Akademia Górniczo-Hutnicza im Stanisława Staszica
Wydział Automatyki, Elektroniki Informatyki i Inżynierii Biomedycznej

PORÓWNANIE ARCHITEKTURY I WYNIKÓW SIECI NEURONOWYCH W KLASYFIKACJI CHORÓB PŁUC

PROJEKT REALIZOWANY W RAMACH PRZEDMIOTU
TECHNIKI OBRAZOWANIA MEDYCZNEGO

Monika Noga
Oskar Trybus
Aleksandra Szmigiel
Izabela Wilusz
Katarzyna Cupiał

Kraków, 09 czerwca 2021

Spis treści

1. Wstęp

1.1. Cel

Celem projektu jest porównanie jak architektura sieci neuronowej wpływa na wyniki klasyfikacji zdjęć rentgenowskich klatki piersiowej dla pacjentów z Covid-19, zmętnieniem płuc i zdrowych.

1.2. Założenia

Podczas pracy nad projektem wykorzystano zbiór zdjęć RTG pacjentów z chorobami płuc z serwisu Kaggle. Klasyfikację przeprowadzono za pomocą dwóch sieci konwolucyjnych zaimplementowanych z pomocą bibliotek PyTorch i TensorFlow dostępnych w języku Python dedykowanych między innymi do tego typu celów.

1.3. Wykres Gantta

Przed przystąpieniem do realizacji projektu, przygotowano plan działania w formie wykresu Gantta (Rysunek 1).

		10.05 - 16.05							17.05 - 23.05							24.05 - 30.05							31.05 - 06.06						
Zadanie	Osoba odpowiedzialna	P	W	Ś	CZ	PT	S	N	P	W	Ś	CZ	PT	S	N	P	W	Ś	CZ	PT	S	N	P	W	Ś	CZ	PT	S	N
Definicja celu projektu	Wszyscy																												
Wybranie zbioru danych	Wszyscy																												
Przegląd literaturowy	Katarzyna Cupiał																												
Augumentacja danych	Monika Noga																												
Preprocessing danych	Monika Noga																												
Implementacja sieci (Pytorch)	Oskar Trybus																												
Implementacja sieci (Tensorflow)	Monika Noga																												
Ewaluacja (Tensorflow)	Izabela Wilusz																												
Ewaluacja (Pytorch)	Aleksandra Szmigiel																												
Dyskusja wniosków	Wszyscy																												

Rysunek 1. Wykres Gantta

1.4. Kamienie milowe

Następnie zdefiniowano wybrane kamienie milowe projektu.

A. Wybranie zbioru danych i zdefiniowanie celu projektu
   a. Problem do rozwiązania: zdefiniowanie celu, wybranie odpowiedniego zbioru danych
   b. Opis prac: określenie celu projektu, przejrzenie dostępnych zbiorów danych i wybranie najbardziej odpowiedniego do spełnienia założonego celu
   c. Parametr(y): wybrany zbiór danych, określony cel projektu
   d. Wpływ nieosiągnięcia kamienia milowego: brak możliwości dalszej pracy nad projektem

B. Augmentacja i preprocessing danych
   a. Problem do rozwiązania: nierówne rozmiary poszczególnych klas, nieodpowiednia struktura danych
   b. Opis prac: wzbogacenie klas o zbyt małej liczebności przez odpowiednie operacje, przygotowanie odpowiedniej struktury danych,
   c. Parametr(y): rozmiary klas, odpowiednia struktura danych,
   d. Wpływ nieosiągnięcia kamienia milowego: problemy z wykorzystaniem danych w sieciach, przeuczenie sieci z powodu zbyt małej liczebności niektórych klas

C. Implementacja
   a. Problem do rozwiązania: stworzenie sieci do klasyfikacji
   b. Opis prac: podziała danych na grupy treningową i testową, przygotowanie dwóch sieci z wykorzystaniem dwóch bibliotek PyTorch i TensorFlow, nauczenie sieci na przygotowanych zbiorach danych
   c. Parametr(y): zbiory treningowy, testowy o podziale 8:2, dokładność sieci na poziomie przynajmniej 80%
   d. Wpływ nieosiągnięcia kamienia milowego: brak wyników do omówienia lub zbyt mało wiarygodne wyniki, aby móc się na nich opierać
D. Ewaluacja wyników
   a. Problem do rozwiązania: sprawdzenie, czy przygotowane sieci klasyfikują dane poprawnie
   b. Opis prac: wybór metryk do oceny wyników, implementacja metryk, opracowanie odpowiednich wizualizacji umożliwiających lepsze zrozumienie danych
   c. Parametr(y): minimum 5 metryk do oceny modeli i odpowiednio 5 wizualizacji
   d. Wpływ nieosiągnięcia kamienia milowego: problem z oceną otrzymanych modeli, brak możliwości wyciągnięcia odpowiednich wniosków
E. Przegląd literaturowy
   a. Problem do rozwiązania: poznanie istniejących już rozwiązań w pracach o podobnych zbiorach danych
   b. Opis prac: zapoznanie się z artykułami o tematyce podobnej do omawianej w projekcie
   c. Parametr(y): przynajmniej 2 artykuły o podobnej tematyce
   d. Wpływ nieosiągnięcia kamienia milowego: brak możliwości porównania wyników otrzymanych w projekcie do innych dostępnych rozwiązań

1.5. Streszczenie

Odpowiednio skonstruowane i wytrenowane sieci neuronowe mogą służyć jako pomoc w rozpoznaniu i klasyfikacji obrazów rentgenowskich płuc ze zmianami COVID-19. W tym projekcie przedstawiamy dwie różne architektury, które zostały wytrenowane na zbiorze danych z serwisu keggle. Zadaniem tych sieci jest klasyfikacja obrazów na trzy kategorie: normal - COVID - lung opacity. Z powodu różnej ilości zdjęć w poszczególnych grupach zdecydowano się zastosować augmentację danych, zwiększając dwukrotnie liczebność klasy COVID. Sieć EfficientNet implementowana wykorzystując bibliotekę PyTorch osiągnęła dokładność na poziomie 97,04%. Natomiast autorska sieć implementowana z wykorzystaniem biblioteki TensorFlow osiągnęła dokładność 91%.

