.016517    162.804330    162.994506    162.080885    161.267122
std       63.632662     63.339645     63.400896     63.188798     63.498181
min        0.000000      0.000000      0.000000      0.000000      0.000000
25%      120.000000    125.000000    128.000000    128.000000    128.000000
50%      183.000000    184.000000    184.000000    183.000000    182.000000
75%      207.000000    207.000000    207.000000    206.000000    205.000000
max      255.000000    255.000000    255.000000    255.000000    255.000000

           pixel784
count  27669.000000
mean     159.932741
std       64.294434
min        0.000000
25%      126.000000
50%      182.000000
75%      204.000000
max      255.000000

[8 rows x 785 columns]
```

Zestaw danych train_df zawiera pierwszą kolumnę reprezentującą etykiety od 1 do 24. Etykiety są ładowane do oddzielnego DataFrame o nazwie „train_label", a kolumna „etykieta" jest usuwana z oryginalnego DataFrame, który teraz zawiera tylko 784 wartości pikseli, które wchodzą w skład pojedyńczego obrazu.

[16]:
```
train_label=train_df['label']
train_label.head()
trainset=train_df.drop(['label'],axis=1)
trainset.head()
```

[16]:
```
   pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  pixel9  \
0   107.0   118.0   127.0   134.0   139.0   143.0   146.0   150.0   153.0
1   155.0   157.0   156.0   156.0   156.0   157.0   156.0   158.0   158.0
2   187.0   188.0   188.0   187.0   187.0   186.0   187.0   188.0   187.0
3   211.0   211.0   212.0   212.0   211.0   210.0   211.0   210.0   210.0
4   164.0   167.0   170.0   172.0   176.0   179.0   180.0   184.0   185.0

   pixel10  ...  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0    156.0  ...     207.0     207.0     207.0     207.0     206.0     206.0
1    157.0  ...      69.0     149.0     128.0      87.0      94.0     163.0
2    186.0  ...     202.0     201.0     200.0     199.0     198.0     199.0
3    211.0  ...     235.0     234.0     233.0     231.0     230.0     226.0
```

```
4    186.0  ...       92.0     105.0     105.0     108.0     133.0     163.0

   pixel781  pixel782  pixel783  pixel784
0     206.0     204.0     203.0     202.0
1     175.0     103.0     135.0     149.0
2     198.0     195.0     194.0     195.0
3     225.0     222.0     229.0     163.0
4     157.0     163.0     164.0     179.0

[5 rows x 784 columns]
```

Po wycięciu etykiet ze zbioru danych należy zmienić strukture danych z DataFrame do tablicy numpy który jest wykorzystywany podczas uczenia modelu CNN. Macierze są konwertowane z 1-D na 3-D, co jest wymaganym wejściem do pierwszej warstwy CNN, z ciągu pixeli tworzymy macierz o wymiarze (28,28). Przeprowadzane kroki dla zbioru uczącego, zostały także wykonane dla zbioru testowego w poniższych blokach kodu.

[17]:
```
X_train = trainset.values
X_train = trainset.values.reshape(-1,28,28,1)
print(X_train.shape)
```

```
(27669, 28, 28, 1)
```

[18]:
```
test_label=test_df['label']
X_test=test_df.drop(['label'],axis=1)
print(X_test.shape)
X_test.head()
```

```
(7251, 784)
```

[18]:
```
   pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  pixel9  \
0     149     149     150     150     150     151     151     150     151
1     126     128     131     132     133     134     135     135     136
2      85      88      92      96     105     123     135     143     147
3     203     205     207     206     207     209     210     209     210
4     188     191     193     195     199     201     202     203     203

   pixel10  ...  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0      152  ...       138       148       127        89        82        96
1      138  ...        47       104       194       183       186       184
2      152  ...        68       166       242       227       230       227
3      209  ...       154       248       247       248       253       236
4      203  ...        26        40        64        48        29        46

   pixel781  pixel782  pixel783  pixel784
0       106       112       120       107
1       184       184       182       180
2       226       225       224       222
3       230       240       253       255
```

