Metody Rozpoznawania Obrazów
Zadanie 02 - Wyciskanie soku z cytryny

Wojciech Michaluk

16.10.2025

Opis kolejnych etapów zadania

Korzystam z frameworku **keras**. W tym frameworku można łatwo załadować interesujący nas zbiór *CIFAR-10*:

```
import keras

(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()
```

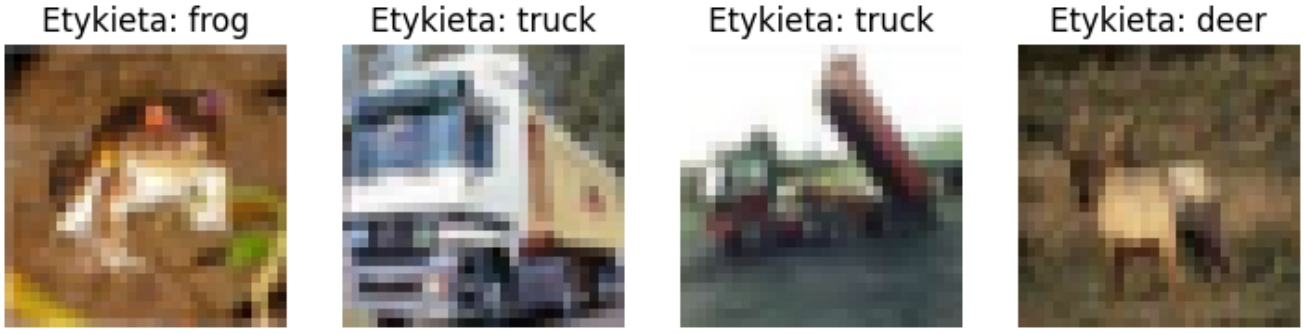
Listing 1: Załadowanie zbioru danych

Aby mieć postać tekstową (a nie liczbową) etykiet obrazów tworzę słownik z nazwami klas:

```
# na podstawie: https://www.cs.toronto.edu/~kriz/cifar.html
classes = {
    0: 'airplane',
    1: 'automobile',
    2: 'bird',
    3: 'cat',
    4: 'deer',
    5: 'dog',
    6: 'frog',
    7: 'horse',
    8: 'ship',
    9: 'truck'
}
```

Listing 2: Mapowanie numerów etykiet na nazwy klas

Wyświetlam przykładowe obrazki wraz z ich etykietami (są to 4 pierwsze obrazki ze zbioru treningowego):



Przy projektowaniu architektury "bazowej" sieci neuronowej chcę odtworzyć sieć z polecenia zadania. W tym miejscu również inicjalizuję sieć właściwymi wartościami losowymi (*He initialization* dla funkcji aktywacji **relu**). W keras-ie realizuje się to za pomocą parametru *kernel_initializer* z wartością "he_normal". Także inicjalizuję *bias* zerami.

```python
from keras import models, layers

# architektura sieci
model = models.Sequential()
model.add(layers.Conv2D(
    6, (5, 5), activation='relu', kernel_initializer='he_normal',
    bias_initializer='zeros', input_shape=(32, 32, 3)
))

model.add(layers.AveragePooling2D((2, 2)))
model.add(layers.Conv2D(
    16, (5, 5), activation='relu', kernel_initializer='he_normal',
    bias_initializer='zeros'
))

model.add(layers.AveragePooling2D((2, 2)))
model.add(layers.Flatten())
model.add(
    layers.Dense(120, activation='relu', kernel_initializer='he_normal',
    bias_initializer='zeros')
)

model.add(
    layers.Dense(84, activation='relu', kernel_initializer='he_normal',
    bias_initializer='zeros')
)

model.add(layers.Dense(10, activation='softmax', bias_initializer='zeros'))
```

Listing 3: Architektura bazowej sieci neuronowej

Poleceniem `model.summary()` wyświetlam architekturę sieci oraz liczbę parametrów:

