Konwolucyjna sieć neuronowa

Patryk Jurewicz

Importujemy tensorflow

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
```

Pobieramy i przygotowujemy zbior danych

Zbiór danych CIFAR10 zawiera 60 000 kolorowych obrazów w 10 klasach, po 6 000 obrazów w każdej klasie. Zbiór danych jest podzielony na 50 000 obrazów szkoleniowych i 10 000 obrazów testowych. Klasy wykluczają się wzajemnie i nie ma między nimi nakładania się.

```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```


```
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
```

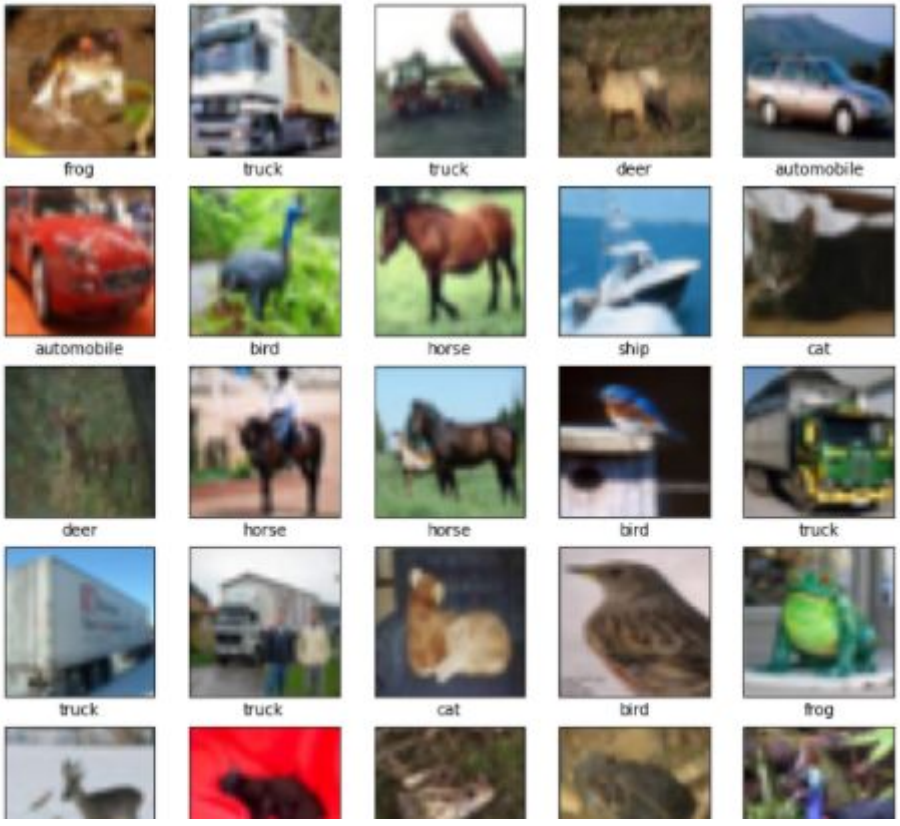

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 11s 0us/step
```

Weryfikujemy dane

Aby sprawdzić, czy zestaw danych wygląda poprawnie, wykreślmy pierwsze 25 obrazów z zestawu uczącego i wyświetlmy nazwę klasy poniżej każdego obrazu.

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    # The CIFAR labels happen to be arrays,
    # which is why you need the extra index
    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```



Tworzymy bazę konwolucyjną

Poniższe 6 wierszy kodu definiuje podstawę splotu przy użyciu wspólnego wzoru: stosu warstw Conv2D i MaxPooling2D .

Jako dane wejściowe CNN przyjmuje tensory kształtu (wysokość_obrazu, szerokość_obrazu, kanały_koloru), ignorując rozmiar partii. Jeśli nie znasz tych wymiarów, color_channels odnosi się do (R, G, B). W tym przykładzie skonfigurujesz nasz CNN do przetwarzania danych wejściowych w kształcie (32, 32, 3), który jest formatem obrazów CIFAR. Możesz to zrobić, przekazując argument input_shape do naszej pierwszej warstwy.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Wskazmy architekturę

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
=================================================================
Total params: 56,320
```

Powyżej widzimy, że dane wyjściowe każdej warstwy Conv2D i MaxPooling2D to trójwymiarowy tensor kształtu (wysokość, szerokość, kanały). Wymiary szerokości i wysokości mają tendencję do kurczenia się w miarę zagłębiania się w sieć. Liczba kanałów

wyjściowych dla każdej warstwy Conv2D jest kontrolowana przez pierwszy argument (np. 32 lub 64). Zwykle w miarę zmniejszania się szerokości i wysokości można pozwolić sobie (obliczeniowo) na dodanie większej liczby kanałów wyjściowych w każdej warstwie Conv2D.

