UNIWERSYTET WSB MERITO W GDAŃSKU
WYDZIAŁ INFORMATYKI I NOWYCH TECHNOLOGII

ANALIZA PORÓWNAWCZA KONWOLUCYJNYCH SIECI NEURONOWYCH INCEPTION RESNET V2 I XCEPTION W ZASTOSOWANIU ICH DO KATEGORYZOWANIA ZMIAN SKÓRNYCH

Praca inżynierska
na kierunku Informatyka

Praca napisana pod kierunkiem
dr inż. Pawła Tomkiewicza

Marek Małek, 56572, Gdańsk 2024

Spis treści

WSTĘP

Celem pracy jest wykonanie analizy porównawczej dwóch, klasycznych już architektur konwolucyjnych sieci neuronowych, którymi są InceptionResNetV2 oraz Xception. Obie te architektury służyć będą za rdzeń kilkunastu wariacji mających na celu osiągnięcie wyników lepszych, niż byłoby to możliwe przy użyciu niezmodyfikowanych sieci i surowych danych. W jej skład wchodzić będą następujące części:

- Krótka, eksploracyjna analiza danych (en. EDA[1]) – jej celem jest sprawdzenie rozkładu klas obrazów
- Wyszukiwanie modelu bazowego – aby wyraźnie zobaczyć przewagę sieci neuronowych w danym zagadnieniu, dobrze jest najpierw zastosować prostsze i szeroko znane techniki oraz sprawdzić jaką one mają wydajność
- Sprawdzenie jak dobrze na zestawie danych HAM10000[2] działa architektura InceptionResNetV2[3], na razie ignorując dysbalans klas; sprawdzenie, czy będzie w stanie pobić model bazowy
- Dostrajanie wybranej architektury tak, aby lepiej radziła sobie na zbiorze danych HAM10000
- Wytrenowanie na tym samym zestawie klas architektury Xception[4] i porównanie jej wyników z InceptionResNetV2

Poza wymienionymi składowymi, wykonane zostaną także skrypty ułatwiające zarządzanie kodem oraz automatyzujące wielokrotne uruchamianie procesów treningowych. Jest to potrzebne, aby móc wyciągnąć poprawne wnioski, ponieważ w szkoleniu każdej sieci neuronowej jest miejsce na element losowości. Stąd dla zrozumienia, która modyfikacja miała pozytywny wpływ na trening potrzebne jest kilkukrotne uruchomienie go oraz uśrednienie wyników, a następnie analiza statystyczna.

W rozdziale pierwszym i drugim została zawarta część teoretyczna pracy. Opisano w nich krótko historię sieci neuronowych, w szczególności splotowych sieci neuronowych (CNN[5]), przybliżono w nich informacje dotyczące technik, które sprawiły że są tak skuteczne

[1] https://en.wikipedia.org/wiki/Exploratory_data_analysis
[2] https://www.kaggle.com/datasets/kmader/skin-cancer-mnist-ham10000
[3] https://arxiv.org/pdf/1602.07261v2.pdf
[4] https://arxiv.org/pdf/1610.02357.pdf
[5] https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html

w rozpoznawaniu obrazów, a także opisano dwie wybrane architektury porównując je w miejscach, w których najmocniej się różnią.

Na początku rozdziału trzeciego opisano proces i skutki trenowania modeli bazowych, które ostatecznie posłużyły jako punkt wyjścia do poszukiwań właściwych architektur neuronowych. Następnie opisano modyfikacje mające usprawnić sieci InceptionResNetV2 oraz Xception i analogicznie do początku rozdziału – opisano proces i skutki trenowania tych modeli.

W rozdziale czwartym przedstawiono proces analizy statystycznej stanowiącej sposób dojścia do opisanych dalej wniosków. Opisano tu też problemy, z którymi spotkano się w trakcie prac.

Rozdział 1 – Historia Sieci Neuronowych

1.1 Wprowadzenie

Większość wytwarzanego oprogramowania od samego początku była algorytmiczna. Program miał zawierać procesy i reguły biznesowe danego przedsiębiorstwa i w ten sposób automatyzować powtarzalne czynności, które można przelać na kod w postaci szeregu instrukcji. Oprogramowanie takie zawsze jest specyficzne, rzadziej dla całej domeny, w której porusza się firma, częściej dla konkretnego jej wycinku, specyfiki działania samego przedsiębiorstwa. Oczywiście istnieją gotowe produkty, takie jak SAP ERP[6], które operują w obrębie całych domen, jednak wciąż ich zasada działania jest taka, jak mniejszych programów – do aplikacji trafiają dane zewnętrzne, które po przetworzeniu przez ściśle określone reguły biznesowe dają spodziewany rezultat. Od zawsze istniała jednak potrzeba istnienia oprogramowania posługującego się logiką rozmytą[7] tak, aby dla pewnych klas problemów nie trzeba było tworzyć tak sztywnych i dokładnych reguł przetwarzania danych. W istocie, być może dla większości tych klas problemów nie byłoby to nawet możliwe. Dla przykładu, w praktyce niemożliwe jest napisanie dobrego klasyfikatora pisma odręcznego.

Różne algorytmy uczenia maszynowego znane były od połowy ubiegłego stulecia, a wciąż krótką historię samych sieci neuronowych podzielić można na kilka okresów:

- Wczesne koncepcje lat 40-tych i 50-tych: wtedy to Warren McCulloch i Walter Pitts tworzą pierwszy „obliczeniowy” model biologicznego neuronu[8].
- Wynalezienie Perceptronu[9] w późnych latach 50-tych: była to wczesna sieć neuronowa zdolna nauczyć się rozpoznawania prostych wzorców, miała jednak spore ograniczenia, np. nie była w stanie rozwiązywać problemów, które nie były liniowo rozdzielne (np. problem XOR[10])
- „Pierwsza Zima Sztucznej Inteligencji” w późnych latach 60-tych i 70-tych: przez ograniczenia perceptronu oraz niewielkiej dostępnej wtedy badaczom mocy obliczeniowej zainteresowanie inwestorów mocno ucierpiały przez co rozwój AI bardzo spowolnił.

[6] https://pl.wikipedia.org/wiki/SAP_ERP
[7] https://towardsdatascience.com/a-very-brief-introduction-to-fuzzy-logic-and-fuzzy-systems-d68d14b3a3b8
[8] https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1
[9] https://maelfabien.github.io/deeplearning/Perceptron/#the-classic-model
[10] https://www.niser.ac.in/~smishra/teach/cs460/2020/lectures/lec19/

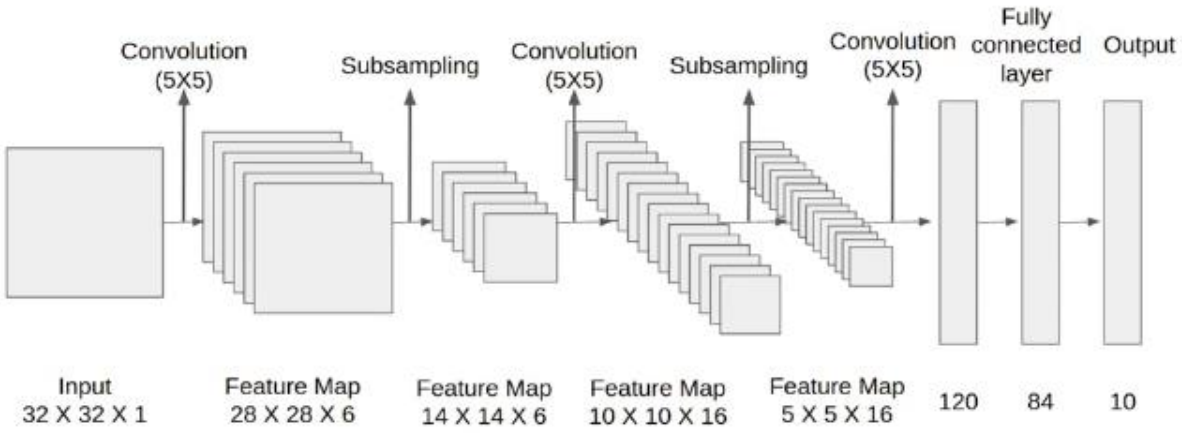
- Wynalezienie algorytmu Propagacji Wstecznej[11]: umożliwiło to sieciom neuronowym uczenie się bardziej skomplikowanych wzorców.
- „Druga Zima Sztucznej Inteligencji" w późnych latach 80-tych i 90-tych: nastała z powodu zbyt dużych oczekiwań i ograniczonej dostępności mocy obliczeniowej. Innymi słowy sprzęt komputerowy wciąż był zbyt słaby, aby sieci neuronowe mogły pokazać swoje prawdziwe możliwości. Wtedy też na popularności zyskały mechanizmy uczenia maszynowego niezwiązane z sieciami neuronowymi, takie jak Support Vector Machines[12]
- Ponowna faza wzrostu zainteresowania AI oraz czas przełomów: od początku lat 2000 ponownie widać było rosnące zainteresowanie sieciami neuronowymi przez badaczy i wielkie firmy technologiczne. W okolicach początku drugiego dziesięciolecia lat dwutysięcznych zaczęły się pojawiać kolejne przełomowe odkrycia i czas ten trwa do teraz. Kilka największych z nich to np. AlexNet[13], która to sieć stanowiła przełom w dziedzinie rozpoznawania obrazów, odkrycie architektur GAN[14] oraz Transformer[15] (która to jest używana np. w święcącym sukcesy ChatGPT).

