Raport - Wykrywanie samochodów na zdjęciach za pomocą konwolucyjnej sieci neuronowej (CNN)

Chylak Michał 97176
Naglak Tomasz 98821

Spis treści

Cel zadania

Celem zadania jest praktyczne zastosowanie konwolucyjnych sieci neuronowych do klasyfikacji obrazów. Model CNN, który ma za zadanie rozpoznawanie obrazów przedstawiających samochody.

Przygotowania

1. Zapoznanie się z materiałami o CNN.
2. Instalacja potrzebnych bibliotek: Zainstaluj Keras i TensorFlow za pomocą Anaconda lub pip. Dodatkowo, zainstaluj bibliotekę Pillow do przetwarzania obrazów.
3. Wykorzystanie wydajnego sprzętu: Zalecane jest użycie komputera z dostępem do GPU.

Zapoznaliśmy się z załączoną w zadaniu literaturą oraz innymi źródłami dostępnymi w internecie. Zdecydowaliśmy się na pracę nad zadaniem lokalnie na naszych komputerach. Kod utrzymywany był za pomoca narzędzia git i utworzonego na potrzeby zadania repozytorium. Nie udało nam się skonfigurować środowiska w taki sposób by napisany przez nas program korzystał z GPU, ale moc CPU pozwalała uczenie modelu w zadowalającym dla nas tempie (szybszym niż za pomocą notatnika Google Colab).

Link do repozytorium git: https://github.com/Michauue/machine_learning/tree/final

Krok 1: Wstępne przetwarzanie danych

Ładowanie danych: Użyj ImageDataGenerator z Keras do wczytania i przeskalowania obrazów ze zbioru danych PASCAL VOC 2010 lub podobnego. Parametry generatora powinny obejmować m.in. skalowanie wartości pikseli, rozmiar obrazu, wielkość partii, tryb klasyfikacji oraz losowość próbek.

Ten etap projektu nie był wymagający, polegał tylko na zastosowaniu wskazanego w poleceniu narzędzia ImageDataGenerator.

Krok 2: Trenowanie konwolucyjnej sieci neuronowej

Definicja modelu CNN: Napisz funkcję make_convnet tworzącą model CNN.
Trenowanie modelu: Użyj metody fit na przygotowanym modelu z użyciem wygenerowanych danych treningowych i walidacyjnych.
Zapis i ocena modelu: Zapisz wyuczone wagi modelu i ocen jego dokładność.

W tej części zadania zaimplementowaliśmy model zgodnie ze wskazówkami jakie udało nam się znaleźć w różnych źródłach. Zaproponowana konfiguracja warstw, funkcji aktywacji oraz optymalizatora osiągnęła po przeprowadzonych testach i porównaniach najlepsze wyniki. Finalny model został przez nas zapisany, by być łatwo i szybko dostępnym do zaimplementowania, bez konieczności przechodzenia przez proces uczenia kolejny raz.

Krok 3: Rozszerzanie danych

Eksperymenty z augmentacją danych: Stwórz nowy ImageDataGenerator do zwiększenia różnorodności danych treningowych przez transformacje obrazu.
Ponowne trenowanie i analiza wyników: Trenuj model z nowymi danymi i analizuj dokładność oraz straty dla kolejnych epok.

Podczas procesu uczenia zauważyliśmy związek między ilością epok, a finalną skutecznością modelu – im większa ilość epok, tym lepsze wyniki osiągał model. Zwiększenie różnorodności danych treningowych w naszych obserwacjach nie wpływało znacząco na poprawę modelu. Na tym etapie zadania postanowiliśmy zaimplementować funkcję zapisującą z epoki o jakim numerze pochodzi model mający najlepsze wyniki. Zdecydowaliśmy się, by najlepszy model determinowany był przez najwyższą wartość dokładności na danych walidacyjnych.

Krok 4: Wykorzystanie wcześniej wytrenowanego modelu

Użycie modelu VGG-16: Załaduj wytrenowany model VGG-16, przeanalizuj możliwości jego zastosowania jako klasyfikatora oraz jako ekstraktora cech.
Eksperymenty z transfer learning: Zastosuj ekstrakcję cech przy użyciu VGG-16 i trenuj własny klasyfikator na tych danych.