2. Wprowadzenie

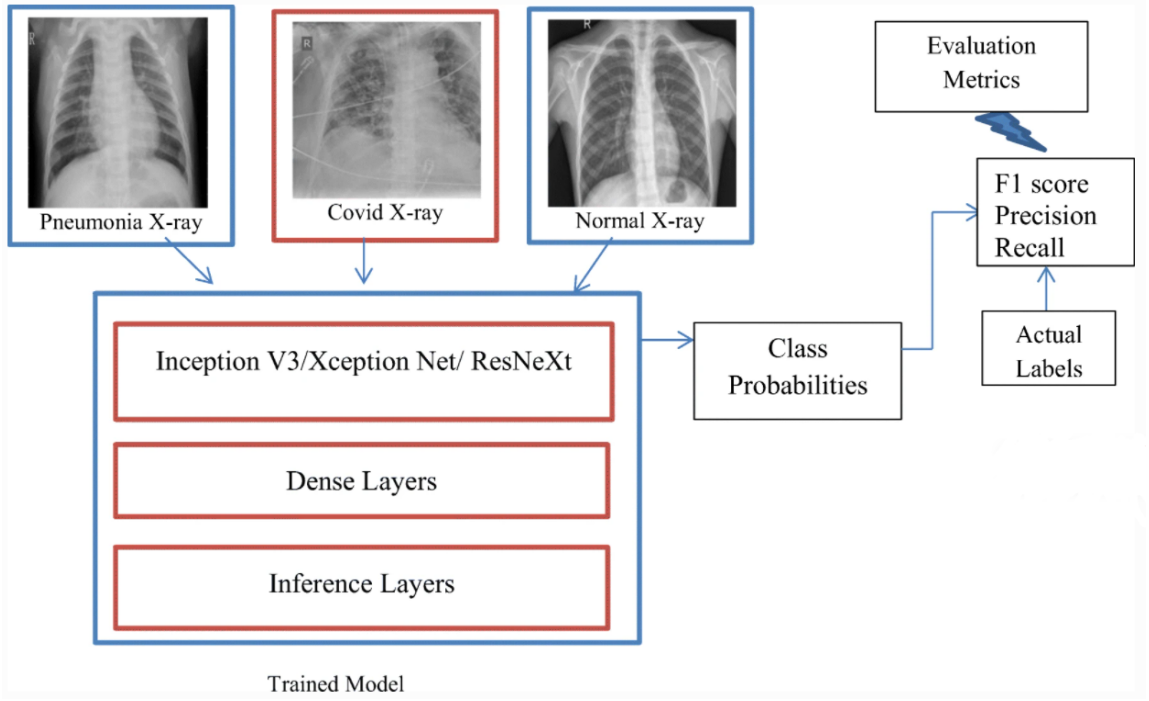
Pojawienie się COVID-19 wywołanego wirusem SARS-CoV-2 wywołało ogromne zmiany w wielu aspektach naszego życia. Naukowcy, zaraz obok medyków skupili całą swoja uwagę i umiejętności w poszukiwaniu sposobów na rozpoznanie i opanowanie choroby. Objawy COVID-19 są bardzo różnorodne, jednak te związane z oddychaniem i płucami są jednymi z najbardziej charakterystycznych. Podczas rozwoju pandemii bardzo ważne okazało się szybkie rozpoznanie choroby. Jedną z metod screeningu są zdjęcia rentgenowskie, jednakże wymagają one analizy radiologa, a ponadto objawy COVID-19 widoczne w płucach bardzo przypominają inne choroby m.in.: zapalenie płuc. Przeszkody te opóźniają szybką i bezbłędną diagnostykę COVID-19, mimo dostępności aparatury. Dlatego

też techniki rozpoznania oparte na sztucznej inteligencji mogą pomóc rozwiązać te problemy. Od początku pandemii wiele zespołów stworzyło lub zmodyfikowało istniejące architektury sieci neuronowych w celu osiągnięcia jak najwyższej dokładności rozpoznania COVID-19.

3. Przegląd literatury

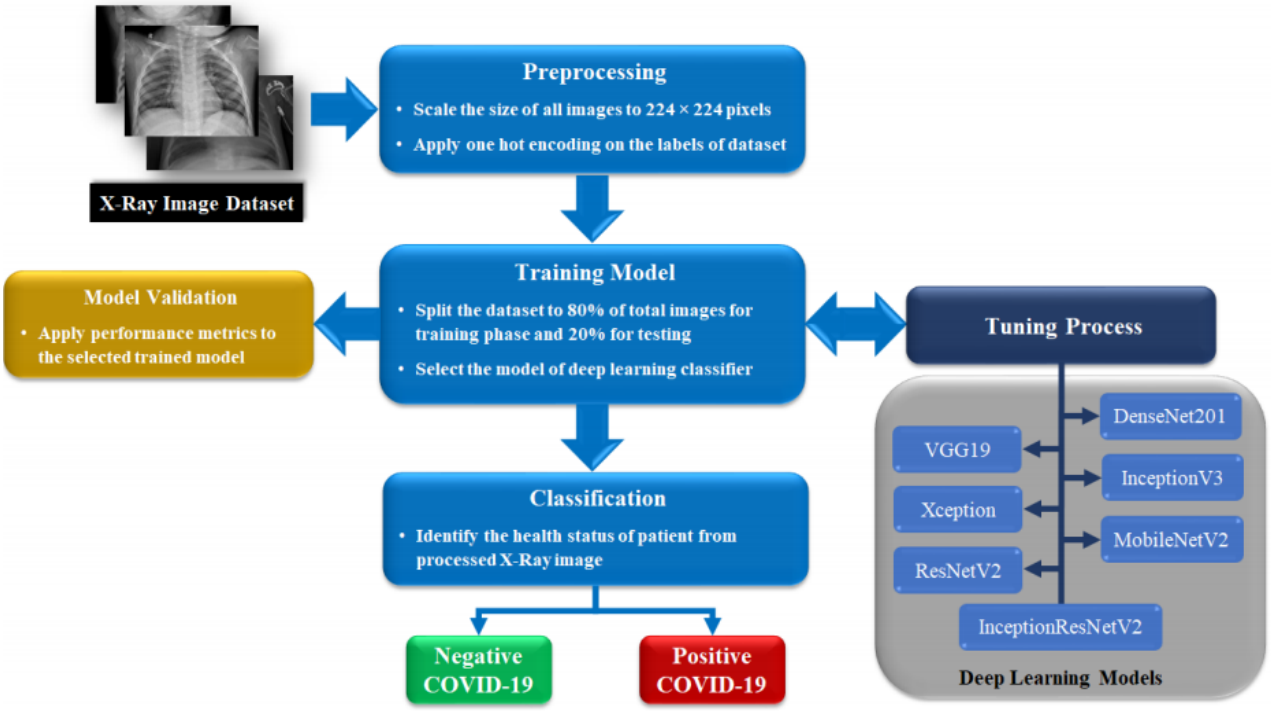
Podczas tworzenia sieci neuronowych służących do rozpoznawania COVID-19 zastosowano różne podejścia. Wiele zespołów zwróciło się do istniejących architektur w celu zbadania jak poradzą sobie ze zdjęciami pacjentów z COVID-19.

Zespół: Rachna Jain, Meenu Gupta, Soham Taneja i D. Jude Hemanth porównał działanie modeli Inception V3, Xception oraz ResNeXt. Pierwszym problemem z którym zmierzyli się autorzy, a który jest obecny do dzisiaj była zbyt mała ilość danych, szczególnie tych medycznych. W celu rozwiązania tej przeszkody posłużono sią augmentacją obrazów w formie rotacji oraz przybliżania. Po przygotowaniu danych przystąpiono do trenowania wybranych modeli, schemat badania przedstawiono na rysunku 2. Najwyższą dokładność, bo aż 97,97% wykazała sieć Xception [1].



Rysunek 2. Schemat badania i ewaluacji 3 sieci w detekcji COVID-19

Inne zespoły stworzyły struktury dedykowane do rozpoznania COVID-19. Zespół: Ezz El-Din Hemdan , Marwa A. Shouman i Mohamed Esmail Karar przedstawił sieć COVIDX-Net. Opiera się ona na 7 innych sieciach: VGG19, DenseNet121, InceptionV3, ResNetV2, Inception-ResNet-V2, Xception, MobileNetV2. Na rysunku 3 przedstawiono workflow struktury COVIDX-Net. Najlepszymi metrykami charakteryzują się VGG19 oraz DenseNet201, ich dokładność wyniosła 90% [2].



Rysunek 3. Schemat działania struktury COVIDX-NET

Od początku pandemii stworzono, wytrenowano i zewaluowano wiele sieci neuronowych do celów rozpoznania COVID-19, w tabeli 1. zebrano część otrzymanych wyników. Badacze podczas badania sieci skupiają się głównie na formach klasyfikacji 2 i 3 klasowej, które kolejno oznaczają przydzielanie obrazów do kategorii COVID - pozostałe oraz COVID - choroby płuc - pozostałe [3].