1.2 Konwolucyjne Sieci Neuronowe (CNN)

Chociaż historia rozpoznawania obrazów przy pomocy sieci neuronowych, przynajmniej na poziomie koncepcji, sięga lat 80-tych, to pierwszym najgłośniejszym zastosowaniem tego typu sieci była architektura LeNet-5, stworzona w późnych latach 90-tych przez obecnie szeroko znanego badacza sztucznej inteligencji – Yanna LeCuna. Celem jej stworzenia było umożliwienie Amerykańskiemu Biuru Pocztowemu automatycznego odczytu kodów pocztowych z kopert. Była to dość prosta (z dzisiejszego punktu widzenia) architektura, której składowe to warstwa wejścia o wymiarach 32x32x1 (gdzie pierwsze dwie liczby to wysokość i szerokość obrazów, a ostatnia to ilość kanałów), warstwa splotowa, warstwa uśredniająca, jeszcze jedna warstwa splotowa, po niej uśredniająca i ostatnia splotowa, po której znajdują się dwie warstwy gęsto połączone oraz warstwa wyjścia, służąca do klasyfikacji liczb. Warstwy splotowe wykrywają wzorce przestrzenne, natomiast warstwy uśredniające zmniejszają rozmiar danych pochodzących z ich poprzedników. Zostało to zilustrowane na Rysunku 1.

[11] https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd
[12] https://pl.wikipedia.org/wiki/Maszyna_wektor%C3%B3w_no%C5%9Bnych
[13] https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
[14] https://arxiv.org/pdf/1406.2661.pdf
[15] https://arxiv.org/pdf/1706.03762.pdf

Rysunek 1 – Architektura Sieci LeNet-5



Źródło: https://www.analyticsvidhya.com/blog/2021/03/the-architecture-of-lenet-5/

Architektura ta stała się punktem wyjścia dla twórców bardziej złożonych sieci neuronowych takich jak AlexNet, VGGNet, czy ResNet.

Być może największym wkładem Yanna LeCuna w rozwój sztucznej inteligencji było zastosowanie mechanizmu tzw. Propagacji Wstecznej. Nie była to całkowicie nowa idea, jednak była świeża w świecie wielowarstwowych modeli konwolucyjnych. Jako, że jest to technika leżąca u podstaw każdego rodzaju sieci, nie tylko CNN, zostanie ona opisana jako pierwsza.

1.3 Mechanizm propagacji wstecznej

Mechanizm ten składa się z kilku kroków:

- Przejście wprzód: warstwa po warstwie, aż do wyjścia przekazywane są dane. W przypadku sieci konwolucyjnej będą to coraz bardziej zmienione – zmniejszone, a na końcu spłaszczone do postaci wektora, dane obrazu.
- Obliczenie straty: funkcja straty (która jest jednym z tzw. hiperparametrów) oblicza różnicę między predykcją modelu, a stanem faktycznym – tym, co chciano by, aby model zwrócił.
- Przejście w tył (propagacja wsteczna): w tym miejscu algorytm oblicza gradienty funkcji straty względem każdej z wag w sieci. Każda z owych wag przyczynia się do uzyskania ostatecznego rezultatu, dlatego też sygnał błędu musi zostać rozpropagowany wstecz.
- Gradienty używane są do aktualizacji wag sieci.

- Cały cykl powtarza się na nowo, w każdym przejściu minimalizując wynik funkcji straty.

1.4 Operacja splotu

Tak jak mechanizm propagacji wstecznej jest kluczowym elementem dla wszystkich rodzajów sieci neuronowych, to operacja splotu jest takim elementem w sieciach konwolucyjnych. Dzięki niej, sieci splotowe są w stanie lepiej, od np. sieci gęsto połączonych wychwytywać złożone wzorce, ale też, do pewnego stopnia są w stanie znajdywać je w różnych miejscach obrazów wejściowych. Składowymi operacji splotu są:

- Filtry (lub kernele): są to małe macierze wag, które podobnie jak wagi warstw gęstych ustawione są początkowo na losowe wartości.
- „Aplikowanie" filtrów na danych wejściowych: w tym momencie każda z wag mnożona jest przez wartość danego piksela. Kiedy mnożenia na obecnej pozycji filtra zostają zakończone, ich wyniki są do siebie dodawane – wynik tych operacji to wynik działania filtra na bieżącym obszarze obrazu.
- „Przesuwanie" filtrów: w tym momencie następuje przesunięcie kerneli o określoną ilość pikseli, aż do „końca" obrazu.
- Wyjściem z tej operacji są tzw. mapy cech, które reprezentują to, co w danym przejściu udało się wykryć warstwie splotowej.
- Na końcu, na każdej mapie cech wywoływana jest nieliniowa funkcja aktywacji (np. ReLU), po to, żeby model był zdolny wychwytywać skomplikowane wzorce.

1.5 Użyte narzędzia i technologie

Jako biblioteki uczenia maszynowego wybrano TensorFlow (w skrócie TF), wraz z często towarzyszącym jej modułowi Keras. Jest to popularny w świecie uczenia maszynowego wybór. TF dostarcza niskopoziomowych funkcji oraz algorytmów (takich jak np. algorytm propagacji wstecznej, czy różniczkowania automatycznego) zaimplementowanych w języku C++, które posiadają tzw. bindingi języka Python – w ten sposób programista może pisać kod, bez konieczności zagłębiania się w trudniejsze aspekty języka C++. Keras jest natomiast frameworkiem, który implementuje wysokopoziomowe mechanizmy potrzebne, aby szybko i w czytelny sposób tworzyć sieci neuronowe. Mechanizmami tymi są m.in. różne rodzaje warstw, jak np. warstwa Dense, Conv2D, czy GlobalAveragePooling. Udostępnia on również bibliotekę wytrenowanych modeli używanych podczas tzw. transfer learning, a także dostarcza narzędzia do preprocessingu danych oraz ich augmentacji.

Treningi modeli odbywały się na komputerze desktopowym wyposażonym w 64gb pamięci RAM, kartę graficzną z rodziny NVidia GeForce RTX 3600, oraz procesor Intel Core i5-10400F o taktowaniu 2.9GHz. Ilość pamięci RAM, oraz moc procesora były istotne podczas trenowania modeli biblioteki sklearn, natomiast karta graficzna używana była podczas uczenia sieci neuronowych.

Rozdział 2 – Architektury InceptionResNetV2 i Xception

2.1 Architektura InceptionResNetV2

Największą zmianą wprowadzoną przez autorów tej architektury w stosunku do jej poprzedników jest użycie modułów incepcyjnych oraz połączeń skrótowych (residual connections). Obie te techniki istniały już wcześniej, jednak użycie ich kombinacji umożliwiło utworzenie dużo głębszych struktur, które oprócz cechowania się wyższą dokładnością, mniej cierpiały z powodu problemu tzw. zanikającego gradientu.

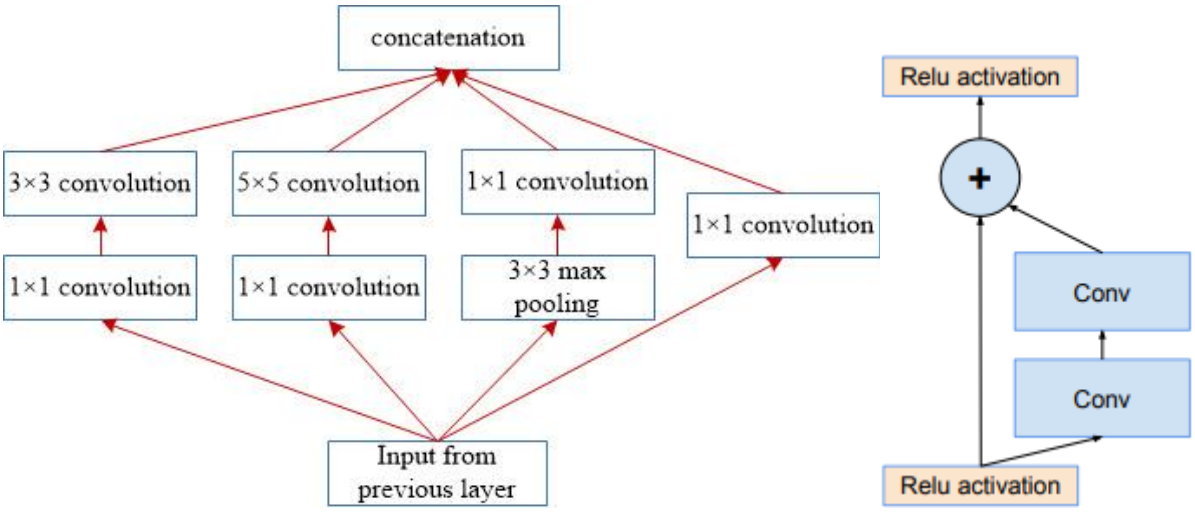
Problem ten został zauważony przez badaczy, kiedy próbowali uczyć bardzo głębokie sieci. Opisany wcześniej mechanizm propagacji wstecznej może cechować się tym, że wagi warstw położonych najdalej są w procesie treningu dostosowywane w najmniejszym stopniu, co z kolei skutkuje tym, że warstwy te przestają się uczyć i sieć przykłada nieproporcjonalnie wyższą uwagę to położonych wyżej. To z kolei sprawia, że stosowanie głębokich sieci neuronowych może nie przynieść żadnych korzyści, lub wręcz sprawić, że ich dokładność będzie niższa od ich prostszych rodzajów. W poszukiwaniu architektur zdolnych rozpoznawać coraz bardziej skomplikowane wzorce, narodziła się konieczność zapobieżenia temu zjawisku. Połączenia skrótowe (pomijające) działają tak, że warstwa poprzedzająca warstwę bezpośrednio nad nią, zwraca swój rezultat do warstwy położonej jeszcze wyżej – oba wyniki są dodawane i w ten sposób mechanizm propagacji wstecznej może w bardziej wydajny sposób rozpropagować gradient funkcji straty.

