AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI

KATEDRA INFORMATYKI

Systemy mobilne

Rozproszone (warstwowe) algorytmy uczenia maszynowego w Mobilnej Chmurze Obliczeniowej

Autorzy: *Monika Darosz, Jakub Jungiewicz, Robert Wcisło*
Kierunek studiów: *Informatyka*
Opiekun projektu: *dr hab. inż. Nawrocki Piotr*

Kraków, 2020

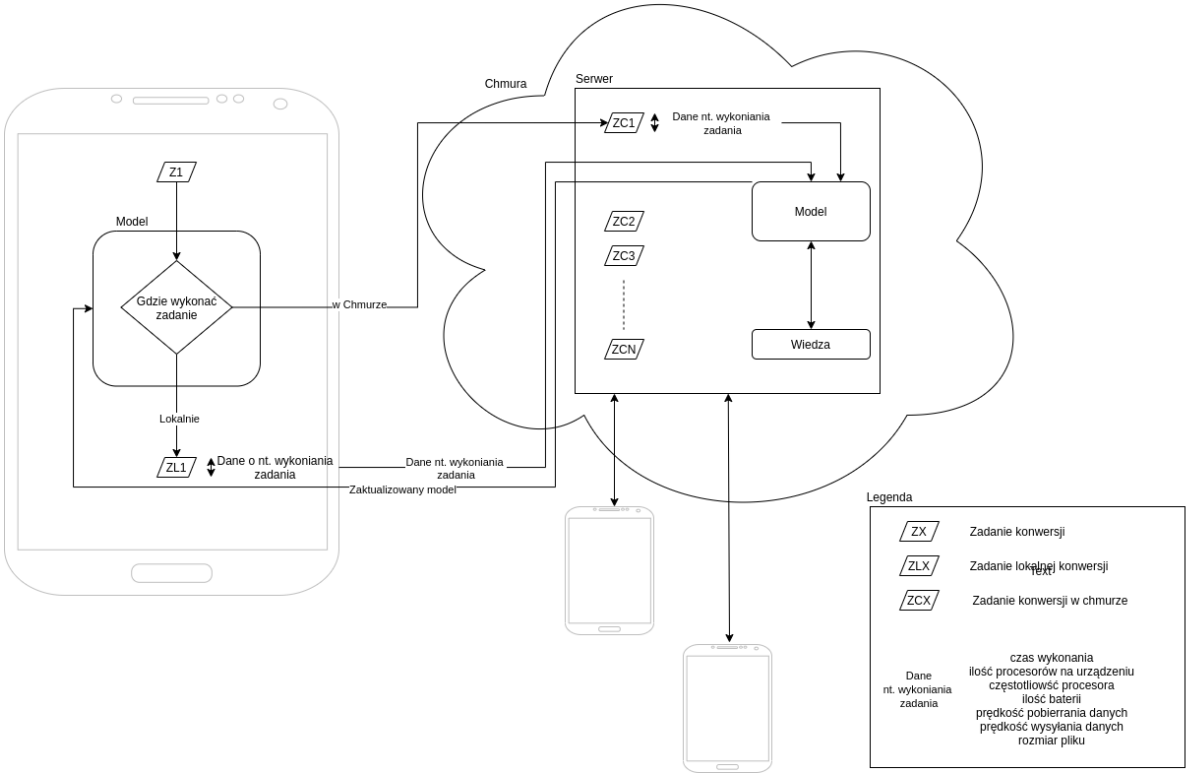
Spis treści

1. Wstęp i cele projektu

Celem projektu było stworzenie aplikacji korzystającej z metod uczenia maszynowego do podejmowania decyzji o wykonaniu zadania (czy wykonać zadanie lokalnie na urządzeniu czy też przesłać je do chmury). Sam model uczenia jest globalny dla całej populacji, co ma na celu przyspieszenie procesu uczenia i zwiększenie uniwersalności oraz skuteczności. Podobne rozwiązania można znaleźć w literaturze czego przykładem są [1, 2, 3].

