Rozpoznawanie wybranych znaków drogowych z wykorzystaniem sieci konwolucyjnej

Mateusz Podmokły, Wojciech Fortuna

3 lipca 2025

1. Wstęp

Celem projektu było stworzenie kompletnego systemu wbudowanego, który umożliwia rozpoznawanie znaków drogowych przy użyciu mikrokontrolera ESP32-CAM wraz z wbudowaną kamerą. System wykorzystuje model uczenia maszynowego oparty na TensorFlow Lite Micro, umożliwiając detekcję i klasyfikację wybranych znaków drogowych w czasie rzeczywistym, bez potrzeby komunikacji z chmurą czy zewnętrznym serwerem.

Projekt obejmuje:

- przygotowanie i wstępne przetworzenie zbioru danych (oryginalnych i syntetycznych),
- trenowanie modelu klasyfikującego znaki drogowe,
- konwersję modelu do formatu kompatybilnego z ESP32 (TFLite Micro),
- implementację kodu w języku C++ oraz integrację z ESP-IDF,
- implementację serwera HTTP do przesyłania wyników klasyfikacji,
- testy funkcjonalne z użyciem rzeczywistej kamery oraz emulowanych danych wejściowych.

System został zoptymalizowany pod kątem ograniczonych zasobów mikrokontrolera, z uwzględnieniem rozmiaru modelu, szybkości działania i wykorzystania pamięci.

2. Specyfikacja sprzętowa

W projekcie wykorzystano moduł **ESP32-CAM**, oparty na układzie ESP32-WROVER-E, wyposażony w kamerę cyfrową OV2640. ESP32-CAM to niewielka, lecz bardzo funkcjonalna płytka mikrokontrolera, która dzięki wbudowanemu modułowi Wi-Fi i obsłudze kamer znajduje szerokie zastosowanie w projektach IoT oraz systemach wizyjnych.

- **Napięcie zasilania:** 5 V;
- **Poziomy logiczne:** 3,3 V;

- **Mikrokontroler:** ESP32-WROVER-E z taktowaniem do 240 MHz;
- **Pamięć:** 8 MB RAM, 4 MB FLASH;
- **Interfejsy:** przyciski RESET i BOOT, wlutowane złącze goldpin (rastr 2,54 mm);
- **Łączność:** zintegrowany moduł Wi-Fi 802.11 b/g/n;
- **Tryby pracy sieciowej:** AP, STA oraz AP+STA;
- **Zabezpieczenia sieciowe:** TKIP, WEP, CRC, CCMP, WPA/WPA2, WPS;
- **Kamera:** OV2640 o rozdzielczości do 2 MPx;
- **Konwerter USB-UART:** CH340;
- **Złącze komunikacyjne:** micro USB;
- **Wymiary płytki:** 50,7 mm × 28 mm × 4,5 mm.

3. Model i dane

W ramach projektu zaprojektowano i przeszkolono konwolucyjną sieć neuronową (CNN) przeznaczoną do klasyfikacji znaków drogowych. Do treningu wykorzystano dane z rzeczywistego zbioru GTSRB, a także dane syntetyczne generowane na podstawie dostępnych przykładów. Model został docelowo zaimplementowany na mikrokontrolerze ESP32-CAM.

3.1 Architektura modelu

Model zbudowano w oparciu o warstwy konwolucyjne i gęsto połączone. Jego uproszczona architektura przedstawia się następująco:

- Wejście: obraz RGB o rozmiarze 64×64 piksele,
- 2 bloki konwolucyjne (Conv2D → ReLU → MaxPool),
- Warstwa spłaszczająca (Flatten),
- 2 warstwy gęste (Fully Connected), ostatnia z softmaxem,
- Wyjście: wektor prawdopodobieństw dla 6 klas.

Sieć uczono przy użyciu funkcji kosztu `CrossEntropyLoss` oraz optymalizatora `Adam`. Trenowanie prowadzono na danych zbalansowanych i zaugmentowanych.

