Sprawozdanie z projektu

Autor: Viktoria Krettek

1. Cel projektu

Celem projektu jest opracowanie sieci neuronowej, która rozpoznaje konkretny znak drogowy na zdjęciu, spośród 43 klas dostępnych w bazie GTSRB. Model ma przypisywać każdemu wejściowemu obrazowi dokładnie jeden rodzaj znaku (np. “stop”, “zakaz skrętu w lewo”, “ograniczenie do 50 km/h”).

Problem klasyfikacji znaków jest kluczowy dla systemów wspomagania kierowcy (ADAS) oraz pojazdów autonomicznych, wymagających szybkiej i niezawodnej interpretacji znaków w różnych warunkach drogowych (zmienne oświetlenie, przesłonięcia, zniekształcenia).

2. Opis danych

W projekcie użyłyśmy bazy German Traffic Sign Recognition Benchmark (GTSRB). Jest to publiczny zbiór danych zawierający obrazy znaków drogowych, podzielony na klasy i zapisany w postaci plików graficznych oraz odpowiadających im metadanych w formacie CSV. Każdy rekord zawiera ścieżkę do obrazu, etykietę klasy (ClassId), wymiary obrazu oraz współrzędne prostokąta obejmującego znak (Region of Interest – ROI).

Łącznie zbiór zawiera 43 różne klasy znaków drogowych, przypisane na podstawie etykiet liczbowych, które służą do jednoznacznej identyfikacji każdego znaku przez model klasyfikujący. Poniżej przedstawiono pełną listę etykiet i ich znaczenie:

- 0 - “Ograniczenie prędkości (20km/h)”
- 1 - “Ograniczenie prędkości (30km/h)”
- 2 - “Ograniczenie prędkości (50km/h)”
- 3 - “Ograniczenie prędkości (60km/h)”
- 4 - “Ograniczenie prędkości (70km/h)”
- 5 - “Ograniczenie prędkości (80km/h)”
- 6 - “Koniec ograniczenia prędkości (80km/h)”
- 7 - “Ograniczenie prędkości (100km/h)”
- 8 - “Ograniczenie prędkości (120km/h)”
- 9 - “Zakaz wyprzedzania”
- 10 - “Zakaz wyprzedzania poj powyżej 3.5 tony”

- 11 - “Pierwszeństwo na skrzyżowaniu”
- 12 - “Droga z pierwszeństwem”
- 13 - “Ustąp pierwszeństwa”
- 14 - “Stop”
- 15 - “Zakaz wjazdu wszelkich pojazdów”
- 16 - “Zakaz wjazdu pojazdów powyżej 3.5 tony”
- 17 - “Zakaz wjazdu”
- 18 - “Uwaga niebezpieczeństwo”
- 19 - “Niebezpieczny zakręt w lewo”,
- 20 - “Niebezpieczny zakręt w prawo”,
- 21 -”Podwójny zakręt"',
- 22 - “Wyboista droga”,
- 23 - “Śliska nawierzchnia”,
- 24 - “Zwężenie jezdni z prawej strony”,
- 25 - “Prace drogowe"',
- 26 - “Sygnalizacja świetlna”,
- 27 - “Piesi”,
- 28 - “Przejście dla dzieci”,
- 29 - “Przejazd dla rowerzystów”,
- 30 - “Uwaga oblodzenie/śnieg na drodze”,
- 31 - “Przejście dzikich zwierząt”,
- 32 - “Koniec ograniczeń prędkości i zakazu wyprzedzania”,
- 33 - “Nakaz skrętu w prawo”,
- 34 - “Nakaz skrętu w lewo”,
- 35 - “Nakaz jazdy prosto”,
- 36 - “Jazda prosto lub w prawo”,
- 37- “Jazda prosto lub w lewo”,
- 38 - “Trzymaj się prawej strony”,
- 39 - “Trzymaj się lewej strony"',
- 40 - “Ruch okrężny - obowiązkowy”,
- 41 - “Koniec zakazu wyprzedzania”,
- 42 - “Koniec zakazu wyprzedzania pojazdów powyżej 3,5 tony”.