```
4        49        46        46        53

[5 rows x 784 columns]
```

Konwersja etykiet liczb całkowitych do postaci binarnej

Ramka danych etykiety składa się z pojedynczych wartości od 1 do 24 dla każdego pojedynczego obrazu. Warstwa wyjściowa CNN będzie składać się z 25 węzłów, ponieważ jako klasyfikator z wieloma etykietami ma 25 różnych etykiet. Stąd każda liczba całkowita jest zakodowana w tablicy binarnej o rozmiarze 25, przy czym odpowiadająca jej etykieta wynosi 1, a wszystkie inne etykiety są równe 0. Na przykład, jeśli y=4, tablica to [0 0 0 1 0 0.....0]. Do przeprowadzenia tej konswersji wykorzystaliśmy pakiet LabelBinarizer z pakietu sklearn.preprocessing. #

Examples

```
>>> from sklearn import preprocessing
>>> lb = preprocessing.LabelBinarizer()
>>> lb.fit([1, 2, 6, 4, 2])
LabelBinarizer()
>>> lb.classes_
array([1, 2, 4, 6])
>>> lb.transform([1, 6])
array([[1, 0, 0, 0],
       [0, 0, 0, 1]])
```

[19]:
```
lb=LabelBinarizer()
y_train=lb.fit_transform(train_label)
y_test=lb.fit_transform(test_label)
y_train
```

[19]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 1, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 1],
       [0, 0, 0, ..., 0, 0, 1],
       [0, 0, 0, ..., 0, 0, 1]])
```

[20]:
```
X_test=X_test.values.reshape(-1,28,28,1)
```

[21]:
```
print(X_train.shape,y_train.shape,X_test.shape,y_test.shape)
```

```
(27669, 28, 28, 1) (27669, 26) (7251, 28, 28, 1) (7251, 26)
```

Rozszerzanie zbioru danych

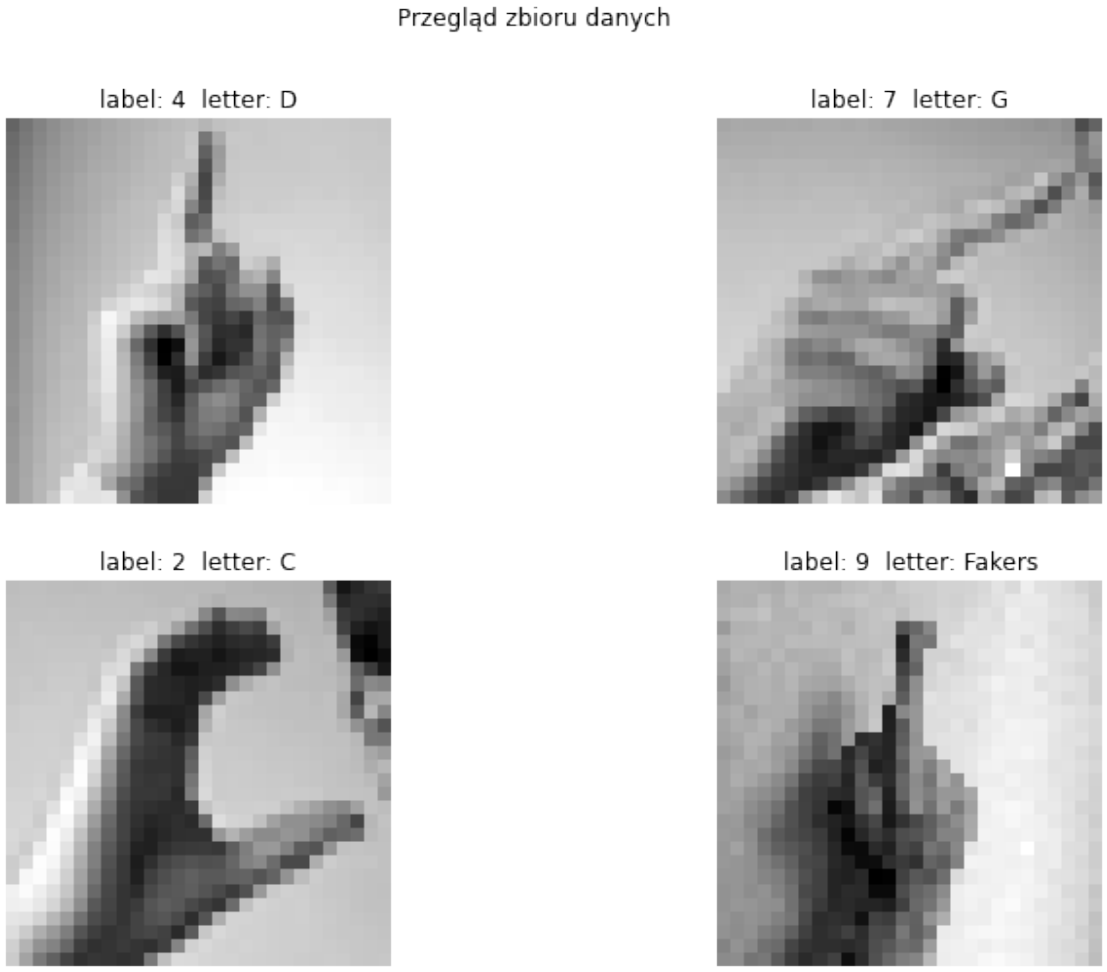
Pakiet ImageDataGenerator z pakietu keras.preprocessing.image umożliwia dodawanie różnych zniekształceń do zbioru danych obrazów poprzez zapewnienie losowego obracania, powiększania/pomniejszania, skalowania wysokości lub szerokości itp. Zestaw danych obrazu jest tutaj również znormalizowany przy użyciu parametru rescale, który dzieli każdy piksel przez 255, tak że wartości pikseli mieszczą się w zakresie od 0 do 1. Dodatkowo dane testowe są także są normalizowane jednak nie zastowaliśmy na nich operacji rozszerzania zbioru.

