Głębokie sieci neuronowe DNN

mgr inż. Grzegorz Kossakowski

17.10.2024

1. Opis architektury

Głęboka sieć neuronowa [1], jest to prosta sieć bez żadnej warstwy splatanej. Składa się z warstwy wejściowej flaten, której podstawowym zadaniem jest wypłaszczenie przekazanego obrazu. Kolejna warstwa to jedna warstwa ukryta oraz warstwy wyjściowej. Celem tego notebook jest uzyskanie danych do porównania, jak mocno poprawą się wyniki po zastosowaniu warstw splątanych.

2. Pobranie potrzebnych bibliotek

Kolejnym kroku wczytujemy wszystkie potrzebne biblioteki, dzięki którym będzie możliwe wykorzystanie ich w procesie uczenia i zapisywania modelu oraz danych.

[2]:
```
TF_ENABLE_ONEDNN_OPTS=0
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Flatten, Dense
from keras.callbacks import ReduceLROnPlateau
from keras.optimizers import Adam
from astropy.io import fits
import pandas as pd
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 uczący, na którym będziemy uczyć nasz model. Drugi to zbiór walidacyjny, na tym zbiorze będziemy sprawdzać, jak uczy się nasz model.

[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. Przygotowanie danych

Modele głębokiej sieci neuronowej [2] 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.

Budowany model jest modelem warstwowym i jako pierwsza warstwa jest to warstwa flatten. Zadaniem tej warstwy jest spłaszczenie obrazu z wymiarów 256 * 256 na pojedynczy ciąg, jest to warstwa wejściowa. Kolejną warstwą jest warstwa ukryta z aktywatorem RELU. Aktywator ten powoduje, że każdy otrzymany wynik ujemy, zostaje zamieniony na zero [3][4]. Pozwala to na przełamanie liniowości procesu. Ostatnią warstwą jest gęsto połączona warstwa wyjściowa. W naszym modelu klasyfikacja odbywa się dla 10 kategorii, dlatego zawiera dokładnie 10 neuronów.

[6]:
```
model = Sequential()
model.add(Flatten(input_shape=(256,256,3)))
model.add(Dense(128, activation='relu'))
model.add(Dense(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 #
=================================================================
 flatten (Flatten)           (None, 196608)            0

 dense (Dense)               (None, 128)               25165952

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 25167242 (96.01 MB)
```

```
Trainable params: 25167242 (96.01 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

6. Uczenie

W tym momencie model zaczyna proces uczenia. Czyli otrzymuje dwa zbiory danych. Pierwszy z nich to dane, na podstawie których model się uczy, czyli zbiór uczący. Drugi mniejszy zbiór jest zbiorem walidacyjnym, który pozwala na sprawdzenie postępów w nauce, na danych, których model jeszcze nie widział. Dzięki temu możemy ocenić postępy w nauce już w czasie uczenia.

[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 [==============================] - 48s 132ms/step - loss: 3.9176 -
accuracy: 0.1922 - val_loss: 2.2946 - val_accuracy: 0.1508 - lr: 0.0010
Epoch 2/10
355/355 [==============================] - 38s 106ms/step - loss: 2.2770 -
accuracy: 0.1468 - val_loss: 2.2581 - val_accuracy: 0.1508 - lr: 0.0010
Epoch 3/10
355/355 [==============================] - 38s 106ms/step - loss: 2.2583 -
accuracy: 0.1480 - val_loss: 2.2580 - val_accuracy: 0.1508 - lr: 1.0000e-06
Epoch 4/10
355/355 [==============================] - 38s 106ms/step - loss: 2.2582 -
accuracy: 0.1480 - val_loss: 2.2580 - val_accuracy: 0.1508 - lr: 1.0000e-09
Epoch 5/10
355/355 [==============================] - 38s 107ms/step - loss: 2.2582 -
accuracy: 0.1480 - val_loss: 2.2580 - val_accuracy: 0.1508 - lr: 1.0000e-12
Epoch 6/10
355/355 [==============================] - 38s 106ms/step - loss: 2.2582 -
accuracy: 0.1480 - val_loss: 2.2580 - val_accuracy: 0.1508 - lr: 1.0000e-15
Epoch 7/10
355/355 [==============================] - 38s 106ms/step - loss: 2.2582 -
accuracy: 0.1480 - val_loss: 2.2580 - val_accuracy: 0.1508 - lr: 1.0000e-18
Epoch 8/10
355/355 [==============================] - 38s 106ms/step - loss: 2.2582 -
accuracy: 0.1480 - val_loss: 2.2580 - val_accuracy: 0.1508 - lr: 1.0000e-21
Epoch 9/10
355/355 [==============================] - 38s 106ms/step - loss: 2.2582 -
accuracy: 0.1480 - val_loss: 2.2580 - val_accuracy: 0.1508 - lr: 1.0000e-24
Epoch 10/10
355/355 [==============================] - 38s 106ms/step - loss: 2.2582 -
accuracy: 0.1480 - val_loss: 2.2580 - val_accuracy: 0.1508 - lr: 1.0000e-27
Potrzebny czas do wykonania operacji to:  7  minut
```

7. Zapis architektury

Jednak my nie będziemy testować od razu naszego modelu. Do tego celu przygotujemy oddzielny notebook. Dlatego, aby nie utracić naszej pracy, zapiszemy nas wyuczony model do pliku.

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

8. Zapis otrzymanych wyników podczas nauki

Po zakończeniu uczenia zapisujemy wyniki, które otrzymaliśmy podczas uczenie modelu do pliku CSV. Pozwoli nam to później przeanalizować proces uczenia i walidacji i porównać te dane z różnymi modelami.

[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/DDN_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 [==============================] - 1s 11ms/step
Otrzymany wynik to:  15.135287485907552  %
```

Literatura

1. Tenzin Migmar 2021 Galaxy Multi-Image Classification with LeNet-5 (Jupiter NoteBook)
2. Paweł Krakowiak, Deep learning w języku Python — Konwolucyjne Sieci Neuronowe
3. https://builtin.com/machine-learning/relu-activation-function
4. https://datascience.eu/pl/uczenie-maszynowe/relu-funkcja-aktywujaca/