3. Przygotowanie danych

Do katalogu roboczego Jupyter Notebook dodałyśmy plik “archive(1).zip”, który zawiera dane GTSRB. Dane zostały rozpakowane do folderu “german-traffic-signs” przy użyciu biblioteki zipfile, co umożliwiło dostęp do obrazów znaków oraz plików zawierających metadane.

Następnie utworzyłyśmy zestaw transformacji wejściowych, aby można je było przekazać do obiektów “Dataset”, które miały na celu ujednolicenie formatu obrazów:

- przeskalowanie obrazów do wymiaru 32x32 piksele,
- przekonwertowanie na tensory - format wymagany przez PyTorch,
- zastosowanie w niektórych przypadkach standaryzacji wartości pikseli.

W następnym kroku zdefiniowałyśmy własną klasę zbiorów danych “GTSRBDataset”. Jej zadaniem było:

- wczytanie danych z plików “.csv”,
- odczytywanie ścieżek do obrazów,
- przypisywanie etykiet,
- automatyczne stosowanie wcześniej zdefiniowanych transformacji.

Kolejno wydzieliłyśmy z oryginalnego zbioru treningowego: zbiór treningowy zawierający 80% danych oraz zbiór walidacyjny - 20% danych. Taki podział pozwala na ocenę modelu podczas trenowania - na danych, które nie były używane w procesie uczenia.

Dla każdego zbioru, czyli treningowego, testowego oraz walidacyjnego, przygotowałyśmy obiekty “DataLoader”, które umożliwiły przetwarzanie danych w paczkach po 64 obrazy. W przypadku zbioru treningowego dane były losowe mieszane przed każdą epoką.

Dodatkowo przeanalizowałyśmy zawartość zbiorów, aby upewnić się, że dane są poprawne. Sprawdziłyśmy zakres oraz liczbę różnych etykiet.

4. Budowa modelu

W projekcie zdecydowałyśmy się na zastosowanie konwolucyjnej sieci neuronowej (CNN), ponieważ tego typu architektura jest szczególnie dobrze przystosowana do analizy danych obrazowych, takich jak fotografie znaków drogowych z bazy GTSRB.

Nasz model składa się z dwóch warstw konwolucyjnych z funkcją aktywacji ReLu i poolingiem.

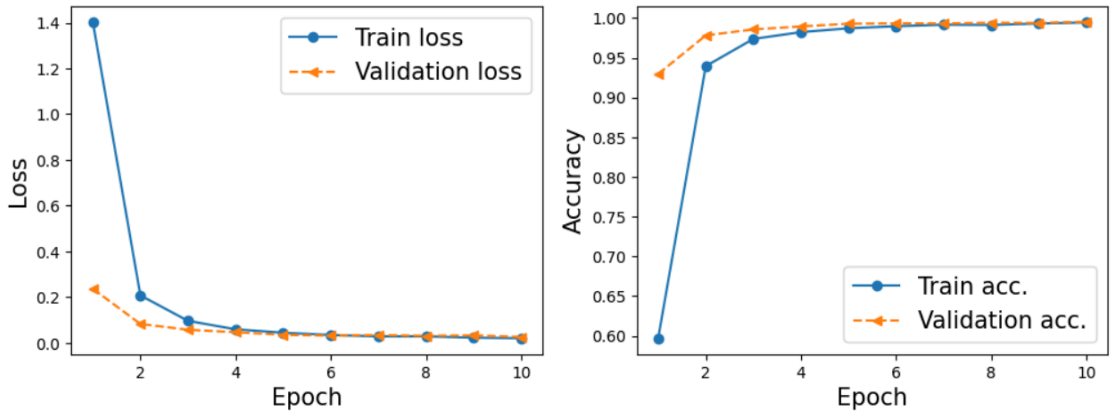
Utworzyłyśmy obiekt device wskazujący, że model będzie trenowany i testowany na procesorze (CPU). Przeniosłyśmy model (czyli wszystkie jego warstwy i parametry) na wskazane urządzenie. Jest to konieczne, aby dane wejściowe i model znajdowały się na tym samym urządzeniu, co umożliwia poprawne obliczenia.