Jeśli zaś chodzi o ideę, która stała za zastosowaniem modułów incepcyjnych, to badaczom chodziło o to, aby móc wykrywać skomplikowane wzorce w różnych skalach. Stąd, moduły incepcyjne przeprowadzają operację splotu używając jednocześnie filtrów o różnych wielkościach (np. 5x5, 3x3).

Na rysunku 2 zaprezentowane zostały oba omawiane moduły. Jest to bardzo podstawowa prezentacja tej idei, a w jej konkretnych implementacjach występują różne bloki incepcyjne oraz pomijające, bardziej rozbudowane od poniższych.

Po lewej stronie widać, że blok incepcyjny rozgałęzia się na kilka „ramion” przechodzących przez dane filtrami różnych wielkości. Na końcu wyniki tych równoległych operacji łączone są do postaci jednej mapy cech.

Rysunek 2 - przykład modułu incepcyjnego



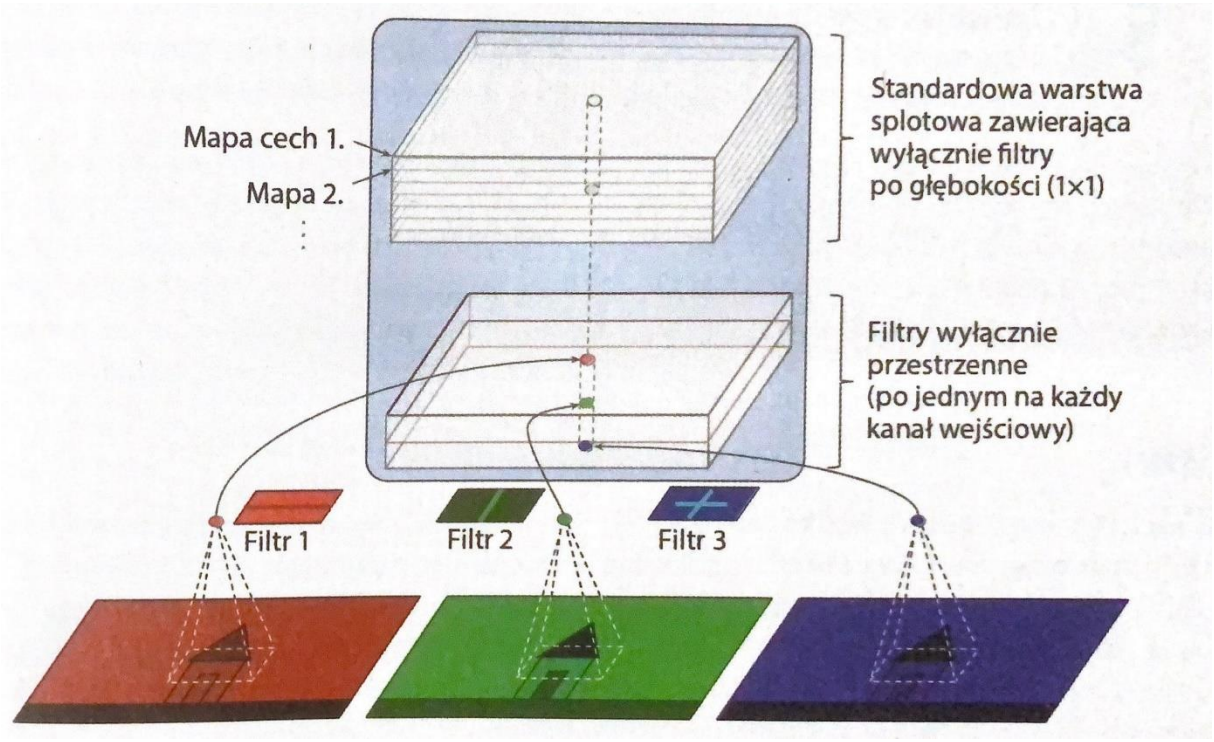
Źródło: https://www.researchgate.net/figure/Basic-architecture-of-inception-block_fig4_330511306
Źródło: https://arxiv.org/pdf/1602.07261v2.pdf? https://arxiv.org/pdf/1602.07261v2.pdf?

Po prawej stronie widać schemat bloku zawierającego połączenie pomijające. W tym przykładzie, dla uproszczenia użyto dwóch warstw splotowych, jednak nic nie stoi na przeszkodzie, aby w ich miejscu pojawił się blok incepcyjny taki, jak na schemacie z lewej strony, co też z resztą ma miejsce w kodzie modelu InceptionResNetV2.

2.2 Architektura Xception

Architektura Xception została zaproponowana jako wariacja podejścia prezentowanego przez Inception, z tą różnicą, że bloki incepcyjne zastąpione zostały rozdzielnymi po głębokości warstwami splotowymi. Klasyczna sieć splotowa wykorzystuje filtry starające się jednocześnie wykrywać wzorce przestrzenne i międzykanałowe, natomiast w rozdzielnej warstwie splotowej użyto założenia, że wzorce przestrzenne i międzykanałowe mogą być modelowane oddzielnie. Między tą architekturą a kolejnymi podejściami do pomysłów zawartych w linii modeli Inception istnieje też sporo innych różnic, jednak ta jest kluczowa i największa. Dzięki użyciu operacji splotu w opisany powyżej, alternatywny sposób, Xception uznawane jest za generalnie skuteczniejszą architekturę. Rysunek 3 zawiera diagram kluczowego elementu:

Rysunek 3- wykorzystanie filtrów po głębokości



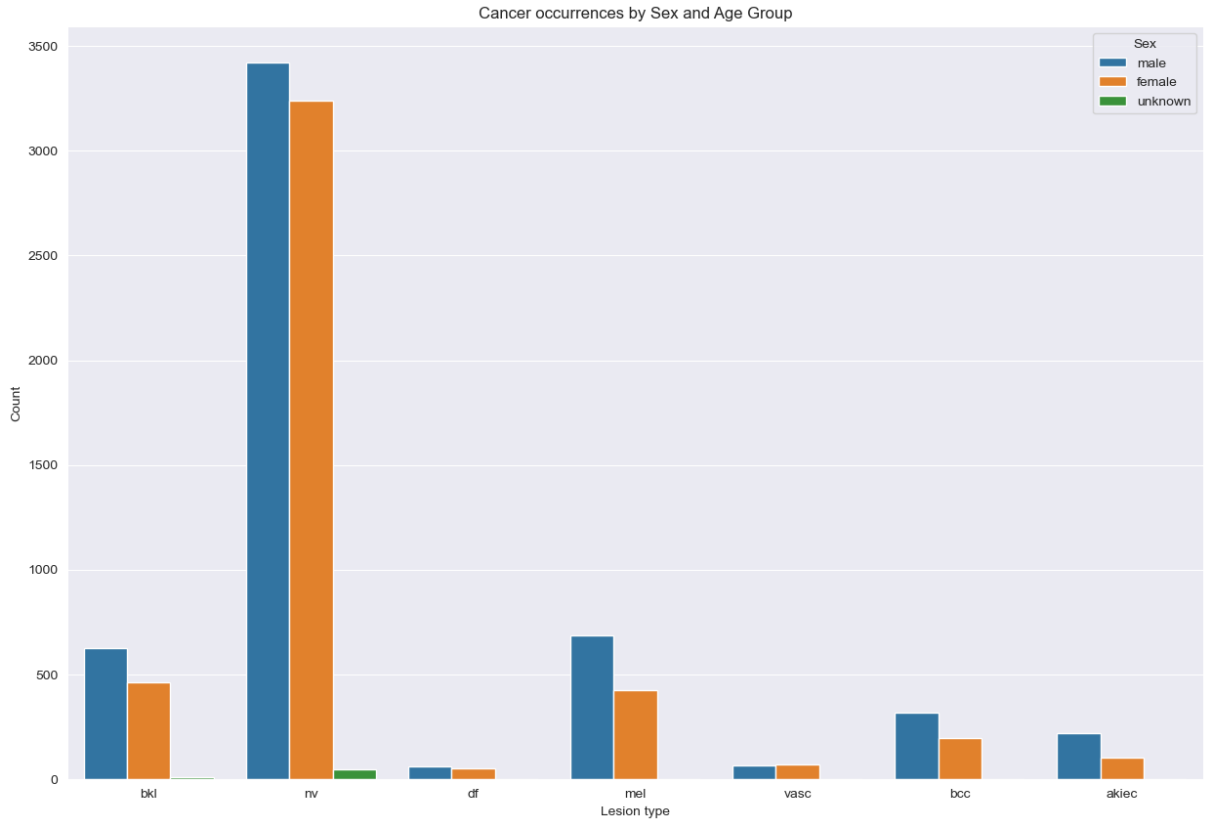
Źródło: Uczenie maszynowe z użyciem Scikit-Learn i TensorFlow

