Raport: Projekt LegoNet Aplikacja do klasyfikacji klocków LEGO i wyszukiwania zestawów

---

Data: 2 czerwca 2025
Autor: Mateusz Klemann

---

1. Wprowadzenie

Niniejszy raport dokumentuje proces wyboru optymalnego modelu sztucznej inteligencji do zadania klasyfikacji typów klocków LEGO. Głównym celem projektu **LegoNet** jest stworzenie aplikacji ułatwiającej użytkownikom identyfikację posiadanych klocków oraz wyszukiwanie zestawów możliwych do zbudowania z dostępnych elementów. Jest to pierwsza, fundamentalna część projektu, skupiająca się na precyzyjnej klasyfikacji klocków.
Porównano dwa modele:

- **Model Własny (LegoNet-CNN):** Architektura konwolucyjnej sieci neuronowej (CNN) zaprojektowana specjalnie dla tego zadania, operująca na obrazach o rozdzielczości 64x64 pikseli.
- **Model MobileNetV2:** Model oparty na architekturze MobileNetV2 z wykorzystaniem transfer learningu, operujący na obrazach o rozdzielczości 160×160 pikseli.

---

2. Analiza Procesu Treningowego Modeli

2.1. Model Własny (LegoNet-CNN)

- **Liczba Epok Treningu:** Model był trenowany przez maksymalnie 25 epok, z zastosowaniem mechanizmu EarlyStopping (cierpliwość 5 epok). Oznacza to, że trening mógł zostać zakończony wcześniej, jeśli strata walidacyjna przestała maleć, a finalny model wykorzystuje wagi z epoki o najlepszej wydajności na zbiorze walidacyjnym.
- Architektura i Konfiguracja:
  Kluczowym elementem była definicja własnej architektury CNN oraz proces jej kompilacji i treningu. Poniżej przedstawiono najważniejsze fragmenty kodu (z pliku train_custom_cnn.py):
  Definicja modelu LegoNet-CNN:
  Python
  ```
  # --- Definicja Modelu CNN ---
  input_shape_model = (IMG_SIZE[0], IMG_SIZE[1], 3) # IMG_SIZE = (64, 64)
  model = keras.Sequential([
     layers.Input(shape=input_shape_model),
     layers.Rescaling(1./255), # Normalizacja
     data_augmentation,        # Warstwa augmentacji (RandomRotation, RandomZoom, RandomContrast)
  ```

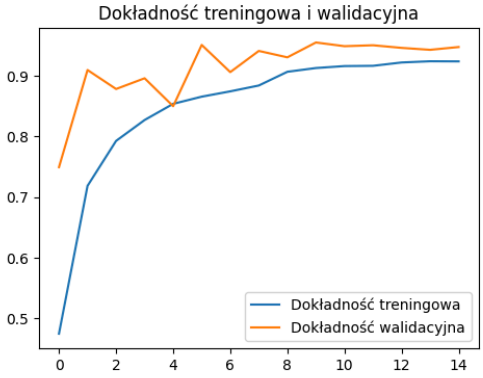
```
    layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu', padding='same'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.5), # Regularyzacja
    layers.Dense(NUM_CLASSES, activation='softmax') # Warstwa wyjściowa
(NUM_CLASSES = 9)
])
```

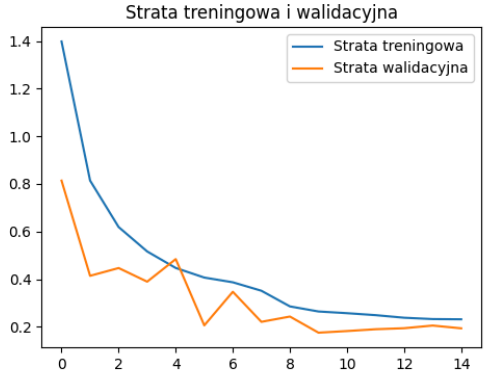
Kompilacja i trening modelu:

```
Python
# --- Kompilacja Modelu ---
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

# --- Trening Modelu ---
history = model.fit(
    train_dataset,
    epochs=EPOCHS, # Max 25
    validation_data=validation_dataset,
    callbacks=[
        keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True),
        keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=2,
min_lr=1e-6)
    ]
)
```

- **Krzywe Uczenia:** Analiza wykresów treningowych dla Modelu Własnego wykazała wysoką stabilność procesu uczenia. Zarówno krzywa dokładności, jak i straty na zbiorze walidacyjnym nie wykazywały gwałtownych fluktuacji. Obserwowano spójny wzrost dokładności walidacyjnej i spadek straty walidacyjnej, co sugeruje dobre dopasowanie modelu i skuteczną generalizację.