Model VGG-16 został zaimplementowany za pomocą biblioteki keras.applications.vgg16. W dalszej części dokumentu przedstawiamy porównanie wyników tego modelu z wynikami naszego własnego modelu CNN. Analiza wyników pokazuje, że model VGG-16 osiągnął wyższą dokładność oraz charakteryzował się niższą wartością funkcji straty w porównaniu z naszym modelem. Ponadto, linie wykresu dokładności dla danych treningowych i walidacyjnych VGG-16 są blisko siebie, co świadczy o jego lepszym dopasowaniu i zdolności do generalizacji. Oznacza to, że model VGG-16 dobrze przewiduje zarówno na zbiorze treningowym, jak i walidacyjnym, bez oznak nadmiernego dopasowania (overfittingu) ani niedopasowania (underfittingu). Jednocześnie, nasz model również wykazuje dobre dopasowanie, choć nie osiąga tak wysokiej precyzji jak VGG-16. Linie wykresu dla naszego modelu są także blisko siebie, co sugeruje, że nasz model jest stabilny i zdolny do generalizacji, mimo że jego architektura i parametry mogą wymagać dalszej optymalizacji, aby dorównać modelowi VGG-16.

Krok 5: Wizualizacja wyuczonych cech

Analiza pierwszej warstwy splotowej: Zwizualizuj cechy wyuczone przez pierwszą warstwę modelu VGG-16 oraz własnego modelu CNN.

Wizualizacja cech wyuczonych przez oba modele jest podobna, jednak w przypadku VGG-16 dwukrotnie większa ilość filtrów w pierwszej warstwie skutkuje większą liczbą “klatek” na wizualizacji. Obrazy uzyskane przez nas podczas realizacji kroku przedstawiamy w dalszej części dokumentu.

Opis kodu

1. Importowanie bibliotek

Kod zaczyna się od importowania potrzebnych bibliotek, takich jak matplotlib, keras, sklearn oraz numpy.

```
import matplotlib.pyplot as plt
from keras.applications.vgg16 import VGG16
from keras.models import Model, Sequential
from keras.layers import Dense, Flatten, Dropout, Conv2D, MaxPooling2D
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, Callback
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.metrics import accuracy_score
from sklearn.manifold import TSNE
```

2. Wczytanie i przeskalowanie obrazów z katalogu treningowego

Stworzenie instancji ImageDataGenerator z różnymi augmentacjami obrazu i przygotowanie generatorów dla zbiorów treningowego i walidacyjnego.

```
datagen = ImageDataGenerator(
    rescale=1.0/255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    vertical_flip=True,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    brightness_range=[0.8, 1.2],
    channel_shift_range=0.2
)

train_generator = datagen.flow_from_directory(
    'CNN_zadanie_images/images/train',
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary',
    shuffle=True
)

validation_generator = datagen.flow_from_directory(
    'CNN_zadanie_images/images/validation',
    target_size=(150, 150),
    batch_size=32,
    class_mode='binary',
    shuffle=True
)
```

3. Definicja modelu VGG-16

Funkcja create_vgg16 definiuje model VGG-16 bez górnych warstw, dodaje własne warstwy na szczycie i kompiluje model. Definiujemy także ilość epok przez którą przprowadzane będzie późniejsze uczenie modelu.

```
xepochs = 50

def create_vgg16(input_shape):
    base_model = VGG16(weights='imagenet', include_top=False, input_shape=input_shape)
    x = base_model.output
    x = Flatten()(x)
    x = Dense(512, activation='relu')(x)
    x = Dropout(0.5)(x)
    predictions = Dense(1, activation='sigmoid')(x)
    model = Model(inputs=base_model.input, outputs=predictions)

    for layer in base_model.layers:
        layer.trainable = False

    model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])

    return model

input_shape = (150, 150, 3)
vgg16_model = create_vgg16(input_shape)
```