Rozdział 3 – Proces Trenowania Modeli

3.1 Zbiór danych i EDA

Jako zbiór danych wybrano dostępny w portalu kaggle.com HAM10000. Zawiera on 10 000 obrazów zmian skórnych dla 7 kategorii. Oprócz samych zdjęć, obecne są także pewne informacje o pacjentach, takie jak wiek, płeć, umiejscowienie zmiany i metoda jej wykrycia. Ostatecznie nie zdecydowano się na ich użycie, aby lepiej skupić się na rdzeniu rozpatrywanego problemu, czyli rozpoznawaniu obrazów. Rysunek 4 prezentuje histogramy zliczające wystąpienia określonych zmian skórnych wśród płci, gdzie na niebiesko zaznaczono mężczyzn, a na pomarańczowo kobiety:

Rysunek 4 - EDA



Źródło: opracowanie własne

Ciekawą obserwację być może stanowi fakt, że dla większości zmian skórnych obecnych w zbiorze HAM 10000 ich ofiarami nieco częściej padają mężczyźni. Nie ma to jednak wpływu na efektywność treningu, wybranych architektur.

Z rysunku 4 płynie jednak wniosek – występuje tu nierównowaga ilości próbek pomiędzy kolejnymi zmianami skórnymi, co będzie miało wpływ na kroki podjęte w celu usprawnienia dokładności klasyfikatorów.

3.2 Trenowanie i rezultaty modeli bazowych

Przez modele bazowe, w kontekście tej pracy należy rozumieć modele niewykorzystujące sieci neuronowych. Wybrano to podejście, aby unaocznić, że prostsze algorytmy uczenia maszynowego nie są skuteczne w obliczu zbioru danych tak dużego i skomplikowanego jak HAM 10000. Chociaż były w stanie osiągnąć zbieżność szybciej od sieci neuronowych, to ich dokładność nigdy nie przekroczyła 41%.

Pierwszy z wybranych algorytmów to KNeighborsClassifier, obecny w pythonowej bibliotece scikit-learn (w skrócie sklearn). W świecie data science jest on dość popularny i często wybierany jest jako klasyfikator nawet dla bardziej złożonych problemów – przez swoją skuteczność i prostotę. Rysunek 5 pokazuje punkt wyjściowy, czyli trening tego klasyfikatora bez żadnych modyfikacji:

Rysunek 5 - pierwsze podejście do treningu KNN

```
neighbors = np.max(y_train)
knn = KNeighborsClassifier(n_neighbors=neighbors)
knn.fit(X_train, y_train)

y_pred = knn.predict(X_test)

print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
```

```
Accuracy: 0.408914728682170555
```

Źródło: opracowanie własne

Dokładność lekko poniżej 41% była niezadawalająca, dlatego też spróbowano usprawnić ten wynik dodając do procesu treningu przeszukiwanie siatki. Polega ono na tym, że interfejs sklearn pozwala użytkownikowi na stworzenie tzw. pipeline, w którym to kolejne kroki uruchamiane są z różnymi kombinacjami parametrów, w ten sposób próbując przetestować każdą taką kombinację i na końcu wybierając najlepszą z nich. Najpopularniejszym podejściem używanym przy treningu modeli dostarczanych przez sklearn jest użycie PCA. Jest to technika redukcji wymiarowości, która powinna nieco uprościć (lub odszumić) dane, dekorelując je i zmniejszając liczbę ich wymiarów. Rysunek 6 pokazuje skutki treningu z użyciem PCA:

Rysunek 6 - trenowanie KNN z pomocą PCA

```
pipe = Pipeline([
    ('reduce_dim', 'passthrough'),
    ('classify', KNeighborsClassifier(n_neighbors=neighbors))
])
param_grid = [
    {
        'reduce_dim': [PCA()],
        'reduce_dim__n_components': [2, 5, 10, 20, 30],
        'reduce_dim__whiten': [True, False]
    }
]
grid = GridSearchCV(pipe, cv=5, param_grid=param_grid, verbose=2)
grid.fit(X_train, y_train)

best_model = grid.best_estimator_
y_pred = best_model.predict(X_test)

print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
print(grid.best_params_)

Accuracy: 0.4263565891472868
```

Źródło: opracowanie własne

Jak widać, nastąpiła nieznaczna poprawa, wciąż jednak było to zbyt mało. Jako kolejny klasyfikator wybrano SGD, również z biblioteki sklearn. Występują między nimi zasadnicze różnice: KNeighborsClassifier może działać dobrze w sytuacji, kiedy granice decyzyjne są nietrywialne (a więc zbiór danych jest złożony), ale gorzej radzi sobie z dużymi zbiorami danych i ich wysoką wymiarowością, natomiast SGD lepiej radzi sobie wieloma danymi, ale wymaga większej dozy strojenia hiperparametrów. Efekty szkolenia drugiego z nich widoczne są na rysunku 5:

Rysunek 7 - trenowanie SGD

```
pipe = Pipeline([
    ('reduce_dim', 'passthrough'),
    ('classify', SGDClassifier())
])
param_grid = [
    {
        'reduce_dim': [PCA()],
        'reduce_dim__n_components': [5, 10, 20],
        'classify__alpha': [0.0001, 0.001, 0.01, 0.1],
        'classify__loss': ['hinge', 'log', 'modified_huber', 'squared_hinge'],
        'classify__penalty': ['l1', 'l2', 'elasticnet'],
        'classify__max_iter': [500, 1000]
    }
]

grid = GridSearchCV(pipe, cv=5, n_jobs=-1, param_grid=param_grid, verbose=2)

grid.fit(X_train, y_train)
```

```
best_model = grid.best_estimator_
y_pred = best_model.predict(X_test)

print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
print(grid.best_params_)
```

```
Accuracy: 0.36627906976744184
```

Źródło: opracowanie własne

Jest to najlepszy wynik, który udało się osiągnąć przy użyciu tego klasyfikatora, jednocześnie jak widać gorszy nawet od uzyskanego przez KNeighborsClassifier. Oprócz tych dwóch popularnych klasyfikatorów podjęto również próbę wytrenowania algorytmu random forest dostarczanego przez bibliotekę XGBoost. Niestety, przez techniczną niemożność szkolenia go na GPU, czas potrzebny do zakończenia jego treningu na CPU byłby dłuższy niż kilka tygodni, dlatego ostatecznie zrezygnowano z tego pomysłu.

3.3 Opis procesu uruchamiania treningów sieci neuronowych

Trening kolejnych wariacji obu wybranych modeli odbywał się na komputerze osobistym o dobrych parametrach. Aby otrzymać statystycznie istotne wyniki, każdą z wariacji uruchomiono po 20 razy. Z jednej strony pozwoliło to mimo pewnej losowości wyników odnaleźć najsilniejszą architekturę, a z drugiej najsilniejszą jej instancję. Niestety, takie podejście poskutkowało bardzo długotrwałym treningiem, który trwał około 2 miesięcy. Z tego też powodu należało napisać skrypt pozwalający zarządzać uruchamianiem treningów, oraz

zdolny je przerywać w przewidywalny sposób, nie skutkujący błędami w zapisanych modelach. Każdy z nich umieszczony został w osobnym notatniku Jupyter, co ułatwiło zarządzanie, modyfikacje oraz prostą, bazującą na wykresie dokładności analizę ostatniego wyniku. Na rysunkach 8 i 9 widać kompletny kod skryptu zarządzającego uruchomieniami, natomiast kolejne podrozdziały prezentują kod kolejnych sieci neuronowych.

Rysunek 8 - pierwsza część skryptu uruchamiającego

```
import os
import sys
import subprocess
import time
import pprint

from functions.program_running import \
    estimate_etas, \
    get_arguments, \
    get_runs_data, \
    should_exit, \
    print_green, \
    print_red


arguments = get_arguments()
model_type = arguments['type']
how_many_runs = int(arguments['runs'])
exit_file = arguments['exitfile']
chosen_model = arguments.get('model')
root_path = os.path.join('roi', 'custom_models', model_type)
runs_data = get_runs_data(root_path)

pprint.pprint(runs_data)

if chosen_model is not None:
    # noinspection PyTypeChecker
    runs_data = dict(filter(lambda pair: chosen_model in pair[0], runs_data.items()))

total_runs = len(runs_data) * how_many_runs
current_run = sum([run - 1 for run in runs_data.values()]) + 1
all_run_times = []
```

Źródło: opracowanie własne

Po zaimportowaniu kilku wbudowanych bibliotek, oraz samodzielnie napisanych funkcji pomocniczych, skrypt odczytuje argumenty linii poleceń takie jak:

- typ (InceptionResNetV2, lub Xception)
- żądana ilość uruchomień każdego modelu
- nazwa pliku, którego pojawienie się w folderze użytkownika oznacza konieczność zakończenia treningu

W dalszej części odbywa się odczyt zapisanych wcześniej ilości uruchomień każdego modelu. Dzieje się tak dlatego, że jeśli trening został przerwany, nie chciano, aby po wznowieniu startował od uruchomienia zerowego, ale od tego, na którym ostatnio skończono.