Jako funkcję strat wybrałyśmy nn.CrossEntropyLoss (ocenia, jak bardzo przewidywania modelu odbiegają od prawidłowej klasy - to na podstawie tej wartości model “uczy się”, jak się poprawić), a jako optymalizator Adam (aktualizuje wagi modelu, aby zmniejszyć wartość funkcji straty, czyli poprawić jakość klasyfikacji znaków drogowych). Ustaliłyśmy lr = 0.001 jako początkową wartość współczynnika uczenia.

Następnie zdefiniowałyśmy funkcję train(), która:

- obsługuje zarówno trening, jak i walidację w każdej epoce,
- gromadzi i zwraca dane o skuteczności modelu,
- jest przygotowana do pracy na CPU (lub GPU),
- jako argumenty przyjmuje model sieci neuronowej, liczbę epok oraz dataloadery dla zbiorów treningowego i walidacyjnego,
- tworzy listy do zapisu średnich strat i dokładności dla każdej epoki (osobno dla treningu i walidacji),
- włącza tryb treningowy modelu,
- przechodzi przez wszystkie batch'e danych treningowych,
- gromadzi sumaryczną stratę i liczbę poprawnych klasyfikacji,
- po każdej epoce uśrednia wyniki,
- wykonuje obliczenia strat i dokładności, ale bez aktualizacji wag.

Uruchomiłyśmy trening modelu przez 10 epok, a następnie sporządziłyśmy wykresy krzywych uczenia:



Wykres po lewej stronie pokazuje jak zmieniała się strata podczas uczenia modelu, a prawy wykres dotyczy dokładności.

Strata na obu zbiorach szybko spada w pierwszych epokach, a następnie stabilizuje się blisko zera. Model bardzo szybko się uczy - osiąga niski błąd już po kilku epokach. Brak oznak przeuczenia - strata walidacyjna nie zaczyna rosnąć ani nie odbiega znacząco od treningowej.

Podczas pierwszych 6 epok dokładność na zbiorze treningowym gwałtownie wzrasta z 60% do blisko 100%. Dokładność walidacyjna także szybko wzrasta i stabilizuje się nie odbiegając od wartości na zbiorze treningowym. Nie zaobserwowałyśmy spadku dokładności na zbiorze walidacji co sugeruje, że model nie uczy się "na pamięć", ale rozpoznaje wzorce.

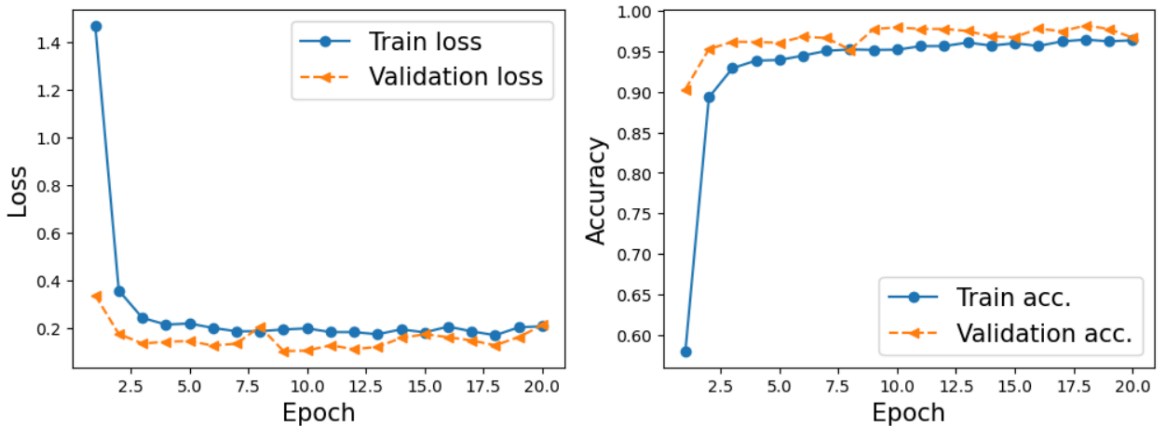
Sprawdziłyśmy działanie modelu na zbiorze testowym. Ogólny procent poprawnych klasyfikacji w całym zbiorze testowym wyniósł 95,15%. Wysoki wynik oznacza, że model dobrze nauczył się rozróżniać między 43 różnymi klasami znaków drogowych.