[22]:
```
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   rotation_range = 0,
                                   height_shift_range=0.2,
                                   width_shift_range=0.2,
                                   shear_range=0,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   fill_mode='nearest')

X_test=X_test/255
```

6 Przegląd danych wykorzystanych do nauki i testowania modelu CNN

[23]:
```
#plt.figure(figsize = (18,8))
fig,axe=plt.subplots(2,2,figsize = (12,8))
fig.suptitle('Przegląd zbioru danych')
axe[0,0].imshow(X_train[0].reshape(28,28),cmap='gray')
axe[0,0].set_title('label: 4  letter: D')
axe[0,0].axis('off')
axe[0,1].imshow(X_train[1].reshape(28,28),cmap='gray')
axe[0,1].set_title('label: 7  letter: G')
axe[0,1].axis('off')
axe[1,0].imshow(X_train[2].reshape(28,28),cmap='gray')
axe[1,0].set_title('label: 2  letter: C')
axe[1,0].axis('off')
axe[1,1].imshow(X_train[27530].reshape(28,28),cmap='gray')
axe[1,1].set_title('label: 9  letter: Fakers')
axe[1,1].axis('off')
```

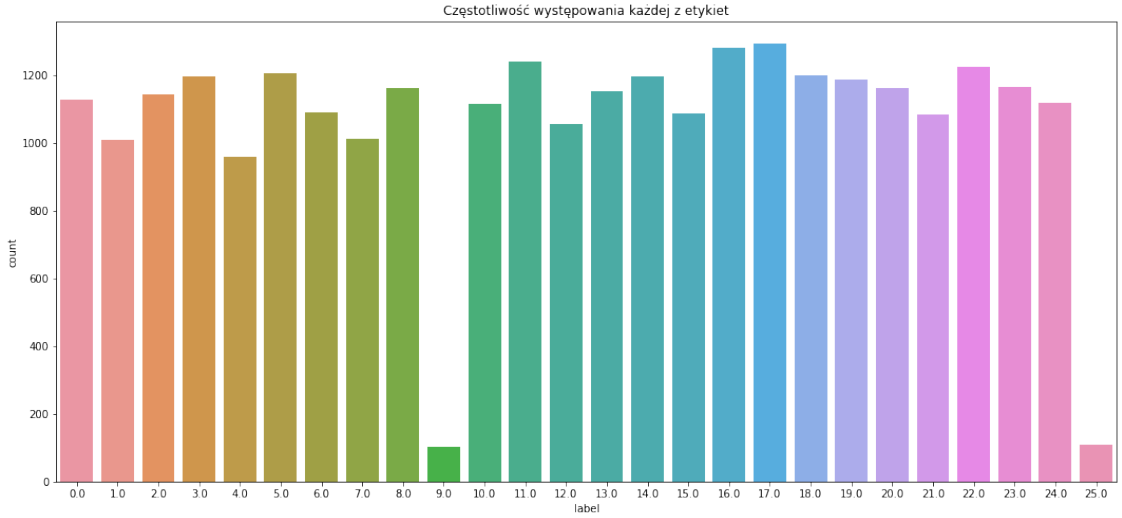
[23]:
```
(-0.5, 27.5, 27.5, -0.5)
```



[24]:
```
plt.figure(figsize = (18,8))
sns.countplot(train_label)
plt.title("Częstotliwość występowania każdej z etykiet")
```

```
c:\Users\Adam\AppData\Local\Programs\Python\Python310\lib\site-
packages\seaborn\_decorators.py:36: FutureWarning: Pass the following variable
as a keyword arg: x. From version 0.12, the only valid positional argument will
be `data`, and passing other arguments without an explicit keyword will result
in an error or misinterpretation.
  warnings.warn(
```