4. Trening modelu VGG-16

Model jest trenowany na danych treningowych przy użyciu metod fit i ModelCheckpoint do zapisu najlepszego modelu.

```
checkpoint_path_vgg16 = "best_model_vgg16.h5"
checkpoint_vgg16 = ModelCheckpoint(checkpoint_path_vgg16, monitor='val_accuracy',
save_best_only=True, mode='max')

history_vgg16 = vgg16_model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // train_generator.batch_size,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // validation_generator.batch_size,
    epochs=xepochs,
    callbacks=[checkpoint_vgg16]
)

vgg16_model.load_weights(checkpoint_path_vgg16)
```

5. Ocena modelu VGG-16

Załadowanie najlepszych wag modelu VGG-16 i ocena na zbiorze walidacyjnym.

```
loss_vgg16, accuracy_vgg16 = vgg16_model.evaluate(validation_generator)
print(f'VGG16 model loss: {loss_vgg16}')
print(f'VGG16 model accuracy: {accuracy_vgg16}')
```

6. Definicja modelu CNN

Funkcja make_convnet tworzy zaproponowany własny model konwolucyjnej sieci neuronowej.

```
def make_convnet(input_shape):
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', input_shape=input_shape))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])
    return model

model = make_convnet(input_shape)
```

7. Przygotowanie klasy zapisującej epokę, w której powstał najlepszy model

Klasa EpochSaver, która zapisuje najlepszy model na podstawie walidacji (inwencja własna, nie wymagana w zadaniu).

```
class EpochSaver(Callback):
    def __init__(self):
        super(EpochSaver, self).__init__()
        self.best_epoch = 0
        self.best_val_accuracy = -np.Inf

    def on_epoch_end(self, epoch, logs=None):
        current_val_accuracy = logs.get('val_accuracy')
        if current_val_accuracy > self.best_val_accuracy:
            self.best_val_accuracy = current_val_accuracy
            self.best_epoch = epoch + 1

    def on_train_end(self, logs=None):
        print(f'Best model saved from epoch {self.best_epoch} with val_accuracy
{self.best_val_accuracy:.4f}')
```

8. Trenowanie modelu CNN

Model jest trenowany na danych treningowych przy użyciu metod fit, ModelCheckpoint oraz EpochSaver.

```
checkpoint_path = "best_model.h5"
checkpoint = ModelCheckpoint(checkpoint_path, monitor='val_accuracy', save_best_only=True,
mode='max')

epoch_saver = EpochSaver()

history = model.fit(
    train_generator,
    steps_per_epoch=train_generator.samples // train_generator.batch_size,
    validation_data=validation_generator,
    validation_steps=validation_generator.samples // validation_generator.batch_size,
    epochs=xepochs,
    callbacks=[checkpoint, epoch_saver]
)
```

9. Ocena modelu na danych walidacyjnych

Ocena własnego modelu na danych walidacyjnych i zapis wyuczonych wag modelu.

```
val_loss, val_accuracy = model.evaluate(validation_generator, steps=validation_generator.samples
// validation_generator.batch_size)
print(f'Validation loss: {val_loss}')
print(f'Validation accuracy: {val_accuracy}')

model.load_weights(checkpoint_path)

model.save('final_model.h5')

loss, accuracy = model.evaluate(validation_generator)
print(f'Final model loss: {loss}')
print(f'Final model accuracy: {accuracy}')
```

10. Wizualizacja historii treningu

Funkcja plot_history tworzy wykresy dokładności i strat dla modelu VGG-16 i własnego modelu.

```
def plot_history(history1, history2, model1_name, model2_name):
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(history1.history['accuracy'], label=f'{model1_name} Training Accuracy')
    plt.plot(history1.history['val_accuracy'], label=f'{model1_name} Validation Accuracy')
    plt.plot(history2.history['accuracy'], label=f'{model2_name} Training Accuracy')
    plt.plot(history2.history['val_accuracy'], label=f'{model2_name} Validation Accuracy')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(history1.history['loss'], label=f'{model1_name} Training Loss')
    plt.plot(history1.history['val_loss'], label=f'{model1_name} Validation Loss')
    plt.plot(history2.history['loss'], label=f'{model2_name} Training Loss')
    plt.plot(history2.history['val_loss'], label=f'{model2_name} Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()

    plt.tight_layout()
    plt.show()

plot_history(history, history_vgg16, 'My Model', 'VGG16')
```

