Wydział Podstawowych Problemów Techniki
Politechnika Wrocławska

Rozpoznawanie emocji tłumów metodami uczenia maszynowego

Eryk Krupa
Nr indeksu: 244993

Praca inżynierska napisana
pod kierunkiem
dra inż. Jakuba Lemiesza

Politechnika
Wrocławska
Wrocław 2020

Spis treści

1 Wstęp **1**

2 Wprowadzenie do uczenia maszynowego **3**
2.1 Podstawowe pojęcia . . . 3
2.2 Przegląd literatury związanej z uczeniem głębokim . . . 4

3 Analiza problemu **5**
3.1 Szczegółowe przedstawienie problemu . . . 5
3.2 Analiza i porównanie istniejących rozwiązań . . . 5
3.3 Proponowane rozwiązania . . . 6

4 Teoria i projekt **7**
4.1 Dane . . . 7
4.2 Model konwolucyjnej sieci neuronowej . . . 7
4.2.1 Warstwy konwolucyjne . . . 7
4.2.2 Warstwy pooling . . . 8
4.3 Augmentacja danych . . . 9
4.4 Rozszerzenie uprzednio trenowanej sieci neuronowej . . . 9
4.5 Projekt aplikacji . . . 10

5 Implementacja i wyniki **11**
5.1 Opis technologii . . . 11
5.2 Implementacja . . . 12
5.2.1 Aplikacja do trenowania sieci . . . 12
5.2.2 Skrypt rozpoznający emocje . . . 14
5.3 Model konwolucyjnej sieci neuronowej . . . 14
5.4 Augmentacja danych . . . 20
5.5 Rozszerzenie uprzednio trenowanej sieci neuronowej . . . 24

6 Instalacja i użytkowanie **29**
6.1 Instalacja . . . 29
6.2 Użytkowanie . . . 29

7 Podsumowanie **31**
7.1 Podsumowanie prac . . . 31
7.2 Podsumowanie wyników . . . 31
7.3 Dalsze kroki . . . 32

Bibliografia **34**

A Zawartość płyty CD **35**

Wstęp

Sieci neuronowe pozwalające na analizę emocji tłumów to wciąż jeszcze słabo przebadany temat. W ciągu ostatnich kilku lat zyskuje jednak zainteresowanie naukowe [26][25][12][24], głównie ze względu na ogromną ilość danych dostępnych na serwisach społecznościowych, zawierających zdjęcia grup osób uczestniczących w różnych wydarzeniach towarzyskich.

Poniższa praca porusza temat uczenia maszynowego, w szczególności aspekty związane z sieciami neuronowymi i ich wykorzystaniem. Praca skupia się na projektowaniu, budowaniu i trenowaniu głębokich konwolucyjnych sieci neuronowych, a także na wykorzystaniu ich w analizie dominujących emocji panujących na zdjęciach tłumów i dużych grup ludzi. Ponadto, w pracy poruszany jest również temat baz danych przechowujących wyżej wspomniane zdjęcia wraz z odpowiadającymi im oznaczeniami wskazującymi dominującą na obrazie emocję, jak również temat aplikacji mającej ułatwić i zautomatyzować proces trenowania sieci neuronowych.

Celem pracy jest zaprojektowanie, zbudowanie i wyszkolenie sztucznej sieci neuronowej zdolnej do analizy emocji tłumów na podstawie zdjęć. Sieć ma być w stanie określić emocję panującą na fotografii, przydzielając ją do jednej z trzech kategorii: emocji pozytywnych, neutralnych bądź negatywnych.

Praca składa się z siedmiu rozdziałów. Niniejszy wstęp stanowi rozdział 1. W rozdziale 2. zaprezentowano najważniejsze pojęcia z dziedziny uczenia maszynowego, których znajomość jest niezbędna podczas lektury niniejszej pracy. Ponadto, znajduje się tam przegląd literatury związanej z sieciami neuronowymi. W rozdziale 3. szczegółowo przeanalizowano rozpatrywany problem, dokładnie sprecyzowano cele pracy, przedstawiono istniejące rozwiązania i opisano zastosowane podejścia do rozwiązania problemu. W rozdziale 4. zaprezentowano informacje dotyczące bazy danych, jak również wyjaśniono, dlaczego właśnie ta baza została wybrana. Ponadto, omówiono architekturę konwolucyjnych sieci neuronowych, podstawy jej działa, oraz problemy, z jakimi należy się mierzyć, ze szczególnym naciskiem na problem nadmiernego dopasowania do danych treningowych. Z tego powodu omówiono również technikę augmentacji danych. Przeanalizowano też modele uprzednio trenowane, których wykorzystanie w procesie transfer learningu może wpłynąć pozytywnie na dokładność walidacji. Na koniec opisano architekturę aplikacji mającej na celu usprawnić proces trenowania sieci neuronowych oraz skryptu pozwalającego na wygodne wykorzystywanie wytrenowanej sieci w celu przewidywania emocji. W rozdziale 5. opisano technologie implementacji projektu: wybrany język programowania oraz frameworki. Przedstawiono architekturę stworzonej aplikacji i skryptu. Zaprezentowano implementacje modelu sieci neuronowej, sposób augmentacji danych i rozszerzenia modelu uprzednio trenowanego, wraz z przedstawieniem wyników analiz obrazów przeprowadzanych przez zaimplementowane modele. W rozdziale 6. przedstawiono sposób instalacji i wdrożenia aplikacji w środowisku docelowym. Rozdział 7. stanowi podsumowanie stanu zakończonych prac projektowych i implementacyjnych wraz z potencjalnymi, dalszymi kierunkami ich rozwoju.

Wprowadzenie do uczenia maszynowego

2.1 Podstawowe pojęcia

Poniżej znajdują się najważniejsze pojęcia, których znajomość jest niezbędna podczas lektury niniejszej pracy.

Sztuczna inteligencja (ang. artificial intelligence) - zautomatyzowany proces myślowy standardowo wykonywany przez ludzi, zdolny do podejmowania decyzji na podstawie analizy środowiska, dążący do maksymalizacji szansy na pomyślne osiągnięcie zdefiniowanego celu.

Uczenie maszynowe (ang. machine learning) - forma sztucznej inteligencji zdolna do doskonalenia się poprzez doświadczenie. Podczas gdy klasyczne algorytmy wykorzystują zdefiniowane przez programistę reguły do przetwarzania danych w celu uzyskania pewnych wyników, algorytmy uczenia maszynowego wykorzystują dane i uprzednio uzyskane wyniki, w celu tworzenia i doskonalenia reguł przetwarzania nowych, podobnych danych.

Uczenie głębokie (ang. machine learning) - dziedzina uczenia maszynowego, sposób uczenia się, który kładzie nacisk na trenowanie modeli o wielu warstwach, z których każda kolejna stanowi coraz to lepszą reprezentację rozpatrywanych danych. Z reguły tymi modelami są sztuczne sieci neuronowe.

Sztuczna sieć neuronowa (ang. artificial neural network) - struktura realizująca obliczenia za pomocą kolejnych warstw elementów przetwarzających, tj. sztucznych neuronów (ang. artificial neuron) zdolnych do odbierania sygnału, przetwarzania go i przekazywania do kolejnych neuronów. Sygnał jest reprezentowany przez liczbę rzeczywistą, a obliczany jest za pomocą nieliniowej funkcji zwanej funkcją aktywacji (ang. activation function). Z reguły neurony są zagregowane w warstwy (ang. layers), a każda z warstw ma przypisaną funkcję aktywacji. W najprostszych modelach sekwencyjnych sygnały są przekazywane od warstwy wejściowej (ang. input layer), kolejno przez wszystkie warstwy, aż do warstwy wyjściowej (ang. output layer). Warstwę nazywamy gęsto połączoną, jeśli każdy jej neuron jest połączony z każdym z warstwy kolejnej. Ponadto, z każdym neuronem związany jest bias (ang. bias), wpływający na wartość, która musi być przekroczona, aby neuron przekazywał sygnał. Połączenia między neuronami reprezentowane są przez zmienne zwane wagami (ang. weights). Modyfikowanie wag i biasów w trakcie prowadzenia obliczeń pozwala na dostrajanie modelu, czyli uczenie się sztucznej sieci neuronowej. Do tego celu potrzebne są dane treningowe, wraz z przypisanymi im etykietami docelowymi. Podczas nauki, sieć wykorzystuje funkcje straty (ang. loss function) do porównywania wartości etykiet docelowych z tymi przewidywanymi przez model. Na podstawie wyniku tej funkcji sieć jest w stanie określić, jak odległa była jej odpowiedź od odpowiedzi poprawnej. Ta wiedza wykorzystywana jest przez optymalizator (ang. optimizer) w celu dostrajania sieci.

Konwolucyjna sieć neuronowa (ang. convolutional neural network, CNN, ConvNet) - to szczególna klasa sieci neuronowych, szeroko wykorzystywana w analizie obrazów i wideo z uwagi na przewagę nad klasycznymi sieciami neuronowymi w przynajmniej dwóch aspektach. Podstawową różnicą między siecią gęsto połączoną a siecią konwolucyjną jest to, że ta pierwsza uczy się rozpoznawania cech globalnych, a ta druga- lokalnych wzorców. W przypadku obrazów cechy są znajdowane w małych, dwuwymiarowych oknach danych wejściowych (czyli małych wycinkach obrazu w przypadku analizy podejmowanej przez pierwszą warstwę sieci). Dzięki temu, wzorce obrazu rozpoznawane przez model są niezależne od pozycji tego wzorca na obrazie. Dla przykładu, sieć konwolucyjna po rozpoznaniu określonego wzoru w prawym dolnym rogu obrazu może rozpoznać go również np. w lewym górnym rogu obrazu. Sieć gęsta musiałaby nauczyć się tej cechy na nowo. Oznacza to, że modele konwolucyjne nie wiążą kształtu z jego położeniem w kadrze. Z tego powodu charak-

teryzują się one większą wydajnością i wymagają mniejszej ilości danych treningowych. Drugą zaletą sieci konwolucyjnych jest to, że uczą się hierarchii wzorców. Oznacza to, że pierwsze warstwy uczą się małych lokalnych wzorców, takich jak na przykład krawędzie czy tekstury, a kolejne bardziej skomplikowanych struktur, składających się z elementów rozpoznanych przez warstwy poprzednie. Takie podejście pozwala na wydajne uczenie się złożonych i abstrakcyjnych wzorców graficznych.

2.2 Przegląd literatury związanej z uczeniem głębokim

Poniżej znajduje się krótki przegląd literatury związanej z implementacją sieci neuronowych.

W 1989 roku w [18] zaimplementowano algorytm propagacji wstecznej w modelu głębokim, w celu rozpoznawania ręcznie pisanych cyfr w wiadomościach pocztowych. Baza danych składała się z 9298 cyfr różnej wielkości, napisanych przez wielu ludzi o różnych stylach pisania. Algorytm propagacji wstecznej pozwolił trenować sieć zaskakująco szybko, biorąc pod uwagę wielkość zbioru danych. Model znalazł zastosowanie w komercyjnych maszynach automatyzujących proces odczytywania ręcznie pisanych cyfr.

W 1992 roku w [27] zaimplementowano model o nazwie Cresceptron, który wykorzystywał uczenie nienadzorowane (nie wymagał wcześniejszego etykietowania danych przez programistów) do rozpoznawania obiektów w kadrach, w których było ich bardzo wiele. Cresceptron uczył się dowolnej liczby cech dla każdej warstwy, a każda cecha była reprezentowana przez jądro konwolucji. Był to pierwszy model wykorzystujący warstwy max pooling do redukcji rozdzielczości tensorów przyjmowanych przez daną warstwę - analizowały one okna konwolucji o rozmiarze 2x2 z krokiem równym 2 (patrz rozdział 4).

W problemach rozpoznawania mowy często używana jest rekurencyjna sieć neuronowa opublikowana w 1997 przez [23], wykorzystująca metodę zwaną long short-term memory (LSTM). Ta sieć jest zdolna do rozwiązywania problemów, które wymagają pamięci o wydarzeniach, które miały miejsce wiele kroków wcześniej, co jest istotne m.in. w kontekście rozpoznawania mowy. Sześć lat później LSTM zaczęło konkurować w rozwiązywaniu pewnych problemów z tradycyjnymi sieciami rozpoznającymi mowę [16]. Później sieci LSTM zostały połączone z CTC (connectionist temporal classification) [17], co zostało wykorzystane przez firmę Google w 2015 roku do podniesienia wydajności sieci rozpoznających mowę w oparciu o dane zebrane dzięki Google Voice Search [22].

Rok 2009 przyniósł przełom w dziedzinie sprzętu wykorzystywanego do trenowania sztucznych sieci neuronowych. Firma Nvidia wytrenowała model na procesorze graficznym, udowadniając, że GPU może zwiększyć szybkość trenowania sieci około 100-krotnie, w szczególności świetnie radząc sobie z obliczeniami przeprowadzanymi na tensorach [20].

Analiza problemu

W tym rozdziale został przedstawiony problem analizy emocji na zdjęciach tłumów i grup ludzi, dotychczasowe podejścia do tego problemu, jak również podejście zastosowane w tej pracy.

3.1 Szczegółowe przedstawienie problemu

Celem pracy jest wytrenowania skutecznego modelu konwolucyjnej sieci neuronowej umożliwiającego klasyfikacje emocji dominującej wśród dużej grupy osób na podstawie obrazu, ze szczególnym naciskiem na to, by wytrenowany model był jak najmniejszy, a czas rozpoznawania emocji - jak najkrótszy. Tak skonstruowana, niewielka sieć będzie mogła zostać zapisana na urządzeniu mobilnym. Z kolei wysoka szybkość działania umożliwia analizę obrazu w czasie zbliżonym do czasu rzeczywistego. Obie te cechy pozwolą na stworzenie aplikacji mobilnej, która np. będzie robić zdjęcia za każdym razem, kiedy w kadrze zostaną rozpoznane emocje pozytywne.

Rozpoznawane emocje będą przypisywane do jednej z trzech grup: emocji pozytywnych, neutralnych i negatywnych. Domyślnym podejściem do podobnych problemów jest podzielenie go na dwa podproblemy. Na początku trenuje się model będący w stanie wykrywać twarze na zdjęciach. Fragmenty obrazu zawierające twarze są następnie przekazywane do drugiego modelu odpowiedzialnego za rozpoznawanie emocji. Takie podejście ma jednak oczywistą wadę, a mianowicie wykrywa się jedynie emocje wyrażone na twarzach, a pomija się te, które wynikają, chociażby, z mowy ciała. Z tego powodu często trenuje się jeszcze jeden model, mający na celu zbadać ogólne cechy danej sceny, tj. ocenić ogólny nastrój zdjęcia. Dopiero tą wartość łączy się z uśrednionymi emocjami wykrytych twarzy i zwraca jako przewidywaną, ogólną emocję panującą na zdjęciu.

3.2 Analiza i porównanie istniejących rozwiązań

W kontekście rozpoznawania emocji na zdjęciach tłumów, a w szczególności w przypadku podziału tych emocji na trzy wyżej wymienione kategorie, wartą przytoczenia jest praca [14]. Autorzy stworzyli zbiór Group Effect Database 3.0 (zobacz rozdział 4), na którym przetestowali modele będące w istocie połączeniem modeli BoW_{AU}, BoW_{LL}, $Scene_{GIST}$ [21] i $Scene_{CENTRIST}$ [28].

BoW (Bag of Word) [19] to reprezentacja, która opiera się na przypisywaniu słów do analizowanych danych, tworząc tym samym zbiór niepowiązanych ze sobą słów kluczowych. BoW_{AU} analizuje napięcie mięśni twarzy, wykorzystując *BoW* do przypisywania słów kluczowych do każdej twarzy na zdjęciu, tworząc w ten sposób zbiór słów związanych z mimiką, takich jak np. uśmiech, szczęście, itd. BoW_{LL} wykrywa niskopoziomowe cechy zdjęć twarzy, takie jak np. krawędzie, nasycenie kolorów czy oświetlenie. $Scene_{GIST}$ i $Scene_{CENTRIST}$ to w istocie deskryptory sceny - analizują cechy zdjęć nie związane z twarzami, ale z elementami otoczenia, takie jak wystrój scenerii, ale też ubiór postaci na zdjęciu. Należy jeszcze dodać, że autorzy w łączeniu modeli wykorzystywali Multiple Kernel Learning (MKL) [11].

