Splątane sieci neuronowe CNN - architektura LeNet-5

mgr inż. Grzegorz Kossakowski

17.10.2024

1. Opis architektury

LeNet-5 jest to pierwsza sieć splątana [1] jaka powstała. Została zbudowana w 1999 przez Yann LeCun, Leon Bottou, Yoshua Bengio i Patricka Haffnera. Celem było rozpoznawanie numerów kodów pocztowych napisanych przez ludzi na listach. Dzięki temu, że projekt odniósł sukces, znaleziono praktyczne zastosowanie dla tej technologii. Zawiera ona dwie warstwy splatane oraz sieć neuronową. W tej sieci wykorzystuje warstwę flaten do spłaszczenia obrazów po przejściu przez warstwy splatane oraz dwie warstwy ukryte gęste. Ostatnia warstwa, jest warstwa wyjściowa, która zawiera 10 neuronów, czyli dokładnie tyle ile jest kategorii w naszych danych.

2. Pobranie potrzebnych bibliotek

Kolejnym krokiem jest wczytanie wszystkich potrzebnych bibliotek, dzięki którym będzie możliwe wykorzystanie ich w procesie klasyfikacji.

[2]:
```
TF_ENABLE_ONEDNN_OPTS=0
from astropy.io import fits
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Conv2D, AveragePooling2D, Flatten, Dense
from keras.callbacks import ReduceLROnPlateau
from keras.optimizers import Adam
from astroNN.datasets import galaxy10sdss
import pandas as pd
from datetime import date
import pathlib
import datetime
from sklearn.metrics import accuracy_score
```

3. Pobranie danych z pliku fits

Dlatego że wcześniej podzieliliśmy dane na odpowiednie części, teraz pobieramy dwa zbiory. Pierwszy z nich to zbiór, na którym będziemy uczyć nasz model. Drugi to zbiór walidacyjny.

[3]:
```
hdu_train = fits.open('Data/train.fits')
hdu_valid = fits.open('Data/valid.fits')
hdu_test = fits.open('Data/test.fits')
x_train = hdu_train[0].data
y_train = hdu_train[1].data
```

```
x_valid = hdu_valid[0].data
y_valid = hdu_valid[1].data
x_test = hdu_test[0].data
y_test = hdu_test[1].data
```

[4]:
```
x_train.shape, x_valid.shape, x_test.shape, type(x_train)
```

[4]:
```
((11350, 256, 256, 3), (2838, 256, 256, 3), (3548, 256, 256, 3), numpy.ndarray)
```

4. Pobranie danych

W tym kroku pobieramy dane, a następnie przygotowujemy je do klasyfikacji. Modele głębokiej sieci neuronowej [4] wymaga danych z zakresu 0..1, dlatego wszystkie wartości w danych są dzielone przez 255. Powodem takiego zachowania jest fakt, że dane obrazów są przechowywane w zakresie liczb 0..255. Dzielenie przez 255 powoduje, że dane zostaną zapisane w zakresie od 0..1, zgodnie z wymaganiami modelu.

[5]:
```
reduceLR = ReduceLROnPlateau(monitor='accuracy', factor=.001, patience=1,␣
 ↪min_delta=0.01, mode="auto")
x_train = x_train / 255.0
x_valid = x_valid / 255.0
x_test = x_test/ 255.0
```

5. Budowa modelu.

Model można podzielić na dwie podstawowe części. Pierwsza część to warstwy splątane. Naprzemiennie są układane warstwy Conv2D, AveragePooling2D. Kolejność ułożenia warstw jest zgodna z LeNet5. Kolejną część to już model głębokiego uczenia. Pierwszą warstwo jest flatten. Zadaniem tej warstwy jest spłaszczenie obrazu po przejściu warstw splątanych do pojedynczego ciągu. Kolejne warstwy to warstwy ukryte z aktywatorem tanh. Czyli funkcja tangensa hiperbolicznego. Jest bardzo podobna do sigmoidu, ale rozciągnieta w zakresie od -1 do 1. Ostatnią warstwą jest gęsto połączona warstwa wyjściowa. W naszym modelu klasyfikacja odbywa się dla 10 kategorii dlatego właśnie zawiera tyle neuronów.