[24]:
```
Text(0.5, 1.0, 'Częstotliwość występowania każdej z etykiet')
```



7 Budowa przygotowanego modelu CNN

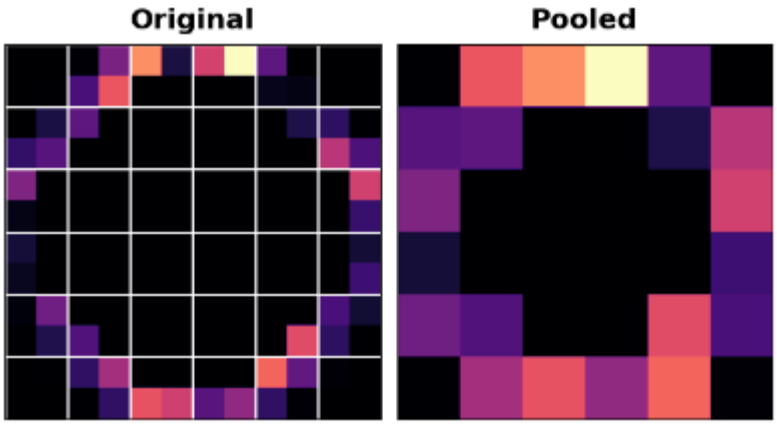
Model składa się z następujących elementów : 1. Trzy warstwy konwolucji, po każdej z nich występuje warstwa `pooling`, która jest odpowiedzialna za skalowanie obrazu w dół dla lepszego rozpoznawania cech 2. Sieć złożona z 512 neuronów 3. Warstwa wyjścia która składa się z 26 klas

Właściwości warstw przygotowanego modelu

1 warstwa konwolucji wytwarza 128 obrazów zawierających różnorakie cechy, które otrzymujemy za pomocą 128 różnych filtrów/jąder. Obrazy te generowane są za pomocą funkcji jądra o wymiarach 3x3. Za pomocą parametru `strides` jesteśmy w stanie manipulować o ile pixeli ma się przesuwać jądro kolejnych krokach generacji obrazu cech. W tej warstwie parametr ten ma wartość 1. Jako funkcję aktywacji wybraliśmy domyślną funkcję ReLu. Parametr padding z wartością `same` zapewnia nas że rozmiar obrazu cech jest taki sam jak obraz wejściowy.

2 warstwa konwolucji 1 warstwa konwolucji wytwarza 64 obrazów zawierających różnorakie cechy. Obrazy te generowane są za pomocą funkcji jądra o wymiarach 2x2. Pozostałe parametry są takie same jak w poprzedniej warstwie

2 warstwa konwolucji 1 warstwa konwolucji wytwarza 32 obrazów zawierających różnorakie cechy. Obrazy te generowane są za pomocą funkcji jądra o wymiarach 2x2. Pozostałe parametry są takie same jak w poprzedniej warstwie



8

Jeśli chodzi o warstwy MaxPool, zostały wykorzystane 3 warstwy kondensujące obraz do mniejszych rozmiarów w celu przyśpieszenia obliczeń.

