AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA TELEKOMUNIKACJI

PRACA INŻYNIERSKA

Interaktywna aplikacja do klasyfikacji obrazów za pomocą sieci neuronowej

An interactive application for image classification with the help of neural network

Autor:	Dawid Jakóbczak
Kierunek studiów:	Teleinformatyka
Typ studiów:	Stacjonarne
Opiekun pracy:	dr inż. Jarosław Bułat

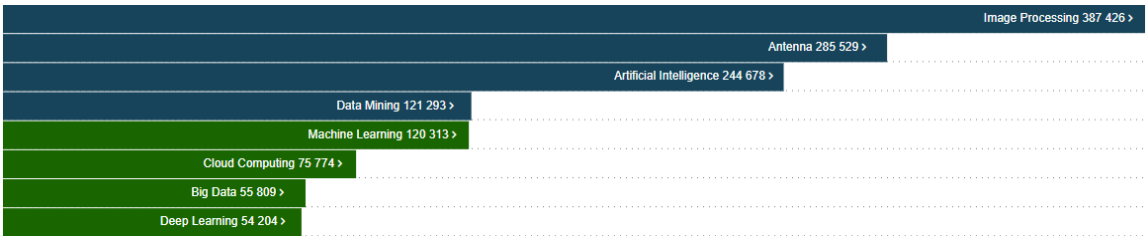
Kraków, 2021

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.) „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej „sądem koleżeńskim", oświadczam, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Spis treści

Wstęp

Od wielu lat rozpoznawanie obrazów (ang. *computer vision*) jest jednym z obszarów IT cieszących się bardzo dużym zainteresowaniem. W ostatnich latach dzięki rozpowszechnieniu się głębokiego uczenia maszynowego (ang. *deep learning*) jej rozwój gwałtownie przyspieszył i jest to dziedzina ciesząca się bardzo dużą popularnością wśród naukowców, co potwierdzają statystyki biblioteki cyfrowej IEEE Xplore przedstawione na rysunku 1.



Rysunek 1: Najczęściej wyszukiwane wyrażenia, dane z biblioteki cyfrowej IEEE Xplore z dnia 23.12.2020

Rozwój technologii oraz ogromna liczba danych generowanych każdego dnia spowodowały, że głębokie uczenie maszynowe już teraz jest wykorzystywane w wielu obszarach [1] takich jak przetwarzanie obrazów, rozpoznawanie mowy, robotyka czy medycyna. Na szczególną uwagę zasługuje architektura splotowych sieci neuronowych (ang. CNN – *Convolutional Neural Network*). Stała się ona podstawowym narzędziem w rozwiązywaniu problemów związanych z analizą i przetwarzaniem obrazów, w tym problemu klasyfikacji obrazów. Z każdym rokiem powstają nowe architektury [2] których skuteczność już od 2016, wraz z opracowaniem architektury sieci resztkowych (ang. *Residual Neural Networks*) [3], często przekracza możliwości człowieka w wybranych dziedzinach – np. rozpoznawaniu obrazów.

Dominującym stylem projektowania aplikacji jest podejście *cloud-native* [4], w którym duży nacisk kładzie się na jasno określony i dobrze udokumentowany interfejs programistyczny (ang. API – *Application Programming Interface*) oparty o zasady REST (ang. *Representational state transfer*) [5]. W celu jego zdefiniowania często korzysta się z dokumentacji OpenAPI [6], która jest wspierana przez takie firmy jak Google, Microsoft, czy IBM.

Celem pracy było zaimplementowanie interaktywnej aplikacji umożliwiającej użytkownikowi śledzenie procesu uczenia klasyfikatora binarnego, opartego o architekturę CNN.

W pierwszym rozdziale opisano zagadnienia, które zapewnią podstawową wiedzę czytelnikowi, niezbędną w zrozumieniu dalszej części pracy. Drugi rozdział poświęcony jest implementacji systemu. Przedstawiona zostanie zarówno architektura jak i opis poszczególnych komponentów aplikacji. Rozdział trzeci stanowi zbiór przykładowych scenariuszy, które mogą zostać zrealizowane z użyciem zaprojektowanej aplikacji.

Rozdział 1

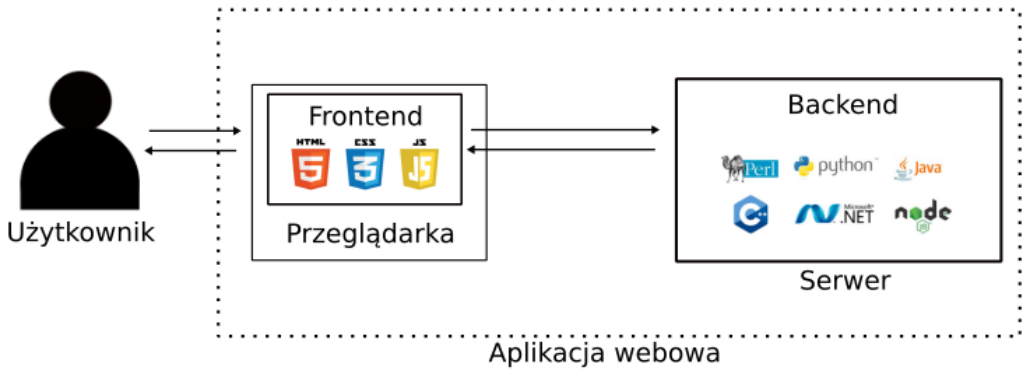
Zagadnienia teoretyczne

Zaprojektowany system łączy w sobie dwa obszary technik informatycznych, jakimi są tworzenie aplikacji webowych oraz problem klasyfikacji obrazów z użyciem splotowych sieci neuronowych. W niniejszym rozdziale opisane zostaną wybrane zagadnienia umożliwiające czytelnikowi zapoznanie się z podstawowymi pojęciami z wyżej wymienionych obszarów.

1.1 Projektowanie aplikacji webowych

Jednym z pierwszych etapów w procesie tworzenia systemu jest wybór architektury, która określa sposób budowy aplikacji. Istnieje wiele architektur, które zostały zaproponowane na przestrzeni ostatnich lat [7], natomiast aktualnie dominującym podejściem w tworzeniu aplikacji jest wykorzystanie technik webowych. Aplikacje webowe opierają się o architekturę klient-serwer, a jako protokół komunikacyjny stosuje się HTTP (ang. *Hypertext Transfer Protocol*). Taki sposób tworzenia aplikacji jest wygodny z punktu widzenia klienta, ponieważ aby skorzystać z aplikacji nie trzeba instalować żadnego dodatkowego oprogramowania, a aplikacja może być używana niemal natychmiastowo po jej wdrożeniu.

Struktura aplikacji webowej



Rysunek 1.1: Schemat aplikacji webowej

Możemy wyróżnić dwa główne komponenty aplikacji webowej, co zaprezentowano na rysunku 1.1.

- *Frontend* - część aplikacji, która jest prezentowana użytkownikowi. Serwer po otrzymaniu żądania od klienta, przetwarza je, a następnie w odpowiedzi zwraca pliki,

które są interpretowane po stronie klienta. W rezultacie przeglądarka generuje interfejs graficzny, który pozwala użytkownikowi w przyjazny sposób wysyłać kolejne żądania,

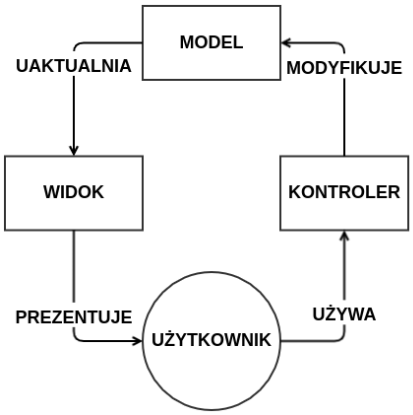
- *Backend* - część aplikacji, która jest odpowiedzialna za przetwarzanie zapytań otrzymanych od klienta. To ona definiuje jaką funkcjonalność zapewnia serwer.

Stosowane techniki

Podstawowymi technikami stosowanymi po stronie klienta są:

- HTML (ang. *Hypertext Markup Language*) - język znaczników, odpowiedzialny za strukturę interfejsu,
- CSS (ang. *Cascade Style Sheet*) - język służący do opisu warstwy prezentacji,
- JavaScript - język programowania, którego funkcją po stronie klienta jest zapewnienie interakcji z użytkownikiem.

Ze względu na niewielkie wymagania, które wprowadza aplikacja webowa po stronie klienta, może być ona używana z poziomu różnych urządzeń, które ze względu na różną wielkość ekranu wymagają odpowiedniego dostosowania struktury interfejsu. Bootstrap [8] jest platformą która umożliwia dynamiczną zmianę widoku w zależności od wykrytej wielkości ekranu zgodnie z technikami RWD (ang. *Responsive Web Design*) i *mobile-first*. Dodatkowo, w celu zapewnienia asynchronicznej komunikacji pomiędzy przeglądarką a serwerem stosuje się technikę AJAX, dzięki której wraz z pomocą JavaScript, możliwe jest aktualizowanie interfejsu graficznego bez konieczności odświeżenia strony.



Rysunek 1.2: Architektura Model-Widok-Kontroler

Z kolei po stronie serwera stosuje się środowiska takie jak Python, node.js, Java, PHP czy Ruby. Ze względu na złożoność funkcji jakie musi pełnić serwer, jego struktura często jest dzielona zgodnie z architekturą Model-Widok-Kontroler (ang. MVC – *Model View Controller*), której schemat przedstawiono na rysunku 1.2. Istnieją różne modyfikacje MVC, natomiast w jej podstawowej formie wydzielono trzy główne komponenty [9]:

- Kontroler - odpowiada za przyjmowanie żądań oraz na podstawie ścieżki zawartej w żądaniu, wywołuje odpowiednią funkcję, której zadaniem jest obsłużenie żądania,

- Model - udostępnia interfejs zapewniający dostęp do danych, a także umożliwia ich manipulację. Komponent ten odpowiada także za walidację danych,
- Widok - na podstawie otrzymanych danych tworzy ich reprezentację zrozumiałą dla klienta.

OpenAPI

W celu zapewnienia komunikacji pomiędzy klientem a serwerem niezbędne jest zdefiniowanie interfejsu programistycznego. Protokół HTTP nie określa jak zaprojektowany powinien zostać taki interfejs, dlatego na przestrzeni lat zdefiniowano zbiór reguł zwanych REST [5]. REST narzuca pewne ograniczenia i wprowadza warstwę abstrakcji, dzięki czemu API zdefiniowane zgodnie z zasadami REST ma dobrze określoną strukturę, zrozumiałą pomiędzy różnymi systemami.

Dokumentacja OpenAPI umożliwia zaprojektowanie interfejsu zgodnie z regułami REST w postaci pliku zrozumiałego zarówno człowiekowi jak i maszynie w jednym z dwóch formatów, którymi są YAML [10] lub JSON [11]. OpenAPI udostępnia wiele narzędzi, które pomagają w projektowaniu i testowaniu API, takich jak SwaggerUI [12] umożliwiający wizualizacje zdefiniowanego interfejsu, a także interaktywne generowanie żądań do serwera. Wiele platform do tworzenia aplikacji webowych jest zintegrowana z OpenAPI, co umożliwia między innymi automatyczną walidację żądań na podstawie schematów zdefiniowanych w OpenAPI. Przykład pliku zgodnego z OpenAPI zaprezentowano w tabeli 1.1.

Tabela 1.1: Przykład pliku zgodnego z dokumentacją OpenAPI. Zdefiniowano jeden punkt końcowy (ang. *endpoint*) `GET /users` umożliwiający pobranie listy użytkowników. Określono także oczekiwany status oraz typ MIME odpowiedzi

```yaml
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description
  version: 0.1.9
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: Optional description e.g. A list of users.
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
              example:
                - name: Jessica Smith
                - name: Ron Stewart
```

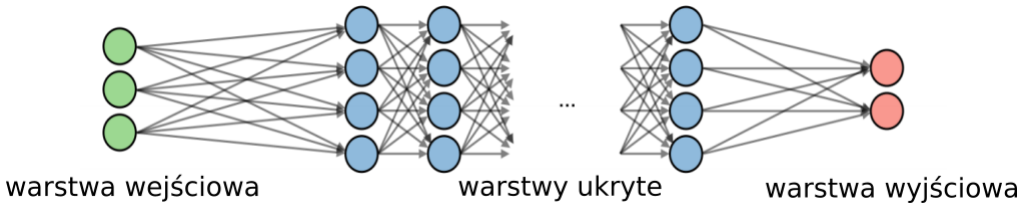
OpenAPI, dzięki wielu opcjonalnym informacjom które możemy zdefiniować, umożliwia zaprojektowanie bardzo dobrze udokumentowanego API. Przykładem jest pole *description*, zawierające opis danego elementu, a także *example* pozwalające na zdefiniowanie

przykładowej reprezentacji danego zasobu, lub nawet całej odpowiedzi na żądanie. Dodatkowo OpenAPI umożliwia definiowanie elementów które mogą być wielokrotnie używane w pliku, co ułatwia utrzymanie API.

1.2 Problem klasyfikacji obrazów

Jednym z podstawowych problemów w dziedzinie analizy obrazów jest klasyfikacja obrazów. W ostatniej dekadzie dzięki dynamicznemu rozwojowi głębokich sieci neuronowych, a w szczególności splotowych sieci neuronowych, nastąpił znaczący postęp w tej dziedzinie. W celu rozwiązania tego problemu najczęściej stosuje się uczenie nadzorowane (ang. *supervised learning*), które zakłada posiadanie zbioru obrazów wraz z ich etykietami (poprawnie oznaczonymi klasami – tzw. ground truth), co w przypadku stosowania uczenia nienadzorowanego (ang. *unsupervised learning*), bądź uczenia ze wzmocnieniem (ang. *reinforcement learning*) nie ma miejsca.

Sztuczne sieci neuronowe



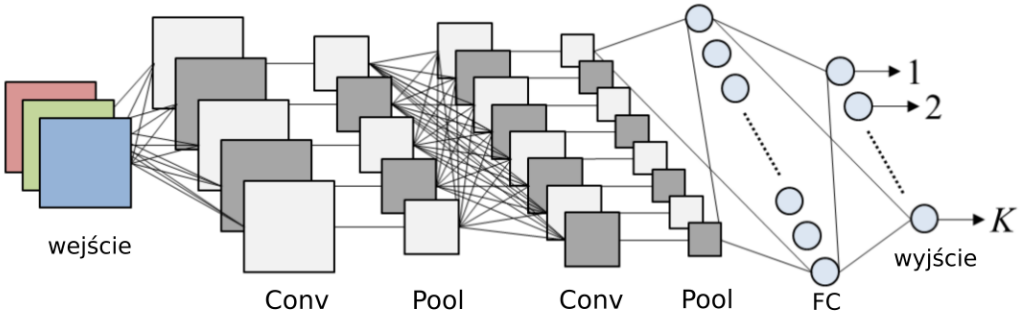
Rysunek 1.3: Architektura sieci neuronowych [13]

Sztuczne sieci neuronowe (ang. ANN - *Artificial Neural Networks*) swoją inspirację czerpią z dotychczasowej wiedzy na temat działania ludzkiego mózgu. Architektura ANN opiera się na połączonych ze sobą kolejnych warstwach neuronów, co zostało zaprezentowane na rysunku 1.3. Sygnał otrzymany na warstwie wejściowej (ang. *input layer*) jest propagowany przez kolejne warstwy ukryte (ang. *hidden layers*), aż w wyniku na wyjściu sieci otrzymujemy wynik, który w przypadku problemu klasyfikacji często jest prawdopodobieństwem przypisania podanego na wejściu sieci obrazu do pewnej klasy. Celem procesu uczenia ANN jest zminimalizowanie błędu określonego za pomocą funkcji kosztu. W tym celu stosuje się algorytm wstecznej propagacji (ang. *backpropagation*), który pozwala w wydajny sposób obliczyć pochodne cząstkowe funkcji kosztu względem optymalizowanych parametrów sieci.

Głębokie uczenie maszynowe jest podzbiorem uczenia maszynowego. Uznaje się, że głęboka sieć neuronowa (ang. DNN – *Deep Neural Network*) to taka która ma wiele warstw ukrytych – od kilku do nawet ponad tysiąca [1]. Zwiększanie liczby warstw pozwala na wydobycie wysokopoziomowych cech sygnału wejściowego.

Splotowe sieci neuronowe

Splotowe sieci neuronowe stały się podstawowym narzędziem używanym w zadaniach związanych z analizą obrazów. Dzięki wielokrotnej operacji splotu, sieci te potrafią rozpoznać bardzo skomplikowane struktury, co przyczyniło się do ich stosowania w wielu obszarach.



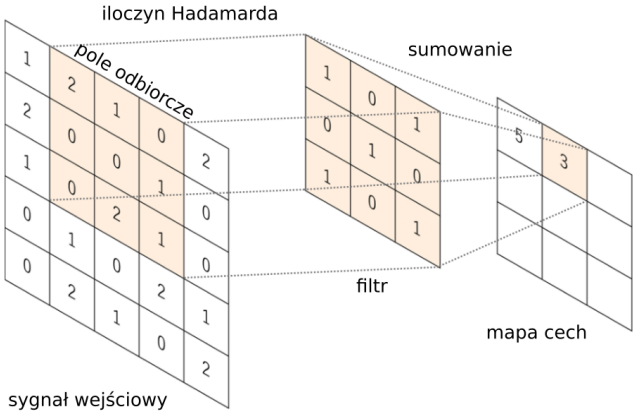
Rysunek 1.4: Przykład architektury splotowych sieci neuronowych [14]

Architektura CNN, która została przedstawiona na rysunku 1.4, składa się najczęściej z wielu bloków, których celem jest wydobycie cech oraz jednej do trzech warstw gęstych odpowiedzialnych za klasyfikacje. Każdy z bloków składa się z następujących warstw:

- jednej lub więcej warstw splotowych (ang. *convolutional layers*) - każda warstwa splotowa aplikuje operację splotu pomiędzy warstwą poprzedzającą a filtrami, których wagi są optymalizowane w procesie uczenia. Na wyjściu warstwy splotowej otrzymujemy mapy cech (ang. *feature maps*), co zostało przedstawione na rysunku 1.5,
- warstwa łącząca (ang. *pooling layer*) - odpowiedzialna za redukcje wymiarowości oraz uodpornienie sieci na niewielkie zmiany sygnału wejściowego (przesunięcie, rotacje) przez agregowania wartości sygnału wejściowego. Najczęściej stosuje się przesuwne okno o wielkości 2x2 i kroku większym niż 1, które aplikuje funkcję maksimum.