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 6)	456
average_pooling2d (AveragePooling2D)	(None, 14, 14, 6)	0
conv2d_1 (Conv2D)	(None, 10, 10, 16)	2,416
average_pooling2d_1 (AveragePooling2D)	(None, 5, 5, 16)	0
flatten (Flatten)	(None, 400)	0
dense (Dense)	(None, 120)	48,120
dense_1 (Dense)	(None, 84)	10,164
dense_2 (Dense)	(None, 10)	850

```
Total params: 62,006 (242.21 KB)

Trainable params: 62,006 (242.21 KB)

Non-trainable params: 0 (0.00 B)
```

Zanim przejdę do treningu sieci, przygotowuję dane, żeby były w odpowiednim formacie.

```
from keras.utils import to_categorical

# przygotowanie danych
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

Listing 4: Przygotowanie danych do treningu

W keras-ie trzeba jeszcze *skompilować* model. To właśnie tutaj ustalam używany optymalizator (*Adam*, z domyślnymi parametrami) oraz funkcję kosztu - **Categorical Cross-Entropy**. Używam metryki dokładności (accuracy).

```
# przygotowanie modelu do treningu
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Listing 5: Kompilowanie modelu

Finalnie wydzielam zbiór walidacyjny (ze zbioru treningowego) i przystępuję do treningu. Przyjmuję "bazowo" 20 epok treningu oraz rozmiar batcha wynoszący 64.

```
from sklearn.model_selection import train_test_split

# wydzielenie zbioru walidacyjnego ze zbioru treningowego
x_train, x_val, y_train, y_val = train_test_split(
    x_train,
    y_train,
    test_size=0.1,
    random_state=42
)

# trening modelu
model.fit(x_train, y_train, epochs=20, batch_size=64, validation_data=(x_val, y_val))
```

Listing 6: Podział danych i trening modelu

Jak widać, nie piszę tutaj pętli treningowej "ręcznie", tylko korzystam z narzędzi oferowanych przez framework - mam nadzieję, że jest to w porządku. Dzięki tej konstrukcji **accuracy** oraz **loss** na zbiorze treningowym oraz walidacyjnym są raportowane co epokę treningu.
Po treningu dokonuję ewaluacji na zbiorze testowym z wykorzystaniem funkcji `model.evaluate()`. Otrzymuję bazowy wynik accuracy wynoszący **0.5999**. Tak niski wynik może świadczyć o *underfitting*'u - sieć jest "za prosta" dla postawionego problemu.

Dalej zgodnie z poleceniem korzystam z biblioteki `Optuna` do przeszukiwania przestrzeni hiperparametrów. Na początku definiuję możliwe wartości hiperparametrów:

```
# korzystam z sugerowanej biblioteki Optuna
import optuna
from optuna.integration import TFKerasPruningCallback

# zakresy wartości dla hiperparametrów
learning_rates = (1e-5, 1e-2)
dropout_rates = (0.2, 0.5)
filters_1 = (4, 12)
filters_2 = (12, 24)

# ustalone wartości do wyboru
max_epochs = [15, 20, 30]
batch_sizes = [32, 64, 128]
```

