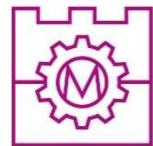




**POLITECHNIKA KRAKOWSKA im. T. Kościuszki**  
Wydział Mechaniczny  
**Katedra Informatyki Stosowanej**



Kierunek studiów: Informatyka stosowana

STUDIA STACJONARNE

# **PRACA DYPLOMOWA**

INŻYNIERSKA

**Adam Gruszczyński**

Metody wdrożenia systemów uczenia maszynowego na przykładzie  
budowy skalowej usługi rozpoznawania obrazów

Methods of implementing machine learning systems on the example of  
building a scalable image recognition service

Promotor:  
Mgr inż. **Marek Lewiński**

Kraków, rok akad. 2024/2025



<sup>\*</sup>) – niepotrzebne skreślić

## OŚWIADCZENIE O SAMODZIELNYM WYKONANIU PRACY DYPLOMOWEJ

Oświadczam, że przedkładana przeze mnie praca dyplomowa inżynierska została napisana przeze mnie samodzielnie. Jednocześnie oświadczam, że ww. praca:

- 1) nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz.U. z 2021 r. poz. 1062) oraz dóbr osobistych chronionych prawem cywilnym, a także nie zawiera danych i informacji, które uzyskałem/amt\* w sposób niedozwolony,
- 2) nie była wcześniej podstawą żadnej innej procedury związanej z nadawaniem tytułów zawodowych, stopni lub tytułów naukowych.

Jednocześnie wyrażam zgodę na:

- 1) poddanie mojej pracy kontroli za pomocą systemu Antyplagiat oraz na umieszczenie tekstu pracy w bazie danych uczelni, w celu ochrony go przed nieuprawnionym wykorzystaniem. Oświadczam, że zostałem/amt\* poinformowany/a\* i wyrażam zgodę, by system Antyplagiat porównywał tekst mojej pracy z tekstem innych prac znajdujących się w bazie danych uczelni, z tekstami dostępnymi w zasobach światowego Internetu oraz z bazą porównawczą systemu Antyplagiat,
- 2) to, aby moja praca pozostała w bazie danych uczelni przez okres wynikający z przepisów prawa. Oświadczam, że zostałem poinformowany i wyrażam zgodę, że tekst mojej pracy stanie się elementem porównawczej bazy danych uczelni, która będzie wykorzystywana, w tym także udostępniana innym podmiotom, na zasadach określonych przez uczelnię, w celu dokonywania kontroli antyplagiatowej prac dyplomowych/doktorskich, a także innych tekstów, które powstaną w przyszłości.

*Adam ... inżynier ...*  
podpis

- 1) Wyrażam zgodę na udostępnianie mojej pracy dyplomowej w Akademickim Systemie Archiwizacji Prac na PK do celów naukowo-badawczych z poszanowaniem przepisów ustawy o prawie autorskim i prawach pokrewnych (Dz.U. z 2021 r. poz. 1062)..

TAK/NIE\*

*Adam ... inżynier ...*  
podpis

Jednocześnie przyjmuję do wiadomości, że w przypadku stwierdzenia popełnienia przez mnie czynu polegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzej pracy, lub ustalenia naukowego, Rektor PK stwierdzi nieważność postępowania w sprawie nadania mi tytułu zawodowego (art. 77 ust. 5 ustawy z dnia 18 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce, (Dz.U. z 2021 r., poz. 478, z późn. zm.)).

*Adam ... inżynier ...*  
podpis



## Spis treści

<b>1. WSTĘP.....</b>	<b>7</b>
<b>2. CEL I ZAKRES PRACY.....</b>	<b>8</b>
<b>3. WPROWADZENIE TEORETYCZNE.....</b>	<b>9</b>
3.1 Uczenie maszynowe i sieci neuronowe.....	9
3.1.1 Definicja i zastosowanie uczenia maszynowego .....	9
3.1.2 Sieci neuronowe .....	9
3.1.3 Konwolucyjne sieci neuronowe .....	12
3.2 Metody wdrażania systemów uczenia maszynowego .....	14
3.2.1 Cechy dobrego systemu uczenia maszynowego .....	14
3.2.2 Sposoby implementacji gotowego modelu .....	17
3.3 Serwis orkiestracyjny .....	24
3.4 Konteneryzacja .....	25
3.5 Skalowanie .....	27
3.6 Technologie i platformy wykorzystane w pracy .....	27
3.6.1 Docker .....	27
3.6.2 Kubernetes.....	29
3.6.3 TensorFlow serving.....	33
3.6.4 TorchServe .....	35
<b>4. CZEŚĆ PROJEKTOWA.....</b>	<b>37</b>
4.1 Definicja celu oraz wymagań .....	37
4.2 Przygotowanie sieci konwolucyjnych do rozpoznawania obrazów .....	38
4.2.1 Przygotowanie fotografii do treningu .....	38
4.2.2 Transformacja danych.....	40
4.2.3 Architektura sieci konwolucyjnej .....	43
4.2.4 Trening i walidacja sieci konwolucyjnych.....	45
4.3 Eksport modeli.....	48
4.3.1 Eksport w TensorFlow .....	48
4.3.2 Eksport PyTorch.....	50
4.3.3 Utworzenie obrazów Docker.....	54
4.4 Utworzenie środowiska klastrowego za pomocą Minikube.....	55
4.5 Wdrażanie skalowalnych systemów uczenia maszynowego.....	57
4.5.1 Skalowanie pionowe .....	58
4.5.2 Skalowanie poziome .....	66
4.6 Podsumowanie.....	72
<b>5. WNIOSKI.....</b>	<b>74</b>
<b>LITERATURA.....</b>	<b>75</b>
<b>SUMMARY.....</b>	<b>77</b>



## **1. Wstęp**

Sztuczna inteligencja w ostatnim czasie zyskała na coraz większej popularności. Nowe możliwości związane z rozwojem technologicznym oraz pogłębianiem wiedzy na temat sieci neuronowych spowodowały w bardzo krótkim czasie wielką przemianę w świecie technologicznym, gdzie obecnie każdy kto posiada komputer, bądź smartfon, ma kontakt ze sztuczną inteligencją. Korzystanie z chat botów oraz innych generatywnych sztucznych inteligencji stało się dla wielu osób zwyczajną codziennością.

Sukces sztucznej inteligencji na rynku komputerowym nie byłby możliwy, gdyby nie stworzono zaawansowanych systemów uczenia maszynowego. Wielkie modele AI oferowane przez firmy są wielkimi systemami, które wymagają odpowiednich zasobów do ich wytrenowania jak i do zwykłego użytkowania. Z tego powodu napotyka się następujący problem. W jaki sposób udostępnić potencjalnym użytkownikom modele AI, żeby działały szybko, niezawodnie przy jak najmniejszych wymaganiach technicznych?

Praca ta jest właśnie poświęcona w celu odpowiedzi na to pytanie. Tekst składa się z części teoretycznej i praktycznej. Teoretyczna ma za zadanie wprowadzić w świat uczenia maszynowego oraz pokazać różne metody wdrażania jak i systemy wykorzystywane do tworzenia systemów uczenia maszynowego. Część praktyczna ma za zadanie przedstawić przykład takiego systemu, pokazać cały jego proces budowy, od tworzenia i trenowania sieci neuronowej, aż po ostateczne konfiguracje systemu oraz jego monitoring. Dodatkowo z całej pracy będzie można zyskać wiedze na temat systemów orkiestracyjnych oraz konteneryzacji, które są popularnie stosowane nie tylko w systemach uczenia maszynowego.

## **2. Cel i zakres pracy**

Celem pracy jest przedstawienie od strony teoretycznej metod wdrażania systemów uczenia maszynowego. Duże znaczenie w pracy ma też pokazanie definicji skalowania oraz jego rodzajów. By w pełni nakreślić ogólny obraz uczenia maszynowego oraz jego systemów, opracowanie teoretyczne składa się z następujących części:

- Wprowadzenie do świata uczenia maszynowego i konwolucyjnych sieci neuronowych;
- Przedstawienie cech dobrego systemu uczenia maszynowego;
- Omówienie metod wdrażania systemów;
- Rozwinięcie definicji skalowania;
- Omówienie idei systemów orkiestracyjnych oraz konteneryzacji.

Dodatkowo w ramach wprowadzenia do części praktycznej, teoria zawiera następujące opisy użytych narzędzi:

- Omówienie programu Docker;
- Przedstawienie architektury systemu Kubernetes;
- Przedstawienie architektury TensorFlow serving;
- Przedstawienie architektury TorchServe.

Część praktyczna ma za zadanie pokazać w małym zakresie jak powstaje system uczenia maszynowego z możliwością jego skalowania oraz zwrócenie uwagi na duże znaczenie metryk systemu. Przykładowy system ma za zadanie rozpoznawać emocję na podstawie fotografii twarzy osób.

### **3. Wprowadzenie teoretyczne**

#### **3.1 Uczenie maszynowe i sieci neuronowe**

##### **3.1.1 Definicja i zastosowanie uczenia maszynowego**

Uczenie maszynowe jest rodzajem sztucznej inteligencji, gdzie tworzony system jest w stanie uczyć się z danych. System rozwija się wraz z wzrostem doświadczenia zwiększąc swoją trafność predykcji [1][2].

Standardowe aplikacje są zaprogramowane zazwyczaj za pomocą sztywno ustalonych instrukcji. Niestety istnieją pewne przypadki, gdzie nie ma możliwości tą metodą stworzyć program, który poradzi sobie w każdym scenariuszu, gdyż ich liczba może być bardzo duża jak i sam programista nie jest w stanie ich przewidzieć. Dobrym tego przykładem jest filtr spamu, którego zadanie polega na przenoszeniu wiadomości e-mail do folderu spam [1]. Wiadomości e-mail mogą mieć różną formę i tematykę, które z czasem ulegają zmianom. Aplikacje programowane w sposób odręczny mają następujące wady:

- Są ograniczone do wykonywania specyficznego zadania w konkretnej dziedzinie. Nawet delikatna zmiana zadania jest w stanie wymusić przeprogramowanie całej logiki programu [3];
- Scenariusze programowane przez programistę wymagają głębszego zrozumienia problemu, na poziomie eksperckim [3].

By poradzić sobie z powyższymi wadami, tworzy się modele sztucznej inteligencji, które uczą się samodzielnie na podstawie przekazanych danych.

Obecnie uczenie maszynowe wykorzystuje się w wielu miejscach i dziedzinach, np.:

- Medycyna (wykrywanie niepoprawności w próbkach krwi, organach oraz kościach) [4];
- W codziennej pracy (pisanie wiadomości elektronicznych, pomoc przy programowaniu, analiza biznesowa)[5];
- E-commerce (tworzenie rekomendacji dla klientów)[6].

##### **3.1.2 Sieci neuronowe**

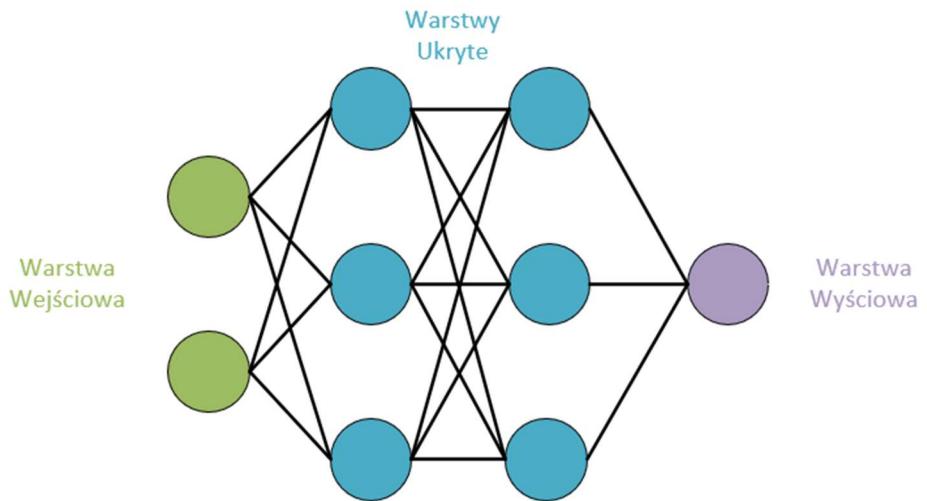
Uczenie maszynowe posiada swoje poddziedziny, które zajmują specyficznymi typami modeli, które można stworzyć. Jednym z bardziej skomplikowanych, ale zarazem oferującym największe możliwości spośród dziedzin uczenia maszynowego to uczenie głębokie, które zajmuje się budowaniem sieci neuronowych.

Sieci neuronowe to złożona architektura pomniejszych modeli liniowych zwanych perceptronami, które można też nazwać neuronami. Samo połączenie tych neuronów nie daje dużej zdolności do generalizacji, dlatego używa się również funkcji aktywacji, które wspomagają proces uczenia sieci oraz likwidując liniowość estymatora, umożliwiając modelowi zdolność do wyuczenia się dowolnej funkcji.

Sieci neuronowe składają się przeważnie z wielu warstw, gdzie można wyróżnić [7]:

- Warstwę wejściową – do niej trafiają wszystkie dane, na której podstawie jest przeprowadzona predykcja. Ilość neuronów zależy tutaj od liczby wartości reprezentującej pojedynczą próbki, zwanych w dziedzinie uczenia maszynowego jako cechy;
- Warstwa wyjściowa – w tej warstwie jest zwracany wynik predykcji. Ilość neuronów jest zależna od tego, jaką jest pożądana liczba wyników. W przypadku regresji funkcji wymagany jest jeden neuron. W problemach klasyfikacyjnych, gdzie do sklasyfikowania jest  $n$  ilość klas, można użyć  $n$  neuronów, jeśli zależy twórcy modelu na tym, żeby model pokazał na ile jest pewien przynależności do danych klas dla danej próbki.
- Warstwy ukryte – te warstwy są głównym powodem skuteczności sieci neuronowych. Sieć w tej części zaczyna dogłębnie analizować przekazane cechy. Jest całkowicie odseparowana od użytkownika (użytkownik ma styczność tylko z warstwą wejściową i wyjściową, nie widzi bezpośrednio tego co się dzieje w ukrytych warstwach), a jego sposób transformacji danych i ich interpretacja jest bardzo ciężka dla człowieka, zwłaszcza przy dużych sieciach neuronowych, w których może znaleźć się miliard neuronów.

Przykładową architekturę sieci neuronowej można zobaczyć na rysunku 3.1.



Rysunek 3.1. Architektura sieci neuronowej

Estymatorem sieci neuronowej jest funkcją, która jest kombinacją kombinacji innych funkcji liniowych. Każda z tych pojedynczych funkcji ma parametry oraz wyraz wolny. Parametry w modelach nazywa się wagami, a wyraz wolny w języku angielskim określa się jako bias, co można przetłumaczyć jako współczynnik obciążenia. Trening sieci neuronowej polega na znalezieniu takich wartości wag i wyrazów wolnych, które będą estymować daną funkcję.

Aby znaleźć te wartości, należy dobrać początkowe wartości, a następnie aktualizować je o gradient funkcji straty, która wylicza błąd, jaki dokonał model na podstawie wartości podanej przez model i wartości rzeczywistej, zgodnie ze wzorem podanym poniżej.

$$W_{ij}^{II} = W_{ij}^I - \eta \frac{\partial L(y_{pred}, y_{test})}{\partial W_{ij}^I}$$

$$b_i^{II} = b_i^I - \eta \frac{\partial L(y_{pred}, y_{test})}{\partial b_i^I}$$

Gdzie:

$W_{ij}^I$  – macierz początkowych wag;

$b_i^I$  – wektor początkowych współczynników obciążenia;

$\eta$  – współczynnik uczenia;

$L(y_{pred}, y_{test})$  – funkcja straty, gdzie  $y_{pred}$  to wynik zwrócony przez model, a  $y_{test}$  to wartość rzeczywista.

Wyliczenie nowych wag nie jest skomplikowaną operacją, ale jest tym bardziej złożona im szersza (szerokość – to ilość neuronów w danej warstwie) i głębsza (głębokość – ilość warstw w sieci) jest sieć neuronowa. Jest to konsekwencja procesu aktualizacji wag, zwany wsteczną propagacją. Trening w dużym uproszczeniu wygląda następująco:

- Przejście danych przez model – model próbuje dokonać predykcji;
- Obliczenie błędu – podstawienie predykcji i faktycznego wyniku do funkcji straty;
- Dokonanie wstecznej propagacji modelu.

Samych wag przy małych modelach może być kilka milionów (około kilkunastu MB). Samo przeliczenie wyniku przez sam model może zająć stosunkową małą ilość czasu, jeśli dysponuję się odpowiednim sprzętem komputerowym, jednakże budowane są takie sieci, które mogą mieć około nawet kilkaset miliardów parametrów, np.: GBT-3, Turing-NLG lub Transformer (GPipe) [8]. Przy takich modelach bardzo ważne jest zorganizować odpowiednio system, by skomplikowana sieć neuronowa nie zwracała wyników użytkownikowi ze zbyt dużym opóźnieniem.

### 3.1.3 Konwolucyjne sieci neuronowe

Sieć konwolucyjna to jedna z rodzajów sieci neuronowych, która jest popularnie używana do analizy obrazów. Sieć odpowiednio uczy się wykrywać charakterystyczne elementy z obrazów, np. podczas analizy zdjęcia pojazdu uczy się wykrywać takie elementy jak lusterka, koła szyby, maskę, zderzak itp.

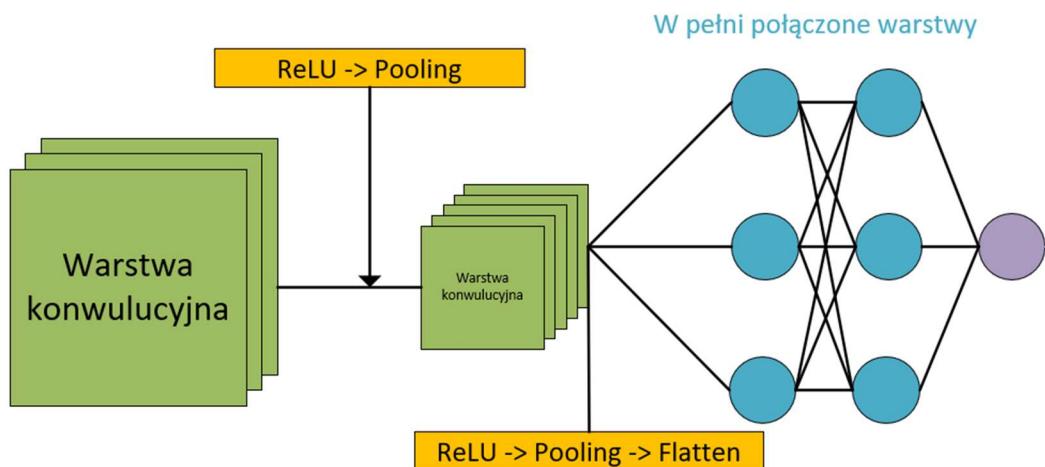
Typowa architektura sieci składa się z następujących elementów i operacji:

- Warstw konwolucyjnych – w tych warstwach sieć uczy się rozpoznawać konkretne elementy obrazu. Trenowane wartości tych warstw to zmienne filtra, które są nakładane na obraz i próbują wydzielić pożądane charakterystyczne elementy. Dane wejściowe są przekazywane w formie tensora dwuwymiarowego [9];
- Funkcja aktywacji ReLU – funkcja ta zeruje wszystkie wartości ujemne zwracane z warstw sieci. Jest to bardzo przydatna funkcja, gdyż tego typu sieć ma duży problem z tak zwanymi zanikającymi gradientami. Jest to niepożądany efekt wstecznej propagacji występującej często w głębokich sieciach. Polega na tym, że gradient obliczany podczas treningu, przy pierwszych warstwach schodzi do wartości bliskich zeru,

co wyhamowuję proces uczenia się modelu. Funkcja ReLU skutecznie minimalizuje to negatywne zjawisko;

- Pooling – jest to operacja, która ma celu wykonanie kompresji analizowanego obrazu, zmniejszając ilość analizowanych cech obrazu oraz dodatkowo wyróżnia z tła obrazu dominujące elementy, co może wspomóc uczenie się sieci [9];
- W pełni połączone warstwy – są to warstwy, gdzie każdy pojedynczy neuron jest połączony z każdym innym neuronem w sąsiedniej warstwie. W tej części nie trenuje się już wartości filtrów, a wagi, które są przyporządkowane każdej wartości wychodzącej z pojedynczego neuronu. Warstwa na podstawie wychwyconych szczegółów przez część konwolucyjną dokonuje analizy i próby klasyfikacji [9];
- Flatten – operacja, która spłaszcza macierz do wektora, konieczna do przekazania danych wyjściowych z części konwolucyjnej do warstw pełni połączonych, gdyż w tym miejscu dane muszą być przekazane w formie tensora jednowymiarowego.
- Funkcja aktywacji przeznaczona do klasyfikacji – potrzebna do uzyskania interpretowalnego wyniku.