Dodatkowo po każdej warstwie splotowej często wprowadza się nieliniowość za pomocą funkcji aktywacji, gdzie najczęściej wybierana jest funkcja ReLu lub jej odmiany [1].

Obrazy są sygnałami wielowymiarowymi, których przetwarzanie z użyciem wyłącznie warstw gęstych (ang. *dense layers*) byłoby zadaniem bardzo wymagającym obliczeniowo. Koncepcja warstw splotowych pozwala na ograniczenie liczby połączeń pomiędzy neuronami.



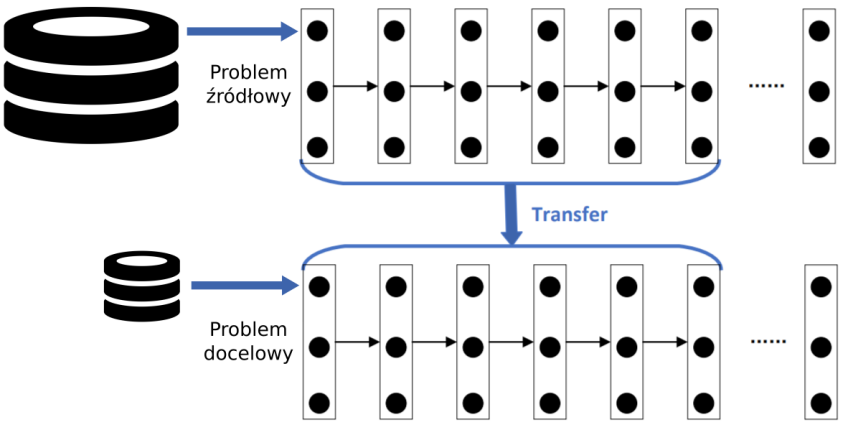
Rysunek 1.5: Operacja splotu stosowana w warstwie splotowej [15]

Dla każdej warstwy splotowej definiowana jest pewna liczba filtrów, a każdy filtr używany jest do wygenerowania mapy jednej z cech. Powstaje ona dzięki aplikacji operacji

iloczynu skalarnego pomiędzy trójwymiarowym filtrem (szerokość, wysokość oraz głębokość, która zależy od liczby kanałów warstwy poprzedzającej) oraz lokalnym otoczeniem neuronu w warstwie poprzedzającej, określanym mianem pola odbiorczego (ang. *receptive field*). Operacja ta dla sygnału wejściowego o jednym kanale została przedstawiona na rysunku 1.5. Filtr jest przesuwany wzdłuż i wszerz sygnału wejściowego z określonym krokiem, dzięki czemu w wyniku otrzymujemy mapę cech zależną od wszystkich neuronów warstwy poprzedzającej. Rozmiar filtru jest mniejszy od rozmiaru sygnału wejściowego, co powoduje że filtr z jednakowymi wagami jest aplikowany na różnych fragmentach obrazu. Skutkuje to dostosowywaniem wag filtru w procesie uczenia w taki sposób, aby wykrywał on konkretną cechę w sygnale wejściowym, niezależnie od jej położenia. Kaskadowe ułożenie warstw splotowych powoduje uzależnienie pojedynczych neuronów w coraz głębszych warstwach od coraz większej liczby neuronów warstwy wejściowej sieci. Taka architektura pozwala kolejnym warstwom na wydobycie wysokopoziomowych cech obrazu.

Transfer learning

Głębokie sieci neuronowe wymagają bardzo dużej liczby próbek (ich liczba jest zależna od złożoności problemu, natomiast często wspomina się o 1000 próbkach na jedną klasę) oraz zasobów obliczeniowych potrzebnych do wytrenowania modelu [16]. W wielu dziedzinach takich jak medycyna [15] nie mamy dostępu do dużego zbioru próbek, które mogłyby zostać użyte w procesie uczenia. W celu zredukowania liczby potrzebnych próbek często korzysta się z techniki zwanej *transfer learning*. Polega ona na skorzystaniu z modelu przetrenowanego na zbiorze zawierającym wiele generycznych obrazów, co pozwala na użycie takiego modelu jako detektora cech w docelowym zadaniu. Schemat użycia techniki *transfer learning* pokazano na rysunku 1.6. W praktyce często stosuje się modele



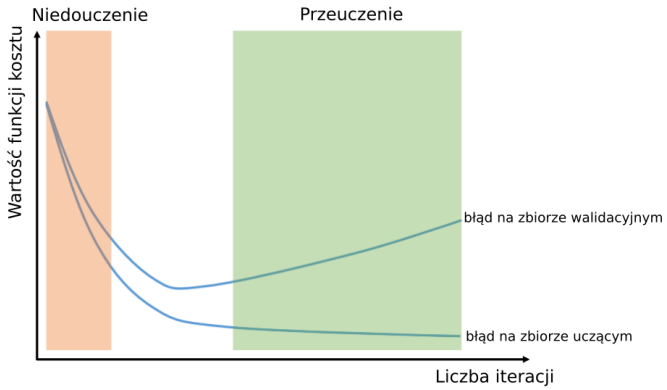
Rysunek 1.6: Schemat [17] zastosowania *transfer learning*. W celu rozwiązania docelowego problemu dostosowujemy ostatnie warstwy modelu przetrenowanego na bardzo dużym zbiorze danych. Taki model następnie możemy przetrenować na niewielkim zbiorze obrazów z obszaru docelowego, aby zaadaptować go do problemu

przetrenowane na zbiorze ImageNet, zawierającym 1.2 mln obrazów należących do 1000 klas. Taka duża liczba klas pozwala na wytrenowanie modelu, który jest w stanie wydobyć bardzo uogólnioną reprezentację cech. W tak przetrenowanym modelu podmieniona zostaje ostatnia warstwa, odpowiedzialna za klasyfikacje, tak aby liczba wyjść modelu odpowiadała docelowemu problemowi. Większość, bądź nawet wszystkie warstwy odpowiedzialne za wydobycie cech są zamrażane [18], co pozwala na przyspieszenia procesu

uczenia. Liczba zamrożonych warstw zależy od wielkości zbioru uczącego, który posiadamy oraz od tego jak bardzo nasz docelowy problem jest specyficzny w odniesieniu do obrazów użytych w procesie uczenia przetrenowanego modelu.

Śledzenie procesu uczenia klasyfikatora binarnego

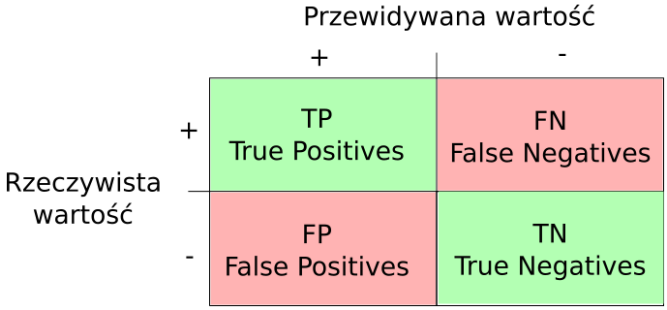
Kluczową cechą sieci neuronowych jest umiejętność generalizacji danego problemu, co w przypadku problemu klasyfikacji obrazów będzie takim wytrenowaniem modelu aby był w stanie poprawnie klasyfikować obrazy, których wcześniej nie widział. Częstym zja-



Rysunek 1.7: Przykład procesu uczenia modelu z zaznaczeniem etapów niedouczenia i przetrenowania sieci [15]

wiskiem, które pojawia się podczas trenowania modelu jest przeuczenie (ang. *overfitting*), charakteryzujące się tym, że model uczy się "na pamięć" obrazów wejściowych zamiast dopasowywać wydobyte cechy do odpowiednich klas. Rysunek 1.7 przedstawia wykres funkcji kosztu, który obrazuje sytuacje niedouczenia (ang. *underfitting*) i przeuczenia modelu. Z przeuczeniem mamy do czynienia w momencie, gdy błąd na zbiorze walidacyjnym zaczyna rosnąć, podczas gdy błąd na zbiorze uczącym nadal maleje. Z drugiej strony niedouczenie sieci to sytuacja, gdy wartość funkcji kosztu jest wysoka, a spowodowane to może być zbyt krótkim czasem uczenia, bądź niedostosowaniem modelu do złożoności danego problemu.

W celu śledzenia procesu uczenia możemy stosować takie metryki jak skuteczność (ang. *accuracy*), precyzja (ang. *precision*), czułość (ang. *recall*), wartość F1 oraz wartość funkcji kosztu. W celu wyliczenia poszczególnych metryk korzysta się z przedstawionej na rysunku 1.8 macierzy pomyłek (ang. *confusion matrix*).



Rysunek 1.8: Macierz pomyłek

Skuteczność (1.1), określa ile próbek zostało zaklasyfikowanych poprawnie względem

całego zbioru.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1.1}$$

Precyzja (1.2) jest miarą dokładności z jaką model przewiduje wyniki, czyli określa jak często model poprawnie klasyfikuje daną klasę. Zwiększając próg odcięcia (ang. *cut off*) klasyfikatora binarnego możemy zwiększyć jego precyzję.

$$Precision = \frac{TP}{TP + FP} \tag{1.2}$$

Czułość (1.3) określa jak wiele próbek z danej klasy zostało poprawnie zaklasyfikowanych. Zmniejszając próg odcięcia klasyfikatora binarnego możemy zwiększyć jego czułość.

$$Recall = \frac{TP}{TP + FN} \tag{1.3}$$

Precyzja i czułość nie idą w parze, zwiększając jedną z tych metryk zazwyczaj zmniejszamy drugą. W celu określenia równowagi pomiędzy precyzją a czułością stosuje się metrykę F1 (1.4).

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{1.4}$$

Rozdział 2

Implementacja

W niniejszym rozdziale zostanie zaprezentowana architektura oraz implementacja serwera HTTP umożliwiającego śledzenie procesu uczenia maszynowego. W pierwszej kolejności zostanie krótko opisane środowisko oraz techniki użyte podczas implementacji wraz ze strukturą plików aplikacji. Następnie opisany zostanie serwer HTTP oraz jego główne komponenty, co pozwoli w dalszej części skupić się na przedstawieniu jego funkcjonalności. Rozdział zostanie zakończony opisem interfejsu graficznego.

2.1 Projekt systemu

Aplikacja łączy w sobie dwie odrębne dziedziny wiedzy, jakimi są tworzenie aplikacji webowych oraz uczenie maszynowe.

Zaimplementowany system pozwala na wizualizację procesu uczenia klasyfikatora binarnego na dowolnym zbiorze danych. System umożliwia interaktywne przypisywanie etykiet obrazom, co pozwala na śledzenie procesu douczania modelu, a po jego wytrenowaniu na generowanie poprawnych etykiet ze skutecznością osiągniętą przez klasyfikator. Ponadto, aplikacja zapewnia możliwość przetestowania wytrenowanego modelu z użyciem zbioru testowego, bądź własnego obrazu wysłanego do serwera. Projekt zakładał zaprojektowanie serwera, który w dużej części może być konfigurowalny, a jego interfejs programistyczny dobrze udokumentowany.

Ważną częścią systemu jest interfejs użytkownika, który pozwala na wygodne korzystanie z aplikacji na różnych urządzeniach. Duży nacisk został położony na informowaniu użytkownika o występujących zdarzeniach, a także zaprojektowaniu interfejsu, który zachęci użytkownika do korzystania z aplikacji.

Środowisko programistyczne

Popularnymi językami programowania używanymi w uczeniu maszynowym [19] są Python, R, Matlab, C++ oraz Java. Do implementacji skorzystano z języka Python z następujących powodów:

- społeczność Pythona zapewnia wiele modułów, a w szczególności:
  - do tworzenia aplikacji webowych, m.in. Flask, Django, FastAPI,
  - do tworzenia modeli uczenia maszynowego, gdzie dominują platformy Tensorflow oraz PyTorch,

- Python jest językiem interpretowanym pozwalającym na relatywnie szybkie pisanie aplikacji względem konkurencyjnych, kompilowanych języków, takich jak C++, Java.

W celu implementacji serwera HTTP wykorzystano platformę Flask, a integracja interfejsu programistycznego zaprojektowanego zgodnie z dokumentacją OpenAPI była możliwa dzięki użyciu modułu connexion. Jako biblioteki do uczenia maszynowego użyto platformy PyTorch [20], która zapewnia taki sam poziom wydajności [21] jak ten osiągany przez bibliotekę Tensorflow. PyTorch został wybrany ze względu na swoją integrację z samym językiem oraz interfejs, który jest zbliżony do tych spotykanych w innych modułach napisanych w języku Python. W celu generowania interaktywnych wykresów, umożliwiających śledzenie uczenia modelu skorzystano z platformy Plotly, a do asynchronicznej komunikacji pomiędzy przeglądarką a serwerem użyto techniki AJAX.

Aplikacja została napisana zarówno pod systemy z rodziny Windows jak i Linux, testy odbyły się na systemie Windows 10 oraz Ubuntu 18.04. Wykorzystana w testach karta graficzna to Gigabyte GeForce GTX 1660 SUPER.

Do zarządzania modułami użyto narzędzia pipenv, które automatycznie tworzy środowisko wirtualne dla aplikacji oraz umożliwia w sposób deterministyczny jej uruchamianie na różnych systemach.

2.2 Struktura aplikacji

```
.
├── mlvt/
│   ├── config/
│   │   └── ...
│   ├── model/
│   │   ├── preprocessing/
│   │   │   └── ...
│   │   ├── models.py
│   │   └── ...
│   └── server/
│       ├── actions/
│       │   └── ...
│       ├── openapi/
│       │   └── openapi.yml
│       ├── static/
│       │   ├── sample_dataset/
│       │   │   └── ...
│       │   ├── js/
│       │   │   └── ...
│       │   ├── images/
│       │   │   └── ...
│       │   └── style.css
│       ├── views/
│       │   └── ...
│       ├── templates/
│       │   └── ...
│       ├── run.py
│       └── ...
├── Pipfile
└── Pipfile.lock
```

Rysunek 2.1: Uproszczona struktura projektu

Najważniejsze pakiety struktury projektu widocznej na rysunku 2.1:

- *config* - zawiera plik konfiguracyjny w formacie YAML oraz interfejs `ConfigManager` umożliwiający odwoływanie się do niego z poziomu widoków,
- *model* - zawiera moduły związane z uczeniem maszynowym, najważniejszym modułem z tego pakietu jest `models.py` w którym zdefiniowany jest model splotowych sieci neuronowych,
- *server* - zawiera moduły implementujące serwer HTTP oraz zarządzające plikami serwera:
  - *actions* - zawiera moduły implementujące długo trwające akcje,
  - *openapi* - zawiera plik `openapi.yml`, w którym zdefiniowano interfejs programistyczny,
  - *static* - zawiera pliki statyczne serwera, przykładem mogą być pliki z kodem javascript oraz obrazy wyświetlane użytkownikowi. Zapisywane są tu także wszystkie niezbędne pliki oraz obrazy związane ze zbiorami danych, które zdefiniowano w pliku konfiguracyjnym,
  - *views* - zawiera moduły widoków, które obsługują żądania HTTP,
  - *templates* - zawiera szablony jinja2 na podstawie których generowane są pliki HTML,
- w pliku *Pipfile* zdefiniowano moduły niezbędne do uruchomienia aplikacji i na jego podstawie generowany jest plik *Pipfile.lock*. Zawiera on wszystkie potrzebne moduły wraz z ich wersjami, co pozwala na utworzenia deterministycznego środowiska, w którym możliwe będzie uruchomienie aplikacji.

Przykładowa struktura zbioru danych

```
.
└── sample_dataset/
    ├── annotations/
    │   ├── unl.json
    │   ├── train.json
    │   ├── valid.json
    │   └── test.json
    ├── data/
    │   ├── predictions.json
    │   ├── test_results.json
    │   └── train_results.json
    ├── images_unl/
    │   ├── img1
    │   └── ...
    ├── images_train/
    │   └── ...
    ├── images_valid/
    │   └── ...
    ├── images_test/
    │   └── ...
    └── models/
        ├── best.pt
        └── training.pt
```

Rysunek 2.2: Przykładowa struktura plików zbioru danych

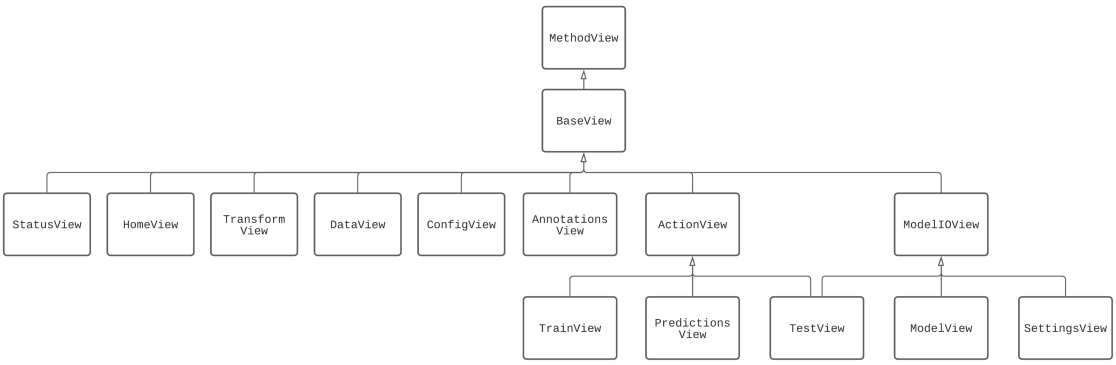
W katalogu *static* znajdują się struktury plików dla poszczególnych zbiorów danych. Przykład takiej struktury zaprezentowano na rysunku 2.2:

- katalog *images_unl* - zbiór obrazów nieetykietowanych,
- katalogi *images_train*, *images_valid*, *images_test* - odpowiednio zbiór uczący, walidacyjny oraz testowy. Każdy z nich powinien posiadać dwa podkatalogi w których będą umieszczone obrazy dla każdej z dwóch klas. Dokładny opis tych zbiorów oraz zbioru nieetykietowanego zamieszczono w podrozdziale 2.5,
- katalog *annotations* zawiera pliki z etykietami obrazów dla wymienionych powyżej zbiorów,
- katalog *data* zawiera pliki wynikowe procesu trenowania i testowania modelu oraz plik zawierający dane na temat ostatniej predykcji modelu,
- katalog *models* zawiera stan bieżącego modelu oraz modelu o najwyższej skuteczności.