Rysunek 9 - druga część skryptu uruchamiającego

```
for notebook_path, run in runs_data.items():
    if run >= how_many_runs + 1:
        continue

    for _ in range(how_many_runs - run + 1):
        if should_exit(exit_file):
            print_red('Exit file encountered, aborting...')
            sys.exit(0)

        start_time = time.time()

        print_green(f'Run {current_run} out of {total_runs}...')
        print_green(f'Running {notebook_path}.')
        subprocess.run(f'jupyter nbconvert --execute --to notebook --inplace {notebook_path}', shell=True)

        end_time = time.time()
        elapsed_seconds = end_time - start_time
        elapsed_minutes = elapsed_seconds / 60

        all_run_times.append(elapsed_seconds)

        estimated_eta_seconds, estimated_eta_minutes = estimate_etas(all_run_times, total_runs)

        print_green(f'Run complete, elapsed seconds: {elapsed_seconds}, elapsed minutes: {elapsed_minutes}.')
        print_green(f'Estimated ETA is: {estimated_eta_seconds} seconds, {estimated_eta_minutes} minutes.')
        print()

        current_run += 1
```

Źródło: opracowanie własne

Następną częścią skryptu uruchamiającego jest pętla, która iterując po kolejnych notatnikach, uruchamia je zadaną ilość razy, w każdej iteracji sprawdzając, czy zażądano zakończenia treningu. Widać tu też, że mierzony jest czas wykonania każdego notatnika, tak, aby pokazać w linii komend szacowany czas zakończenia treningu całości.

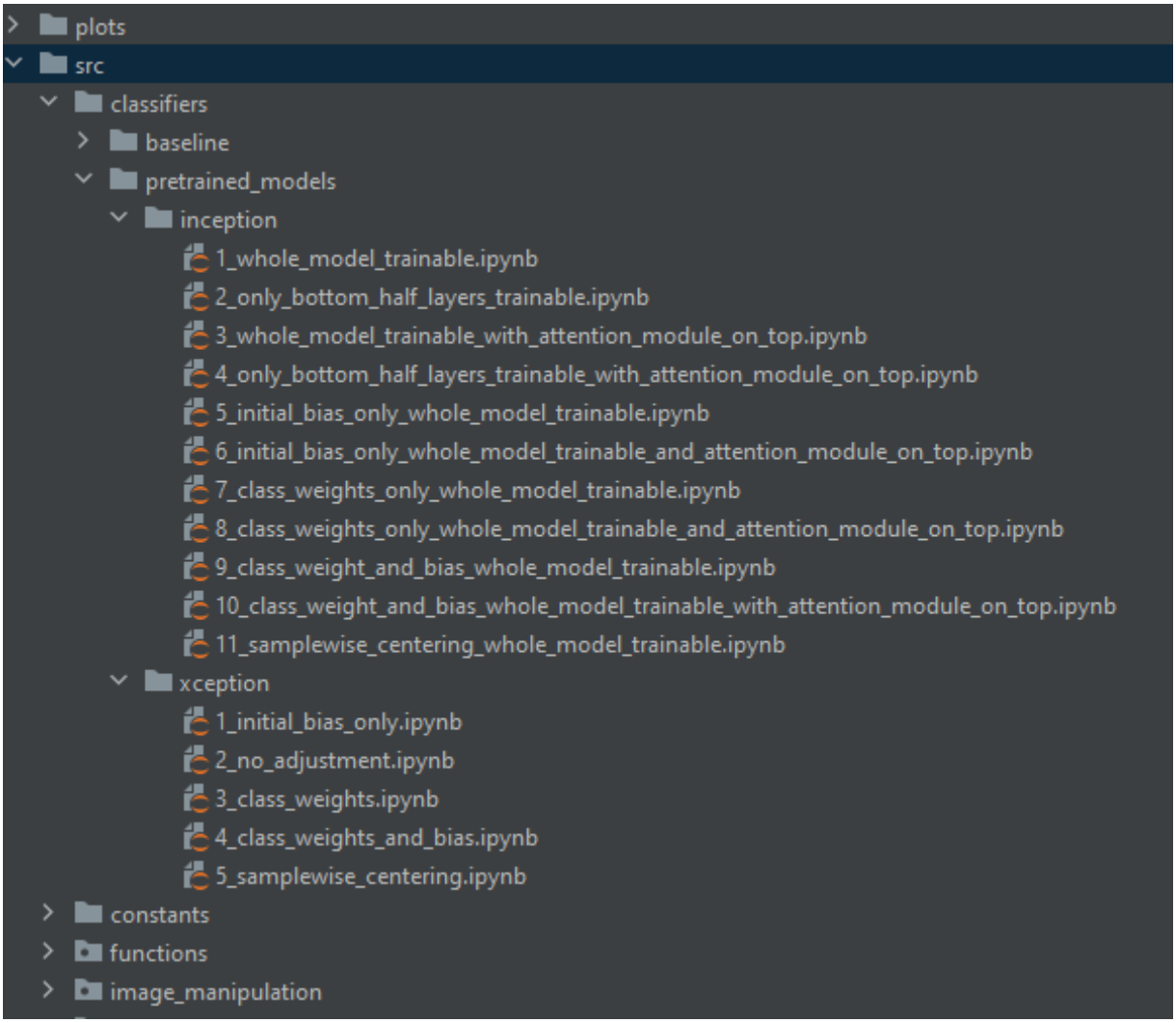
3.4 Trenowanie sieci neuronowych

Jako, że budowanie własnych architektur sieci neuronowych jest bardzo trudne, zdecydowano się na użycie sugerowanego przez wiele podręczników alternatywnego podejścia – skorzystano z tzw. transfer learning. Oznacza to, że użyte zostały gotowe modele, udostępniane przez bibliotekę TensorFlow. Następnie starano się użyć samodzielnie zbudowanych modułów, mających usprawnić działanie modeli bazowych oraz w określony sposób nadawano wartości pewnym istotnym dla procesu treningu parametrom modeli.

Rysunek 10 pokazuje strukturę folderów projektu. Najbardziej interesująca jest rozwinięta część, ponieważ pokazuje nazwy plików wskazujące na to, jakie modyfikacje bazowych architektur poczyniono. Widać tu też dwukrotną przewagę ilości modeli architektury InceptionResNetV2 (w skrócie: Inception) Xception. Różnica ta miała dwa powody:

- Oszczędność czasu – trenowanie modeli Xception było porównywalnie wolne do treningu modeli Inception, stąd dołożenie kolejnych 6 wariacji skutkowałoby kolejnym miesiącem treningu.
- Xception, jako nowsza architektura, lepiej sprawdzająca się w konkursach klasyfikacji obrazów powinien pobić architekturę Inception nawet bez żadnych usprawnień. Postanowiono to sprawdzić.

Rysunek 10 - struktura projektu



Źródło: opracowanie własne

Wariacje architektury InceptionResNetV2 znajdują się w kolejnych notatnikach, których wyjściowymi wersjami są cztery opisane dalej:

1. whole_model_trainable – jest to wariacja wyjściowa, bez modyfikacji, ani ustawionych parametrów początkowych. Żadne warstwy nie zostały w tym przypadku zamrożone, a więc w procesie treningu każda waga może zostać zmodyfikowana.
2. only_bottom_half_layers_trainable – jw. z tą różnicą, że model jest zamrożony „od połowy". Górne warstwy sieci neuronowej znają reprezentacje bardziej wysokopoziomowych cech, takich jak różnego rodzaju linie i krzywe, natomiast dolne – ich składowe. Zamrożenie modelu „od połowy" miało poskutkować tym, że zawarta w modelu wysokopoziomowa wiedza nie zostanie naruszona, zmienią się natomiast składowe kierujące model do określonych wniosków.
3. whole_model_trainable_with_attention_module_on_top – moduł uwagi zostanie opisany w kolejnym podrozdziale, oprócz niego model jest identyczny z pierwszym.
4. only_bottom_half_layers_trainable_with_attention_module_on_top – taki sam jak model drugi, prócz dodanego modułu uwagi.

Dalsze notatniki do powyższych dodają kolejno: odpowiadającą rozkładowi klas inicjalizację wag, wagi klas oraz technikę augmentacji danych znaną pod nazwą centrowania próbki. Każda z nich zostanie opisana w podrozdziale dotyczącym poczynionych usprawnień.

Kolejny interesujący folder to /plots, gdzie notatniki Jupyter składowały wykresy wygenerowane na koniec treningu każdej wariacji. Służyły one wizualnej inspekcji wyników, ale nie brały udziału w ostatecznej analizie, która była czysto statystyczna. W folderach /src/constants/ /src/functions/ oraz /src/image_manipulation umieszczone zostały fragment kodu, z których notatniki później korzystały.

