Splątane sieci neuronowe CNN - architektura AlexNet

mgr inż. Grzegorz Kossakowski

17.10.2024

1. Opis architektury

AlexNet [1][2] jest architekturą splątanych sieci neuronowych CNN. Została stworzona przez Alex Kizhevsky i Ilya Sutskever. W pracach uczestniczył również Geoffrey Hinton, który był promotorem doktoratu as Krizhevsky. Jest bardziej rozbudowany niż LeNet5. Głównym celem było uczestnictwo w zawodach ImageNet Large Scale Visual Recognition Challenge (ILSVRC), które wygrał w 2012 roku.

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 keras.callbacks import ReduceLROnPlateau
from keras.optimizers import Adam
from keras import Sequential
from tests.test_layers import Dense, Flatten
from keras.layers import Conv2D, Dropout, Flatten, Dense, MaxPool2D,␣
↪BatchNormalization
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.

[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, BatchNormalization oraz MaxPool2D. Kolejność ułożenia warstw jest zgodna z AlexNet. Kolejne warstwy zaczynające się od warstwy flatten są zwykłym modelem głębokiego uczenia. Zadaniem warstwy flaten jest spłaszczenie obrazu z wymiarów 256*256 na pojedynczy ciąg. 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 właśnie taka wartość jest wybrana.

[6]:
```
model = Sequential()
model.add(Conv2D(filters=96, kernel_size=(11, 11), strides=(4, 4),␣
 ↪activation="relu", input_shape=(256, 256, 3)))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(3, 3), strides= (2, 2)))
model.add(Conv2D(filters=256, kernel_size=(5, 5), strides=(1, 1),␣
 ↪activation="relu", padding="same"))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(3, 3), strides=(2, 2)))
model.add(Conv2D(filters=384, kernel_size=(3, 3), strides=(1, 1),␣
 ↪activation="relu", padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(filters=384, kernel_size=(3, 3), strides=(1, 1),␣
 ↪activation="relu", padding="same"))
model.add(BatchNormalization())
model.add(Conv2D(filters=256, kernel_size=(3, 3), strides=(1, 1),␣
 ↪activation="relu", padding="same"))
model.add(BatchNormalization())
```

```
model.add(MaxPool2D(pool_size=(3, 3), strides=(2, 2)))
model.add(Flatten())
model.add(Dense(4096, activation="relu"))
model.add(Dropout(0.5))
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 #
=================================================================
 conv2d (Conv2D)             (None, 62, 62, 96)        34944

 batch_normalization (Batch  (None, 62, 62, 96)        384
 Normalization)

 max_pooling2d (MaxPooling2  (None, 30, 30, 96)        0
 D)

 conv2d_1 (Conv2D)           (None, 30, 30, 256)       614656

 batch_normalization_1 (Bat  (None, 30, 30, 256)       1024
 chNormalization)

 max_pooling2d_1 (MaxPoolin  (None, 14, 14, 256)       0
 g2D)

 conv2d_2 (Conv2D)           (None, 14, 14, 384)       885120

 batch_normalization_2 (Bat  (None, 14, 14, 384)       1536
 chNormalization)

 conv2d_3 (Conv2D)           (None, 14, 14, 384)       1327488

 batch_normalization_3 (Bat  (None, 14, 14, 384)       1536
 chNormalization)

 conv2d_4 (Conv2D)           (None, 14, 14, 256)       884992

 batch_normalization_4 (Bat  (None, 14, 14, 256)       1024
 chNormalization)

 max_pooling2d_2 (MaxPoolin  (None, 6, 6, 256)         0
 g2D)
```

```
 flatten (Flatten)           (None, 9216)              0

 dense (Dense)               (None, 4096)              37752832

 dropout (Dropout)           (None, 4096)              0

 dense_1 (Dense)             (None, 10)                40970

=================================================================
Total params: 41546506 (158.49 MB)
Trainable params: 41543754 (158.48 MB)
Non-trainable params: 2752 (10.75 KB)
_________________________________________________________________
```

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.

[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 [==============================] - 255s 711ms/step - loss: 5.4553 -
accuracy: 0.2071 - val_loss: 2.5066 - val_accuracy: 0.1920 - lr: 0.0010
Epoch 2/10
355/355 [==============================] - 247s 696ms/step - loss: 2.0252 -
accuracy: 0.2663 - val_loss: 1.9046 - val_accuracy: 0.2999 - lr: 0.0010
Epoch 3/10
355/355 [==============================] - 245s 691ms/step - loss: 1.8727 -
accuracy: 0.3031 - val_loss: 1.7225 - val_accuracy: 0.3386 - lr: 0.0010
Epoch 4/10
355/355 [==============================] - 244s 688ms/step - loss: 1.7589 -
accuracy: 0.3395 - val_loss: 1.5415 - val_accuracy: 0.4091 - lr: 0.0010
Epoch 5/10
355/355 [==============================] - 243s 686ms/step - loss: 1.7524 -
accuracy: 0.3441 - val_loss: 1.7676 - val_accuracy: 0.3520 - lr: 0.0010
Epoch 6/10
355/355 [==============================] - 241s 679ms/step - loss: 1.6696 -
accuracy: 0.3543 - val_loss: 1.5769 - val_accuracy: 0.3999 - lr: 1.0000e-06
Epoch 7/10
355/355 [==============================] - 242s 681ms/step - loss: 1.6730 -
```

```
accuracy: 0.3514 - val_loss: 1.5646 - val_accuracy: 0.4006 - lr: 1.0000e-06
Epoch 8/10
355/355 [==============================] - 244s 686ms/step - loss: 1.6559 -
accuracy: 0.3582 - val_loss: 1.5639 - val_accuracy: 0.4003 - lr: 1.0000e-09
Epoch 9/10
355/355 [==============================] - 242s 682ms/step - loss: 1.6518 -
accuracy: 0.3574 - val_loss: 1.5644 - val_accuracy: 0.4006 - lr: 1.0000e-12
Epoch 10/10
355/355 [==============================] - 240s 676ms/step - loss: 1.6544 -
accuracy: 0.3667 - val_loss: 1.5645 - val_accuracy: 0.4006 - lr: 1.0000e-15
Potrzebny czas do wykonania operacji to:  41  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/AlexNet_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 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/AlexNet_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 [==============================] - 20s 177ms/step
Otrzymany wynik to:  42.13641488162345  %
```

Literatura

1. https://medium.com/@siddheshb008/alexnet-architecture-explained-b6240c528bd5 dostęp 4.10.2024
2. Bartosz Michalski, Małgorzata Plechawska-Wójcik, Porównanie modeli LeNet-5, AlexNet i GoogLeNet w rozpoznawaniu pisma ręcznego, 2022

3. https://builtin.com/machine-learning/relu-activation-function
4. https://datascience.eu/pl/uczenie-maszynowe/relu-funkcja-aktywujaca