Punkty końcowe udostępnione przez serwer automatycznie tworzą zaprezentowaną strukturę zgodnie z plikiem konfiguracyjnym opisanym w podrozdziale 2.5.

Struktura oparta o widoki

W klasycznym podejściu tworzenia aplikacji przy użyciu platformy Flask definiowane są funkcje do obsługi żądań dla wszystkich ścieżek URL udostępnianych przez serwer. Struktura ta staje się niewygodna w momencie gdy interfejs programistyczny definiuje wiele metod HTTP dla danej trasy. Zdecydowanie bardziej modularnym i łatwiejszym w utrzymaniu podejściem jest zdefiniowanie widoków w postaci klas oraz funkcji służących do obsługi poszczególnych metod HTTP. Strukturę widoków przedstawiono na rysunku 2.3.

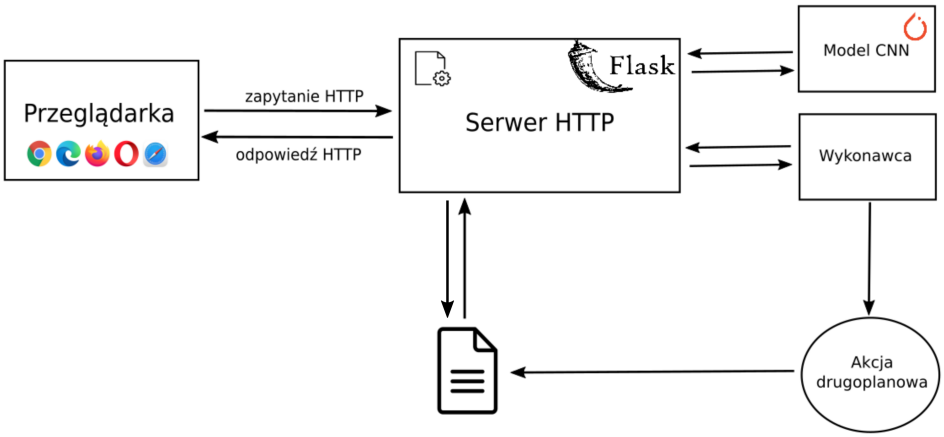


Rysunek 2.3: Diagram przedstawiający hierarchię zaimplementowanych widoków

Klasą bazową dla wszystkich widoków jest klasa `MethodView` udostępniana przez platformę Flask. Wszystkie widoki obsługujące żądania, które zdefiniowano w aplikacji są klasami pochodnymi klasy `BaseView`. Jej głównym zadaniem jest zapewnienie interfejsu do pliku konfiguracyjnego udostępnionego za pomocą klasy `ConfigManager`. Klasy pochodne nie mające klas potomnych są klasami obsługującymi żądania HTTP. Widoki dziedziczące po klasie `ActionView` obsługują akcje długo trwające, które opisano w podrozdziale 2.3. Klasa `ModelIOView` udostępnia interfejs umożliwiający wczytanie oraz zapis modelu sieci neuronowych.

2.3 Architektura systemu

Zakres pracy obejmuje implementacje serwera działającego w oparciu o protokół HTTP [22]. Serwer w odpowiedzi na żądania zwraca plik HTML, który jest gotowy do wyświetlenia po stronie klienta. Schemat architektury systemu przedstawiono na rysunku 2.4.



Rysunek 2.4: Schemat architektury systemu

System składa się z czterech głównych komponentów:

- serwera HTTP - główny komponent systemu, który przyjmuje żądania od klienta i w celu wykonania akcji komunikuje się z pozostałymi komponentami,
- modelu CNN - binarny klasyfikator obrazów oparty na splotowych sieciach neuronowych, którego stanem zarządza serwer HTTP zgodnie z otrzymanymi żądaniami,
- modułu zarządzającego plikami - wiele informacji o aktualnym stanie modelu, bądź wynikach poprzednio wykonanych akcji jest zapisywanych w systemie plików serwera. Moduł ten ma za zadanie zapewnić interfejs, za pomocą którego inne komponenty mogłyby odczytywać, bądź modyfikować zawartość wspomnianych plików,
- modułu zarządzającego wątkami - zewnętrzny moduł, którego zadaniem jest utworzenie wątku wykonującego określoną akcję. Ogranicza możliwość rozpoczęcia więcej niż jednej DTA, ponieważ wszystkie te akcje korzystają ze współdzielonego zasobu jakim jest model CNN. Dzięki temu niwelujemy ryzyko konfliktów podczas zapisywania wyników akcji.

Dodatkowo wszystkie powyższe komponenty odwołują się do pliku konfiguracyjnego, który zawiera między innymi ścieżki do pozostałych plików. Dokładniejszy opis tego pliku wraz z przedstawieniem jego interfejsu znajduje się w sekcji 2.5.

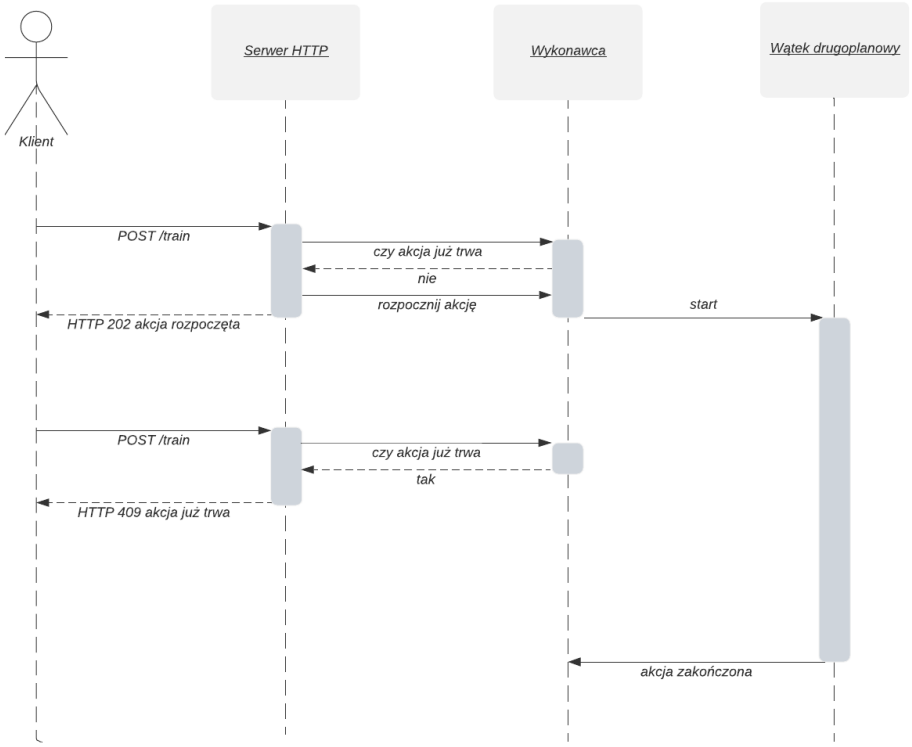
Przetwarzanie zapytań

Sposób w jaki serwer przetwarza zapytanie jest uzależniony od rodzaju akcji jaką musi podjąć. Możemy wyróżnić dwa typy akcji:

- akcja natychmiastowa - serwer jest gotowy niemal natychmiastowo wykonać żądaną akcję, w przypadku powodzenia zwracany jest kod odpowiedzi 200, informujący o wykonaniu akcji z sukcesem. Tego typu akcje najczęściej polegają na odczycie

i przetworzeniu zapisanych w plikach wynikowych danych (modyfikowanych przez długo trwające akcje),

- długo trwająca akcja (DTA) - akcja wymaga długiego przetwarzana, aby nie blokować serwera zwracany jest kod 202 informujący o tym, że serwer zaakceptował żądanie, natomiast jego przetwarzanie nie zostało jeszcze zakończone. Wykonanie akcji zostaje oddelegowane do nowo utworzonego wątku, a klient może kontrolować stan akcji przez skorzystanie z udostępnionego przez serwer API.



Rysunek 2.5: Diagram sekwencyjny, żądanie wykonania akcji przed zakończeniem przetwarzania poprzedniego zapytania

Rysunek 2.5 przedstawia diagram sekwencyjny opisujący zachowanie systemu w przypadku otrzymania żądania od klienta w trakcie przetwarzania długo trwającej akcji. Klient odwołuje się do API serwera przez wysłanie żądania z metodą POST wskazując na zasób *train*.

Tabela 2.1: Zlecenie utworzenia wątku który wykona DTA

```
def run_action(self, action, executable, **kwargs):
    """Add action to execution in background thread

    Args:
        action (Action): enumeration that defines action type
        executable (obj): function that will be executed in separate thread
    """
    self._fail_if_ongoing_action(action)
    executor.submit_stored(action, executable, **kwargs)
    LOG.info(f"New action ({action.value}) added to execution")
```

Serwer rozpoczyna przetwarzanie DTA, co w głównej mierze sprowadza się do wywołania metody `run_action()`, którą przedstawia tabela 2.1, wraz z typem akcji oraz referencją do obiektu funkcji, która ma zostać wywołana w drugoplanowym wątku. Przed utworzeniem nowego wątku serwer sprawdza czy DTA jest w trakcie wykonywania, jeśli tak się stało to zostanie rzucony wyjątek, a serwer zwróci kod odpowiedzi 409 wraz z informacją że akcja jest w trakcie przetwarzania.

2.4 Interfejs programistyczny

Interfejs programistyczny serwera przedstawiono na rysunku 2.6. Został on zdefiniowany w pliku YAML zgodnie z dokumentacją OpenAPI, która pozwala na zaprojektowanie bardzo dobrze udokumentowanego interfejsu opartego o zasady dobrego projektowania REST [5].

Punkty końcowe zostały logicznie wydzielone na pięć sekcji ze względu na funkcjonalność którą zapewniają:

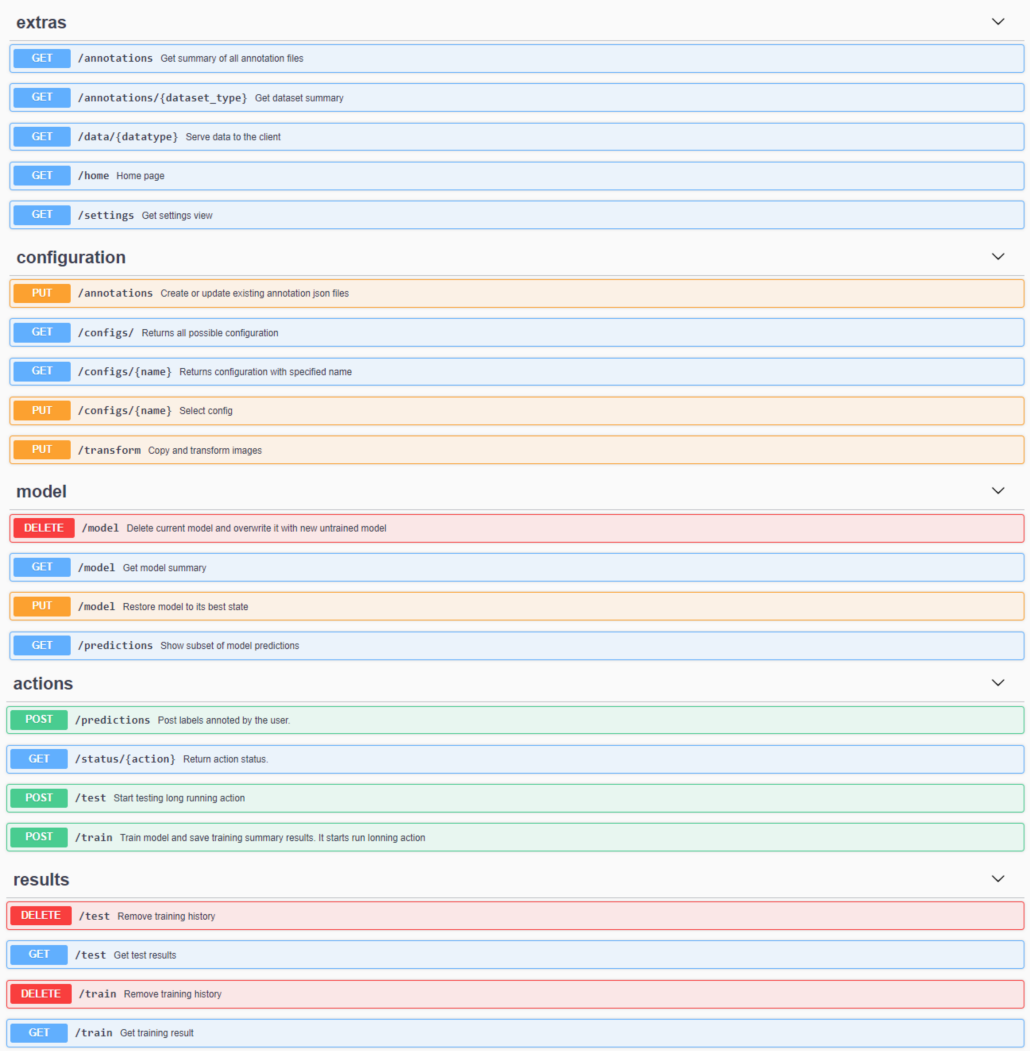
- *extras* - pełnią rolę pomocniczą, służą do generowania widoków lub zapewniają dostęp do pewnych danych,
- *configuration* - zdefiniowane tutaj punkty końcowe udostępniają możliwość zarządzania konfiguracją serwera oraz umożliwiają wstępne przygotowanie zbiorów danych i niezbędnych plików,
- *model* - określone tutaj metody zapewniają interfejs do zarządzania stanem modelu sieci neuronowych,
- *actions* - sekcja ta zawiera punkty końcowe, które rozpoczynają DTA. Wyjątkiem jest `GET /status/{action}`, który służy do sprawdzania statusu akcji,
- *results* - punkty końcowe umożliwiające pobieranie bądź usuwanie wyników procesu trenowania oraz testowania.

Przykładową dokumentację punktu końcowego przedstawiono na rysunku 2.7. Ścieżka `/status/{action}` została sparametryzowana, więc aby serwer przyjął żądanie GET do omawianej ścieżki należy określić obowiązkowy (ang. *requried*) parametr *action*, który jest typu string oraz przyjmuję jedną z trzech wartości: *train*, *test* lub *predictions*. W przypadku wysłania żądania, które naruszy schemat (ang. *schema*) określony w dokumentacji, serwer zwróci status odpowiedzi 400 wraz z opisem błędu. W sekcji *Responses* dostajemy szczegółowe informacje o możliwych odpowiedziach serwera. W przypadku braku wystąpienia błędu serwer powinien zwrócić kod odpowiedzi 200, a w ciele (ang. *body)* odpowiedzi powinniśmy się spodziewać danych w formacie *application/json*.

Integracja OpenAPI oraz modułu Flask

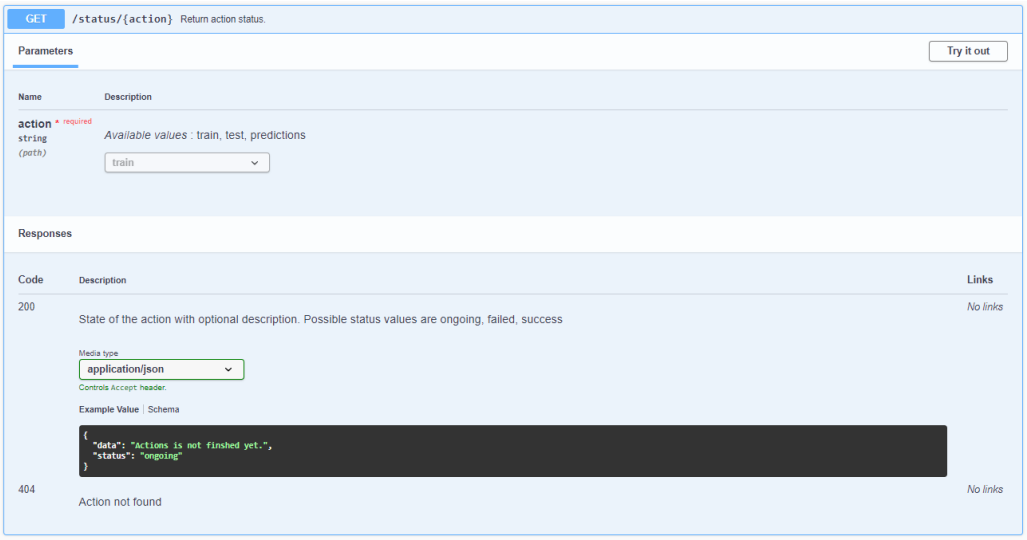
Zdefiniowane w dokumentacji punkty końcowe zostały odwzorowane na odpowiednie funkcje obsługujące konkretne metody HTTP za pomocą modułu `connexion`. Zapewnia on walidację każdego żądania oraz odpowiedzi serwera z użyciem schematów JSON określonych w pliku `openapi.yml`, zawierającym definicję API. Przykładowo dla ścieżki `/train` określono trzy metody GET, POST oraz DELETE, a schemat widoku implementującego obsługę żądań przedstawiono w tabeli 2.2.

Moduł `connexion` automatycznie przekazuje obsługę żądania zgodnie z zasadami:



Rysunek 2.6: Interfejs programistyczny, wizualizacja z użyciem SwaggerUI [12]

- widok obsługujący daną ścieżkę musi nosić nazwę «PunktKońcowy»View,
- widok musi mieć zdefiniowane metody odpowiadające nazwą metod HTTP zdefiniowanych w pliku zawierającym dokumentację OpenAPI. Wyjątkiem jest metoda `search()`, która odpowiada obsłudze żądania GET bez sprecyzowania konkretnego zasobu (metoda `get()` jest zarezerwowana do obsługi żądań sparametryzowanego zasobu, przykładem może być `GET /status/{action}`),
- parametry żądań zdefiniowane w OpenAPI są automatycznie przekazywane do atrybutu `view_args` obiektu `request` z modułu Flask, co skutkuje możliwością zdefiniowania ich jako parametry metody obsługującej żądanie.



