Raport Projektu: Klasyfikator Stanu Opon

Automatyczna klasyfikacja zużycia opon z wykorzystaniem sieci neuronowych

Nad projektem pracowali:

Viktoriia Tsiupiak
Maciej Kaźmierczak
Krzysztof Teodorowicz
Dawid Malinowski
Kondrat Janczak

1. Wstęp

Projekt **Tire Classifier** to kompleksowe rozwiązanie do automatycznej klasyfikacji stanu zużycia opon samochodowych przy użyciu technik uczenia głębokiego. System wykorzystuje sieć neuronową **ResNet-18** do rozróżnienia między oponami nowymi a zużytymi na podstawie analizy obrazów.

Cel projektu: opracowanie narzędzia wspomagającego ocenę stanu technicznego opon, które może znaleźć zastosowanie w:

- Stacjach kontroli pojazdów
- Serwisach samochodowych
- Aplikacjach mobilnych dla kierowców
- Systemach automatycznej inspekcji

2. Architektura projektu

2.1 Struktura projektu:

```
ML_Tire_Wear/
├── src/
│   ├── data_loader.py      # Ładowanie i preprocessing danych
│   ├── model.py            # Definicja architektury modelu
│   ├── train.py            # Skrypt treningowy
│   ├── evaluate.py         # Ewaluacja modelu
│   ├── visualize.py        # Wizualizacja wyników
│   ├── gui.py              # Interfejs graficzny (Tkinter)
│   └── streamlit_app.py    # Aplikacja webowa (Streamlit)
├── notebooks/
│   ├── eda.ipynb           # Eksploracyjna analiza danych
│   └── test_loader.ipynb   # Testowanie ładowania danych
└── data/                   # Zbiór danych (train/val/test)
```

2.2 Technologie i Biblioteki

W projekcie wykorzystano różne biblioteki i frameworki, które wspierają proces uczenia maszynowego, przetwarzania danych oraz tworzenia interfejsów użytkownika.

Do głębokiego uczenia maszynowego zastosowano framework **PyTorch**, natomiast do wstępnego przetwarzania obrazów oraz korzystania z gotowych modeli pretrenowanych użyto biblioteki **torchvision**.

W zakresie przetwarzania danych wykorzystano bibliotekę **Pillow (PIL**) do operacji na obrazach, **NumPy** do obliczeń numerycznych, a **scikit-learn** do oceny modeli za pomocą różnych metryk.

Interfejs użytkownika zrealizowano za pomocą dwóch podejść: **Tkinter** posłużył do stworzenia aplikacji desktopowej, natomiast **Streamlit** umożliwił przygotowanie wersji webowej. Do wizualizacji wyników zastosowano biblioteki **Matplotlib** i **Seaborn.**

3. Model i Metodologia

3.1 Architektura Modelu ResNet-18

W projekcie zastosowano architekturę ResNet-18, która charakteryzuje się kilkoma istotnymi cechami. Model ten został wstępnie wytrenowany na zbiorze danych ImageNet, co pozwala na skuteczne wykorzystanie wcześniej nabytej wiedzy dzięki podejściu Transfer Learning.

Aby dostosować model do konkretnego zadania klasyfikacji obrazów na dwie klasy – nowe i zużyte – zmodyfikowano jego ostatnią warstwę, zastępując ją warstwą wyjściową odpowiednią dla dwóch kategorii.

Model przyjmuje na wejściu obrazy o rozdzielczości 224×224 piksele i zawiera około 11 milionów parametrów, co czyni go stosunkowo lekką, a zarazem skuteczną architekturą głębokiego uczenia.

```
# Kluczowa modyfikacja modelu
model = models.resnet18(pretrained=True)
in_features = model.fc.in_features
model.fc = nn.Linear(in_features, num_classes=2)
```

3.2 Wstępne przetwarzanie danych

W procesie przygotowania danych treningowych zastosowano szereg transformacji mających na celu augmentację danych. Wśród użytych technik znalazły się: **RandomHorizontalFlip(),** czyli losowe odbicie poziome obrazu, oraz **RandomRotation(10°),** pozwalająca na losową rotację o maksymalnie 10 stopni.

Dodatkowo obrazy zostały przeskalowane do stałego rozmiaru 224×224 piksele za pomocą transformacji **Resize(224, 224).**

W celu ujednolicenia wartości pikseli dokonano również normalizacji kolorów, przyjmując średnie wartości (**mean**) równe [0.5, 0.5, 0.5] oraz odchylenia standardowe (**std**) także [0.5, 0.5, 0.5].