3.2 Zbiór danych

W projekcie wykorzystano niemiecki zbiór znaków drogowych GTSRB (German Traffic Sign Recognition Benchmark). Oryginalny zbiór zawiera ponad 50 klas i tysiące zdjęć wykonanych w warunkach drogowych. Na potrzeby projektu wybrano 6 klas:

- 2 – ograniczenie prędkości do 50 km/h,
- 13 – ustąp pierwszeństwa,
- 14 – znak STOP,
- 15 – zakaz wjazdu pojazdów,
- 17 – zakaz wjazdu,
- 27 – przejście dla pieszych.

Z danych rzeczywistych pozyskano obrazy należące do powyższych klas, a każda klasa została uzupełniona o dodatkowe dane syntetyczne.
Poniżej przykładowe zdjęcia ze zbioru GTSRB użyte do treningu:

Rysunek 1: Przykładowe obrazy rzeczywiste z GTSRB

3.3 Generowanie danych syntetycznych

Aby zwiększyć różnorodność i liczebność zbioru treningowego, wygenerowano dane syntetyczne na podstawie dostępnych znaków drogowych. Każdy znak został osadzony na sztucznie utworzonym tle i poddany losowym przekształceniom.

Do generowania użyto m.in. następujących technik:

- wybór tła spośród: jednolitego koloru, szumu lub gradientu,
- losowe pozycjonowanie i skalowanie znaku,
- obrót znaku do ±20 stopni,
- przekształcenia z biblioteki `Albumentations`, takie jak:
  - rozmycie Gaussa oraz symulacja ruchu,
  - dodanie szumu ISO,
  - zmiany jasności i kontrastu,
  - przekształcenia perspektywiczne.

Każdy wygenerowany obraz miał wymiary 64×64 piksele i stanowił realistyczne, choć sztucznie stworzone, odwzorowanie sceny drogowej z pojedynczym znakiem.

Przykładowy fragment kodu augmentującego dane:

```
augment = A.Compose([
    A.GaussianBlur(p=0.2),
    A.MotionBlur(p=0.2),
    A.ISONoise(p=0.3),
    A.RandomBrightnessContrast(p=0.3),
    A.Perspective(p=0.3),
    A.Rotate(limit=20),
    A.Resize(40, 40)
])
```

3.4. Przygotowanie danych

Obrazy zostały przeskalowane do rozmiaru 64×64 pikseli oraz poddane dodatkowej augmentacji w czasie uczenia (m.in. `RandomAffine`, `ColorJitter`). Dane rzeczywiste i syntetyczne zostały połączone, a następnie podzielone w proporcji 80% do 20% na zbiór treningowy i walidacyjny.

3.5. Architektura modelu

Zaimplementowano własną konwolucyjną sieć neuronową, składającą się z 3 bloków:

- `Conv2d` + `ReLU` + `MaxPool2d`,
- końcowa warstwa `AdaptiveAvgPool2d(1)`,
- warstwa w pełni połączona (`Linear`).

Fragment implementacji modelu:

```
class MediumCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),
            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.AdaptiveAvgPool2d(1)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64, 6)
        )
```

3.6. Uczenie modelu

Do trenowania użyto:

- optymalizatora Adam z krokiem nauki 0.001,
- funkcji straty CrossEntropyLoss,
- harmonogramu zmniejszającego LR co 5 epok,
- 20 epok treningu na GPU (jeśli dostępne).

Zastosowano wagę klas (`compute_class_weight`), lecz ostatecznie nie została przekazana do funkcji straty.

3.7. Wyniki

Model trenowano przez 20 epok, osiągając najlepszą dokładność walidacyjną na poziomie **70,40%**. Na początku treningu skuteczność wynosiła 33,97%, a poprawa była stopniowa i stabilna. Uzyskane wyniki potwierdzają skuteczność podejścia z wykorzystaniem danych syntetycznych i prostej architektury CNN.