Rysunek 2.7: Dokumentacja punktu końcowego `GET /status/{action}`

Tabela 2.2: Przykład implementacji widoku dla ścieżki `/train`

```
class TrainView(ActionView):
    def search(self, nepochs, reverse):
        ...

    def post(self, epochs=None, batch_size=None, query=None):
        ...

    def delete(self):
        ...
```

2.5 Przygotowanie danych i modelu

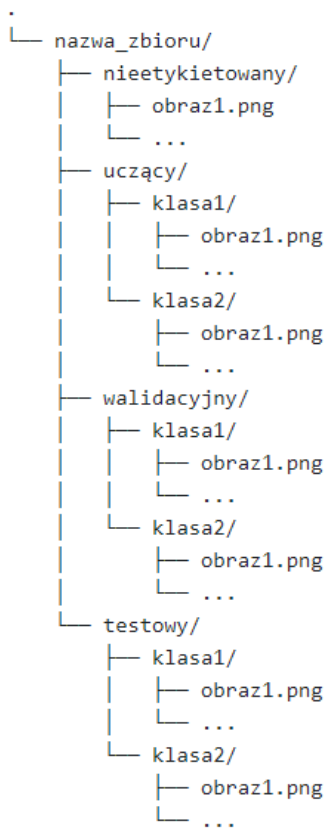
Aplikacja umożliwia obserwowanie procesu trenowania na dowolnym zbiorze danych który spełnia pewne kryteria:

- zbiór zawiera dwie klasy obrazów,
- obrazy podzielono na podzbiory zgodnie ze strukturą zaprezentowaną na rysunku 2.8, gdzie zbiór uczący może być pusty,
- obrazy muszą być w formacie obsługiwanym przez bibliotekę Pillow [23].

Podział zbioru danych

Zbiór danych powinien być podzielony na cztery podzbiory:

- nieetykietowany - zbiór obrazów na którym model będzie dokonywał predykcji, a następnie użytkownik będzie mógł poprawić błędnie sklasyfikowane obrazy. W wyniku, dzięki interakcji użytkownika obrazy z tego zbioru otrzymają etykiety i przejdą do zbioru uczącego,
- uczący - obrazy z tego zbioru będą używane do trenowania modelu, system dopuszcza aby wstępnie był on pusty. Obrazy te powinny być reprezentatywną próbą [24], co pozwoli wytrenowanemu modelowi na poprawną generalizacje problemu,



Rysunek 2.8: Przykładowo struktura katalogowa zbioru danych

- walidacyjny - zbiór, używany w celu śledzenia procesu uczenia się modelu. Aplikacja używa zbioru walidacyjnego aby obliczyć skuteczność i wartość funkcji kosztu po każdej epoce. Zbiór ten nie powinien zawierać obrazów, których użyto w zbiorze uczącym,
- testowy - zbiór obrazów używany po zakończeniu procesu trenowania, pozwala on sprawdzić jak dobrze model potrafi generalizować dany problem.

Plik konfiguracyjny

Plik ten zawiera parametry konfiguracyjne zbiorów danych, w szczególności ścieżki pod którymi serwer może znaleźć obrazy, pliki zawierające etykiety przypisane obrazom oraz ścieżki do modeli. Przykładową konfigurację przedstawiono w tabeli 2.3. W jednym pliku konfiguracyjnym można zdefiniować wiele konfiguracji dla różnych zbiorów danych, a przełączanie pomiędzy zbiorami jest możliwe z poziomu interfejsu użytkownika bez konieczności ponownego uruchomienia serwera.

Tabela 2.3: Przykładowa zawartość pliku `config.yml`

```yaml
Cars_vs_Aircrafts:
  dataset: Cars vs Aircrafts
  description: Cars and Aircrafts dataset.

  general:
    model_name: resnet34
    image_size: 256
    label_mapping:
      Aircraft: 0
      Car: 1
    unknown: 255

  paths:
    server_base: ['.', 'mlvt', 'server', 'static', 'cars_aircrafts']
    data_base: ['.', 'data', CarsAircrafts]
    annotations: # server_base prefix will be added
      dir: [annotations]
      unl: unl.json
      train: train.json
      test: test.json
      validation: validation.json
    models: # server_base prefix will be added
      best: [models, cars_aircrafts_best.pt]
      training: [models, cars_aircrafts_training.pt]
    images:
      raw:  # data_base prefix will be added
        unl: [unlabelled]
        test: [test]
        train: [train]
        validation: [valid]
      transformed:  # server_base prefix will be added
        unl: [images_tranformed]
        test: [images_test]
        train: [images_train]
        validation: [images_valid]
    data:  # server_base prefix will be added
      train_results: [data, train_results.json]
      test_results: [data, test_results.json]
      predictions: [data, predictions.json]
      last_user_test: [data, last_user_test.json]

  train:
    predictions: 10
    epochs: 10
    batch_size: 128

  test:
    outputs: 18
```

Najważniejsze sekcje pliku konfiguracyjnego:

- *general* - zawiera nazwę modelu który ma zostać użyty (wspierane architektury przedstawiono w podrozdziale 2.6) oraz odwzorowanie pomiędzy nazwami klas a ich numeryczną reprezentacją, dodatkowo zdefiniowana jest tutaj docelowa wielkość obrazów po zastosowaniu ich wstępnego przetwarzania,

- *paths* - ścieżki do plików, podane w formie list, co zapewnia niezależność od różnych systemów operacyjnych. Ścieżki prowadzące do katalogów z obrazami powinny posiadać podkatalogi o nazwach odpowiadających nazwom klas zdefiniowanych w sekcji *general* (nie dotyczy to zbioru nieetykietowanego),

  - *annotations* - sekcja ta określa ścieżki do plików w formacie JSON zawierających etykiety dla wszystkich obrazów z poszczególnych zbiorów. Poglądową zawartość takiego pliku prezentuje tabela 2.4. Pliki te są tworzone przez serwer

korzystając z funkcji `os.path.join()`, co zapewnia niezależność od systemu operacyjnego,

  - *data* - pliki w formacie JSON, które zawierają dane wynikowe DTA. Przykładowy plik zawierający historię trenowania przedstawiono w tabeli 2.5,

- *train* - zawiera wartości domyślne parametrów użytych podczas trenowania oraz predykcji modelu. Aplikacja umożliwia ich nadpisanie z poziomu interfejsu użytkownika.

Tabela 2.4: Plik `train.json` zawierający etykiety dla obrazów ze zbioru uczącego

```
{
"0": [
    ".\\mlvt\\server\\static\\cats_vs_dogs\\training\\5753.jpg",
    ".\\mlvt\\server\\static\\cats_vs_dogs\\training\\7863.jpg",
  ...],
"1": [
    ".\\mlvt\\server\\static\\cats_vs_dogs\\training\\4917.jpg",
    ".\\mlvt\\server\\static\\cats_vs_dogs\\training\\5287.jpg",
  ...]
}
```

Tabela 2.5: Plik `train_results.json` przechowujący historię procesu trenowania

```
{
    "n_images": [
        22,
        ...
    ],
    "train_acc": [
        0.3636363744735718,
        ...
    ],
    "train_loss": [
        0.7238350510597229,
        ...
    ],
    "val_acc": [
        0.5734679102897644,
        ...],
    "val_loss": [
        0.6725834421813488,
        ...
    ]
}
```

Do zarządzania zmiennymi zdefiniowanymi w pliku konfiguracyjnym napisano interfejs w postaci klasy `ConfigManager`. Za jego pomocą każdy z widoków ma dostęp do danych zawartych w tym pliku.

Wstępne przetwarzanie obrazów

Serwer udostępnia punkt końcowy (`PUT /transform`) umożliwiający przekopiowanie obrazów ze struktury katalogowej określonej w pliku konfiguracyjnym w sekcji *raw* do struktury katalogowej określonej w sekcji *transformed*. Podczas kopiowania następuje zmiana wielkości obrazów do rozmiaru wymaganego przez warstwę wejściową modelu, co pozwala zaoszczędzić znaczącą ilość czasu. Dodatkowo, przed przekazywaniem obrazów do modelu, obrazy są normalizowane wartościami zalecanymi przez dokumentację PyTorch,

a podczas trenowania dodatkowo obrazy są losowo odwracane [25] w celu ograniczenia zjawiska przetrenowania.

2.6 Model splotowych sieci neuronowych

W praktycznych zastosowaniach, tworzenie od podstaw modelu splotowych sieci neuronowych bardzo rzadko ma miejsce, ponieważ mało kto dysponuje wystarczająco dużymi zasobami obliczeniowymi oraz odpowiednio dużym zbiorem obrazów aby wytrenować taki model. Zdecydowanie częściej korzysta się z modeli wstępnie przetrenowanych na pewnym dużym zbiorze obrazów, a następnie dostosowuje się jedynie ostatnie warstwy takiego modelu do własnego problemu. Taki proces znany jest pod nazwą *transfer learning*.

Aplikacja wspiera użycie następujących przetrenowanych modeli splotowych sieci neuronowych:

- Alexnet [26] - architektura z 2012 roku, która zapoczątkowała rewolucje głębokich splotowych sieci neuronowych,
- MobileNets [27] - architektura splotowych sieci neuronowych zaprojektowana z myślą o urządzeniach mobilnych,
- ResNet [3] - model resztkowych sieci neuronowych, który dzięki wprowadzeniu dodatkowych połączeń między warstwami, ogranicza problem zanikającego gradientu, dzięki czemu jego optymalizacja jest łatwiejsza. Aplikacja umożliwia użycie modelu ResNet z 18, 34, 50 lub 104 warstwami splotowymi.

W dalszej części pracy, jeżeli użyty model nie został sprecyzowany, należy zakładać że wykorzystano przetrenowany model sieci resztkowych o 34 warstwach splotowych. Model ten jest kompromisem pomiędzy skutecznością a wymaganymi zasobami obliczeniowymi [16].

Przetrenowany model zdefiniowano jako komponent klasy opakowującej (ang. *wrapper class*) `Model` co przedstawiono w tabeli 2.6. Model domyślnie będzie korzystał z zasobów procesora graficznego, jeżeli system na którym uruchomiono serwer ma zainstalowaną bibliotekę CUDA. W trakcie inicjalizacji przetrenowanego modelu następuje zamrożenie wszystkich warstw splotowych [18], pozwala to na przyspieszenie procesu trenowania bez znaczącego wpływu na skuteczność modelu.

Tabela 2.6: Inicjalizacja przetrenowanego modelu wewnątrz klasy `Model`

```
class Model:
    def __init__(self, training_model_path, best_model_path,
            model_name=ModelType.RESNET_34.value,  state=None, n_out=2,
            lr=1e-3, gamma=0.5, dropout=0.2, overwrite=False, milestones=None):
        self.device = torch.device(
            "cuda:0" if torch.cuda.is_available() else "cpu")
        ...

        model = self._get_pretrained_model(model_name)
        self.model_conv = model.to(self.device)
        self.criterion = nn.CrossEntropyLoss().to(self.device)
        self.init_optimizer(lr)

    def _get_pretrained_model(self, model_name):
        # initialize pretrained model from torchvision
        model = getattr(models, f"{model_name}")(pretrained=True)

        # freeze conv layers
        for param in model.parameters():
            param.requires_grad = False

        last_layer = lambda num_ftrs: nn.Sequential(
            nn.Dropout(self.dropout),
            nn.Linear(num_ftrs, self.n_out))

        # define last layer based on model type
        if "mobilenet" in model_name:
            model.classifier[1] = last_layer(model.classifier[1].in_features)

        elif "resnet" in model_name:
            model.fc = last_layer(model.fc.in_features)

        elif model_name == "alexnet":
            model.classifier[6] = last_layer(model.classifier[6].in_features)

        else:
            raise ValueError(f"Model ({model_name}) is not supported")

        return model
```

Następnie podmieniana jest ostatnia warstwa modelu na blok sekwencyjny składający się z dwóch warstw:

- `nn.Dropout` - warstwa odpowiadająca za regularyzacje i pozwalająca na ograniczenie problemu przeuczenia modelu. Jej działanie opiera się na usuwaniu sygnałów wyjściowych neuronów w trakcie uczenia sieci z prawdopodobieństwem zdefiniowanym w kodzie jako *dropout*,
- `nn.Linear` - warstwa gęsta, której wyjściem jest liniowe przekształcenie wejścia. Pierwszym argumentem jest liczba neuronów na wejściu tej warstwy, a drugim liczba neuronów na wyjściu, która w przypadku klasyfikatora binarnego wynosi 2.

Ze względu na zróżnicowaną implementację modeli w PyTorch, dostęp do ostatniej warstwy dla poszczególnych modeli wymaga odwołania się do różnych atrybutów. W tym celu zdefiniowano wyrażenie `lambda` przyjmujące liczbę neuronów na wyjściu ostatniej splotowej warstwy, a jej wynik jest odpowiednio przypisywany dla poszczególnych modeli. Ze względu na użycie entropii krzyżowej (ang. *Crossentropy*) jako funkcji kosztu oraz jej wewnętrzną implementacje przez platformę PyTorch, nie należy normalizować wartości wyjściowych modelu przy użyciu funkcji softmax. W celu znalezienia minimum funkcji kosztu użyto optymalizacji Adam [28], która jest wariacją SGD (ang. *Stochastic Gradient Descent*). Adam adaptacyjnie dostosowuje szybkość uczenia (ang. *learning rate*)

dla poszczególnych parametrów, wykorzystując średnią oraz wariancję gradientu danego parametru.

Wczytanie oraz zapis modelu

Obiekt klasy `Model` jest tworzony tylko dla żądań rozpoczynających DTA, a po jej zakończeniu zasoby są zwalniane. Interfejs klasy `Model` umożliwia wczytanie parametrów modelu oraz optymalizatora, co umożliwia zachowanie stanu modelu pomiędzy kolejnymi żądaniami HTTP. Aplikacja przechowuje dwa stany modelu:

- bieżący - stan ten jest aktualizowany po każdej epoce w trakcie procesu trenowania, umożliwia to aktualizowanie widoku klienta w czasie rzeczywistym bez konieczności oczekiwania na zakończenie długo trwającej akcji,
- o najwyższej skuteczności - stan modelu o najwyższej osiągniętej skuteczności na zbiorze walidacyjnym. Nadpisania stanu bieżącego stanem o najwyższej skuteczności jest możliwe za pomocą punktu końcowego `PUT /model`.

Sposób w jaki zapisywany jest stan modelu przedstawiono w tabeli 2.7.

Tabela 2.7: Zapis stanu modelu

```
def save_state(self, acc, is_best, epoch):
    """Save state of the model and optimizer.

    Args:
        acc (float): current model accuracy
        is_best (bool): True if model achieved its best accuracy
        epoch (int): training epoch
    """
    state = {
        'model': self.model_conv.state_dict(),
        'optimizer': self.optimizer.state_dict(),
        'acc': acc,
        'epoch': epoch
    }
    torch.save(state, self.training_model_path)
    if is_best:
        shutil.copyfile(self.training_model_path, self.best_model_path)
```

Na stan modelu składają się:

- wszystkie parametry modelu, które są optymalizowane w trakcie procesu uczenia,
- parametry optymalizatora wraz z użytymi hiperparametrami,
- skuteczność modelu,
- numer epoki.

Klient ma możliwość nadpisania stanów modelu losowymi wartościami. Funkcjonalność ta jest realizowana za pomocą punktu końcowego `DELETE /model`. W wewnętrznej implementacji dokonywane jest to dzięki przekazaniu `True` jako wartość parametru `overwrite` w konstruktorze klasy `Model`.

Tabela 2.8: Opis długo trwających akcji z poziomu interfejsu klasy `Model` oraz z punktu widzenia klienta

metoda	ścieżka	opis
`predict_all( self, unl_loader)`	`GET /predictions? new_predictions=true`	Model dokonuje predykcji wszystkich obrazów ze zbioru nieetykietowanego. Wynik tej operacji jest zapisywany w pliku, co zapewnia szybki dostęp do tych danych w kolejnych żądaniach.
`train( self, train_loader, validation_loader, epochs, ...)`	`POST /train`	Model rozpoczyna proces trenowania z użyciem zbioru uczącego. Po każdej epoce aktualizowany jest plik zawierający wyniki treningu oraz zapisywany jest bieżący stan modelu, a w przypadku uzyskania najlepszej skuteczności dodatkowo nadpisywany jest stan najlepszego modelu.
`test( self, test_loader)`	`POST /test`	Model rozpoczyna proces testowania modelu na zbiorze testowym. Po zakończeniu tej akcji wyniki są zapisywane w pliku wynikowym.

Akcje wykonywane przez model

Tak jak wspomniano w podrozdziale 2.3, część żądań rozpoczyna DTA. Wszystkie te akcje związane są z przetwarzaniem obrazów przez model, a ich podsumowanie zostało zawarte w tabeli 2.8.

Dostęp do obrazów

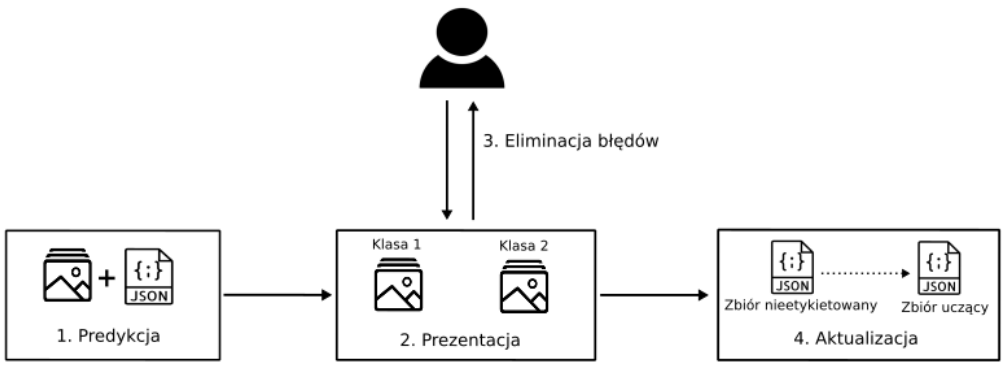
Każda z przedstawionych metod, jako jeden z parametrów przyjmuje instancję klasy `DataLoader`, która umożliwia iteratywne wczytywanie obrazów w paczkach. Ich wielkość została określona przez parametr `batch_size`, który jest konfigurowalny z poziomu interfejsu graficznego, a jego wartość domyślna pobierana jest z pliku konfiguracyjnego. Podczas inicjalizacji obiektu klasy `DataLoader`, należy podać referencję do obiektu `Dataset` zapewniającego interfejs pozwalający na dostęp do danych. Taki interfejs musi mieć zdefiniowane dwie metody: `__len__`, zwracającą liczbę danych w zbiorze oraz `__getitem__` pozwalający na dostęp do danych z użyciem indeksu. W aplikacji zdefiniowano klasy odpowiadające za wczytywanie obrazów, które dziedziczą po klasie `Dataset`. Tak jak to przedstawiono w tabeli 2.9, obrazy nie są przetrzymywane w pamięci, a na bieżąco wczytywane na podstawie ścieżek wczytanych do zmiennej `all_annotations`. Po wczytaniu obrazu aplikowane są na nim pewne transformacje, najczęściej jest to konwersja do typu `torch.Tensor` oraz normalizacja, a dla zbioru uczącego dodatkowo losowe odwrócenie.