Przykładową architekturę sieci konwolucyjnej można zobaczyć na rysunku 3.2.



Rysunek 3.2. Architektura sieci konwolucyjnej

Każda kolejna warstwa konwolucyjna pogłębia analizę wyszczególnionych elementów. Wyjaśniając na przykładzie analizy obrazu pojazdu. Pierwsza warstwa

wychwytyuje z fotografii cały pojazd. Kolejna warstwa na podstawie znalezionej samochodu, znajduje koła. Dalsza warstwa analizuje to koło i wyszczególnia logo z tego koła. W ten sposób sieć wychwytyuje cechy charakterystyczne obrazu i jest w stanie dokonać na tej podstawie klasyfikacji [9]. Trzeba ostrożnie podchodzić głębokości takiej sieci, ponieważ początkową pierwsze warstwy nauczą się rozpoznawać faktyczne elementy. Kolejne natomiast zaczynają wykrywać takie obiekty jak rysy na lakierze, kurz, błoto albo całkowicie abstrakcyjny obiekt, który nie powinien być brany pod uwagę. Przez zbędne informacje model będzie doszukiwać się zależności, które nie istnieją i stanie się podatny na przetrenowywanie.

### 3.2 Metody wdrażania systemów uczenia maszynowego

#### 3.2.1 Cechy dobrego systemu uczenia maszynowego

Systemy uczenia maszynowego przed wydaniem, by zapewnić jak największą jakość swoim konsumentom, muszą posiadać cztery cechy [10]:

- Niezawodny – systemy uczenia maszynowego muszą prawidłowo spełniać swoje funkcje. Można wyróżnić dwie podstawowe kategorie, w których może nieprawidłowo działać:

- Pierwsza kategoria to problemy związane bezpośrednio z modelem zaimplementowanym w systemie. Podczas fazy produkcyjnej jak i fazie poprodukcyjnej, należy kontrolować, jaka jest jakość wyników zwracana przez program. Do tego zadania używa się różnego rodzaju metryk, które walidują dane wyjściowe. Można wymienić przykładowe z nich:

- Dokładność =  $\frac{PP+PF}{N}$ ;
- Współczynnik błędów =  $\frac{FP+FF}{N}$ ;
- Precyzja =  $\frac{PP}{PP+FP}$ ;
- Czułość =  $\frac{PP}{PP+FN}$ ;
- Specyficzność =  $\frac{PF}{PF+FP}$ .

Gdzie: PP – prawdziwa prawda, PF – prawdziwy fałsz, FP – fałszywa prawda, FF – fałszywy fałsz, N – suma wszystkich danych wyjściowych [11]. Metryki te służą do wykrycia pogarszającej się skuteczności estymatora. Powody tego zjawiska są powiązane między innymi z: błędym założeniem dystrybucji danych realnego świata oraz ciągle zmieniającymi się czynnikami

determinujące estymowane zjawisko. Podczas treningu modelu gromadzi się dane, które są próbami reprezentującymi całą populację. Jeśli próbki nie zostaną w prawidłowy sposób pobrane, mogą zakłamać cały obraz estymowanej funkcji, a co za tym idzie, wytrenowany model będzie zwracał nieprawidłowe wyniki. Nawet jeśli dane zostaną prawidłowo zebrane, trzeba pamiętać, że z czasem pewne zachowanie zjawisk może się zmieniać w raz upływu czasu. Przykładowo, system, który zwraca rekomendowaną muzykę, jeśli został wytrenowany w okolicach grudnia, jest duża szansa, że będzie proponował często muzykę świąteczną, co jak na ten okres jest oczekiwany wynikiem. Z biegiem czasu moda na tego typu utwory minie, więc jeśli model nie dostosuje się do nowych warunków, będzie wciąż proponował świąteczną muzykę, pomimo tego, że może być już lepiej i słuchacze bardziej chcą słuchać muzyki wakacyjnej.

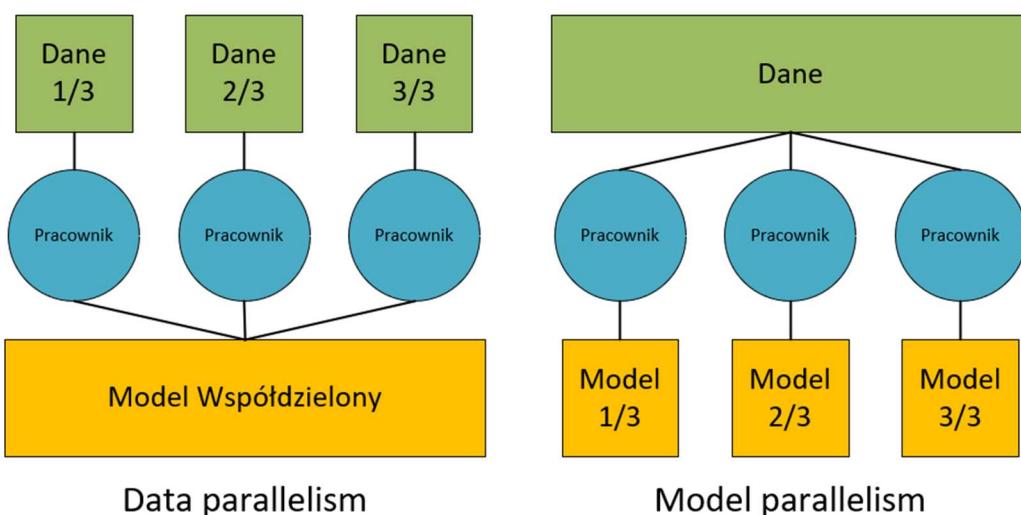
- Druga kategoria będąca powodem zwiększenia zawodności produktu, to czynniki związane z całym oprogramowaniem systemu. Mogą one wynikać ze zmian wprowadzanych podczas aktualizacji. Dobrym tego przykładem jest przypadek opisany na oficjalnym forum Keras na stronie github [12], gdzie programiści zgłaszały problem z zapisem modelu do formy gotowej do zaimplementowania na serwer. Najnowsze wersje TensorFlow od wersji 2.16, zaczęły korzystać z nowszej wersji biblioteki Keras, co spowodowało błędny zapis modelu. Jeśli dane przedsiębiorstwo nie wiedziały o tym błędzie i nie przetestowało wyeksportowanego modelu, mogliby narazić konsumentów na użycie niedziałającego produktu. Wina niepoprawnego działania systemu może leżeć również po stronie oprzyrządowania, np. awaria serwerów.
- Skalowalny – systemu ML (Machine Learning) muszą być odpowiednio zoptymalizowane do obsłużenia konkretnej ilości użytkowników. Musi zakładać możliwość powiększania się liczby konsumentów jak i samej ilości modeli, które mogą zostać wprowadzone. Skalowalność systemu spotyka się z następującymi problemami:
  - Zbyt duża ilość danych do przetworzenia podczas treningu modelu;
  - Optymalizacja modeli;
  - Zarządzanie zasobami;

- Organizacja środowiska developerskiego.

Systemy ML muszą się liczyć z dużą ilością danych do przetworzenia, zwłaszcza gdy muszą na bieżąco dostosowywać się do użytkowników. Ta ilość danych często przekracza fizyczne możliwości nośników pamięci RAM, dlatego by poradzić sobie z tym problemem stosuje się dwa następujące rozwiązania:

- Data parallelism;
- Model parallelism.

Idea tych rozwiązań opiera się na użyciu więcej niż jednej jednostki obliczeniowej w celu wytrenowania pojedynczego modelu, gdzie data parallelism polega na trenowaniu wielu modeli, który każdy z nich jest trenowany na osobnym zbiorze danych, a następnie otrzymane gradienty z klonów modeli są odpowiednio przeliczane tak, aby główny model został wytrenowany odpowiednio wytrenowany. Natomiast model parallelism zakłada, że model jest rozdzielony architektonicznie na nie jedną a kilka jednostek obliczeniowych, gdzie w każdej, odpowiednie warstwy sieci neuronowe wykonują operacje. Gdy jedna warstwa skończy swoją pracę, przesyła swoje dane wyjściowe jako dane wejściowe dla kolejnych warstw. Rysunek 3.3 przedstawia rysunek podgladowy, pokazujący architekturę tych rozwiązań.



Rysunek 3.3. Działanie Data parallelism i Model parallelism

- Łatwy w utrzymaniu – systemy ML potrafią składać się z wielu komponentów. Nawet pojedynczy model w systemie może posiadać wiele replik. Część z nich będzie poświęcona dla konsumentów, część z nich będzie pełnić rolę środowiska

testowego dla programistów. Dodatkowo też duży system ML może posiadać wielką bazę danych, która musi być uzupełniana w trakcie pracy modeli konsumentów oraz musi udostępniać swoje dane do modeli testowych jak i do modeli po stronie klienta, gdy serwer zwraca predykcje na za pomocą wsadów (predykcja wsadowa będzie wyjaśniona w rozdziale 3.2.2). W celu odpowiedniej kontroli całej architektury konieczne jest zapoznanie się z obecnymi możliwościami wdrożeń modeli oraz zrozumienie czym jest konteneryzacja i orkiestracja. Oba zagadnienia zostaną poruszone w kolejnych rozdziałach;

- Adaptywny – ML system musi sprostać wymaganiom biznesowym, dlatego należy zadbać o ciągłość dostarczanej usługi razem z możliwością jego rozwoju. Trzeba posiadać odpowiednio skonfigurowane środowisko i wielkie zasoby, aby w łatwy sposób można było zarządzać strukturami danego systemu. Patrząc od strony czysto modelu. Świat danych jest bardzo zmiennym środowiskiem, dlatego poza odpowiednim zarządzaniem zasobami, należy brać pod uwagę to, by model, o ile jest to możliwe, zawsze był w stanie wyuczyć się obecnego zachowania danych. Modele niestety przez ciągły trening mogą zacząć się przetrenowywać lub mieć ogólne problemy z estymowaniem danych funkcji, dlatego biblioteki do tworzenia sieci neuronowych oferują tworzenie punktów kontrolnych, które zapisują konkretny stan modelu w danym momencie treningu. Jest to bardzo pomocne, ponieważ gdy model w procesie uczenia pogorszy swoją dokładność i nie będzie w stanie w trakcie dalszego treningu z powrotem uzyskać swojej poprzedniej skuteczności, można wczytać jego ostatni punkt, odzyskując odpowiedni stan modelu. Może to okazać się przydatne w modelach, które uczą się preferencji swojego użytkownika. Program stopniowo uczy się tendencji konsumenta. W momencie zmiany zakresu upodobań klienta, doświadczenie modelu jest już niezgodne z rzeczywistością, dlatego w takim przypadku należy zresetować jego pamięć, żeby był w stanie dostosować się do nowych preferencji użytkownika.

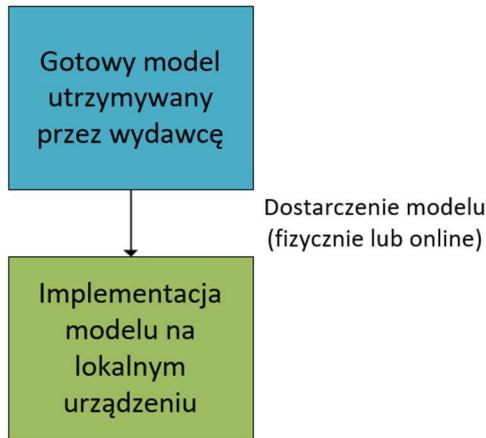
### 3.2.2 Sposoby implementacji gotowego modelu

Gdy model jest gotowy, należy go pewną metodą udostępnić konsumentom do użytku. Istnieje kilka podejść, które umożliwiają zrealizowanie tego zadania.

Jednym z najprostszych sposobów jest implementacja modelu po stronie klienta. Może być to zrealizowane na dwa sposoby [13]:

- Klient pobiera gotowy model z serwera na swoje urządzenie;
- Użytkownik korzysta z modelu, który jest zaimplementowany na stronie internetowej.

Pierwsze sposób jest jednym z najprostszych technik implementacji modelu, którego idea jest przedstawiona na rysunku 3.4.



Rysunek 3.4. Implementacja po stronie klienta

Model wcześniej jest tworzony i trenowany po stronie wydawcy, który zazwyczaj ma lepsze zasoby od docelowego klienta, które pozwalają mu na stworzenie oczekiwanej produkcji. Gotowy model jest zapisywany i udostępniany klientowi albo przez serwer, który przechowuje wersje modeli lub implementowany na miejscu na życzenie klienta. Modele obecnie mogą być zapisywane na różne sposoby, w różnych formatach.

Pierwszy sposób zapisu modelu polega na zapisie całej struktury modelu razem z wagami. TensorFlow oferuje następujące formaty plików, w których można przechowywać gotową sieć neuronową [14]:

- *.keras* jest najnowszym formatem używanym do zapisu całych modeli w TensorFlow, który jest obecnie zalecany od kiedy najnowsze wersje TensorFlow zaczęły używać Keras 3;
- *.h5* jest starszym formatem używanym w TensorFlow, który obecnie ma zastosowanie dla starszych systemów, które wciąż nie korzystają z biblioteki Keras 3;

- Saved Model w porównaniu do poprzednich form zapisu, model nie jest zapisywany w postaci jednego pliku a jako folder z następującą zawartością pokazaną na rysunku 3.5.

assets	6d ago
variables	6d ago
fingerprint.pb	6d ago
saved_model.pb	6d ago

Rysunek 3.5. Zawartość folderu przy zapisie formatem Saved Model

Zawartość folderu jest następująca [14]:

- Folder *assets* zawiera wszystkie elementy, które nie są powiązane bezpośrednio z samym zapisanym modelem, ale są wymagane do jego prawidłowego działania. Może to być przykładowo słownik, który może być wykorzystywany przez sieć przetwarzającą ludzką mowę;
- Variables przetrzymuje wszystkie trenowane zmienne. Znajdują się w nim dwa pliki: *variables.data-?????-of?????*, *variables.index*. Pierwszy wymieniony plik przetrzymuje faktyczne wartości zmiennych, a drugi zawiera informację, jakie konkretnie zmienne są zapisane, mając kontrolę nad tym czym są dane w *.data* pliku;
- Plik *fingerprint.pb* jest plikiem kontrolnym, który służy do kontroli użytkowania modeli oraz ich pochodzenia. Jest szczególnie przydatny, gdy posiadana jest do duża ilość modeli;
- Ostatni plik *saved\_model.pb* zawiera faktyczny model, który został zapisany.

Z reguły, do szybkiego zapisu modelu, powinien zostać użyty format *keras*, jednakże gdy model zawiera dodatkowe modyfikacje, należy wykorzystać zapis *Saved Model*.

W przypadku modeli zaprogramowanych za pomocą biblioteki PyTorch, odpowiednikiem będzie zapis modelu w postaci pliku z rozszerzeniem *pt* [15]. Dodatkowo można cały model zapisać w tym pliku w postaci *TorchScript*, dzięki czemu model może zostać odtworzony w środowisku nie związanym z językiem programowania Python.

Użytkownik, który chce użyć gotowych modeli, musi zaopatrzyć się w środowisko, które będzie w stanie skompilować dany model. Jeśli bardzo zależy konsumentowi, by model był użyty poza środowiskiem języka Python, powinien użyć *Saved Model* lub *TorchScript*, gdyż są do tego specjalnie przeznaczone.

Niestety przesyłanie klientom całego modelu może się wiązać z dużymi transferami danych. Teoretycznie jest możliwe ograniczyć ilość przesyłanych danych, nie wysyłając za każdym razem całego modelu, a tylko jego wagi. Jest to drugi sposób, w jaki można udostępnić model, ale zakłada on, że klient ma już całą architekturę modelu. W TensorFlow wagi jest możliwe przesłać w plikach *ckpt* [14], a w PyTorch w plikach *pt* [15].

Pomimo tego rozwiązania wciąż istnieje problem z dużym transferem danych, gdyż same wagi potrafią zajmować więcej pamięci niż cała reszta modelu. Dodatkowo, też trzeba zwrócić uwagę, że docelowe urządzenie może mieć znacznie mniejsze zasoby, niż te, którymi dysponuje wydawca systemu.

By sprostać temu problemowi, modele odpowiednio zostają uproszczone. Można to zrobić m.in. poprzez kwantyzację, która polega na zmniejszaniu precyzji wag, dzięki czemu można znaczco zmniejszyć rozmiar modelu za cenę jego dokładności. W celu zmniejszenia wielkości modelu TensorFlow oferuję TensorFlow lite [14], a PyTorch, PyTorch mobile [15]. Oba rozwiązania odpowiednio upraszczają modele do tego stopnia, że stają się możliwe do zaimplementowania na takich urządzeniach jak telefony komórkowe.

Drugim sposobem wdrożenia systemu ML po stronie użytkownika jest implementacja modelu na stronie internetowej. Polega na tym, że cały model jest zaimplementowany w skrypcie strony internetowej, co niweluje potrzebę pobierania jakiegokolwiek modelu, dodatkowo nie trzeba mieć przygotowanego środowiska, wystarczy posiadanie zwykłej przeglądarki internetowej. Takie rozwiązanie oferuje TensorFlow.js [14]. W przypadku PyTorcha nie istnieje jak na razie oficjalny odpowiednik, ale dostępna jest możliwość przekonwertowania modelu PyTorch na model ONNX, który można zaimplementować w skrypcie strony internetowej używając ONNX Runtime Web [15][16].

Kolejnym sposobem wdrażania modelu jest implementacja po stronie serwera. W takim rozwiązaniu cała logika i architektura modelu nie znajduje się po stronie klienta, więc jedyny dostęp, jaki do niego ma, jest możliwy poprzez odpowiednie API,

czyli tak zwany pośrednik pomiędzy serwerem a użytkownikiem końcowym. Można dany model udostępnić na dwa sposoby.

Pierwszy z nich opiera się na ciągłym udostępnianiu w czasie rzeczywistym modelu, zwany jako API streaming[13]. Jego działanie jest przedstawione na rysunku 3.6.