5. Dopracowanie modelu

Przetestowałyśmy różne opcje:

A. Aby dopracować model początkowo architektura sieci zostaje identyczna jak w poprzednim przypadku. Zmieniłyśmy natomiast dwa parametry treningu:

- learning rate ustawiłyśmy na 0.005,
- zwiększyłyśmy liczbę epok do 20.



Na wykresie straty widzimy, że zarówno train loss, jak i validation loss początkowo szybko maleją, co oznacza, że model uczy się poprawnie. Jednak od około 7–8 epoki strata walidacyjna przestaje spadać i zaczyna oscylować, co sugeruje, że model przestaje się poprawiać pod względem generalizacji.

Model uzyskał dokładność na zbiorze testowym równą 89,56%, jest to niższy wynik niż w poprzednim eksperymencie.

Podsumowując zwiększenie liczby epok i zastosowanie większego learning rate nie poprawiło działania modelu, natomiast spowodowało pogorszenie dokładności na zbiorze testowym.

B. Inna architektura sieci

Dokonałyśmy modyfikacji poprzedniej architektury poprzez:

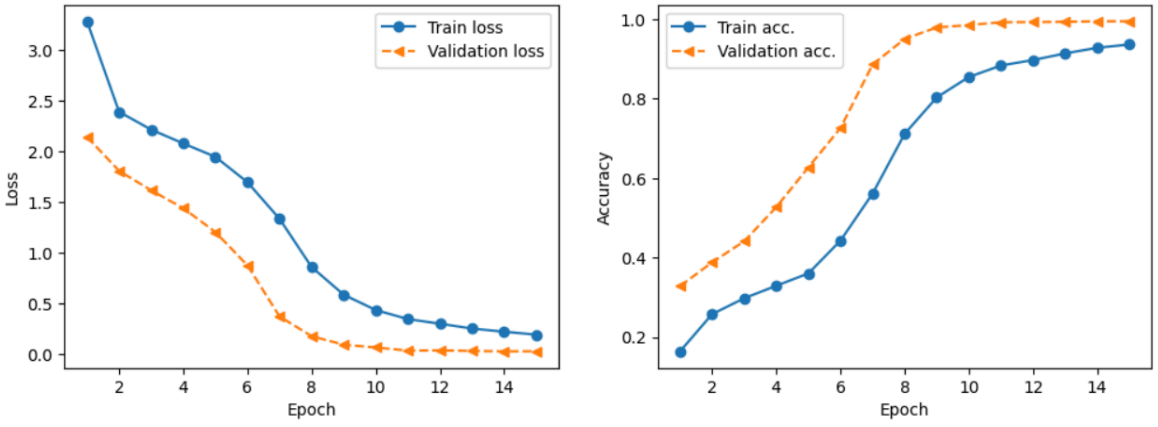
- pogłębienie sieci - 4 warstwy konwolucyjne zamiast 2,
- użycie BatchNorm w celu normalizacji aktywacji w trakcie uczenia, co przyspiesza trening, stabilizuje gradienty, redukuje ryzyko przeuczenia, pozwala na użycie większych learning rate'ów,
- zastosowanie małych jąder 3x3 z paddingiem (pozwala to lepiej zachować rozdzielczość obrazu, efektywniej wyłapuje lokalne cechy, umożliwia większą głębokość modelu bez drastycznego zmniejszania rozmiaru obrazu),
- połączenie warstwy z Dropoutem (512 neuronów) (dropout pomaga w redukcji przeuczenia, losowo "wyłączając" część neuronów podczas treningu),
- wzbogacenie reprezentacji znaku drogowego przed klasyfikacją - po 2 poolingach mapa cech ma rozmiar 8×8×128 = 8192 cechy.

Funkcja strat i optymalizator zostały takie same jak w poprzednim modelu, natomiast learning rate wynosił 0.005.

Następnie zdefiniowałyśmy funkcję train_model(), która od poprzednio zdefiniowanej funkcji train() różni się:

- prostszym liczeniem strat (loss.item() bez mnożenia),
- dokładnością (accuracy) liczoną jako liczba trafień podzielona przez total_train lub total_valid - efektywnie to samo.