Tabela 2.9: Implementacja metody zapewniającej dostęp do obrazów za pomocą indeksu

```
def __getitem__(self, idx):
    """Access data at specific index from dataset.

    Args:
        idx (int): data index

    Returns:
        tuple: contains loaded image and corresponding label with optionally added path
    """
    img_path = self.all_annotations[idx]
    img = Image.open(img_path).convert('RGB')
    target_label = self._get_label(img_path)

    if self.transforms:
        img = self.transforms(img)

    return (img, target_label, img_path) if self.return_paths \
        else (img, target_label)
```

2.7 Interaktywne przypisywanie etykiet nieoznaczonym obrazom

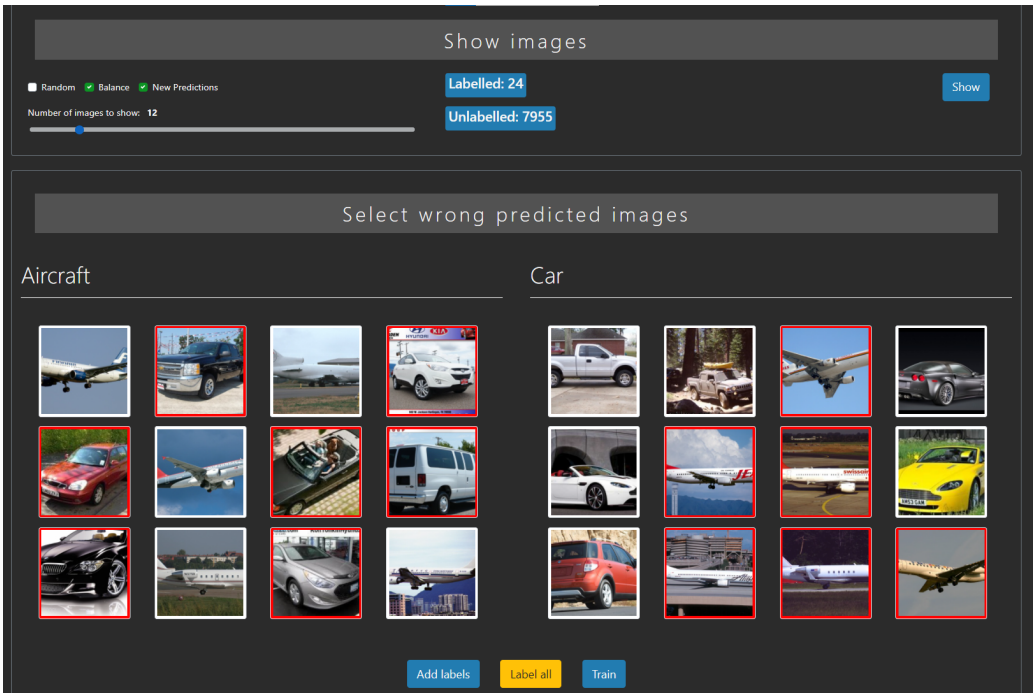
Aplikacja udostępnia możliwość dynamicznego tworzenia zbioru uczącego, co pozwala na korzystanie z aplikacji mimo wstępnego braku obrazów w zbiorze uczącym. Schemat działania tego mechanizmu przedstawiono na rysunku 2.9.



Rysunek 2.9: Schemat interaktywnego tworzenia zbioru uczącego

1. Predykcja - model, wykorzystując plik zawierający ścieżki do obrazów nieetykietowanych dokonuje predykcji. W wyniku otrzymywany jest plik widoczny w tabeli 2.10, który zawiera odwzorowanie pomiędzy ścieżkami do obrazów a wartościami otrzymanymi na wyjściu modelu (wartości te są dodatkowo przekazywane do funkcji softmax, aby w wyniku otrzymać prawdopodobieństwo danej klasy) dla każdego obrazu. Użytkownik ma kontrolę nad tym jakie oraz ile obrazów będzie mu prezentowanych,

2. Prezentacja - użytkownikowi prezentowane są obrazy wraz z etykietami przyznanymi przez model,



Rysunek 2.10: Widok aplikacji umożliwiający interaktywne tworzenie zbioru uczącego

3. Eliminacja błędów - użytkownik ma możliwość zaznaczenia błędnie sklasyfikowanych obrazów, a następnie dodania ich do zbioru uczącego,

4. Aktualizacja - podpisywanie nieetykietowanych obrazów sprowadza się do usunięcia ze zbioru nieetykietowanego ścieżek, które prowadzą do nowo podpisanych obrazów oraz dodanie tych ścieżek do pliku opisującego zbiór uczący wraz z etykietami przyznanymi przez użytkownika. Obrazy nie są przenoszone pomiędzy katalogami, zmiany są dokonywane tylko w plikach JSON. Dodatkowo ścieżki usunięte ze zbioru nieetykietowanego muszą zostać również usunięte z pliku, który zawiera ostatnią predykcję modelu.

Tabela 2.10: Przykład pliku zawierającego zestaw ostatnich predykcji modelu

```
{
    "paths": [
        ".\\mlvt\\server\\static\\cats_vs_dogs\\images_tranformed\\6790.jpg",
        ".\\mlvt\\server\\static\\cats_vs_dogs\\images_tranformed\\7555.jpg",
        ...
    ],
    "predictions": [
        [
            0.9887653589248657,
            0.011234634555876255
        ],
        [
            0.2735075056552887,
            0.7264925241470337
        ],
        ...
    ]
}
```

Rysunek 2.10 przedstawia interfejs użytkownika umożliwiający interaktywne tworzenie zbioru uczącego. W górnej części widoku oznaczonego nagłówkiem *Show images* użytkownikowi prezentowana jest aktualna liczba obrazów w zbiorach uczącym (etykieta *Labelled*) oraz nieetykietowanym (etykieta *Unlabelled*). Za pomocą panelu po lewej stronie użytkownik może kontrolować jakie parametry zostaną wysłane do serwera w żądaniu `GET /predictions?...`, a ich podsumowanie zostało zawarte w tabeli 2.11. W dolnej części rysunek 2.10 przedstawia obrazy zaklasyfikowane przez model do jednej z dwóch klas. Użytkownik może zaznaczyć błędnie sklasyfikowane obrazy (po zaznaczeniu kolor obramowania zmienia się na czerwono) a następnie za pomocą przycisku *Add labels* dodać obrazy do zbioru uczącego.

Tabela 2.11: Parametry ustawiane przez użytkownika, które określają jakie obrazy zostaną wyświetlone

nazwa parametru	opis
`random`	Ustawienie tej flagi na `false` (pole odznaczone) powoduje wyświetlenie użytkownikowi obrazów co do których model jest najbardziej niepewny. Jako miarę niepewności uznano najmniejszą bezwzględną różnicę pomiędzy wartościami przewidywanych klas. Taka strategia wyboru jest wykorzystywana w uczeniu aktywnym [29]. W przeciwnym przypadku wyświetlane są losowe obrazy.
`balance`	Ustawienie tego parametru na `true` powoduje próbę wyświetlenia takiej samej liczby obrazów dla obu klas. Wyjątkiem jest sytuacja w której model nie przypisał wystarczającej liczby etykiet którejś z klas.
`new_predictions`	Ustawienie tej flagi na `true` powoduje rozpoczęcie DTA, której wynikiem jest plik zawierający predykcje modelu dla wszystkich obrazów ze zbioru nieetykietowanego. W przeciwnym przypadku serwer skorzysta z zapisanej ostatniej predykcji modelu.
`n_images`	Liczba obrazów z przedziału od 0 do 100 określająca ile obrazów ma być wyświetlonych po przetworzeniu żądania. Wartość ta dotyczy tylko jednej z klas więc przy ustawieniu flagi `balance` liczba wyświetlanych obrazów jest podwojona. W momencie przetwarzania, wartość tego parametru może zostać zmniejszona, jeżeli w zbiorze nieetykietowanym nie ma wystarczającej liczby obrazów.

Po rozszerzeniu zbioru uczącego, natychmiastowo prezentowane są użytkownikowi kolejne obrazy, co jest możliwe dzięki zapisowi ostatniej predykcji modelu. Alternatywnie, użytkownik może dodać wszystkie wszystkie obrazy ze zbioru nieetykietowanego do zbioru uczącego, korzystając z ostatniej predykcji modelu, a jest to możliwe za pomocą przycisku *Label all*.

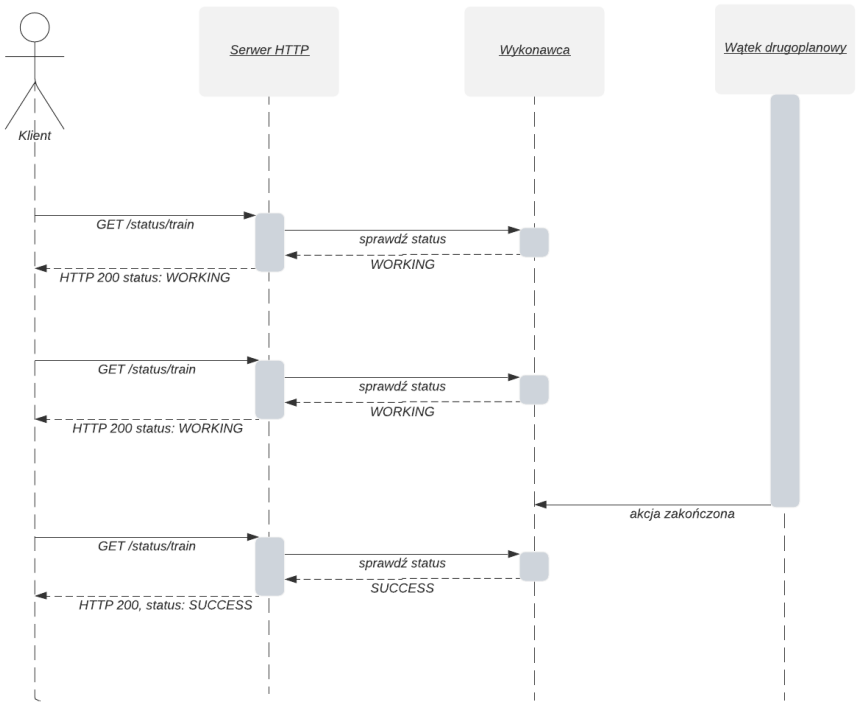
2.8 Wizualizacja procesu uczenia modelu

Główną funkcjonalnością aplikacji jest umożliwienie śledzenia procesu uczenia klasyfikatora binarnego. W celu implementacji wykorzystano bibliotekę Plotly oraz technikę AJAX. Połączenie to umożliwia obserwację procesu uczenia w czasie rzeczywistym bez konieczności przeładowywania widoku przeglądarki.

Rysunek 2.11: Panel umożliwiający rozpoczęcie procesu trenowania

Rozpoczęcie procesu trenowania jest możliwe z poziomu interfejsu użytkownika za pomocą przycisku *Train* co zaprezentowano na rysunku 2.11. Użytkownik może określić liczbę epok przez którą model będzie się uczył oraz wielkość paczki jaka będzie użyta podczas wczytywaniu obrazów. Przycisk *Delete training history* umożliwia usunięcie historii uczenia.

Po odebraniu przez serwer żądania rozpoczęcia procesu trenowania, akcja ta jest oddelegowana do wątku drugoplanowego, a serwer zwraca kod odpowiedzi 202. Takie działanie zapewnia responsywność serwera na dalsze żądania.



Rysunek 2.12: Diagram sekwencyjny, periodyczne odpytywanie stanu akcji

Po rozpoczęciu akcji, klient zaczyna odpytywać serwer o stan akcji `train`, proces ten przedstawiono na rysunku 2.12. Każde żądanie stanu akcji skutkuje odpytaniem *Wykonawcy*, czyli modułu zarządzającego wątkami o to czy dana akcja się zakończyła.

Stan akcji jest typem wyliczeniowym przyjmującym jedną z trzech wartości:

- `ONGOING` - akcja nadal trwa,
- `SUCCESS` - akcja zakończyła się sukcesem,
- `FAILED` - akcja zakończyła się niepowodzeniem.

Odpowiedź serwera na żądanie `GET /status/{action}` jest obiektem typu JSON, który zawiera stan akcji oraz opcjonalnie dodatkowe dane, które są odbierane od procesu wykonującego daną akcję po jej zakończeniu. Zazwyczaj jest to informacja o rodzaju

błędu który wystąpił. Po odpytaniu o stan akcji klient wysyła również żądanie o aktualne dane treningowe w których skład wchodzi pięć list zawierających historię procesu uczenia (rysunek 2.5). Następnie klient sprawdza czy pojawiły się nowe dane względem ostatniego żądania, jeśli tak to następuje aktualizacja widoku trenowania za pomocą javascript, dzięki czemu strona nie musi być odświeżana.

Epoch	Training accuracy	Training loss	Validation accuracy	Validation loss	Images
1	0.5	0.8394	0.604	0.65553	10
2	0.7	0.474	0.553	0.67387	10
3	0.8	0.44774	0.535	0.69417	10
4	0.7	0.397	0.537	0.69073	10
5	0.7	0.3839	0.558	0.65791	10
6	0.8	0.35544	0.605	0.60598	10
7	0.8	0.30387	0.685	0.54424	10
8	0.9	0.26864	0.771	0.48493	10
9	1.0	0.21451	0.84	0.43513	10
10	1.0	0.18823	0.883	0.39531	10

Rysunek 2.13: Wizualizacja procesu uczenia za pomocą wykresu skuteczności (wykres górny) i wartości funkcji kosztu (wykres dolny) modelu oraz tabeli zawierającej dane z wszystkich epok

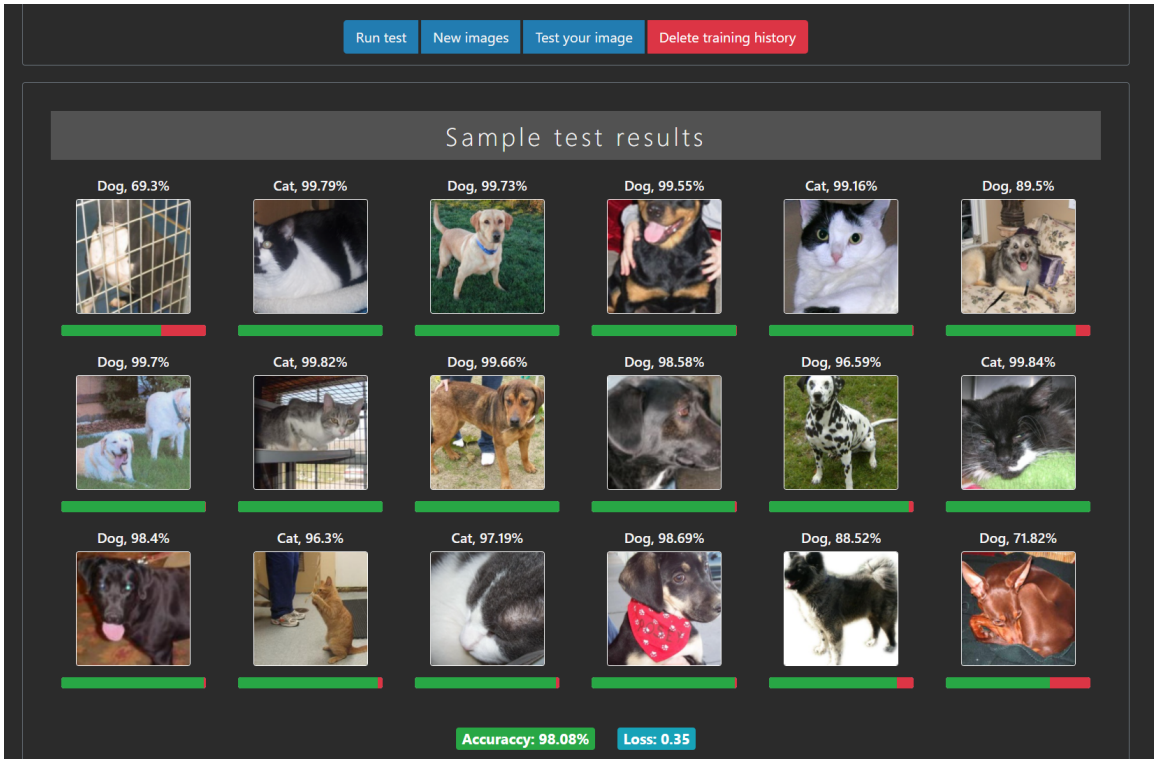
Proces uczenia modelu może być śledzony za pomocą tabeli lub wykresów, co przedstawiono na rysunku 2.13. Zarówno tabela jak i wykresy są dynamicznie aktualizowane.

Dodatkowo użytkownikowi przedstawiane są maksymalne skuteczności modelu oraz minimalne wartości funkcji kosztu osiągnięte na zbiorze uczącym (etykiety po lewej stronie widoku) oraz walidacyjnym (etykiety po prawej stronie widoku). Po najechaniu kursorem myszy na jedną z etykiet, klient otrzymuje informacje na której epoce uzyskano dana wartość.