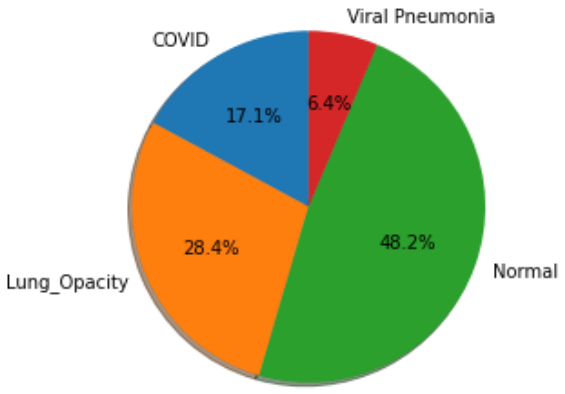
Tabela 1. Zestawienie sieci badanych w kontekście rozpoznawania COVID-19

Sieć	Zadanie	Dokładność [%]	Architektura
COVID-Net	2 klasy	93,3	warstwy konwolucyjne z różnymi rozmiarami kernela (7x7 do 1x1)
	3 klasy		
Faster-RCNN	2 klasy	97,36	oparta na VGG-16
VGG-19	2 klasy	98	uczenie transferowe, warstwy konwolucyjne 3x3
	3 klasy	93	
COVIDX-Net	2 klasy	90	VGG9, DenseNet201, ResNetV2, InceptionV3, InceptionResNetV2, Xception i MobileNetV2
Dark CovidNet	2 klasy	98,08	oparta na DarkNet - open sourcowa architektura używająca CUDA
	3 klasy	87,02	
Insta CovNet-19	2 klasy	99,53	Xception (71 warstwy - 36 konwolucyjne 3x3), ResNet101, MobileNet, InceptionV3 i NasNet
	3 klasy	99,08	
COVID-ResNet	2 klasy	96,23	oparta na ResNet50 - 48 warstw konwolucyjnych, 1 max pool, 1 average pool
	3 klasy		

4. Materiały i metody

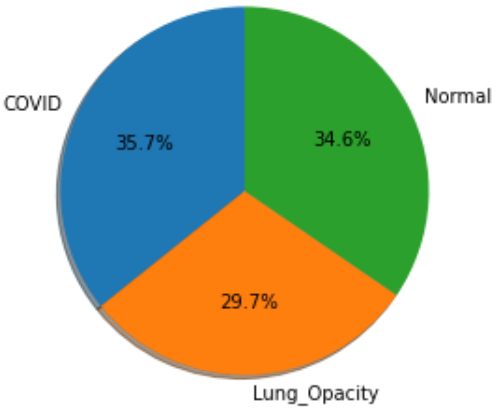
4.1. Zbiór danych

Wybrany zestaw danych składał się z około 20 000 prześwietleń klatki piersiowej. Wszystkie obrazy miały równe rozmiary i liczbę kanałów. Zdjęcia pochodziły od pacjentów z czterech różnych klas: Viral Pneumonia, COVID, Lung Opacity oraz Normal. Niestety rozkład liczebności klas był bardzo nierównomierny co pokazano na rysunku poniżej (Rysunek 4.).



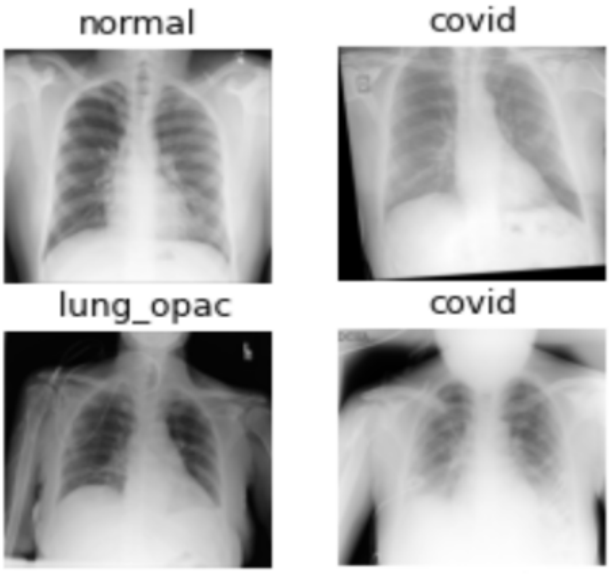
Rysunek 4. Rozkład danych w klasach przed przygotowaniem danych.

Z powodu różnej ilości zdjęć w poszczególnych grupach zdecydowano się zastosować augmentację danych, zwiększając dwukrotnie liczebność klasy COVID (z ok. 3600 na ok. 7200 zdjęć). Augmentacja polegała na zastosowaniu transformacji afinicznej, na którą składała się rotacja o kąt losowany z zakresu od -7 do 7 stopni z wyłączeniem zera oraz ścinanie obrazu przez dodanie niewielkiego współczynnika również losowanego dla każdego zdjęcia oddzielnie. Oprócz zwiększenia liczebności klasy COVID zdecydowano się zrezygnować z wykorzystania klasy Viral Pneuomonia z powodu zbyt małej ilości obiektów oraz części losowo wybranych obrazów z klasy Normal. Otrzymany w ten sposób rozkład ilości obiektów w klasach był zdecydowanie bardziej wyrównany. Każda klasa stanowiła w przybliżeniu 30% całego zbioru. Poniżej przedstawiono wykres prezentujący rozkład po wyrównaniu liczebności klas (Rysunek 5.) [4].



Rysunek 5. Rozkład danych w klasach po augmentacji danych.

Na kolejnym rysunku (Rysunek 6.) przedstawiono przykładowe obrazy z poszczególnych klas. Dla klasy COVID pokazano dodatkowo zdjęcie po przeprowadzonej augmentacji (górny rząd).



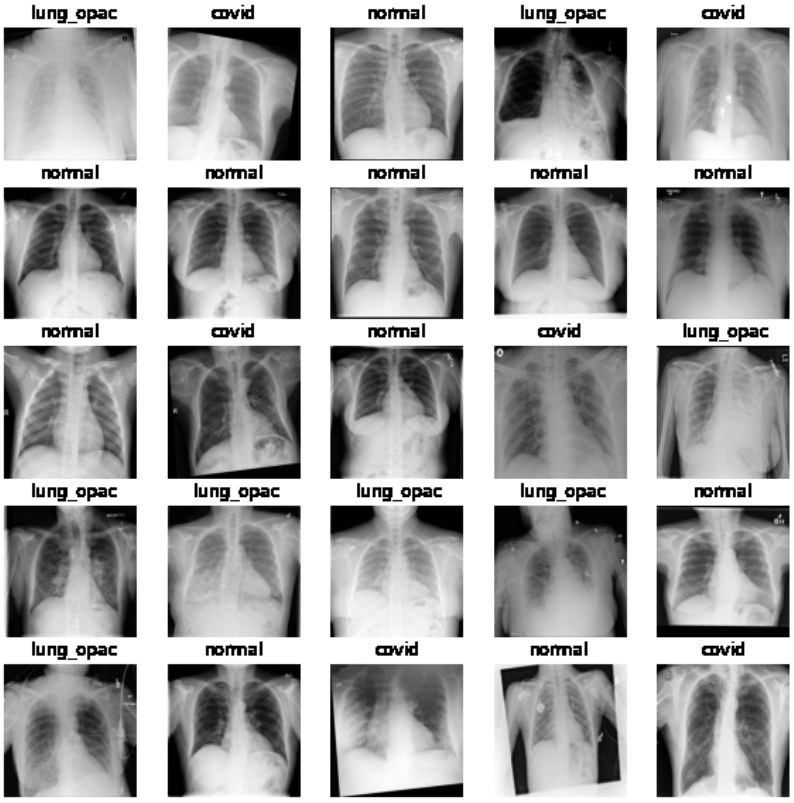
Rysunek 6. Przykładowe obrazy dla poszczególnych klas, z dodatkowym obrazem dla klasy COVID prezentującym, jak wyglądają dane po transformacji afinicznej.

4.2. Metody

Podczas projektu zastosowano dwie sieci konwolucyjne, wykorzystujące dwie różne biblioteki dostępne w języku Python. Pierwsza sieć opiera się na zastosowaniu biblioteki PyTorch oraz propozycji rozwiązania dostępnego w Internecie, jakim jest EfficientNet. Druga sieć została zaprojektowana samodzielnie oraz opierała się na bibliotece TensorFlow. W dalszej części każda z nich zostanie szczegółowo omówiona wraz z procesem jej powstawania oraz przedstawieniem architektury i osiągniętych wyników.