3.8. Testy odporności modelu

Aby ocenić odporność modelu na zakłócenia oraz jakość generalizacji, przeprowadzono testy na zbiorze testowym GTSRB z różnymi rodzajami transformacji. Testy wykonano bez ponownego trenowania modelu.

Wykorzystane transformacje:

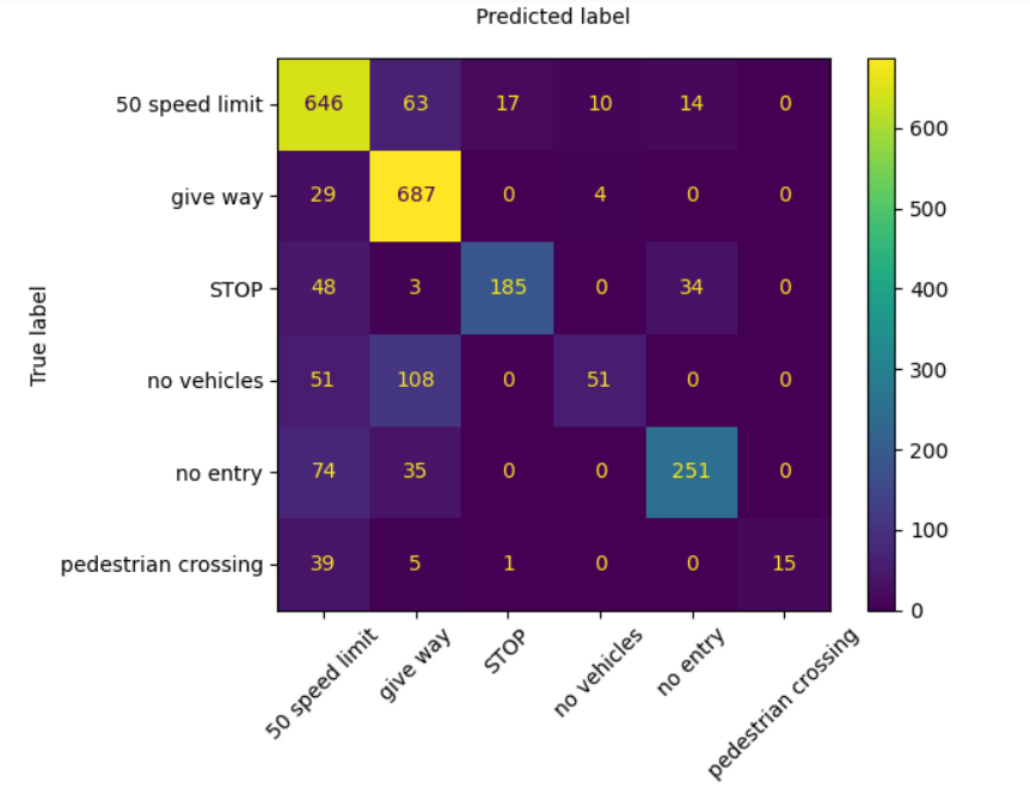
- **Clean (baseline)** – dane bez zakłóceń,
- **ColorJitter** – zmiana jasności, kontrastu i nasycenia,
- **Gaussian Blur** – rozmycie obrazu,
- **Affine** – obrót, przesunięcie, ścinanie,
- **RandomErasing** – losowe wymazanie fragmentu obrazu.

Uzyskane dokładności na danych testowych:

Rodzaj danych	**Dokładność**
Clean (baseline)	82.91%
ColorJitter	76.24%
Gaussian Blur	80.93%
Affine	82.49%
RandomErasing	73.59%

Model wykazuje dobrą odporność na zakłócenia geometryczne i rozmycie, jednak jego skuteczność spada przy mocnych zakłóceniach treści (jak `RandomErasing`).