11. Ekstrakcja cech z obrazów przy użyciu VGG-16

Funkcja extract_features używa warstw konwolucyjnych VGG-16 do ekstrakcji cech z obrazów.

```
def extract_features(model, generator, steps):
    features = []
    labels = []
    for i in range(steps):
        x_batch, y_batch = next(generator)
        features_batch = model.predict(x_batch)
        features.append(features_batch)
        labels.append(y_batch)
    return np.vstack(features), np.hstack(labels)

base_model = VGG16(weights='imagenet', include_top=False, input_shape=input_shape)
feature_extractor = Model(inputs=base_model.input,
outputs=base_model.get_layer('block5_pool').output)

train_features, train_labels = extract_features(feature_extractor, train_generator,
train_generator.samples // train_generator.batch_size)

validation_features, validation_labels = extract_features(feature_extractor, validation_generator,
validation_generator.samples // validation_generator.batch_size)
```

12. Klasyfikator regresji logistycznej

Utworzenie klasyfikatora przy użyciu regresji logistycznej, trenowanie na wyekstrahowanych cechach i ocena na danych walidacyjnych.

```
classifier = make_pipeline(StandardScaler(), LogisticRegression(max_iter=10000))
classifier.fit(train_features.reshape(train_features.shape[0], -1), train_labels)

val_predictions = classifier.predict(validation_features.reshape(validation_features.shape[0], -
1))
accuracy = accuracy_score(validation_labels, val_predictions)
print(f'Feature extractor + Logistic Regression accuracy: {accuracy}')
```

13. Wizualizacja wyekstrahowanych cech przy użyciu t-SNE

Funkcja plot_tsne wizualizuje wyekstrahowane cechy przy użyciu t-SNE.

```
def plot_tsne(features, labels, title):
    tsne = TSNE(n_components=2, random_state=0)
    features_2d = tsne.fit_transform(features)

    plt.figure(figsize=(8, 8))
    for label in np.unique(labels):
        indices = np.where(labels == label)
        plt.scatter(features_2d[indices, 0], features_2d[indices, 1], label=f'Class {label}')

    plt.legend()
    plt.title(title)
    plt.show()

plot_tsne(train_features.reshape(train_features.shape[0], -1), train_labels, 'VGG16 Features
(Train)')

plot_tsne(validation_features.reshape(validation_features.shape[0], -1), validation_labels, 'VGG16
Features (Validation)')
```

14. Wizualizacja warstw konwolucyjnych

Funkcja visualize_conv_layer wizualizuje wyuczone cechy pierwszej warstwy splotowej dla modelu VGG-16 oraz własnego modelu.

```
def visualize_conv_layer(model, layer_name, image):
    intermediate_layer_model = Model(inputs=model.input,
                                     outputs=model.get_layer(layer_name).output)
    intermediate_output = intermediate_layer_model.predict(image)

    num_filters = intermediate_output.shape[-1]
    size = intermediate_output.shape[1]
    display_grid = np.zeros((size, size * num_filters))

    for i in range(num_filters):
        x = intermediate_output[0, :, :, i]
        x -= x.mean()
        x /= (x.std() + 1e-5)
        x *= 64
        x += 128
        x = np.clip(x, 0, 255).astype('uint8')
        display_grid[:, i * size : (i + 1) * size] = x

    scale = 20. / num_filters
    plt.figure(figsize=(scale * num_filters, scale))
    plt.title(layer_name)
    plt.grid(False)
    plt.imshow(display_grid, aspect='auto', cmap='viridis')

sample_image = next(train_generator)[0][0]
sample_image = np.expand_dims(sample_image, axis=0)

visualize_conv_layer(model, 'conv2d', sample_image)

visualize_conv_layer(vgg16_model, 'block1_conv1', sample_image)

plt.show()
```

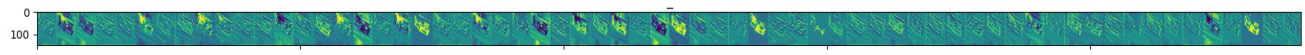
Wizualizacje

Wizualizacja cech 1 warstwy modelu własnego i VGG16

Poniżej znajdują się wizualizacje cech wyuczonych przez pierwszą warstwę modelu własnego oraz modelu VGG-16.