Zastosowanie augmentacji danych miało na celu zwiększenie różnorodności zbioru treningowego, poprawę zdolności generalizacji modelu oraz redukcję zjawiska przeuczenia (**overfittingu**).

3.3 Trening

W procesie treningu modelu zastosowano następujące parametry i ustawienia. Do optymalizacji wag wykorzystano optymalizator **Adam** z ustalonym learning rate równym **0.001**. Jako funkcję straty przyjęto **CrossEntropyLoss**, odpowiednią dla zadania klasyfikacji wieloklasowej. Trening odbywał się w partiach o rozmiarze **batch size = 32**, a liczba epok wynosiła domyślnie **15**, z możliwością modyfikacji.

Urządzenie obliczeniowe było wybierane automatycznie – model trenował się na GPU (CUDA), jeśli było dostępne, w przeciwnym razie wykorzystywał CPU.

Przyjęto również strategię zapisywania modelu, w której stan modelu był zapisywany wyłącznie w przypadku poprawy dokładności (accuracy) na zbiorze walidacyjnym. W takich sytuacjach tworzony był checkpoint o nazwie model_best.pt.

4. Funkcjonalności projektu

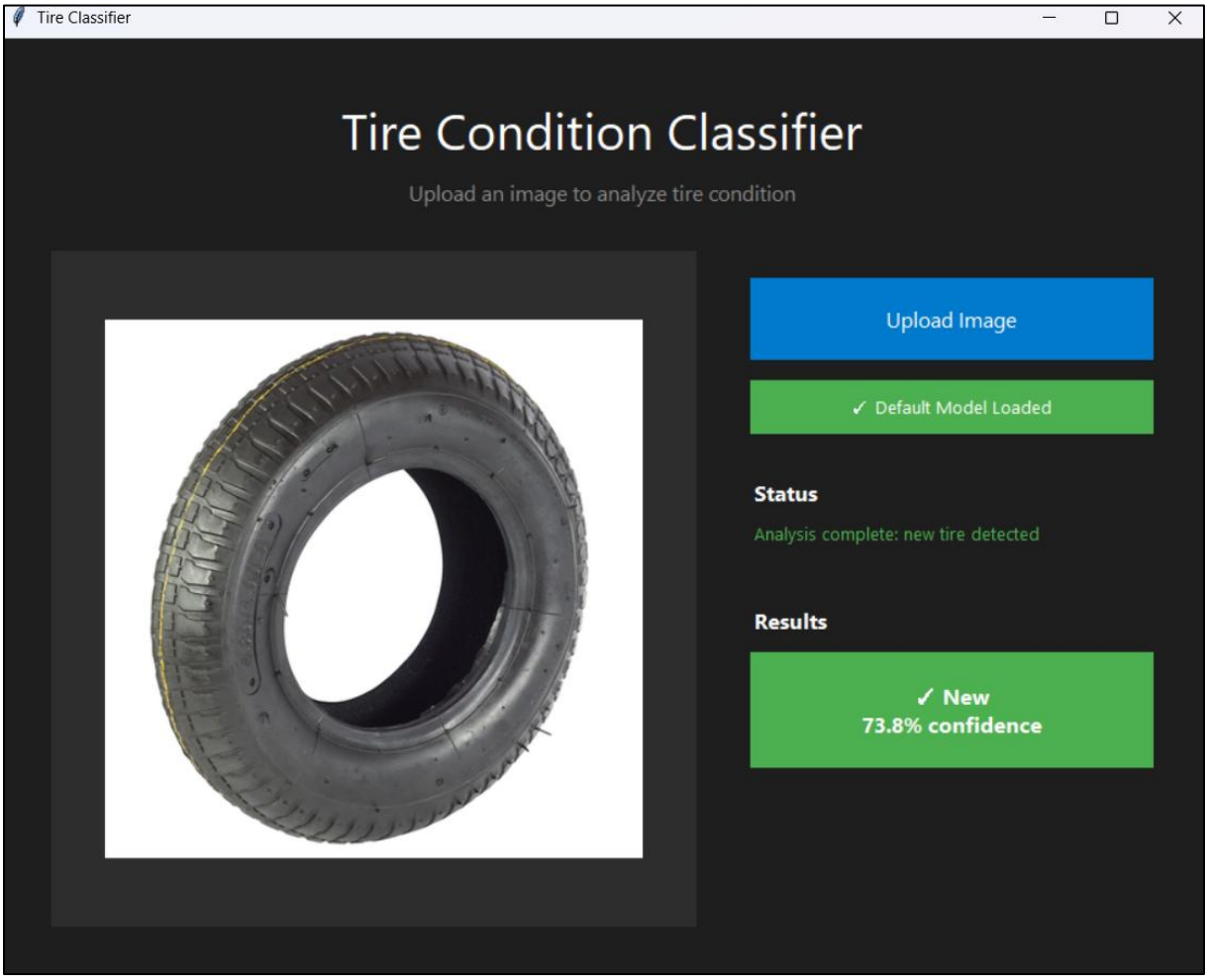
4.1 Interfejs Desktop (Tkinter)