4.2.1. Implementacja rozwiązania wykorzystującego bibliotekę PyTorch

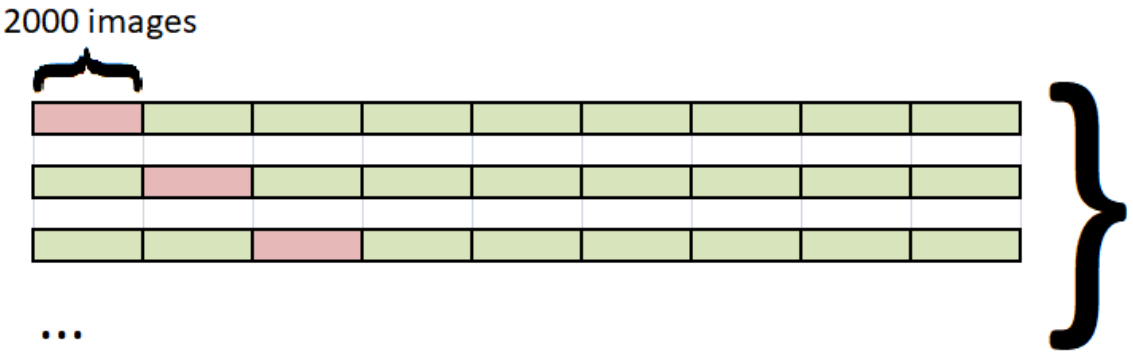
PyTorch jest otwartoźródłową biblioteką języka Python stworzoną przez oddział sztucznej inteligencji Facebooka [5]. Została ona użyta w rozwiązaniu problemu będącego tematem projektu. Ostateczny skrypt w Pythonie, którego użyto, jest dodany jako załącznik do niniejszego projektu. Po zaimportowaniu odpowiednich bibliotek i połączeniu się z dyskiem Google zaimplementowano fragment kodu, który odpowiadał za stworzenie listy ścieżek do plików oraz funkcji, która na ich podstawie określa klasę danego zdjęcia. Następnie zaprezentowano fragment danych, który stanowiło 25 zdjęć wraz z opisem przynależności do klas. Zdjęcia te zaprezentowano na rysunku poniżej.



Rysunek 7. Efekt wstępnej prezentacji dostępnych danych.

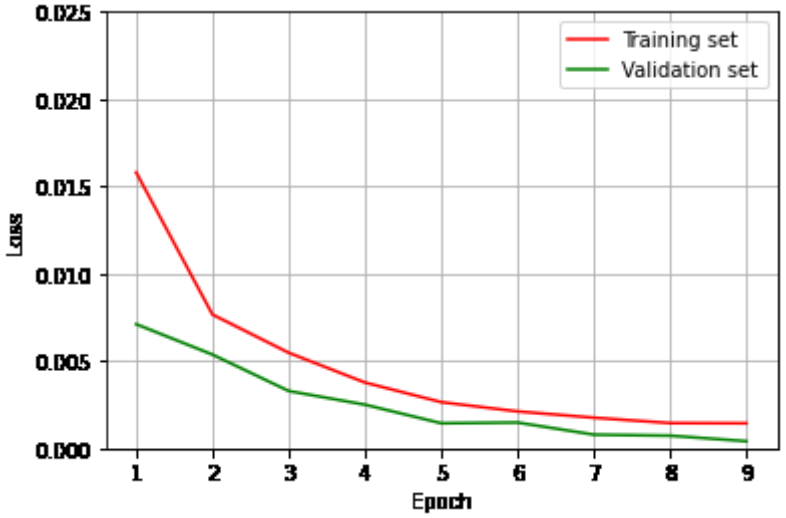
Następnie zaimplementowano klasę LungsRtgDataset, która służyła do operowania danymi w czasie procesu uczenia oraz ewaluacji. Na jej podstawie stworzono odpowiednie tzw. „DataLoadery". Istniała potrzeba zdefiniowania transformacji danych przed rozpoczęciem procesu uczenia. Zastosowano w tym celu normalizację oraz funkcję, która powtarzała obraz na trzy kanały, ponieważ wymagała tego użyta finalnie sieć EfficientNet [6]. Z całego zbioru 20 244 obrazów RTG (zbiór opisany wcześniej, po augmentacji danych) wydzielono części treningowe, walidacyjne oraz testowe. Do części treningowo-walidacyjnej

użyto łącznie 18 tysięcy zdjęć, natomiast do części testowej 2025. Część treningowo-walidacyjna została podzielona w sposób zaprezentowany na rysunku 8.

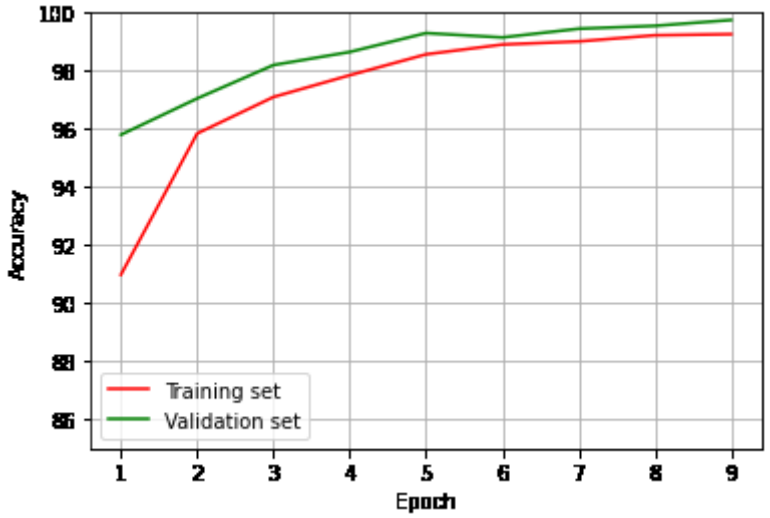


Rysunek 8. Podział danych na część treningową i walidacyjną.

18 tysięcy zdjęć zostało podzielone na 9 równolicznych (po 2000 zdjęć) zbiorów. Ponieważ wykorzystano walidację krzyżową, w każdej epoce uczenia zbiór walidacyjny stanowiła inna część zbioru treningowo-walidacyjnego. Część ta jest reprezentowana przez kolor czerwony na rysunku 8. Pozostałą część zbioru stanowiły dane ze zbioru treningowego. Po przygotowaniu odpowiednich "loaderów" dla wszystkich zbiorów, zdefiniowano architekturę oraz hiperparametry sieci. Po licznych próbach (opisanych poniżej) zdecydowano się wykorzystać wspomnianą już wcześniej sieć EfficientNet, która była już wstępnie przetrenowana. Wybrano optymalizator "Adam", liczbę epok równą 9, funkcję straty entropii krzyżowej oraz learning rate na poziomie 10^(-4). Batch size ustalono na 16. Następnie rozpoczęto proces uczenia. Jego wyniki są zaprezentowane na rysunkach 9 i 10, prezentujących odpowiednio zmianę funkcji straty oraz parametr „accuracy" na przestrzeni poszczególnych epok.



Rysunek 9. Zmiana funkcji straty dla zbioru treningowego oraz walidacyjnego.