MaxPool wartwa 1:wymiar jądra 3x3; parametr przesunięcia jądra 2

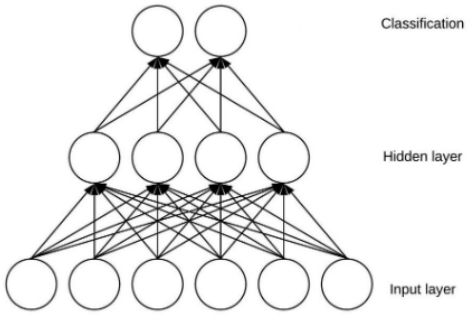
MaxPool wartwa 2:wymiar jądra 2x2; parametr przesunięcia jądra 2

MaxPool wartwa 3:wymiar jądra 2x2; parametr przesunięcia jądra 2

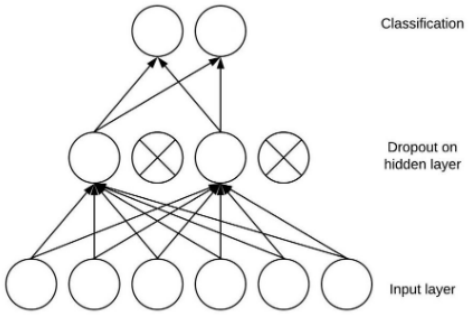
[25]:
```
model=Sequential()
model.add(Conv2D(128,kernel_size=(5,5),
                 ␣
 ↪strides=1,padding='same',activation='relu',input_shape=(28,28,1)))
model.add(MaxPool2D(pool_size=(3,3),strides=2,padding='same'))
model.add(Conv2D(64,kernel_size=(2,2),
                 strides=1,activation='relu',padding='same'))
model.add(MaxPool2D((2,2),2,padding='same'))
model.add(Conv2D(32,kernel_size=(2,2),
                 strides=1,activation='relu',padding='same'))
model.add(MaxPool2D((2,2),2,padding='same'))

model.add(Flatten())
```

Opisanie węzłów wejściowych oraz warstwy wyjścia

Po przepuszczeniu obrazu przez wszystkie przygotowane warstwy naszego modelu otrzymujemy 32 obrazy o wymiarach 4x4 które następnie konwertujemy do postaci wektora jednowymiarowego o długości 512. Tą wiedze wykorzystujemy do utworzenia wartwy wejścia naszej sieci. Jeśli chodzi o parametr Droput jest on wyjaśniony na poniższym rysunku. W skrócie poprzez ustawienia progu wszystkie węzły z ostatniej warstwy sieci, które mają mniejszą od jego ustalonej wartości zostają odrzucane. Warstwa wyjść jest złożon z 26 klas ponieważ tyle znaków posiadamy w naszym zbiorze danych.



Without Dropout



With Dropout

9

[26]:
```
model.add(Dense(units=512,activation='relu'))
model.add(Dropout(rate=0.25))
model.add(Dense(units=26,activation='softmax'))
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 128)       3328

 max_pooling2d (MaxPooling2D  (None, 14, 14, 128)      0
 )

 conv2d_1 (Conv2D)           (None, 14, 14, 64)        32832

 max_pooling2d_1 (MaxPooling  (None, 7, 7, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 7, 7, 32)          8224

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 32)         0
 2D)

 flatten (Flatten)           (None, 512)               0

 dense (Dense)               (None, 512)               262656

 dropout (Dropout)           (None, 512)               0

 dense_1 (Dense)             (None, 26)                13338

=================================================================
Total params: 320,378
Trainable params: 320,378
Non-trainable params: 0
```

```
----------------------------------------------------------------
```

[27]:
```
model.
 ↪compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
```

Trenowanie modelu

Do trenowania wykorzystujemy wcześniej utworzony obiekt "train_datagen", który generuje dodatkowe obrazy(przesunięte, przybliżone, obrócone horyzontalnie lub wertykalnie). Ilość dodatkowych danych zależy od parametru "batch_size", przyjeliśmy wartość 200, która daje bardzo zadowalające wyniki podczas trenowania. Metoda fit potrzebuje informacji ile wynosi liczba epok przez które będzie uczył się model, przekazujemy także dane testowe w celu sprawdzenia efektywności modelu.