Żaden z tych modeli nie zapewnia na walidacyjnym zbiorze danych dokładności wyższej niż 51%. Jednakże ich połączenie pozwoliło na osiągnięcie zdecydowanie lepszych wyników. [14] osiągnął najlepsze rezultaty (na poziomie 67.64% na zbiorze walidacyjnym), łącząc BoW_{AU}, BoW_{LL} i $Scene_{CENTRIST}$ za pomocą MKL. Na ten wynik złożyło się 83.72% dokładności dla zdjęć emocji pozytywnych, 80% dla emocji neutralnych i tylko 31.03% dla emocji negatywnych. Należy przy tym zauważyć, że zbiór zdjęć przedstawiających emocje negatywne w bazie Group Effect Database 3.0 jest najmniejszy.

Inna zaprezentowana przez [14] architektura, składająca się z BoW_{AU}, BoW_{LL} i $Scene_{GIST}$ połączonych za pomocą MKL, uzyskała, co prawda, nieznacznie gorszą ostateczną dokładność (67.15%), jednakże osiągając wynik na poziomie 50% dla zdjęć emocji negatywnych.

Kolejnym przykładem pracy poruszającej temat trójklasowej klasyfikacji w problemie rozpoznawania emocji tłumów jest [15]. Autor wykorzystuje hybrydowy system, który składa się z głębokich konwolucyjnych sieci neuronowych i klasyfikatora bayesowskiego. Na początku wydobywa zdjęcia twarzy z obrazu przedstawiającego tłum lub grupę ludzi, a następnie przekazuje je do sieci neuronowych, które analizują wyrażane emocje na poszczególnych zdjęciach twarzy, zwracając uśredniony wynik. Jednocześnie, klasyfikator bayesowski wykorzystuje deskryptor sceny, by wydobyć ze zdjęcia wizualne cechy otoczenia, a następnie integruje je, przewidując globalną emocję związaną z obrazem. Na końcu sieci neuronowe przekazują rozpoznane emocje do klasyfikatora, a ten zwraca ostateczną, ogólną decyzję odnośnie emocji panującej na zdjęciu. Wykorzystując bazę zdjęć Group Affect Datebase 3.0 do trenowania powyższych modeli, [15] osiągnął wynik na poziomie 65.27% dokładności na walidacyjnym zbiorze danych.

3.3 Proponowane rozwiązania

W tej pracy przetestowane zostaną nieco inne rozwiązania rozważanego problemu. Pierwszym jest zaimplementowanie jednego model sieci neuronowej, który analizuje całe zdjęcie i określa dominującą na nim emocje. To podejście zostanie wsparte techniką augmentacji danych. Największą jego zaletą jest niewielka waga wytrenowanego modelu i krótki czas przetwarzania zdjęć. Umożliwi to wykorzystanie sieci w aplikacjach przeznaczonych na urządzenia mobilne, których celem jest analiza emocji panujących na zdjęciach w czasie zbliżonym do rzeczywistego. Kolejną zaletą tego podejścia jest konieczność zaimplementowania tylko jednego modelu, z jednoczesnym uwzględnienia zarówno emocji wypisanych na twarzach, tych wynikających z mowy ciała, jak i tych, na które wskazuje ubiór postaci czy otoczenie.

Drugim zastosowanym podejściem jest wykorzystanie modeli wytrenowanych wcześniej na innych, dużo większych i bardziej ogólnych zbiorach danych, w celu skorzystania z ich hierarchii cech w charakterze ogólnego modelu przetwarzania obrazu (ang. transfer learning). Taką hierarchię można wykorzystać do rozpoznawania klas, do których sieć nie była wcześniej szkolona. Może to szczególnie zwiększyć wydajność modelu w przypadku małego zbioru treningowego i problemów związanych ze zbytnim dopasowywaniem się sieci do posiadanego zbioru (ang. overfitting).

Teoria i projekt

W tym rozdziale zostały zaprezentowane informacje dotyczące bazy danych, konwolucyjnych sieci neuronowych, augmentacji danych, wykorzystania modeli uprzednio trenowanych (ang. transfer learning), jak i implementowanej aplikacji mającej na celu usprawnić proces trenowania modeli.

4.1 Dane

Do wytrenowania sieci neuronowej zdolnej rozpoznawać emocje na zdjęciach tłumów i grup ludzi potrzebny jest odpowiedni, tj. poprawnie poetykietowany zbiór danych. Znalezienie takiego zestawu okazało się nietrywialne z uwagi na brak publicznie dostępnych, poetykietowanych zbiorów zawierających zdjęcia tłumów. Ostatecznie wykorzystano zbiór Group Affect Database 3.0 [14]. Nie jest on dostępny w internecie, ponieważ autorzy zapewniają do niego dostęp jedynie uczestnikom corocznie organizowanego przez nich konkursu Emotion Recognition in the Wild Challenge [2][13], jednak został wyjątkowo udostępniony wyłącznie do celów akademickich na potrzeby niniejszej pracy i nie może być przekazywany dalej bez zgody autorów.

Zbiór Group Affect Database 3.0 to ponad 2-gigabajtowa baza danych, na którą składają się 17 172 zdjęcia przypisane do jednej z trzech kategorii: emocje pozytywne (np. szczęście, miłe zaskoczenie), neutralne (np. brak emocji czy znudzenie) oraz negatywne (złość, strach, smutek, obrzydzenie czy niemiłe zaskoczenie). Ponadto, zbiór jest podzielony na 3 podzbiory: treningowy, walidacyjny i testowy (odpowiednio 57.2%, 25,3% i 17.5% wszystkich zdjęć), co w domyśle ma ułatwić proces uczenia modelu. Tutaj warto zauważyć, że zbiory różnych emocji nie są równoliczne. W treningowym zbiorze danych zdjęcia pozytywne stanowią około 40.5% zdjęć, zdjęcia neutralne- 31.4%, a zdjęcia negatywne tylko 28.1%. W przypadku zbioru walidacyjnego jest to odpowiednio 40.2%, 31.5% i 28.3%.

W poprzednim rozdziale (3) przytoczono, że autorzy bazy osiągnęli dokładność na walidacyjnym zbiorze danych rzędu 67.64% i 67.15% w przypadku swoich dwóch najlepszych modeli. Zauważono też, że niezależni deweloperzy również osiągają wysokie, przekraczające 65% wyniki. Jest to istotne z punktu widzenia tej pracy, ponieważ daje lepszą wizję tego, jaka dokładność modelu może zostać uznana za dobrą. Trzeba jednak zauważyć również, że z uwagi na próby utrzymania modelu w jak najmniejszym rozmiarze, osiągnięcie podobnych wyników może okazać się trudne lub wręcz niemożliwe.

4.2 Model konwolucyjnej sieci neuronowej

Model rozpoznający emocje zostanie oparty na głębokiej konwolucyjnej sieci neuronowej, z uwagi na to, że są one najefektywniejsze w przypadku problemów rozpoznawania obrazów. Model będzie przyjmował trójwymiarowe tensory (mapy cech), których dwie pierwsze osie będą definiować wysokość i szerokość, natomiast trzecia oś będzie osią głębi (kanałów) obrazu. Z racji tego, że rozpatrywane będą kolorowe obrazy RGB, trzecia oś będzie mieć dokładnie trzy wymiary (dla kolorów czerwonego, zielonego i niebieskiego).

4.2.1 Warstwy konwolucyjne

Kolejne tensory tworzone przez następujące po sobie warstwy sieci również będą tensorami trójwymiarowi, ale rozmiar poszczególnych wymiarów będzie się zmieniał. I tak, szerokość i wysokość będzie maleć lub zachowywać ten sam rozmiar, natomiast wymiar głębi może przyjąć dowolną wartość, z uwagi na to, że po przejściu przez pierwszą warstwę przestaje on reprezentować nasycenie poszczególnych kolorów składowych,

a zaczyna przechowywać filtry. Filtry stanowią reprezentacje określonych cech danych. Liczba filtrów jest parametrem warstwy. W pierwszych warstwach filtry mogą kodować krawędzie czy tekstury, w kolejnych proste kształty, i tak dalej, przez coraz bardziej skomplikowane struktury, by w ostatnich warstwach reprezentować najbardziej złożone obiekty, jak ludzkie sylwetki czy twarze.

Warstwa będzie przyjmować mapę cech i dokonywać ekstrakcji łat z tych cech wejściowych. Łaty to niewielkie, dwuwymiarowe „okienka" danych, z reguły o rozmiarze 3x3 lub 5x5. Można powiedzieć, że za pomocą tych okienek warstwa obserwuje tensor wejściowy. Tensorem wejściowym dla pierwszej warstwy jest obraz, a więc obserwuje ona jego kolejne fragmenty o wymiarze 3x3 lub 5x5 pikseli i przeprowadza operacje celem utworzenia wyjściowego tensora, który posłuży za mapę cech do kolejnej warstwy. Warto nadmienić, że rozmiar okna, zupełnie jak liczba filtrów, również jest parametrem warstwy i może różnić się, w zależności od jej głębokości. Z reguły jednak zachowuje stałą wartość dla całej sieci.

Konwolucja to przesuwanie okien po trójwymiarowej mapie cech. Warstwa umieszcza okno w każdym możliwym miejscu otrzymanego tensora, każdorazowo dokonując ekstrakcji łaty o rozmiarze (wysokość_okna, szerokość_okna, głębia_wejściowa). Następnie każda łata jest przekształcana przez jądro konwolucji w celu uzyskania jednowymiarowego wektora o długości równej wyjściowemu rozmiarowi głębi. Warstwa łączy wszystkie te wektory w jeden trójwymiarowy tensor o rozmiarze (wysokość_wyjściowa, szerokość_wyjściowa, wyjściowy_rozmiar_głębi). Należy zaznaczyć, że wektory są układane w sposób odpowiadający ich położeniu w tensorze wejściowym. Na koniec, nowo stworzony tensor jest przekazywany do następnej warstwy.

Wysokość i szerokość tensora wejściowego może się różnić (i z reguły się różni) o tych samych parametrów tensora wyjściowego. Wynika to z faktu, że warstwa nie jest w stanie dopasować tylu okienek, ile wynosi iloczyn szerokości i wysokości tensora wejściowego. Dla przykładu, jeśli okienko jest rozmiarów 3x3, to wyjściowa szerokość będzie równa szerokości wejściowej pomniejszonej o 2 (analogicznie dla wysokości). Można temu zaradzić, stosując technikę dopełnienia. Polega ona na dodawaniu wierszy i kolumn do wejściowej mapy cech w taki sposób, żeby można było zmieścić odpowiednią liczbę okien konwolucji. Na przykład, dla okna rozmiaru 3x3 wystarczy dodać po jednym wierszu u góry i dołu oraz po jednej kolumnie po lewej i prawej stronie. W praktyce jednak nie stosuje się tej techniki często, zamiast tego pozwalając tensorowi zmniejszać swoją szerokość i wysokość z warstwy na warstwę.

Warstwa nie musi ustawiać okien konwolucji w każdym możliwym miejscu. Biorąc pod uwagę fakt, że różne okna stojące obok siebie analizują częściowo te same dane, czasami opłacalne może okazać się „skakanie" np. co dwie albo co trzy możliwe pozycje. Tę technikę nazywa się konwolucją kroczącą. Należy zwrócić uwagę na to, że przeskakiwanie co n-tej pozycji zmniejszy liczbę wygenerowanych okienek n^2 razy (ponieważ analizowane będzie n razy mniej wierszy i n razy mniej kolumn). W związku z tym tensor wyjściowy również będzie analogicznie mniejszy. Ponownie jednak, w praktyce raczej nie stosuje się tej techniki często, zamiast tego analizując wszystkie możliwe okna konwolucji.

4.2.2 Warstwy pooling

Sieci konwolucyjne nie składają się tylko i wyłącznie z warstw konwolucyjnych. Pomiędzy tymi warstwami umieszcza się również warstwy pooling, których zadaniem jest z reguły drastyczne zmniejszenie przekazywanego tensora, z jednoczesnym zachowaniem najważniejszych cech danych. Warstwy pooling również korzystają z koncepcji okien i kroczenia, ale w tym przypadku pojedynczej łaty danych nie przekształca się za pomocą jądra konwolucji, a wykonuje się na niej prostą operację tensorową, taką jak np. branie maksimum lub wartości średniej. Podczas gdy warstwy konwolucyjne z reguły używają okien 3x3 z krokiem równym jeden, warstwy pooling używają raczej okien 2x2 z krokiem równym 2. Dzięki temu każda wartość analizowana jest dokładnie raz, a wyjściowy tensor jest drastycznie zmniejszany. Ponadto, zastosowanie funkcji zwracającej maksimum sprawia, że z analizowanej łaty najprawdopodobniej zostanie wyciągnięta najbardziej istotna wartość. Dzieje się tak, gdyż cechy z reguły reprezentują obecność jakiegoś wzorca lub koncepcji, a najłatwiej jest je zauważyć, wyciągając właśnie elementy maksymalne. Z tego też powodu funkcja zwracająca maksimum jest częściej wykorzystywana, niż funkcja zwracająca wartość średnią.

Zmniejszanie rozdzielczości kolejnych map cech przekazywanych między warstwami ma kilka zasadniczych zalet. Po pierwsze, ograniczana jest liczba współczynników, jakie muszą być wytrenowane przez model. Dzięki temu skracany jest czas trenowania sieci, ale co ważniejsze z punktu widzenia tego projektu, zmniejszany jest jej rozmiar. Ponadto, ograniczanie z góry liczby parametrów modelu pozwala na uniknięcie nadmiernego dopasowania do danych treningowych. Oprócz tego architektura o „zwężających się", ale coraz głębszych

warstwach ułatwia sieci zaimplementowanie hierarchii filtrów przestrzennych, ponieważ każda następna warstwa analizuje coraz większe okna obrazu wejściowego.

4.3 Augmentacja danych

Baza danych, która posłuży do trenowania modelu, zawiera 9 815 zdjęć w treningowym zbiorze danych. Biorąc pod uwagę jak skomplikowane zagadnienie jest analizowane, ta liczba zdjęć może okazać się niewystarczająca, aby uchronić sieć przed nadmiernym dopasowaniem do danych treningowych. W każdym kolejnym cyklu trenowania model będzie otrzymywał te same dane wejściowe. Z tego powodu sieć może nie stworzyć pożądanych uogólnień (odpowiedniej hierarchii filtrów przestrzennych), które sprawdzą się podczas przetwarzania nowych danych, a zamiast tego po prostu uczyć się rozpoznawać konkretne zdjęcia ze zbioru treningowego. Aby temu zapobiec zostanie zastosowana, często wykorzystywana w kontekście przetwarzania obrazów, technika augmentacji danych.

Augmentacja danych pozwala na rozszerzenie zbioru treningowego o nowe elementy (wygenerowane poprzez losowe przekształcenia już istniejących obrazów), które wyglądają wiarygodnie i można je przypisać do tej samej kategorii, co obraz wejściowy. Dzięki temu, w kolejnych epokach trenowania model nie będzie musiał korzystać z tych samych danych, lecz za każdym razem zostanie dla niego wygenerowany nowy zbiór. Będzie on podobny do zbioru treningowego, ale losowe przekształcenia zagwarantują, że każde zdjęcie przekazane do modelu będzie unikatowe. Zmniejszy to możliwość dopasowywania się modelu do zbioru treningowego i pomoże w stworzeniu bardziej uniwersalnych filtrów, które przydadzą się do analizy nowych zdjęć.