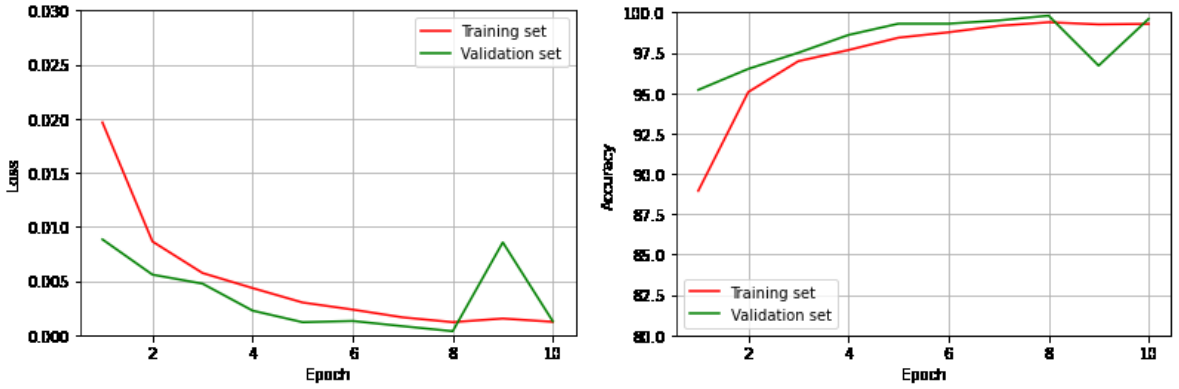


Rysunek 10. Zmiana parametru ,,accuracy” dla zbioru treningowego oraz walidacyjnego.

Otrzymane wykresy okazały się obiecujące, można wnioskować, że nie nastąpiło przeuczenie sieci. Po przeprowadzeniu procesu uczenia wyznaczono dokładność dla zbioru testowego, która wyniosła 97,04%, co okazało się satysfakcjonującym wynikiem. Dla zbioru testowego wyznaczono również macierz pomyłek, a następnie przekazano sieć do dalszej ewaluacji.

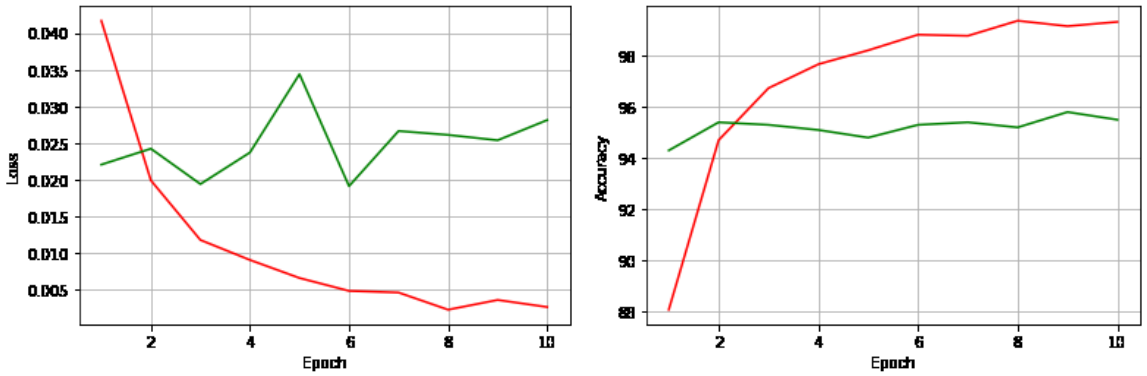
Zanim zdecydowano się na użyte wyżej rozwiązanie, sprawdzano wyniki klasyfikacji w przypadku użycia innych sieci, innej ilości danych lub użycia sieci z innymi hiperparametrami.

Dla przykładu, w przypadku użycia takiej samej sieci oraz 10 tysięcy zdjęć w zbiorze treningowym dokładność wyniosła 95,34%, a wyniki uczenia przedstawiały się jak przedstawiono na rysunku 11. W związku z tym zwiększenie ilości danych poprawiło uzyskane wyniki.



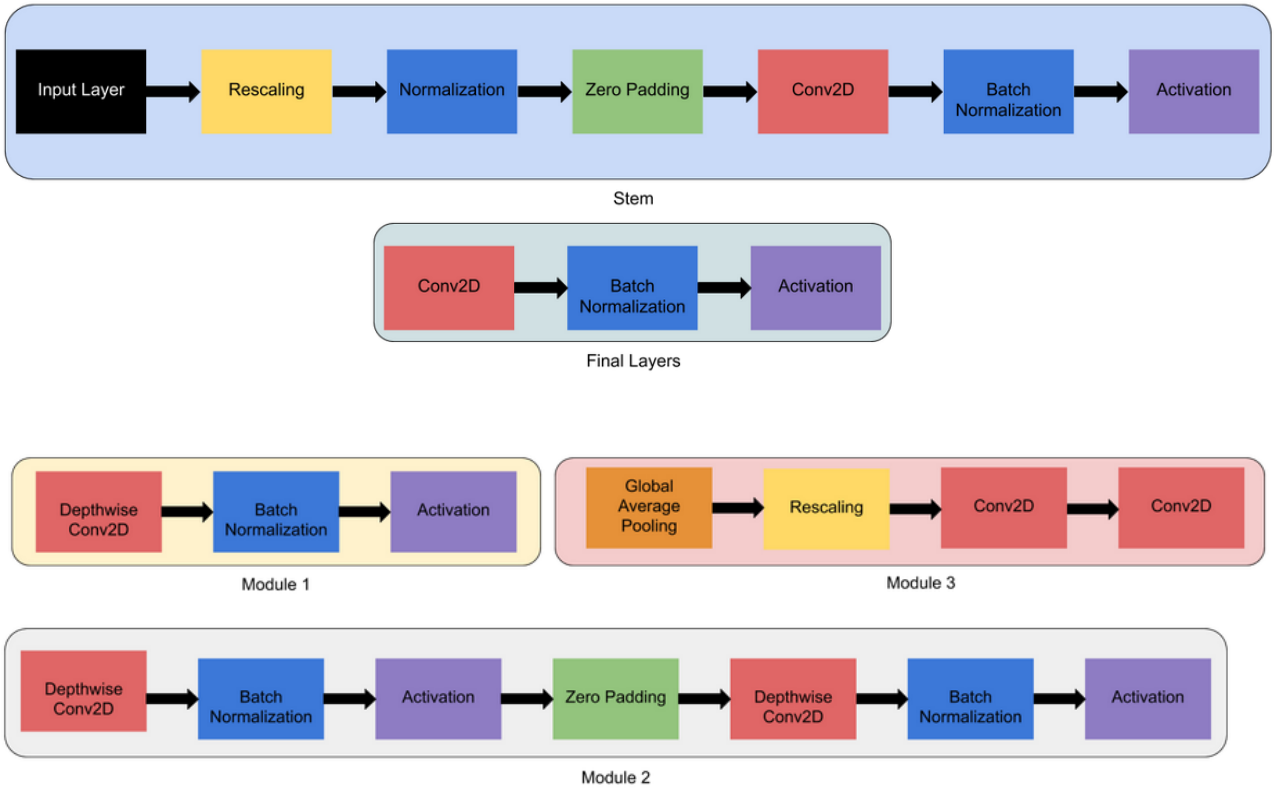
Rysunek 11. Dokładność oraz wartość funkcji straty przy 10000 zdjęć w zbiorze treningowym.

Innym przykładem może być zastosowanie sieci z tymi samymi parametrami, również dla 10 000 tysięcy zdjęć, ale wielkości batcha równej 8. Na podstawie wyników procesu uczenia, przedstawionego na rysunku 12. można wnioskować, iż już na samym początku wystąpił tzw. ,,overfitting”. Udało się go uniknąć poprzez zwiększenie rozmiaru batcha do 16.