2.2. Model MobileNetV2

- **Liczba Epok Treningu:** Zgodnie z dokumentacją projektu, model był trenowany w dwóch głównych fazach, łącznie przez 30 epok:
  1. Początkowy trening (na bazie zamrożonego base_model): 10 epok (plik training.py).
  2. Dotrenowywanie/Fine-tuning (z odblokowaniem części warstw base_model i niższym współczynnikiem uczenia): 20 dodatkowych epok (plik continue_training.py).
- Architektura i Konfiguracja:
  Model wykorzystywał transfer learning. Poniżej kluczowe fragmenty kodu:
  Budowa modelu z MobileNetV2 jako bazą (plik training.py):
  Python

```
# --- Budowa Modelu (Transfer Learning z MobileNetV2) ---
base_model = MobileNetV2(input_shape=(IMAGE_SIZE[0], IMAGE_SIZE[1], 3), # IMAGE_SIZE = (160,160)
                         include_top=False, # Bez górnej warstwy klasyfikującej
                         weights='imagenet') # Wagi pre-trenowane na ImageNet

base_model.trainable = False # Zamrożenie wag modelu bazowego

model = Sequential([
    base_model,
    GlobalAveragePooling2D(),
    Dense(128, activation='relu'),
    Dropout(0.3), # Regularyzacja
    Dense(NUM_CLASSES, activation='softmax') # Warstwa wyjściowa (NUM_CLASSES = 9)
])
```

  Przygotowanie danych z ImageDataGenerator (plik training.py):
  Python

```
# --- Przygotowanie Danych (Generatory Danych) ---
```

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=30,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest',
    validation_split=0.2  # Wydzielenie 20% danych na zbiór walidacyjny
)
# ... konfiguracja validation_datagen i generatorów ...
```

Kompilacja i trening (plik training.py dla początkowych 10 epok):

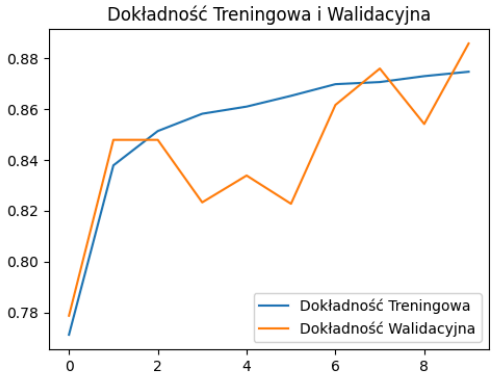
```
Python
# --- Kompilacja Modelu ---
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
# --- Trening Modelu ---
history = model.fit(
    train_generator,
    # ... steps_per_epoch, validation_data, validation_steps ...
    epochs=10 # Początkowe 10 epok
)
```

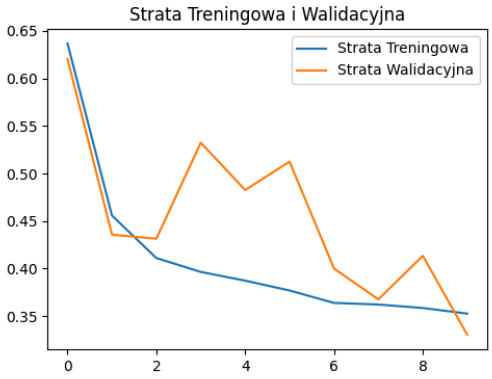
Dotrenowywanie (fragment z continue_training.py dla dodatkowych 20 epok):