Aplikacja desktopowa została zbudowana w Tkinterze jako prosty interfejs GUI. Posiada ciemny motyw z kolorowymi akcentami, automatyczne skalowanie do rozmiaru okna oraz obsługę Drag & Drop do wygodnego ładowania obrazów.

Wyświetla wynik klasyfikacji z poziomem pewności (%) oraz informacje zwrotne w czasie rzeczywistym (statusy, pasek postępu).

Workflow użytkownika:

1. Uruchomienie aplikacji (ładowany domyślny model),
2. Wybór lub przeciągnięcie zdjęcia opony,
3. Automatyczna analiza i wyświetlenie wyniku,
4. Opcjonalne wczytanie własnego modelu.



4.2 Narzędzia ewaluacji

Plik **visualize.py** oferuje funkcje do oceny i prezentacji wyników modelu. Umożliwia wygenerowanie macierzy pomyłek (**Confusion Matrix**) z wizualizacją, siatki przykładów predykcji (**Sample Predictions**) z oznaczeniem poprawnych i błędnych klasyfikacji oraz obliczenie podstawowych metryk skuteczności, takich jak **dokładność**, **precyzja** i **recall**.

Przykład użycia:

```python
plot_confusion(model, test_loader, ["New", "Worn"], device)
show_predictions(model, test_loader, ["New", "Worn"], device, n=50)
```

4.3 Modularność projektu

System został zaprojektowany w sposób modularny, co ułatwia jego rozwijanie i utrzymanie.

Plik **data_loader.py** odpowiada za elastyczne ładowanie zbiorów treningowego, walidacyjnego i testowego, z automatyczną detekcją dostępnych folderów. Umożliwia również konfigurację transformacji danych i rozmiaru batcha.

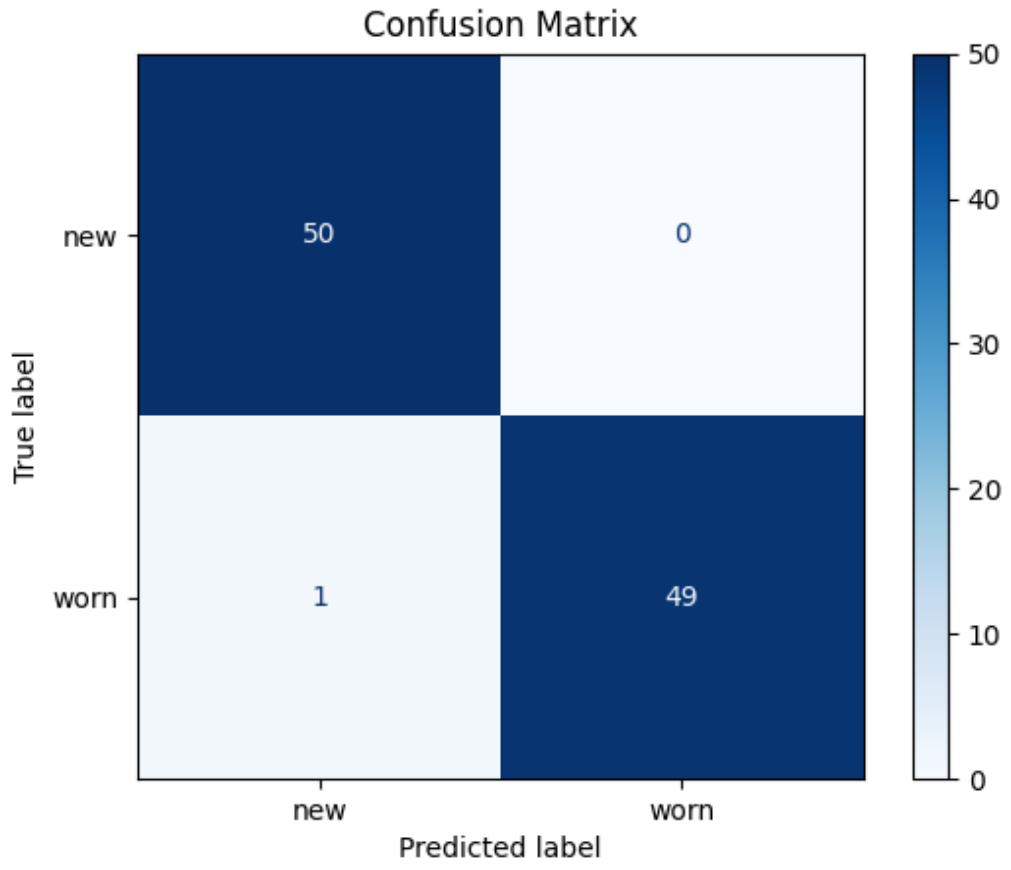
Z kolei **model.py** zawiera logikę związaną z modelem – zapewnia łatwą zmianę architektury, obsługę różnych liczby klas oraz hermetyzację konstrukcji i działania modelu, co upraszcza jego integrację z resztą systemu.