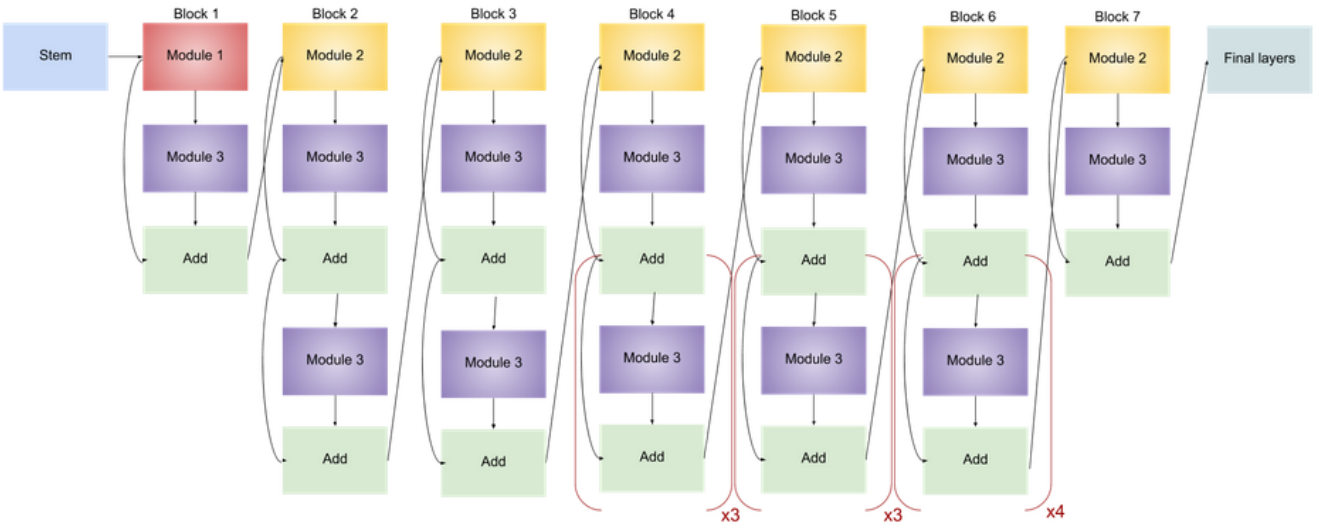


Rysunek 12. Dokładność oraz wartość funkcji straty przy 10000 zdjęć w zbiorze treningowym oraz wielkości batch_size wynoszącej 8.

Na początku próbowano też użyć sieci o dużo prostszej prostszej architekturze, jednak uzyskiwane wtedy wyniki nie były satysfakcjonujące, dlatego zdecydowano się użyć gotowej sieci EfficientNet (wersja b3), której architektura jest schematycznie przedstawiona na rysunku 14. Sieć składa się z modułów zaprezentowanych na rysunku 13.



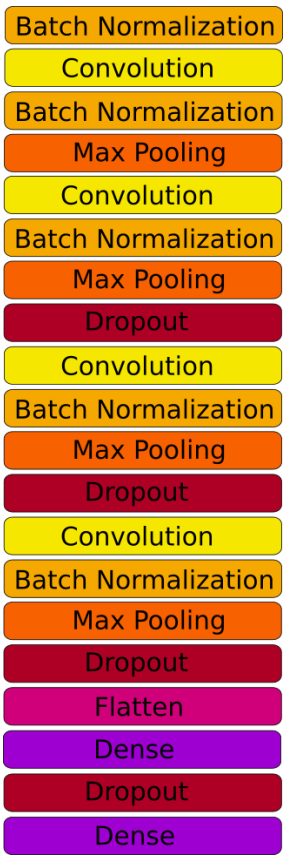
Rysunek 13. Moduły sieci EfficientNet b3 [7].



Rysunek 14. Architektura sieci EfficientNet b3 [7].

4.2.2. Implementacja rozwiązania wykorzystującego bibliotekę TensorFlow

Podczas pracy nad siecią opartą na bibliotece TensorFlow największym problemem okazało się przeuczanie sieci. Aby sobie z nim poradzić stosowano różne podejścia standardowo używane w takiej sytuacji. Pierwszym, najczęściej stosowanym rozwiązaniem jest przeprowadzenie augmentacji danych, jednak w tym przypadku nie przyniosła on oczekiwanych efektów. Następnym krokiem było sprawdzenie, czy architektura sieci nie była zbyt skomplikowana lub wręcz przeciwnie, za mało złożona. W ten sposób wybrano optymalne rozwiązanie. Dodatkowo do architektury sieci dodano warstwy dropout, co częściowo poprawiło wyniki, jednak nie było to wciąż zadowalające. Ostatecznym sposobem rozwiązania problemu przeuczania było zastosowanie warstw normalizacyjnych po każdej warstwie konwolucyjnej. To podejścia okazało się najbardziej skuteczne, a po złożeniu wszystkich metod pozbycia się nadmiernego dopasowywania modelu do danych treningowych, otrzymano zadowalającą dokładność klasyfikacji dla zbioru walidacyjnego. Finalnie zastosowana architektura została przedstawiona na następnym rysunku (Rysunek 15.) Jak widać składa się ona z powtarzanych warstw konwolucyjnych, normalizacyjnych, max pooling, dropout, jednej warstwy flatten oraz warstw gęstych.



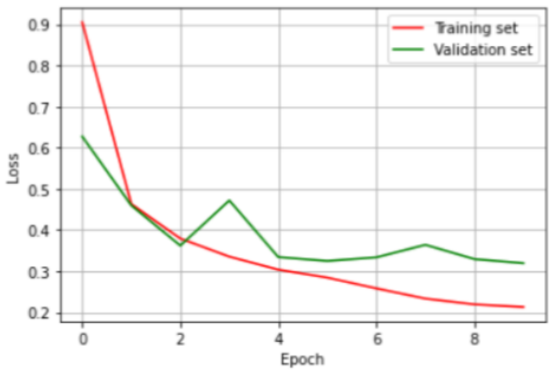
Rysunek 15. Ostateczna architektura sieci opartej na bibliotece TensorFlow.

Kolejny rysunek obrazuje w jaki sposób wyglądała implementacja omawianej architektury, wraz z wszystkimi zastosowanymi parametrami i funkcjami aktywacji.

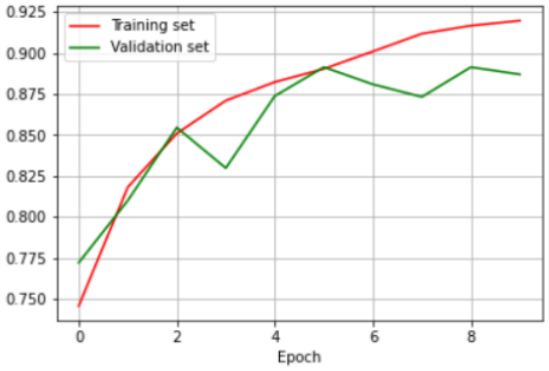
```
def modelCreator():
  model = tf.keras.models.Sequential()
  model.add(layers.BatchNormalization())
  model.add(layers.Conv2D(32, (3,3), activation='relu'))
  model.add(layers.BatchNormalization())
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(64, (3,3), activation='relu'))
  model.add(layers.BatchNormalization())
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Dropout(0.2))
  model.add(layers.Conv2D(32, (3,3), activation='relu'))
  model.add(layers.BatchNormalization())
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Dropout(0.2))
  model.add(layers.Conv2D(128, (3,3), activation='relu'))
  model.add(layers.BatchNormalization())
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Dropout(0.2))
  model.add(layers.Flatten())
  model.add(layers.Dense(256, activation='relu'))
  model.add(layers.Dropout(0.15))
  model.add(layers.Dense(3, activation='softmax'))
  return model
```

Rysunek 16. Sposób implementacji ostatecznej architektury sieci wraz z wszystkimi parametrami i zastosowanymi funkcjami aktywacji.

Podczas pracy wykorzystano optymalizator Adam, jako funkcję straty wybrano wbudowaną funkcję ‘categorical_crossentropy’ oraz przyjęto 10 epok. Otrzymane wykresy funkcji straty oraz dokładności przedstawiono na następnym rysunku (Rysunek 17.).



Validation loss: 0.2983141243457794



Validation accuracy: 0.9051851630210876

Rysunek 17. Wykresy funkcji straty i dokładności w poszczególnych epokach dla zbiorów treningowego i walidacyjnego.