Wszystkie notatniki posiadały identyczną strukturę, którą widać na rysunku 11. Najpierw importowane są potrzebne biblioteki systemowe, oraz własne, następnie nadawane są wartości stałym, które używane są na końcu przez dołączoną w pierwszym kroku funkcję run_model. Funkcja ta jest sercem każdego notatnika, ponieważ to ona definiuje kolejne kroki treningu, a jej interfejs pozwala na przekazanie do niej tzw. fabryk modeli – jako że funkcja ta używana jest wszędzie, a każdy notatnik reprezentuje inny model, to musiała zostać napisana w ogólny sposób, czyli tak, aby mogła pracować na każdym rodzaju kombinacji architektura/wariacja. Rysunek 12 zawiera tylko jej wycinek, ponieważ kod jest dość długi.

Rysunek 11 - struktura notatnika Jupyter

```
import os
import sys

module_path = os.path.abspath(os.path.join('..', '..', '..'))

if module_path not in sys.path:
    sys.path.append(os.path.join(module_path))

from functions.augmentation import get_augmentation_layers
from functions.model_running import get_run_number, run_model
from models.inception_resnet_v2_models import get_basic_model
```

```
WIDTH = 150
HEIGHT = 150
ROOT = os.path.join('..', '..', '..', '..')
DS_NAME = 'data1'
DATA_DIR = os.path.join(
    ROOT,
    '..',
    DS_NAME,
    'images_original_inception_resnet_v2_150x150_categorized')
MODEL_NAME_BASE = 'inception_1_whole_model_trainable'
```

```
run_model(
    ROOT,
    HEIGHT,
    WIDTH,
    DATA_DIR,
    DS_NAME,
    MODEL_NAME_BASE,
    lambda num_classes: lambda: get_basic_model(HEIGHT, WIDTH, num_classes),
    get_augmentation_layers)
```

Źródło: opracowanie własne

Rysunek 12 - ciało funkcji run_model

```python
def run_model(
        root: str,
        height: int,
        width: int,
        data_dir: str,
        dataset_name: str,
        model_base_name: str,
        model_getter: Callable,
        augmentation_getter: Callable,
        batch_size: int = 32,
        stopping_patience: int = 20,
        train_dataset: tf.data.Dataset = None,
        steps_per_epoch: int = None,
        epochs: int = 100,
        class_weight: dict[int, float] = None) -> (keras.Model, any):
    if train_dataset is None:
        train_dataset = load_dataset(width, height, data_dir, 'training', batch_size)

    num_classes = len(train_dataset.class_names)
    valid_dataset = load_dataset(width, height, data_dir, 'validation', batch_size)
    run_number = get_run_number(model_base_name)
    data_augmentation = augmentation_getter()
    train_dataset = prepare_train_dataset(train_dataset, data_augmentation)
    valid_dataset = prepare_valid_dataset(valid_dataset)
    model_name = f'{model_base_name}_{run_number}'
    logs_path = os.path.join(root, 'tensor_logs', dataset_name)

    if not os.path.exists(logs_path):
        os.makedirs(logs_path)

    history, model = fit_model(
        train_dataset,
        valid_dataset,
        model_getter(num_classes),
        os.path.join(root, 'tmp_models', model_name + '_{epoch}'),
        os.path.join(logs_path, model_name),
        monitor='val_loss',
        mode='min',
        reduction_patience=10,
        stopping_patience=stopping_patience,
        steps_per_epoch=steps_per_epoch,
        epochs=epochs,
        class_weight=class_weight)
```

Źródło: opracowanie własne

3.5 Usprawnianie sieci neuronowych

Jak zostanie pokazane w rozdziale opisującym analizę porównawczą, chociaż między modelami bez dużych usprawnień, a takimi, które te usprawnienia zawierają nie było wielkich różnic, to statystycznie dowiedzione zostało, że proponowane usprawnienia miały skutek w postaci wyższej dokładności zmienionych modeli. W przypadku medycyny jest to bardzo istotne, ponieważ wzrost skuteczności modelu o 1% może oznaczać ocalenie dodatkowych setek tysięcy istnień ludzkich.

Pierwszym takim usprawnieniem było dodanie modułu uwagi. Pomysł na niego zainspirowany został architekturą sieci SENet. W tym modelu moduły uwagi mają nieco inną nazwę – Squeeze and Excitation – oraz wbudowane są wewnątrz sieci, natomiast w rozważanym przykładzie taki moduł został dodany między wyjście z architektury InceptionResNetV2, a warstwy klasyfikatora. Kolejne kroki działania tego mechanizmu to:

1. Użycie warstwy GlobalAveragePooling2D interfejsu Keras oraz następująca po niej operacja Reshape: warstwa GAP oblicza średnią każdej mapy cech, redukując jej wymiary do jednej liczby, zachowując jednak wymiar głębokości, a więc wynikiem działania takiej warstwy jest tensor 1x1xGŁĘBOKOŚĆ – jego wartość można zinterpretować jako skompresowaną reprezentację danych wejściowych; taką, która pozwala mocniej wybić się globalnym cechom. W terminologii architektury SENet jest to etap nazwany „squeeze”.
2. Użycie dwóch warstw gęsto połączonych: celem pierwszej jest jedynie zwiększenie mocy poznawczej sieci, natomiast druga, poprzez użycie funkcji softmax, zamienia wyuczone w dwóch wcześniejszych krokach reprezentacje na wektor wag zawierający wartości w przedziale [0, 1]. W terminologii SENet jest to krok nazwany „excitation”.
3. Mapy cech dostarczone modułowi uwagi mnożone są przez wynik działania drugiej warstwy Dense. W tym miejscu następuje rekalibracja tych map cech. Dzięki niej, sieć może uwypuklić istotne mapy cech i wytłumić mniej istotne.

Jeśli chodzi o architekturę Xception, jak zostało wspomniane w podrozdziale 3.4, miała być ona w stanie pobić InceptionResNetV2 bez głębszych usprawnień, stąd zdecydowano się jedynie na uwzględnienie modyfikacji pewnych hiperparametrów sieci tak, aby nieco szybciej osiągała zbieżność, po uwzględnieniu nierównowagi klas. Podobnie do modeli Inception, również tutaj użyto centrowania próbki.

Kolejnym usprawnieniem było zainicjowanie wag w taki sposób, aby ich pierwotne stany odpowiadały rozkładowi klas. Aby obliczyć właściwe wartości wag utworzono funkcję calculate_class_weight, która zwraca słownik mapujący kategorię (zakodowaną w postaci liczby całkowitej) na wartość zmiennoprzecinkową. Wartości te nie sumują się do 1, ponieważ celem funkcji nie jest obliczenie rozkładu prawdopodobieństwa pomiędzy klasami, lecz nadanie każdej klasie liczby odpowiadającej jej ważności w zbiorze danych.

Rysunek 13 – funkcja calculate_class_weight

```python
def calculate_class_weight(dataset: tf.data.Dataset, alpha: float) -> dict[int, float]:
    """
    :param dataset:
    :param alpha: raw_weights can be far off from each other if the disbalance is very big, and this can
                  lead the model astray because it will focus too much on the underrpresented classes.
                  Better if it focuses on them more, but still gives some attention to the other classes.
                  This parameter should be used to balance the weights. Using 0 will make the weights
                  equal to raw_weights, and using 1 will make weights contain averages.
    :return:
    """
    class_sums = None

    for _, labels in dataset:
        batch_sum = np.sum(labels.numpy(), axis=0)
        if class_sums is None:
            class_sums = batch_sum
        else:
            class_sums += batch_sum

    total_samples = sum(class_sums)
    n_classes = len(class_sums)
    raw_weights = {i: (1 / class_sum) * total_samples / n_classes for i, class_sum in enumerate(class_sums)}
    average_weight = sum(raw_weights.values()) / n_classes
    weights = {i: alpha * average_weight + (1 - alpha) * raw_weight for i, raw_weight in raw_weights.items()}

    return weights
```

Źródło: opracowanie własne

Najbardziej interesujące są tu obliczenia widoczne w liniach przypisujących wartości zmiennym raw_weights, average_weight oraz weights.

- $\text{raw\_weight}_i = \frac{1}{\text{class\_sum}_i} \times \frac{\text{total\_samples}}{\text{n\_classes}}$ – celem tego równania jest nadanie klasom o mniejszej ilości próbek wyższego priorytetu i obniżenie go dla klas o większej ilości próbek.
- $\text{average\_weight} = \frac{\sum(\text{raw\_weights})}{\text{n\_classes}}$ – równanie to oblicza średnią wartość wag, która w następnym kroku zostanie użyta do moderowania wpływu zmienności liczb zawartych w raw_weights.
- $\text{weight}_i = \alpha \times \text{average\_weight} + (1 - \alpha) \times \text{raw\_weight}_i$ – dzięki dostrajaniu parametru alfa, funkcja calculate_class_weights może zwrócić wagi bliższe średniej (a

więc takie, jakie zostałyby zastosowane przez Keras, jeśli nie użyto by tej funkcji), albo bliższe obliczonym w pierwszym równaniu, podnosząc tym samym istotność kategorii gorzej reprezentowanych w zbiorze danych.