Rysunek 3.6. API streaming

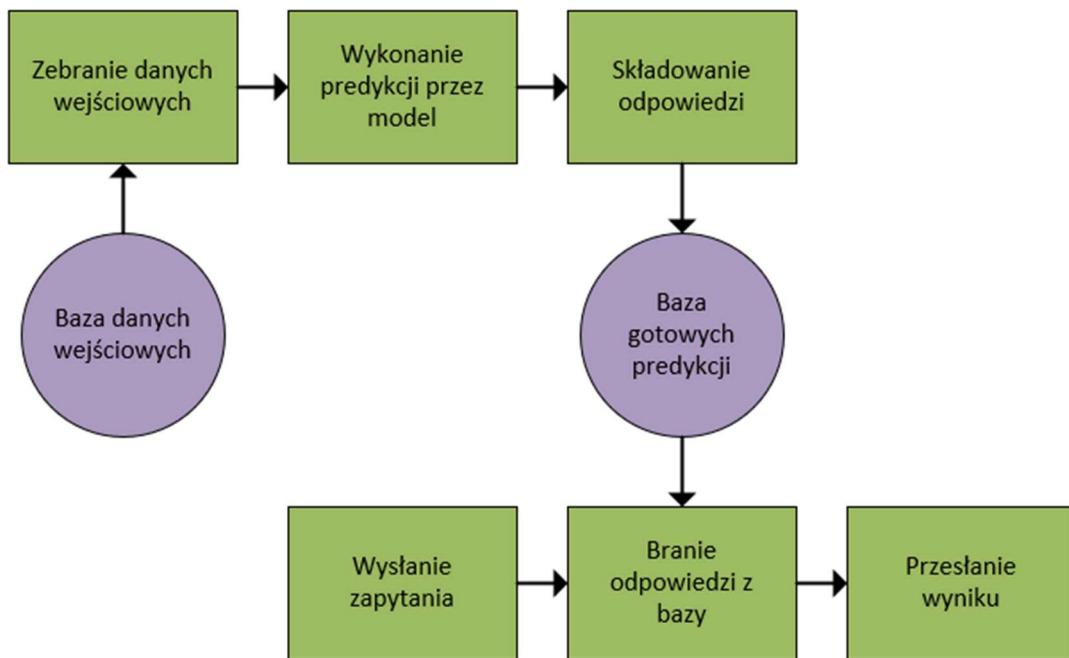
Cały proces możemy podzielić na 7 etapów:

- Pierwszy etap – użytkownik wysyła żądanie do systemu;
- Drugi etap – sprawdzane jest żądanie pod względami typu czy dana osoba jest autoryzowana do korzystania z tego modelu jak i jest sprawdzana poprawność formy przekazywanych danych;
- Trzeci etap – zbierane są dodatkowe dane, wymagane do prawidłowego działania modelu (mogą to być np. informacje na temat ostatniej słuchanej muzyki, jeśli mamy do czynienia z modelem rekomendującym muzykę);
- Czwarty etap – dane muszą zostać odpowiednio przetransformowane, w celu wprowadzenia ich do modelu;
- Piąty etap – przetworzone dane trafiają bezpośrednio do modelu;
- Szósty etap – przetworzenie danych wejściowych do formy odpowiednio czytelnej dla użytkownika;
- Siódmy etap – przesłanie odpowiedzi z wynikiem do użytkownika końcowego.

Drugi sposób udostępniania modelu jest mniej bezpośredni, ponieważ nie wymaga ciągłego udostępniania modelu. Użytkownicy w pewnych aspektach potrafią wykazywać pewne podobieństwa. Przykładowo, sklep posiada swoją stronę e-sklep z systemem rekomendacji produktów. Przykładowo, w okresie Bożonarodzeniowym najczęściej użytkownicy będą wyszukiwać takie rzeczy jak: prezenty, dekoracje świąteczne itp. Ze względu naczęstość pojawiania się podobnych danych wejściowych można je zebrać i podać je modelowi w jednym wsadzie by wykonał odpowiednie predykcje. Wyniki są zapisywane w bazie danych, skąd są pobierane i wysyłane użytkownikowi w momencie,

gdy dane wejściowe będą takie same jak te, które zostały użyte do wytworzenia danej predykcji.

Takie typu rozwiązanie nosi nazwę wsadowej predykcji [13], która jest przedstawiona na rysunku 3.7.



Rysunek 3.7. Predykcja wsadowa

Proces można podzielić na dwa osobne procesy. Pierwszy z nich to przygotowanie predykcji:

1. Uzbieranie wszystkich danych.
2. Przetworzenie danych wejściowych i zebranie ich we wsady.
3. Przetworzenie wsadów przez model.
4. Przetworzenie danych wyjściowych.
5. Składowanie uzyskanych wyników w bazie danych.

Drugi etap to obsługa użytkownika:

1. Użytkownik wysyła zapytanie.
2. API pobiera odpowiednie predykcje.
3. API przesyła predykcje do użytkownika.

Oba przedstawione rozwiązania należy dobrać zgodnie z wymaganiami danego systemu. API streaming jest zalecany w przypadku, gdy predykcja powinna być wykonywana w czasie rzeczywistym oraz żeby występowały niskie opóźnienia [10].

API streaming jest często stosowany w:

- Predykcjach finansowych w czasie rzeczywistym;
- W detekcji anomalii;
- Systemy rekommendacji czasu rzeczywistego;
- W predykcji uszkodzeń maszyn w trakcie użytkowania.

Predykcja wsadowa w porównaniu do API streaming jest dostosowany do systemów, które muszą charakteryzować się dużą przepustowością oraz gdy predykcje nie muszą być wykonywane natychmiastowo [10].

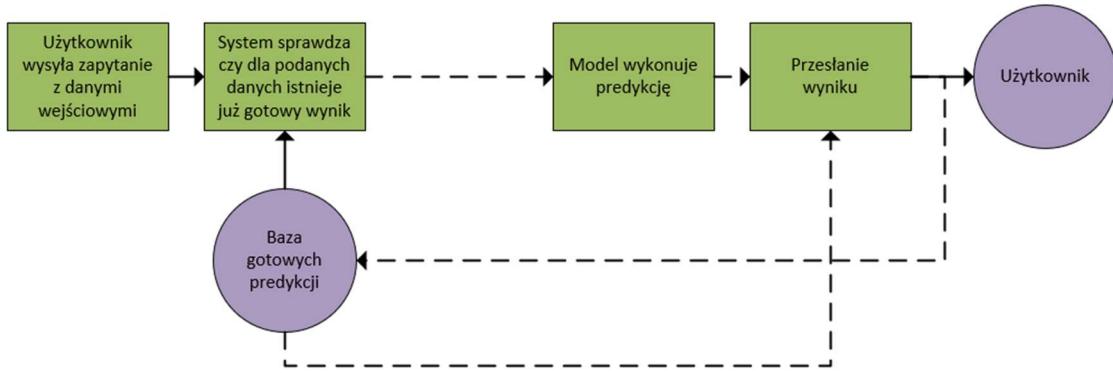
Predykcja wsadowa stosowana jest w:

- Generowaniu opisów książek w e-sklepach;
- Segmentacji klientów (po ich preferencjach i ostatnich zakupach);
- Wykrywaniu nadużyć bankowych;
- Ocenie ryzyka w opiece zdrowotnej.

Wszystkie powyższe metody udostępniania modeli po stronie serwera można też zastosować jednocześnie tworząc system hybrydowy, który daje dużą przepustowość jak i mały czas opóźnienia w przesłaniu odpowiedzi. Taki system działa w następujący sposób:

- System podstawowo operuje tak samo jak w predykcji wsadowej;
- W sytuacji gdy system trafi na zapytanie, które nie jest zapisane w bazie odpowiedzi, zapytanie jest przekazywane do przetworzenia przez model;
- Wynik predykcji zostaje po odpowiednim przetworzeniu końcowym zwracany do użytkownika oraz jest zapisywany w bazie razem z innymi predykcjami.

Cała logika działania została zilustrowana na rysunku 3.8.



Rysunek 3.8. Metoda hybrydowa

### 3.3 Serwis orkiestracyjny

Duże systemy ML ze względu na swoją wielkość oraz skomplikowaną architekturą potrzebują pewnego sposobu, który zagwarantuje zachowanie takich kluczowych cech jak skalowalność, adaptowalność i niezawodność. W tym celu tworzy się serwisy orkiestracyjne.

Serwis orkiestracyjny – jego definicja była podawana i rozwijana przez różne organizacje takie jak National Institute of Standards and Technology (NIST), European Telecommunications Standards Institute (ETSI), Open Networking Foundation (ONF). Spośród przedstawionych definicji serwis orkiestracyjny można opisać jako system, który zarządza całą infrastrukturą sieciową w sposób automatyczny, poprzez przekierowywanie zapytań do odpowiednich punktów końcowych, skalowanie ilości dostępnych zasobów sieci, zabezpieczanie komponentów systemów sieci oraz monitorowanie stanu całej architektury [17].

Serwis orkiestracyjny powinien oferować następujące funkcjonalności [18]:

- Koordynacja – serwis musi odpowiednio zarządzać i koordynować odpowiednio swoimi zasobami, aby zagwarantować ciągłość usługi. Wiąże się to z przede wszystkim przekierowywaniem zapytań do optymalnego zasobu, który w danej chwili jest w stanie jak najszybciej obsłużyć zapytanie użytkownika. Ponadto serwis musi mieć pod kontrolą ilość aktywnych zasobów, aby uruchamiać lub wyłączać odpowiednie punkty końcowe, w zależności od natężenia przepływu zapytań w danym czasie. System musi też zagwarantować właściwą komunikację pomiędzy zasobami, które są konieczne do zrealizowania zapytania;

- Automatyzacja – wszystkie procesy realizowane przez serwis powinny być wykonywane w sposób automatyczny przy jak najmniejszej ingerencji człowieka. System powinien posiadać opcje konfiguracji, które pełniłyby role reguł oraz scenariuszy przypadków użycia, które serwis powinien starać się zrealizować, a w szczególnych wypadkach sam podjąć decyzję, która by przyniosła jak najmniejsze szkody w stosunku do całej architektury systemu sieciowego;
- Zapewnienie zasobów i monitoring – serwis orkiestracyjny musi na bieżąco monitorować stan całego systemu, by w razie nieoczekiwanej awarii któregoś z komponentów, mógł szybko wyznaczyć następcę oraz podjąć próbę naprawy uszkodzonego zasobu.

### **3.4 Konteneryzacja**

Systemy uczenia maszynowego, żeby były łatwo skalowalne, muszą mieć sposób szybkiej implementacji. Firmy są zmuszone budować lub wynajmować różne serwery, które mogą mieć różne systemy operacyjne oraz z czasem są aktualizowane. By poradzić sobie z tymi wyzwaniami, firmy stosują tak zwaną konteneryzację oprogramowań.

Konteneryzacja – jest to sposób pakowania całego oprogramowania z kodem, bibliotekami a nawet systemem operacyjnym [19]. Konteneryzacja jest obecnie nieodłącznym elementem w budowaniu skalowalnych systemów. Technologia ta upakowuje całe środowiska w formie obrazu, który jest przepisem na stworzenie kontenera, który zawiera dokładnie wszystko to, co zostało utworzone w oryginalnym środowisku, od którego obraz pochodzi.

Definicja kontenera może być mylona z definicją maszyny wirtualnej, dlatego poniżej są tu przedstawione najważniejsze różnice pomiędzy tymi dwoma pojęciami [20]:

*Tabela 3.1. Porównanie kontenerów do maszyn wirtualnych*

Kontener	Maszyna wirtualna
Budowane są samowystarczalne pakiety oprogramowania, które działają niezależnie od maszyny i systemu operacyjnego, na którym działa. Obrazy takiego kontenera są tylko read-only co oznacza, że nie jest możliwe modyfikowanie takiego pliku, a jedynie jego odczyt i utworzenie na jego podstawie kontenera lub kolejnego obrazu.	Maszyny wirtualne działają jako niezależne wirtualne środowisko na komputerze, które może być swobodne modyfikowane bez narażenia głównego systemu na niepożądane zmiany. Eksportowany plik środowiska służy jedynie do jego ponownego utworzenia. Nie może być użyty bezpośrednio do stworzenia nowego pliku eksportowego.
Kontener w celu porozumiewania się z głównym systemem używa oprogramowań takich jak silnik kontenera lub środowisko wykonawcze kontenera. Te oprogramowania zapewniają dostęp do wymaganych zasobów, które są potrzebne dla działania kontenera.	Maszyna wirtualna używa tak zwanego hiperwizora (nadrządcy), który zarządza zasobami komputera w taki sposób, aby wszystkie procesy przeprowadzane przez maszynę wirtualną były całkowicie odseparowane od głównego systemu.
Pojemność kontenerów mieści się w jednostkach MB.	Maszyny wirtualne przez to, że są kopią całego systemu, ich pojemność jest wyrażana w GB.

Oba rozwiązania oferują uruchomienie danych oprogramowania na dowolnej maszynie z dowolnym systemem operacyjnym, co jest bardzo ważną zaletą, gdyż częstokroć istnieje potrzeba rozszerzenia całej architektury systemu. Może to się wiązać z powiększeniem systemu na inne usługi serwerowe, które mogą oferować inne oprogramowania. Użycie w takich przypadkach maszyn wirtualnych lub kontenerów jest wręcz wskazane, jednakże znaczącą przewagę ma tu konteneryzacja, ponieważ maszyny wirtualne są znacznie wolniejsze w implementacji w porównaniu z drugim rozwiązaniem. Kontenery są bardzo szybkie w implementacji i łatwe

w przechowaniu, dlatego są jednym z najlepszych rozwiązań do skalowania systemów sieciowych.

### 3.5 Skalowanie

Skalowanie jest jednym z głównych tematów omawianych w pracy dyplomowej. Ukazane zostaną dwa rodzaje skalowań, dlatego sprecyzowano definicje skalowania.

Skalowanie to dostosowanie zasobów systemu w stosunku do rosnącego zapotrzebowania na przetwarzanie danych [21].

Wyróżnia się dwa typy skalowania [21]:

- Skalowanie pionowe – jest powiązane z jednym urządzeniem. Wiąże się m.in. z fizycznym zwiększeniem zasobów, np. dodając do urządzenia kolejny dysk pamięci. Pionowe skalowanie wiąże się też z udostępnianiem większych zasobów wirtualnemu urządzeniowi od głównego urządzenia;
- Skalowanie poziome – wiąże się zwiększeniem ilości pojedynczych maszyn, powiększając możliwości całego systemu. Można to zrobić poprzez dodawanie dodatkowych fizycznych maszyn lub tworząc dodatkowe instancje maszyn wirtualnych.

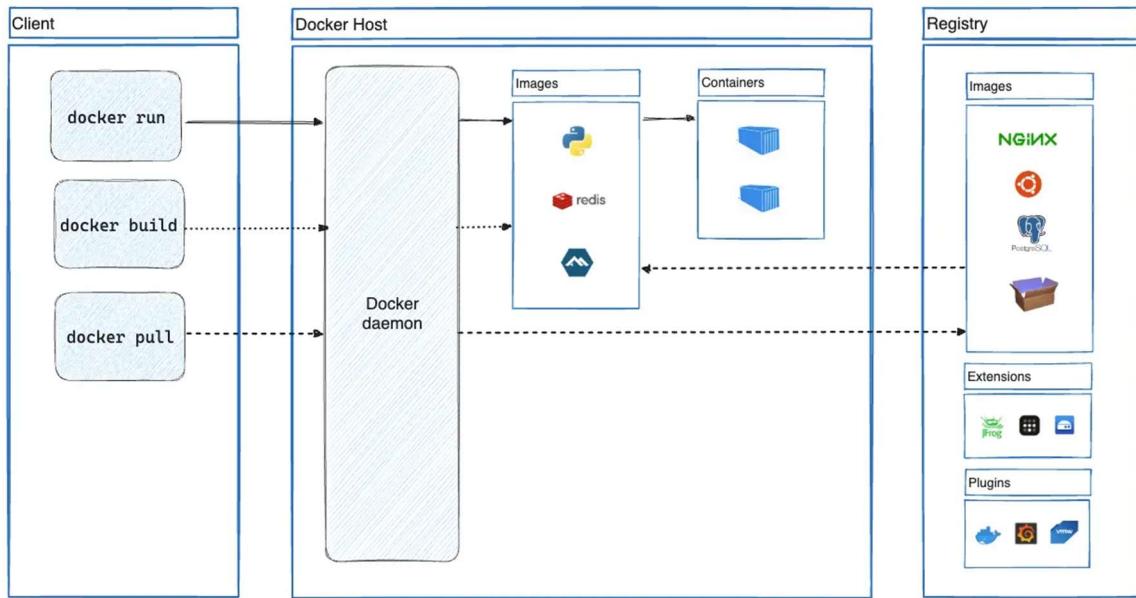
### 3.6 Technologie i platformy wykorzystane w pracy

#### 3.6.1 Docker

Docker jest jednym z narzędzi wykorzystywanych do konteneryzacji oprogramowań. Platforma oferuje prosty sposób pakowania i uruchamiania aplikacji w odizolowanych kontenerach, które mogą działać niezależnie od środowiska, równolegle względem siebie, nie narażając systemu na niepożądane interakcje pomiędzy nimi. Docker wspiera implementacje kontenerów w systemach orkiestrowanych. Zapewnia responsywną implementację i łatwą skalalność razem z szybkim działaniem i niewielką wagą samego kontenera.

Architektura Dockera jest oparta na relacji klient-serwer. Użytkownik za pomocą interfejsu *Docker client* może komunikować się z programem *Docker daemon*, który jest głównym zarządcą i kreatorem kontenerów. Interfejs *Docker client* i program *Docker daemon* mogą znajdować się na tej samej maszynie lub *interfejs* może zostać zdalnie połączony z programem *Docker daemonem* znajdujący się w innym środowisku.

W celu komunikacji został tu użyty popularny protokoł REST API, poprzez gniazda *UNIX* lub interfejs sieciowy [22].



Rysunek 3.9. Architektura Dockera [22]

Na powyższym rysunku 3.9 poza omówionymi elementami, można zauważyć rejestr Dockera, który składa się przede wszystkim obrazy kontenerów. Publiczny ogólnodostępny rejestr oferuje Docker HUB [22].

Obrazy mogą być tworzone według instrukcji zawartych w plikach bez rozszerzenia o nazwie *Dockerfile* lub na podstawie utworzonego kontenera. Popularną praktyką jest tworzenie jednego obrazu opierający się na drugim obrazie, co znaczco moze przyspieszyć tworzenie kontenerów, ponieważ duża ilość obrazów z gotowymi systemami oraz oprogramowaniem są ogólnodostępne już w Docker HUB. Można też przygotować podstawę dla danego obrazu, co nie zmusza do ciągłego powtarzania konfiguracji tego samego środowiska.

### 3.6.2 Kubernetes

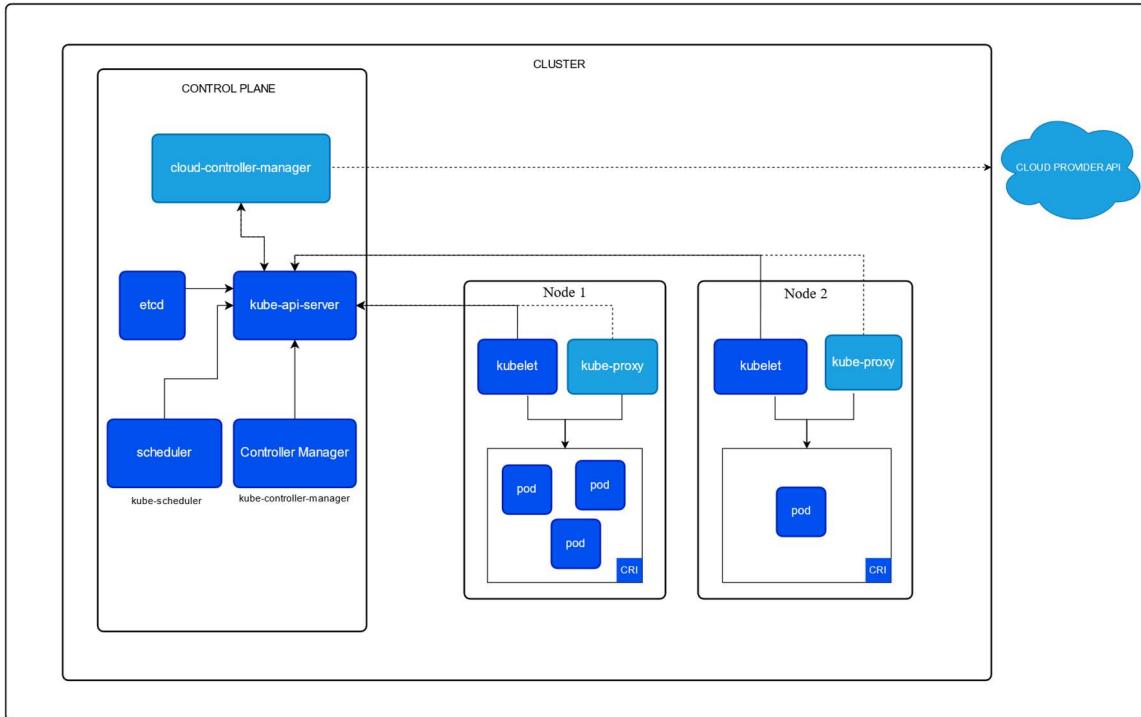
Kubernetes to przenośna i rozszerzalna *open source* platforma, która jest jednym z serwisów orkiestracyjnych. Umożliwia zarządzaniem serwisami, które działają w formie kontenerów. Platforma oferuje wiele funkcji w tym [23]:

- Wykrywanie usług i równoważenie obciążenia systemu;
- Orkiestracji pamięci masowej;
- Samo naprawa;
- Skalowanie poziome.