```
Python
# --- Wczytanie Zapisanego Modelu ---
model = load_model('lego_classifier_mobilenetv2.h5') # Model po 10 epokach
# --- Kompilacja Modelu ---
model.compile(optimizer=Adam(learning_rate=0.0001), # Niższy learning rate
              loss='categorical_crossentropy',
              metrics=['accuracy'])
# --- Dotrenowywanie Modelu ---
history_continued = model.fit(
    train_generator,
    # ...
    epochs=10 + 20, # Całkowita liczba epok (10 initial + 20 additional)
    initial_epoch=10  # Rozpocznij numerację od epoki 10
)
```

- **Krzywe Uczenia:** Wykresy treningowe dla modelu opartego na MobileNetV2 wskazały

na znaczną niestabilność. Dokładność i strata walidacyjna wykazywały duże wahania, co jest mniej pożądane.





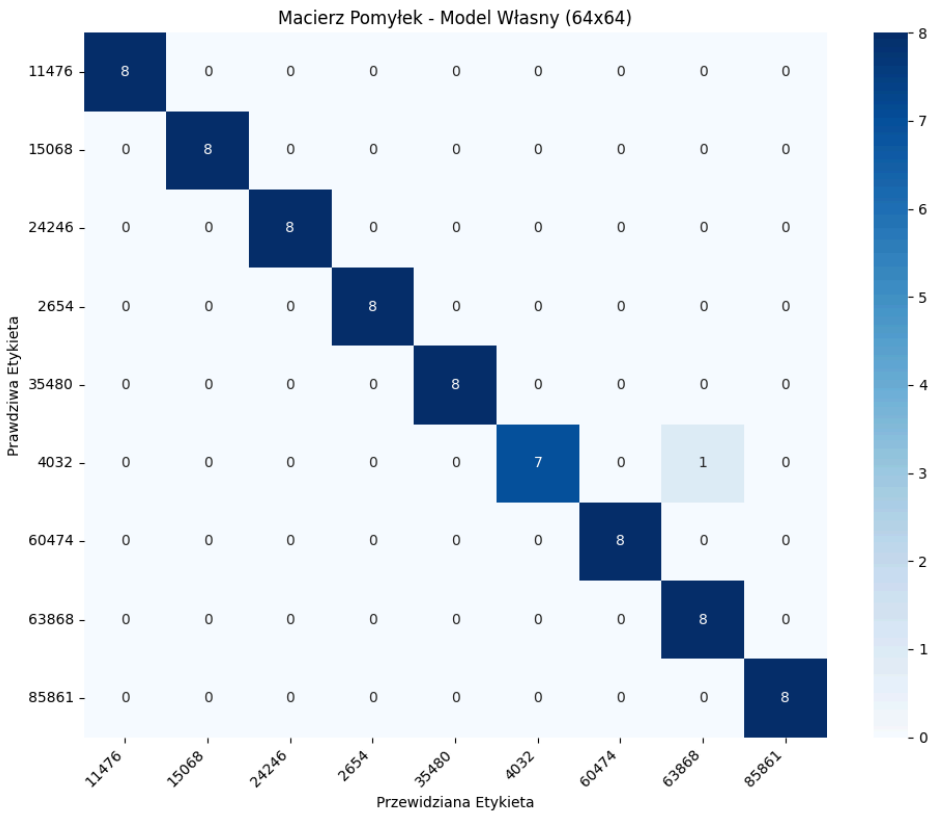
---

3. Wyniki Ewaluacji na Zbiorze Testowym

Oba modele zostały przetestowane na wspólnym zbiorze dataset_test.

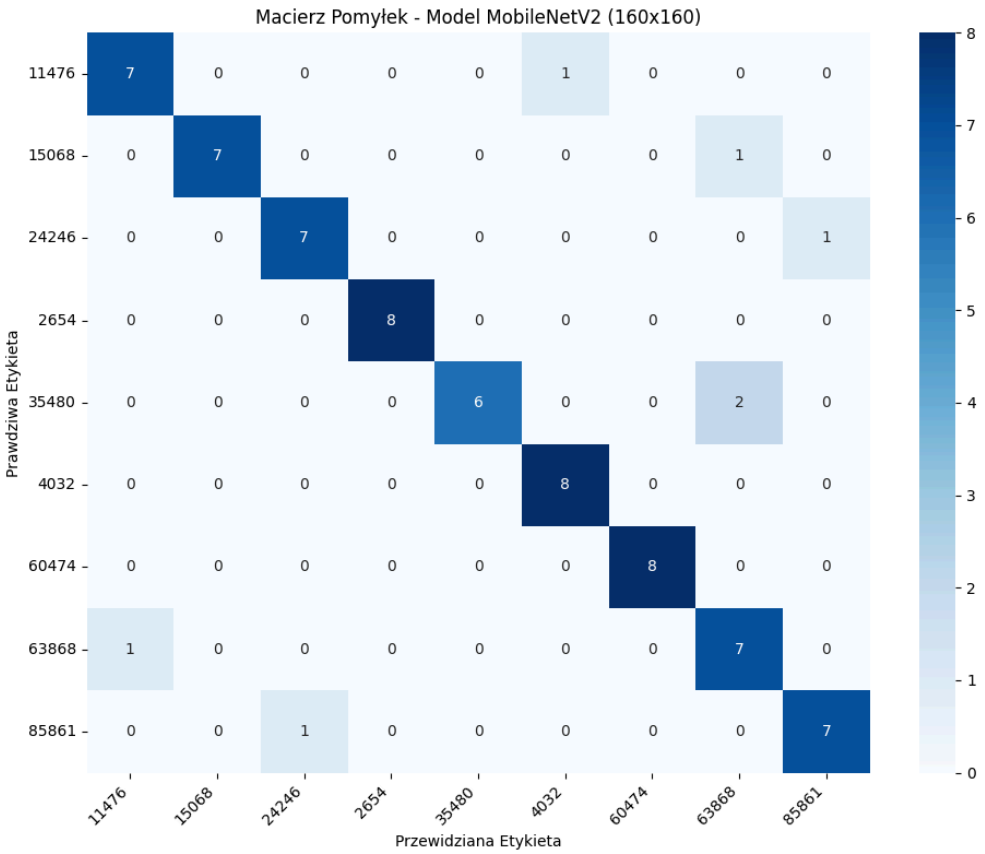
3.1. Model Własny (LegoNet-CNN)

- **Całkowita Dokładność: 98.61%**
- Raport Klasyfikacji:
Klasa Klocka	Precyzja	Czułość (Recall)	F1-Score	Support
11476	1.000	1.000	1.000	8
15068	1.000	1.000	1.000	8
24246	1.000	1.000	1.000	8
2654	1.000	1.000	1.000	8
35480	1.000	1.000	1.000	8
4032	1.000	0.875	0.933	8
60474	1.000	1.000	1.000	8
63868	0.889	1.000	0.941	8
85861	1.000	1.000	1.000	8
macro avg	0.988	0.986	0.986	72
weighted avg	0.988	0.986	0.986	72
- Macierz Pomyłek: Model popełnił tylko jeden błąd: jeden klocek 4032 został błędnie zaklasyfikowany jako 63868.



3.2. Model MobileNetV2 (160×160)

- **Całkowita Dokładność: 90.28%**
- Raport Klasyfikacji:
Klasa Klocka	Precyzja	Czułość (Recall)	F1-Score	Support
11476	0.875	0.875	0.875	8
15068	1.000	0.875	0.933	8
24246	0.875	0.875	0.875	8
2654	1.000	1.000	1.000	8
35480	1.000	0.750	0.857	8
4032	0.889	1.000	0.941	8
60474	1.000	1.000	1.000	8
63868	0.700	0.875	0.778	8
85861	0.875	0.875	0.875	8
macro avg	0.913	0.903	0.904	72
weighted avg	0.913	0.903	0.904	72
- Macierz Pomyłek: Model MobileNetV2 popełnił łącznie 7 błędów klasyfikacyjnych, rozłożonych na różne klasy.



---

4. Dyskusja i Argumentacja Wyboru

Porównując oba modele, kluczowe różnice i obserwacje są następujące:

- **Dokładność na Zbiorze Testowym:** Model Własny (LegoNet-CNN) (98.61%) wykazał się znacznie wyższą dokładnością niż Model MobileNetV2 (90.28%).