1.1. Architektura rozwiązania

Zdecydowaliśmy się na podejście, w którym model, który będzie podejmował decyzję o tym gdzie wykonać zadanie będzie znajdował się w chmurze, urządzenia będą okresowo synchronizować swoje modele z tym z chmury.



Rys. 1.1. Architektura systemu

2. Model

Problemem, który mieliśmy rozwiązać było podjęcie decyzji, czy pewne obliczenia (u nas konwersja pliku .png na .jpg) opłaca się wykonać lokalnie, czy w chmurze. W tym celu powstały dwie sieci neuronowe przewidujące czas wykonania obliczeń odpowiednio:

1. zdalnie - na komputerze, i w chmurze (Google Cloud)
2. lokalnie, czyli na urządzeniu mobilnym

Dodatkowo powstał trzeci model - klasyfikator agregujący obydwie sieci. Model ten wyznacza przewidywany czas obliczeń dla obydwu podejść, porównuje i w odpowiedzi wskazuje pożądany sposób wykonania.

2.1. Architektura sieci

Wszystkie sieci zostały stworzone w bibliotece PyTorch. Zdecydowaliśmy się na tę bibliotekę z kilku powodów:

- możliwość tworzenia sieci, których można używać na Androidzie
- Modułowa architektura pozwalająca w łatwy sposób agregować poszczególne warstwy w sieć oraz sieci w bardziej złożone modele
- wcześniejsze doświadczenie z tą biblioteką

Obydwie sieci posiadają identyczną architekturę - różnią się jedynie wymiarowością danych wejściowych:

- moduł przewidujący czas wykonania zdalnego bierze pod uwagę jedynie wielkość pliku - nie mają tutaj znaczenia parametry opisujące aktualny stan urządzenia
- moduł przewidujący czas wykonania na urządzeniu mobilnym bierze pod uwagę szereg czynników, takich jak:
  - Wielkość pliku (długość boku w pikselach)
  - Liczba rdzeni urządzenia.

- Częstotliwość rdzeni wyrażona w Hz.
- Ilość pamięci operacyjnej wyrażona w kB.
- Ilość baterii wyrażona w procentach.
- Typ sieci.
- Prędkość pobierania i wysyłania wyrażona w kb/s.

Obydwie sieci, to niewielkie, trójwarstwowe sieci Feed Forward z aktywacjami ReLU, jak przedstawiono w 2.1.

```
nn.Sequential(
            nn.Linear(input_size, 16),
            nn.ReLU(),
            nn.Linear(16, 32),
            nn.ReLU(),
            nn.Linear(32, 1)
        )
```

Listing 2.1. Architektura sieci przewidującej czas wykonania

Klasyfikator przedstawiony jest na 2.2

```
class Mobilenet(nn.Module):
    def __init__(self, input_size: int, lr: float = 0.0000001, momentum: float =
        0.9, save_interval: int = 100):

        super().__init__()
        self.inner_net_local = nn.Sequential(
            nn.Linear(input_size, 16),
            nn.ReLU(),
            nn.Linear(16, 32),
            nn.ReLU(),
            nn.Linear(32, 1)
        )
        self.inner_net_cloud = nn.Sequential(nn.Linear(input_size, 16),
                                             nn.ReLU(),
                                             nn.Linear(16, 32),
                                             nn.ReLU(),
                                             nn.Linear(32, 1))

        if path.exists(GLOBAL_MODEL_PATH_LOCAL):
            self.inner_net_local.load_state_dict(torch.load(GLOBAL_MODEL_PATH_LOCAL)
                )

        if path.exists(GLOBAL_MODEL_PATH_CLOUD):
            self.inner_net_cloud.load_state_dict(torch.load(GLOBAL_MODEL_PATH_CLOUD)
                )

