Przetwarzanie i Analiza Obrazów Medycznych

Tytuł projektu: Klasyfikacja stopnia retinopatii cukrzycowej

Autor raportu: Łukasz Nowosad

Co zostało zrobione?

Moim zdaniem w tym projekcie było przygotowanie odpowiedniej struktury projektu na GitHubie, README, automatycznego pobierania oraz przeprocesowania danych, stworzenie data module oraz architektury własnych sieci neuronowych.

Struktura projektu

W celu zachowania odpowiedniej struktury i porządku w repozytorium stworzono szkielet na main, aby potem osoby zaangażowane w dalszą pracę mogły w prosty sposób zpullować projekt z gałęzi main. Po odpaleniu skryptu wyglądał on następująco:

```
PAOM_retinopathy/
├── data/
│   ├── processed/
│   └── raw/
├── notebooks/
├── src/
│   ├── models/
│   ├── datamodules/
│   ├── training/
│   ├── utils/
│   └── main.py
├── .venv/
├── README.md
└── requirements.txt
```

Rys. 1. Drzewko projektu

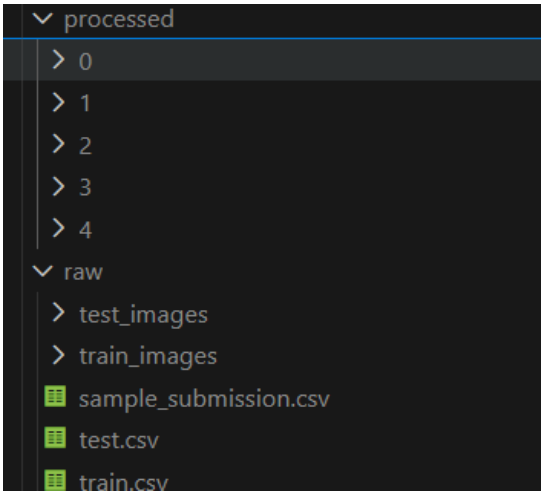
Występuje tutaj logiczne rozmieszczenie folderów, tak aby każdy nowo tworzony plik znajdował się w odpowiednim dla siebie folderze. Ostatecznie ze względu na to, ze wykorzystany został tylko jeden plik *.ipynb zrezygnowano z folderu notebooks.

W ramach tworzenia projektu zajęto się również środowiskiem wirtualnym, tj. stworzono plik z bibliotekami potrzebnymi do dalszej pracy.

Preporcessing oraz pobieranie danych

Z racji tego, że dataset dostępny był na kagglu, postanowiono wykorzystać jego API w celu automatycznego pobierania potrzebnych data setów do treningu i ewaluacji. Konieczne kroki zostały opisane w README, jednak można podsumować to tym, że wyamgane było założenie konta i wzięcie udziału w zawodach, aby następnie samemu za pomocą swojego API-Key pobrać dany dataset.

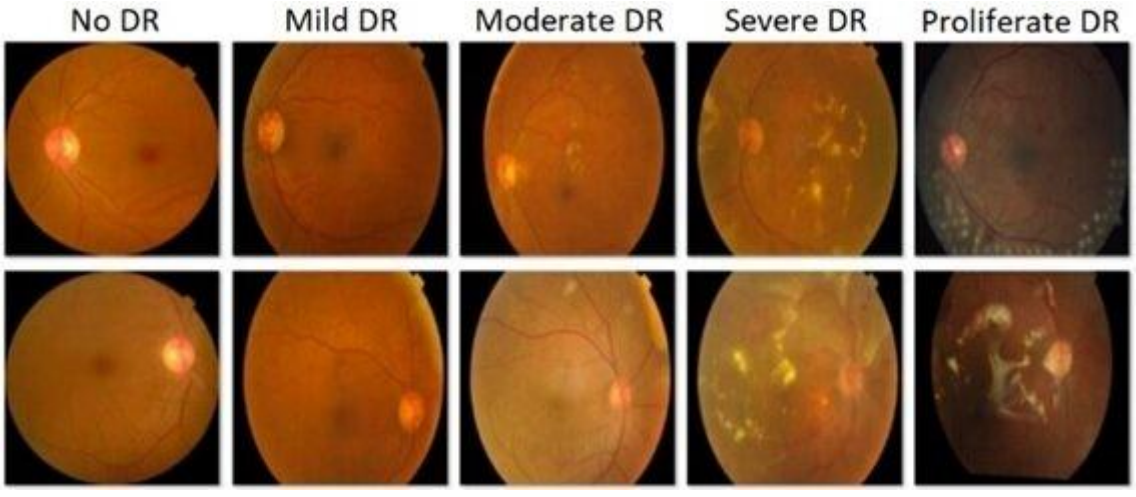
Następnie dla surowych danych wykonany został preprocessing, na podstawie danych z pliku train.csv stworzono uporządkowaną strukturę ze względu na klasy, transformacja wyglądała następująco:



Rys. 2. Transformacja datasetu z kaggle na bardziej użyteczny

Datamodule oraz Dataset

Kolejnym krokiem była implementacja RDDataset oraz RDDatamodule. Szczegóły implementacyjne znajdują się na GitHubie, natomiast ważnym w tym kontekście tematem była odpowiednia transformacja danych. Zdjęcia z datasetu charakteryzowały się tym, że wokół istotnej dla nas informacji (czyli skanu oka) występowała czarna kołowa obwiednia.