O ile ustawianie wag klas istotne jest z punktu widzenia inicjalizacji wag modelu oraz funkcji straty (podczas procesu propagacji wstecznej kontrybucja funkcji straty dla niedoreprezentowanej klasy jest zwiększana), o tyle ustawianie tzw. initial bias istotne jest z punktu widzenia ostatniej warstwy klasyfikatora. Przy użyciu tego parametru model będzie przywiązywał w module klasyfikatora większą wagę rzadziej występującym klasom. Wartość tego hiperparametru jest najistotniejsza u początku treningu, przyspieszając osiągnięcie zbieżności – model zaczyna z pewną początkową wiedzą na temat danych.

Rysunek 14 - obliczanie initial bias

```
class_counts = [len(os.listdir(class_dir)) for class_dir in class_dirs]
total_samples = np.sum(class_counts)
initial_biases = np.log(class_counts / (total_samples - class_counts))
```

Źródło: opracowanie własne

Użycie logarytmu naturalnego w ostatnim równaniu sprawia, że liczby zawarte w wektorze będącym argumentem do funkcji np.log będą bliżej siebie. W innym przypadku duży initial bias jednej klasy mógłby zdominować pozostałe.

Ostatnim z wybranych usprawnień jest centrowanie próbki, którego kod zawiera rysunek 16. Spośród proponowanych technik miało ono największy wpływ na stabilizację treningu. Funkcja centrująca korzysta z innej pomocniczej funkcji – generate_mask:

Rysunek 15 – generowanie maski

```
def generate_mask(img: np.ndarray) -> np.ndarray:
    grayscale = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    _, binary = cv2.threshold(grayscale, 1, 255, cv2.THRESH_BINARY_INV)
    binary = binary.astype(np.uint8)
    h, w = binary.shape[:2]
    mask = np.zeros((h + 2, w + 2), np.uint8)
    starting_points = [(0, 0), (0, h-1), (w-1, 0), (w-1, h-1),
                       (w//2, 0), (w//2, h-1), (0, h//2), (w-1, h//2)]

    for sp in starting_points:
        cv2.floodFill(binary, mask, sp, 255)

    mask = cv2.bitwise_not(binary)

    return mask
```

Źródło: opracowanie własne

Niektóre z obrazów zawartych w zbiorze danych HAM 10000 posiadały czarną otoczkę u krawędzi. Technika centrowania użyteczna jest, kiedy kontrast między ciemnymi, a jasnymi elementami zdjęcia jest znaczny (np. na pewne z nich pada cień). Jeśli dany obraz zawiera pewne artefakty, takie jak właśnie czarna otoczka, będą one miały wpływ na centrowanie i nie będzie ono skuteczne. Funkcja generate_mask maskuje te obszary, tak aby algorytm centrowania mógł skupić się na pikselach stanowiących interesującą go część obrazu.

Rysunek 16 – centrowanie obrazu

```
def call(self, inputs: tf.Tensor, *args, **kwargs) -> tf.Tensor:
    batch_masks = np.array([generate_mask(img) for img in inputs.numpy()])
    batch_masks = tf.convert_to_tensor(batch_masks, dtype=tf.float32)
    batch_masks = tf.expand_dims(batch_masks, axis=-1)
    included_pixel_sum = tf.reduce_sum(inputs * batch_masks, axis=[1, 2], keepdims=True)
    count = tf.reduce_sum(batch_masks, axis=[1, 2], keepdims=True) + 1e-7
    sample_means = included_pixel_sum / count
    centered_inputs = inputs - sample_means

    return centered_inputs
```

Źródło: opracowanie własne

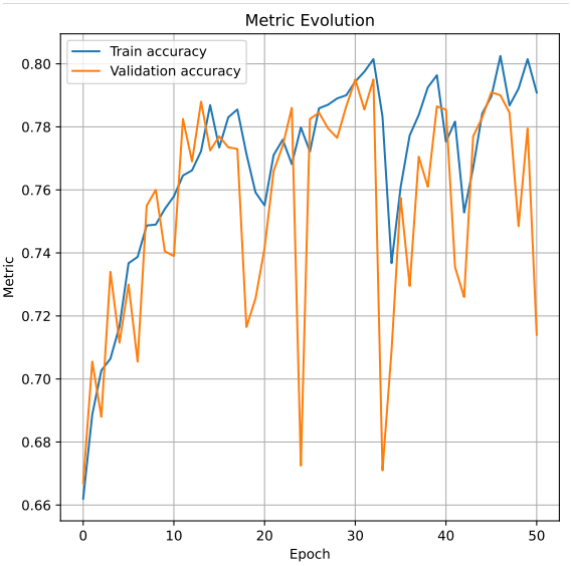
Istotą pokazanego tu kodu jest przemieszczenie wartości pikseli obrazu tak, aby były skupione wokół średniej wartości równej 0.

3.6 Wizualne porównanie wyników

Średnia dokładność wytrenowanych sieci osiągnęła rząd 80%, jednak najlepsze z nich przekroczyły 82%. Między architekturami InceptionResNetV2 i Xception wystąpiły nieznaczne różnice na korzyść pierwszej z nich, co zostanie szerzej opisane w następnym rozdziale. Wymienione wcześniej liczby odwołuję się do dokładności ogólnej, a nie w poszczególnych klasach. Jest to szczególnie istotne w problemach dotyczących medycyny, ponieważ nawet 100% dokładności w wykrywaniu łagodnych zmian skórnych nie czyni dobrego klasyfikatora, jeśli nie potrafił poprawnie skategoryzować zmiany złośliwej. Więcej na ten temat również znajdzie się w kolejnym rozdziale.

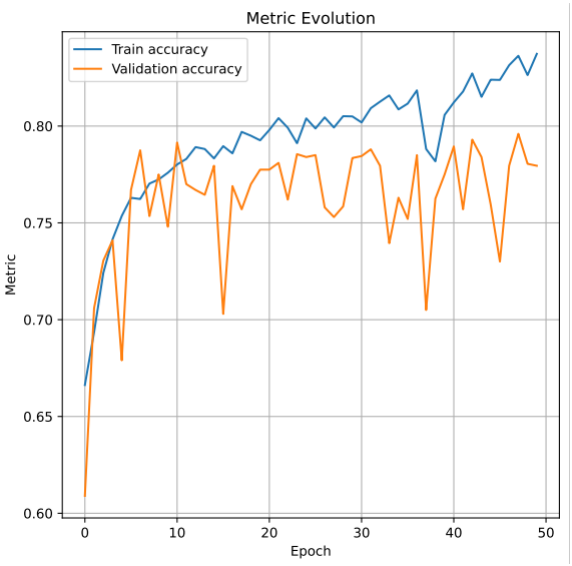
Pewne wnioski wypłynęły jednak nawet zanim przeprowadzono statystyczne porównanie modeli. Wizualna inspekcja diagramów dokładności ujawnia pewne różnice w tym jak w poszczególnych wariacjach zmieniała się owa dokładność. Na kolejnych rysunkach pokazano najbardziej interesujące wykresy reprezentatywne dla całego zbioru:

Rysunek 17



Źródło: opracowanie własne

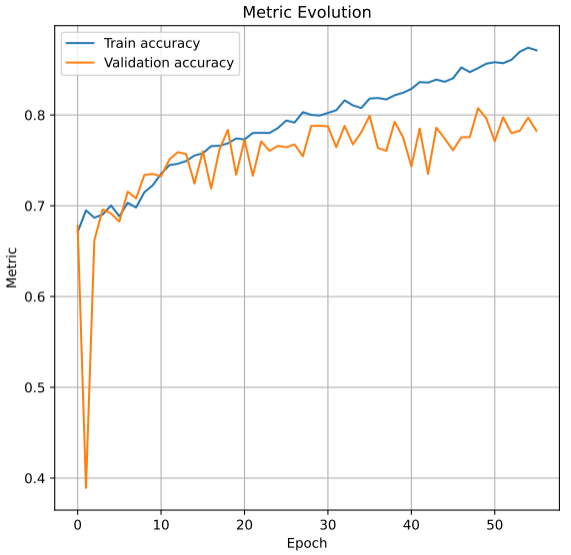
Rysunek 18



Źródło: opracowanie własne

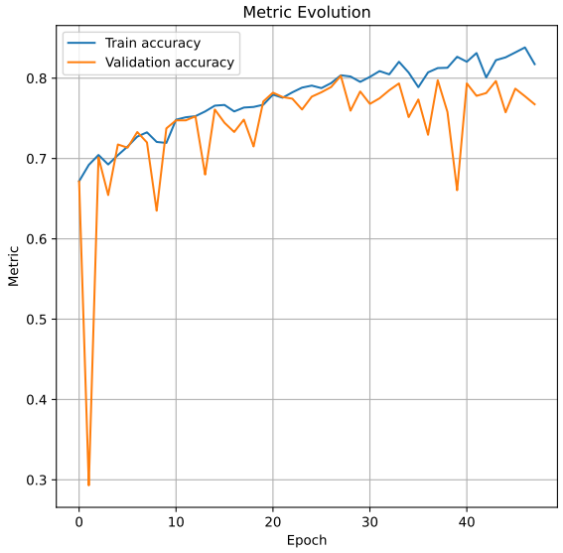
Już między rysunkiem 13, a 14 widać pewne różnice. Pierwszy z nich odpowiada notebookowi o nazwie: 1_whole_model_trainable, a drugi: 2_only_bottom_half_layers_trainable. Trening, w którym zdecydowano się na zamrożenie części sieci wykazał się większą stabilnością wag. Podobna tendencja została uwidoczniona na kilku kolejnych wykresach.