Dodatkowo wygenerowano macierz pomyłek na zbiorze testowym:



Rysunek 2: Macierz pomyłek na danych testowych (Clean)

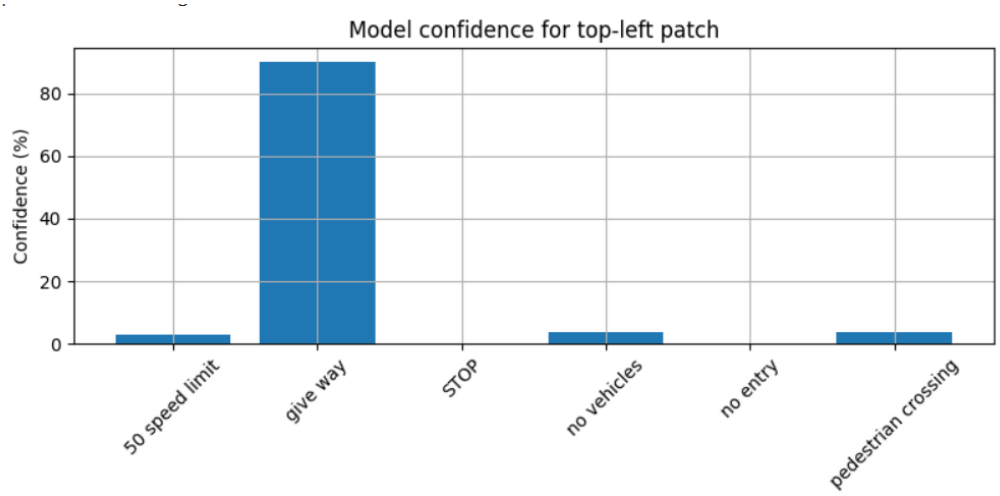
Najczęstsze pomyłki pojawiały się między znakami: *no entry* i *give way*, co może wynikać z ich podobnego kształtu i dominującej czerwieni. Ponadto model nie radził sobie dobrze z wykrywaniem *pedestrian crossing*.

3.9. Przypadek błędnej klasyfikacji

W celu sprawdzenia reakcji modelu na realne obrazy w niskiej rozdzielczości, przeprowadzono eksperyment z fragmentem zdjęcia przedstawiającego znak ograniczenia prędkości do 50 km/h. Wycięto górny lewy fragment obrazu (240x240), a następnie przeskalowano go do rozmiaru 64×64.

Rysunek 3: Porównanie: oryginalny fragment (240x240) vs przeskalowany (64x64)

Obraz został podany do klasyfikatora. Oto uzyskane prawdopodobieństwa dla każdej klasy:



Rysunek 4: Rozkład prawdopodobieństw (softmax) dla analizowanego fragmentu

Mimo że na obrazie znajduje się wyraźny znak *50 km/h*, model z wysokim prawdopodobieństwem zaklasyfikował go jako **give way** (89.98%). Klasa *50 speed limit* otrzymała jedynie 2.73% zaufania.

Wniosek: model może być wrażliwy na niską jakość obrazu, niepełne informacje lub zakłócenia. Przeskalowanie może prowadzić do utraty kluczowych cech wizualnych znaku, co skutkuje błędną klasyfikacją.

3.10. Eksport modelu do systemu wbudowanego

Po zakończeniu trenowania modelu przeprowadzono jego konwersję do formatu umożliwiającego użycie w systemach wbudowanych, takich jak ESSP-CAM.