5. Opis wyników treningu

5.1 Skuteczność modelu

Model oparty na architekturze ResNet18 osiągnął bardzo dobre wyniki w zadaniu klasyfikacji stanu opon. Najlepsza dokładność walidacyjna wyniosła 98% i została osiągnięta już w czwartej epoce, natomiast końcowa dokładność treningowa osiągnęła 93%, a walidacyjna 94% po piętnastej epoce.

Analiza macierzy pomyłek potwierdza wysoką jakość klasyfikacji – wszystkie próbki klasy "new" zostały sklasyfikowane poprawnie (100% recall), a spośród próbek klasy "worn", 49 na 50 zostały zaklasyfikowane prawidłowo (98% recall). Ostateczna dokładność klasyfikatora wyniosła 99%, a jedyny błąd polegał na błędnym zakwalifikowaniu jednej zużytej opony jako nowej. Wyniki te wskazują na bardzo wysoką skuteczność modelu w rozróżnianiu opon nowych i zużytych.



5.2 Proces uczenia

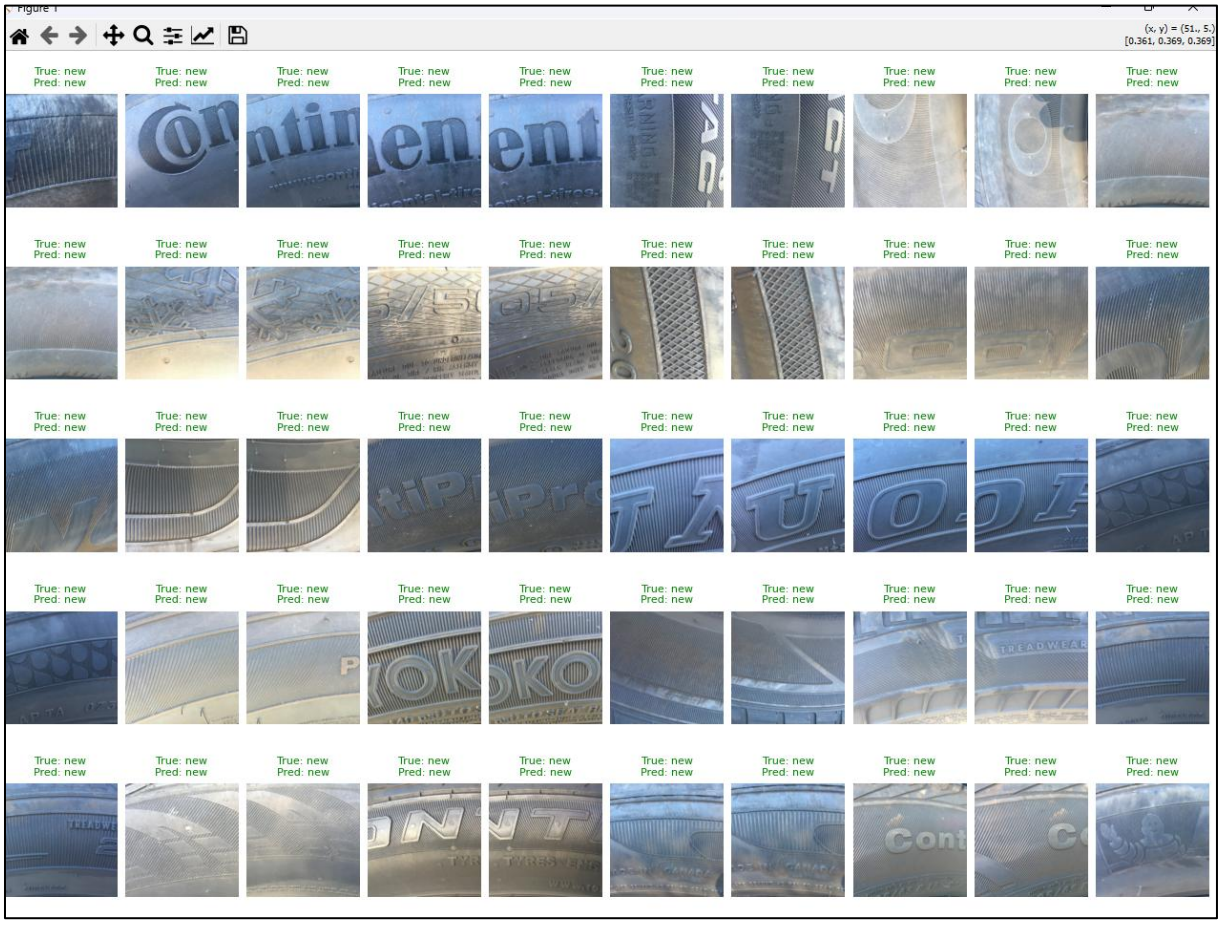
Model wykazywał szybką konwergencję, osiągając 80% dokładności już po pierwszej epoce i aż 98% po czwartej. Użycie wstępnie wytrenowanych wag w modelu ResNet18 znacząco przyspieszyło proces uczenia.