Wizualizacja cech pierwszej warstwy modelu własnego:

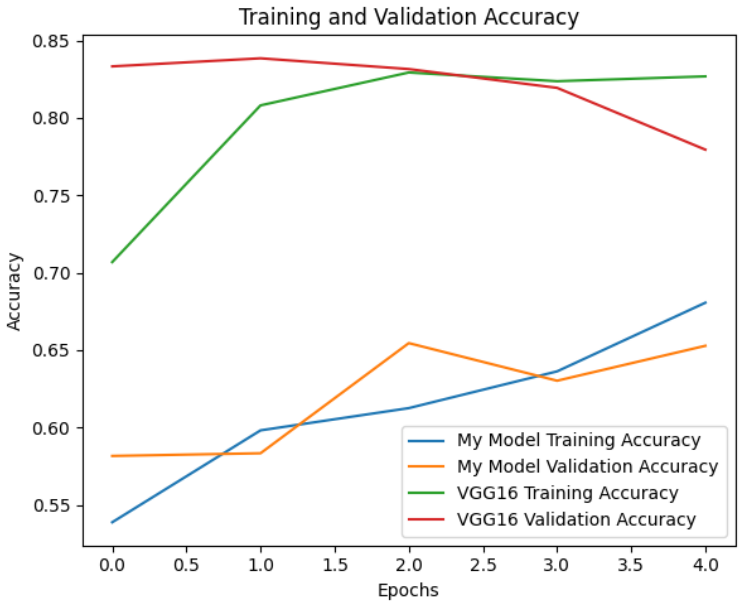
Wizualizacja cech pierwszej warstwy modelu VGG-16:

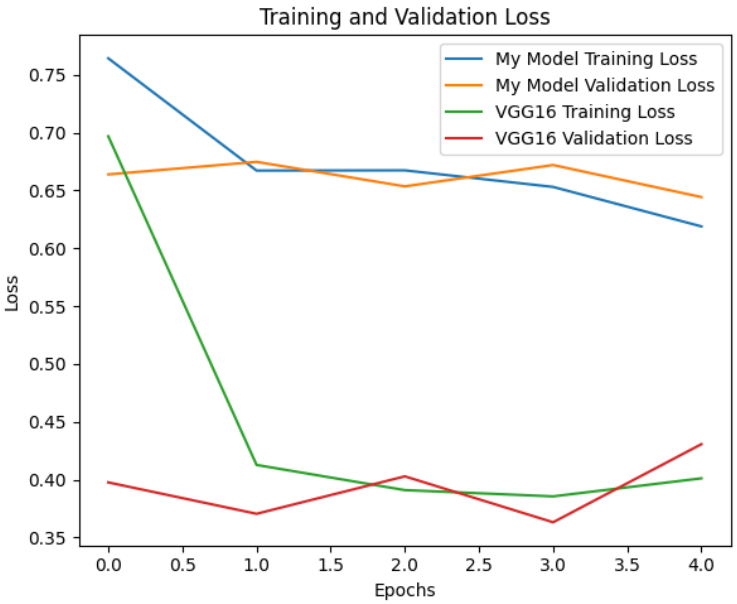


Dokładność i strata

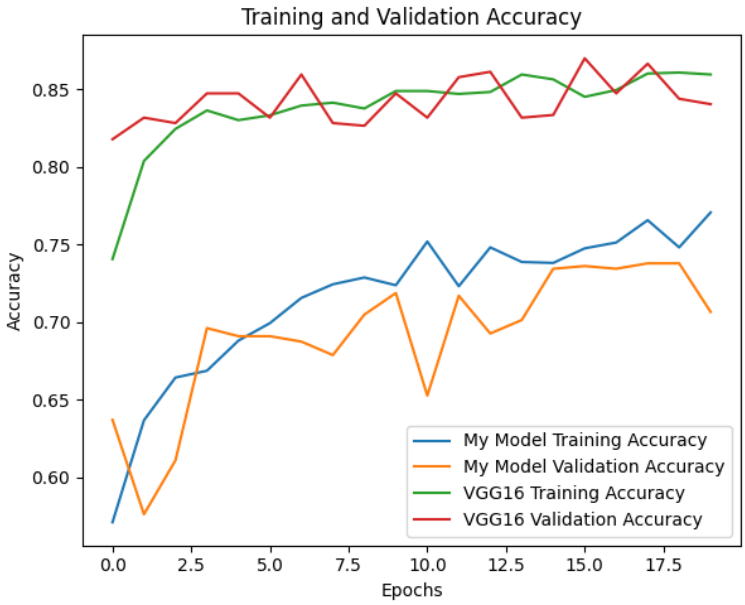
Poniżej znajdują się wykresy dokładności i straty dla autorskiego modelu CNN oraz VGG-16 po 5, 20 i 50 epokach:

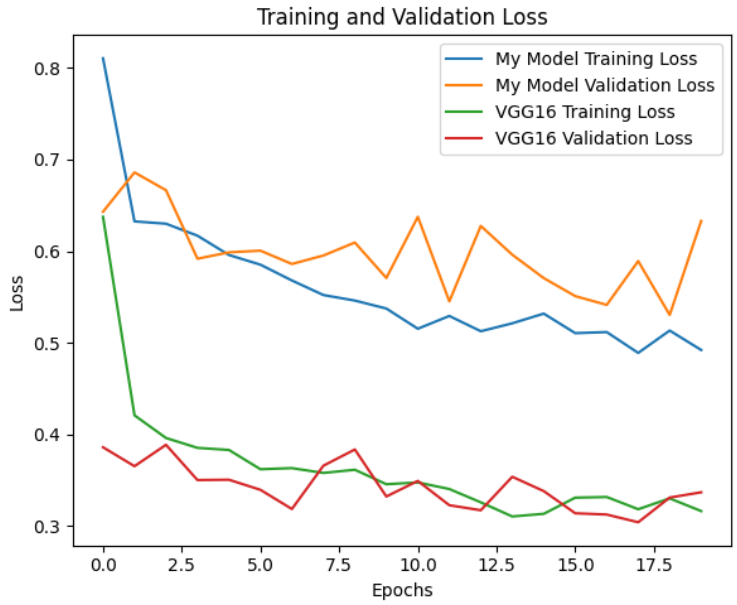
Dokładność i strata po 5 epokach



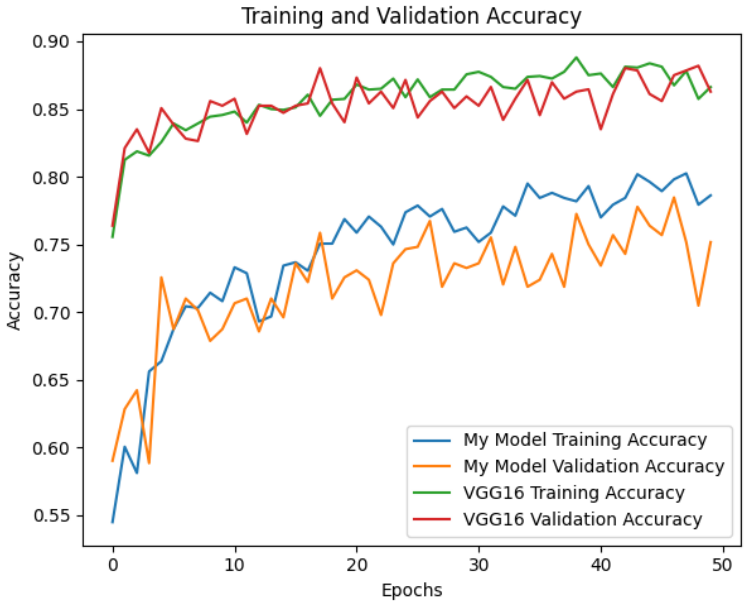


Dokładność i strata po 20 epokach



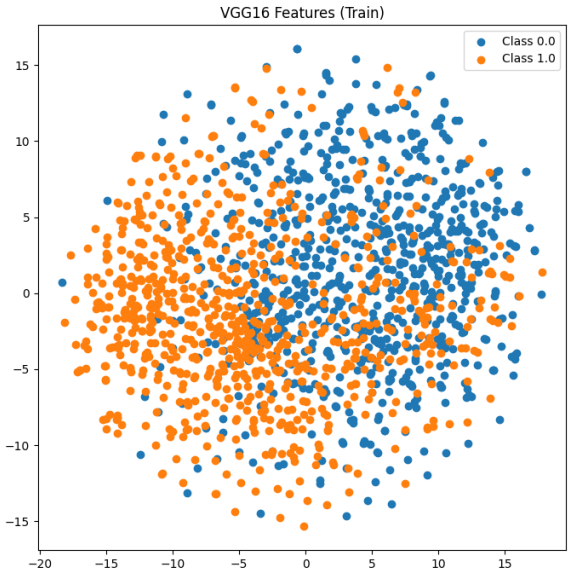


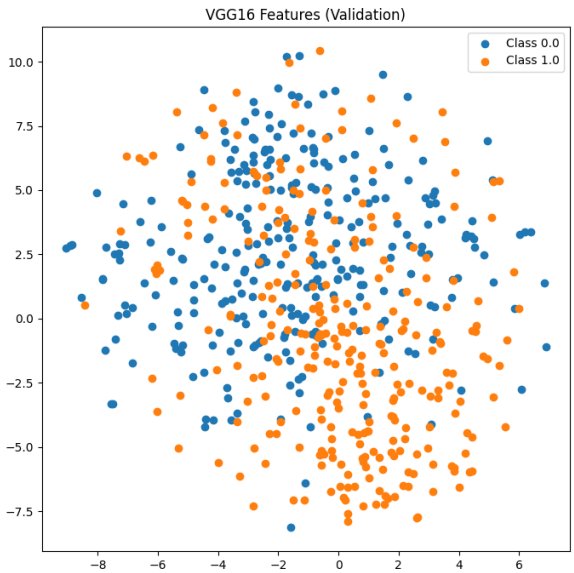
Dokładność i strata po 50 epokach



Wizualizacja ekstraktora cech

Poniżej znajdują się wykresy t-SNE dla wyekstrahowanych cech ze zbioru treningowego i walidacyjnego:





Podsumowanie

Porównanie modelu VGG-16 z przygotowanym przez nas modelem CNN wykazało, że VGG-16 osiągnął wyższą dokładność i charakteryzował się lepszym dopasowaniem, co sugeruje jego zdolność do generalizacji. Niemniej jednak, jesteśmy zadowoleni z naszej pracy, której efektem jest własny model również wykazujący zadowalającą stabilność i skuteczność, choć osiągnięte wyniki nie dorównały modelowi VGG-16. Mogliśmy także przekonać się jak ilość danych i ich różnorodność, struktura modelu wraz z jego cechami oraz ilość epok, przez które jest uczony wpływa na końcowy rezultat.

Literatura i źródła

- Sandipan Dey, Python Image Processing Cookbook, 2020 Packt Publishing
- Tarek Amr, Hands-On Machine Learning with scikit-learn and Scientific Python Toolkits, 2020 Packt Publishing
- Alberto Artasanchez, Prateek Joshi, Artificial Intelligence with Python Second Edition, 2020 Packt Publishing
- Antonio Gulli , Sujit Pal, Deep Learning with Keras, 2017 Packt Publishing
- https://www.tensorflow.org/learn?hl=pl
- https://www.tensorflow.org/api_docs/python/tf/all_symbols
- https://keras.io/api/
- https://scikit-learn.org/stable/index.html