Wykresy

Na obu prezentowanych wykresach przedstawiono trzy statystyki. Na wykresie górnym wizualizowana jest skuteczność modelu na zbiorze uczącym (niebieska linia) oraz na zbiorze walidacyjnym (czerwona linia). Z kolei wykres dolny przedstawia wartość funkcji kosztu na zbiorze uczącym oraz walidacyjnym, zaznaczone odpowiednio kolorem niebieskim oraz czerwonym. Dodatkowo na obu wykresach kolorem turkusowym zaznaczono liczbę obrazów użytych w procesie uczenia. Oś x w przypadku obu wykresów przedstawia liczbę epok. Zaprezentowane wykresy są w pełni interaktywne, umożliwiają one odczytanie każdego punktu na wykresie, a także dowolne ich skalowanie, czy też usunięcie którejś ze statystyk.

Testowanie modelu



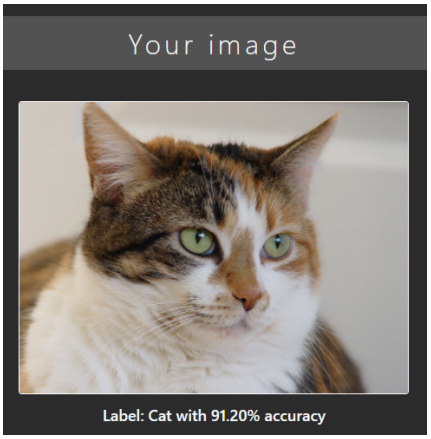
Rysunek 2.14: Interfejs użytkownika umożliwiający testowanie modelu

Serwer udostępnia dwa sposoby na przetestowanie modelu:

- test modelu na zbiorze testowym,
- test z użyciem dowolnego obrazu wysłanego do serwera.

Na rysunku 2.14 zaprezentowano widok strony umożliwiającej testowanie modelu. Przycisk *Run test* rozpoczyna DTA, a po jej zakończeniu użytkownikowi jest prezentowane powiadomienie z prośbą o przeładowanie strony. Procentowa skuteczność modelu

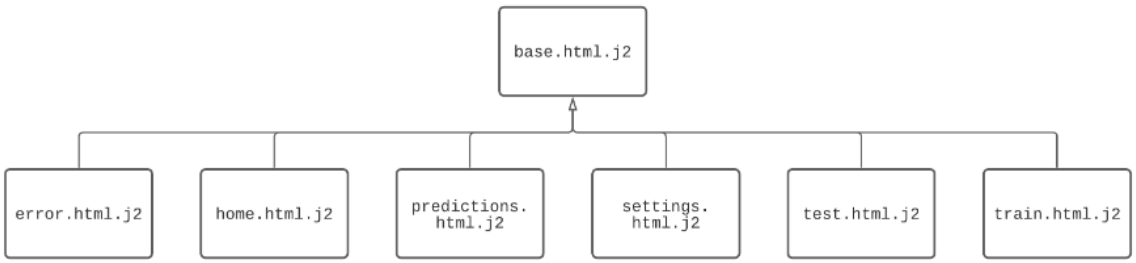
oraz wartość funkcji kosztu jest prezentowana u dołu ekranu. W głównej części widoku prezentowane są obrazy, których liczba może być ustalona w pliku konfiguracyjnym, wraz z etykietami przyznanymi przez model oraz wartością funkcji softmax modelu względem danej klasy w postaci numerycznej oraz zielono-czerwonego wskaźnika. Przycisk *New images* pozwala na wyświetlenie kolejnego podzbioru obrazów bez konieczności przeładowania strony. Przycisk *Test your image* pozwala na wysłanie do serwera własnego obrazu, a w odpowiedzi w przeglądarce zostanie wyświetlony obraz wraz z etykietą modelu, co przedstawiono na rysunku 2.15



Rysunek 2.15: Test modelu z użyciem własnego obrazu

2.9 Interfejs użytkownika

Implementacja interfejsu użytkownika opiera się na bibliotece Bootstrap [8], jQuery [30] oraz silniku szablonów Jinja2 [31]. Wykorzystując bibliotekę Bootstrap widok strony dostosowuje się do wielkości okna przeglądarki zgodnie z architekturą RWD. Jinja2 umożliwia dynamiczne generowanie plików HTML bazując na parametrach przekazanych z języka Python, co pozwala na odseparowanie warstwy prezentacji aplikacji.

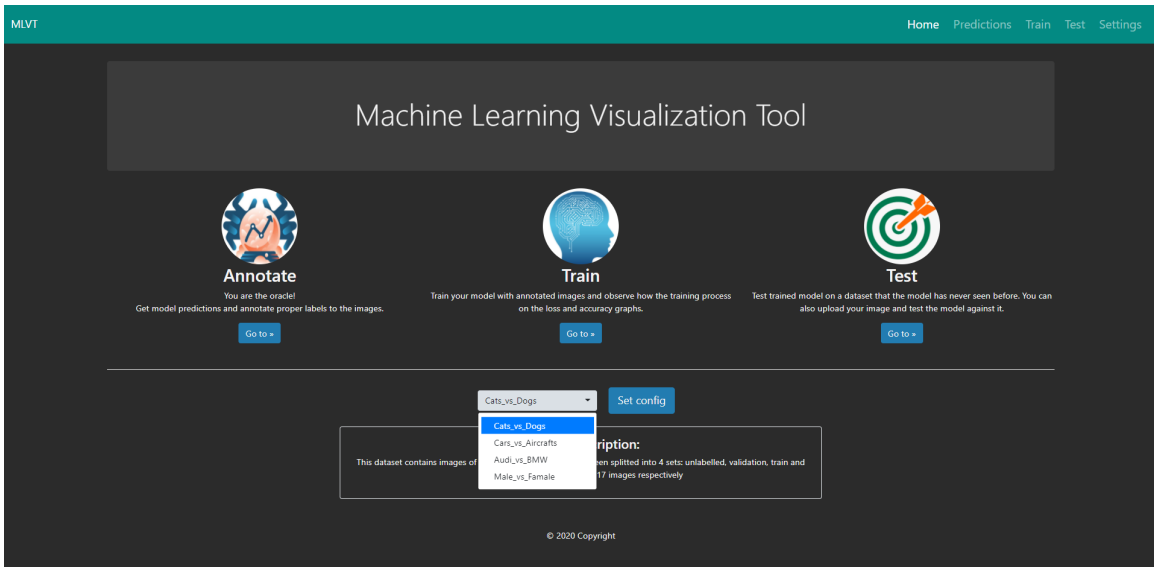


Rysunek 2.16: Hierarchia szablonów Jinja2

Zaimplementowane szablony przedstawiono na rysunku 2.16. Wszystkie szablony wykorzystywane do generowania pliku HTML dziedziczą po szablonie `base.html.j2`, w którym zdefiniowano wspólne elementy dla wszystkich widoków, jak na przykład pasek nawigacyjny (ang. *navbar*). Przy tworzeniu szablonów wykorzystano makra Jinja2, które zdefiniowano w pliku `macros.html.j2`.

Prezentacja widoków

W podrozdziałach 2.7 i 2.8 zaprezentowano interfejs użytkownika generowany z użyciem odpowiednio szablonów `train.html.j2` oraz `test.html.j2`, dlatego w tym podrozdziale zostanie to pominięte. Na rysunku 2.17 zaprezentowano widok strony głównej



Rysunek 2.17: Strona domowa aplikacji

dostępny pod ścieżką `/home`. Oprócz funkcji nawigacyjnej, widok ten umożliwia wybór konfiguracji z której chcemy korzystać, aby wczytać wybraną konfigurację należy użyć przycisku *Set config*. Opis wybranej konfiguracji jest generowany na podstawie wartości obiektu *description* w pliku konfiguracyjnym.

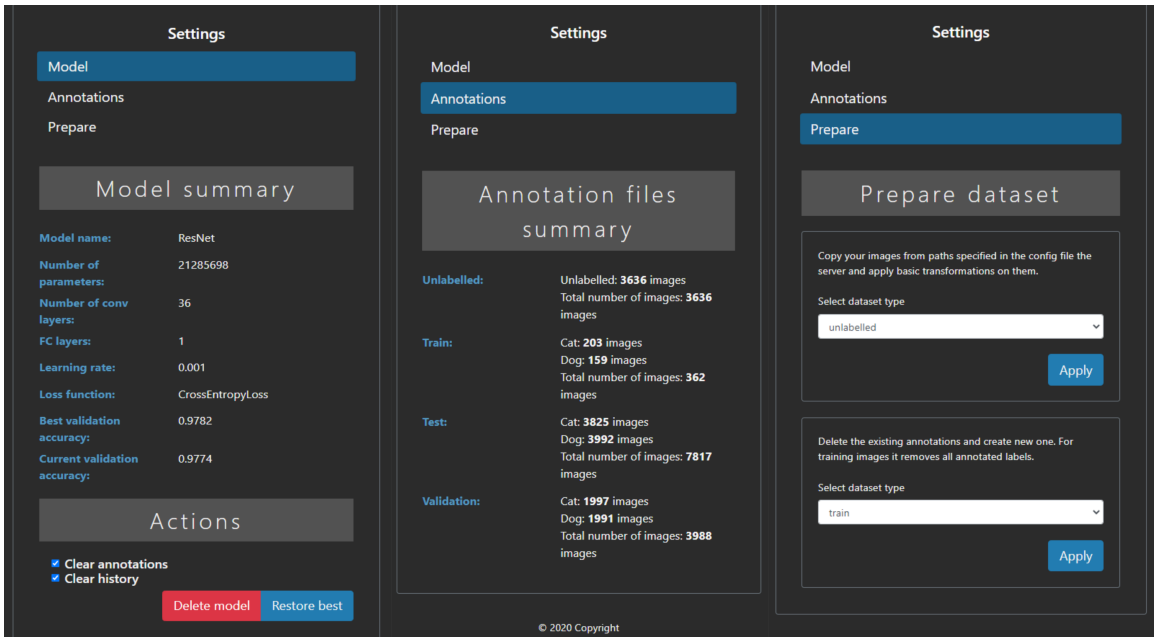
Szablon `settings.html.j2` użyto do wygenerowania widoku dla punktu końcowego `GET /settings`, którego podsumowanie zaprezentowano na rysunku 2.18. Każdy z trzech złączonych ze sobą widoków został wygenerowany dla szerokości okna przeglądarki równej 630 pikseli, co potwierdza zgodność z architekturą RWD.

Widok *Settings* przedstawia krótkie podsumowanie obecnie używanego modelu a także zapewnia interfejs umożliwiający inicjalizacje modelu losowymi wagami oraz przywrócenie modelu do stanu w którym osiągnął najwyższą skuteczność na zbiorze walidacyjnym. Żądania te przyjmują dwa opcjonalne parametry:

- *Clear annotations* - zaznaczenie tego pola powoduje ustawienie parametru żądania `clear_annotations` na wartość `true`, co spowoduje ponowne utworzenie wszystkich plików zawierających odwzorowanie pomiędzy ścieżkami do obrazów a etykietami im przypisanymi. W wyniku, obrazy które zostały podpisane przez użytkownika zostaną z powrotem umieszczone w zbiorze nieetykietowanym,

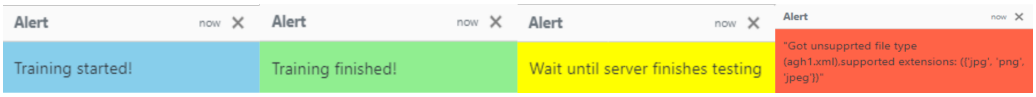
- *Clear history* - zaznaczenie tego pola powoduje ustawienie parametru żądania `clear_history` na wartość `true`, co spowoduje wyczyszczenie plików zawierających historię uczenia oraz testowania modelu.

Widok *Annotations* przedstawia liczbę obrazów w poszczególnych zbiorach, natomiast widok *Prepare* służy do przygotowaniu zbioru obrazów bazując na pliku konfiguracyjnym.



Rysunek 2.18: Podsumowanie widoku ustawień

Powiadomienia



Rysunek 2.19: Przykładowe powiadomienia generowane przez aplikację

W przypadku wystąpienia zdarzenia, jak na przykład rozpoczęcie DTA, jej zakończenie lub wystąpienie błędu aplikacja informuje o tym użytkownika za pomocą pojawiających się w prawym dolnym rogu ekranu powiadomień, które automatycznie znikają po 10 sekundach. Przykłady takich powiadomień przedstawiono na rysunku 2.19. Kolor powiadomień jest związany z otrzymanym przez przeglądarkę kodem odpowiedzi, co podsumowano w tabeli 2.12.

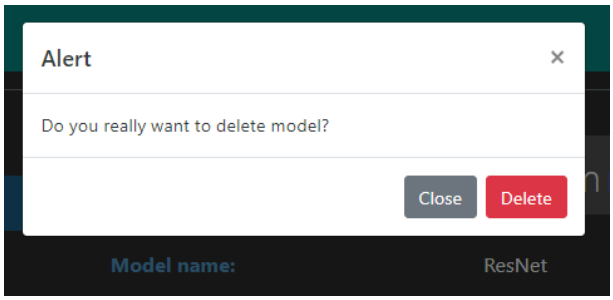
Tabela 2.12: Powiązanie pomiędzy kolorem powiadomienia a kodem odpowiedzi

kolor	kod odpowiedzi	przykład
niebieski	202	rozpoczęcie procesu uczenia
zielony	200	DTA zakończyła się sukcesem
żółty	409	inna DTA trwa
czerwony	400	wystąpił nieoczekiwany błąd w trakcie DTA

Zanim pewne akcje zostaną podjęte, generowane jest okno dialogowe pozwalające na zaniechanie wysłania żądania do serwera. Chroni to użytkownika przed przypadkowym podjęciem działania. Przykład takiego okna dialogowego przedstawiono na rysunku 2.20.

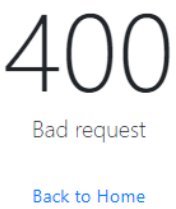
Sytuacje wyjątkowe

Serwer przetworzy tylko żądania które są zgodne z dokumentacja OpenAPI, w przypadku braku zgodności serwer zwróci plik HTML z informacją o błędzie oraz możliwością

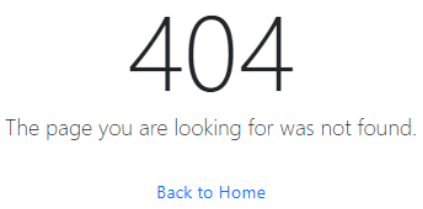


Rysunek 2.20: Przykładowe okno dialogowe, użytkownik musi potwierdzić chęć wykonania akcji aby żądanie zostało wysłane do serwera

powrotu do strony głównej co przedstawiono na rysunku 2.21.



(a) Żądanie nie jest zgodne z schematem zdefiniowanym w pliku `openapi.yml`



(b) Serwer nie obsługuje podanej w żądaniu ścieżki

Rysunek 2.21: Odpowiedź serwera na błędne żądanie

Rozdział 3

Analiza otrzymanych rezultatów

W niniejszym rozdziale zastaną przedstawione przykładowe scenariusze działania aplikacji wraz z ich omówieniem. Przed każdym scenariuszem model został zainicjalizowany losowymi wagami.

3.1 Trenowanie z użyciem zrównoważonego zbioru uczącego

Aplikacja umożliwia śledzenie procesu uczenia modelu korzystając z wcześniej przygotowanego zbioru uczącego lub za pomocą zbioru stworzonego dzięki interakcji aplikacji i użytkownika.

Uczenie z użyciem przygotowanego zbioru uczącego

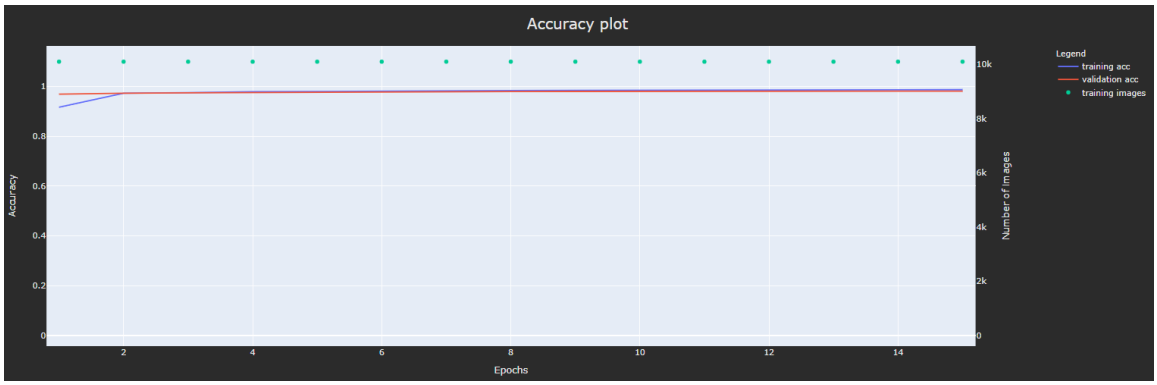
W tym scenariuszu użyto zbiór obrazów, który nosi nazwę "*Cats vs Dogs*". Zawiera on obrazy które możemy sklasyfikować do jednej z dwóch klas: kot lub pies, a jego podsumowanie zostało zawarte w tabeli 3.1. Dla każdego podzbioru liczba obrazów należąca do jednej z klas jest bliska liczbie obrazów z klasy drugiej, co oznacza że podzbiory te są zrównoważone. Proces uczenia trwał 15 epok oraz ustawiono szybkość uczenia na wartość 0.001, a wartość parametru *batch size* na 128.

Tabela 3.1: Liczba obrazów w zbiorze "*Cats vs Dogs*" zawierającym klasy kot i pies

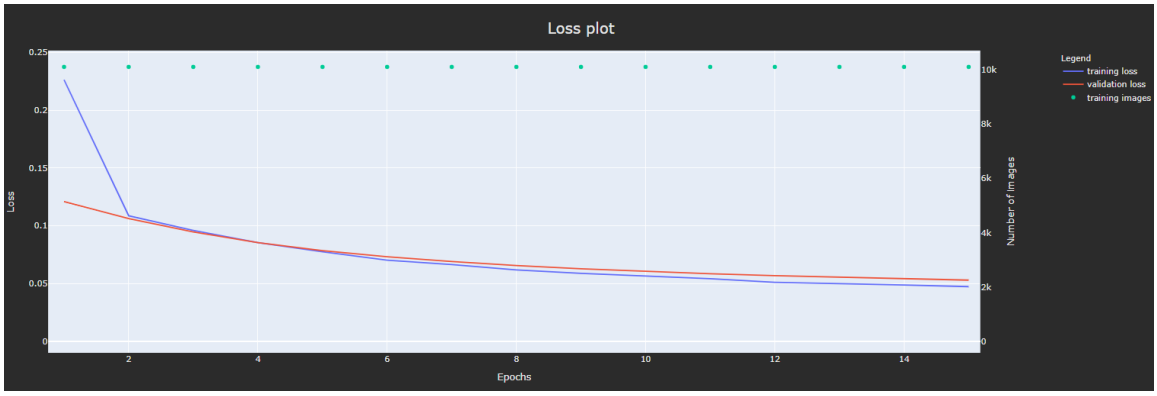
	liczba obrazów w klasie		
zbiór	kot	pies	suma
uczący	5016	5084	10100
walidacyjny	5159	5315	10474
testowy	663	668	1331

Na rysunku 3.1 przedstawiono wykresy wygenerowane przez aplikację w trakcie procesu uczenia. Można zauważyć, że w związku z bardzo dużą liczbą obrazów w zbiorze uczącym, model bardzo szybko osiągnął wysoki poziom skuteczności.