Trening przebiegał stabilnie, o czym świadczy systematyczny spadek wartości funkcji straty z 12.26 do 4.41 oraz wzrost dokładności treningowej z 80% do 98% w kolejnych epokach.

5.3 Identyfikacja problemów

W trakcie analizy pojawiły się oznaki **overfittingu**. Już w piątej epoce dokładność treningowa osiągnęła 94%, natomiast walidacyjna spadła do 72%, co wskazuje na istotną różnicę i przeuczenie modelu. W dalszych epokach odnotowano wahania dokładności walidacyjnej w zakresie 92–98%, mimo ciągłego wzrostu dokładności treningowej.

Dodatkowo, analiza przykładowych predykcji wykazała, że model jest nadwrażliwy na warunki oświetleniowe, w szczególności na jasność i kontrast zdjęć. Przykładowo, nowe opony widoczne na ciemnych zdjęciach były czasem klasyfikowane jako zużyte. Sugeruje to, że model mógł nauczyć się błędnej korelacji między jasnością obrazu a stanem opony, co może wpływać negatywnie na skuteczność w rzeczywistych warunkach.



5.5 Rekomendacje na przyszłość

Aby zredukować overfitting, warto wdrożyć mechanizmy takie jak early stopping, zastosować regularyzację (np. Dropout lub Weight Decay), wykorzystać walidację krzyżową (k-fold) oraz zwiększyć rozmiar zbioru danych, szczególnie walidacyjnego.

Dalszą poprawę może przynieść optymalizacja hiperparametrów – eksperymenty z różnymi wartościami learning rate, testowanie innych architektur (np. ResNet50, EfficientNet) oraz dopasowanie parametrów augmentacji.

Ewaluację warto rozszerzyć o dodatkowe metryki jakości (precision, recall, F1-score dla każdej klasy), szczegółową analizę błędnych klasyfikacji oraz testy na danych rzeczywistych, które nie były używane w procesie trenowania.

6. Zastosowania praktyczne

- **Branża Motoryzacyjna**

System znajduje zastosowanie w stacjach kontroli pojazdów, gdzie może przyspieszyć proces inspekcji, uobiektywnić ocenę stanu opon oraz umożliwić cyfrową dokumentację wyników.

W serwisach samochodowych narzędzie może służyć jako wsparcie w doradztwie dla klientów, uzasadnienie potrzeby wymiany opon oraz narzędzie kontroli jakości świadczonych usług.

- **Aplikacje Consumer**

W kontekście użytkowników indywidualnych, system może być zaimplementowany w aplikacjach mobilnych, umożliwiając samoobsługową kontrolę stanu opon (self-check), przypomnienia o konieczności ich sprawdzenia oraz integrację z systemami telematycznymi pojazdu.

W sektorze e-commerce rozwiązanie może służyć do oceny stanu technicznego używanych opon, automatycznej wyceny, a także wsparcia klientów w podejmowaniu decyzji zakupowych.