- **Stabilność Treningu:** Model Własny charakteryzował się stabilnym procesem uczenia. Model MobileNetV2, szczególnie w początkowej fazie, wykazywał znaczne fluktuacje na krzywych walidacyjnych, co podważa jego niezawodność.
- **Szczegółowe Metryki Klasyfikacji:** Model Własny osiągnął lepsze uśrednione wartości precyzji, czułości i F1-score (macro avg F1-score 0.986 vs 0.904 dla MobileNetV2).
- **Liczba Błędów:** Model Własny popełnił tylko jeden błąd na zbiorze testowym, podczas gdy Model MobileNetV2 popełnił siedem błędów.
- **Złożoność i Zasoby:** Model Własny, operujący na mniejszych obrazach (64x64) i posiadający prostszą architekturę, jest efektywniejszy pod względem zasobów niż Model MobileNetV2 (obrazy 160×160). Dla aplikacji docelowej LegoNet, która ma być przyjazna użytkownikowi i potencjalnie działać na różnych urządzeniach, efektywność jest istotnym czynnikiem.

---

5. Prototyp Aplikacji Webowej LegoNet

W celu praktycznej demonstracji możliwości wytrenowanego modelu **LegoNet-CNN** oraz zarysowania funkcjonalności aplikacji LegoNet, stworzono prototyp aplikacji webowej. Aplikacja została zaimplementowana w języku Python z wykorzystaniem biblioteki **Streamlit**, co pozwoliło na szybkie stworzenie interaktywnego interfejsu użytkownika.

5.1. Architektura i Technologie

- **Backend i Frontend:** Streamlit
- **Model AI:** Wytrenowany model lego_brick_classifier_model.keras (LegoNet-CNN)
- **Główne biblioteki Python:** streamlit, tensorflow, Pillow (do obsługi obrazów), numpy.

5.2. Kluczowe Funkcjonalności Prototypu

Aplikacja została podzielona na dwa główne moduły dostępne z panelu bocznego:

5.2.1. Moduł: Klasyfikator Klocków LEGO

Ten moduł pozwala użytkownikowi na interakcję z sercem systemu – modelem klasyfikującym klocki.

- **Przesyłanie Zdjęcia:** Użytkownik może załadować zdjęcie klocka LEGO poprzez dedykowany interfejs. Kluczowy fragment kodu odpowiedzialny za tę funkcjonalność (plik app.py):
  Python
  ```
  # Fragment kodu Streamlit do przesyłania pliku
  uploaded_file = st.file_uploader("Wybierz zdjęcie klocka...", type=["jpg", "jpeg", "png"])
  if uploaded_file is not None:
      image_pil = Image.open(uploaded_file)
      st.image(image_pil, caption="Przesłane zdjęcie", use_column_width=True)
  ```

- **Klasyfikacja:** Po przesłaniu zdjęcia i kliknięciu przycisku "Sklasyfikuj Klocek", obraz jest przetwarzany i przekazywany do modelu LegoNet-CNN. Fragment logiki predykcji:
  Python
  ```