3.10.1 Konwersja do TFLite

Model został najpierw wyeksportowany z PyTorch do formatu ONNX, a następnie przekonwertowany na TensorFlow i TFLite:

```
torch.onnx.export(model, dummy_input, "sign_model.onnx", ...)
onnx_model = onnx.load("sign_model.onnx")
tf_rep = prepare(onnx_model)
tf_rep.export_graph("sign_model_tf")

converter = tf.lite.TFLiteConverter.from_saved_model("sign_model_tf")
tflite_model = converter.convert()

with open("sign_model_float32.tflite", "wb") as f:
    f.write(tflite_model)
```

3.10.2 Wygenerowanie pliku C/C++

Model zapisany jako `.tflite` został przekształcony do postaci tablicy bajtów z użyciem komendy:

```
xxd -i sign_model_float32.tflite > sign_model.cc
```

Taki plik został bezpośrednio załączony do projektu mikrokontrolera, umożliwiając wykonanie inferencji z użyciem TFLite Micro bez dostępu do systemu plików.

4. Wykorzystanie modelu na mikrokontrolerze ESP32-CAM

Projekt wykorzystuje mikrokontroler **ESP32-CAM** z wbudowaną kamerą OV2640 oraz wsparciem dla Wi-Fi. Po przetrenowaniu i konwersji modelu do formatu `.tflite`, został on osadzony w pamięci mikrokontrolera jako tablica bajtów w pliku `sign_model.cc`. Mikrokontroler wykonuje klasyfikację obrazu przechwyconego z kamery oraz przesyła go przez HTTP.

4.1 Inicjalizacja systemu

Podczas uruchamiania programu inicjalizowane są następujące komponenty:

- system plików SPIFFS – do zapisu obrazu z kamery w formacie JPEG,
- kamera OV2640 – ustawiona na rozdzielczość QVGA (320×240), format JPEG,
- połączenie z siecią Wi-Fi – zgodnie z danymi z pliku `wifi_config.h`,
- serwer HTTP – umożliwia m.in. streaming obrazu z kamery oraz zapis zdjęcia,
- interpreter TFLite Micro – z załączonym modelem rozpoznawania znaków drogowych.

4.2 Wczytanie modelu i konfiguracja interpretera

Model zapisany w pliku `sign_model.cc` jest interpretowany jako wskaźnik do struktury `tflite::Model`. Następnie tworzony jest interpreter TFLite Micro z ręcznie zadeklarowanymi operatorami (m.in. `Conv2D`, `Pooling`, `Softmax`). Pamięć robocza (arena tensorów) alokowana jest w zewnętrznej pamięci SPIRAM i wynosi 384 kB.
Rozmiar wejściowy modelu to `64×64×3`, dane wejściowe są typu `float32`, a wyjściowe również `float32`.

4.3 Przetwarzanie obrazu i klasyfikacja

Po przechwyceniu obrazu przez kamerę wykonywane są kolejne kroki:

1. obraz w formacie JPEG jest dekodowany do bufora RGB888 przy użyciu biblioteki `esp_jpeg_decode`,
2. tworzone są fragmenty (okna) obrazu o wymiarach 64×64 piksele,

3. każdy fragment jest normalizowany i przekazywany do modelu,

4. dla każdego wycinka wyliczany jest wynik klasyfikacji.

Detekcja znaków odbywa się w osobnym wątku `detect_task()`, co pozwala na równoległe przetwarzanie obrazu bez blokowania głównej pętli programu. Jeżeli żaden fragment obrazu nie spełni kryteriów klasyfikacji, model zwraca wartość `-1`, a w systemowych logach pojawia się komunikat `"Brak znaku"`. Taka obsługa pozwala na skuteczne pomijanie nieistotnych lub niejednoznacznych ujęć.

5. Detekcja znaku na zdjęciu

Na mikrokontrolerze ESP32 zaimplementowano uproszczony system wykrywania znaków drogowych na obrazie RGB. System składa się z: detekcji kandydatów na znaki na podstawie koloru, przeszukiwania obrazu w wielu skalach oraz klasyfikacji patcha za pomocą modelu TFLite Micro.