```

```
24          if not path.exists(CLOUD_RESULTS_PATH):
25              init_csv(CLOUD_RESULTS_PATH)
26
27          if not path.exists(LOCAL_RESULTS_PATH):
28              init_csv(LOCAL_RESULTS_PATH)
29
30          self.criterion_l = nn.MSELoss()
31          self.criterion_c = nn.MSELoss()
32          self.optimizer_l = optim.SGD(self.inner_net_local.parameters(), lr=lr,
                momentum=momentum)
33          self.optimizer_c = optim.SGD(self.inner_net_cloud.parameters(), lr=lr,
                momentum=momentum)
34          self.losses = []
35          self.state_interval = save_interval
36          self.counter = 1
37          self.cloud_results = []
38          self.local_results = []
39
40      def forward(self, input: Tensor, verbose=False):
41          cloud_cost = self.inner_net_cloud(input)
42          local_cost = self.inner_net_local(input)
43
44          if verbose:
45              print(f'local = {local_cost}, cloud = {cloud_cost}')
46          # 1. -cloud, 0 - local
47
48          return (cloud_cost < local_cost).double()
```

Listing 2.2. Architektura finalnego klasyfikatora i przebieg samej klasyfikacji

2.2. Sieć mobilna

PyTorch jest biblioteką stworzoną dla języka Python. Z pomocą narzędzia TorchScript oraz biblioteki *org.pytorch:pytorch_android:1.4.0* można serializować model stworzony w Pythonie, a następnie wczytać go w javie, stworzyć tensor wejściowy i dokonywać inferencji na wytrenowanym modelu. Tworzenie sieci w Pytorchu jest wygodne między innymi dzięki narzędziu *AutoGrad*, które automatycznie śledzi wszystkie operacje przeprowadzane na tensorach i jest w stanie wyliczyć ich pochodne, przez co wsteczna propagacja błędu staje się bardzo prosta w implemenacji. Niestety, model wczytany w Javie na Androidzie może działać jedynie w trybie inferencji i nie pozwala na dowolne aktualizacje parametrów, przez co konieczne jest wysyłanie wyników do chmury i cykliczna sychronizacja parametrów.

2.3. Przebieg uczenia

Postawiony problem okazał się być problemem w zasadzie liniowym. Wpływ pozostałych parametrów okazał się być niemal niezauważalny, a model bardzo szybko uczył się liniowej zależności pomiędzy wielkością obrazu, a czasem wykonania. Warto tutaj odnotować, że zależność ta jest w zasadzie kwadratowa - wynika to z faktu, że przez wielkość obrazu rozumiemy rozmiar boku kwadratowego obrazu liczonego w pikselach. Model mógł się nauczyć liniowej zależności z dwóch powodów - wciąż ma zbyt małą liczbę neuronów, lub nie testowaliśmy jeszcze modelu na wystarczająco dużych obrazach. Parametry uczenia:

- learning_rate = 0.0000001; Warto odnotować, że dopiero dla tak małego kroku model zaczął się czegokolwiek uczyć.