Kubernetes nie posiada zaawansowanych konfiguracji działania całego systemu jak i nie ma żadnego języka programistycznego, który by do tego służył. Kubernetes jest konfigurowalny deklaratywnie poprzez wbudowane API [23]. Użytkownik opisuje jaki stan systemu oczekuje, a platforma stara się to wykonać. Takie rozwiązanie sprawia, że użytkownik musi jedynie posiadać minimalną wiedzę o budowaniu dużych systemów sieciowych. Kubernetes umożliwia kolekcjonowanie i eksport metryk systemu (po wcześniejszym pobraniu odpowiedniego komponentu).

Jak wcześniej było wspomniane Kubernetes używa API do konfiguracji, dlatego można bezpośrednio wysyłać do platformy zapytania lub przygotować plik konfiguracyjny z rozszerzeniem *yaml* lub *json* albo skorzystać z *Kubernetes Dashboard*, będący graficznym interfejsem dla użytkownika.

By zrozumieć jak działa Kubernetes, należy przeanalizować przykładową architekturę systemu sieciowego opartej na platformie Kubernetes, która jest pokazana na rysunku 3.10.



Rysunek 3.10 Architektura klastra Kubernetes [23]

Cała architektura systemu znajduje się w klastrze. Wewnątrz klastra znajdują się węzły, które posiadają własne zasoby, czy to fizycznego urządzenia lub wirtualnego. Istnieją dwa rodzaje węzłów:

- Podstawowy węzeł – zawiera pody posiadający kontenery aplikacji, API, itp. Ich liczba zależy od potrzeb administratora i od tego jakie usługi oferuję, dlatego tych podów może być od kilku, aż po setki.;
- Panel kontrolny – jest rdzeniem całej logiki Kubernetes.

Panel kontrolny ma interfejs *kube-api-server*, który wysłuchuje wszystkie zapytania związane z serwisem Kubernetes jak i z usługami dostępnymi na innych węzłach.

Do serwera API są podłączone trzy ważne komponenty:

- Menadżer kontrolny – jednostka operacyjna, która wykonuje wszystkie procesy związane z zarządzaniem i utrzymaniem całej architektury systemu;
- Scheduler – zarządza kiedy i w jakim węźle zostanie zaimplementowany nowo stworzony pod [23];

- etcd – pamięć zapasowa całego stanu klastra, służąca do przywrócenia prawidłowego działania klastra oraz jego elementów w razie awarii.

Dodatkowym elementem panelu sterowania jest zarządcza *cloud-controler-manager*, który służy do interakcji z usługą w chmurze, skutecznie odizolowując komponenty działające w chmurze od tych, które znajdują się w klastrze [23].

W przypadku inicjowania architektury klastra dobrą praktyką jest posiadanie więcej niż jeden węzła z panelem sterowania, gdyż jeśli dojdzie do awarii tego komponentu, cały system ze wszystkimi usługami, jakie posiada przestaną całkowicie działać, co może spowodować paraliż po stronie konsumentów i kosztować administratora jak i klientów dużą ilość pieniędzy.

Pozostałe węzły, które udostępniają wszelkie usługi całego systemu, mają trzy ważne komponenty:

- Pod – jest to najmniejsza struktura w całym klastrze, która zawiera jeden lub więcej kontenerów danego serwisu lub aplikacji, korzystających z zasobów pojedynczego węzła [23]. Norma, przy tworzeniu poda określa, że powinna się znajdować jeden kontener z konkretnym oprogramowaniem, jednak Kubernetes nie ogranicza liczby możliwych instancji kontenera na jeden pod, więc gdy istnieje potrzeba podłączenia wymaganych kontenerów do działania głównego programu, Kubernetes na to zezwala. Pod z technicznego punktu widzenia jest abstrakcją wokół kontenera lub kontenerów, ustanawiając jeden pojedynczy element. Dzięki temu Kubernetes nie komunikuje się bezpośrednio z kontenerem, a jest wykonywana interakcja tylko z podem, ujednolicając cały protokół komunikacyjny. Może to ułatwiać korzystanie z kontenerów od różnych serwisów, takich jak m.in. Docker, więc używanie różnych rodzajów kontenerów w różnych podach nie powinno mieć większego znaczenia dla funkcjonalności całego węzła;
- Kubelet – agent działający wewnętrz węzła, który zapewnia działanie kontenerów w podach zgodnie z założeniami przekazanymi przez administratora oraz umożliwia komunikację pomiędzy innymi elementami w całym klastrze [23];
- Container runtime – kieruje działaniem kontenerów. Odpowiada za całe wykonywanie jego operacji oraz jego cykl życia [23].

Żeby wszystkie węzły mogły ze sobą się komunikować, w klastrze jest utworzona wirtualna sieć. W tej sieci wszystkie pody dostają swój własny adres IP, umożliwiając komunikację z danymi usługami. Niestety adres ten nie jest stały. Gdy pod przestanie funkcjonować, Kubernetes przeniesie jego zadania na replikę tego poda, jeśli taki istnieje, a uszkodzony pod będzie starał się odtworzyć ze stanu zapisanego w komponencie *etcd*, co skutkuje przypisaniem innego adresu IP, uniemożliwiając w ten sposób innym elementom na skuteczną interakcję z tym podem. By poradzić sobie z tym problemem, do poda przypina się serwis komponent, który daje mu stałe IP, które nie zmienią się, nawet po ponownym odtworzeniu poda.

Klaster w Kubernetes może być skonfigurowany na sztywno manualnie przez użytkownika, określając stałą ilość węzłów. Przy większych sieciach to podejście może okazać się mało optymalne, dlatego lepiej zautomatyzować skalowanie w klastrze. Istnieje wiele narzędzi, które do tego służą, dlatego zostaną tu wspomniane dwa główne:

- Cluster Autoscaler – automatycznie dostosowuje rozmiar klastra, gdy zachodzi przynajmniej jeden z dwóch wypadków [23]:
  - Istnieją pody, które nie mogą prawidłowo działać, gdyż nie posiadają wystarczającą ilość zasobów;
  - Istnieją węzły, które przez dłuższy okres czasu, nie były w pełni wykorzystywane oraz mogą być umieszczone w innych istniejących już węzłach;
- Karpenter – jest to *open-source* menadżer do zarządzania projektami zbudowanymi w Kubernetes [24]. Jest dostosowany do skalowania całego systemu biorąc pod uwagę całkowity koszt utrzymania infrastruktury. Jest zintegrowany z dostawcą zasobów w chmurze, przez to jest w stanie sam wynajmować potrzebne zasoby.

Poza skalowaniem samego klastra, jest możliwe też skalowanie komponentów w pojedynczym węźle. Wspomniane było, że węzeł może posiadać nawet setki podów, dlatego ręczne ich zarządzanie jest bardzo problematyczne, dlatego pody są umieszczane w obiektach zwanych *workload*. Dzięki temu panel sterujący jest w stanie zautomatyzować zarządzanie podami, w tym wprowadzić auto skalowanie.

Kubernetes jest w stanie realizować skalowanie obiektu *workload* poziomo jak i pionowo. Skalowanie poziome realizuje poprzez manipulację ilości replik podów w zależności od obserwowanego zużycia zasobów. Narzędzie implikujące tą możliwość to

*HorizontalPodAutoscaler* (*HPA*). Skalowanie pionowe jest realizowane dzięki obiekowi *VerticalPodAutoscaler* (*VPA*). Program opiera się na tworzeniu własnych definicji zasobów, które można później użyć zamiast tych wbudowanych. *VPA* ma dostępne trzy tryby pracy [23]:

- Recreate – tryb ten odpowiednio skaluje pody przy ich tworzeniu jak i w trakcie ich działania;
- Initial – skaluje pody tylko przy ich tworzeniu;
- Off – wyłącza usługę auto skalowania, ale przesyła wciąż rekomendację działań jakie można byłoby podjąć w kwestii skalowania.

### 3.6.3 TensorFlow serving

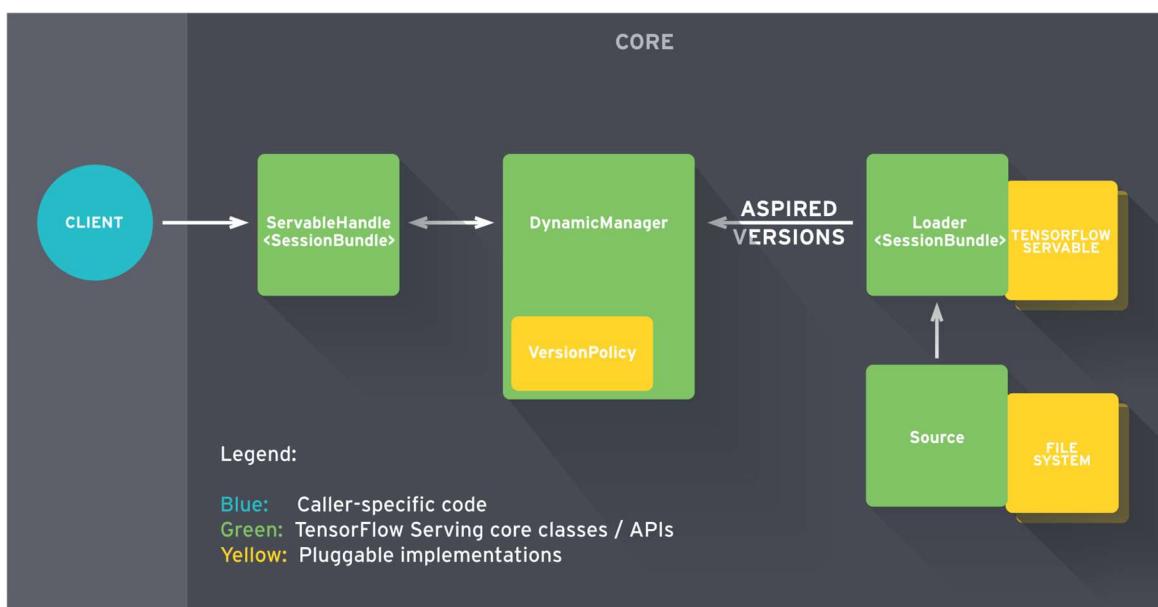
TensorFlow serving jest elastycznym systemem udostępniającym systemy uczenia maszynowego, skierowany dla środowisk produkcyjnych. Jest dostosowany do wdrażania modeli stworzonych w technologii TensorFlow.

Architektura serwera składa się z następujących komponentów [14]:

- Servables – są to abstrakcje, z których korzystają użytkownicy (np. predykcja modelu). Nie zarządzają swoją żywotnością, jedynie udostępniają swoje zaprogramowane usługi. Interfejsy *servables* mogą być wersjowane oraz można korzystać z dowolnych ich instancji. Dzięki temu można eksperymentować z różnymi wersjami oraz nie ograniczać użytkownika, tylko jedną wersją interfejsu *servables*. Model na takim serwerze jest umieszczony jako jeden lub wiele interfejsów *servables*;
- Loader – jest zarządcą obiektów *servables*. Standaryzuje sposób ładowania i rozładowywania instancji *servables*, niezależnie z jakimi algorytmami ma do czynienia;
- Sources – to wtyczka, która udostępnia zero lub wiele strumieni instancji *servable* (strumień obiektu *servable* to łańcuch różnych jego wersji ułożonych w kolejności rosnącej względem numeru wersji). Jest też elementem, który tworzy obiekt *loader* do interfejsu *servable*;
- Manager (tłum. Menadżer) – odpowiada za obserwację zbioru *aspired versions*. Wczytuje wszystko co jest zawarte w tej liście i udostępnia te obiekty. Gdy strumień zbioru *aspired version* zmieni swoją zawartość,

to wczyta wszystkie nowe wersje interfejsów *servable* oraz odłączy te, które nie były zawarte w strumieniu.

Wyjaśniając działanie i zbiór *aspired versions*. Wtyczki *sources* obserwują wszelkie zmiany w plikach, do których się odnoszą. Jeśli zostanie utworzona wtyczka *source* lub jego obserwowalna zawartość się zmieni, pobiera odpowiednie komponenty, tworząc łańcuch wersji, gdzie dla każdej wersji jest tworzony obiekt *loader*. Obiekty są gromadzone w zbiorze nazywanym *aspired versions*. Menadżer dostaje sygnał o nowych gotowych wersjach do załadowania, więc po sprawdzeniu wszelkich wymagań związane z polityką wersji, przekazuje do obiektu *loader* odpowiednie zasoby i go uruchamia w celu pobrania interfejsu *servable*. Od tego momentu manager może przyjmować i przetwarzanie zapytania skierowane do konkretnej instancji *servable* [14].



Rysunek 3.11. Architektura TensorFlow serving [14]

Cała architektura znajduje się wewnątrz rdzenia TensorFlow serving, który odpowiada za zarządzaniem całym okresem życia komponentów oraz udostępnia metryki [14]. Jest to jedyny element, z którym użytkownik faktycznie zachodzi w interakcje. Cała reszta architektury jest odizolowana i niewidoczna dla niego.

TensoFlow serving oferuje możliwość dodania własnych wersji powyższych komponentów, poza menadżerem. Menadżer może być zmodyfikowany jedynie poprzez nadanie polityki wersji, która określa zasady wczytywania i usuwania interfejsów *servable*.

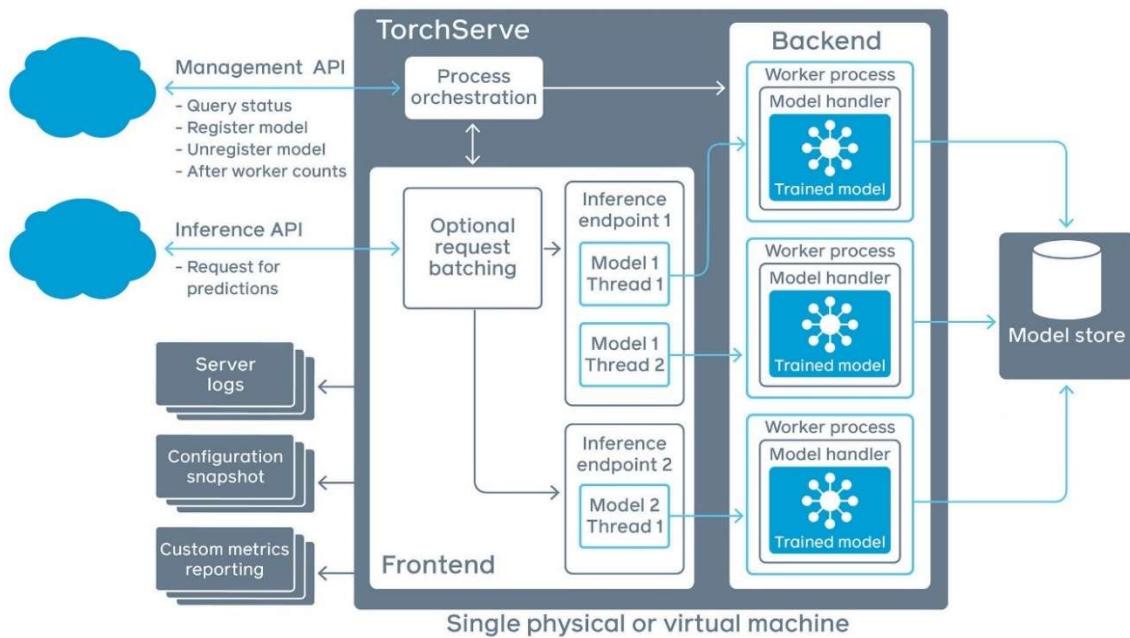
TensorFlow serving wspiera RESTful API jak i gRPC API (rozwiązanie pochodzące od firmy Google).

Serwer może być skalowany w pionie poprzez:

- Serwis wsadowy – jest to sposób serwisu, który zakłada, żeby zamiast przetwarzania każdego zapytania pojedynczo, zebrać je w jeden wsad i przekazać je do modelu. Jest możliwość skonfigurowania:
  - maksymalnej wielkości wsadu;
  - maksymalny czas oczekiwania na utworzenie się wsadu – ustawia się to dla zachowania ciągłości usługi, żeby zapytania zostały zrealizowane, pomimo że wsad nie osiągnął swojej maksymalnej wielkości;
  - maksymalną ilość wątków, gdzie wsady zostają przetworzone przez model;
  - maksymalna ilość wsadów do przetworzenia – serwer względem tego parametru wybiera ustawioną ilość wsadów do obsłużenia, biorąc pod uwagę te, które zostaną zrealizowane najszybciej, cała reszta wsadów jest odrzucana. Stosuje się to w celu zniwelowania opóźnień w kolejce.
- Zwiększenie ilości używanych rdzeni procesora, do działania serwera – można ustawić ile rdzeni ma być używanych równolegle, w ramach wykonania zadań na jedną operację oraz ile rdzeni może być używanych równolegle do wykonania operacji ogółem.

#### 3.6.4 TorchServe

TorchServe to odpowiednik TensorFlow serving dla modeli tworzonych w technologii PyTorch. W porównaniu do swojej konkurencji, serwer ten jest bardziej przyjazny dla programistów Python, gdyż duża część kodu jest napisana w tym języku, co ułatwia wprowadzenie modyfikacji oraz możliwość korzystania z bibliotek Pythona. TensorFlow serving jest w większości oparty na C++ i nie ma tak dużego wsparcia dla modyfikacji wprowadzanych w Pythonie. Architektura TorchServe, pokazana na rysunku 3.12, wygląda następująco:



Rysunek 3.12. Architektura TorchServe [15]

- Model store – zawiera pliki *mar*, w którym znajdują się wszystkie komponenty potrzebne do implementacji modeli do serwera. By taki plik uzyskać, PyTorch udostępnia narzędzie zwane *torch-model-archiver*. Modele też mogą być wersjowane i udostępniane na podobnej zasadzie co TensorFlow serving;
- Backend jest głównie napisany w Pythonie. Posiada następujące elementy:
  - Worker (pracownik) – odpowiada za faktyczne działanie operacyjne całego modelu na serwerze;
  - Model handler – jest to program wewnątrz pracownika. W nim można określić, w jaki sposób cały model pracuje, np. przetwarzanie danych wejściowych jak i wyjściowych. Można też zainportować biblioteki niepochodzące od TorchServe. Istnieje tam duża swoboda pod względem tego co można zaprogramować;
  - Trained model – faktyczny model importowany i obsługiwany przez program *model handler*.

- Frontend jest napisany głównie w języku Java i odpowiada za tworzenie punktów końcowych, zarządza wszystkimi zapytaniami oraz zbiera wszelkie metryki i logi serwera [15].

Skalowanie pionowe na tym serwerze można zrealizować na następujące sposoby:

- Inicjowanie większej ilości pracowników – każdy model może mieć więcej niż jednego pracownika. Pojedynczy pracownik zajmuje jeden wątek;
- Serwis wsadowy – konfiguracja wygląda trochę inaczej niż w TensorFlow serving. Można tu skonfigurować tylko:
  - Maksymalna wielkość wsadu;
  - Maksymalny czas oczekiwania na utworzenie wsadu.

## 4. Cześć projektowa

### 4.1 Definicja celu oraz wymagań

Ze względu na dużą ilość metod budowania systemów oraz ograniczenia sprzętowe, wybrano przedstawić metodę, polegającą na zbudowaniu streaming API. Jest to jeden z najbardziej popularnych metod, dodatkowo posiada duże pole manewru pod względem realizowania różnych metod skalowania. W ramach części praktycznej zostały zbudowane serwery lokalne, na których wykonano testy w zakresie opóźnień na odpowiedź od serwera. W ten sposób było możliwe określenie skuteczności skalowania serwerów.

Przedstawione zostały dwa sposoby skalowania: pionowy i poziomy. Projekt skupił się wokół dwóch rozwiązań używanych do implementacji systemów ML: TensorFlow serving oraz TochServe.