5. Ewaluacja wyników

Po przeprowadzeniu uczenia sieci dokonano oceny ich wyników za pomocą: raportów klasyfikacyjnych, macierzy pomyłek oraz krzywych ROC.

5.1 Raporty klasyfikacyjne

Raporty klasyfikacyjne zawierają takie informacje jak:

- precision (precyzja) - stosunek poprawnie rozpoznanych elementów z klasy A (TP) do wszystkich, które klasyfikator oznaczył jako A (TP+FP),
- recall (czułość) - stosunek poprawnie rozpoznanych elementów z klasy A (TP) do wszystkich, które powinny być rozpoznane jako A, czyli do całej klasy A (TP+FN),
- F1-score - średnia harmoniczna precyzji i czułości
- support - liczba elementów z klasy “wspierających” wyliczenie danego parametru.

W raporcie klasyfikacyjnym dla implementacji rozwiązania wykorzystującego bibliotekę PyTorch (Rysunek 18) klasy zostały przedstawione jako numery:

- 0 - normal
- 1 - lung_opacity
- 2 - covid

	precision	recall	f1-score	support
0	0.96	0.97	0.97	684
1	0.98	0.96	0.97	620
2	0.97	0.98	0.98	721
accuracy			0.97	2025
macro avg	0.97	0.97	0.97	2025
weighted avg	0.97	0.97	0.97	2025

Rysunek 18. Raport klasyfikacyjny (implementacja rozwiązania wykorzystującego bibliotekę PyTorch)

W raporcie klasyfikacyjnym dla implementacji rozwiązania wykorzystującego bibliotekę TensorFlow (Rysunek 18) klasy zostały przedstawione jako numery:

- 0 - covid
- 1 - lung_opacity
- 2 - normal

	precision	recall	f1-score	support
0	0.97	0.93	0.95	724
1	0.84	0.89	0.86	601
2	0.90	0.89	0.90	700
accuracy			0.91	2025
macro avg	0.90	0.90	0.90	2025
weighted avg	0.91	0.91	0.91	2025

Rysunek 19. Raport klasyfikacyjny (implementacja rozwiązania wykorzystującego bibliotekę TensorFlow)

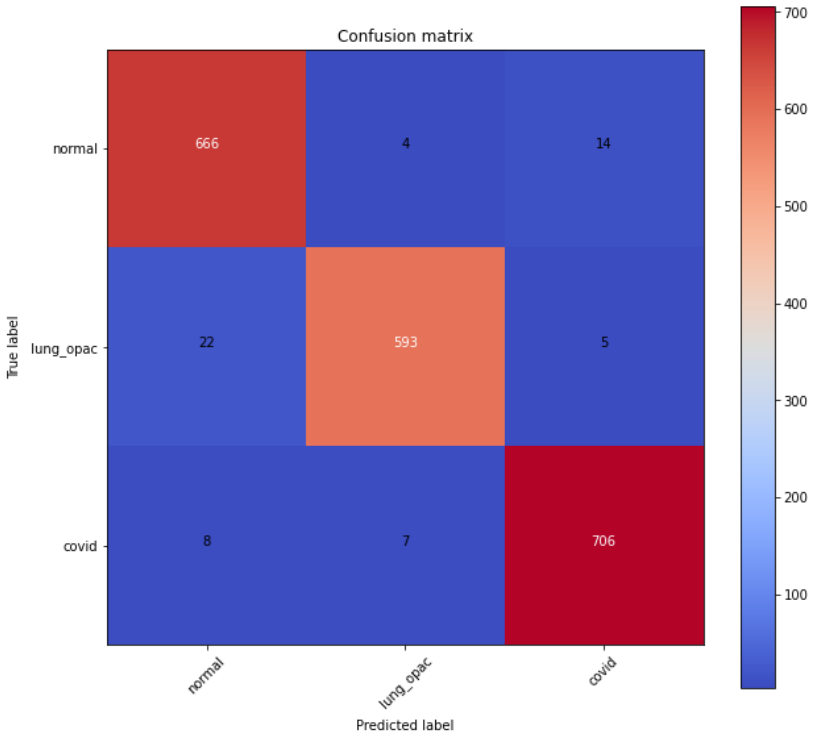
Jak można zauważyć dokładność (accuracy) wyliczona dla zbioru testowego wynosi 97% dla implementacji w PyTorchu oraz 91% dla implementacji w TensorFlow. Przez dokładność rozumiemy liczbę wszystkich poprawnych przewidywań podzieloną przez całkowitą liczbę wykonanych predykcji. W obu przypadkach do procesu ewaluacji użyto 2025 zdjęć. Wszystkie z wyliczonych metryk przedstawionych na raporcie klasyfikacyjnym zarówno dla sieci PyTorch jak i sieci TensorFlow mają wysokie wartości i oscylują około 90%.

5.2 Macierze pomyłek

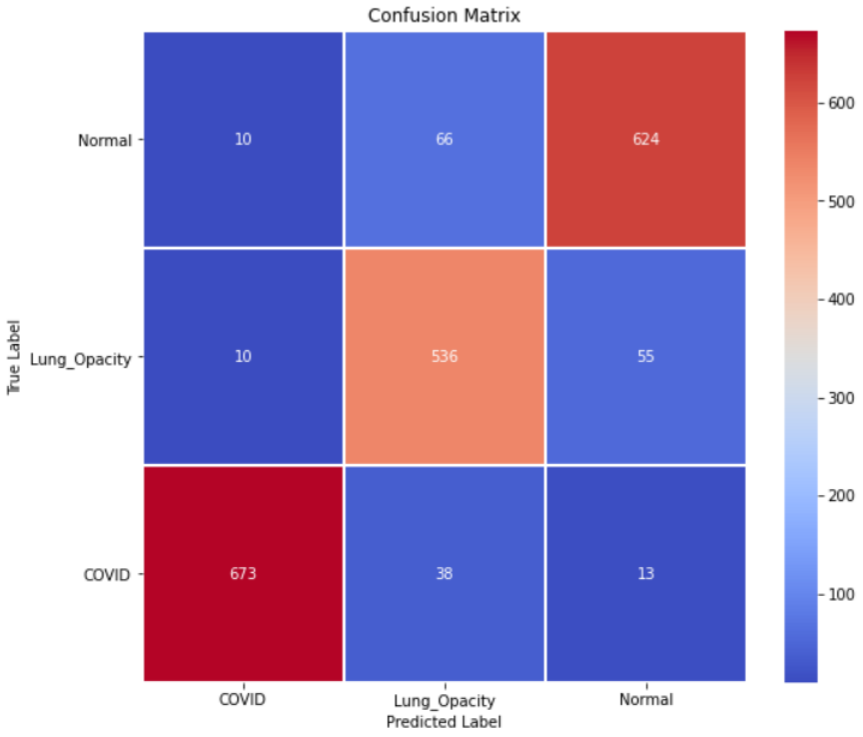
Następnie wykonano macierze pomyłek (Rysunek x oraz Rysunek x). Macierze są swoim lustrzanym odbiciem - wynika to z innej kolejności podpisania nazw klas na osi poziomej dla obu sieci (Predicted label). Na przekątnych pojawiają się najwyższe wartości i są one zaznaczone na czerwono. W pozostałych miejscach macierzy komórki mają tylko i wyłącznie kolor niebieski. Oznacza to, że w zdecydowanej większości elementy zostały poprawnie zaklasyfikowane w przypadku obu sieci. Po intensywności koloru czerwonego możemy wnioskować, że obie sieci najlepiej poradziły sobie z klasyfikacją klasy covid, następnie normal i lung_opacity.