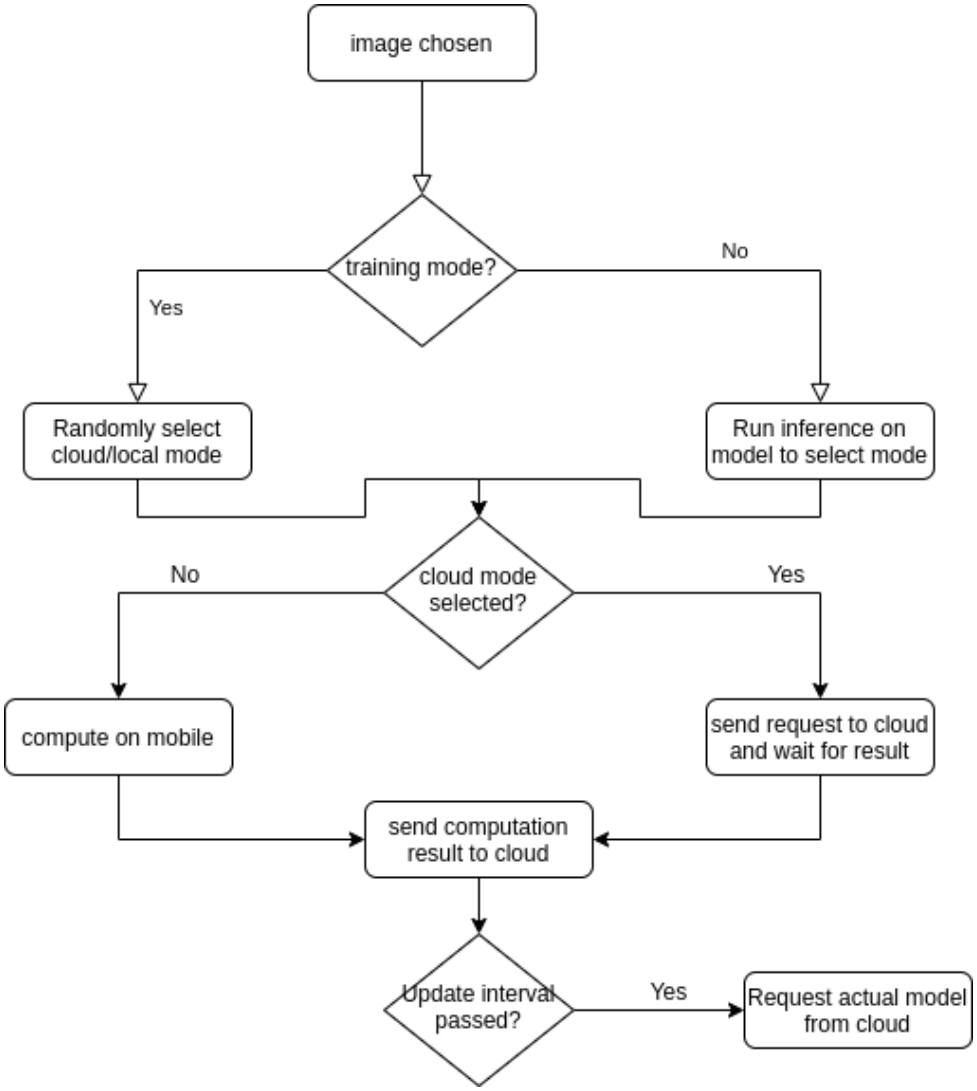
- momentum = 0.9

Samo uczenie realizowane jest w chmurze. W początkowej fazie uczenia (do pierwszej synchronizacji) urządzenia mobilne podejmują losową decyzję (chcemy uniknąć sytuacji, w której niewytrenowana sieć zwraca zawszę jedną opcję). Każdorazowo urządzenie liczy czas wykonania. Po konwersji wynik, tzn stan urządzenia sprzed podjęcia decyzji oraz czas wykonania wysyłane są na serwer, który dokonuje aktualizacji modelu.

- jeśli urządzenie wybrało wykonanie zdalne aktualizowany jest jedynie model przewidujący czas wykonania zdalnego

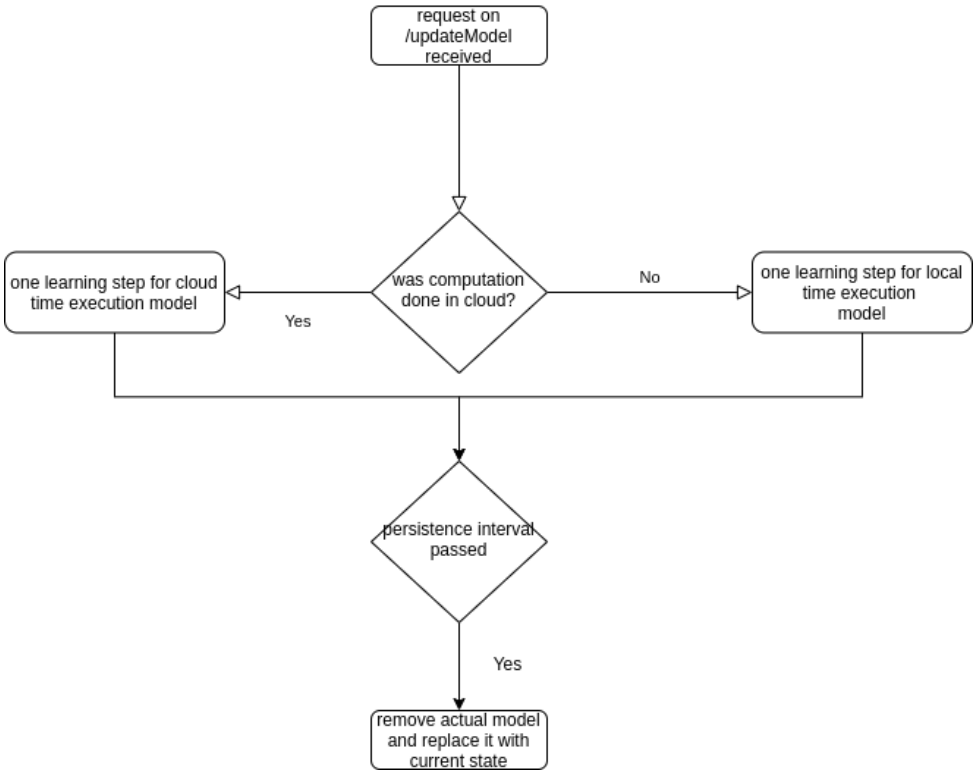
- jeśli urządzenie wybrało wykonanie lokalne aktualizowany jest jedynie model przewidujący czas wykonania na urządzeniu mobilnym