Etapy realizacji projektu są następujące:

- Zbudowanie oraz wytrenowanie sieci konwolucyjnych zdolnych do rozpoznawania emocji na podstawie fotografii z twarzą człowieka;
- Zaprogramowanie funkcji, które zapewnią pożądane działanie serwerów;
- Eksport gotowych funkcji i modeli;
- Utworzenie klastra z aplikacją internetową Grafana oraz z serwerem do zbierania metryk Prometheus;
- Przeprowadzenie skalowania w pionie:
  - Przygotowanie skryptów do testów obciążeniowych;

- Inicjowanie serwerów z różnymi konfiguracjami oraz ich testowanie;
- Analiza otrzymanych wyników.
- Testowanie skuteczności skalowania w poziomie:
  - Przeprowadzenie skalowania poprzez tworzenie replik oraz konfiguracja komponentu HPA;
  - Przeprowadzenie testów obciążeniowych
  - Analiza otrzymanych wyników oraz obserwacja zachowania się klastra.

## 4.2 Przygotowanie sieci konwolucyjnych do rozpoznawania obrazów

### 4.2.1 Przygotowanie fotografii do treningu

Wykorzystany został fragment zbioru fotografii nazwanej AffectNet, dostępnej na stronie internetowej kaggle.com [25]. Zdjęcia zostały pobrane i przeniesione do folderu emotions. Fotografie były uporządkowane w foldery według emocji, którą przedstawiały. Foldery nie były podpisane nazwami emocji, lecz zostały ponumerowane, a baza danych nie posiadała opisu, dlatego nadano im nazwy zgodnie z subiektywną oceną. W folderze emotions było 5 kategorii emocji: złość, radość, neutralny (brak dużego nacechowania emocjonalnego), smutek i strach.

W celu sprawnego wczytania zdjęć został stworzony plik *emotions.csv*, który zawiera poniższą tabelę, przedstawioną na rysunku 4.1.

	image_path	label
1	/home/adam/Bachelor/emotions/sad/image0029680.jpg	sad
2	/home/adam/Bachelor/emotions/sad/image0024066.jpg	sad
3	/home/adam/Bachelor/emotions/sad/image0009807.jpg	sad
4	/home/adam/Bachelor/emotions/sad/image0018272.jpg	sad
5	/home/adam/Bachelor/emotions/sad/image0026864.jpg	sad

Rysunek 4.1. *emotions.csv*

W celu utworzenia takiego pliku został napisany program, który jest pokazany na rysunku 4.2.

```
[1]: import os
      import pandas as pd

[2]: root_dir = '/home/adam/Bachelor/emotions'

[3]: image_paths = []
      labels = []

[4]: for root, dirs, files in os.walk(root_dir):
      for file in files:
          image_path = os.path.join(root, file)
          label = os.path.basename(root)
          image_paths.append(image_path)
          labels.append(label)

[6]: df = pd.DataFrame({
      'image_path': image_paths,
      'label': labels
})

[8]: df.head()

[8]:
```

	image_path	label
0	/home/adam/Bachelor/emotions/sad/image0029680.jpg	sad
1	/home/adam/Bachelor/emotions/sad/image0024066.jpg	sad
2	/home/adam/Bachelor/emotions/sad/image0009807.jpg	sad
3	/home/adam/Bachelor/emotions/sad/image0018272.jpg	sad
4	/home/adam/Bachelor/emotions/sad/image0026864.jpg	sad

```
[9]: df.to_csv("emotions.csv", index=False)
```

Rysunek 4.2. Program do utworzenia pliku emotions.csv

#### 4.2.2 Transformacja danych

By fotografie mogły zostać prawidłowo załadowane do modelu, należy przekształcić je z formy graficznej na formę numeryczną. Każde zdjęcie kolorowe jest możliwe przedstawić za pomocą formatu RGB, gdzie obraz jest zapisywany za pomocą trzech macierzy, reprezentujący zgodnie z podaną kolejnością kolory: czerwony, zielony oraz niebieski. Macierze zawierają wartości pikseli, które tworzą obraz. Zakres liczbowy w tym formacie wynosi 0-255. Liczby te reprezentują jasność, z jaką świeci dany piksel. Im większa liczba, tym większa intensywność danego koloru w danym pikselu.

Samo sprowadzenie fotografii do postaci numerycznej nie jest wystarczające. Kolory nie pełnią dużej roli w wykrywaniu emocji, dlatego można je uproszczyć, przekształcając obrazy na format czarno-biały. W ten sposób otrzymano analogiczny format do RGB, ale była tu obecna tylko jedna macierz, opisująca skale szarości pikseli. Dzięki temu skrócono niepotrzebną liczbę informacji, upraszczając model oraz skracając czas potrzebny na trening.

Kolejny etap to przekształcenie rozmiarów zdjęć. Modele sieci neuronowych domyślnie przyjmują próbki o stałej ilości cech. Różne rozmiary zdjęć, co za tym idzie, różne wymiary macierzy mogły okazać się bardzo problematyczne, dlatego każda fotografia została przekształcona do rozmiaru 100 na 100 pikseli.

Ostatni etap to standaryzacja wartości pikseli. Zakres liczbowy był zbyt duży, co mogło negatywnie wpłynąć na obliczanie gradientów podczas treningu, dlatego wszystkie wartości pikseli zostały podzielone przez maksimum możliwej wartości, czyli przez 255. Skutek jest taki, że zakres liczbowy po zastosowaniu tej operacji wynosił od 0 do 1, co skutecznie usprawniło proces uczenia.

Podczas treningu poza fotografiami, muszą zostać dostarczone etykiety zdjęć w celu wyliczenia błędu modelu, służący do treningu całej sieci neuronowej, dlatego wykonano transformacje nazw na liczby zgodnie z rysunkiem 4.3.

```
{'sad': 0, 'scared': 1, 'neutral': 2, 'happy': 3, 'angry': 4}
```

Rysunek 4.3. Słownik emocji

Ilość zdjęć jest na każdą kategorię równa 5000, co razem daje 25000 fotografii. Zbiór podzielono na część treningową (15000 fotografii) i testową (10000 fotografii). Zbiór treningowy musiał zostać wymieszany, by nie sugerować modelowi,

że istnieje jakaś zależność związana z kolejnością zdjęć. Kod podziału danych można zobaczyć na rysunku 4.4.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True, test_size=0.4, random_state=123, stratify=y)

len(X_train), len(X_test)

(15000, 10000)
```

Rysunek 4.4. Podział danych oraz ich wymieszanie

Ilość zdjęć była zbyt duża na możliwości obliczeniowe używanej maszyny obliczeniowej, dlatego stworzono odpowiednio dla każdej technologii (TensorFlow oraz PyTorch) obiekt *dataset*, który przekształca oraz przekazuje próbki do modelu, przygotowując w trakcie treningu wsad z dziesięcioma zdjęciami, bez potrzeby zapisu całego gotowego zbioru do pamięci podręcznej urządzenia. Kod tych obiektów jest widoczny na rysunkach 4.5, 4.6.1, 4.6.2.

```
dataset_train = tf.data.Dataset.from_tensor_slices((X_train, y_train))
dataset_test = tf.data.Dataset.from_tensor_slices((X_test, y_test))

10000 00:00:1735590794.334346    5016 gpu_device.cc:2022] Created device
-Q Design, pci bus id: 0000:01:00.0, compute capability: 6.1

def load_image(path, label):
    image = tf.io.read_file(path)
    image = tf.image.decode_jpeg(image, channels=3)
    image = tf.image.rgb_to_grayscale(image)
    image = tf.image.resize(image, [100, 100])
    image = image / 255.0
    return image, label

dataset_train = dataset_train.map(load_image)
dataset_test = dataset_test.map(load_image)

dataset_train = dataset_train.batch(10).prefetch(tf.data.AUTOTUNE)
dataset_test = dataset_test.batch(10).prefetch(tf.data.AUTOTUNE)
```

Rysunek 4.5. dataset w TensorFlow

```

class CTDataset(Dataset):
    def __init__(self, image_paths, labels, device = 'cpu'):
        self.image_paths = image_paths
        self.labels = labels
        self.device = device

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        label = self.labels[idx]
        image = cv.imread(img_path, cv.IMREAD_COLOR)
        image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
        image = cv.resize(image, (100, 100))
        image = torch.tensor(image, dtype=torch.float32)
        image = image / 255.0
        image = image.unsqueeze(0).to(device)
        label = torch.tensor(label).to(device)
        return image, label

```

Rysunek 4.6.1. dataset w PyTorch

```

train_data = CTDataset(X_train, y_train, device=device)
train_data_loader = DataLoader(train_data, batch_size=10, shuffle=False)

test_data = CTDataset(X_test, y_test, device=device)
test_data_loader = DataLoader(test_data, batch_size=10, shuffle=False)

```

Rysunek 4.6.2. dataset w PyTorch

Po wykonanych transformacjach model otrzymywał obrazy, które w postaci czytelnej dla ludzkiego oka wyglądały tak jak na rysunku 4.7.



Rysunek 4.7. Przetworzone fotografie

#### 4.2.3 Architektura sieci konwolucyjnej

Architektura sieci jest zgodna z opisem kodu na rysunku 4.8.

```
model = Sequential()
# warstwa konwolucyjna kernel = (3,3) wejście: 1 wyjście: 10 funkcja aktywacji ReLU
model.add(Conv2D(10, (3, 3), activation = 'relu', input_shape=(100, 100, 1)))
# Max pooling kernel = (2,2)
model.add(MaxPooling2D((2, 2)))
# Dropout z szansą 25%
model.add(Dropout(0.25))
# warstwa konwolucyjna kernel = (3,3) wejście: 10 wyjście: 30 funkcja aktywacji ReLU
model.add(Conv2D(30, (3, 3), activation = 'relu'))
# Max pooling kernel = (2,2)
model.add(MaxPooling2D((2, 2)))
# Dropout z szansą 25%
model.add(Dropout(0.25))
# Zamiana macierzy na wektor
model.add(Flatten())
# w pełni połączona sieć neuronowa wejście: 15870 wyjście: 500 funkcja aktywacji ReLU
model.add(Dense(500, activation = 'relu'))
# Dropout z szansą 50%
model.add(Dropout(0.5))
# w pełni połączona sieć neuronowa wejście: 500 wyjście: 5 funkcja aktywacji log_softmax
model.add(Dense(5, activation = 'log_softmax'))
```

Rysunek 4.8. Architektura sieci neuronowej w TensorFlow

Gdzie:

- Kernel oraz zapis np. (3,3) opisuje wymiar macierzy używanej w danej warstwie;
- Wejście i wyjście mówi w przypadku warstw konwolucyjnych ile macierzy zostaje dostarczanych oraz ile jest przekazywanych do dalszej warstwy. Dla warstw pełni połączonych są już to pojedyncze neurony.

Zastosowano na samym końcu funkcję aktywacji *log softmax*, który stosuje się w celu uzyskania wektora, gdzie wartości liczbowe przedstawiają prawdopodobieństwo przynależności do danej kategorii emocji. Aby uzyskać procentowy wynik, należy podnieść otrzymany wyniki do potęgi liczby e oraz wymnożyć je liczbą 100. Wynik taki po wykonaniu obliczeń przyjmuje postać tablicy, która może wyglądać w następujący sposób: [75, 5, 5, 5, 10]. Słownik z rysunku 4.3 przypisuje indeks tablicy, której ta emocja dotyczy.

Dodatkowo można zauważyc operacje *dropout*. Jest to sposób minimalizacji szans na przetrenowanie modelu. Działa w taki sposób, że zgodnie z podanym prawdopodobieństwem decyduje się na wyzerowanie wyniku wychodzącego z danego

neuronu. Czyli przykładowo dla warstwy, dla której jest 500 neuronów wyjściowych, to około 250 z nich, ich wartości wyjściowe będą wyzerowane.

Analogicznie zbudowano sieć używając biblioteki PyTorch. Kod architektury przedstawiony jest na rysunku 4.9.

```
class CNN_model(nn.Module):
    def __init__(self):
        super().__init__()
        # First layer
        self.conv1 = nn.Conv2d(1, 10, (3, 3))
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(2, 2)
        self.dropout1 = nn.Dropout(p=0.25)
        # Second layer
        self.conv2 = nn.Conv2d(10, 30, (3, 3))
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(2, 2)
        self.dropout2 = nn.Dropout(p=0.25)
        # To linear layer
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(15870, 500)
        self.relu3 = nn.ReLU()
        self.dropout3 = nn.Dropout(p=0.5)
        # Final layer
        self.linear2 = nn.Linear(500, 6)
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.maxpool1(x)
        x = self.dropout1(x)

        x = self.conv2(x)
        x = self.relu2(x)
        x = self.maxpool2(x)
        x = self.dropout2(x)

        x = self.flatten(x)
        x = self.linear1(x)
        x = self.relu3(x)
        x = self.dropout3(x)

        x = self.linear2(x)
        x = self.logsoftmax(x)

        return x
```

Rysunek 4.9. Architektura sieci neuronowej w PyTorch

#### 4.2.4 Trening i walidacja sieci konwolucyjnych

Dla sieci ustawiono następujące parametry do treningu:

- Liczba epok równa 30 – pełne przejście wszystkich danych ze zbioru treningowego nazywa się jedną epoką;
- Współczynnik uczenia równy 0,001 – tempo uczenia się im większy tym mniej epok jest potrzebny do wytrenowania sieci. Zbyt duża wartość skutkuje utratą zdolności do uczenia się;
- Optymalizator Adam – określa sposób aktualizacji wag. Charakteryzuje się dynamiczną zmianą współczynnika uczenia, zwiększając prędkość treningu sieci.

Każdy model ma ustawioną funkcję straty pochodzącą z biblioteki, za pomocą której są zaprogramowane. Oba są dostosowane do oceny prawdopodobieństw w problemach klasyfikacyjnych.

Poniżej na rysunkach 4.10, 4.11, 4.12, 4.13 zostały przedstawione zaprogramowane parametry oraz powstałe po wykonanym treningu krzywe uczenia, które przedstawiają wartość funkcji straty w danej epoce (gdy funkcja zbliża się do wartości zerowej, znaczy, że model został wytrenowany).

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
model.compile(optimizer = optimizer, loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), metrics=['accuracy'])
```

Rysunek 4.10. Parametry w TensorFlow

```
def train_model(data, model, n_epochs = 50, learning_r=0.001):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    model.train()
    opt = optim.Adam(model.parameters(), lr=learning_r)
    loss_fn = nn.NLLLoss().to(device)

    losses = []
    epoches = []

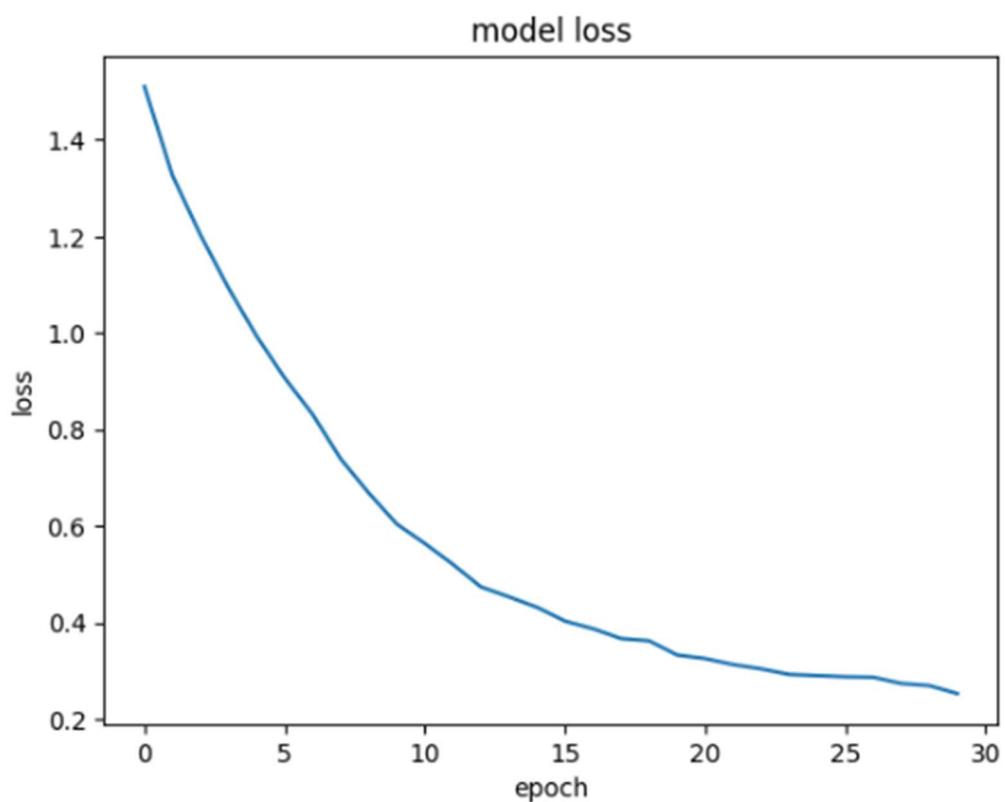
    for epoch in range(n_epochs):
        N = len(data)
        for i, (x, y) in enumerate(data):
            opt.zero_grad()
            losse_value = loss_fn(model(x), y)
            losse_value.backward()
            opt.step()

            epoches.append(epoch+i/N)
            losses.append(losse_value.item())
            print(f'Epoch: {epoch+1}/{n_epochs}, loss: {losse_value}')

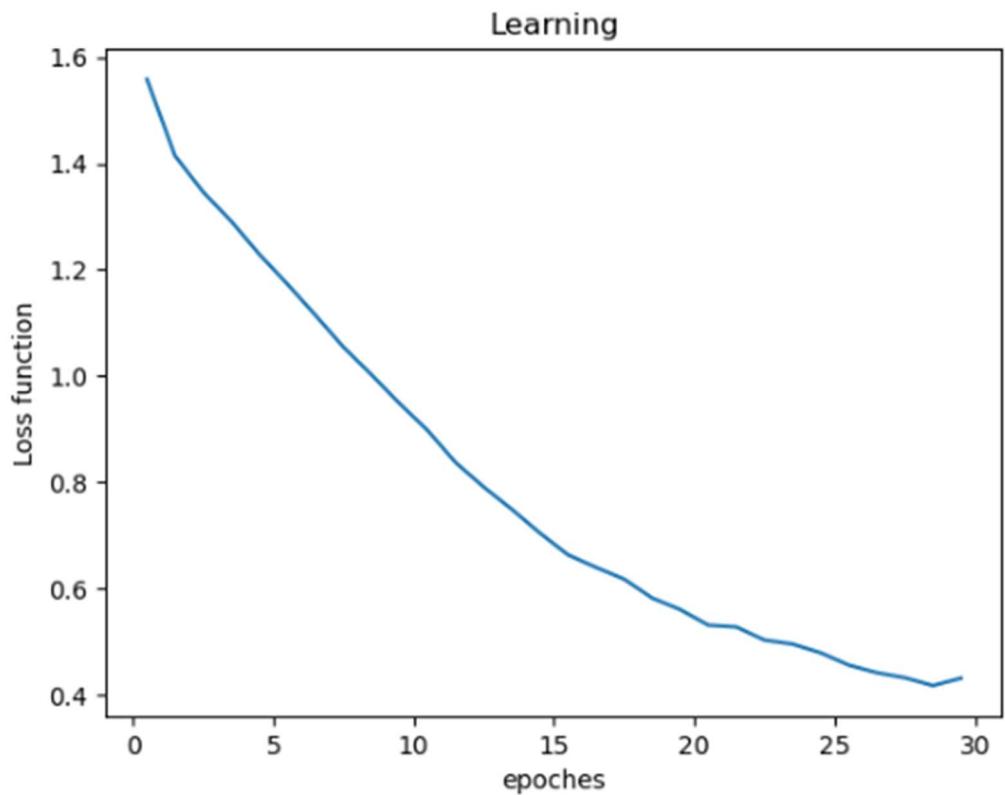
    return np.array(epoches), np.array(losses)

epoch_n = 30
learning_rate = 0.001
f = CNN_model().to(device)
epoches, losses = train_model(train_data_loader, f, n_epochs=epoch_n, learning_r=learning_rate)
```

Rysunek 4.11. Parametry w PyTorch



Rysunek 4.12. Krzywa uczenia w TensorFlow



Rysunek 4.13. Krzywa uczenia w PyTorch

Modelom została zmierzona dokładność, gdzie dla modelu wykonanego w TensorFlow dla zbioru treningowego wynosi około 99% a dla testowego 52%. Dla PyTorch analogicznie 97% i 51%. Klasyfikacja zdjęć poprzez losowanie dałoby dokładność około 20%, więc można wywnioskować, że modele zdołały się nauczyć pewnych wzorców i nie określały wyników w sposób przypadkowy.