Listing 7: Zdefiniowanie hiperparametrów i ich możliwych wartości

Przedstawiam funkcję tworzącą sieć z wybranymi parametrami, trenującą ją oraz zwracającą *accuracy* na zbiorze testowym w całej okazałości:

```
# funkcja tworzy sieć i zwraca 'accuracy' wytrenowanej sieci na zbiorze testowym
def objective(trial):
    # wybór hiperparametrów
    fil_1 = trial.suggest_int('filters_1', filters_1[0], filters_1[1])
    fil_2 = trial.suggest_int('filters_2', filters_2[0], filters_2[1])
    dropout_rate = trial.suggest_float(
        'dropout_rate', dropout_rates[0], dropout_rates[1]
    )

    learning_rate = trial.suggest_float(
        'learning_rate', learning_rates[0], learning_rates[1], log=True
    )

    batch_size = trial.suggest_categorical('batch_size', batch_sizes)
    epochs = trial.suggest_categorical('epochs', max_epochs)

    # budowa modelu z wybranymi parametrami
    model = models.Sequential()
    model.add(layers.Conv2D(
        fil_1, (5, 5), activation='relu', kernel_initializer='he_normal',
        bias_initializer='zeros', input_shape=(32, 32, 3)
    ))

    model.add(layers.AveragePooling2D((2, 2)))
    model.add(layers.Conv2D(
        fil_2, (5, 5), activation='relu', kernel_initializer='he_normal',
        bias_initializer='zeros'
    ))

    model.add(layers.AveragePooling2D((2, 2)))
    model.add(layers.Flatten())
    model.add(layers.Dense(
        120, activation='relu', kernel_initializer='he_normal', bias_initializer='zeros'
    ))

    model.add(layers.Dropout(dropout_rate)) # dodana warstwa dropout'u
    model.add(layers.Dense(
        84, activation='relu', kernel_initializer='he_normal', bias_initializer='zeros'
    ))

    model.add(layers.Dropout(dropout_rate)) # dodana warstwa dropout'u
    model.add(layers.Dense(10, activation='softmax', bias_initializer='zeros'))

    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
        loss='categorical_crossentropy',
        metrics=['accuracy']
    )

    # callback: pruner, żeby szybciej przerywać słabe próby
    prune_callback = TFKerasPruningCallback(trial, 'val_accuracy')

    model.fit(
        x_train, y_train, epochs=epochs, batch_size=batch_size,
```

```
        validation_data=(x_val, y_val), callbacks=[prune_callback], verbose=0
    )

    _, test_acc = model.evaluate(x_test, y_test, batch_size=batch_size)

    return test_acc
```

Listing 8: Funkcja wydzielająca tworzenie modelu dla wybranych hiperparametrów

Na podstawie powyższego, jako hiperparametry przyjmuję:

- liczbę filtrów w obu warstwach konwolucyjnych,
- stałą uczącą (learning rate) dla optymalizatora Adam,
- prawdopodobieństwo dropout'u,
- liczbę epok treningu,
- rozmiar batcha.

Poza samym przeszukiwaniem hiperparametrów, stosuję ulepszenia w postaci warstw dropout'u, które dodaję po warstwach *fully-connected* (`layers.Dense()`) oraz *pruning* - na podstawie dokładności na zbiorze walidacyjnym niektóre próby są przerywane, jeżeli naprawdę "źle rokują".
Powyższa funkcja jest użyta jako *objective* dla obiektu `Study` używanej biblioteki, który chcemy zmaksymalizować.

```
# przeszukiwanie przestrzeni hiperparametrów
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=30)
```

Listing 9: Uruchomienie poszukiwania hiperparametrów

Przyjmuję 30 iteracji poszukiwania - uwzględniając sprune'owane próby, całość liczyła się około 30 minut na moim komputerze, co wydaje się rozsądnym progiem.
Najlepsza konfiguracja została znaleziona w iteracji nr 26. Osiągnięto dokładność na zbiorze testowym **0.6607**. Wartości parametrów wynoszą odpowiednio:

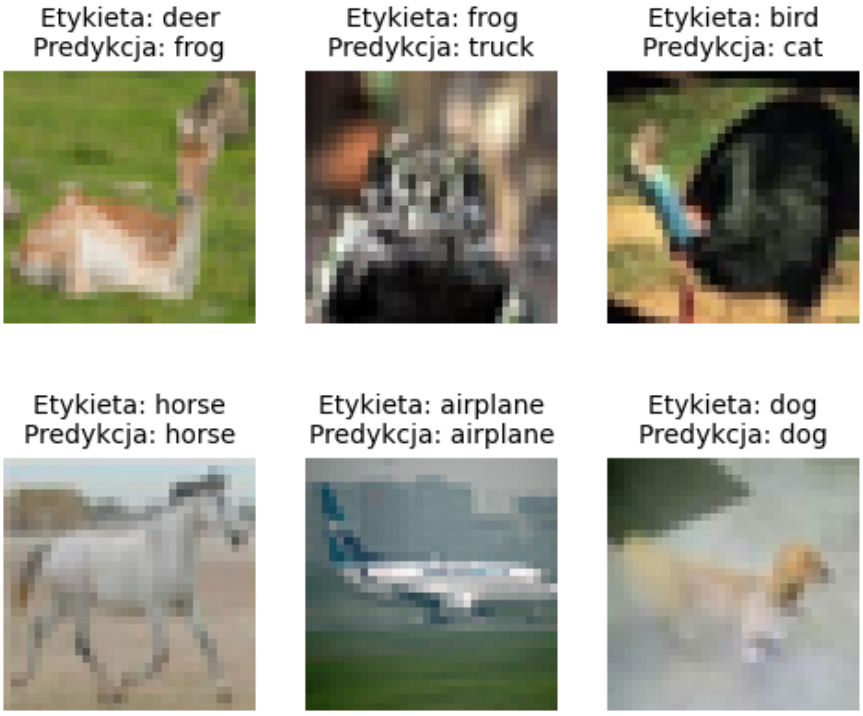
1. 'filters_1': 11,
2. 'filters_2': 19,
3. 'dropout_rate': 0.2745,
4. 'learning_rate': 0.000772448577,
5. 'batch_size': 64,
6. 'epochs': 20.

Wnioski

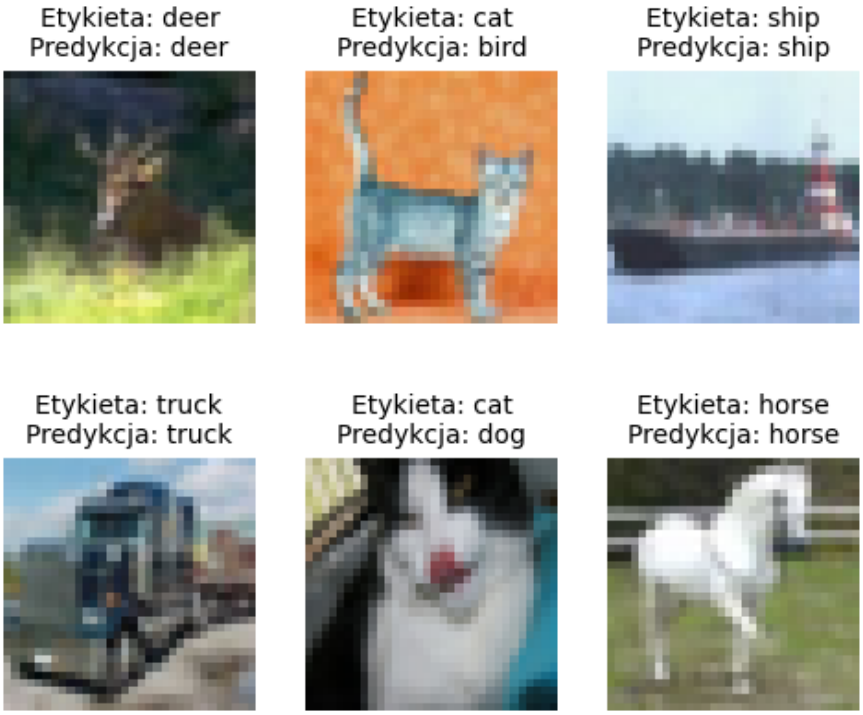
Optymalizacja hiperparametrów przyniosła poprawę wyniku, ale nie jest ona bardzo wysoka - dokładność na poziomie 66% jest wciąż nienajlepszym wynikiem. Zapewne zmiana sieci polegająca na dodaniu kolejnej warstwy konwolucyjnej, z większą liczbą filtrów mogłaby pomóc, ale wydaje mi się, że jest to zbyt duża ingerencja - powstaje sieć o znacznie odmiennej architekturze. Dlatego w aspekcie samej architektury sieci manipuluję jedynie liczbą filtrów w już istniejących warstwach oraz dodaję dropout, żeby zmniejszyć przeuczenie (*overfitting*).
Podobnie jak dodanie nowych warstw, według mnie np. *augmentacja* danych wykracza poza zakres "ulepszeń", które powinny zostać przetestowane. Powyższe sprawia, że potencjalny underfitting nadal jest problemem.
Z kolei już samo zwiększenie liczby filtrów (co za tym idzie, więcej parametrów do nauczenia) powinno poprawić wynik. Natomiast specyfika danych utrudnia uzyskanie wysokiej dokładności, ponieważ są to kolorowe obrazki o wymiarach $32x32$ - dość małe, niską rozdzielczość widać na przykładzie obrazków pokazanych na początku dokumentu (oraz poniżej). W niektórych przypadkach być może nawet człowiek mógłby się pomylić (gdyby się dokładnie nie przyjrzał).