[28]:
```
model.fit(train_datagen.flow(X_train,y_train,batch_size=200),
          epochs = 50,
           validation_data=(X_test,y_test),
           shuffle=1
          )
```

```
Epoch 1/50
139/139 [==============================] - 36s 258ms/step - loss: 3.1057 -
accuracy: 0.0808 - val_loss: 2.6182 - val_accuracy: 0.2655
Epoch 2/50
139/139 [==============================] - 38s 272ms/step - loss: 2.2927 -
accuracy: 0.2819 - val_loss: 1.3725 - val_accuracy: 0.5572
Epoch 3/50
139/139 [==============================] - 37s 268ms/step - loss: 1.6765 -
accuracy: 0.4506 - val_loss: 1.1541 - val_accuracy: 0.5969
Epoch 4/50
139/139 [==============================] - 34s 241ms/step - loss: 1.3096 -
accuracy: 0.5670 - val_loss: 0.8558 - val_accuracy: 0.6878
Epoch 5/50
139/139 [==============================] - 36s 258ms/step - loss: 1.0711 -
accuracy: 0.6411 - val_loss: 0.7295 - val_accuracy: 0.7395
Epoch 6/50
139/139 [==============================] - 41s 295ms/step - loss: 0.9206 -
accuracy: 0.6882 - val_loss: 0.5504 - val_accuracy: 0.8133
Epoch 7/50
139/139 [==============================] - 39s 279ms/step - loss: 0.8037 -
accuracy: 0.7290 - val_loss: 0.4795 - val_accuracy: 0.8395
Epoch 8/50
139/139 [==============================] - 34s 248ms/step - loss: 0.6832 -
accuracy: 0.7685 - val_loss: 0.3304 - val_accuracy: 0.9058
Epoch 9/50
139/139 [==============================] - 36s 258ms/step - loss: 0.5991 -
accuracy: 0.7980 - val_loss: 0.3008 - val_accuracy: 0.9035
Epoch 10/50
139/139 [==============================] - 33s 240ms/step - loss: 0.5502 -
```

```
accuracy: 0.8138 - val_loss: 0.2361 - val_accuracy: 0.9253
Epoch 11/50
139/139 [==============================] - 34s 242ms/step - loss: 0.4794 -
accuracy: 0.8406 - val_loss: 0.1891 - val_accuracy: 0.9441
Epoch 12/50
139/139 [==============================] - 36s 257ms/step - loss: 0.4436 -
accuracy: 0.8499 - val_loss: 0.1551 - val_accuracy: 0.9560
Epoch 13/50
139/139 [==============================] - 35s 251ms/step - loss: 0.3974 -
accuracy: 0.8647 - val_loss: 0.1264 - val_accuracy: 0.9686
Epoch 14/50
139/139 [==============================] - 35s 253ms/step - loss: 0.3633 -
accuracy: 0.8782 - val_loss: 0.0975 - val_accuracy: 0.9799
Epoch 15/50
139/139 [==============================] - 34s 243ms/step - loss: 0.3388 -
accuracy: 0.8860 - val_loss: 0.1406 - val_accuracy: 0.9526
Epoch 16/50
139/139 [==============================] - 38s 276ms/step - loss: 0.3096 -
accuracy: 0.8953 - val_loss: 0.0962 - val_accuracy: 0.9777
Epoch 17/50
139/139 [==============================] - 36s 260ms/step - loss: 0.2988 -
accuracy: 0.8990 - val_loss: 0.0995 - val_accuracy: 0.9750
Epoch 18/50
139/139 [==============================] - 37s 268ms/step - loss: 0.2726 -
accuracy: 0.9076 - val_loss: 0.1028 - val_accuracy: 0.9648
Epoch 19/50
139/139 [==============================] - 37s 269ms/step - loss: 0.2510 -
accuracy: 0.9166 - val_loss: 0.0807 - val_accuracy: 0.9742
Epoch 20/50
139/139 [==============================] - 37s 264ms/step - loss: 0.2337 -
accuracy: 0.9228 - val_loss: 0.0650 - val_accuracy: 0.9839
Epoch 21/50
139/139 [==============================] - 37s 264ms/step - loss: 0.2277 -
accuracy: 0.9248 - val_loss: 0.0551 - val_accuracy: 0.9866
Epoch 22/50
139/139 [==============================] - 37s 263ms/step - loss: 0.2157 -
accuracy: 0.9283 - val_loss: 0.0544 - val_accuracy: 0.9844
Epoch 23/50
139/139 [==============================] - 34s 245ms/step - loss: 0.2024 -
accuracy: 0.9314 - val_loss: 0.0849 - val_accuracy: 0.9720
Epoch 24/50
139/139 [==============================] - 38s 273ms/step - loss: 0.1994 -
accuracy: 0.9322 - val_loss: 0.0513 - val_accuracy: 0.9822
Epoch 25/50
139/139 [==============================] - 39s 279ms/step - loss: 0.1912 -