Rys. 3. Przykładowe skany oka w kontekście RD

Było to dosyć ważne w kontekście tego, że chcieliśmy uniknąć pikseli, które nie niosą ze sobą żadnych informacji. W tym celu została zaimplementowana transformacja CropTransform, która przycinała obraz, tak aby pozostawić jak najmniej czarnej obwiedni. Operacja ta polegała na konwersji obrazu do skali szarości oraz wyznaczeniu maski pikseli o intensywności większej od ustalonego progu, co pozwalało na lokalizację właściwego obszaru obrazu zawierającego informacje diagnostyczne. Następnie obraz był przycinany do minimalnego prostokąta obejmującego te piksele. Dodatkowo wykorzystano CLAHE, którego celem było zwiększenie lokalnego kontrastu obrazu.

```
self.train_transform = transforms.Compose([
    CropTransform(),
    transforms.Resize((224, 224)),
    CLAHETransform(),
    transforms.ToTensor(),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5])
])

self.val_test_transform = transforms.Compose([
    CropTransform(),
    transforms.Resize((224, 224)),
    CLAHETransform(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5])
])
```

Rys. 4. Transformacje testowe i val/test

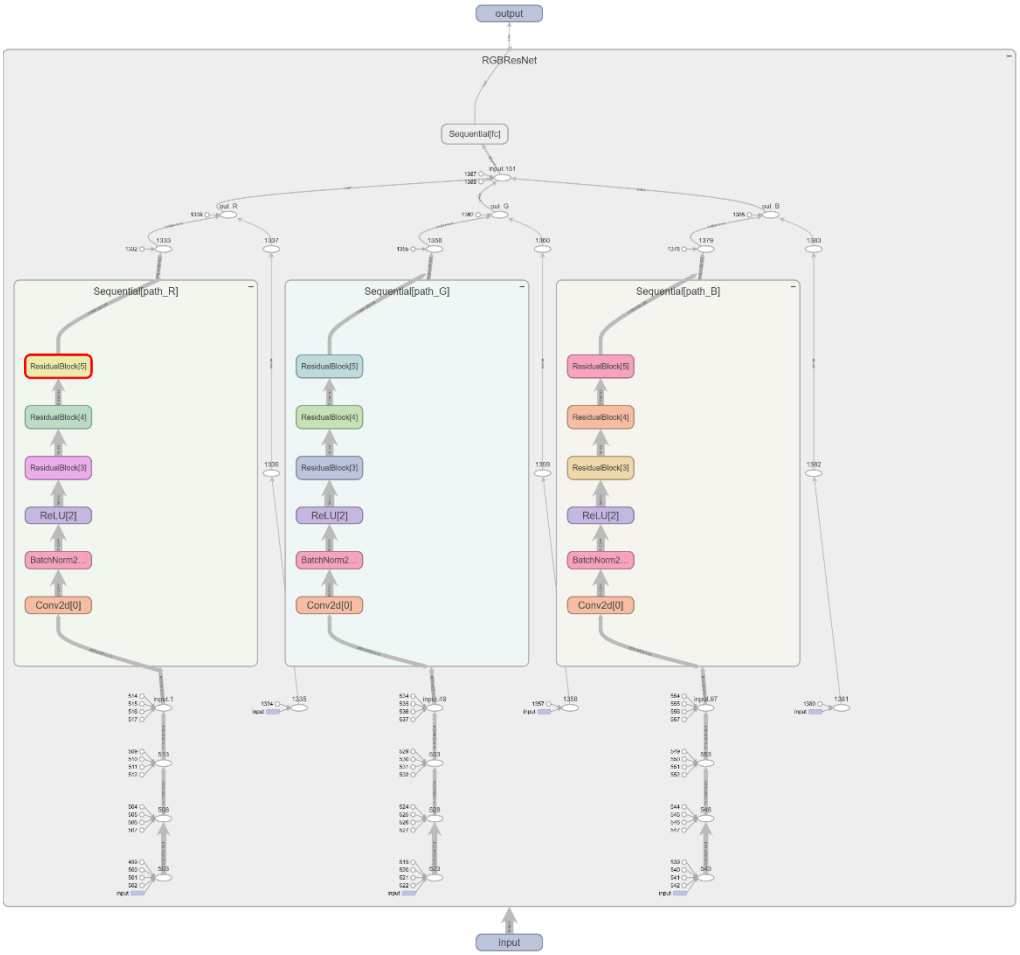
Stworzone architektury

Stworzono dwie architektury sieci, jedną podstawową konwolucyjną zaczerpniętą z zajęć oraz drugą bardziej skomplikowaną. Sieci można sobie zwizualizować za pomocą tensorboarda, kod również dostępny jest na githubie.

Sieć DeepCNN składa się z kilku kolejnych bloków konwolucyjnych, zawierających warstwy konwolucyjne, normalizację batcha oraz funkcję aktywacji ReLU, pomiędzy którymi zastosowano operacje max pooling.

Sieć RGBResNet jest oparta na blokach rezydualnych. Architektura ta została zaprojektowana w taki sposób, aby niezależnie przetwarzać informacje zawarte w poszczególnych kanałach barwnych obrazu RGB. Obraz wejściowy jest rozdzielany na kanały R, G oraz B, a następnie każdy z nich przetwarzany jest przez osobną, identyczną ścieżkę konwolucyjną.

Każda ścieżka składa się z warstwy konwolucyjnej oraz kilku bloków rezydualnych, które umożliwiają efektywniejsze uczenie głębokiej sieci dzięki zastosowaniu połączeń skrótowych (*skip connections*). Po zakończeniu części konwolucyjnej stosowane jest globalne uśrednianie przestrzenne (*Global Average Pooling*), a uzyskane wektory cech z trzech kanałów są łączone i przekazywane do części w pełni połączonej odpowiedzialnej za klasyfikację.



Rys. 5. Archtiektura sieci RGBResNet