[6]:
```
model = Sequential()
model.add(Conv2D(filters=6, kernel_size=(5,5), strides=(1,1), activation='tanh',␣
 ↪input_shape=(256,256,3)))
model.add(AveragePooling2D(pool_size=(2,2), strides=(2,2)))
model.add(Conv2D(filters=16, kernel_size=(5,5), strides=(1,1),␣
 ↪activation='tanh'))
model.add(AveragePooling2D(pool_size=(2,2), strides=(2,2)))
model.add(Flatten())
model.add(Dense(units=120, activation='tanh'))
model.add(Dense(units=84, activation='tanh'))
model.add(Dense(units=10, activation='softmax'))
model_optimizer = Adam(learning_rate=0.001)
```

```
model.compile(optimizer=model_optimizer, loss='sparse_categorical_crossentropy',␣
 ↪metrics=["accuracy"])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 252, 252, 6)       456

 average_pooling2d (Average  (None, 126, 126, 6)       0
 Pooling2D)

 conv2d_1 (Conv2D)           (None, 122, 122, 16)      2416

 average_pooling2d_1 (Avera  (None, 61, 61, 16)        0
 gePooling2D)

 flatten (Flatten)           (None, 59536)             0

 dense (Dense)               (None, 120)               7144440

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 7158326 (27.31 MB)
Trainable params: 7158326 (27.31 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

6. Uczenie

W tym momencie model zaczyna proces uczenia. Czyli otrzymuje dwa zbiory danych i etykiet. Pierwszy z nich to dane, na podstawie których model się uczy. Drugi mniejszy zbiór jest zbiorem walidacyjnym, który pozwala na sprawdzenie postępów w nauce, na danych, których model jeszcze nie widział. Pozwala to ocenić postępy w nauce już w czasie uczenia. Kolejny zbiór danych zostanie wykorzystany na końcu celem ostatecznego sprawdzenia poprawności działania modelu.

[7]:
```
now = datetime.datetime.now()
history = model.fit(x_train, y_train, epochs=10,␣
 ↪callbacks=[reduceLR],validation_data=(x_valid, y_valid))
time = datetime.datetime.now()-now
print("Potrzebny czas do wykonania operacji to: ",int(time.seconds/60)," minut")
```

```
Epoch 1/10
355/355 [==============================] - 42s 115ms/step - loss: 2.2537 -
```

```
accuracy: 0.1402 - val_loss: 2.2709 - val_accuracy: 0.1388 - lr: 0.0010
Epoch 2/10
355/355 [==============================] - 33s 94ms/step - loss: 2.2353 -
accuracy: 0.1434 - val_loss: 2.2404 - val_accuracy: 0.1508 - lr: 0.0010
Epoch 3/10
355/355 [==============================] - 33s 93ms/step - loss: 2.2432 -
accuracy: 0.1480 - val_loss: 2.2374 - val_accuracy: 0.1508 - lr: 1.0000e-06
Epoch 4/10
355/355 [==============================] - 33s 93ms/step - loss: 2.2415 -
accuracy: 0.1480 - val_loss: 2.2374 - val_accuracy: 0.1508 - lr: 1.0000e-09
Epoch 5/10
355/355 [==============================] - 33s 94ms/step - loss: 2.2415 -
accuracy: 0.1480 - val_loss: 2.2374 - val_accuracy: 0.1508 - lr: 1.0000e-12
Epoch 6/10
355/355 [==============================] - 34s 95ms/step - loss: 2.2415 -
accuracy: 0.1480 - val_loss: 2.2374 - val_accuracy: 0.1508 - lr: 1.0000e-15
Epoch 7/10
355/355 [==============================] - 34s 95ms/step - loss: 2.2415 -
accuracy: 0.1480 - val_loss: 2.2374 - val_accuracy: 0.1508 - lr: 1.0000e-18
Epoch 8/10
355/355 [==============================] - 33s 94ms/step - loss: 2.2415 -
accuracy: 0.1480 - val_loss: 2.2374 - val_accuracy: 0.1508 - lr: 1.0000e-21
Epoch 9/10
355/355 [==============================] - 34s 95ms/step - loss: 2.2415 -
accuracy: 0.1480 - val_loss: 2.2374 - val_accuracy: 0.1508 - lr: 1.0000e-24
Epoch 10/10
355/355 [==============================] - 33s 94ms/step - loss: 2.2415 -
accuracy: 0.1480 - val_loss: 2.2374 - val_accuracy: 0.1508 - lr: 1.0000e-27
Potrzebny czas do wykonania operacji to:  6  minut
```

7. Zapis architektury

[8]:
```
model.save('Models/LeNet5_full.keras')
```

8. Zapis otrzymanych danych podczas nauki

Po zakończeniu uczenia zapisujemy dane, które otrzymaliśmy podczas uczenie do pliku CSV. Pozwoli nam to później przeanalizować dane w późniejszym czasie.

[9]:
```
historyModelLearning = pd.DataFrame()
historyModelLearning['loss'] = history.history['loss']
historyModelLearning['accuracy'] = history.history['accuracy']
historyModelLearning['val_loss'] = history.history['val_loss']
historyModelLearning['val_accuracy'] = history.history['val_accuracy']
historyModelLearning.to_csv('ResultLearning/LeNet5_full.csv', index=True)
```

9. Sprawdzenie uzyskanych wyników

Celem tego elementu jest wstępne sprawdzenie uzyskanych wyników. Pozwoli to na porównanie wyników z predykcją w zapisanym modelu. Dzięki temu uzyskamy informację czy otrzymane wyniku różnią się od siebie.

[10]:
```
predict = model.predict(x_test).argmax(axis=1)
print("Otrzymany wynik to: ",(accuracy_score(y_test, predict)*100)," %")
```

```
111/111 [==============================] - 3s 24ms/step
Otrzymany wynik to:  15.135287485907552  %
```

Literatura

1. Bartosz Michalski, Małgorzata Plechawska-Wójcik, Porównanie modeli LeNet-5, AlexNet i GoogLeNet w rozpoznawaniu pisma ręcznego, 2022
2. https://builtin.com/machine-learning/relu-activation-function
3. https://datascience.eu/pl/uczenie-maszynowe/relu-funkcja-aktywujaca