### 4.3 Eksport modeli

Gotowe modele należy odpowiednio wyeksportować. Każda technologia ma swój własny sposób eksportu, lecz każdy z nich składa się z dwóch etapów: napisanie funkcji obsługującej model przez serwer oraz tworzenie odpowiednich plików.

Są dwa założenia dla funkcji obsługujących model. Pierwsze z nich zakłada, że obrazy będą przekazywane w formie *base64*. Drugie założenie zakłada, że oczekiwane dane wyjściowe będą przedstawione w formie słownika, który może wyglądać przykładowo w następujący sposób: *{sad: 0.000000, scared: 0.000539, neutral: 0.915596, happy: 0.000001, angry: 0.083864}*.

#### 4.3.1 Eksport w TensorFlow

Aby model był odpowiednio obsługiwany przez serwer w TensorFlow serving należy zaprogramować funkcję *signature*, w którym zostaną zdefiniowane procesy transformacji danych wejściowych, uzyskanie predykcji z modelu oraz transformacje danych wyjściowych. Istnieją gotowe funkcje *signature*, jednakże na potrzeby projektu zaprogramowano autorskie rozwiązanie. Funkcja odpowiedzialna za to zadanie jest pokazana na rysunku 4.14.

```

@tf.function(input_signature=[tf.TensorSpec(shape=(None), dtype=tf.string)])
def pre_and_pro_process(images_bytes):
    def preprocess(image_bytes):
        len_img = tf.strings.length(image_bytes)
        image_raw = tf.strings.substr(image_bytes, pos=2, len=len_img - 3)
        image_raw = tf.io.decode_base64(image_raw)
        image = tf.image.decode_image(image_raw, channels=3, expand_animations = False)
        image = tf.image.rgb_to_grayscale(image)
        image = tf.image.resize(image, [100, 100])
        image = image / 255.0

        return image

    # preprocess
    images = tf.map_fn(preprocess, images_bytes, dtype=tf.float32)

    # predict
    predictions = tf.math.exp(model(images))

    # postprocess
    def map_to_dict(prediction):
        emotions = ['sad', 'scared', 'neutral', 'happy', 'angry']
        prediction_strings = tf.strings.as_string(prediction)
        pairs = tf.stack([emotions, prediction_strings], axis=1)

        def join_pair(pair):
            return tf.strings.join([pair[0], ": ", pair[1]])

        joined_pairs = tf.map_fn(join_pair, pairs, dtype=tf.string)

        result = tf.strings.join(["{", tf.strings.reduce_join(joined_pairs, separator=", "), "}"])
        return result

    formatted_predictions = tf.map_fn(map_to_dict, predictions, dtype=tf.string)

    return formatted_predictions

```

Rysunek 4.14. Signature

Funkcja `preprocess` działa prawie tak samo jak funkcja używana do przygotowania datasetu do treningu, z tą różnicą, że obraz był przekazany w formie tekstowej w formacie `base64`, dlatego wykonywane jest wcześniej wydzielanie odpowiedniego fragmentu tekstowego z `image_bytes` oraz jest poddawana dekodowaniu do formatu *RGB*.

Predykcje są uzyskiwane poprzez przekazanie danych do modelu oraz podniesienie wyników do potęgi e.

Dane wyjściowe są odpowiednio mapowane według instrukcji zawartych w funkcji `map_to_dict`. Zwrócony wynik predykcji ma następującą postać: `[[0.000000, 0.000539, 0.915596, 0.000001, 0.083864]...]`. Jest to tablica tablic. Wewnętrzne tablice zawierają pojedynczą predykcję. Liczby przedstawiają kolejno następujące emocje: smutek, strach, neutralny, radość oraz złość. W tym przypadku model uznał, że osoba przedstawiona na zdjęciu ma neutralną mimikę twarzy, gdyż jest pewny tego

najbardziej, w porównaniu z innymi opcjami (pewność dla tej emocji wynosi około 92%). Ilość tablic jest zależny od wielkości wsadu dostarczonego do modelu. Funkcja *map\_to\_dict* przyjmuje jedną predykcję i tworzy tablice, które zawierają wartość prawdopodobieństwa oraz etykietę. Później te tablice są przekształcane w tekst w postaci etykieta: *prawdopodobieństwo*. Następnie wszystkie teksty są sprowadzane do postaci zgodnej z założeniem.

Jedyną rzeczą odstającą od standardowych funkcji w języku Python jest dekorator *tf.function*. Jest to sposób interpretacji przez TensorFlow funkcji w postaci grafowej. Jest to oryginalny sposób działania tej biblioteki, za nim wprowadzono możliwość wykonywania kodu w tak zwanym trybie *eager mode*, gdzie operacje są od razu wykonywane bez budowania grafu. TensorFlow serving jest dostosowany do działania na kodzie przekazanym w postaci grafu, dlatego istniała potrzeba użycia tego dekoratora. Poza jego wywołaniem określa się też w nim normę, w jakiej postaci ma się spodziewać otrzymania danych wejściowych (*tf.TensorSpec*). W tym przypadku jest to forma tekstowa.

Z gotową funkcją można wyeksportować model do formatu *Saved Model*. Kod tego procesu jest pokazany na rysunku 4.15.

```
export_archive = ExportArchive()
export_archive.track(model)
export_archive.add_endpoint(name="serve", fn=pre_and_pro_process)
export_archive.write_out("model_store_tf/1/")
```

Rysunek 4.15. Eksport modelu w TensorFlow

#### 4.3.2 Eksport PyTorch

W przypadku biblioteki PyTorch w celu wyeksportowania modelu należy utworzyć plik mar za pomocą programu *torch-model-archiver*. Narzędzie to potrzebuje plik modelu oraz zdefiniowany plik *handler*, który definiuje inicjalizację modelu oraz całe jego działania podobnie jak program *signature* w TensorFlow. TorchServe posiada wbudowane programy *handler*, ale na potrzeby spełnienia założeń projektu została zaprogramowana autorska wersja programu *handler*. By zbudować ten program, należało stworzyć klasę, która dziedziczy od klasy *BaseHandler*. Klasa ta posiada wiele funkcji, które można nadpisać. W projekcie zostały użyte następujące z nich:

- initialize – funkcja pobierająca model;
- preprocess – odpowiada za transformacje danych wyjściowych;
- inference – zwraca dane z predykcji modelu;

- postprocess – transformuje dane wyjściowe;
- handle – odpowiada za odpowiednie wywołanie funkcji preprocess, inference oraz postprocess.

Zaprogramowana klasa *handler*, nazwana ModelHandler, posiada następujące pola:

- context – pole to przyjmuje obiekt, który zawiera wszelkie informacje na temat właściwości serwera (np. dostępne jednostki obliczeniowe) oraz ścieżki kierujące do plików zawierający model. Każde inne pole poza zmienną *initialized* przyjmują wartości pochodzące z tego pola;
- initialized – zwraca informacje czy model został zainicjowany do serwera;
- model – przypisuje mu się obiekt modelu;
- device – przyjmuje wartość jednostki obliczeniowej, która jest dostępna (w tym wypadku wybór jest pomiędzy CPU a GPU);
- model\_pt\_path – ścieżka do pliku zawierający wagi modelu;
- manifest – przypisany jest mu obiekt zawierający ścieżki do plików.

Istnieje wiele sposobówinicjalizacji modelu. W tym konkretnym przypadku model aby został zainicjowany, potrzebne są dwa pliki: plik programu Python zawierający klasę modelu, oraz plik z rozszerzeniem *pt*, który zawiera wagi modelu. Funkcja *initialize* realizująca wczytanie modelu jest przedstawiona na rysunku 4.16.

```
def initialize(self, context):
    self.manifest = context.manifest

    properties = context.system_properties
    model_dir = properties.get("model_dir")
    self.device = torch.device("cuda:" + str(properties.get("gpu_id")) if torch.cuda.is_available() else "cpu")

    serialized_file = self.manifest["model"]["serializedFile"]
    self.model_pt_path = os.path.join(model_dir, serialized_file)
    model_file = self.manifest["model"].get("modelFile", "")

    model_class = self.getModel(model_file)
    self.model = model_class()
    self.model.to(self.device)
    state_dict = torch.load(self.model_pt_path, map_location=self.device)
    self.model.load_state_dict(state_dict)

    self.model.eval()

    self.initialized = True
```

Rysunek 4.16. Funkcja initialize

Dodatkowo została stworzona funkcja *getModel*, która importuje z pliku *py* pierwszą napotkaną klasę. Jest pokazana na rysunku 4.17.

```
def getModel(self, model_file):
    module = importlib.import_module(model_file.split(".")[0])
    model = None
    for attrName in dir(module):
        attribute = getattr(module, attrName)
        if inspect.isclass(attribute):
            model = attribute
    return model
```

Rysunek 4.17. Funkcja *getModel*

Kolejna funkcja *preprocess*, widoczna na rysunku 4.18, działa analogicznie jak klasa *CTDataset*, z tą różnicą, że obrazy są przesyłane w formacie *base64*. Dodatkowo funkcja jest dostosowana do gromadzenia wielu żądań w jeden wsad, który zostaje przekazany modelowi.

```
def preprocess(self, data):
    images = []
    for row in data:
        preprocessed_data = row.get("data")
        if preprocessed_data is None:
            preprocessed_data = row.get("body")

        base_64_data = base64.urlsafe_b64decode(preprocessed_data["data"][-len(preprocessed_data["data"])-1:])
        np_preprocessed_data = np.frombuffer(base_64_data, dtype=np.uint8)
        image = cv.imdecode(np_preprocessed_data, flags=cv.IMREAD_COLOR)
        image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
        image = cv.resize(image, (100, 100))
        image = torch.tensor(image, dtype=torch.float32)
        image = image / 255.0
        image = image.unsqueeze(0)
        images.append(image)

    return torch.stack(images).to(self.device)
```

Rysunek 4.18. Funkcja *preprocess*

Funkcja *inference* odpowiada za zrealizowanie predykcji i zwrócenie jej wyniku. Nie posiada żadnych dodatkowych funkcji, gdyż w tym wypadku nie były one wymagane.

Kod funkcji *postprocess* (rysunek 4.19) zwraca pożądany wynik predykcji. Jej działanie opiera się na zaprogramowanej funkcji *translateOutput* (rysunek 4.20), która podobnie jak w przypadku technologii TensorFlow przyjmuje tablice, które po podniesieniu jego wartości do potęgi e przyjmują postać  $[[0.000000, 0.000539, 0.915596, 0.000001, 0.083864]...]$ . Funkcja iteruje po całym wsadzie, a następnie iteruje po pojedynczych predykcjach przypisując każdej wartości odpowiednią emocję.

Podpisane wyniki trafiają do słownika *answer*, który później jest składowany w tablicy *answers*, zawierający wszystkie docelowe odpowiedzi serwera.

```
def postprocess(self, inference_output):
    postprocess_output = self.translateOutput(inference_output)
    return postprocess_output
```

Rysunek 4.19. Funkcja *postprocess*

```
def translateOutput(self, output):
    emotions = {'sad': 0, 'scared': 1, 'neutral': 2, 'happy': 3, 'angry': 4}
    answers = []
    exp_output = torch.exp(output)
    for pred in exp_output:
        answer = {}
        for i in emotions:
            answer[i] = pred[emotions[i]].item()
        answers.append(json.dumps(answer))

    return answers
```

Rysunek 4.20. Funkcja *translateOutput*

Ostatnia funkcja *handle* (rysunek 4.21) wywołuje wszystkie funkcje potrzebne do wygenerowania odpowiedzi serwera oraz wczytuje wartości dla pola *context*.

```
def handle(self, data, context):
    self.context = context

    model_input = self.preprocess(data)
    model_output = self.inference(model_input)
    return self.postprocess(model_output)
```

Rysunek 4.21. Funkcja *handle*

Po zaprogramowaniu programu *handler*, należy utworzyć plik *pt* oraz użyć narzędzia *torch-model-archiver*, który wygeneruje plik *mar* potrzebny serwerowi do zainicjowania usługi dla modelu. Eksport jest wykonywany za pomocą kodu, pokazanego na rysunku 4.22.

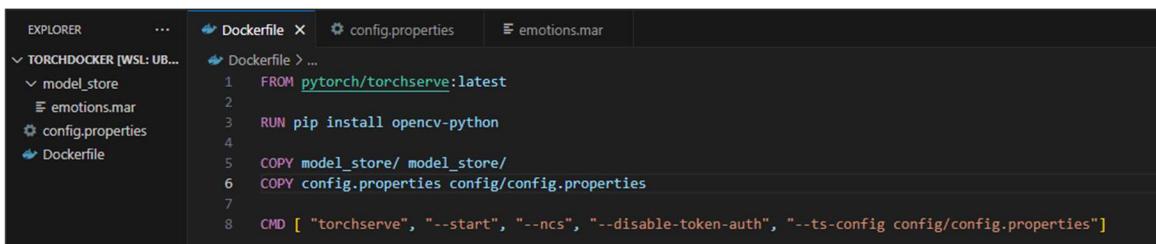
```
# Export to model.pt
torch.save(f.state_dict(), 'model.pt')
```

Rysunek 4.22. Eksport modelu w PyTorch

Komenda dla torch-model-archiver wygląda następująco: `torch-model-archiver --model-name emotions --version 1.0 --model-file /home/adam/Bachelor/model.py --serialized-file /home/adam/Bachelor/model.pt --export-path /home/adam/Bachelor/model_store --handler /home/adam/Bachelor/model_handler.py`.

#### 4.3.3 Utworzenie obrazów Docker

W projekcie stworzono wiele obrazów, które posiadają różne konfiguracje związane ze skalowaniem pionowym serwerów. Poniżej zostanie przedstawiony i omówiony przykładowy *Dockerfile*, który jest wykorzystywany do utworzenia serwera TorchServe.



The screenshot shows a code editor with a dark theme. On the left, there's an 'EXPLORER' sidebar showing a project structure: 'TORCHDOCKER [WSL: UB...]' containing 'model\_store', 'emotions.mar', 'config.properties', and 'Dockerfile'. The main area displays a Dockerfile with the following content:

```
FROM pytorch/torchserve:latest
RUN pip install opencv-python
COPY model_store/ model_store/
COPY config.properties config/config.properties
CMD [ "torchserve", "--start", "--ncs", "--disable-token-auth", "--ts-config config/config.properties"]
```

Rysunek 4.23. Dockerfile

Na rysunku 4.23 jest przedstawiony *Dockerfile*, za pomocą którego Docker wie, jaki ma stworzyć obraz. Można tu zauważyć pięć komend:

- FROM – służy do importowania gotowych obrazów, które stanowią bazę dla nowo utworzonej wersji. W tym przypadku pobrany zostaje obraz torchserve, gdzie fragment *:latest* wskazuje na to, by podczas tworzenia obrazu została użyta najnowsza wersja TorchServe;
- RUN – komenda, która jest wykonywana w trakcie tworzenia obrazu. Używa się ją m.in. do instalowania potrzebnych zależności;
- COPY – służy do kopирования folderów oraz plików do kontenera, który powstaje na podstawie obrazu. Pierwsza wskazana ścieżka, wskazuje na zasoby należące do urządzenia użytkownika. Druga ścieżka wskazuje na miejsce, w którym mają się znaleźć skopiowane zasoby wewnątrz kontenera.
- CMD – komenda wywoływana po utworzeniu kontenera. Pokazana komenda inicjuje TorchServe.

#### 4.4 Utworzenie środowiska klastrowego za pomocą Minikube

W celu przedstawienia możliwości skalowania TorchServe oraz TensorFlow serving utworzono środowisko testowe za pomocą Minikube, który jest kontenerem inicjującym Kubernetes klaster na poziomie lokalnym. W celu potwierdzenia działania stosowanych parametrów do skalowania serwerów zostaną utworzone dwa pody, które zawierają następujące usługi:

- Prometheus serwer – umożliwia zbieranie metryk z serwerów w formacie prometheus;
- Grafana – usługa internetowa, która umożliwia wizualizacje zebranych metryk pochodzących od Prometheus serwer.

By zainicjować pody, należy przygotować pliki konfiguracyjne *yaml*. Każdy tego typu plik użyty w projekcie ma podobną konstrukcję i zawartość, dlatego zostanie omówiony tylko jeden z nich.

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: prometheus-config
5  data:
6    prometheus.yml: |
7      global:
8        scrape_interval: 10s
9
10     scrape_configs:
11       - job_name: 'tensorflow_serving'
12         static_configs:
13           - targets: ['tensorflow-serving-svc:8501']
14             labels:
15               __metrics_path__: '/monitoring/prometheus/metrics'
16 ---
17  apiVersion: v1
18  kind: Service
19  metadata:
20    name: prometheus-server
21  spec:
22    ports:
23      - port: 9090
24        targetPort: 9090
25    selector:
26      app: prometheus
27 ---
```

Rysunek 4.24.1. Plik konfiguracyjny Kubernetes