Przykładowe modyfikacje zdjęć, które mogą zostać zastosowane, to rozjaśnianie lub przyciemnianie, przesuwanie lub odbijanie obrazów w osi poziomej lub pionowej, przybliżanie pewnych fragmentów czy obrót o pewien losowy, niewielki kąt.

Innym wykorzystywanym razem z augmentacją danych sposobem na zmniejszenie nadmiernego dopasowywania się sieci do danych treningowych jest użycie warstwy porzucenia (ang. dropout). W tym przypadku kluczową ideą jest losowe porzucanie neuronów z modelu (razem z ich połączeniami) podczas trenowania. Zapobiega to zbytniej wspólnej adaptacji neuronów, a w konsekwencji istotnie zmniejsza nadmierne dopasowanie modelu do danych treningowych. Tę warstwę wykorzystuje się często nie tylko przy rozpoznawaniu obrazów, ale też np. przy rozpoznawaniu mowy.

4.4 Rozszerzenie uprzednio trenowanej sieci neuronowej

Problem nadmiernego dopasowania dla niewielkich zbiorów danych można skutecznie rozwiązać również korzystając z uprzednio wytrenowanej konwolucyjnej sieci neuronowej. To podejście polega na wykorzystaniu sieci, która została już wcześniej wytrenowana na innym, bardzo dużym zbiorze obrazów. Ten zbiór nie musi być jednak podobny do zbioru, którym dysponujemy. Istotne jest, aby był na tyle duży i ogólny, by można było użyć jego hierarchii cech w charakterze ogólnego modelu przetwarzania obrazu. Taka sieć jest już nauczona pewnych mniej lub bardziej skomplikowanych kształtów. Ta wiedza może przydać się również do rozwiązywania problemów, do których sieć nie była wcześniej szkolona. W analizowanym problemie powinna być więc szczególnie przydatna.

Oczywiście, nie można skorzystać z takiej sieci bez wprowadzania modyfikacji. Modele o najlepszych hierarchiach cech są uprzednio trenowane do rozpoznawania tysięcy różnych klas, a w analizowanym problemie nasz model powinien zwracać jedną z trzech kategorii opisujących emocje. Dlatego też, z wcześniej wytrenowanego modelu zostanie wyciągnięta tylko baza konwolucyjna, a następnie zostanie do niej podpięty nowy, gęsto połączony klasyfikator, który po przemapowaniu otrzymanych od niej danych będzie w stanie rozpoznawać pożądane emocje. Baza konwolucyjna to fragment sieci CNN składający się z warstw konwolucyjnych i warstw pooling, bez warstw gęstych.

By nie zniszczyć wyuczonej hierarchii cech w bazie konwolucyjnej można ją „zamrozić”, tj. nie pozwolić modelowi na trenowanie jej, a skupić się za to na gęsto połączonym klasyfikatorze. To tylko jedno z możliwych podejść, ale ma wielką zaletę: możliwość cachowania.

Skoro w każdym cyklu trenowania sieć będzie otrzymywać te same dane (w tym przypadku augmentacja danych nie zostanie użyta), a jednocześnie baza konwolucyjna będzie zamrożona (jej wagi i biasy nie będą modyfikowane), oznacza to, że uprzednio trenowana baza konwolucyjna będzie za każdym razem przekazywać identyczne tensory do gęsto połączonego klasyfikatora. Nie ma więc powodu, by każdorazowo odczytywać z dysku tysiące zdjęć, a następnie w każdej iteracji trenowania karmić nimi tę samą sieć, oczekując tych samych wyników. Zamiast tego, zostanie zaimplementowany cache, który dla danej bazy konwolucyjnej i dla danego sposobu preprocessingu danych, przetworzy zdjęcia tylko raz, a następnie zapisze zwrócone tensory w pamięci po to, by gęsto połączony klasyfikator mógł z nich korzystać bez każdorazowego przetwarzania ich przez bazę konwolucyjną. Oczywiście, ten mechanizm musi być w stanie rozpoznawać, czy przy kolejnym uruchomieniu aplikacji dane zostały przetworzone w taki sam sposób i czy została użyta ta sama baza konwolucyjna. Jeśli coś się zmieniło, nowe wartości muszą zostać przetworzone od nowa i również zapisane na dysku.

Dzięki wykorzystaniu tego podejścia zostanie wyeliminowany najbardziej kosztowny obliczeniowo element przetwarzania danych, co ma olbrzymie znaczenia praktyczne. Po pierwsze, pozwoli to na przeprowadzanie nawet setek cykli uczenia bez konieczności oczekiwania wielu godzin na wyniki. Po drugie, pozwoli na „hurtowe" trenowanie modeli dla różnych metaparametrów, co ułatwi podejmowanie decyzji, który z nich jest warty dalszego rozwijania.

4.5 Projekt aplikacji

Wyżej opisana funkcjonalność będzie elementem aplikacji, która zostanie zaimplementowana w celu możliwie maksymalnego zautomatyzowania procesu trenowania sieci neuronowych. Sercem aplikacji, a zarazem sposobem na łatwe nią sterowanie, będzie plik konfiguracyjny przechowujący ścieżki do wszystkich katalogów wykorzystywanych przez aplikacje. Oprócz tego będzie przechowywał metaparametry wykorzystywane przy wstępnej obróbce danych, augmentacji danych i trenowaniu modeli, a ponadto również informacje o sposobie formatowania wykresów prezentujący wyniki oraz o tym, jakie logi mają być wyświetlane w konsoli dla użytkownika.

Korzystając z tych danych, aplikacja zbuduje swój model sieci neuronowej, lub skorzysta ze zdefiniowanej bazy konwolucyjnej. W obu przypadkach odpowiednio przygotuje dane do procesu trenowania. W pierwszym przypadku wykorzysta dane z odpowiedniej ścieżki, uprzednio te dane augmentując lub nie, w zależności od konfiguracji. W drugim przypadku wykorzysta ścieżkę do cachowanych danych, a jeśli tej ścieżki nie odnajdzie, lub nie odnajdzie cacha zapisanego dla tej konkretnej bazy konwolucyjnej i sposobu preprocessingu danych, skorzysta ze ścieżki do bazy celem stworzenia odpowiedniego cacha. Oprócz tego aplikacja, korzystając ze zdefiniowanych ścieżek, zapisze na dysku każdy wytrenowany model, jak i dane potrzebne do zbudowania wykresów podsumowujących jego wydajność. Ponadto, każdorazowo zbuduje i wyświetli wspomniane wykresy przedstawiające osiągniętą dokładność oraz stratę, zarówno dla zbioru walidacyjnego, jak i treningowego. Z uwagi na to, że niektóre czynności są dość czasochłonne (np. tworzenie cacha dla całego zbioru danych), wszystkie kluczowe akcje podejmowane przez aplikacje będą logowane, by użytkownik mógł wiedzieć, nad czym w danym momencie aplikacja pracuje. Oprócz tego w konsoli wypisywana będzie też architektura modelu i ostrzeżenia związane z przepełnianiem się pamięci karty graficznej. Wszystkie te informacje użytkownik będzie mógł włączyć lub wyłączyć za pomocą pliku konfiguracyjnego.

Ponadto, zostanie utworzony skrypt pozwalający na wygodne testowanie zapisanych wcześniej modeli dla dowolnych zdjęć. Skrypt będzie korzystał z innego pliku konfiguracyjnego, który pozwoli użytkownikowi na zdefiniowanie ścieżki do modelu do wczytania, wypisanie wyników na konsoli lub zdefiniowanie pliku do ich zapisu w wygodnym formacie, ale też umożliwi określenie poziomu szczegółowości wypisywanych rezultatów (zwięźle - tylko przewidywana emocja, bądź obszernie - prawdopodobieństwo dla każdej emocji) oraz zdefiniowanie informacji, które mają być logowane w konsoli. Ponadto, użytkownik podczas uruchomienia skryptu będzie mógł za pomocą parametrów uruchomienia przekazać mu ścieżki do plików lub katalogów zawierających zdjęcia. Skrypt załaduje wskazany w pliku konfiguracyjnym model, następnie przetworzy te z otrzymanych plików ze zdjęciami, które są w odpowiednim formacie, a ostatecznie zwróci pożądane przewidywania w zdefiniowanej przez użytkownika formie.

Implementacja i wyniki

Ten rozdział zawiera szczegółowe informacje na temat wykorzystywanych technologii, struktury aplikacji i skryptu, architektury wytrenowanych modeli oraz wyników podsumowujących możliwości wytrenowanych sieci neuronowych.

5.1 Opis technologii

W projekcie wykorzystano język Python 3 [7]. Python jest interpretowanym, wysokopoziomowym językiem ogólnego zastosowania, o czytelnej, klarownej i zwięzłej składni. Umożliwia programowanie w wielu paradygmatach. Paradygmaty obiektowy i funkcyjny okazały się szczególnie przydatne w trakcie rozwoju projektu. Python jest dynamicznie typowany oraz ma wbudowany garbage collector do zarządzania pamięcią. Często wykorzystywany jest jako język skryptowy. Został wybrany do tego projektu głównie z uwagi na to, że jest on też językiem szeroko wykorzystywanym w kontekście uczenia maszynowego, przez co posiada solidną bazę bibliotek i przydatnych narzędzi.

Jednym z nich jest framework Keras [4], który został wykorzystany do zaimplementowania modelu sieci neuronowej. Zdecydowano się na ten wybór ze względu na prostotę jego nauki i użytkowania, ale też z uwagi na to, że jest to framework powszechnie używany, co ułatwia wyszukiwanie potrzebnych informacji w internecie. Keras sam w sobie zapewnia tylko wysokopoziomowy interfejs służący do tworzenia modeli uczenia głębokiego. Pakiet ten nie implementuje żadnych niskopoziomowych operacji, takich jak przeprowadzanie działań na tensorach czy różniczkowanie. W tych kwestiach polega na wyspecjalizowanych i silnie zoptymalizowanych bibliotekach obsługujących tensory. Autorzy Keras nie wybrali do tego celu jednej konkretnej biblioteki, tym samym wiążąc z nią implementacje pakietu. Zamiast tego, problem ten został rozwiązany modułowo, a więc Keras może użyć jednej z kilku bibliotek jako swojego silnika bazowego. Są to na przykład TensorFlow od Google [9], Theano [10] czy Microsoft Cognitive Toolkit (CNTK) [5]. Kod napisany w Keras może być uruchomiony na dowolnej platformie wybranej spośród tych trzech, bez żadnej modyfikacji. Platformę można zmienić w dowolnym momencie pracy, co przydaje się, gdy któraś z nich okazuję się szybsza podczas rozwiązywania problemu.

W projekcie skorzystano z TensorFlow, czyli biblioteki opensource wspierającej trenowanie głębokich sieci neuronowych. TensorFlow pozwala na korzystanie z możliwości nie tylko procesorów CPU, ale też GPU. W tym pierwszym przypadku obudowuje niskopoziomową bibliotekę operacji tensorowych Eigen [1], natomiast w przypadku procesorów graficznych, TensorFlow stanowi interfejs dla wysoce zoptymalizowanej biblioteki uczenia głębokiego Nvidia CUDA Deep Neural Network (cuDNN) [6].

Kod uczenia głębokiego, z uwagi na ogrom operacji przeprowadzanych na tensorach, wykonuje się zdecydowanie szybciej na procesorach graficznych. Jest to szczególnie zauważalne w przypadku trenowania konwolucyjnych sieci neuronowych. Podczas pracy nad projektem, przejście z CPU Intel Core i5-7300HQ 2.50GHz na procesor graficzny Nvidia GeForce MX150 pozwoliło na skrócenia czasu trenowania modeli od 6 do 10 razy. Z tego powodu instalacja cuDNN jest zdecydowanie zalecana w przypadku dłuższej pracy z aplikacją.

5.2 Implementacja

5.2.1 Aplikacja do trenowania sieci

W celu możliwie jak największego ułatwienia i zautomatyzowania procesu trenowania sieci neuronowych została stworzona aplikacja w języku Python z wykorzystaniem frameworka Keras. Implementacja została oparta w głównej mierze na paradygmacie funkcyjnym, ale wykorzystuje również paradygmat obiektowy. Aplikacja zawiera plik konfiguracyjny `train_config.json` odpowiedzialny za sterowanie nią i procesem trenowania sieci. We wspomnianym pliku należy wyspecyfikować następujące parametry:

- `data_directory` - string, ścieżka do bazy danych ze zdjęciami. W katalogu wskazanym przez ten parametr muszą znaleźć się dwa foldery: `train` i `validation`, a każdy z nich musi przechowywać trzy foldery bezpośrednio zawierające zdjęcia w formacie `.jpg`: `positive`, `neutral` i `negative`;
- `models_directory` - string, ścieżka do folderu, w którym mają być zapisywane wytrenowane modele;
- `results_directory`- string, ścieżka do folderu, w którym mają być zapisywane wyniki trenowania sieci;
- `extracted_data_cache_directory` - string, ścieżka do folderu, w którym zapisywany jest cache przyśpieszający trenowania z użyciem transfer learningu;
- `use_pretrained_model` - boolean, jeśli true, buduje model z użyciem pretrenowanego modelu wyspecyfikowanego w `pretrained_models`, jeśli false, buduje standardowy model;
- `pretrained_model` - string, nazwa uprzednio trenowanego modelu z pakietu Keras, ma zastosowanie tylko, jeśli `use_pretrained_model` jest równe true;
- `epochs` - int, liczba iteracji trenowania;
- `batch_size` - int, liczba zdjęć, na których sieć jest trenowana jednocześnie;
- `picture_size` - int, składowa określająca kształt obrazu (rozdzielczość), do jakiego obrazy są sprowadzane w procesie wstępnej obróbki danych, `picture_shape = (picture_size, picture_size)`;
- `activation` - string, nazwa funkcji aktywacji wykorzystywanej przez sieć;
- `optimizer` - string, nazwa optymalizatora wykorzystywanego przez sieć;
- `optimizer_learning_rate` - float, wartość learning_rate optymalizatora wykorzystywanego przez sieć;
- `dropout_rate` - float, współczynnik porzucenia dla warstwy dropout umieszczonej na końcu części konwolucyjnej modelu, określa ułamek wszystkich wartości wejściowych do warstwy, które mają zostać porzucone;
- `kernel_size` - int, składowa określająca kształt jądra konwolucji, `kernel_shape` = (`kernel_size`, `kernel_size`);
- `pool_size` - int, składowa określająca wielkość łaty danych wykorzystywanej przez warstwy pooling, `pool_shape = (pool_size, pool_size)`;
- `data_augmentation` - boolean, jeśli true, augmentacja danych zostanie wykorzystana, jeśli false, augmentacja danych nie zostanie wykorzystana w trakcie treningu;
- `rotation_range` - int, zakres wyrażony w stopniach, o jakie zdjęcie może zostać obrócone podczas augmentacji danych;
- `width_shift_range` - float, zakres wyrażony w ułamku całkowitej szerokości, w jakiej zdjęcie może zostać przesunięte w kierunku poziomym podczas augmentacji danych;
- `height_shift_range` - float, zakres wyrażony w ułamku całkowitej wysokości, w jakiej zdjęcie może zostać przesunięte w kierunku pionowym podczas augmentacji danych;

- `brightness_range` - float, określa stopień, w jakim zdjęcie ma zostać rozjaśnione lub przyciemnione podczas augmentacji danych;
- `zoom_range` - float, zakres losowego przybliżenia lub oddalenia obrazu podczas augmentacji danych;
- `horizontal_flip` - boolean, jeśli true, zdjęcie może zostać odbite w osi poziomej podczas augmentacji danych, jeśli false, nie może zostać odbite;
- `vertical_flip` - boolean, jeśli true, zdjęcie może zostać odbite w osi pionowej podczas augmentacji danych, jeśli false, nie może zostać odbite;
- `fill_mode` - string, definiuje, jaki typ wypełnienia ma zostać użyty, gdy podczas augmentacji danym w danym miejscu obrazu nie można wygenerować żadnych pikseli, z uwagi na to, że na obrazie oryginalnym nie istnieje miejsce, które odpowiada temu miejscu. „nearest" oznacza, że w danym miejscu zostanie umieszczony piksel z najbliższej znanej pozycji, która mogła zostać zmapowana. Daje to efekt „rozciągnięcia" obrazu w tym miejscu. „reflect" oznacza lustrzane odbicie obrazu wzdłuż krawędzi ostatnich znanych pikseli. „wrap" oznacza zawijanie obrazu, czyli skopiowanie fragmentu z drugiego końca zdjęcia na to niezajęte pole. Parametr `fill_mode` jest wykorzystywany przy okazji użycia `width_shift`, `height_shift`, ale też `rotation_range`;
- `train_line_style` - string, formatuje linię na wykresie dokładności modelu na treningowym zbiorze danych w czasie kolejnych iteracji trenowania;
- `validation_line_style` - string, formatuje linię na wykresie dokładności modelu na walidacyjnym zbiorze danych w czasie kolejnych iteracji trenowania;
- `info_log` - boolean, jeśli true, wyświetla logi informujące o aktualnych czynnościach podejmowanych przez aplikację;
- `model_summary_log` - boolean, jeśli true, przed rozpoczęciem trenowania wyświetla szczegółową architekturę modelu, jaki został zbudowany,;
- `tensorflow_log` - boolean, jeśli true, wyświetla wszystkie logi standardowo wyświetlane przez TensorFlow.