Dokładność, którą przedstawiono w punkcie 5.1 można objaśnić na podstawie macierzy pomyłek- jest to suma wartości na przekątnej do liczby wszystkich dokonywanych predykcji. Przykładowo dla sieci w PyTorch:

$$\text{accuracy}=\frac{666+593+706}{2025}=97\ \%$$



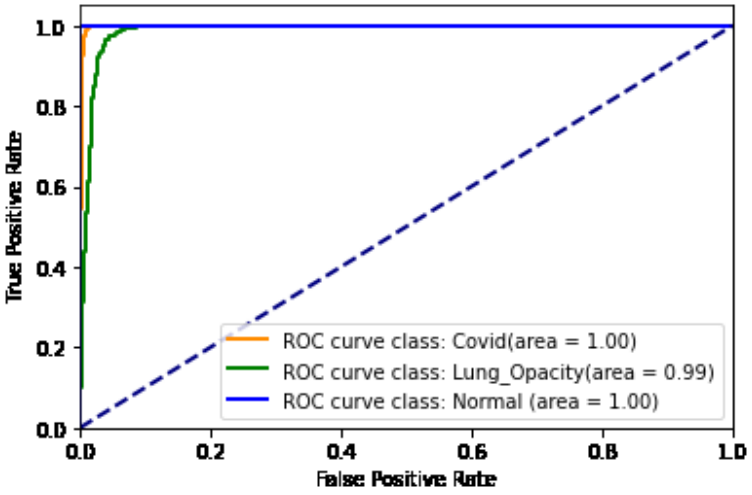
Rysunek 20. Macierz pomyłek (implementacja rozwiązania wykorzystującego bibliotekę PyTorch)



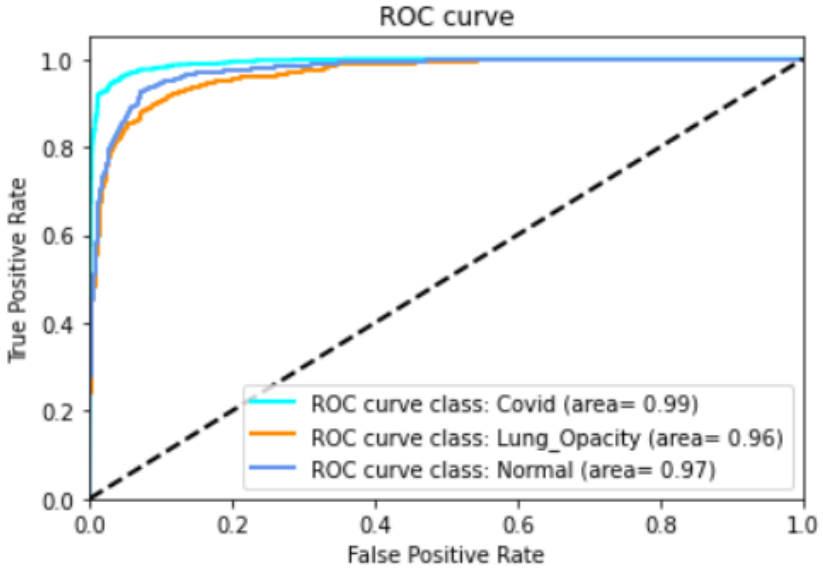
Rysunek 21. Macierz pomyłek (implementacja rozwiązania wykorzystującego bibliotekę TensorFlow)

5.3 Krzywe ROC

Krzywą ROC stosuje się głównie w ewaluacji klasyfikatorów binarnych. Posiada współczynniki wyników prawdziwie pozytywnych na osi Y oraz współczynniki wyników fałszywie pozytywnych na osi X. Oznacza to, że lewy górny róg wykresu jest "idealnym" punktem .Krzywą ROC rozszerzono na klasyfikację z wieloma klasami. W przypadku trzech klas utworzono trzy krzywe ROC dla każdej klasy- brano ją wtedy jako klasę pozytywną i zgrupowano pozostałe klasy razem jako klasę negatywną.W przypadku implementacji w bibliotece PyTorch sieć najlepiej klasyfikowała przypadki określane jako Normal i Covid, natomiast w przypadku implementacji w bibliotece TensorFlow przypadki określane jako Covid. Stwierdzono to na podstawie samego kształtu krzywej jak i oceny parametru AUC-pola pod krzywą. Model doskonały ma AUC bliskie 1, co oznacza, że w dokonuje klasyfikacji niemalże bezbłędnie. Model słaby ma AUC bliskie 0, co oznacza gorsze separowanie poszczególnych klas.



Rysunek 22. Krzywe ROC (implementacja rozwiązania wykorzystującego bibliotekę PyTorch)



Rysunek 23. Krzywe ROC (implementacja rozwiązania wykorzystującego bibliotekę TensorFlow)

6. Dyskusja

- Wyniki z sieci utworzonej w TensorFlow wskazują na jej przeuczanie, wciąż możliwa jest więc praca nad ulepszaniem zaproponowanego rozwiązania. Dalsze plany obejmują sprawdzenie działania sieci z użyciem walidacji krzyżowej.
- Czas uczenia sieci stworzonej w bibliotece PyTorch jest bardzo długi, dalsze prace mogą obejmować więc jego zoptymalizowanie.
- Podczas pracy nad projektem napotkano problem różnych implementacji funkcji straty w poszczególnych bibliotekach, co uniemożliwia ich porównanie i zostawia miejsce do pracy nad wspólną metryką pozwalającą na wyłowienie różnic między osiągnięciami sieci.
- Porównując wypracowane wyniki do rozwiązań omawianych w przeglądzie literatury, możemy stwierdzić, że nie odbiegają one znacząco od wyników proponowanych sieci neuronowych. Obie prezentowane w projekcie sieci osiągnęły więc zadowalające wyniki.

7. Podsumowanie i wnioski

Sieć stworzona w bibliotece PyTorch charakteryzowała się dużo większą złożonością oraz dużo większą ilością warstw. Czas uczenia tej sieci, w zależności od urządzenia, liczono w godzinach natomiast sieć utworzona w TensorFlow o mniejszej ilości warstw umożliwiła uczenie w czasie kilkunastu minut. Różnice w przypadku strat dla sieci implementowanych w obu bibliotekach okazały się znaczne i mogły wynikać z różnych implementacji funkcji entropii krzyżowej w bibliotece TensorFlow oraz PyTorch. Podsumowując, wyniki z obu sieci można uznać za zadowalające mimo znacznej różnicy w ich złożoności.

8. Bibliografia

[1] R. Jain, M. Gupta, S. Taneja, and D. J. Hemanth, “Deep learning based detection and analysis of COVID-19 on chest X-ray images,” *Applied Intelligence*, vol. 51, no. 3, pp. 1690–1700, Oct. 2020, doi: 10.1007/s10489-020-01902-1.

[2] E. E.-D. Hemdan, M. A. Shouman, and M. E. Karar, “COVIDX-Net: A Framework of Deep Learning Classifiers to Diagnose COVID-19 in X-Ray Images,” *arXivLabs*, vol. 1, no. 1, 2020.

[3] N. Y. Khanday and S. A. Sofi, “Deep insight: Convolutional neural network and its applications for COVID-19 prognosis,” *Biomedical Signal Processing and Control*, vol. 69, p. 102814, Aug. 2021, doi: 10.1016/j.bspc.2021.102814.

[4] “How to Avoid Overfitting in Deep Learning Neural Networks,” *Machine Learning Mastery*, Dec. 16, 2018. https://machinelearningmastery.com/introduction-to-regularization-to-reduce-overfitting-and-improve-generalization-error/ (accessed Jun. 18, 2021).

[5] W. projektów F. Wikimedia, “PyTorch,” *Wikipedia*. https://pl.wikipedia.org/wiki/PyTorch (accessed Jun. 18, 2021).

[6] “efficientnet-pytorch,” *PyPI*. https://pypi.org/project/efficientnet-pytorch/ (accessed Jun. 18, 2021).

[7] V. Agarwal, “Complete Architectural Details of all EfficientNet Models,” *Towards Data Science*, May 31, 2020.