Utworzyłyśmy nowe loadery danych i rozpoczęłyśmy trening przez 15 epok. Sprawdziłyśmy działanie nowego modelu na zbiorze testowym, jego wynik to 97,31%. Wykresy krzywych uczenia prezentowały się w następujący sposób:

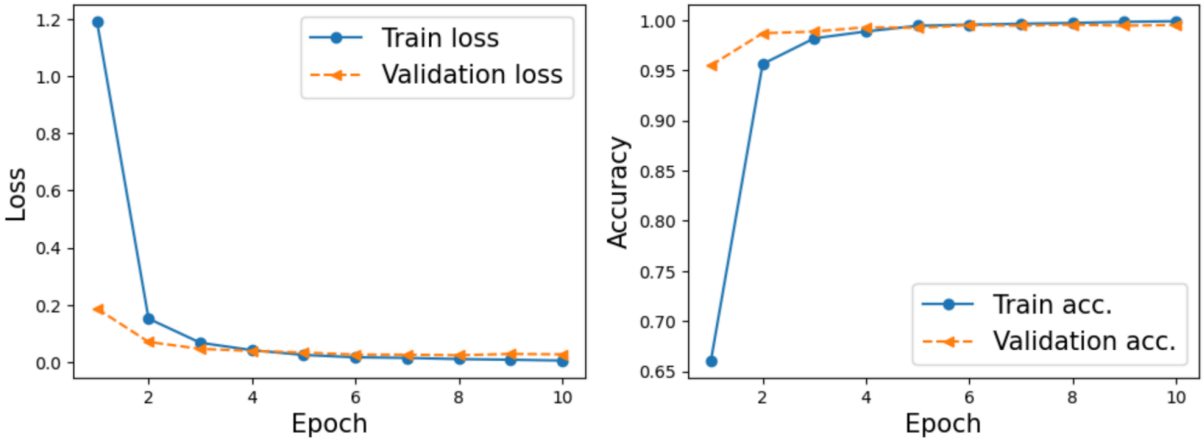


Validation accuracy rosło szybciej niż train accuracy, które nie dogania dokładności walidacyjnej. Może to sugerować, że model jest dobrze dopasowany do danych

walidacyjnych, ale nie nauczył się jeszcze wystarczająco dobrze zbioru treningowego, być może ze względu na zwiększoną złożoność sieci. W porównaniu do pierwszego modelu, tutaj widać większą różnicę między krzywymi dokładności.

C. Następnie używając początkowego modelu, czyli bez dodawania kolejnych warstw, zmodyfikowałyśmy sposób aktualizacji danych:

- zastosowałyśmy scheduler Exponential Learning Rate Decay (gamma = 0.89),
- learning rate = 0.001,
- liczba epok 10.



Wykresy strat bardzo szybko spadają w pierwszych kilku epokach i stabilizują się na niskim poziomie. Nie widać oznak przeuczenia ani wzrostu błędu walidacyjnego. Train accuracy rośnie gwałtownie, osiągając blisko 100% już w 5. epoce i pozostaje stabilna. Validation accuracy również osiąga bardzo wysoki poziom, co oznacza bardzo dobrą jakość klasyfikacji na danych walidacyjnych.

Model osiągnął test accuracy = 95.22%, czyli wynik praktycznie taki sam jak w pierwszym modelu, ale uzyskany dzięki zastosowaniu scheduler'a i przy tej samej liczbie epok.

Podsumowując zastosowanie ExponentialLR pozwoliło uzyskać bardzo dobrą jakość klasyfikacji, przy zachowaniu stabilnych wykresów strat i dokładności. Wyniki testowe są porównywalne z najlepszym pierwszym modelem. Ten model można uznać za równie skuteczny jak pierwszy, z dodatkową korzyścią lepszej kontroli procesu treningu dzięki użyciu scheduler'a.

6. Prezentacja działania najlepszego modelu

Poniżej przedstawiłyśmy 12 przykładów poprawnej klasyfikacji znaków drogowych, w tym celu skorzystałyśmy z najlepszego modelu. Naszym zdaniem był to model pierwszy. Warto zaznaczyć, że dokładność ostatniego modelu jest porównywalna.

Zamieściłyśmy również pierwsze 12 próbek, gdzie model nie poradził sobie dobrze. Zaobserwowałyśmy, że są to głównie zdjęcia wykonane w niekorzystnych warunkach (złe oświetlenie, rozmazany obraz).