Na podstawie tego pliku tworzony jest obiekt klasy `Config` znajdującej się w pliku `config.py` w pakiecie `config`. Obiekt ten jest dostępny w całej aplikacji, aby za jego pomocą można było wygodnie ją konfigurować i udostępniać dane wprowadzone przez użytkownika poprzez plik `train_config.json`.

Po uruchomieniu pliku `train_main.py`, odwołuje się on do pliku `launch.py` z pakietu `launch`, który jest odpowiedzialny za uruchamianie poszczególnych funkcji z pozostałych pakietów.

Nim aplikacja zbuduje sieć neuronową, przeprowadzana jest wstępna obróbka danych. Odpowiedzialny jest za to pakiet `data`, w którym można znaleźć pliki `preprocessing.py` i `pretrained_preprocessing.py`. Oba są odpowiedzialne za przetwarzanie danych przed wprowadzeniem ich do sieci neuronowej, jednak pierwszy odpowiada za modele zbudowane w całości przez aplikację, a drugi za te wykorzystujące bazę konwolucyjną z uprzednio trenowanego modelu. W pierwszym przypadku aplikacja wykorzysta dane z odpowiedniej ścieżki, uprzednio te dane augmentując lub nie, w zależności od ustawień obiektu konfiguracyjnego. W drugim przypadku wykorzysta ścieżkę do zcachowanych danych, a jeśli tej ścieżki nie odnajdzie, lub nie odnajdzie cacha zapisanego dla tej konkretnej bazy konwolucyjnej i sposobu preprocessingu danych, skorzysta ze ścieżki do bazy celem stworzenia odpowiedniego cacha. W pakiecie `data` w pliku `emotion.py` znajduje się klasa `Emotion`, reprezentująca jedną z trzech emocji. W tym samym pakiecie można znaleźć plik `data_set.py`, w którym znajduje się klasa `DataSet`, reprezentująca jeden z trzech zbiorów danych w bazie zdjęć.

W pakiecie `model`, na podstawie architektury wyspecyfikowanej w pliku `preparation.py` oraz przy użyciu obiektu konfiguracyjnego, przygotowywana jest sieć neuronowa. Po stworzeniu i wytrenowaniu tegoż modelu, zostaje on zapisany na dysku w formacie .h5. Oprócz tego, w plikach z rozszerzeniem `.npy` zapisywana jest również historia dokładności osiąganych podczas trenowania. Za obie powyższe funkcjonalności odpowiedzialne są metody znajdujące się w pliku `saving.py` w pakiecie `model`, wykorzystujące obiekt konfiguracyjny (odpowiednio parametry `models_directory` i `results_directory`) w celu zapisywania w żądanej przez użytkownika ścieżce. W tym samym pakiecie znajduje się również plik `visualization.py`, dzięki któremu

wyświetlane są wykresy prezentujące dokładność i stratę w kolejnych iteracjach treningu, zarówno dla zbioru walidacyjnego, jak i treningowego.

Ponadto, w pakiecie `utils` znajduje się plik `utils.py`, w którym przechowywane są użyteczne metody wykorzystywane w innych miejscach aplikacji. W tym samym pakiecie tworzony jest też obiekt klasy `Logger`, znajdującej się w pliku `logger.py`. Obiekt ten, podobnie jak obiekt klasy `Config`, jest dostępny w całej aplikacji, ale wykorzystywany jest do informowania użytkownika o aktualnym statusie i akcjach podejmowanych przez aplikacje. Z uwagi na to, że niektóre akcje mogą być czasochłonne, świadomość aktualnego stanu aplikacji pozwoli użytkownikowi uniknąć wrażenia, że aplikacja przestała odpowiadać. Ponadto, w konsoli wypisywana jest też architektura modelu i ostrzeżenia związane z przepełnianiem się pamięci karty graficznej. Wszystkie wyżej wymienione typy logów mogą być wyłączone za pomocą pliku konfiguracyjnego.

5.2.2 Skrypt rozpoznający emocje

W celu wygodnego uruchamiania wytrenowanych sieci dla dowolnych zdjęć i formatowania otrzymanych wyników został napisany skrypt wykorzystujący plik konfiguracyjny `predict_config.json`. Użytkownik podczas uruchomienia skryptu może przekazać mu ścieżki do plików lub katalogów zawierających obrazy. Skrypt załaduje wskazany w pliku konfiguracyjnym model, następnie przetworzy te otrzymane pliki, które są w odpowiednim formacie, a ostatecznie zwróci żądane przewidywania w zdefiniowanej przez użytkownika formie. Plik `predict_config.json` pozwala skonfigurować powyższe czynności za pomocą następujących parametrów odpowiedzialnych za sterowanie działaniem skryptu:

- `model_path` - string, ścieżka do wytrenowanego modelu zapisanego w formacie `.h5`;
- `output_file` - string lub null, jeśli string, wyniki zostaną zapisane do pliku z rozszerzeniem `.json` pod ścieżką zdefiniowaną przez ten parametr. Jeśli parametr nie zostanie podany lub zostanie do niego przypisany null, wyniki zamiast do pliku zostaną wyświetlone w konsoli;
- `verbose` - boolean, jeśli false, zostaną wypisane tylko informacje o klasie przewidywanej dla każdego obrazu, jeśli true, oprócz tego zostaną też wypisane informacje o prawdopodobieństwie przynależności obrazu do każdej z klas. Ma zastosowanie zarówno podczas wypisywania wyników w konsoli, jak i podczas zapisu do pliku;
- `info_log` - boolean, jeśli true, wyświetla logi informujące o aktualnych czynnościach podejmowanych przez skrypt;
- `tensorflow_log` - boolean, jeśli true, wyświetla wszystkie logi normalnie wyświetlane przez TensorFlow.

5.3 Model konwolucyjnej sieci neuronowej

Pierwszym zastosowanym podejściem jest zaimplementowanie od podstaw własnej konwolucyjnej sieci neuronowej i sprawdzenie, jakie wyniki jest ona w stanie osiągnąć. Przygotowywanie danych do pierwszych testów rozpoczęto od modyfikowania rozdzielczości zdjęć w taki sposób, żeby wszystkie były rozmiaru 150x150 pikseli. Obrazy podczas preprocessingu często sprowadza się do tej rozdzielczości z racji tego, że często okazuje się ona być złotym środkiem pomiędzy jakością obrazu, a jego wielkością (czyli jednocześnie ilością danych, jakie model musi przetworzyć). Taka rozdzielczość pozwala dość dokładnie określić co dzieje się na zdjęciu, np. człowiek nie ma żadnych problemów z rozpoznawaniem szczegółów na obrazie w takiej rozdzielczości. Jednocześnie zdjęcie składające się z większej liczny pikseli przekłada się na większy tensor, jaki musi być przetworzony przez pierwszą warstwę. Z kolei większy tensor wchodzący do pierwszej warstwy oznacza z reguły (jest to też zależne od architektury sieci), że kolejne tensory również będą większe. Wydłuża to czas treningu, a także czas działania modelu. Jest to zdecydowanie niepożądane w przypadku modeli wykorzystywanych na urządzeniach o ograniczonych zasobach, jak np. urządzenia mobilne.

Do pierwszego testu wykorzystano architekturę sieci składającą się z czterech bloków, w której każdy blok stanowi parę złożoną z dwóch warstw - warstwy konwolucyjnej o pewnej liczbie filtrów i pewnej funkcji aktywacji oraz warstwy max pooling. Liczby filtrów w każdej następnej warstwie konwolucyjnej wynoszą

odpowiednio 32, 64, 128 i 128. Celem takiej zwężającej się, ale coraz głębszej architektury jest ułatwienie sieci tworzenia hierarchii wzorców przestrzennych (patrz rozdział (4). Każda z warstw konwolucyjnych wykorzystuje jądra o standardowym rozmiarze równym 3x3. Każda z warstw max pooling również jest sparametryzowana w standardowy sposób, to znaczy używa okien konwolucyjnych o rozmiarze 2x2 i przesuwa je z krokiem równym 2. Można łatwo obliczyć, że ostatnia warstwa w tak zbudowanej bazie konwolucyjnej ma rozmiar (7, 7, 128). Do niej zostały dopięte jeszcze dwie warstwy gęste, wewnętrzna o 512 neuronach i wyjściowa o 3 neuronach reprezentujących 3 klasy wyjściowe. Z uwagi na to, że rozwiązywany jest problem klasyfikacji wieloklasowej, gdzie każdy obraz przypisywany jest do dokładnie jednej klasy, ostatnia warstwa korzysta z funkcji softmax, dzięki czemu suma wartości wyjściowych na tych trzech neuronach wynosi 1, co może być interpretowane jako prawdopodobieństwo, stopień pewności sieci o przynależności rozpatrywanego zdjęcia do danej klasy. Jako funkcja straty została wykorzystana kategorialna entropia krzyżowa.

Przeprowadzono 8 testów, z wykorzystaniem 4 różnych funkcji aktywacji: ReLU, tangensa hiperbolicznego (tanh), sigmoid i softmax. Dla każdej z tych funkcji model został skompilowany dwa razy za pomocą dwóch różnych optymalizatorów: RMSProp i Adam, oba parametryzowane learning rate równym 10^{-4}. RMSProp (Root Mean Square Propagation) został wybrany z uwagi na to, że jest to optymalizator często wykorzystywany w problemach klasyfikacji obrazów. Przewagą Adam (Adaptive Moment Estimation) z kolei jest to, że charakteryzuje się on krótszym czasem osiągania dobrych wyników, jednakże w niektórych problemach klasyfikacji obrazów może nie być w stanie osiągnąć rozwiązania optymalnego.

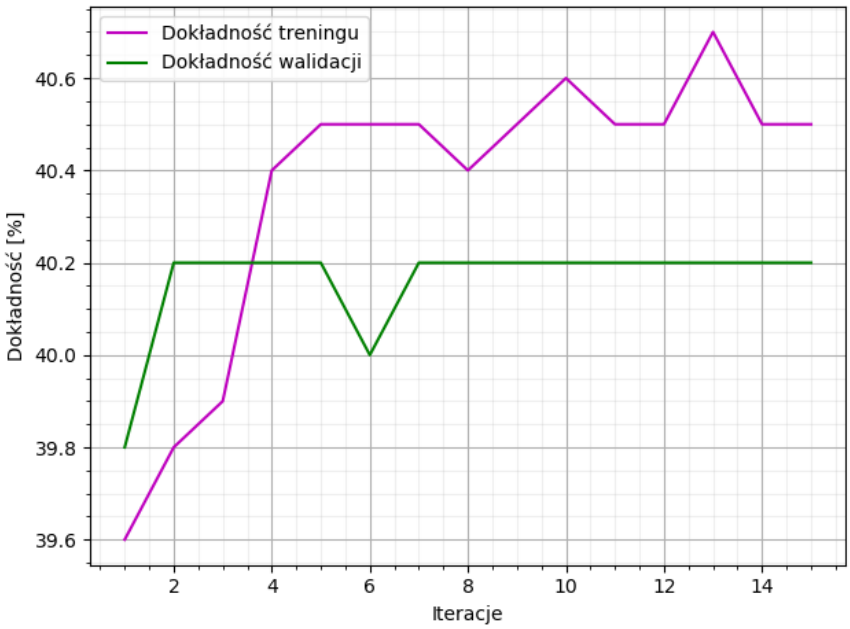
Dokładności osiągnięte przez sieci na walidacyjnym zbiorze danych po 15 iteracjach trenowania zostały przedstawione w tabeli 5.1.

Tablica 5.1: Dokładności osiągnięte po 15 iteracjach trenowania dla opisywanego modelu przetwarzającego zdjęcia w rozmiarze 150x150.

Sieć	RMSProp(10^{-4})	Adam(10^{-4})
ReLU	40.2%	**45.2%**
Tanh	40.2%	**47.7%**
Sigmoid	40.4%	40.2%
Softmax	40.2%	40.2%

W tabeli 5.1 pogrubioną czcionką zaznaczono wyniki wyraźnie lepsze od reszty. Już po pierwszym rzucie oka nasuwa się pytanie - dlaczego zdecydowana większość sieci osiągnęła identyczne lub niemal identyczne wyniki, w okolicach 40.2%? Wykresy przedstawiające historie dokładności osiąganej na walidacyjnym zbiorze danych (przykład dla funkcji sigmoid i optymalizatora ADAM przedstawiono na wykresie 5.1) wskazują na to, że większość z tych sieci osiągnęła tę wydajność już po pierwszej lub drugiej iteracji. Natomiast w przypadku zbioru testowego sieci z reguły bardzo szybko osiągały wyniki na poziomie 40.5%. Przyjrzenie się bazie danych pokazuje, że liczba zdjęć przedstawiających emocje pozytywne w zbiorze walidacyjnym wynosi 40.2% wszystkich zdjęć w tym zbiorze, natomiast w przypadku zbioru treningowego analogiczna wartość wynosi 40.5% (zobacz rozdział 4). W obu przypadkach jest to podzbiór większy od podzbiorów zdjęć emocji neutralnych czy negatywnych. Sugeruje to więc, że sieci wpadały w swego rodzaju lokalne maksima, zawsze zakładając, że na zdjęciu wyrażone są emocje pozytywne, ponieważ podczas treningu właśnie taka odpowiedź miała największe szanse na bycie odpowiedzią poprawną.

Rysunek 5.1: Historia wydajności przy zastosowaniu funkcji aktywacji sigmoid i optymalizatora Adam.



Po uruchomieniu tych sieci i przekazaniu do klasyfikacji całego zbioru testowego oraz walidacyjnego okazało się to prawdą - sieci niemal w każdym przypadku rozpoznawały emocję pozytywną. Co więcej, nierzadko robiły to z niemal stuprocentową pewnością. Sieci utykały więc w lokalnym maksimum.

Dalsze testy z wykorzystaniem optymalizatora RMSProp nie dały lepszych wyników- niezależnie od wybranej funkcji aktywacji i niezależnie od wykorzystanego learning rate (który przyjmował wartości od 10^{-1} do 10^{-7}), wszystkie sieci z użyciem tego optymalizatora uzyskały wyniki równe 40.2% lub nawet gorsze (w przypadku najniższych learning rate). Podobnie rzecz miała się w przypadku optymalizatora Adam w połączeniu z funkcjami aktywacji sigmoid lub softmax, żaden test nie wykazał przekroczenia „granicy" 40.2%.