Klasyfikator `resnet34` już po pierwszej epoce osiągnął bardzo wysoką skuteczność na zbiorze walidacyjnym bliską 0.98, a przez kolejne epoki nie był w stanie znacząco poprawić tego wyniku. Zostało to spowodowane tym, że zbiór uczący posiadał wystarczająco dużą liczbę obrazów w stosunku do złożoności problemu. W tabeli 3.2 przedstawiono statystyki podsumowujące proces uczenia. Skuteczność osiągnięta na zbiorze walidacyjnym jak



(a) Wykres skuteczności modelu w funkcji liczby epok



(b) Wykres wartości funkcji kosztu w funkcji liczby epok

Rysunek 3.1: Wizualizacja procesu uczenia modelu na zbiorze zawierającym przygotowany podzbiór uczący

i uczącym jest zbliżona co oznacza że model nauczył się dobrze generalizować zadany problem, co potwierdzają statystyki osiągnięte na zbiorze testowym. Wartość funkcji kosztu

Tabela 3.2: Podsumowanie procesu trenowania stosując duży zbiór uczący

zbiór	maksymalna skuteczność	minimalna wartość funkcji kosztu
uczący	0.9870	0.0475
walidacyjny	0.9815	0.0531
testowy	0.9805	0.0542

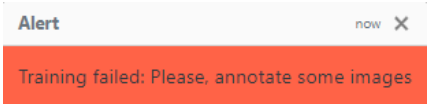
jest malejąca, a swoje najlepsze rezultaty model osiągnął w ostatniej epoce, co sugeruje że model mógłby osiągnąć jeszcze nieznacznie wyższą skuteczność, gdyby zwiększyć liczbę epok.

Trenowanie z użyciem niewielkiego zbioru uczącego

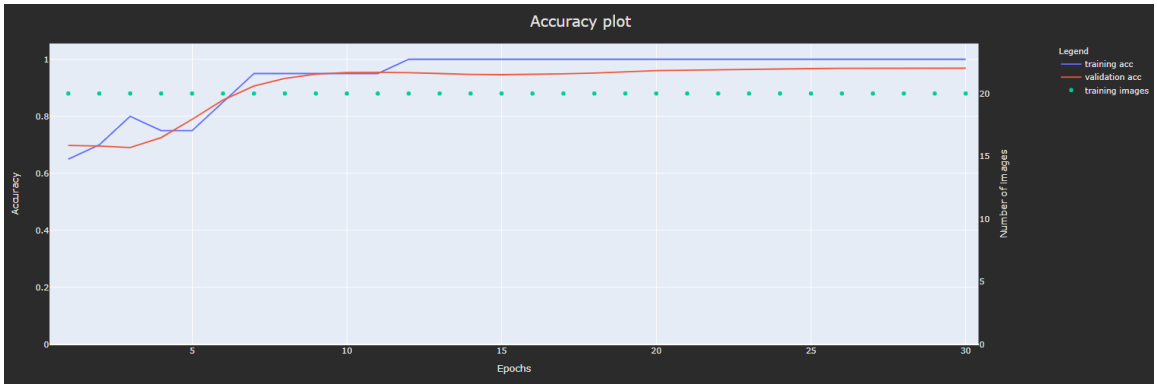
W tym scenariuszu również użyto zbioru "*Cats vs Dogs*", z tą różnicą że początkowo liczba obrazów w zbiorze uczącym będzie wynosiła zero. Zbiór walidacyjny i testowy będzie zawierał te same obrazy, które były użyte w poprzednim scenariuszu, a ich liczba zostanie również taka sama. Dodatkowo zostanie użyty zbiór nieetykietowany zawierający 3998 obrazów.

W momencie podjęcia próby rozpoczęcia trenowania modelu na pustym zbiorze uczącym serwer odrzuci takie żądanie generując powiadomienie przedstawione na rysunku

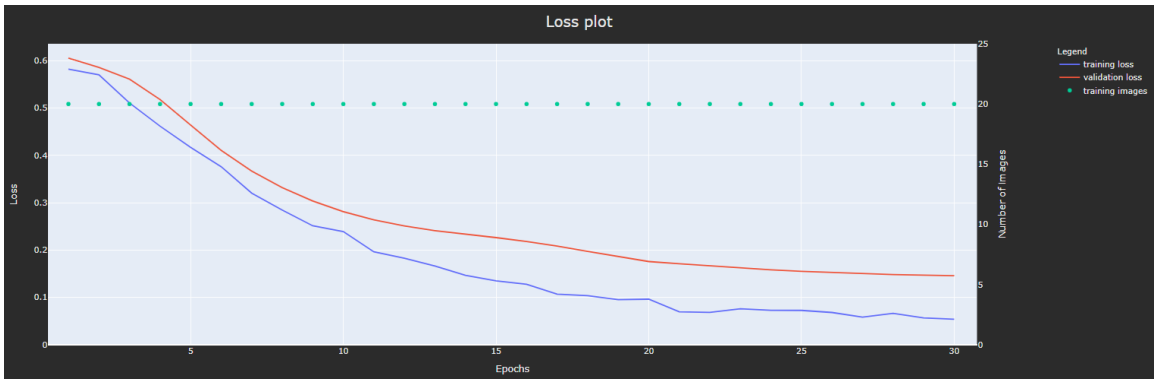
3.2.



Rysunek 3.2: Powiadomienie informujące o braku obrazów w zbiorze uczącym



(a) Wykres skuteczności modelu w funkcji liczby epok



(b) Wykres wartości funkcji kosztu w funkcji liczby epok

Rysunek 3.3: Wizualizacja procesu uczenia modelu na niewielkim zbiorze uczącym

W celu zaprezentowania procesu uczenia na niewielkim zbiorze, najpierw zostały przypisane etykiety 20 obrazom (10 dla klasy kot oraz 10 dla klasy pies). Proces trenowania został zaprezentowany na rysunku 3.3. Ze względu na dużo mniejszą liczbę obrazów, zwiększono liczbę epok do 30, a parametr *batch size* pozostał niezmienny. Tabela 3.3 przedstawia podsumowanie procesu uczenia. Porównując ten scenariusz do poprzedniego w którym użyto zdecydowanie większej liczby obrazów do trenowania modelu, możemy zauważyć:

- końcowa skuteczność jaką osiągnął model korzystając z zaledwie 20 obrazów jest tylko niecałe 2 punkty procentowe mniejsza od wyniku osiągniętego na zbiorze zawierającym ponad 10000 obrazów w zbiorze uczącym (tabela 3.2),
- w odróżnieniu od poprzedniego scenariusza w którym model niemal natychmiastowo osiągnął rezultat bliski końcowego, teraz model przez pierwszych 10 epok gwałtownie zwiększał swoją skuteczność, a wypłaszczenie wartości funkcji kosztu miało miejsce dopiero w okolicach 25 epoki,

- w obu scenariuszach modelowi udało się zminimalizować błąd na zbiorze uczącym do podobnej wartości, natomiast w drugim scenariuszu wartość funkcji kosztu na zbiorze walidacyjnym jest zauważalnie większa niż na zbiorze uczącym, co sugeruje że nastąpiło niewielkie przeuczenie modelu spowodowane zbyt małą liczbą obrazów w zbiorze uczącym,
- model bardzo szybko osiągnął 100% skuteczność na zbiorze uczącym, co jest zjawiskiem oczekiwanym dla tak małej liczby próbek, natomiast fakt uzyskania blisko 97% skuteczności na zbiorze walidacyjnym pokazuje, że testowany model bardzo dobrze potrafi wyciągnąć z obrazów cechy które są potrzebne do odróżnienia psa i kota.

Tabela 3.3: Podsumowanie procesu trenowania stosując niewielki zbiór uczący

zbiór	maksymalna skuteczność	minimalna wartość funkcji kosztu
uczący	1.0000	0.0540
walidacyjny	0.9686	0.1457
testowy	0.9609	0.1562

3.2 Douczanie wytrenowanego modelu

W tym scenariuszu użyto zbiór zawierający obrazy ludzi w różnym przedziale wiekowym (od 10 do 100 lat) oraz należących do różnych ras [32], a zadaniem klasyfikatora było określenie czy obraz przedstawia mężczyznę czy kobietę.

Uczenie z użyciem przygotowanego zbioru uczącego

Aplikacja zapisuje stan modelu po każdej epoce, co umożliwia wielokrotne douczanie modelu z użyciem tego samego lub zmienionego zbioru uczącego. W prezentowanym scenariuszu model był douczany co 15 lub 30 epok, a liczba obrazów w zbiorze uczącym była zwiększana każdorazowo dwukrotnie, począwszy od 10 obrazów. Podczas zwiększania liczby obrazów po równo brane były obrazy o największej niepewności oraz te wybrane losowo ze zbioru nieetykietowanego.

Analizując proces uczenia zamieszczony na rysunku 3.4 oraz jego podsumowanie w tabeli 3.4 można wysnuć następujące wnioski:

- model mając niewiele obrazów w zbiorze uczącym był w stanie bardzo szybko nauczyć się "na pamięć" tych obrazów osiągając 100% skuteczności na zbiorze uczącym, natomiast wynik na zbiorze walidacyjnym był zdecydowanie niższy,
- w momencie zbliżania się do 100% skuteczności na zbiorze uczącym, skuteczność na zbiorze walidayjnym przestawała rosnąć, a nawet zaczynała spadać (epoka 14, 15),
- po każdym rozszerzeniu zbioru uczącego skuteczność modelu gwałtownie spadała, ponieważ model musiał dostosować się do nowych obrazów, które wcześniej były błędnie klasyfikowane,
- powiększenie zbioru uczącego z 160 na 320 obrazów tylko w minimalnym stopniu zwiększyło skuteczność modelu, co świadczy o tym że model w zadanej architekturze i dla tego zbioru danych nie jest w stanie już znacząco zwiększyć swojej skuteczności,

(a) Wykres skuteczności modelu w funkcji liczby epok

(b) Wykres wartości funkcji kosztu w funkcji liczby epok

Rysunek 3.4: Wizualizacja procesu douczania modelu na coraz większym zbiorze uczącym

- krótkotrwały, gwałtowny wzrost wartości funkcji kosztu na zbiorze uczącym w 136 epoce był spowodowany próbą douczenia modelu na tym samym zbiorze obrazów (rozpoczęto nowy proces uczenia), problem ten często występuje przy próbie wznowienia uczenia, a może być związane z błędną inicjalizacją obiektu `optim.Adam`.

Tabela 3.4: Podsumowanie procesu douczania modelu

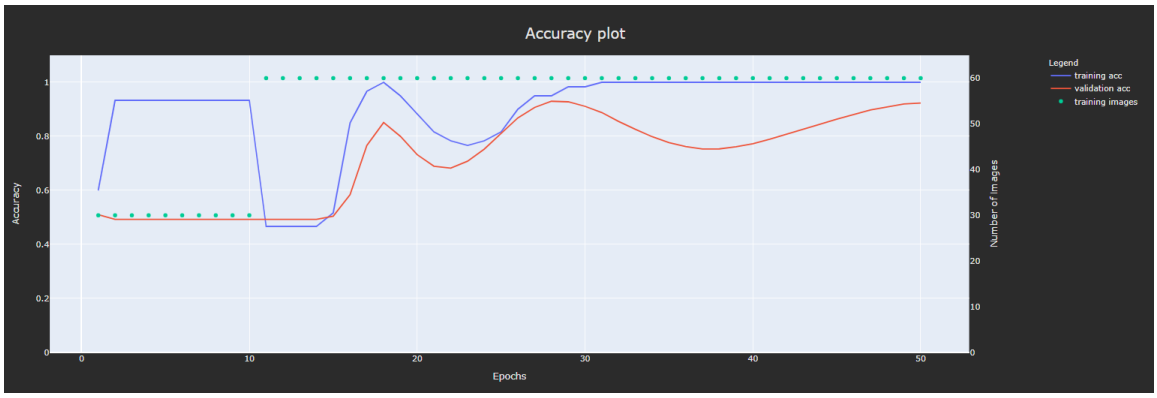
zbiór	maksymalna skuteczność	minimalna wartość funkcji kosztu
uczący	1.0000	0.1687
walidacyjny	0.8609	0.3273
testowy	0.8665	0.3291

Uczenie z użyciem niezbalansowanego zbioru

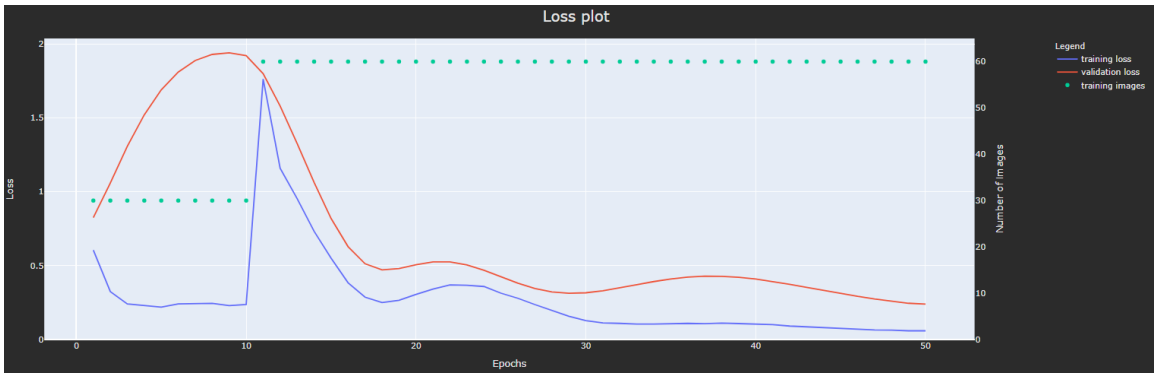
Przedstawienie tego scenariusza ma jedynie charakter eksperymentalny, ponieważ aby poprawnie wytrenować model na niezbalansowanym zbiorze stosuje się szereg technik [33], takich jak przepróbkowanie (ang. *resampling*), czy skorzystanie z metod wrażliwych na koszt (ang. *cost-sensitive*).

W pierwszej kolejności dodano etykiety dla 30 obrazów, ale ich rozłożenie było nierównomierne (28 obrazów zaklasyfikowanych jako koty a tylko pozostałe 2 jako psy). Następnie wykonano proces uczenia modelu przez 10 epok, po czym dodano 30 nowych obrazów zaklasyfikowanych jako psy, co pozwoliło na zbalansowanie zbioru uczącego.

Wykres 3.5 przedstawia wynik procesu trenowania modelu:



(a) Wykres skuteczności modelu w funkcji liczby epok



(b) Wykres wartości funkcji kosztu w funkcji liczby epok

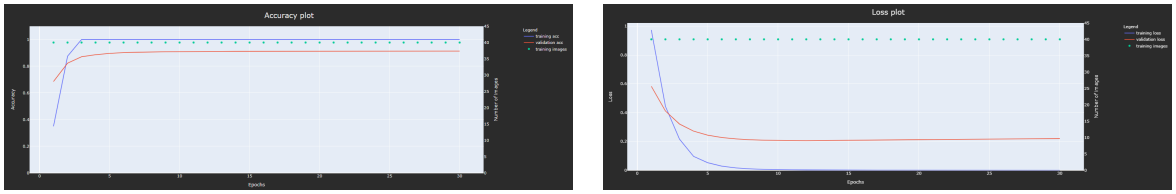
Rysunek 3.5: Wizualizacja procesu uczenia z użyciem niezbalansowanego zbioru uczącego oraz douczenie modelu modelu już na zrównoważonym zbiorze

- przez pierwszych 10 epok model nauczył się klasyfikować wszystkie obrazy do jednej, dominującej w zbiorze uczącym klasie. Osiągnął wtedy 93,33% skuteczności na zbiorze uczącym i około 50% na zbiorze walidacyjnym, co oznacza że model nauczył się klasyfikować wszystkie obrazy do jednej dominującej klasy w zbiorze uczącym,
- gdyby zbiór walidacyjny, podobnie jak zbiór uczący, był niezbalansowany to skuteczność jako funkcja celu byłaby bardzo myląca. Dlatego w takich przypadkach stosuje się inne metryki jakimi przykładowo są F1, bądź krzywa ROC (ang. *Receiver Operating Characteristic*),
- po dodaniu nowych obrazów do zbioru uczącego model bardzo szybko zaczął się uczyć, co dobrze obrazuje wykres funkcji kosztu, której wartość natychmiastowo zaczęła gwałtownie spadać, a skutek tego w postaci zwiększonej skuteczności jest widoczny po kilku epokach.