Rysunek 19



Źródło: opracowanie własne

Rysunek 20



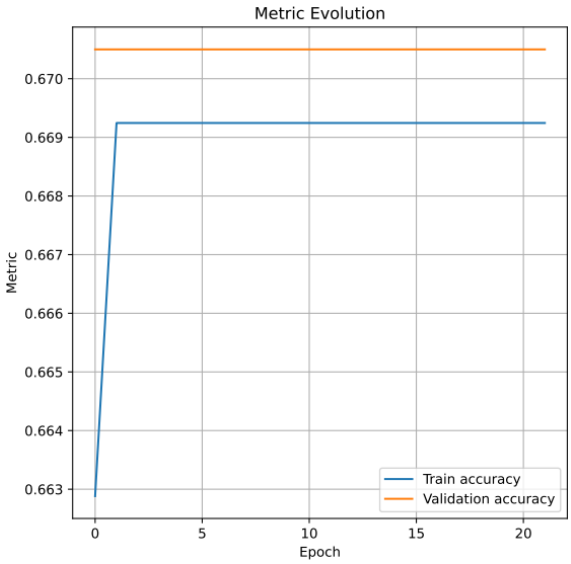
Źródło: opracowanie własne

Rysunki 15 i 16 pokazują, co stało się, kiedy dodano moduły uwagi do dwóch pierwszych sieci. Również tutaj widać, że stabilność treningu wzrosła, chociaż w pierwszym przypadku można dostrzec też pewne symptomy tzw. overfittingu.

Wygenerowano znacznie więcej tego typu wykresów, jednak kolejne przypominają te już pokazane. Tendencja była dość jasna – kolejne usprawnienia, w rodzaju alternatywnej inicjalizacji hiperparametrów, czy centrowania próbki podnosiły nieznacznie skuteczność, natomiast dodawanie modułu uwagi stabilizowało trening, dowodząc tym samym, że przy jego użyciu sieci rzeczywiście mniej dopasowują się do szumu, i lepiej wychwytują bardziej istotne informacje.

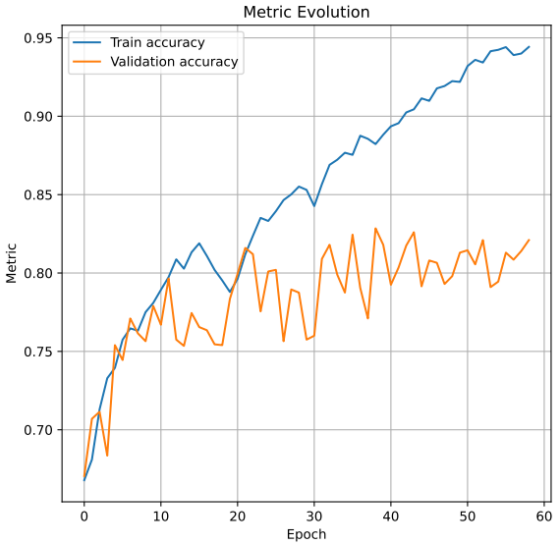
Jeśli chodzi o architekturę Xception, to dzięki wizualnej inspekcji jej wykresów zauważono, że znacznie częściej od InceptionResNetV2 występowało w niej zjawisko overfittingu. Diagramy sugerowały także nieco lepszą skuteczność, jednak ogólna niestabilność treningu oraz overfitting mogły prowadzić do wniosku, że w ostatecznym rozrachunku dokładność tej architektury będzie gorsza. Modele z tej rodziny także kilkukrotnie utknęły w tzw. minimum lokalnym, nie ucząc się od początku żadnych reprezentacji. Ostatni rysunek pokazuje wykres krzywej dokładności z treningu sieci Xception, w której użyto jedynie centrowania próbki i wynik ten okazał się najlepszy spośród wszystkich jej wariacji.

Rysunek 21



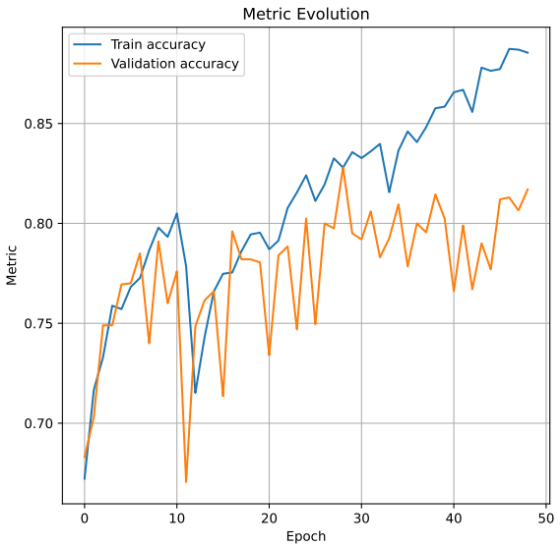
Źródło: opracowanie własne

Rysunek 22



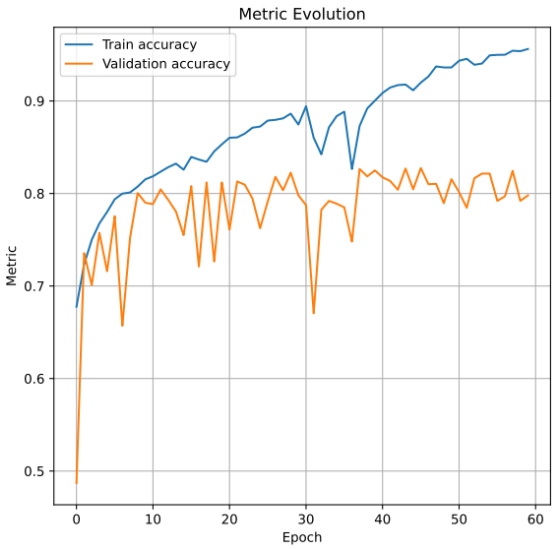
Źródło: opracowanie własne

Rysunek 23



Źródło: opracowanie własne

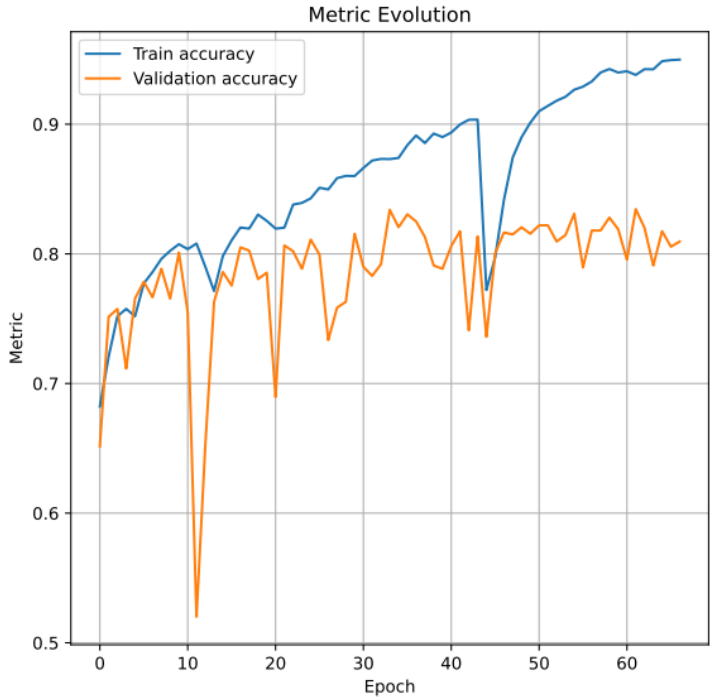
Rysunek 24



Źródło: opracowanie własne

Rysunek 21 pokazuje jeden z wykresów modelu bez usprawnień – ta wariacja najczęściej nie osiągała zbieżności. Na rysunku 22 widać charakterystyczny dla całej rodziny Xception (w problemie zbioru danych HAM10000) overfitting. Na kolejnych dwóch ilustracjach ten problem również jest dość jaskrawy, jednak nieco mniej, niż na poprzedniej. Co ciekawe, użycie samych wag klas, sprawiło że podczas niektórych uruchomień trening był bardzo niestabilny, natomiast użycie ich razem z nadaniem wartości hiperparametrowi initial bias, zdołało nieco trening ustabilizować.

Rysunek 25



Źródło: opracowanie własne

Rozdział 4 – Statystyczna Analiza Porównawcza

4.1 Opis procesu wyboru najlepszej instancji InceptionResNetV2

4.2 Opis procesu wyboru najlepszej instancji Xception

4.3 Porównanie wyników

BIBLIOGRAFIA

Opracowania książkowe

Netografia

SPIS RYSUNKÓW