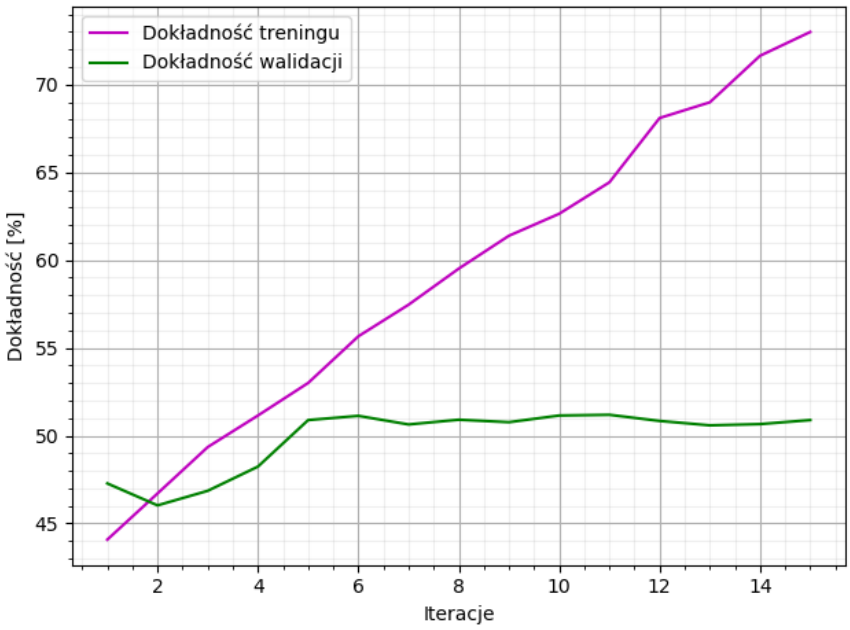
Jednak połączenie optymalizatora Adam z funkcjami aktywacji ReLu i tanh zasługuje na szczególną uwagę. Obie te architektury również zostały przetestowano dla różnych learning rate. Każdy test przeprowadzono dla 15 cykli trenowania. Wyniki tych testów zostały przedstawione w tabeli 5.2.

Tablica 5.2: Dokładności osiągnięte po 15 iteracjach trenowania dla modelu kompilowanego z użyciem optymalizatora Adam.

Learning rate	ReLU	Tanh
10^{-3}	40.2%	40.2%
10^{-4}	**45.2%**	**47.4%**
10^{-5}	**44.1%**	**50.9%**
10^{-6}	40.2%	**47.7%**
10^{-7}	40.2%	40.2%

We wszystkich przypadkach 15 epok trenowania wystarczyło, by sieci osiągnęły maksimum swojej dokładności na zbiorze walidacyjnym- z reguły osiągały ją już w okolicach od 5. do 8. iteracji. Na wykresie 5.2 przedstawiono przykładową historię treningu dla funkcji aktywacji tanh i optymalizatora Adam z learning rate wynoszącym 10^{-5}. Dalej można zaobserwować poprawę dokładności tylko dla zbioru treningowego- klasyczne nadmierne dopasowanie. By rozwiązać ten problem, w następnym podrozdziale zostanie wykorzystana augmentacja danych, by spróbować jeszcze ulepszyć te modele, które już do tej pory wypadły najlepiej, tj. np. model z funkcją aktywacji tanh dla learning rate równego 10^{-5} czy model wykorzystujący ReLU z learning rate równym 10^{-4}.

Rysunek 5.2: Historia wydajności przy zastosowaniu funkcji aktywacji tanh i optymalizatora Adam z learning rate wynoszącym 10^{-5}



Każdy z dotychczas wytrenowanych modeli (z uwagi na podobną architekturę) waży tyle samo, tj. 39.5 MB, dzięki czemu nie powinno być większych problemów z wykorzystywaniem takich sieci w aplikacjach przeznaczonych na różne rodzaje sprzętu, w tym urządzenia mobilne. W celu sprawdzenia, czy inna głębokość warstw poprawi wydajność modelu, kolejnym krokiem jest przeprowadzenie testów dla sieci składających się z innych liczb filtrów wykorzystywanych przez te warstwy. Przypomnijmy, że wyjściowa architektura składała się z czterech par warstw, z których każda złożona była z warstwy konwolucyjnej i max pooling. Liczba filtrów w warstwach konwolucyjnych wynosiła odpowiednio: 32, 64, 128, 128. Do kolejnych testów zbudowano 3 nowe sieci o następujących liczbach filtrów, sieć A: 16, 32, 64, 64; sieć B: 16, 32, 64, 128; oraz sieć C: 32, 64, 128, 256. Każdy model skompilowano kilka razy za pomocą optymalizatora Adam z learning rate na takim poziomie, które w poprzednich testach zapewniało najlepszą wydajność. Wykorzystano również funkcje aktywacji z najlepszymi do tej pory wynikami, tj. ReLu i tanh. Każdy model był trenowany przez 15 epok. Wyniki tych testów zamieszczono w tabeli 5.3.

Tablica 5.3: Dokładności osiągnięte po 15 iteracjach trenowania dla modeli o różnych liczb filtrów

F. aktywacji	L. rate	Model wyjściowy	Model A	Model B	Model C
ReLU	10^{-4}	45.2%	44.5%	45.0%	45.0%
ReLU	10^{-5}	44.1%	43.6%	43.6%	44.3%
ReLU	10^{-6}	40.2%	40.2%	40.2%	40.2%
Tanh	10^{-4}	47.4%	47.0%	47.2%	47.5%
Tanh	10^{-5}	**50.9%**	**50.4%**	**50.4%**	**50.7%**
Tanh	10^{-6}	47.7%	47.4%	47.3%	47.8%

Testy wykazują, że zwiększanie liczby filtrów nie zdaje egzaminu- model C osiągnął wyniki niemal identyczne z modelem wyjściowym, co czyni go gorszym rozwiązaniem- ten model, z uwagi na większą liczbę parametrów do wytrenowania, potrzebował więcej czasu na osiągnięcie pełnej wydajności, a jednocześnie zajmuje niemal dwukrotnie więcej miejsca na dysku (78 MB). Lżejsze modele o mniejszej liczbie filtrów, tj. A (19.1 MB) i B (37.9 MB) uzyskały wyniki podobne do siebie, a jednocześnie niewiele gorsze od wyników uzyskanych przez model wyjściowy. Jednak z uwagi na to, że są lżejsze, zdają się być warte dalszej analizy z uwzględnieniem augmentacji danych (szczególnie ponad dwukrotnie mniejszy model A).

Kolejne testy zostały przeprowadzone z wykorzystaniem jąder konwolucji o większym rozmiarze- 5x5. Wykonano je dla kilku modeli wybranych z poprzednich rozważań. W tabeli 5.4 zamieszczono porównanie wyników w przypadku użycia jąder konwolucji o rozmiarze 3x3 i 5x5.

Tablica 5.4: Dokładności osiągnięte po 15 iteracjach trenowania dla modeli o różnych wielkościach jąder konwolucji

F. aktywacji	L. rate	Model	Kernel 3x3	Kernel 5x5
Tanh	10^{-5}	Model wyjściowy	**50.9%**	49%
Tanh	10^{-5}	Model C	**50.7%**	48.4%
Tanh	10^{-6}	Model C	47.8%	46.1%
ReLU	10^{-4}	Model wyjściowy	45.2%	44.6%

W każdym z przetestowanych przypadków wykorzystanie mniejszego jądra konwolucji zapewnia nieznacznie lepsze wyniki. Z drugiej strony, większe jądro konwolucji sprawia, że wysokość i szerokość analizowanego tensora spada szybciej z warstwy na warstwę, przez co modele z jego wykorzystaniem są lżejsze. Dla przykładu, model wyjściowy o jądrze konwolucji rozmiaru 3x3 waży 39.5 MB, podczas gdy model o tej samej architekturze, ale z jądrem konwolucji rozmiaru 5x5 waży 26.4 MB.

Następnym krokiem jest przeprowadzenie testów z wykorzystaniem modeli zaprojektowanych do analizy zdjęć o rozdzielczości 224x224 pikseli. Większa rozdzielczość zdjęć wejściowych wydłuży proces trenowania i zwiększy wagę modelu, jednakże może również zwiększyć dokładność przewidywań, jeśli okaże się, że sieć jest w stanie dostrzec szczegóły obrazu, których nie dostrzegła na mniejszych zdjęciach.

Pierwszy zaprojektowany model złożony jest z czterech bloków warstw. W skład pierwszych dwóch bloków wchodzą po dwie warstwy konwolucyjne i jedna warstwa max pooling. Na ostatnie dwa bloki składają się 3 warstwy konwolucyjne i jedna warstwa max pooling. Liczby filtrów w każdej warstwie konwolucyjnej w danym bloku jest taka sama i wynosi odpowiednio 32, 64, 128 i 128 w kolejnych blokach. Tak samo jak w przypadku poprzednich modeli przetwarzających mniejsze zdjęcia, tak i teraz, każda z warstw max pooling jest sparametryzowana w standardowy sposób, to znaczy używa okien konwolucyjnych o rozmiarze 2x2 i przesuwa je z krokiem równym 2. Warstwy konwolucyjne również wykorzystują jądra o standardowym rozmiarze równym 3x3. Można obliczyć, że ostatnia warstwa w tak zbudowanej bazie konwolucyjnej ma rozmiar (8, 8, 128). Z uwagi na to, że jest to rozmiar podobny do tego z modelu dla mniejszych zdjęć, zostanie dopięty podobny klasyfikator, o dwóch warstwach gęstych, wewnętrznej o 512 neuronach i wyjściowej o trzech neuronach reprezentujących 3 klasy wyjściowe. Ostatnia warstwa będzie używać funkcji aktywacji softmax. Jako funkcja straty została wykorzystana kategorialna entropia krzyżowa.

Wykonano 8 testów, z wykorzystaniem 4 różnych funkcji aktywacji: ReLU, tanh, sigmoid i softmax. Dla każdej z tych funkcji model był kompilowany wielokrotnie za pomocą dwóch różnych optymalizatorów: RMSProp i Adam, oba z learning rate z przedziału od 10^{-7} do 10^{-3}. Testy przeprowadzone na tym modelu dały rezultaty bardzo podobne do tych przeprowadzonych na sieci dostosowanej do analizy zdjęć w rozdzielczości 150x150. Zarówno optymalizator RMSProp jak i funkcje sigmoid i softmax są nieodpowiednie do tego zadania, po raz kolejny modelom je wykorzystującym nie udało im się wyjść z lokalnego maksimum. Funkcje tanh i ReLU poradziły sobie lepiej, ale wciąż nie tak dobrze, jak w przypadku modelu wykorzystywanego do analizy zdjęć w rozdzielczości 150x150 pikseli. Najlepsze wyniki przedstawiono w tabeli 5.5. Nie uwzględniono tam sieci osiągających najsłabsze rezultaty.

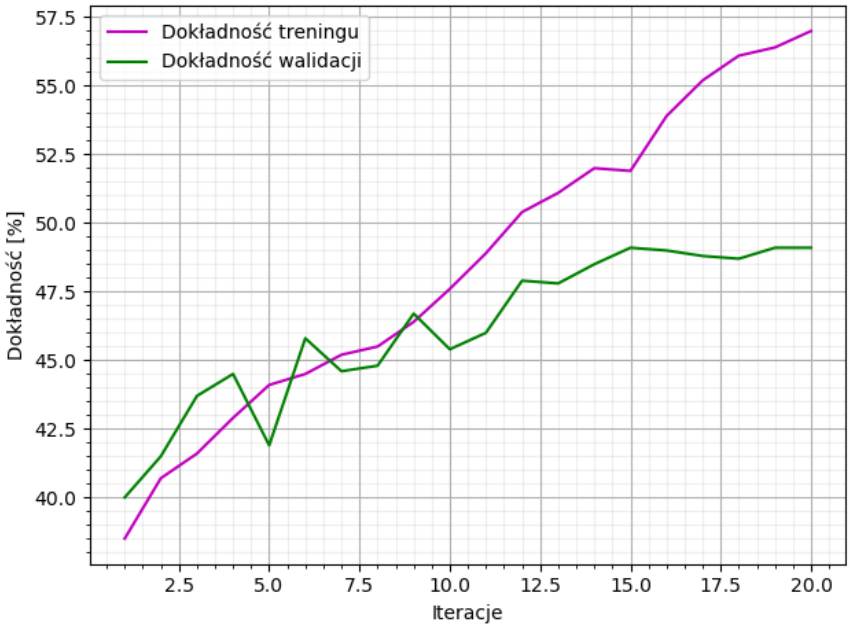
Tablica 5.5: Dokładności osiągnięte po 20 iteracjach trenowania dla modelu kompilowanego z użyciem optymalizatora Adam i analizującego zdjęcia w rozmiarze 224x224 pikseli

Learning rate	ReLU	Tanh
10^{-4}	44.7%	43.4%
10^{-5}	**47.7%**	**47.7%**
10^{-6}	40.2%	**49.1%**

Można zauważyć, że rozkład wyników jest bardzo podobny do przypadku, w którym analizowane były sieci dla zdjęć rozdzielczości 150x150. Trening sieci tym razem wymaga jednak większej liczby cykli, jednakże

w każdym przypadku 20 iteracji wystarczyło, by modele osiągnęły pełnie swoich możliwości. Na wykresie 5.3 przedstawiono historie treningu modelu, który osiągnął najlepsze rezultaty, tj. wykorzystującego funkcję aktywacji tanh i optymalizator Adam z learning rate wynoszącym 10^{-6}. Można zaobserwować, że od pewnego momentu wydajność wzrasta już tylko dla zbioru treningowego. Ten problem można częściowo rozwiązać za pomocą augmentacji danych, jednakże zdaje się, że wcześniej rozpatrywane modele były bardziej warte uwagi w tej kwestii, szczególnie biorąc pod uwagę, że nowe modele zajmują zdecydowanie więcej miejsca na dysku, ponieważ każdy z nich waży 58.1 MB.

Rysunek 5.3: Historia wydajności przy zastosowaniu funkcji aktywacji tanh i optymalizatora Adam z learning rate wynoszącym 10^{-6}



Przed wykonaniem testów z wykorzystaniem augmentacji danych, zostaną jeszcze przeprowadzone modyfikacje liczby filtrów wykorzystywanych przez warstwy w celu sprawdzenia, czy inna głębokość warstw poprawi wydajność modelu. Przypomnijmy, że wyjściowa architektura składała się z czterech bloków warstw, z których każdy złożony był z dwóch albo trzech warstw konwolucyjnych i jednej warstwy max pooling. Liczba filtrów w warstwach konwolucyjnych tego samego bloku była taka sama i wynosiła odpowiednio: 32, 64, 128, 128 dla kolejnych bloków. Do kolejnych testów zbudowano 3 nowe sieci o następujących liczbach filtrów, sieć A: 16, 32, 64, 128; sieć B: 32, 64, 128, 256; sieć C: 64, 128, 256, 256. Wyniki wydajności osiągniętych przez nowe modele kompilowane za pomocą optymalizatora Adam zostały przedstawione w tabeli 5.6.

Tablica 5.6: Dokładności osiągnięte po 20 iteracjach trenowania dla modeli kompilowanych z użyciem optymalizatora Adam

F. aktywacji	L. rate	Model wyjściowy	Model A	Model B	Model C
ReLU	10^{-4}	44.7%	42.4%	41.5%	41.7%
ReLU	10^{-5}	47.7%	45.0%	**49.3**%	**48.2**%
ReLU	10^{-6}	40.2%	40.2%	40.2%	40.2%
Tanh	10^{-4}	43.4%	42.1%	43.2%	43.4%
Tanh	10^{-5}	47.7%	45.8%	44.7%	47.8%
Tanh	10^{-6}	**49.1**%	47.5%	**48.2**%	**49.2**%

W tym przypadku zwiększenie liczby filtrów pomogło w niektórych konfiguracjach- widać to wyraźnie dla funkcji aktywacji ReLU i learning rate o wartości 10^{-5}. Wydajność w tym przypadku dla modelu B wzrosła

o 1.6 p.p., jednakże jest on cięższy, bo waży aż 117 MB, co w zasadzie wyklucza go z dalszych rozważań. W pozostałych przypadkach modele B i C w większości osiągnęły wyniki porównywalne do tych osiągniętych przez model wyjściowy, ale z uwagi na, że są zdecydowanie cięższe (model C waży aż 136 MB) nie będą dalej rozważane. Z kolei modele o mniejszej głębokości warstw radziły sobie widocznie słabiej, najwidoczniej mała liczba filtrów nie pozwoliła wykształcić tak dobrej hierarchii cech.