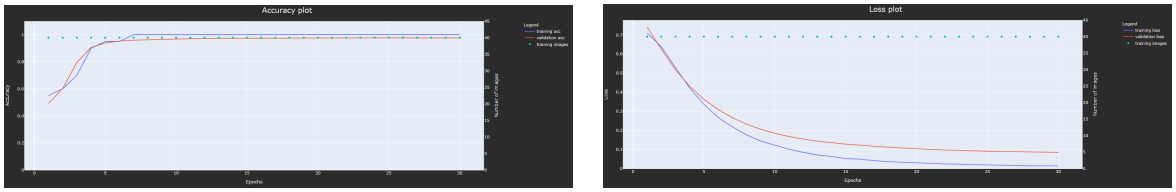
3.3 Porównanie przetrenowanych modeli CNN

W tym scenariuszu porównano cztery przetrenowane modele: `alexnet`, `mobilent_v2`, `resnet18` oraz `resnet34`. Wszystkim modelom zamieniono ostatnią warstwę na blok sekwencyjny składający się z warstwy `Dropount` oraz `Linear` o dwóch wyjściach. Użyty został przedstawiony w tabeli 3.1 zbiór "*Cats vs Dogs*", ale z 40 obrazami w zbiorze uczącym. Proces trenowania przebiegał jednakowo dla wszystkich czterech modeli:

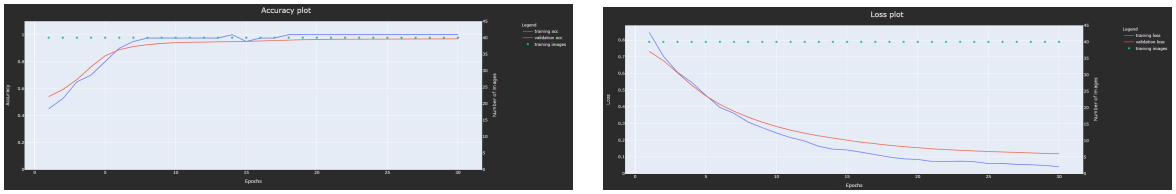
- trwał on 30 epok,
- parametr *batch size* wynosił 128,
- parametr *learning rate* wynosił 0.001,
- zbiór uczący zawsze zawierał te same obrazy.



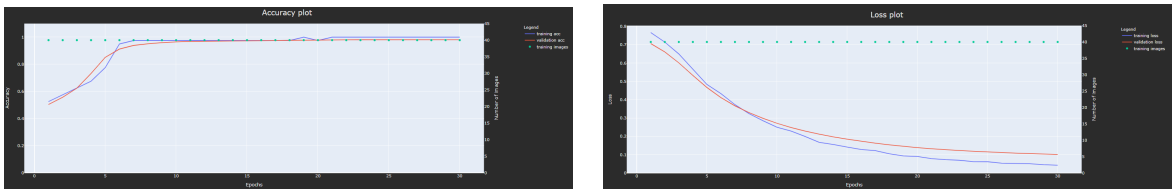
AlexNet



MobileNetv2



ResNet18



ResNet34

Rysunek 3.6: Porównanie procesu uczenia wybranych przetrenowanych modeli splotowych sieci neuronowych. Kolumna po lewej stronie przedstawia wykresy skuteczności modelu, natomiast kolumna po prawej wartość funkcji kosztu

Poddając analizie wykresy zamieszczone na rysunku 3.6 oraz zbiorcze statystyki modeli w tabeli 3.5 można wysnuć następujące wnioski:

- najwyższą skuteczność na zbiorze walidacyjnym osiągnął model ResNet34, natomiast zdecydowanie najniższą model AlexNet,
- architektura AlexNet najgorzej generalizuje zadany problem, a świadczy o tym nie tylko najniższa osiągnięta skuteczność, ale przede wszystkim różnica pomiędzy wartością błędu pomiędzy zbiorami walidacyjnym a uczącym – nastąpiło przeuczenie modelu,

Tabela 3.5: Porównanie modeli splotowych sieci neuronowych

	zbiór uczący		zbiór walidacyjny	
model	skuteczność/epoka	błąd/epoka	skuteczność/epoka	błąd/epoka
alexnet	1.0000/3	0.0002/30	0.9135/29	0.2058/12
mobilenet	1.0000/7	0.0136/30	0.9759/22	0.8490/30
resnet18	1.0000/14	0.0405/30	0.9689/30	0.1185/30
resnet34	1.0000/19	0.0425/30	0.9810/26	0.1013/30

- wszystkie modele osiągnęły 100% skuteczność na zbiorze uczącym,
- model który uzyskał najwyższą skuteczność na zbiorze walidacyjnym, najpóźniej uzyskał 100% skuteczność na zbiorze uczącym,
- wszystkie modele osiągnęły najniższy błąd na zbiorze uczącym w ostatniej epoce,
- lepszy wynik architektury ResNet o większej liczbie warstw splotowych jest zgodny z główną zaletą tego modelu opisaną przez twórców tego modelu [3],
- architektura MobileNet spisuje się lepiej od ResNet18, ale gorzej od ResNet34, a ponieważ wymaga ona najmniejszych zasobów obliczeniowych spośród tej trójki [16], jest to architektura godna uwagi gdy system ma ograniczone zasoby. Przykładem mogą być urządzenia mobilne.

3.4 Użycie wytrenowanego modelu

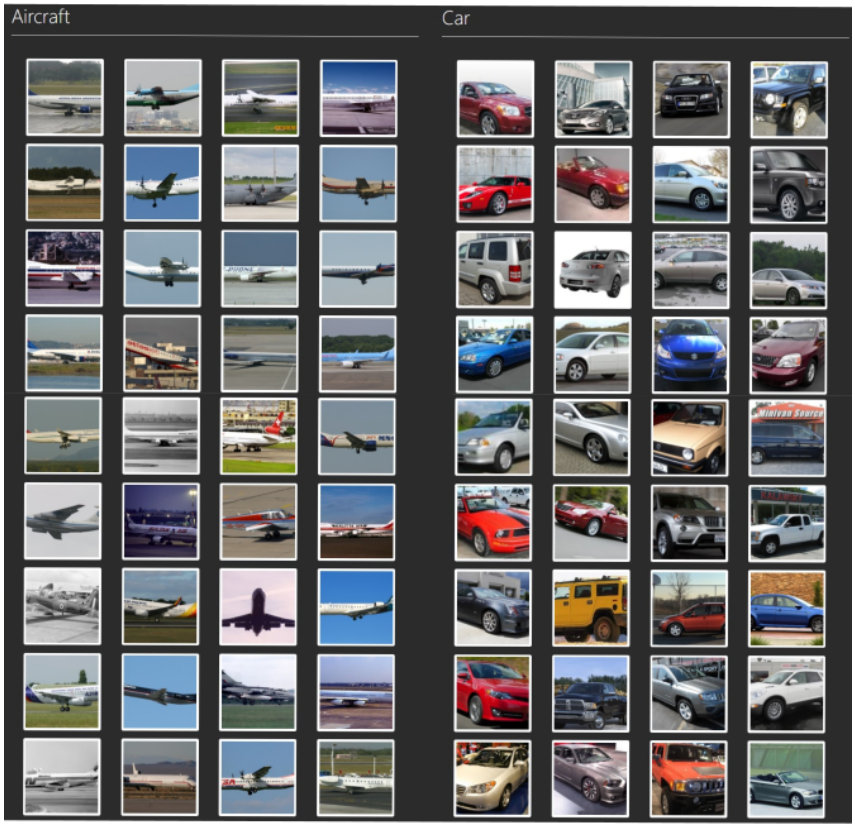
Aplikacja umożliwia zarówno trenowanie modelu jak i przypisywanie etykiet. To połączenie umożliwia użycie aplikacji do sprawnego tworzenia etykietowanego zbioru. W tym celu należy najpierw wytrenować model, a następnie taki model może być użyty do etykietowania obrazów. Aplikacja umożliwia zarówno iteratywne dodawanie etykiet, dzięki czemu użytkownik ma kontrolę nad przypisywaniem etykiet, jak i dodanie etykiet do wszystkich nieetykietowanych obrazów korzystając z ostatniej zapisanej predykcji modelu. Pozwala to na niemal natychmiastowe wygenerowanie etykietowanego zbioru danych, a dzięki znajomości estymowanej skuteczności modelu możemy kontrolować ilość szumu, który zostanie dodany do nowo stworzonego zbioru. Umyślne dodawanie szumu do zbioru danych może być korzystnym zjawiskiem, które odpowiednio użyte może poprawić skuteczność modelu [34].

W celu prezentacji tego scenariusza skorzystano ze zbioru zawierającego obrazy samochodów i samolotów, zawierającego 7979 obrazów nieetykietowanych oraz 4063 w zbiorze walidacyjnym. Kolejne kroki wyglądały następująco:

1. przypisanie 40 etykiet (21 obrazów samochodów, 19 obrazów samolotów),
2. rozpoczęcie procesu trenowania modelu, w którym model osiągnął 99.58% skuteczności na zbiorze walidacyjnym,
3. zażądanie nowej predykcji modelu,
4. dodanie etykiet do wszystkich obrazów ze zbioru nieetykietowanego, korzystając z predykcji modelu z bardzo wysoką skutecznością.

W wyniku uzyskano blisko 8000 nowych etykiet, które zostały przypisane w bardzo krótkim czasie. Co więcej, chcąc podpisać większą liczbę obrazów wystarczy dodać nowe obrazy do zbioru nieetykietowanego i ponownie wysłać żądanie `POST /predictions?all_annotations=true`.

Kontrolowane tworzenie zbioru etykietowanego



Rysunek 3.7: Wizualizacja klasyfikacji obrazów z użyciem wytrenowanego modelu

Jeżeli użytkownikowi zależy aby wszystkie obrazy były poprawnie sklasyfikowane to aplikacja zapewnia taką możliwość. Na rysunku 3.7 przedstawiono interfejs użytkownika zwrócony w odpowiedzi na żądanie `GET /predictions` dla wytrenowanego modelu. Można zauważyć że wszystkie obrazy zostały sklasyfikowane poprawnie, więc użytkownik może dodać te obrazy do zbioru etykietowanego i przejrzeć kolejne obrazy.

Problem predykcji nieznanej klasy

Korzystając z wytrenowanego binarnego klasyfikatora do klasyfikacji obrazów należących do różnych, ale jasno określonych klas (przykładowo kot i pies), pojawia się problem w momencie próby klasyfikacji obrazu który nie jest ani psem ani kotem. Taki przypadek przedstawiono na rysunku 3.8. W celu uniknięcia tego problemu należy wytrenować klasyfikator binarny do rozpoznawania jednej docelowej klasy, a jako drugą klasę przyjąć wszystkie obrazy które nie są klasą docelową. Drugą możliwością jest użycie klasyfikatora wieloklasowego, natomiast to wykracza poza obszar pracy.

Rysunek 3.8: Problem klasyfikatora binarnego wytrenowanego do klasyfikacji dwóch klas, zdjęcie drona zostało sklasyfikowane jako kot

Podsumowanie

W ramach projektu zaimplementowano aplikację umożliwiającą śledzenie i wizualizację procesu uczenia klasyfikatora binarnego. Ponadto zaimplementowany system umożliwia interaktywne tworzenie zbioru uczącego oraz douczanie wytrenowanego modelu. Działanie aplikacji oparte jest o plik konfiguracyjny w formacie YAML, co zapewnia możliwość dostosowania aplikacji do potrzeb użytkownika. Przede wszystkim użytkownik może śledzić proces uczenia na dowolnym zbiorze obrazów oraz testować jeden z kilku wspieranych przez aplikacje modeli splotowych sieci neuronowych. Przejrzysty interfejs graficzny zapewnia intuicyjne korzystanie z aplikacji, a dynamicznie generowane wykresy skuteczności i wartości funkcji kosztu wpływają pozytywnie na wygodę korzystania z systemu. Aplikacja zapewnia możliwość testowania wytrenowanego modelu, a dzięki jej wizualizacji możemy określić z jakimi obrazami model ma największe trudności. Dodatkowo aplikacja zapewnia szereg funkcji ułatwiających jej używanie z poziomu użytkownika, a są to przede wszystkim:

- możliwość zmiany zbioru danych bez konieczności restartowania serwera,
- możliwość inicjalizacji modelu losowymi wagami,
- możliwość usunięcia dotychczasowo przyznanych etykiet,
- wstępne przetwarzanie obrazów.

Największym nierozwiązanym problemem projektu jest krótkotrwały spadek skuteczności na zbiorze uczącym w momencie douczania modelu. Oprócz tego system nie posada testów jednostkowych, czy integracyjnych, które są elementami wartymi zaimplementowania.

Rozszerzenie projektu o wsparcie dla większej liczby modeli CNN nie powinno stanowić problemu, a w przypadku dalszego rozwoju aplikacji należałoby się zastanowić nad wydzieleniem mikroserwisów w postaci serwera HTTP, przyjmującego żądania oraz mikroserwisu odpowiedzialnego za zarządzanie modelem sieci neuronowych. Do komunikacji można skorzystać z jednej z dostępnych kolejek komunikatów takich jak ZeroMQ, RabbitMQ, czy Redis. Taka architektura zapewniłaby bardzo dobrą skalowalność systemu oraz ułatwiłaby testowanie aplikacji.

Bibliografia

[1] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang i Joel Emer. „Efficient Processing of Deep Neural Networks: A Tutorial and Survey". W: *Proceedings of the IEEE* 105 (mar. 2017).

[2] Asifullah Khan, Anabia Sohail, Umme Zahoora i Aqsa Saeed Qureshi. „A survey of the recent architectures of deep convolutional neural networks". W: *Artificial Intelligence Review* 53.8 (kw. 2020), s. 5455–5516. ISSN: 1573-7462.

[3] K. He, X. Zhang, S. Ren i J. Sun. „Deep Residual Learning for Image Recognition". W: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, s. 770–778.

[4] A. Balalaie, A. Heydarnoori i P. Jamshidi. „Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". W: *IEEE Software* 33.3 (2016), s. 42–52.

[5] P. Adamczyk, P.H. Smith, R.E. Johnson i M. Hafiz. „REST and Web Services: In Theory and In Practice". W: *REST: From Research to Practice*. Springer, New York, 2011.

[6] SmartBear Software. *OpenAPI Specification*. URL: `https://swagger.io/`.

[7] Anubha Sharma, Manoj Kumar i Sonali Agarwal. „A Complete Survey on Software Architectural Styles and Patterns". W: *Procedia Computer Science* 70 (grud. 2015), s. 16–28.

[8] Mark Otto i Thornton Jacob. *biblioteka Bootstrap*. URL: `https://getbootstrap.com/`.

[9] M. R. Mufid, A. Basofi, M. U. H. Al Rasyid, I. F. Rochimansyah i A. rokhim. „Design an MVC Model using Python for Flask Framework Development". W: *2019 International Electronics Symposium (IES)*. 2019, s. 214–219.

[10] Oren Ben-Kiki, Clark Evans i Ingy döt Net. *YAML*. URL: `https://yaml.org/`.

[11] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. STD 90. RFC Editor, grud. 2017.

[12] SmartBear Software. *Swagger UI*. URL: `https://swagger.io/tools/swagger-ui/`.

[13] Shervine Amidi. *stanford-cs-229-machine-learning*. URL: `https://github.com/afshinea/stanford-cs-229-machine-learning/blob/master/en/cheatsheet-deep-learning.pdf`.

[14] Akinori Hidaka i Takio Kurita. „Consecutive Dimensionality Reduction by Canonical Correlation Analysis for Visualization of Convolutional Neural Networks". W: t. 2017. Grud. 2017, s. 160–167.

[15] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do i Kaori Togashi. „Convolutional neural networks: an overview and application in radiology". W: *Insights into Imaging* 9.4 (sierp. 2018), s. 611–629. ISSN: 1869-4101.

[16] S. Bianco, R. Cadene, L. Celona i P. Napoletano. „Benchmark Analysis of Representative Deep Neural Network Architectures". W: *IEEE Access* 6 (2018), s. 64270–64277.

[17] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang i Chunfang Liu. „A Survey on Deep Transfer Learning". W: *CoRR* abs/1808.01974 (2018).

[18] Andrew Brock, T. Lim, James Ritchie i Nick Weston. „FreezeOut: Accelerate Training by Progressively Freezing Layers". W: (czer. 2017).

[19] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík i Ladislav Hluchý. „Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey". W: *Artificial Intelligence Review* 52.1 (czer. 2019), s. 77–124. ISSN: 1573-7462.

[20] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai i Soumith Chintala. „PyTorch: An Imperative Style, High-Performance Deep Learning Library". W: *Advances in Neural Information Processing Systems 32*. Red. H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox i R. Garnett. Curran Associates, Inc., 2019, s. 8024–8035.

[21] K. Dinghofer i F. Hartung. „Analysis of Criteria for the Selection of Machine Learning Frameworks". W: *2020 International Conference on Computing, Networking and Communications (ICNC)*. 2020, s. 373–377.

[22] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach i Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Czer. 1999.

[23] Alex Clark. *Pillow - wspierane formaty obrazów*. URL: `https://pillow.readthedocs.io/en/latest/handbook/image-file-formats.html`.

[24] Aurlien Gron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O'Reilly Media, Inc., 2017.

[25] J. Shijie, W. Ping, J. Peiyi i H. Siping. „Research on data augmentation for image classification based on convolution neural networks". W: *2017 Chinese Automation Congress (CAC)*. 2017, s. 4165–4170.

[26] Alex Krizhevsky, Ilya Sutskever i Geoffrey E Hinton. „ImageNet Classification with Deep Convolutional Neural Networks". W: *Advances in Neural Information Processing Systems*. Red. F. Pereira, C. J. C. Burges, L. Bottou i K. Q. Weinberger. T. 25. Curran Associates, Inc., 2012, s. 1097–1105.

[27] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto i Hartwig Adam. „MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". W: *arXiv e-prints* (kw. 2017).

[28] Diederik Kingma i Jimmy Ba. „Adam: A Method for Stochastic Optimization". W: *International Conference on Learning Representations* (grud. 2014).

[29] L. Sun i X. Wang. „A survey on active learning strategy". W: *2010 International Conference on Machine Learning and Cybernetics*. T. 1. 2010, s. 161–166.

[30] The jQuery Team. *biblioteka jQuery*. URL: `https://jquery.com/`.

[31] Armin Ronacher. *silnik szablonów Jinja2*. URL: `https://palletsprojects.com/p/jinja/`.

[32] Zhifei Zhang, Yang Song i Hairong Qi. „Age Progression/Regression by Conditional Adversarial Autoencoder". W: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2017.

[33] S. Wang, W. Liu, J. Wu, L. Cao, Q. Meng i P. J. Kennedy. „Training deep neural networks on imbalanced data sets". W: *2016 International Joint Conference on Neural Networks (IJCNN)*. 2016, s. 4368–4374.

[34] Qizhe Xie, Minh-Thang Luong, Eduard Hovy i Quoc V. Le. „Self-training with Noisy Student improves ImageNet classification". W: *arXiv e-prints*, arXiv:1911.04252 (list. 2019), arXiv:1911.04252.