accuracy: 0.9366 - val_loss: 0.0432 - val_accuracy: 0.9895
Epoch 26/50
139/139 [==============================] - 36s 260ms/step - loss: 0.1896 -
```

```
accuracy: 0.9357 - val_loss: 0.0398 - val_accuracy: 0.9876
Epoch 27/50
139/139 [==============================] - 37s 265ms/step - loss: 0.1655 -
accuracy: 0.9445 - val_loss: 0.1038 - val_accuracy: 0.9686
Epoch 28/50
139/139 [==============================] - 38s 271ms/step - loss: 0.1693 -
accuracy: 0.9429 - val_loss: 0.0303 - val_accuracy: 0.9926
Epoch 29/50
139/139 [==============================] - 41s 292ms/step - loss: 0.1531 -
accuracy: 0.9478 - val_loss: 0.0214 - val_accuracy: 0.9939
Epoch 30/50
139/139 [==============================] - 38s 275ms/step - loss: 0.1547 -
accuracy: 0.9476 - val_loss: 0.0341 - val_accuracy: 0.9892
Epoch 31/50
139/139 [==============================] - 38s 270ms/step - loss: 0.1481 -
accuracy: 0.9492 - val_loss: 0.0249 - val_accuracy: 0.9919
Epoch 32/50
139/139 [==============================] - 39s 277ms/step - loss: 0.1416 -
accuracy: 0.9520 - val_loss: 0.0275 - val_accuracy: 0.9899
Epoch 33/50
139/139 [==============================] - 35s 252ms/step - loss: 0.1387 -
accuracy: 0.9536 - val_loss: 0.0260 - val_accuracy: 0.9903
Epoch 34/50
139/139 [==============================] - 37s 269ms/step - loss: 0.1333 -
accuracy: 0.9553 - val_loss: 0.0322 - val_accuracy: 0.9903
Epoch 35/50
139/139 [==============================] - 37s 268ms/step - loss: 0.1281 -
accuracy: 0.9576 - val_loss: 0.0430 - val_accuracy: 0.9861
Epoch 36/50
139/139 [==============================] - 38s 276ms/step - loss: 0.1292 -
accuracy: 0.9578 - val_loss: 0.0196 - val_accuracy: 0.9959
Epoch 37/50
139/139 [==============================] - 36s 256ms/step - loss: 0.1158 -
accuracy: 0.9616 - val_loss: 0.0335 - val_accuracy: 0.9888
Epoch 38/50
139/139 [==============================] - 36s 261ms/step - loss: 0.1114 -
accuracy: 0.9618 - val_loss: 0.0120 - val_accuracy: 0.9979
Epoch 39/50
139/139 [==============================] - 36s 256ms/step - loss: 0.1168 -
accuracy: 0.9613 - val_loss: 0.0216 - val_accuracy: 0.9916
Epoch 40/50
139/139 [==============================] - 35s 252ms/step - loss: 0.1183 -
accuracy: 0.9607 - val_loss: 0.0176 - val_accuracy: 0.9945
Epoch 41/50
139/139 [==============================] - 34s 244ms/step - loss: 0.1192 -
accuracy: 0.9595 - val_loss: 0.0236 - val_accuracy: 0.9935
Epoch 42/50
139/139 [==============================] - 36s 261ms/step - loss: 0.1085 -
```

```
accuracy: 0.9647 - val_loss: 0.0232 - val_accuracy: 0.9891
Epoch 43/50
139/139 [==============================] - 33s 240ms/step - loss: 0.1110 -
accuracy: 0.9626 - val_loss: 0.0304 - val_accuracy: 0.9895
Epoch 44/50
139/139 [==============================] - 34s 247ms/step - loss: 0.1084 -
accuracy: 0.9640 - val_loss: 0.0131 - val_accuracy: 0.9943
Epoch 45/50
139/139 [==============================] - 34s 247ms/step - loss: 0.0931 -
accuracy: 0.9681 - val_loss: 0.0362 - val_accuracy: 0.9866
Epoch 46/50
139/139 [==============================] - 36s 256ms/step - loss: 0.1031 -
accuracy: 0.9652 - val_loss: 0.0278 - val_accuracy: 0.9908
Epoch 47/50
139/139 [==============================] - 37s 268ms/step - loss: 0.1040 -
accuracy: 0.9648 - val_loss: 0.0261 - val_accuracy: 0.9923
Epoch 48/50
139/139 [==============================] - 39s 283ms/step - loss: 0.0993 -
accuracy: 0.9674 - val_loss: 0.0177 - val_accuracy: 0.9960
Epoch 49/50
139/139 [==============================] - 35s 251ms/step - loss: 0.0907 -
accuracy: 0.9694 - val_loss: 0.0216 - val_accuracy: 0.9921
Epoch 50/50
139/139 [==============================] - 36s 261ms/step - loss: 0.0879 -
accuracy: 0.9710 - val_loss: 0.0213 - val_accuracy: 0.9927
```