```

28   apiVersion: apps/v1
29   kind: Deployment
30   metadata:
31     name: prometheus
32   spec:
33     replicas: 1
34     selector:
35       matchLabels:
36         app: prometheus
37     template:
38       metadata:
39         labels:
40           app: prometheus
41       spec:
42         containers:
43           - name: prometheus
44             image: prom/prometheus:latest
45             ports:
46               - containerPort: 9090
47             volumeMounts:
48               - name: config-volume
49                 mountPath: /etc/prometheus
50                 readOnly: true
51             volumes:
52               - name: config-volume
53             configMap:
54               name: prometheus-config

```

Rysunek 4.24.2. Plik konfiguracyjny Kubernetes

Powyżej na rysunkach 4.24.1 i 4.24.2 są przedstawione następujące konfiguracje dla Prometheus serwer. Można znaleźć tutaj trzy rodzaje komponentów:

- ConfigMap – służy do przechowywania zmiennych środowiskowych, komend oraz pliki konfiguracyjne. W powyższym przypadku jest tu napisany plik, który wskazuje adres, skąd mają zostać pobrane metryki oraz dodatkowo ustalono interwał pobierania metryk, który wynosi 10 sekund;
- Deployment – tutaj jest inicjowany faktyczny serwer. Definiuje się w tym miejscu jego parametry oraz zależności związane z resztą komponentów utworzonych w podzie. Są tutaj trzy ważne parametry:

- replicas – definiowana jest tutaj ilość replik poda. Jest to podstawowy parametr, który umożliwia skalowanie w poziomie;
- image – stąd Kubernetes ma informacje jakiego obrazu powinien użyć do stworzenia kontenera;
- containerPort – aby pod miał możliwość komunikacji z kontenerem, należy określić port kontenera;
- Service – służy do ustalenia stałego adresu IP, który służy do komunikacji z podem.

Mając tak przygotowany plik konfiguracyjny można poprzez komendę `kubectl apply -f <ścieżka do pliku>` utworzyć pod. W razie konieczności nadania zmian, można zmodyfikować plik i powtórnie użyć tej samej komendy. Kubernetes wykryje zmiany i spróbuje zrealizować nowe wytyczne. Utworzonye pody i serwisy są widoczne na rysunkach 4.25 i 4.26.

(base) adam@Adam:~\$ kubectl get pods	NAME	READY	STATUS	RESTARTS	AGE
	grafana-6b7696bdd7-ht64w	1/1	Running	3 (20h ago)	2d19h
	prometheus-65655d754-sk7sk	1/1	Running	1 (20h ago)	21h
	tensorflow-serving-5dc7f7d676-f5ng7	1/1	Running	1 (27m ago)	20h

Rysunek 4.25. Status podów

(base) adam@Adam:~\$ kubectl get services	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
	grafana	ClusterIP	10.97.39.29	<none>	80/TCP	2d21h
	kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	21d
	prometheus-server	ClusterIP	10.101.117.185	<none>	9090/TCP	22h
	tensorflow-serving-svc	ClusterIP	10.98.196.169	<none>	8501/TCP	22h

Rysunek 4.26. Serwisy dostępne w klastrze

## 4.5 Wdrażanie skalowalnych systemów uczenia maszynowego

W tym rozdziale pokazane zostanie część możliwości, jakie oferują narzędzia TensorFlow serving, TorchServe oraz Kubernetes. Przy skalowaniu pionowym zostaną zaprezentowane ważne znaczenie metryk podczas kontroli czasu oczekiwania zapytań oraz w konfiguracji serwisu wsadowego. W ramach skalowania poziomego ukazane zostaną możliwości tworzenia replik podów oraz zostanie skonfigurowana opcja auto skalowania.

#### 4.5.1 Skalowanie pionowe

Na początku przedstawione zostaną możliwości TensorFlow serving.

Parametry, które będą manipulowane to:

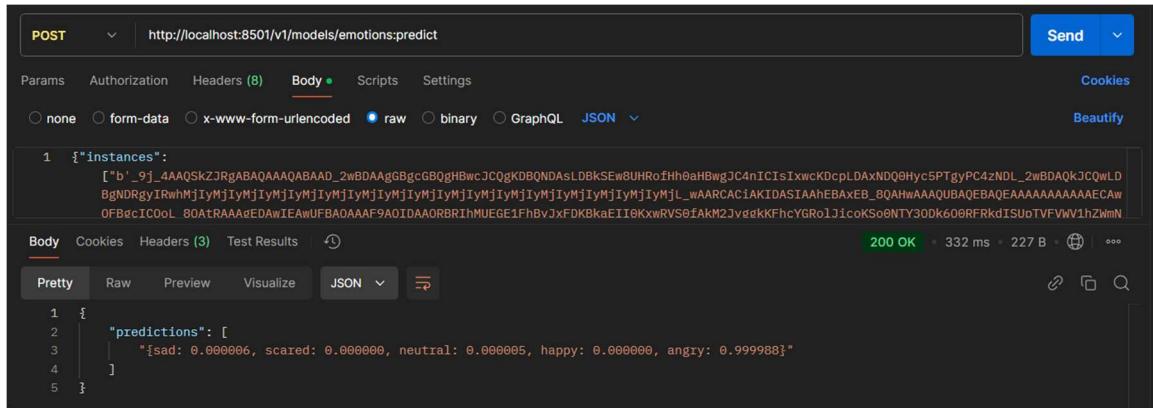
- tensorflow\_intra\_op\_parallelism – parametr wyznaczający ilość wątków wykorzystywanych do realizacji pojedynczej operacji. Może to się tyczyć realizacji operacji macierzowych w ramach operacji predykcji modelu;
- tensorflow\_inter\_op\_parallelism – analogicznie jak poprzedni parametr, jednak obejmuje równoległą realizację wielu operacji. Wspomaga obsługiwanie wiele żądań jednocześnie;
- w przypadku włączenia serwisu wsadowego:
  - max\_batch\_size – maksymalna wielkość wsadu, którą model może obsłużyć;
  - batch\_timeout\_micros – czas oczekiwania na zapełnienie wsadu. Parametr ten zapobiega niepotrzebnym oczekiwaniom na odpowiedź serwera, jednakże przy pojedynczych zapytaniach, serwer będzie zwracał odpowiedź z opóźnieniem.

Najpierw zainicjowany został serwer na domyślnych ustawieniach, które obejmują wyłączony serwis wsadowy oraz tensorflow\_intra\_op\_parallelism i tensorflow\_inter\_op\_parallelism przyjmując wartości równą ilości dostępnych rdzeni procesora. Pierwsza czynność, jaka została wykonana to weryfikacja działania usługi. Przesłany został do predykcji w formacie base64 obraz przedstawiony na rysunku 4.27.



Rysunek 4.27. Fotografia testowa

Wysłano żądanie za pomocą narzędzia Postman do serwera, który wyświetlił następującą odpowiedź pokazaną na rysunku 4.28.



Rysunek 4.28. Działanie serwera

Serwer odpowiedział na żądanie, więc następną czynnością do wykonania jest przeprowadzenie testu obciążeniowego w celu weryfikacji wydajności serwera. Do wykonania testów użyto programu k6. Na rysunku 4.29 można zobaczyć kod, który konfiguruje scenariusz, jaki będzie realizować test.

The screenshot shows the VS Code interface with the following details:

- EXPLORER**: Shows a tree view with a node labeled **TEST** expanded, containing files: `sample.json`, `sampleTorch.json`, and `test.js`.
- TEST**: The active folder in the Explorer.
- test.js**: The active file in the code editor.
- Content of test.js:**

```
JS test.js  x

JS test.js > ⚡ default
1 import http from 'k6/http';
2 import { sleep } from "k6";
3
4 const data = JSON.parse(open("./sample.json"))
5
6 export const options = {
7   vus: 25,
8   duration: "2m",
9 };
10
11 export default function () {
12   const getRandom = (array) => {
13     const random = Math.floor(Math.random() * array.length);
14     return array[random];
15   }
16
17   const url = 'http://localhost:8501/v1/models/emotions:predict';
18
19   const res = http.post(url, JSON.stringify(getRandom(data)), {
20     headers: { 'Content-Type': 'application/json' },
21   });
22
23   sleep(1)
24 }
```

Rysunek 4.29. Skrypt testowy

W pliku *sample.json* znajdowało się 50 żądań, z różnymi fotografiami, które były w sposób losowy wybierane dla modelu oraz wysyłane do serwera na wskazany adres *url*. Symulowana została wzajemna konkurencja 25 uczestników, którzy przez 2 minuty w odstępach jednosekundowych wysyłali zapytania o klasyfikacje obrazów.

Taka sytuacja dobrze obrazuje aplikacje, które z obrazu z kamerki na bieżąco dokonują klasyfikacji obrazu. Przed wykonaniem testu dokonano konfiguracji wizualizacji metryk. Docelowa metryka to ukazanie wykresu przedstawiający opóźnienie serwera w  $\mu\text{s}$  na jedno żądanie. By otrzymać taką wartość, należy napisać odpowiednią kwerendę, która jest pokazana na rysunku 4.30.

```
increase(:tensorflow:serving:request_latency_sum[$_rate_interval]) / increase
(:tensorflow:serving:request_latency_count[$_rate_interval])
```

Rysunek 4.30. Kwerenda do Prometheus serwera

Obie metryki *request\_latency\_sum* (suma opóźnień wszystkich żądań) oraz *request\_latency\_count* (ilość wszystkich żądań) są metrykami, które zliczają tylko sumarycznie daną metrykę, żeby zdobyć konkretną wartość opóźnienia, należy obliczyć ich wzrost w danym momencie czasowym. Ten moment jest automatycznie dobierany przez Grafana za pomocą zmiennej *\$\_rate\_interval*.

Po wykonaniu testów otrzymano poniższy wykres (rysunek 4.31) oraz podsumowanie (rysunek 4.32).



Rysunek 4.31. Opóźnienie w mikrosekundach na jedno żądanie

```

execution: local
script: test.js
output: -

scenarios: (100.00%) 1 scenario, 25 max VUs, 2m30s max duration (incl. graceful stop):
* default: 25 looping VUs for 2m0s (gracefulStop: 30s)

data_received.....: 636 kB 5.3 kB/s
data_sent.....: 184 MB 1.5 MB/s
http_req_blocked.....: avg=39.87µs min=0s med=0s max=6.82ms p(90)=0s p(95)=0s
http_req_connecting.....: avg=28.56µs min=0s med=0s max=5.31ms p(90)=0s p(95)=0s
http_req_duration.....: avg=75.52ms min=7.66ms med=60.43ms max=679.42ms p(90)=133.2ms p(95)=219.57ms
{ expected_response:true }.....: avg=75.52ms min=7.66ms med=60.43ms max=679.42ms p(90)=133.2ms p(95)=219.57ms
http_req_failed.....: 0.00% 0 out of 2801
http_req_receiving.....: avg=351.04µs min=0s med=235.5µs max=4.95ms p(90)=909.4µs p(95)=996.2µs
http_req_sending.....: avg=454.31µs min=0s med=433.9µs max=12.67ms p(90)=876.4µs p(95)=1.02ms
http_req_tls_handshaking.....: avg=0s min=0s med=0s max=0s p(90)=0s p(95)=0s
http_req_waiting.....: avg=74.72ms min=6.91ms med=59.64ms max=676.78ms p(90)=132.59ms p(95)=218.62ms
http_reqs.....: 2801 23.139728/s
iteration_duration.....: avg=1.07s min=1s med=1.06s max=1.69s p(90)=1.13s p(95)=1.22s
iterations.....: 2801 23.139728/s
vus.....: 16 min=16 max=25
vus_max.....: 25 min=25 max=25

running (2m01.0s), 0/25 VUs, 2801 complete and 0 interrupted iterations
default ✓ [=====] 25 VUs 2m0s

```

Rysunek 4.32. Raport k6

Interpretując metrykę *http\_req\_waiting* można wyczytać m.in. że średnia czasu oczekiwania wynosi 74,72 ms, maksymalny czas wyniósł 676,78 ms, a dla 95% zapytań czekała około 218,62 ms. Takie typu wyniki sugerują, że czas oczekiwania był bardzo zmienny, co potwierdza również obserwacja wykresu. Posiadając te informacje, można podjąć próbę wyskalowania serwera, w celu ujednolicenia czasu oczekiwania.

Kolejny test objął serwer, w którym ustalono kolejno parametry: *tensorflow\_intra\_op\_parallelism* na 1 oraz *tensorflow\_inter\_op\_parallelism* na wartość 2.



Rysunek 4.33. Opóźnienie w mikrosekundach na jedno żądanie

```

http_req_waiting.....: avg=71.17ms min=7.51ms med=64.56ms max=1.12s p(90)=85.9ms p(95)=117.5ms

```

Rysunek 4.34. Raport k6

Raport (rysunek 4.34) oraz wykres (rysunek 4.33) wskazują na powodzenie w dokonaniu próby ujednolicenia opóźnień zapytań. Teraz każdy użytkownik oczekwał na obsłużenie zapytania w podobny krótkim czasie.

Następną modyfikacją, która może powiększyć przepustowość jak i zmniejszyć opóźnienia w predykcjach to włączenie serwis wsadowy. Docelowo model ma zbierać zapytania we wsady o wielkości dziesięciu obrazów. Ustawiono w pliku konfiguracyjnym parametr *max\_batch\_size* na wartość 10. Dokonano testu obciążeniowego. Analogicznie do przypadku opóźnienia tworzona jest kwerenda (rysunek 4.35), która wylicza średnią wielkość wsadu w danym momencie.

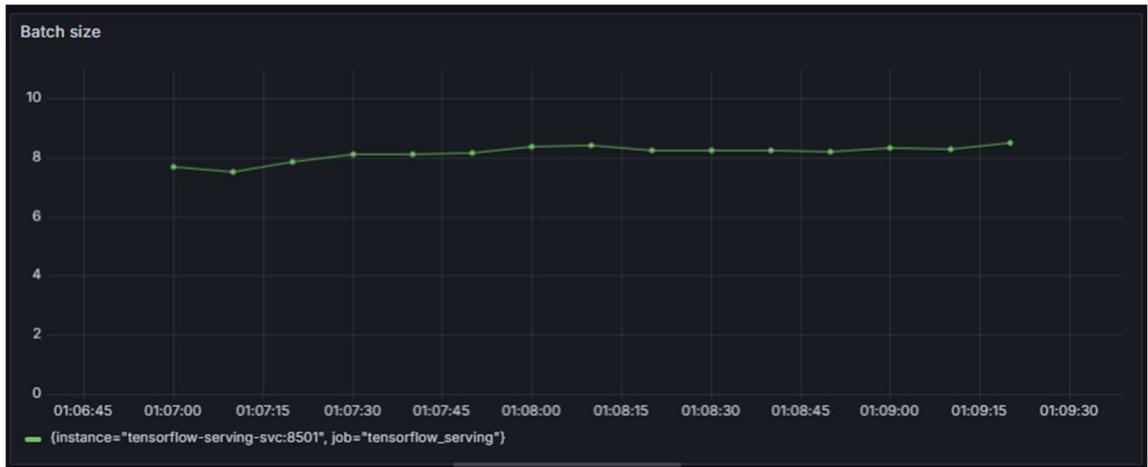
```
increase(:tensorflow:serving:batching_session:processed_batch_size_sum[$__rate_interval]) / increase(:tensorflow:serving:batching_session:processed_batch_size_count[$__rate_interval])
```

Rysunek 4.35. Kwerenda do Prometheus serwera



Rysunek 4.36. Średnia wielkość wsadu

Pomimo nadanej konfiguracji serwer tworzy średnio wsady dwu elementowe, wnioskując z obserwacji wykresu z rysunku 4.36. Jest to spowodowane czasem oczekiwania serwera na zapytania. Podczas tego testu parametr za to odpowiedzialny był ustawiony na 10000 μs. Kolejna próba obejmie dziesięciokrotnie większy czas oczekiwania.

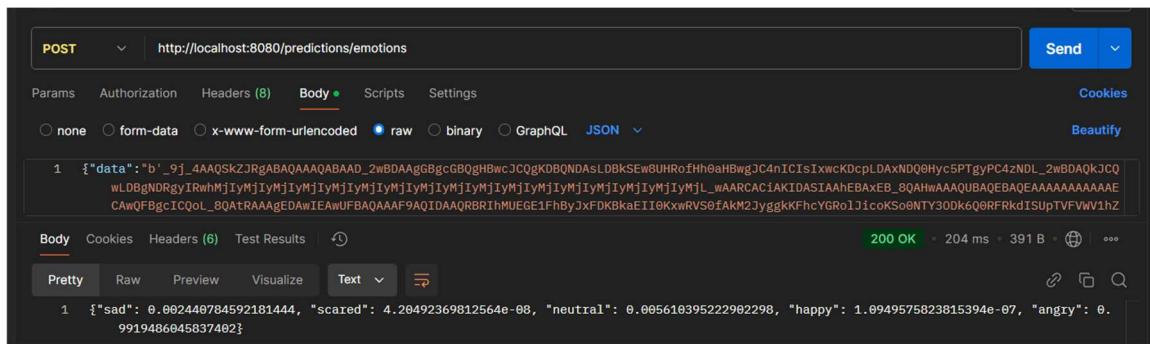


Rysunek 4.37. Średnia wielkość wsadu

Zwiększenie parametru `batch_timeout_micros` pomogło zwiększyć średnią wielkość wsadów, co potwierdza wykres z rysunku 4.37, jednakże wciąż nie została osiągnieta ich maksymalna wielkość.

Zaprezentowane możliwości TensorFlow serving oraz wykonane testy pokazały część możliwości skalowania pionowego możliwa do wykonania w tej technologii. Pomimo posiadania możliwości skalowania, staje się ta zaleta mało przydatna, gdy nie ma możliwości wglądu na wydajność serwera. Gdyby nie metryki oraz serwery umożliwiające ich analizowanie, nie byłoby możliwe skuteczne wykorzystanie w pełni możliwości serwera TensorFlow. Trzeba też pamiętać o tym, że pomimo możliwości przeprowadzenia testów obciążeniowych, wciąż nie jest możliwe w pełni dostosowanie się do ruchu panującego w sieci, gdyż testy ograniczają się do scenariuszy, natomiast metryki oferują wgląd na życie serwera w czasie rzeczywistym.

Do prezentacji pozostał TorchServe. W porównaniu do swojego konkurenta umożliwia auto skalowanie. Poprzez parametry `minWorkers` i `maxWorkers` można ustawić minimalną ilość pracowników, którzy obsługują zapytania w ramach jednego modelu (jeden pracownik zajmuje jeden wątek). Początkowo ustawiono stałą liczbę pracowników, wynoszącą. Drugi test obejmował konfiguracji, gdzie `minWorkers=1` a `maxWorkers=10`. Na rysunkach poniżej przedstawiono test działania TorchServe (rysunek 4.38), kwerendę na wartość opóźnień na jedno zapytanie użytkownika (rysunek 4.39) oraz wyniki testów dwóch konfiguracji wspomnianych wcześniej (rysunki 4.40 i 4.41).



Rysunek 4.38. Test TorchServe

```
increase(ts_inference_latency_microseconds[$__rate_interval]) / increase(ts_inference_requests_total[$__rate_interval])
```

Rysunek 4.39. Kwerenda do Prometheus serwera



Rysunek 4.40. Opóźnienie w mikrosekundach na jedno żądanie (stała liczba pracowników)



Rysunek 4.41. Opóźnienie w mikrosekundach na jedno żądanie (zakres pracowników 1-10)

Jak można zauważyc, przy stałej wartości pracowników, serwer osiąga większe opóźnienia. W przypadku ustawienia konkretnego zakresu serwer sam jest w stanie się skalować. Jest to duży komfort, gdyż takie rozwiązanie pozwala na większą elastyczność serwera, dzięki czemu można zaoszczędzić m.in. na energii elektrycznej, poprzez mniejsze nadwyrężanie jednostek obliczeniowych.

TorchServe oferuje więcej możliwości skalowania m.in. odpowiednik do serwisu wsadowego w TensorFlow serving. Z tą różnicą, że nie oferuje metryk, które kontrolują wielkość wsadów. Nie jest to jednak duży problem, gdyż w handler można zaprogramować osobiste wersje metryk. Poza tym należy również zmodyfikować pochodzący z dokumentacji plik *metrics.yaml*, w którym są zdefiniowane wszystkie metryki. Poniżej przedstawione są rysunki pokazujące potrzebne modyfikacje (4.42 i 4.43) oraz przykład odczytu tych metryk (4.44 i 4.45).

```
70 model_metrics:
71     # Dimension "Hostname" is automatically added for model metrics in the backend
72     counter:
73         - name: RequestBatchSizeSum
74             unit: count
75             dimensions: [*model_name]
76         - name: RequestBatchSizeCount
77             unit: count
78             dimensions: [*model_name]
```

Rysunek 4.42. Modyfikacja *metrics.yaml*

```
def preprocess(self, data):
    metrics = self.context.metrics

    request_batch_size_metric = metrics.get_metric(
        metric_name="RequestBatchSizeSum", metric_type=MetricTypes.COUNTER
    )

    request_batch_count_metric = metrics.get_metric(
        metric_name="RequestBatchSizeCount", metric_type=MetricTypes.COUNTER
    )

    request_batch_size_metric.add_or_update(
        value=len(data), dimension_values=[self.context.model_name]
    )

    request_batch_count_metric.add_or_update(
        value=1, dimension_values=[self.context.model_name]
    )
```

Rysunek 4.43. Modyfikacja funkcji *preprocess*

```
increase(RequestBatchSizeSum[$__rate_interval]) / increase(RequestBatchSizeCount[$__rate_interval])
```

Rysunek 4.44. Kwerenda do Prometheus serwera



Rysunek 4.45. Średnia wielkość wsadu

#### 4.5.2 Skalowanie poziome

Skalowanie poziome jest dobrym rozwiązaniem, gdy do dyspozycji są posiadane dodatkowe nieużywane jednostki obliczeniowe. Ze względu na posiadanie tylko jednego urządzenia, skalowanie odbędzie się poprzez duplikacje serwerów w ramach jednego urządzenia. W tym celu wykorzystany został Kubernetes.

Skalowanie można kontrolować za pomocą wcześniej już wspomnianego parametru *replicas*. Ustawiając parametr na wartość 2 uzyskano dwa działające serwery TorchServe (rysunek 4.46).

NAME	READY	STATUS	RESTARTS	AGE
grafana-6b7696bdd7-ht64w	1/1	Running	6 (16h ago)	5d17h
prometheus-65655d754-lpnxm	1/1	Running	0	18s
torchserve-545c8fd59d-dtqw7	1/1	Running	0	3m18s
torchserve-545c8fd59d-r4mlt	1/1	Running	0	3m20s

Rysunek 4.46. Duplikaty TorchServe

Nadano również typ serwisu TorchServe na *LoadBalancer*. Domyślnie Kubernetes ustawia serwis jako *ClusterIP*. Oznacza to, że adres jest tylko widoczny w zasięgu klastra. Jedyny sposób wystawienia tego adresu poza klaster to użycie komendy *kubectl port-forward*, który był używany przy skalowaniu pionowym. Do kiedy istniała jedna instancja serwera, był to dobry sposób na wykonywanie testów. Teraz do dyspozycji są dwie instancje, które mają siebie wzajemnie odciążać w obowiązkach, dlatego zmieniono

typ serwisu na *LoadBalancer* (*rysunek 4.47*), który jest używany najczęściej do udostępniania usługi. Poprzednie rozwiązywanie tego by nie umożliwiło, ponieważ korzystałby wtedy jedynie z jednego serwera.