5.4 Augmentacja danych

Kolejny etap ulepszania modeli opiera się na wykorzystaniu augmentacji danych. W rozdziale 4. zostały opisane modyfikacje obrazów, które zostaną zastosowane w celu wzbogacania zbioru zdjęć. Z uwagi na zdecydowanie większą ilość czasu potrzebną do trenowania sieci z wykorzystaniem tej techniki, testy zostaną przeprowadzone na tych modelach, które w poprzednim podrozdziale uzyskały najlepszą dokładność na walidacyjnym zbiorze danych.

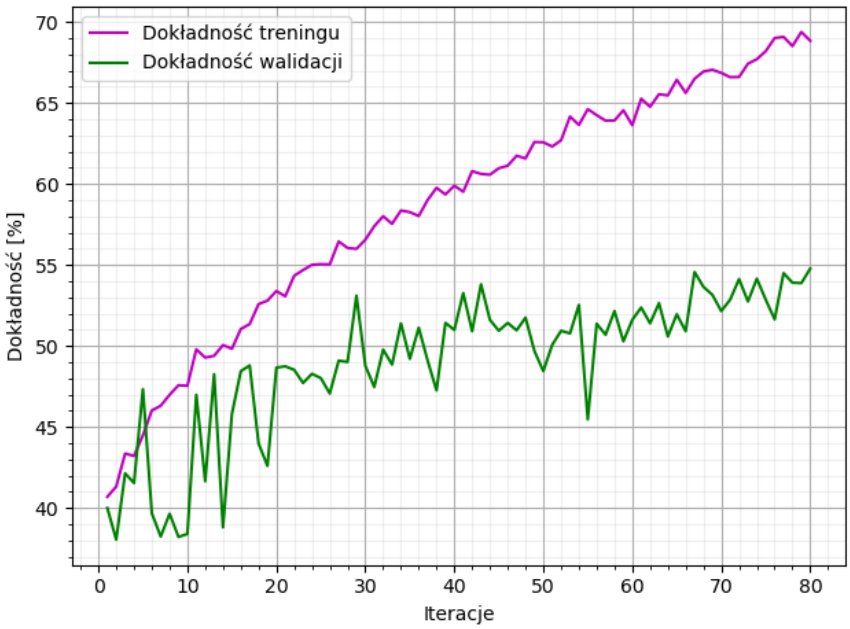
W pierwszym teście augmentacja zostanie przeprowadzona z następującymi argumentami opisanymi na początku tego rozdziału: rotation_range = 40, width_shift = 0.2, height_shift = 0.2, brightness_range = 0.2, zoom_range = 0.2, horizontal_flip = true, vertical_flip = false, fill_mode =„nearest” i dropout_rate = 0.5. Jest to często wykorzystywana konfiguracja. Tabela 5.7 przestawia porównanie wydajności sieci, które w poprzednim podrozdziale dawały najlepsze wyniki, tym razem trenowane w 80 iteracjach z wykorzystaniem augmentacji i bez niej. Wszystkie z testowanych modeli przetwarzały zdjęcia w rozmiarze 150x150 i używały optymalizatora Adam.

Tablica 5.7: Dokładności osiągnięte po 80 iteracjach trenowania dla modeli, które osiągnęły najlepsze rezultaty bez augmentacji danych

F. aktywacji	L. rate	Model	Bez augmentacji	Z augmentacją
Tanh	10^{-5}	Model wyjściowy	50.9%	52.4%
Tanh	10^{-5}	Model C	50.7%	52.7%
Tanh	10^{-6}	Model C	47.8%	50.1%
ReLU	10^{-4}	Model wyjściowy	45.2%	**54.8**%

Zgodnie z oczekiwaniami, augmentacja danych poprawiła wydajność w przypadku każdej sieci, jednak na szczególną uwagę zasługuje tutaj sieć wykorzystująca funkcję ReLU w charakterze funkcji aktywacji. W tym przypadku wydajność osiągnęła 54.8% na walidacyjnym zbiorze danych, co jest najlepszym wynikiem do tej pory. W stosunku do tego samego modelu nie wykorzystującego augmentacji danych, wydajność poprawiła się o 9.6 p.p, czyli aż o 21% w skali względnej. Historię trenowania tej sieci przedstawiono na wykresie 5.4.

Rysunek 5.4: Historia wydajności przy zastosowaniu funkcji aktywacji ReLU i augmentacji danych.



Można zauważyć, że wykresy wykorzystujące augmentacje danych zachowują się inaczej niż te, które tego nie robią. Te ostatnie zdecydowanie szybciej osiągają pełną wydajność dla zbioru treningowego (przetrenowanie), a jeszcze szybciej dla zbioru walidacyjnego (brak rzeczywistego treningu z uwagi na zbytnie dostosowanie do zbioru treningowego). W przypadku modelu wykorzystującego augmentacje danych linie zdają się być bardziej skorelowane ze sobą. Mimo tego, że wartości nie są podobne do siebie, to dokładność dla zbioru walidacyjnego rośnie stale wraz z dokładnością dla zbioru treningowego. Na wykresie można zobaczyć, że mimo wykonania 80 cyklu trenowania, dokładność treningu nawet nie zbliżyła się do 100%, chociaż widać pewne oznaki przetrenowania (wartości coraz bardziej odbiegają od tych dla zbioru walidacyjnego). Jednakże dokładność walidacji stale rośnie i wydaję się prawdopodobne, że np. kolejne kilkadziesiąt iteracji pozwoliłoby zwiększyć wydajność jeszcze o kilka punktów procentowych. Wykorzystanie augmentacji danych nie ma wpływu na ilość miejsca, jaką sieć zajmuje na dysku, ponieważ augmentacja danych modyfikuje tylko dane treningowe, w żaden sposób nie wpływając na kształt architektury sieci. Z tego powodu model, którego historia trenowania została przedstawiona, waży 39.5 MB, czyli dokładnie tyle samo, co odpowiadający mu model opisywany w poprzednim rozdziale.

W przypadku modeli wykorzystujących funkcję tanh w charakterze funkcji aktywacji, ostateczna dokładność walidacji z użyciem augmentacji danych wzrosła w najlepszym przypadku tylko o około 2 p.p. Może to częściowo wynikać z faktu, że funkcja tanh już wcześniej osiągała lepszą wydajność, więc niewątpliwie ciężej było jeszcze bardziej ją zwiększyć. Jednakże trzeba też zauważyć, że funkcja ReLU, mimo tego, że w analizowanym zestawieniu bez augmentacji danych osiągała najgorsze wyniki, po użyciu augmentacji stała się najbardziej wydajna.

Z tego powodu, w kolejnych testach zostaną wzięte pod uwagę inne modele wykorzystujące funkcję ReLU. Testy zostaną przeprowadzone dla różnych wartości parametru `dropout_rate`. Ma to na celu wyrównanie dokładności dla zbioru treningowego i walidacyjnego, ponieważ w poprzednim teście wartości te coraz bardziej różniły się od siebie w kolejnych iteracjach. Wyniki zostały przedstawione w tabeli 5.8.

Tablica 5.8: Dokładności osiągnięte po 80 iteracjach trenowania z wykorzystaniem różnych wartości współczynnika porzucenia.

F. akt.	L. rate	Model	Rozmiar	Bez aug.	Z augmentacją		
					Dropout 0.5	Dropout 0.35	Dropout 0.6
ReLU	10^{-4}	M. wyjściowy	150x150	45.2%	**54.8%**	51.9%	40.2%
ReLU	10^{-5}	M. wyjściowy	150x150	44.1%	**51.5%**	50.6%	40.2%
ReLU	10^{-4}	M. wyjściowy	224x224	44.7%	**50.2%**	46.9%	40.2%
ReLU	10^{-5}	Model B	224x224	49.3%	**54.6%**	49.6%	40.2%

Okazuje się, że podniesienie współczynnika porzucenia do 0.6 całkowicie uniemożliwia sieci wykształcenie przestrzennej hierarchii wzorów. Lepiej sprawdza się obniżenie `dropout_rate`, ale tutaj z kolei ponownie pojawia się problem nadmiernego dopasowania- dokładność treningowa rośnie szybciej, niż w przypadku `dropout_rate` = 0.5, osiągając ostatecznie wartości bliskie 100%, jednocześnie uniemożliwiając tym samym wykształcenie cech, które przydałyby się podczas analizy zdjęć nie wchodzących w skład zbioru treningowego. Obniżenie wartości `dropout_rate` wypada szczególnie słabo w przypadku sieci analizujących zdjęcia w rozmiarze 224x224.

W ostatnim teście została użyta sieć, która do tej pory osiągała najlepsze wyniki, tj. model wyjściowy analizujący zdjęcia w rozdzielczości 150x150 z wykorzystaniem ReLU w charakterze funkcji aktywacji oraz optymalizatora Adam o wartości learning rate wynoszącej 10^{-4}. Test zakłada modyfikowanie parametrów augmentacji. Wyniki zostały przedstawione w tabeli 5.9.

Tablica 5.9: Dokładności osiągnięte po 80 iteracjach trenowania z wykorzystaniem różnych współczynników augmentacji.

rotation_r.	width_s.	height_s.	brightness_r.	zoom_r.	horizontal_f.	fill_m.	Dokładność
40	0.2	0.2	0.2	0.2	true	„nearest”	**54.8**%
30	0.15	0.15	0.15	0.15	false	„nearest”	54.4%
50	0.25	0.25	0.25	0.25	true	„nearest”	46.9%
0	0.25	0.25	0.25	0.25	true	„nearest”	48.1%
40	0.2	0.2	0.2	0.2	true	„reflect”	**55.1**%
40	0.2	0.2	0.2	0.2	true	„wrap”	**54.9**%

Testy wykazały, że w przypadku obniżenia wartości argumentów, sytuacja ma się bardzo podobnie, jak w przypadku zastosowania argumentów wyjściowych- wyniki są tylko nieznacznie słabsze. Z kolei zwiększenie „siły” augmentacji wpływa negatywnie na zdolność uczenia się modelu- co prawda nie występuje problem przetrenowania, ale mimo 80 iteracji nauki, sieci nie udało się wykształcić cech, które pozwoliłyby na rozpoznawanie emocji na walidacyjnym zbiorze danych. Sytuacja ma się podobnie, kiedy przy tych samych argumentach zdjęcie przestanie być obracane (`rotation_range` = 0). Zastosowanie trybu wypełnienia nieznanych pikseli (`fill_mode`) innego niż „wrap” daje za to nieznacznie lepsze efekty- wynika to najprawdopodobniej z faktu, że w miejscu, na które nie można było zmapować żadnych pikseli, nie znajduje się już „rozciągnięta” krawędź obrazu, a inny jego fragment. Szczególną poprawę widać dla trybu `fill_mode` = „reflect”, gdyż przy jego użyciu zdjęcie jest w tym miejscu odbite w lustrze, co widocznie jest bardziej naturalne, niż krawędź, na której zbiegają się dwa niepołączone ze sobą fragmenty obrazu (`fill_mode` = „wrap”).

Wykres 5.5 przedstawia historie trenowania dla modelu, który osiągnął najwyższą dokładność na walidacyjnym zbiorze danych. Jest to model przetwarzający zdjęcia w rozdzielczości 150x150, wykorzystujący ReLU w charakterze funkcji aktywacji oraz optymalizator Adam z learning rate wynoszącym 10^{-4}, trenowany z użyciem następującej konfiguracji augmentacji danych: rotation_range = 40, width_shift = 0.2, height_shift = 0.2, brightness_range = 0.2, zoom_range = 0.2, horizontal_flip = true, vertical_flip = false, fill_mode =„reflect” oraz dropout_rate = 0.5. Można zauważyć, że w kolejnych iteracjach sieć coraz wolniej zwiększa wydajność, ale za to coraz bardziej zmniejsza amplitudę wahań, jednocześnie coraz wyraźniej zarysowując próg dokładności, którego nie jest w stanie przekroczyć. Ostatecznie dodatkowe 120 iteracji pozwoliło nieznacznie zwiększyć wydajność, osiągając 55.5% na walidacyjnym zbiorze danych i jednocześnie zachowując rozmiar na dysku nieprzekraczający 40 MB, co umożliwia lokalne przechowywanie modelu nawet w przypadku aplikacji

mobilnych.

Rysunek 5.5: Historia dokładności w trakcie 200 iteracji uczenia dla najwydajniejszej architektury sieci trenowanej z użyciem augmentacji danych

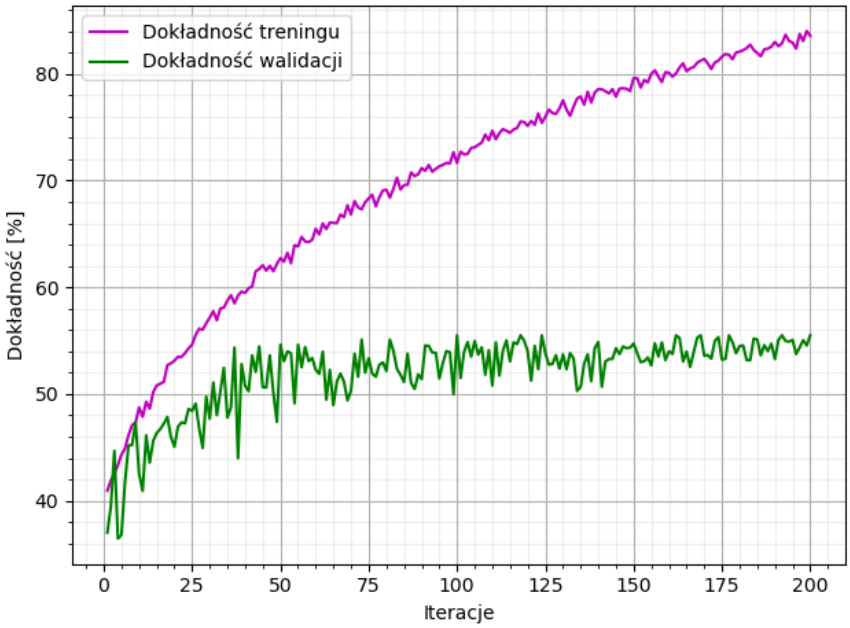


Tabela 5.10 przestawia szczegółowe wyniki osiągnięte przez tę sieć na konkretnych podzbiorach walidacyjnego zbioru danych. Tabela 5.11 przedstawia z kolei te same dane wyrażone w wartościach mówiących, jaki procent danych z danego podzbioru został rozpoznany jako wskazana wartość. Dla przykładu, emocje pozytywne zostały rozpoznane na 11.9% zdjęć przedstawiających emocje negatywne. Wartości pokazują, że sieć w przypadku każdej emocji zawsze miała największe szanse na wskazanie właśnie tej właściwej. Model wykazał się szczególną dokładnością podczas analizy zdjęć negatywnych- w 68.2% przypadkach analizy takich zdjęć uzyskano poprawny wynik. W tym miejscu warto przypomnieć, że model wytrenowany przez [14], który osiągał najlepszą średnią wydajność dla wszystkich emocji, osiągnął tylko 31.03% podczas analizy emocji negatywnych, czyli aż o 37.16 p.p. mniej. Ta emocja była najczęściej udzielaną odpowiedzią w przypadku analizowanego testu- ogólnie padła w 39.4% przypadków. Była też drugą najczęściej udzielaną odpowiedzią dla podzbiorów zdjęć pozytywnych i neutralnych, odpowiednio 25.4% i 31.4%. Model najsłabiej poradził sobie z rozpoznawaniem emocji neutralnych- rozpoznał je poprawnie tylko w 42.8% przypadków analizy zdjęć przedstawiających te emocje. Może to mieć też związek z tym, że była to najrzadziej udzielana odpowiedź- została udzielona tylko w 26.4% przypadków. Najczęściej popełnianym błędem pod względem ilościowym było rozpoznawanie zdjęcia pozytywnego jako negatywnego, taka błędna odpowiedź padła 444 razy, a więc w 25.4% przypadków analizy zdjęć pozytywnych. Z kolei najczęściej popełnianym błędem w stosunku do wielkości analizowanego podzbioru, było rozpoznawania negatywnej emocji w trakcie analizy podzbioru zdjęć neutralnych- tak było w 31.3% przypadków dla analizy tego podzbioru, a więc ogólnie 430 razy.