[28]:
```
<keras.callbacks.History at 0x27509096b60>
```

[29]:
```
model.save("finalModel.h5")
```

[30]:
```
model = tf.keras.models.load_model("finalModel.h5")
```

Ocena modelu

[31]:
```
(ls,acc)=model.evaluate(x=X_test,y=y_test)
```

```
227/227 [==============================] - 3s 10ms/step - loss: 0.0213 -
accuracy: 0.9927
```

[32]:
```
print('MODEL ACCURACY = {}%'.format(acc*100))
```

```
MODEL ACCURACY = 99.26906824111938%
```

[33]:
```
def getLetter(result):
    classLabels = { 0: 'A',
                    1: 'B',
                    2: 'C',
                    3: 'D',
                    4: 'E',
```

```
                    5: 'F',
                    6: 'G',
                    7: 'H',
                    8: 'I',
                    9: 'FAK',
                    10: 'K',
                    11: 'L',
                    12: 'M',
                    13: 'N',
                    14: 'O',
                    15: 'P',
                    16: 'Q',
                    17: 'R',
                    18: 'S',
                    19: 'T',
                    20: 'U',
                    21: 'V',
                    22: 'W',
                    23: 'X',
                    24: 'Y',
                    25: 'ILY'}
    try:
        res = int(result)
        return classLabels[res]
    except:
        return "Error"

def keras_predict(model, image):
        pred_probab = model.predict(image)[0]
        pred_class = list(pred_probab).index(max(pred_probab))
        return max(pred_probab), pred_class
```

10 Praktyczne wykorzystanie wytrenowanego modelu

10.0.1 Aplikacja rozpoznająca symbole w czasie rzeczywistym z wykorzystaniem komputerowej kamery

Import bibliotek

[34]:
```
import cv2
import numpy as np
from PIL import Image
from keras import models
import time
from time import time
import tensorflow as tf
```

OpenCV2 Istotnym elementem działania aplikacji jest biblioteka openCV2, która pozwala na implementację licznych technik związanych z widzeniem maszynowym w czasie rzeczywistym. Dłoń przedstawiająca symbol powinna znajdować się wewnątrz wyznaczonego obszaru ekranu. Zabieg wydzielenia dozwolonego obszaru zwiększył skuteczność prawidłowego klasyfikowania symboli, ponieważ model jest czuły na obraz wejściowy (najlepszy efekt można uzyskać na jednolitym, jasnym tle). Pokazywanemu symbolowi zostaje przypisana i wyświetlona odpowiadająca mu cecha ze słownika.

[ ]:
```
cap = cv2.VideoCapture(0)

model = models.load_model('finalModel.h5')


while True:

    ret, frame = cap.read()

    ##############################
    frame=cv2.flip(frame, 1)

    #define region of interest
    roi = frame[100:400, 320:620]
    #cv2.imshow('roi', roi)
    roi = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
    roi = cv2.resize(roi, (28, 28), interpolation = cv2.INTER_AREA)

    cv2.imshow('roi scaled and gray', roi)
    copy = frame.copy()
    cv2.rectangle(copy, (320, 100), (620, 400), (255,0,0), 5)

    roi = roi.reshape(1,28,28,1)
    roi = roi/255
    pred_probab, pred_class = keras_predict(model,roi)

    print(pred_probab*100)
        #print(X_train[np.argmax(Y_proba)])

    #wynik = np.argmax(Y_proba)
    litera = getLetter(pred_class)
    cv2.putText(copy, litera, (300 , 100), cv2.FONT_HERSHEY_COMPLEX, 2, (0, 255,␣
↪0), 2)
    cv2.imshow('frame', copy)

    if cv2.waitKey(1) == 13: #13 is the Enter Key
        break
```

[39]:
```
cap.release()
cv2.destroyAllWindows()
```