```
apiVersion: v1
kind: Service
metadata:
| name: torchserve-svc
spec:
| selector:
| | app: torchserve
ports:
- name: inference
  protocol: TCP
  port: 8080
  targetPort: 8080
- name: metrics
  protocol: TCP
  port: 8082
  targetPort: 8082
type: LoadBalancer
```

*Rysunek 4.47. Modyfikacja serwisu TorchServer*

Dodatkowo zmodyfikowano konfiguracje samego Prometheus serwera oraz TorchServer. Wcześniej metryki były pobierane poprzez nawigowanie do serwisu, a nie bezpośrednio do podów. Kubernetes oraz Prometheus umożliwiają inne podejście. W pliku konfigurującym Prometheusa w parametrze *scrape\_configs* można ustawić specjalne parametry, które działają tylko w środowisku klastrowym Kubernetes.

```

prometheus.yml: |
  global:
    | scrape_interval: 10s

  scrape_configs:
    - job_name: 'torchserve'
      kubernetes_sd_configs:
        - role: pod
      relabel_configs:
        - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
          action: keep
          regex: true
        - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
          action: replace
          target_label: __metrics_path__
          regex: (.+)
        - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_port]
          action: replace
          regex: ([^:]+)(?::\d+)?;(\d+)
          replacement: $1:$2
          target_label: __address__

```

Rysunek 4.48. Modyfikacja Prometheus serwera

Analizując powyższe ustawienia widoczne na rysunku 4.48, parametr *kubernetes\_sd\_configs* z wartością *role: pod* mówi serwerowi, żeby poszukiwał metryki w podach. W sekcji *relabel\_configs* są konfigurowane wartości, które nakierowują Prometheusa na adres z metrykami. Są tu trzy filtry zastosowane. Pierwszy sprawdza zmienną

*\_\_meta\_kubernetes\_pod\_annotation\_prometheus\_io\_scrape* czy zwraca wartość *true*.

Drugi wyczytuje zmienną

*\_\_meta\_kubernetes\_pod\_annotation\_prometheus\_io\_path* ścieżkę, w której znajdują się metryki. Ostatni z zmiennej *\_\_meta\_kubernetes\_pod\_annotation\_prometheus\_io\_port* uzyskiwany jest port, za pomocą którego modyfikuję się docelowy adres poda, w taki sposób, aby serwer próbował odczytać metryki pod właściwym portem. Zmienne *\_\_meta* są odczytywane z konfiguracji podów, co za tym idzie, trzeba je własnoręcznie ustawić. Nadanie wartościami tych zmiennych można zobaczyć na rysunku 4.49.

```

replicas: 2
selector:
  matchLabels:
    app: torchserve
template:
  metadata:
    annotations:
      prometheus.io/scrape: "true"
      prometheus.io/path: "/metrics"
      prometheus.io/port: "8082"

```

Rysunek 4.49. Definicja zmiennych w Kubernetes

W ten o to sposób są odczytywane wszystkie metryki z podów serwerów TorchServe. Wykonując test obciążeniowy, można wyświetlić dwa wykresy przedstawiające opóźnienia na jedno zapytanie (rysunek 4.50).



Rysunek 4.50. Opóźnienie w mikrosekundach na jedno żądanie

Zapisane kwerendy muszą być bardziej sprecyzowane, dlatego dopisuje się filtr, który odwołuje się do konkretnego poda. Przykładowy filtr szukający metryki pod konkretnym adresem IP można zobaczyć na rys 4.51.

```

increase(ts_inference_latency_microseconds{instance="10.244.0.192:8082"}[$__rate_interval]) / increase
(ts_inference_requests_total{instance="10.244.0.192:8082"}[$__rate_interval])

```

Rysunek 4.51. Kwerenda do Prometheus serwera

Pomimo wielu adresów, dzięki komponentowi service wszystkie zapytania są obsługiwane pod jednym adresem <http://localhost:8080/predictions/emotions>. Kubernetes odpowiednio przechwytuje zapytania i sam rozdziela je pomiędzy podami.

Takie typu skalowanie zmusza do ustawiania stałej ilości podów. Kubernetes w celu zaimplementowania większej elastyczności w zarządzaniu zasobami, oferuje komponent *HorizontalPodAutoscaler*, który sam zarządza ilością podów konkretnych usług. Zmieniono ilość replik na jeden pod oraz dodano do pliku ustawienia, które są pokazane na rysunku 4.52.

```

47  apiVersion: autoscaling/v2
48  kind: HorizontalPodAutoscaler
49  metadata:
50    name: torchserve-hpa
51  spec:
52    scaleTargetRef:
53      apiVersion: apps/v1
54      kind: Deployment
55      name: torchserve
56    minReplicas: 1
57    maxReplicas: 2
58    metrics:
59      - type: Resource
60        resource:
61          name: cpu
62        target:
63          type: Utilization
64          averageUtilization: 50

```

Rysunek 4.52. Konfiguracja HPA

W pliku przekazano informacje, żeby utrzymał średnią utylizację procesora w okolicach 50%, poprzez samoistne tworzenie podów, gdzie ich ilość ma się mieścić w zakresie 1-2. Gdy utylizacja będzie wynosić powyżej 50%, zostanie utworzony dodatkowy pod. W przypadku odwrotnym, gdzie utylizacja wynosi mniej niż 50%, dodatkowy pod zostanie usunięty. Po konfiguracji komponentu *HorizontalPodAutoscaler* można wyświetlić jego stan, który został pokazany na rysunku 4.53.

(base) adam@Adam:~\$ kubectl get hpa torchserve-hpa	NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
	torchserve-hpa	Deployment/torchserve	cpu: <unknown>/50%	1	2	1	12m

Rysunek 4.53. HPA status

Zauważono problem, że jest nieznane zużycie procesora. Jest to wynikiem tego, że Kubernetes domyślnie nie zbiera metryk tego rodzaju. Trzeba do klastra dodać pod z serwerem metryk, który plik konfiguracyjny można pobrać z repozytorium Kubernetes. Dodatkowo należy w komponencie TorchServer należy dopisać jaką ilość cpu powinna

zostać mu przekazana. W tym przypadku ustawiono zakres pomiędzy połową rdzenia (500 milicores) a jednym rdzeniem (rysunek 4.54).

```
resources:  
  requests:  
    cpu: "500m"  
  limits:  
    cpu: "1"
```

Rysunek 4.54. ograniczenia CPU

Serwer metryk ma też problem z protokołem TLS, który zapewnia większe bezpieczeństwo klastra. TLS nie jest potrzebny w celach testowych, dlatego otrzymany plik konfiguracyjny został odpowiednio zmodyfikowany tak, aby wyłączyć to zabezpieczenie (dodano do zmiennych parametr `--kubelet-insecure-tls`, który jest widoczny na rysunku 4.55).

```
containers:  
  - args:  
    - --cert-dir=/tmp  
    - --secure-port=10250  
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname  
    - --kubelet-use-node-status-port  
    - --metric-resolution=15s  
    - --kubelet-insecure-tls
```

Rysunek 4.55. wyłączenie TLS

Po nadanych poprawkach stan komponentu auto skalującego wygląda zgodnie z rysunkiem 4.56.

```
(base) adam@Adam:~$ kubectl get hpa torchserve-hpa  
NAME      REFERENCE      TARGETS      MINPODS   MAXPODS   REPLICAS   AGE  
torchserve-hpa  Deployment/torchserve  cpu: 0%/50%   1         2          1          27m
```

Rysunek 4.56. HPA status

Działanie auto skalowania zostało zweryfikowane poprzez test obciążeniowy, którego skutki są widoczne na rysunku 4.57.

```
NAME      REFERENCE      TARGETS      MINPODS   MAXPODS   REPLICAS   AGE  
torchserve-hpa  Deployment/torchserve  cpu: 169%/50%  1         2          2          50m
```

Rysunek 4.57. HPA status

Konfiguracja została przeprowadzona prawidłowo. Kubernetes teraz sam zarządza ilością podów.

## 4.6 Podsumowanie

W części praktycznej udało się spełnić założenia projektu. Utworzono sieci konwolucyjne w TensorFlow oraz PyTorch. Zaprogramowano odpowiednie przekształcanie danych w celu odpowiedniego wczytania obrazów oraz uzyskania czytelnej odpowiedzi serwera. Wszystkie modele zostały właściwie wyeksportowane. Pomyślnie zrealizowano konfiguracje klastra Kubernetes oraz przeprowadzono skalowanie pionowe i poziome serwerów uczenia maszynowego. Wszystkie konfiguracje były testowane testami obciążeniowymi oraz analizowane poprzez metryki.

Przy skalowaniu pionowym w TensorFlow serving domyślne ustawienia nie były skuteczne, gdyż mogły zabierać zbyt dużą ilość zasobów, nadmiernie obciążając procesor. Środowisko Minikube wymaga minimalnie dwóch rdzeni procesora. Wszystkie elementy działające w tle obciążały procesor, który posiadał tylko 4 rdzenie. TensorFlow serving domyślnie przypisuje sobie cały dostępny procesor, dlatego serwer działał w sposób niestabilny. Odpowiednia konfiguracja odejmująca obciążenie procesora ustabilizowała działanie serwera. Można to zauważać na rysunkach 4.31 i 4.33. Pokazany wykres w 4.31 w teście dla ustawień domyślnych zachowuje się chaotycznie, użytkownicy czasem czekali dłużej lub krócej na odpowiedź, gdzie zakres czasowy wynosił około od 16000  $\mu$ s do 39000  $\mu$ s. Po zastosowaniu odpowiedniego skalowania, wykres 4.33 wskazuje na ujednolicony czas w okolicach 20000  $\mu$ s.

Skalowanie TorchServe jest łatwiejsze ze względu na auto skalowanie. Umożliwia swobodne ustawianie zakresów, gdzie pomimo nadania zbyt dużej maksymalnej ilości liczby pracowników, serwer wciąż był w stanie właściwie dostosować się, o ile wartość minimalna pracowników była właściwa dla procesora. Można to zaobserwować, porównując wykresy z rysunków 4.40 oraz 4.41. Dla stałej ilości pracowników równej 10 występowały o wiele większe opóźnienia (maksymalne opóźnienie: około 360000  $\mu$ s), zwłaszcza przy początkowym czasie pracy serwera, w porównaniu z serwerem, którego zakres pracowników wynosił od 1 do 10 (maksymalne opóźnienie: około 65000  $\mu$ s).

Konfiguracja serwisu wsadowego okazała się trudna w zastosowaniu, przy małej mocy obliczeniowej. Gdyż serwer nie posiadał wystarczających zasobów, aby skutecznie uporządkować zapytania we wsadzie. Docelowy rozmiar wynosił 10 próbek. Wykonano dwie próby. Pierwsza dla 10000  $\mu$ s, druga 100000  $\mu$ s. Wykresy z rysunków 4.36 i 4.37 pokazują, że w żadnym wypadku nie otrzymano wsadu wynoszącego 10 próbek.

Rozwiązywanie serwisu wsadowego może okazać się dobrym rozwiązaniem tylko w przypadku posiadania dużej mocy obliczeniowej. Nawet jeśli zastosowana konfiguracja działa prawidłowo, należy sprawdzić opóźnienia, czy dane rozwiązanie pomogło podwyższyć wydajność serwera.

Analizując wykresy z rysunku 4.50, można zauważyc, że Kubernetes przy udostępnianiu dwóch replik serwera TorchServe, bardzo równomiernie rozkładał obciążenie tych serwerów. Wykresy trwania opóźnień na jedno zapytanie prawie się wzajemnie nachodzą. Jest to bardzo duża zaleta Kubernetes, ponieważ w pełni używała dostępne zasoby, nie marnując żadnych z nich oraz nie przeciągała niepotrzebnie pody.

Przedstawione metody oraz sposób ich realizacji pokazały metody skalowania systemów, przedstawiając ich wady, zalety oraz aspekty, który muszą być kontrolowane. Pomimo że system był konfigurowany w małym środowisku, pokazane rozwiązania mogą być stosowane w praktyce w wielkich firmach, które dysponują znacznie większą ilością i jakością zasobów.

## 5. Wnioski

W części teoretycznej pracy przedstawiono koncept uczenia maszynowego oraz omówiono różne metody wdrażania systemów uczenia maszynowego poruszając również tematy związane ze skalowaniem, takie jak konteneryzacja oraz systemy orkiestracyjne. Dodatkowo wytlumaczono architektury i zasady działania narzędzi stosowanych do tworzenia systemu ML.

Część praktyczna objęła aspekt tworzenia sieci neuronowych, ich eksportu oraz wdrażania do systemu ML za pomocą narzędzi przedstawionych w części teoretycznej. Dokonano skalowania pionowego jak i poziomego a wszelkie konfiguracje były testowane testami obciążeniowymi, a systemy były monitorowane w czasie rzeczywistym. Dzięki tym przykładom można poznać podglądowy obraz procesu wdrażania systemów ML oraz zrozumieć jak duże ma znaczenie analiza metryk systemu.

Pokazano w praktyce tylko jedną metodę wdrażania ze względu na dużą czasochłonność w projektowaniu architektury oraz implementacji. Wybrany do ukazania został streaming API ze względu na duże możliwości w przeprowadzeniu procesu skalowania. Nie było możliwe zbudowanie większego systemu ML ze względu na brak odpowiednich zasobów.

Cel pracy został zrealizowany. Pokazano kluczowe koncepcje związane z wdrażaniem systemów uczenia maszynowego razem z realizacją skalowania. Praca ta może służyć jako źródło wiedzy w tematyce wdrażania systemów ML oraz w tworzeniu skalowalnych usług, które można zastosować również poza dziedziną sztucznej inteligencji.

## Literatura

- [1] Jason Bell: Machine Learning: Hands-On for Developers and Technical Professionals, John Wiley & Sons, Inc. 10475 Crosspoint Boulevard Indianapolis, IN 46256, rok 2015.
- [2] Dipanjan Sarkar, Raghav Bali, Tushar Sharma: Practical Machine Learning with Python, APRESS MEDIA, LLC, 233 Spring Street, 6th Floor, New York, rok 2018.
- [3] Andreas C. Müller i Sarah Guido: Introduction to Machine Learning with Python, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, rok 2016.
- [4] Hafsa Habehh i Suril Gohel: Machine Learning in Healthcare, Current Genomics, s. 291-300, DOI: 10.2174/138920292266210705124359, rok 2021.
- [5] Erik Brynjolfsson, Danielle Li, Lindsey R. Raymond: GENERATIVE AI AT WORK, NBER WORKING PAPER SERIES, no. 31161, DOI: 10.3386/w31161, rok 2023.
- [6] Sabina-Cristiana Necula i Vasile-Daniel Pavaloaia: AI-Driven Recommendations: A Systematic Review of the State of the Art in E-Commerce, Applied Sciences, DOI: <https://doi.org/10.3390/app13095531>, rok 2023.
- [7] Charu C. Aggarwal: Neural Networks and Deep Learning, Springer Nature Switzerland AG, Gewerbestrasse 11, 6330 Cham, Switzerland, rok 2018.
- [8] Zixuan Ma, Jiaao He, Jiezhong Qiu, Huanqi Cao, Yuanwei Wang, Zhenbo Sun, Liyan Zheng, Haojie Wang, Shizhi Tang, Tianyu Zheng, Junyang Lin, Guanyu Feng, Zeqiang Huang, Jie Gao, Aohan Zeng, Jianwei Zhang, Runxin Zhong, Tianhui Shi, Sha Liu, Weimin Zheng, Jie Tang, Hongxia Yang, Xin Liu, Jidong Zhai, Wenguang Chen: BaGuaLu: Targeting Brain Scale Pretrained Models with over 37 Million Cores, PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, DOI: <https://doi.org/10.1145/3503221.3508417>, rok 2022.
- [9] Sieci konwolucyjne, według firmy IBM: <https://www.ibm.com/think/topics/convolutional-neural-networks>, dostęp: 17.01.2025.
- [10] Chip Huyen: Designing Machine Learning Systems, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, rok 2022.
- [11] Ethem Alpaydın: Introduction to Machine Learning, The MIT Press Cambridge, Massachusetts London, Anglia, rok 2014.
- [12] Wsparcie Keras: <https://github.com/keras-team/keras/issues/20095>, dostęp 23.12.2024.
- [13] Emmanuel Ameisen: Building Machine Learning Powered Applications, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, rok 2020.
- [14] Dokumentacja TensorFlow: <https://www.tensorflow.org>, dostęp: 23.12.2024.
- [15] Dokumentacja PyTorch: <https://pytorch.org>, dostęp: 23.12.2024.
- [16] Dokumentacja ONNX Runtime <https://onnxruntime.ai/docs/>, dostęp: 18.01.2025.

- [17] Nathan F. Saraiva de Sousa, Danny A. Lachos Perez, Raphael V. Rosa, Mateus A. S. Santos, Christian Esteve Rothenberg: Network Service Orchestration: A Survey, Computer Communications, s. 69-94, DOI: <https://doi.org/10.1016/j.comcom.2019.04.008>, rok 2019.
- [18] Charalampos Rotsos, Daniel King, Arsham Farshad, Jamie Bird, Lyndon Fawcett, Nektarios Georgalas, Matthias Gunkel, Kohei Shiomoto, Aijun Wang, Andreas Mauthe, Nicholas Race, David Hutchison, Network service orchestration standardization: A technology survey, Computer Standards & Interfaces, s. 203-215, DOI: <https://doi.org/10.1016/j.csi.2016.12.006>, rok 2017.
- [19] Definicja konteneryzacji, według firmy IBM: <https://www.ibm.com/think/topics/containerization>, dostęp 23.12.2024.
- [20] Różnice pomiędzy kontenerami a maszynami wirtualnymi według firmy Amazon Web Services, <https://aws.amazon.com/compare/the-difference-between-containers-and-virtual-machines/>, dostęp: 23.12.2024.
- [21] Ahmed Hussein Ali i Mahmood Zaki Abdullah: A Survey on Vertical and Horizontal Scaling Platforms for Big Data Analytics, INTERNATIONAL JOURNAL OF INTEGRATED ENGINEERING vol. 11 no. 6, s. 138-150, rok 2019.
- [22] Dokumentacja Docker: <https://docs.docker.com>, dostęp: 23.12.2024.
- [23] Dokumentacja Kubernetes: <https://kubernetes.io/docs/>, dostęp: 23.12.2024.
- [24] Dokumentacja Karpenter: <https://karpenter.sh/docs/>, dostęp: 23.12.2024.
- [25] Zbiór fotografii: <https://www.kaggle.com/datasets/thienkhonghoc/affectnet>, dostęp 18.01.2025.

## **Summary**

The purpose of present thesis was to describe methods used to create machine learning systems. Today, people have access to different machine learning models through the Internet. They can be used by computer or smartphone. To see how this is possible and understand more about machine learning systems, the thesis is divided into two parts.

The first part describes the definition of machine learning and tells more about convolutional neural networks. It also discusses the characteristics of good machine learning systems and explains different methods for implementing these systems. The purpose of this section was also to present the concept of scalability, introducing definitions of orchestration systems and containers.

The second part shows whole process of building a scalable machine learning system to recognize emotions from photographs of people's faces. The system was developed using orchestration system Kubernetes, TensorFlow serving, TorchServe and Docker. All these technologies are well explained in the first part of the thesis. The building begins with creating convolutional neural networks, and exporting them as servable systems to Docker images. After that, they were implemented in system and scaled, by using different configurations which each of them where tested. The performance of systems was monitored using Prometheus server and Grafana.