5.1 Wstępna analiza kolorystyczna

Pierwszym krokiem jest identyfikacja potencjalnych obszarów zawierających znak drogowy. W tym celu obraz RGB o rozdzielczości 320×240 przeszukiwany jest fragmentami (ang. *patches*) o różnych rozmiarach. Dla każdego patcha wywoływana jest funkcja:

```
bool is_candidate(uint8_t* rgb, int width, int x, int y, int patch_size);
```

Jej zadaniem jest ocena, czy dany fragment może zawierać znak, na podstawie koloru i kontrastu. W szczególności:

- Dla każdego piksela obliczany jest odcień (`hue`) i nasycenie (`saturation`) bezpośrednio ze składników RGB — bez korzystania z zewnętrznych bibliotek.
- Piksele uznawane są za czerwone, jeśli `hue` znajduje się w zakresie $< 30°$ lub $> 330°$ oraz `saturation > 0.25`.
- Dodatkowo liczony jest kontrast pomiędzy centrum a otoczeniem patcha — musi on przekraczać 0.2.

Uwaga: Kolor czerwony nie jest wykrywany prostym sprawdzeniem R > G i R > B, ponieważ:

- rzeczywiste znaki mogą być ciemne (np. `(100, 10, 10)`),
- oświetlenie wpływa na jasność — nie na odcień (hue),
- analiza HSV pozwala wykryć "barwę czerwoną"niezależnie od jasności.

Tylko fragmenty spełniające oba warunki (kolor + kontrast) są uznawane za kandydatów do klasyfikacji.

5.2 Przeszukiwanie wieloskalarne (multi-scale)

Aby umożliwić detekcję znaków o różnych rozmiarach, obraz jest analizowany w kilku skalach jednocześnie. Patch'e o rozmiarach od 100% do 17% szerokości obrazu są przesuwane po obrazie z krokiem równym połowie patcha.
Dla każdego patcha:

1. wykonywana jest analiza koloru (`is_candidate`),
2. jeśli pozytywna — fragment kopiowany jest do bufora `patch_src`,
3. fragment przeskalowywany jest do 64×64 pikseli (format wejściowy modelu),
4. rozpoczynana jest klasyfikacja.

5.3 Przygotowanie danych wejściowych do modelu

Zeskalowany obraz (patch) konwertowany jest do formatu float32, z normalizacją do zakresu $[-1, 1]$:

```
in[j] = (resized_patch[j] / 255.0f - 0.5f) / 0.5f;
```

Dane są wpisywane do tensora wejściowego modelu TFLite.

5.4 Wykonanie inferencji i analiza wyniku

Model wywoływany jest za pomocą:

```
interpreter->Invoke();
```

Wyjściem modelu są logity — surowe wyniki klasyfikacji. Są one przeliczane na prawdopodobieństwa metodą softmax. Dodatkowo wyznaczany jest tzw. **margin** — różnica między najlepszą a drugą najlepszą klasą.

5.5 Akceptacja lub odrzucenie predykcji

Predykcja uznawana jest za poprawną tylko jeśli:

- prawdopodobieństwo zwycięskiej klasy > 0.4,
- różnica względem drugiej klasy > 0.1.

Jeśli tak — zwracany jest indeks klasy i poziom ufności. W przeciwnym razie analizowane są kolejne fragmenty obrazu.

Jeśli po przetworzeniu wszystkich skal i pozycji żaden fragment nie spełni kryteriów akceptacji, system uznaje, że na analizowanym obrazie nie znajduje się żaden rozpoznawalny znak drogowy.

5.6 Bufory i pamięć

System wykorzystuje pamięć PSRAM, gdzie alokowane są:

- `patch_src` — bufor na pełnowymiarowy fragment ($\leq 240 \times 240 \times 3$),
- `resized_patch` — bufor $64 \times 64 \times 3$ do wejścia modelu,
- `tensor arena` — blok 384 KB do działania interpretera TFLite Micro.

Dzięki temu możliwe jest działanie modelu nawet na ograniczonych zasobach ESP32.