  # Fragment logiki predykcji w Streamlit (plik app.py)
  # model to załadowany lego_brick_classifier_model.keras
  # IMG_SIZE = (64,64), CLASS_NAMES = [...]
  if st.button("🔎 Sklasyfikuj Klocek"):
      if model is not None:
          predicted_label, confidence = predict_lego_brick(model, image_pil, IMG_SIZE,
  CLASS_NAMES)
          st.success(f"**Przewidziany kod klocka:** {predicted_label}")
          st.info(f"**Pewność predykcji:** {confidence*100:.2f}%")
  ```

- **Wyświetlanie Wyniku:** Aplikacja prezentuje przewidziany kod (ID) klocka oraz pewność tej predykcji.
- **Wirtualna Kolekcja (Prototyp):** Każdy sklasyfikowany klocek jest tymczasowo dodawany do "wirtualnej kolekcji" użytkownika.

5.2.2. Moduł: Definiowanie Zestawu LEGO (Prototyp)

Ten moduł służy do prototypowego wprowadzania informacji o zestawach LEGO.

- **Wprowadzanie Danych Zestawu:** Użytkownik może wprowadzić nazwę zestawu oraz listę potrzebnych klocków (ID i ilość). Fragment formularza (plik app.py):
  Python
  ```
  # Fragment formularza Streamlit do definiowania zestawu
  with st.form("new_lego_set_form", clear_on_submit=True):
      set_name_input = st.text_input("Nazwa zestawu LEGO:")
      # ... pola do wprowadzania ID klocka i ilości ...
      add_brick_button = st.form_submit_button("➕ Dodaj Klocek do listy")
      submitted_set_form = st.form_submit_button("💾 Zdefiniuj Ten Zestaw LEGO")
  ```

- **Zarządzanie Listą Klocków i Zapis Prototypowy:** Dodane klocki są wyświetlane; zdefiniowane zestawy są przechowywane w pamięci sesji aplikacji.
- **Przeglądanie Zdefiniowanych Zestawów:** Aplikacja wyświetla listę zdefiniowanych zestawów.

5.3. Interfejs Użytkownika (UI)
Interfejs, stworzony przy pomocy Streamlit, jest prosty i intuicyjny, z nawigacją w panelu bocznym.
5.4. Uruchomienie Aplikacji
Aplikacja jest uruchamiana lokalnie poleceniem streamlit run app.py.

---

6. Wnioski i Dalsze Kierunki Rozwoju Projektu LegoNet

Stworzony prototyp aplikacji webowej z powodzeniem integruje wytrenowany model **LegoNet-CNN** do klasyfikacji klocków LEGO oraz wprowadza podstawowe mechanizmy do definiowania zestawów. Model klasyfikacyjny, osiągając dokładność **98.61%** na zbiorze testowym oraz wykazując stabilność treningu, stanowi solidny fundament dla aplikacji LegoNet.

---

6. Źródła

- **Zbiór danych** - link
- **Projekt o tej samej tematyce** - link
- **Źródło informacji o lego** - link