Tablica 5.10: Liczba rozpoznanych emocji dla konkretnych podzbiorów zdjęć.

Liczba zdjęć: 4346		Przewidywania			Suma
		Pozytywne	Neutralne	Negatywne	
Rzeczywiste	Pozytywne	**986**	317	444	1747
	Neutralne	353	**585**	430	1368
	Negatywne	147	244	**840**	1231
Suma		1486	1146	1714	**2411**

Tablica 5.11: Procentowy stosunek rozpoznanych emocji dla konkretnych podzbiorów danych.

		Przewidywania			Suma
		Pozytywne	Neutralne	Negatywne	
Rzeczywiste	Pozytywne	**56.4%**	18.1%	25.4%	40.2%
	Neutralne	25.8%	**42.8%**	31.4%	31.5%
	Negatywne	11.9%	19.8%	**68.2%**	28.3%
Suma		34.2%	26.4%	39.4%	**55.5%**

5.5 Rozszerzenie uprzednio trenowanej sieci neuronowej

Jako ostatnie przetestowane zostanie podejście wykorzystujące transfer learning, tj. oparte na użyciu modelu wytrenowanego wcześniej na bardziej ogólnym zbiorze danych zawierającym większą liczbę klas.

Framework Keras zawiera wbudowanych kilkanaście takich modeli. Wszystkie są dostępne w pakiecie *keras.applications*. Na początku wyselekcjonowane zostaną te, które cechują się największą dokładnością w problemach zbliżonych do rozpatrywanego w tej pracy. Każdy z nich został wytrenowany wcześniej na zbiorze ImageNet [3]. ImageNet to baza zdjęć zawierająca ponad 14 milionów obrazów podzielonych na tysiące klas. Z tego powodu sieci na niej trenowane cechują się dobrymi, tj. bardzo ogólnymi przestrzennymi hierarchiami cech. Do testów wytypowano 10 modeli, były to: MobileNetV2, MobileNet, NASNetMobile, VGG16, Xception, InceptionV3, ResNet50, ResNet50V2, ResNet101V2 i InceptionResNetV2.

W pierwszym teście od każdego z nich został odpięty gęsto połączony klasyfikator, a podpięty został nowy, wykorzystujący metaparametry, które najlepiej sprawdziły się w poprzednich testach. Nowy klasyfikator składał się z trzech warstw: warstwy porzucenia ze współczynnikiem równym 0.5, warstwy gęstej o 256 neuronach z funkcją aktywacji ReLU oraz warstwy gęstej o 3 neuronach, z funkcją aktywacji softmax. W procesie preprocessingu danych wszystkie zdjęcia, zarówno ze zbioru treningowego, jak i walidacyjnego, zostały skompresowane do takich samych rozmiarów, mianowicie do 150x150 pikseli. Jako funkcja straty została wykorzystana kategorialna entropia krzyżowa. Dla każdej sieci zostały przeprowadzone dwa testy, każdy z nich z wykorzystaniem innego optymalizatora. Do pierwszego testu wybrano RMSProp z learning rate równym $2 \cdot 10^{-5}$. W drugim teście wykorzystano optymalizator Adam z learning rate wynoszącym $2 \cdot 10^{-4}$.

Wyniki modeli osiągnięte na walidacyjnym zbiorze danych po przeprowadzeniu 12 cykli trenowania zostały przedstawione w tabeli 5.12.

Tablica 5.12: Wyniki osiągnięte przez modele wykorzystujące transfer learning

Sieć	RMSProp($2 \cdot 10^{-5}$)	Adam($2 \cdot 10^{-4}$)
MobileNetV2	**53.6%**	**54.8%**
MobileNet	**52.8%**	**53.5%**
NASNetMobile	41.9%	42.4%
VGG16	**56.1%**	43.0%
Xception	44.3%	**55.1%**
InceptionV3	42.4%	46.3%
ResNet50	**61.1%**	**52.4%**
ResNet50V2	40.1%	46.9%
ResNet101V2	43.9%	43.6%
InceptionResNetV2	41.2%	43.9%

Do dalszych testów zostały wybrane tylko te pary modeli i optymalizatorów, których wyniki przekroczyły 50% (zaznaczone pogrubioną czcionką). Szczególną uwagę należy zwrócić na model ResNet50, który osiągnął bardzo dobre wyniki, szczególnie przy użyciu optymalizatora RMSprop. 61.1% to wynik mniejszy tylko o 6.5 p.p. od najlepszego wyniku osiągniętego na tej bazie danych i jednocześnie lepszy od wszystkich wyników uzyskanych w poprzednich testach, ale należy pamiętać, że ResNet50 w tym przypadku miał tylko 12 epok trenowania do dyspozycji. Sieci MobileNet i MobileNetV2 poradziły sobie zadowalająco dobrze, by warto było je dopuścić do kolejnych testów. Sieć VGG16 osiągnęła wynik nawet lepszy od nich, ale tylko przy użyciu

optymalizatora RMSprop, dlatego tylko on został wykorzystany w następnych testach. Z kolei sieć Xception osiągnęła lepszy wynik z wykorzystaniem optymalizatora Adam. Można też zauważyć, że optymalizator Adam okazał się lepszy w siedmiu na dziesięć przypadków.

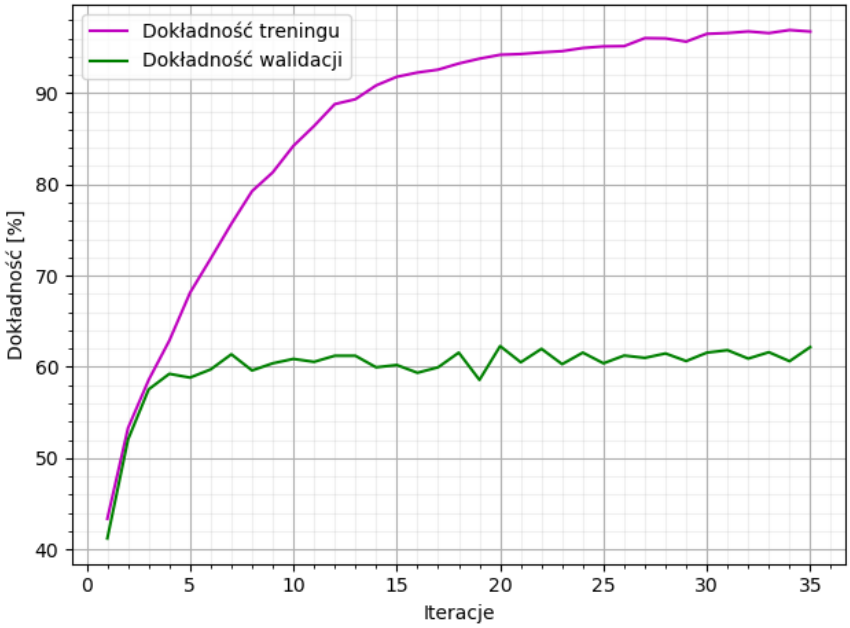
W kolejnym podejściu zmodyfikowano architekturę gęsto połączonego klasyfikatora. Środkowa gęsta warstwa została czterokrotnie rozszerzona, do 1024 neuronów. Zmodyfikowany został też optymalizator. Learning rate dla optymalizatora RMSProp został ustawiony na $2 \cdot 10^{-4}$, natomiast w przypadku Adam- $2 \cdot 10^{-3}$. Tym razem zwiększono liczbę cykli trenowania. Dokładności osiągnięte na walidacyjnym zbiorze danych po 35 iteracjach zostały przedstawione w tabeli 5.13.

Tablica 5.13: Wyniki osiągnięte przez najlepsze modele wykorzystujące transfer learning.

Sieć	RMSProp($2 \cdot 10^{-4}$)	Adam($2 \cdot 10^{-3}$)
MobileNetV2	59.7%	58.5%
MobileNet	56.7%	56.5%
VGG16	**62.3%**	N/A
Xception	N/A	58.5%
ResNet50	**62.5%**	59.4%

Tabela 5.13 jasno wskazuje, że sieci VGG16 i ResNet50 z optymalizatorem RMSProp osiągnęły wyniki wyraźnie lepsze od pozostałych par modeli i optymalizatorów. Jednocześnie wskazane sieci osiągały maksimum swojej wydajności w okolicach 7. epoki trenowania. Na wykresie 5.6 przedstawiono przykład dla modelu VGG16 wykorzystującego RMSProp z learning rate wynoszącym $2 \cdot 10^{-4}$. Tak szybkie osiąganie maksimum wydajności jasno wskazuje na brak potrzeby zwiększania liczby epok trenowania, a zamiast tego sugeruje konieczność skupienia się na poszukiwaniu lepszej konfiguracji gęsto zakończonego optymalizatora.

Rysunek 5.6: Historia dokładności w trakcie 35 iteracji uczenia dla modelu VGG16 i optymalizatora RMSProp



Kolejne testy skupiały się na różnych próbach modyfikacji gęsto połączonych klasyfikatorów modeli VGG16 i ResNet50. Punktem wyjścia był klasyfikator użyty w ostatnim teście, za pomocą którego uzyskano najlepsze wydajności (z optymalizatorem RMSProp i learning rate równym $2 \cdot 10^{-4}$). Pogrubioną czcionką zaznaczono sytuacje, w których udało się poprawić wyniki. Każdy test został przeprowadzony na 40 epokach treningu, a ich wyniki zamieszczono w tabeli 5.14.

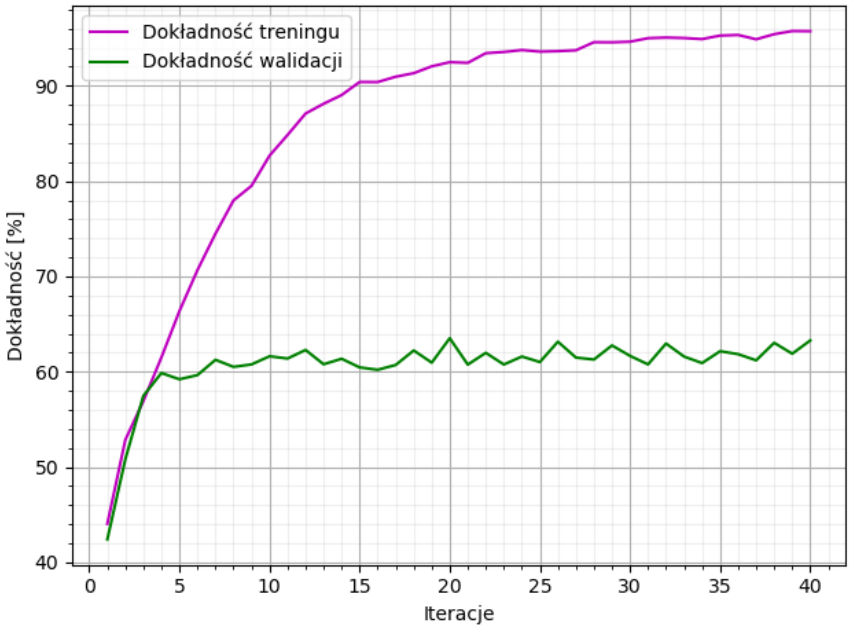
Tablica 5.14: Wyniki osiągnięte poprzez różne wybrane modyfikacja modelu z optymalizatorem RMSProp.

Modyfikacja	VGG16	ResNet50
Funkcja aktywacji tanh	49.9%	41.3%
Jedna warstwa 2048 neuronów	61.2%	61.3%
Dwie warstwy po 1024 neuronów	**62.6%**	**62.8%**
Dwie warstwy po 512 neuronów	**63%**	**63.6%**
Learning rate- $2 \cdot 10^{-3}$	52.1%	54.9%
Learning rate- $2 \cdot 10^{-5}$	**62.7%**	62.2%

Najlepszy wynik osiągnął model ResNet50 dla dwóch warstw złożonych z 512 neuronów, zdecydowanie przekraczając 63%. Ogólne wyniki wskazują na to, że dodanie kolejnej warstwy gęsto połączonego klasyfikatora może pozytywnie wpłynąć na wydajność modelu, szczególnie jeśli nowo dodana warstwa nie jest zbyt szeroka. Ponadto, dla modelu VGG16 zauważono wzrost wydajności w przypadku zastosowania optymalizatora o niższym learning rate, choć liczba iteracji potrzebnych do osiągnięcia takich wyników wynosi wtedy około 2-3 razy więcej, niż dla pozostałych rozpatrywanych przypadków.

Z tego powodu, w kolejnym teście użyto modelu VGG16 zakończonego gęsto połączonym klasyfikatorem o dwóch warstwach po 512 neuronów, z optymalizatorem RMSProp o learning rate wynoszącym $2 \cdot 10^{-5}$. Model osiągnął wydajność na poziomie **63.4%**, a historię jego trenowania przedstawiono na wykresie 5.7. Wynika z niego, że 40 iteracji nie było potrzebnych, ponieważ model osiągnął pełną wydajność już w połowie treningu. Sieć ResNet50 z poprzedniego testu wykorzystująca taki sam klasyfikator, ale z learning rate wynoszącym $2 \cdot 10^{-4}$, uzyskała wynik lepszy o 0.2 p.p., jednakże składa się ona aż z 50 013 699 możliwych do trenowania parametrów. Dla porównania, model VGG16 jest zbudowany tylko z 19 173 699 parametrów. Ponadto, ResNet50 to sieć na tyle duża, że do przeprowadzania pojedynczej sekwencji obliczeń, nawet dla bardzo małego wsadu danych, potrzebna jest pamięć większa niż ta, jaką jest w stanie zapewnić karta graficzna Nvidia GeForce MX150. Z tego powodu obliczenia musiały być wykonywane na CPU, co w połączeniu z dużą większą liczbą parametrów koniecznych do wytrenowania sprawiło, że trening dla modelu ResNet50 zajmował około 35-40 razy więcej czasu, niż w przypadku VGG16.

Rysunek 5.7: Historia dokładności w trakcie 40 iteracji uczenia dla modelu VGG16 zakończonego dwiema warstwami gęstymi.



Niewielki rozmiar sieci VGG16 niesie ze sobą jeszcze jedną zaletę. Cały model oparty na tej bazie konwo-

lucyjnej, wraz z wytrenowanym gęstym klasyfikatorem, po zapisaniu na dysku waży 73.1 MB. W przypadku modelu ResNet50 jest to około 2.6 razy więcej- 191 MB. Wykorzystanie mniejszego modelu skraca czas jego ładowania z dysku oraz pozwala na szybszą klasyfikację- obie te zalety mają bardzo duże znaczenie dla użytkownika końcowego. Niewielka waga modelu potencjalnie pozwala na wykorzystanie go na więcej sposobów. Dla przykładu, model o takim rozmiarze może być bez większych problemów zainstalowany na urządzeniu mobilnym- czyli sprzęcie, na którym z perspektywy użytkownika, możliwość rozpoznawania emocji na zdjęciach jest wyjątkowo pożądana.