Urządzenie mobilne co pewną liczbę kroków – N aktualizuje swój model na ten znajdujący się aktualnie w chmurze. Opisany mechanizm od strony urządzenia mobilnego możemy zobaczyć na diagramie 2.1



Rys. 2.1. Diagram przedstawiający podejmowanie decyzji przez urządzenie mobilne

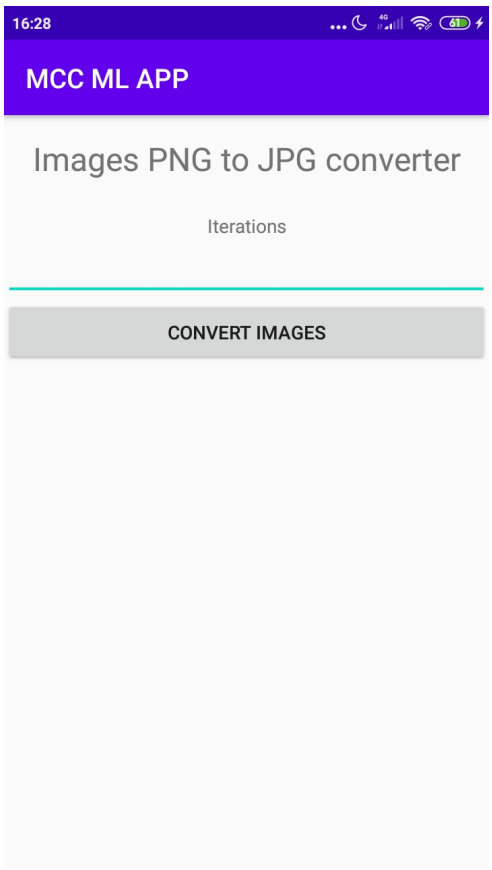
Serwer uczy model każdorazowo po otrzymaniu zapytania HTTP z urządzenia mobilnego. W zależności od tego, czy problem z zapytania wykonany był na urządzeniu mobilnym czy w chmurze wykonywany jest jeden krok uczenia *online* dla odpowiedniego modelu. Co N kroków uczenia aktualny model jest serializowany - staje się wtedy dostępny do pobrania dla urządzenia mobilnego. Serializacja jest niezależna od liczby otrzymywanych zapytań o akutalny model. W praktyce zapisujemy model co kilkadziesiąt kroków uczenia, więc każde urządzenie może mieć niemal aktualny stan, a z drugiej strony serwer nie traci czasu na każdorazową serializację na żądanie aktualizacji modelu przez urządzenie mobilne. Przedstawiony mechanizm opisuje diagram 2.2