Dodajemy gęste warstwy na wierzchu

Aby ukończyć nasz model, podasz ostatni tensor wyjściowy z podstawy splotu (o kształcie (4, 4, 64)) do jednej lub więcej warstw gęstych w celu przeprowadzenia klasyfikacji. Gęste warstwy przyjmują wektory jako dane wejściowe (które są 1D), podczas gdy wyjście prądowe jest tensorem 3D. Najpierw spłaszczysz (lub rozwiniesz) wyjście 3D do 1D, a następnie dodasz jedną lub więcej gęstych warstw na górze. CIFAR ma 10 klas wyjściowych, więc używasz końcowej warstwy gęstej z 10 wyjściami.

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

Architektura pełna modelu

```
model.summary()
```

```
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 1024)              0
_________________________________________________________________
dense (Dense)                (None, 64)                65600
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 122,570
Trainable params: 122,570
Non-trainable params: 0
_________________________________________________________________
```

Jak widać, nasze (4, 4, 64) wyjścia zostały spłaszczone do wektorów kształtu (1024) przed przejściem przez dwie warstwy Gęste.

Kompilujemy i trenujemy model

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

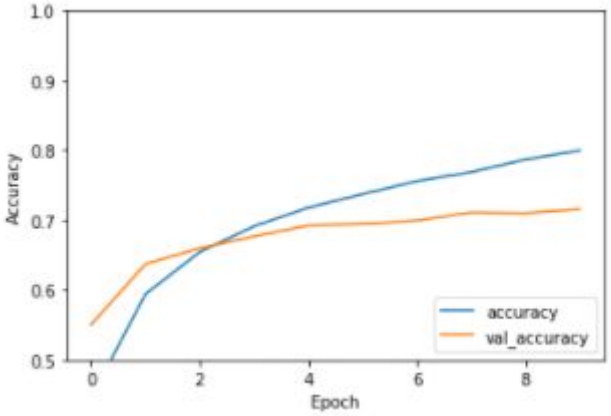
```
Epoch 1/10
1563/1563 [==============================] - 18s 4ms/step - loss: 1.7606 - accuracy: 0.3488 - val_loss: 1.2753 - val_accuracy: 0.5504
Epoch 2/10
1563/1563 [==============================] - 6s 4ms/step - loss: 1.1977 - accuracy: 0.5751 - val_loss: 1.0409 - val_accuracy: 0.6371
Epoch 3/10
1563/1563 [==============================] - 6s 4ms/step - loss: 1.0137 - accuracy: 0.6439 - val_loss: 0.9613 - val_accuracy: 0.6597
Epoch 4/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.8844 - accuracy: 0.6908 - val_loss: 0.9272 - val_accuracy: 0.6766
Epoch 5/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.7950 - accuracy: 0.7205 - val_loss: 0.8712 - val_accuracy: 0.6923
Epoch 6/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.7410 - accuracy: 0.7388 - val_loss: 0.8894 - val_accuracy: 0.6943
Epoch 7/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.6924 - accuracy: 0.7547 - val_loss: 0.8980 - val_accuracy: 0.6993
Epoch 8/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.6453 - accuracy: 0.7728 - val_loss: 0.8695 - val_accuracy: 0.7109
Epoch 9/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.5999 - accuracy: 0.7881 - val_loss: 0.8887 - val_accuracy: 0.7098
Epoch 10/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.5461 - accuracy: 0.8088 - val_loss: 0.8840 - val_accuracy: 0.7157
```

Oceń model

```
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.5, 1])
plt.legend(loc='lower right')

test_loss, test_acc = model.evaluate(test_images,  test_labels, verbose=2)
```

313/313 - 1s - loss: 0.8840 - accuracy: 0.7157




```
print(test_acc)
```

0.7156999707221985

Kod zaprezentowany powyżej osiągnął dokładność neuronową na poziomie 70%.

Źródło:

https://www.tensorflow.org/
https://www.tensorflow.org/tutorials/images/cnn
https://www.tensorflow.org/api_docs/python/tf/GradientTape