Podsumowując, ze wszystkich przedstawionych podejść do rozwiązania problemu, najskuteczniejszym i zapewniającym największą dokładność okazało się wykorzystanie transfer learningu. Mimo tego, że uzyskane sieci zajmują więcej miejsca na dysku, to oba te wyniki, zarówno **63.4%** dla VGG16, jak i **63.6%** dla ResNet50 można uznać za bardzo dobre, biorąc pod uwagę fakt, że większe, bardziej skomplikowane modele trenowane przez [14] osiągały niewiele wyższą wydajność na zbiorze walidacyjnym.

Instalacja i użytkowanie

6.1 Instalacja

Do działania aplikacji wymagany jest interpreter języka Python. Kod źródłowy został napisany dla wersji 3.7, która wraz z innymi może zostać pobrana z [8] dla systemu Windows. W przypadku systemu Linux, interpreter Pythona w wersji 3.7 może zostać zainstalowany za pomocą polecenia:

```
sudo apt install python3.7
```

Po zainstalowaniu Pythona należy pobrać kody źródłowe aplikacji, po czym przejść do jej folderu głównego. Tam za pomocą poniższego polecenia należy utworzyć folder o nazwie venv, zawierający środowisko wirtualne:

```
Windows: python -m venv venv
Linux: python3 -m venv venv
```

Przed każdym rozpoczęciem pracy z aplikacją należy aktywować środowisko wirtualne:

```
Windows: venv\Scripts\activate.bat
Linux: source venv/bin/activate
```

W tym momencie w terminalu przed ścieżką do aktualnego katalogu powinien pojawić się (*venv*). Od tego momentu polecenia `python` i `pip` (dla systemu Windows) oraz `python3` i `pip3` (dla systemu Linux) będą korzystały ze środowiska wirtualnego. Jeśli zostało one aktywowane po raz pierwszy, należy zainstalować wszystkie niezbędne biblioteki i zależności wymagane przez aplikacje za pomocą polecenia:

```
Windows: pip install -r requirements.txt
Linux: pip3 install -r requirements.txt
```

W przypadku posiadania procesora graficznego, zamiast `requirements.txt` można skorzystać również z `requirements-gpu.txt`, w celu zainstalowania biblioteki TensorFlow w wersji, która pozwala na obsługę GPU. W tym przypadku niezbędne jest również zainstalowanie sterowników Nvidia cuDNN, które można pobrać z [6]. Po wykonaniu wszystkich powyższych poleceń aplikacja jest gotowa do pracy.

6.2 Użytkowanie

W każdej nowej sesji terminala należy aktywować środowisko wirtualne przed pierwszym uruchomieniem zarówno aplikacji do trenowania sieci neuronowych, jak i skryptu do rozpoznawania emocji. Można to zrobić za pomocą polecenia:

```
Windows: venv\Scripts\activate.bat
Linux: source venv/bin/activate
```

Aplikację do trenowania sieci można uruchomić za pomocą:

```
Windows: python src\train_main.py
Linux: python3 src/train_main.py
```

Nie przyjmuje ona żadnych argumentów. Zamiast tego, jest sterowana za pomocą pliku konfiguracyjnego `train_config.json`. Wszystkie parametry tego pliku, jak również sposób działa aplikacji, zostały szczegółowo opisane w rozdziale 5.

Skrypt rozpoznający emocje można uruchomić za pomocą:

```
Windows: python src\predict_main.py path1 path2 path3...
Linux: python3 src/predict_main.py path1 path2 path3...
```

Przyjmuje on dowolną dodatnią liczbę argumentów, która przechowuje ścieżki do plików i folderów zawierających zdjęcia w formacie .jpg, które mają zostać poddane analizie. Pozostałe parametry skryptu można konfigurować za pomocą pliku `predict_config.json`. Zostały one szczegółowo opisane w rozdziale 5, wraz ze sposobem działania i możliwościami skryptu.

Podsumowanie

7.1 Podsumowanie prac

Celem pracy było zaprojektowanie, zbudowanie i wyszkolenie sztucznej sieci neuronowej zdolnej do analizy emocji tłumów na podstawie zdjęć. Sieć miała być w stanie rozpoznawać emocję panującą na fotografii, przydzielając ją do jednej z trzech kategorii: emocji pozytywnych, neutralnych bądź negatywnych. Prócz maksymalizacji dokładności przewidywań, nacisk został również położony na szybkie działanie modelu i utrzymywanie go w jak najmniejszym rozmiarze, by umożliwić wykorzystywanie go na urządzeniach mobilnych.

By zautomatyzować proces trenowania sieci neuronowych, została napisana aplikacja w języku Python, która buduje i trenuje modele na podstawie wartości zdefiniowanych przez użytkownika w pliku konfiguracyjnym. Powstała ona w odpowiedzi na największe wyzwanie napotkane podczas pracy nad projektem, jakim była konieczność wytrenowania dużej liczby sieci, w celu znalezienia tej dającej najlepsze rezultaty. Dzięki tej aplikacji można w wygodny sposób zakolejkować do trenowania wiele modeli o różnej konfiguracji, na koniec uzyskując wyniki w przystępnej formie, wraz z wykresami historii trenowania. Powstał też skrypt, również sterowany za pomocą pliku konfiguracyjnego, za pomocą którego można uruchomić model, przekazać do niego zdjęcia i otrzymać wyniki w wygodnej formie. Testy zostały przeprowadzane dla modeli budowanych i trenowanych na 3 różne sposoby: z wykorzystaniem augmentacji danych, bez niej, oraz z wykorzystaniem transfer learningu. Wyniki testów zostały zgromadzone i przedstawione w niniejszym dokumencie (rozdział 5). Podsumowując, udało się wykonać wszystkie zaplanowane na początku pracy czynności.

7.2 Podsumowanie wyników

Z pośród modeli powstałych podczas treningu na zbiorze Group Affect Database 3.0 (opisanym w rozdziale 4) na największą uwagę zasługują sieci podsumowane w tabeli 7.1. Najmniejszy model osiągający zadowalające rezultaty (wynik wyższy niż 50% na walidacyjnym zbiorze danych) został wytrenowany z użyciem augmentacji danych. Jest to model przetwarzający zdjęcia w rozmiarze 150x150 pikseli. Składa się on z czterech par warstw, z których każda złożona jest z warstwy konwolucyjnej o pewnej liczbie filtrów i funkcji aktywacji ReLU oraz warstwy max pooling. Liczby filtrów w każdej następnej warstwie konwolucyjnej wynoszą odpowiednio 32, 64, 128 i 128. Każda z nich wykorzystuje jądra o standardowym rozmiarze równym 3x3. Każda z warstw max pooling również jest sparametryzowana w standardowy sposób, to znaczy używa okien konwolucyjnych o rozmiarze 2x2 i przesuwa je z krokiem równym 2. Do sieci została jeszcze dopięta warstwa porzucenia o współczynniku równym 0.5 oraz dwie warstwy gęste, wewnętrzna o 512 neuronach z funkcją ReLU i wyjściowa o 3 neuronach, korzystająca z funkcji softmax. Sieć wykorzystuje optymalizator Adam z learning rate wynoszącym 10^{-4}. Jako funkcja straty została wykorzystana kategorialna entropia krzyżowa. Model był trenowany z użyciem następującej konfiguracji augmentacji danych: rotation_range = 40, width_shift = 0.2, height_shift = 0.2, brightness_range = 0.2, zoom_range = 0.2, horizontal_flip = true, vertical_flip = false oraz fill_mode =„reflect”. Wydajność na poziomie **55.5%** została osiągnięta po 200 iteracjach treningu. Ta sieć charakteryzuje się szybkim przetwarzaniem zdjęć i najmniejszym w tym zestawieniu rozmiarem, co może przydać się w przypadku wykorzystania tego modelu na urządzeniach mobilnych.

Sieć, która osiągnęła największą wydajność, została zbudowana z wykorzystaniem uprzednio trenowanego modelu ResNet50, analizującego zdjęcia w rozmiarze 150x150 pikseli. Została zakończona gęsto połączonym klasyfikatorem składającym się z warstwy porzucenia o współczynniku 0.5, dwóch warstw po 512 neuronów,

wykorzystujących funkcję aktywacji ReLU, oraz ostatniej warstwie o 3 neuronach z funkcją aktywacji softmax. Używając optymalizatora RMSProp z learning rate wynoszącym $2 \cdot 10^{-4}$ oraz kategorialnej entropii krzyżowej w charakterze funkcji straty, model ten osiągnął dokładność na poziomie **63.6%**. Jest to najwyższa wydajność osiągnięta w tych testach, jednakże duża wielkość modelu (191 MB) może uniemożliwić wykorzystanie go na wszystkich typach urządzeń. Znajdzie on jednak zastosowanie wszędzie tam, gdzie szybkość działania sieci nie jest kluczowym czynnikiem.

Model, który zdaje się stanowić złoty środek pomiędzy wydajnością, a ilością zajmowanego miejsca na dysku, został zbudowany z wykorzystaniem uprzednio trenowanej sieci VGG16. Pozostała część jego architektury nie różni się niczym od tej wykorzystanej dla modelu ResNet50, w wyjątkiem użycia learning rate równego $2 \cdot 10^{-5}$. Sieć osiągnęła wydajność na poziomie **63.4%**, co jest wynikiem lepszym o 7.9 p.p. od tego osiągniętego przez najlepszy model wykorzystujący augmentacje danych, a jednocześnie waży 117.9 MB mniej niż sieć oparta na ResNet50. Z tego powodu, ten model zdaje się oferować najszerszy zakres potencjalnych możliwości wykorzystania, oferując zarówno wysoką wydajność jak i niski czas przetwarzania zdjęć.

Tablica 7.1: Najlepsze wytrenowane sieci

Sieć	Dokładność	Waga
Augmentacja	55.5%	**39.5 MB**
ResNet50	**63.6%**	191 MB
VGG16	63.4%	73.1 MB

Dotychczas, najlepsze rezultaty na wyżej wspomnianej bazie danych zostały osiągnięte przez jej autorów. [14] osiągnął wyniki na poziomie 67.64% na zbiorze walidacyjnym, łącząc BoW_{AU}, BoW_{LL} i $Scene_{CENTRIST}$ za pomocą MKL (patrz rozdział 3). Ta sieć faworyzowała jednak emocje pozytywne i neutralne, ponieważ były to najczęściej udzielane przez nią odpowiedzi. Inna zaprezentowana przez autorów architektura, składająca się z BoW_{AU}, BoW_{LL} i $Scene_{GIST}$ połączonych za pomocą MKL uzyskała, co prawda, nieznacznie gorszą ostateczną dokładność (67.15%), ale jednocześnie osiągając porównywalne wyniki na każdym podzbiorze danych.

7.3 Dalsze kroki

Kolejnym krokiem w rozwoju napisanej aplikacji może być dodanie wizualizacji pośrednich danych wyjściowych, tj. mapy cech, która jest generowana przez różne konkretne funkcje aktywacji warstw konwolucyjnych na podstawie danych wejściowych. Oprócz tego można wizualizować również mapy ciepła, czyli te miejsca na zdjęciu, które najbardziej przyczyniły się do tego, że dany obraz został przypisany do tej, a nie innej kategorii. Obie te funkcjonalności pozwoliłyby lepiej zrozumieć co dzieje się wewnątrz sieci i być może, dzięki temu, wytrenować lepszy model.

Same modele, z uwagi na niewielki rozmiar, mogą być wykorzystywane na urządzeniach mobilnych, a więc z tego powodu kolejnym krokiem może być stworzenie aplikacji na tę platformę, która np. robiłaby zdjęcie za każdym razem, kiedy rozpoznałaby, że w kadrze dominują pozytywne emocje. Podobna aplikacja rozpoznająca emocje w czasie rzeczywistym mogłaby zostać wykorzystana np. podczas koncertów czy innych wydarzeń rozrywkowych. Umożliwiłoby to wykrywanie momentów wydarzenia, które były najbardziej angażujące dla publiki.

Bibliografia

[1] Eigen. Strona główna: http://eigen.tuxfamily.org/, 01 2021.

[2] Emotion recognition in the wild challenge (emotiw). Strona: https://sites.google.com/view/emotiw2018, 01 2021.

[3] Imagenet. Strona główna: http://www.image-net.org/, 01 2021.

[4] Keras. Strona główna: https://keras.io/, 01 2021.

[5] Microsoft cognitive toolkit. Dokumentacja: https://docs.microsoft.com/en-us/cognitive-toolkit/, 01 2021.

[6] Nvidia cuda deep neural network. Strona główna: https://developer.nvidia.com/cudnn, 01 2021.

[7] Python. Strona główna: https://www.python.org/, 01 2021.

[8] Python. Do pobrania: https://www.python.org/downloads/, 01 2021.

[9] Tensorflow. Strona główna: https://www.tensorflow.org/, 01 2021.

[10] Theano. Repozytorium: https://github.com/Theano/Theano, 01 2021.

[11] S. Bucak, R. Jin, A. K. Jain. Multi-label multiple kernel learning by stochastic approximation: Application to visual object recognition. *Advances in Neural Information Processing Systems*, 2010.

[12] A. Cerekovic. A deep look into group happiness prediction from images. 2016.

[13] A. Dhall, J. Joshi, R. Goecke, T. Gedeon, M. Wagner. Emotion recognition in the wild challenge (emotiw) challenge and workshop summary. 2013.

[14] A. Dhall, J. Joshi, K. Sikka, R. Goecke, N. Sebe. The more the merrier: Analysing the affect of a group of people in images. *IEEE FG*, 2015.

[15] S. Garg. Group emotion recognition using machine learning. 2019.

[16] A. Graves, D. Eck, N. Beringer, J. Schmidhuber. Biologically plausible speech recognition with lstm neural nets. *International Workshop on Biologically Inspired Approaches to Advanced Information Technology*, strony 127–136. Springer, 2004.

[17] A. Graves, S. Fernández, F. Gomez, J. Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. *Proceedings of the 23rd international conference on Machine learning*, strony 369–376, 2006.

[18] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), 1989.

[19] A. C. Murillo, I. S. Kwak, L. Bourdev, D. J. Kriegman, S. Belongie. Urban tribes: Analyzing group photos from a social perspective. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition and Workshops (CVPRW)*, 2012.

[20] K.-S. Oh, K. Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

[21] A. Oliva, A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 2001.

[22] H. Sak, A. Senior, K. Rao, F. Beaufays, J. Schalkwyk. Google voice search: faster and more accurate. *Google Research Blog, 2015, http://googleresearch. blogspot. ch/2015/09/google-voice-search-faster-and-more. html*, 2015.

[23] J. Schmidhuber, S. Hochreiter. Long short-term memory. *Neural Comput*, 9(8):1735–1780, 1997.

[24] B. Sun, Q. Wei, L. Li, Q. Xu. Lstm for dynamic emotion and group emotion recognition in the wild. 2016.

[25] L. Surace, M. Patacchiola, E. B. Sőnmez, W. Spataro. Emotion recognition in the wild using deep neural networks and bayesian classifiers. 2017.

[26] L. Tan, K. Zhang, K. Wang, X. Zeng. Group emotion recognition with individual facial emotion cnns and global image based cnns. 2017.

[27] J. Weng, N. Ahuja, T. S. Huang. Cresceptron: a self-organizing neural network which grows adaptively. *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, wolumen 1, strony 576–581 vol.1, 1992.

[28] J. Wu, J. M. Rehg. Centrist: A visual descriptor for scene categorization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2011.

Zawartość płyty CD

Na zawartość dołączonej płyty CD składa się niniejszy dokument w formacie `.pdf` wraz z jego kodami źródłowymi w formacie `.tex` zamieszczonymi w folderze `tex`. Ponadto, w folderze `src` został umieszczony kod aplikacji automatyzującej trenowanie sieci neuronowych, oraz skrypt wykorzystywany do uruchamiania tych sieci i formatowania zwracanych przez nie wyników. Na płycie znajdują się również pliki `train_config.json` i `predict_config.json`, pozwalające na sterowanie wyżej wspomnianą aplikacją i skryptem, oraz pliki `requirements.txt` i `requirements-gpu.txt`, służące do instalacji bibliotek wymaganych w środowisku wirtualnym.