Predykcja sieci

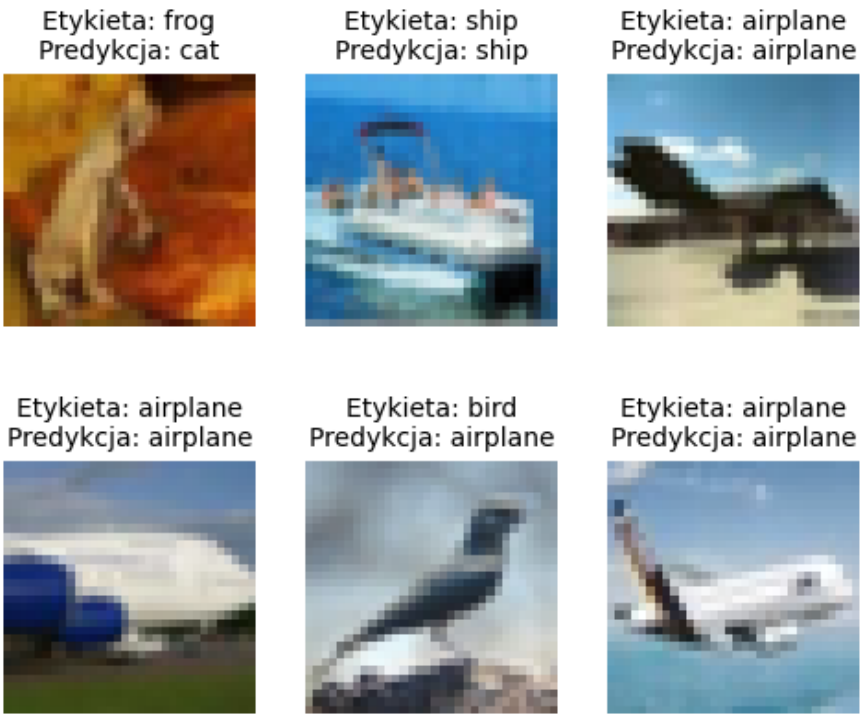
Dla najlepszej konfiguracji (najlepszej sieci) przeprowadzam predykcję etykiety dla 6 losowo wybranych obrazków ze zbioru testowego. Powtórzyłem ten eksperyment trzykrotnie:



Rysunek 1: Pierwsza predykcja - 3/6 obrazków poprawnie sklasyfikowanych



Rysunek 2: Druga predykcja - 4/6 obrazków poprawnie sklasyfikowanych



Rysunek 3: Trzecia predykcja - 4/6 obrazków poprawnie sklasyfikowanych

Jak widać sieci zdarza się mylić i to nierzadko. W szczególności problematyczne wydają się **zwierzęta** (pomyłki deer -> frog, bird -> cat, cat -> bird, cat -> dog oraz frog -> cat). Poza tym frog -> truck i bird -> airplane (podobieństwo tych kategorii się nasuwa, ale czy dla tych konkretnych obrazków?). W przypadku **środków transportu** - truck, ship, airplane sytuacja wygląda dużo lepiej.
Niektóre pomyłki można zrozumieć (np. frog -> cat, tutaj sam mógłbym się pomylić - może pora pójść do okulisty), natomiast widać, że nawet najlepsza sieć nie radzi sobie dobrze z tym zbiorem danych i nie jest do niego sensownie dostosowana - tego można było się spodziewać, patrząc na uzyskany wynik dokładności.