6. Przykładowe wyniki

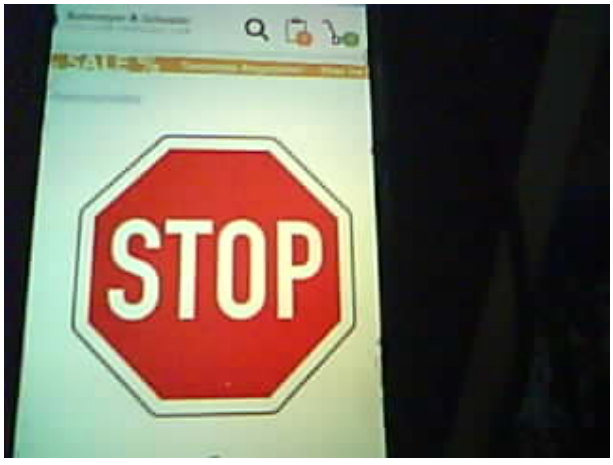
6.1 Eskperyment 1

```
I (8196) DETECTOR: Znak: give way (conf: 0.68)
```

Rysunek 5: Znak: give way

6.2 Eskperyment 2

```
I (8228) DETECTOR: Znak: STOP (conf: 0.59)
```



Rysunek 6: Znak: stop

6.3 Eskperyment 3

I (25618) DETECTOR: Znak: no entry (conf: 0.98)

Rysunek 7: Znak: no entry

7. Wnioski

Projekt pokazał, że możliwe jest stworzenie kompletnego systemu rozpoznawania znaków drogowych na mikrokontrolerze ESP32-CAM z wykorzystaniem uproszczonego modelu konwolucyjnej sieci neuronowej i frameworka TensorFlow Lite Micro. Zrealizowano pełny pipeline – od przygotowania danych i treningu modelu, po jego integrację i uruchomienie na urządzeniu wbudowanym.

Mimo działania systemu w czasie rzeczywistym i braku potrzeby komunikacji z zewnętrznymi serwerami, osiągnięte rezultaty nie są w pełni zadowalające. Skuteczność klasyfikacji na zbiorze testowym osiągnęła około 70–83%, jednak w praktyce pojawiały się istotne problemy z poprawnym rozpoznawaniem znaków. System często mylił ze sobą wizualnie podobne znaki — np. znak „give way" był rozpoznawany jako „no entry" lub odwrotnie. Znak „pedestrian crossing" był z kolei trudny do prawidłowego zaklasyfikowania w większości przypadków.

Największą słabością modelu okazała się jego ograniczona zdolność do odróżniania znaków o zbliżonej kolorystyce i kształcie, szczególnie przy niskiej rozdzielczości lub po przeskalowaniu fragmentu obrazu do wejściowego rozmiaru 64×64 pikseli. Przypadki błędnej klasyfikacji występowały także wtedy, gdy znak był częściowo zasłonięty lub oświetlony nierównomiernie.

Choć zastosowanie danych syntetycznych poprawiło ogólną jakość modelu, nie rozwiązało problemów z generalizacją na realnych danych. Dodatkowo, wymagania dotyczące wysokiej ufności klasyfikacji i progu marginesu między dwiema najlepszymi klasami nie zawsze chroniły przed błędnymi decyzjami.

Projekt należy traktować jako wstępny prototyp, pokazujący możliwości techniczne, ale również ograniczenia stosowania prostych modeli CNN w zastosowaniach wbudowanych.

Podsumowując, chociaż projekt spełnia swoje założenia funkcjonalne, dokładność klasyfikacji jest niewystarczająca do niezawodnego rozpoznawania znaków w warunkach rzeczywistych. Model myli znaki zbyt często, by mógł zostać wykorzystany w zastosowaniach praktycznych, ale stanowi wartościowy krok w kierunku dalszych prac nad lekkimi systemami wizyjnymi.