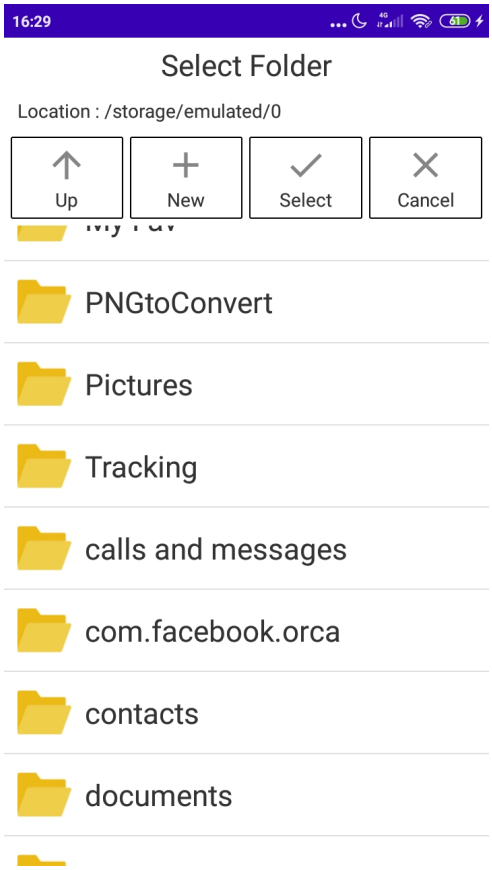
Rys. 2.2. Diagram przedstawiający sposób uczenia modeli po otrzymaniu przez serwer zapytania HTTP

3. Aplikacja mobilna

Aplikacja mobilna to natywna aplikacja na Androida, została zaimplementowana w języku JAVA w środowisku programistycznym Android Studio. Aplikacja posiada jeden widok - Rys. 3.1, w którym znajdują się dwie kontrolki. EditText służy do wprowadzenia ilości iteracji jaka ma zostać przeprowadzona, a naciśniecie przycisku powoduje otworzenie DirectoryPickera - Rys. 3.2 służącego do wybrania folderu z plikami typu PNG.



Rys. 3.1. Główny widok aplikacji mobilnej



Rys. 3.2. DirectoryPicker aplikacji mobilnej

W klasie głownej czynności (MainActivity) została nadpisana metoda onActicvityResult, która w momencie wybrania folderu przy użyciu DirectoryPickera przechwytuje wynik i uruchamia asynchroniczne zadanie w tle - Listing 3.1.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == FOLDER_PICKER_CODE && resultCode == Activity.RESULT_OK) {

        String folderLocation = data.getExtras().getString("data");
        Log.i("folderLocation", folderLocation);
        File directory = new File(folderLocation);
        File[] files = directory.listFiles();
        int iterations = Integer.parseInt(iterationsEditText.getText().toString());
        new ConvertPngsTask(this, files, iterations, folderLocation).execute();

    }
    super.onActivityResult(requestCode, resultCode, data);
}
```

Listing 3.1. Tworzenie asynchronicznego zadania w tle

Następnie w klasie ConvertPngsTask - Listing 3.2 - dla podanej ilości iteracji wykonujemy n razy konwersje na wszystkich plikach typu PNG w wybranym przez użytkownika folderze. Kompresja przebiega następująco:

```
@Override
protected Void doInBackground(Void... voids) {
    for(int iteration = 0; iteration<iterations; iteration++) {
        for(File file: files) {
            if (file.getName().toLowerCase().endsWith(".png")) {
                ResourcesReader resourcesReader = new ResourcesReader(this.context);
                Resources resources = resourcesReader.readResources();
                String filePath = file.getAbsolutePath();
               ImageModel model = mainActivity.getModel();
               ImageModel.Execution mode = model.classify(filePath, resources);
                Log.i("convertImage", file.getName());
                Log.i("place", mode.toString());
                switch (mode) {
                    case LOCAL:
                        long startTime = System.nanoTime();
                        boolean converted = new Converter().convertImage(filePath,
                            directoryPath, iteration);
                        if (converted) {
                            long endTime = System.nanoTime();
                            long timeElapsedMillis = (endTime - startTime) / 1
                                _000_000;
                            InferenceResult result = new InferenceResult(
                                    TensorUtils.prepareInput(filePath, resources),
                                    (float) timeElapsedMillis,
                                    ImageModel.Execution.LOCAL.ordinal()
                            );
                            new RequestSender().updateModel(result);
                        }
                        break;
                    case CLOUD:
                        new RequestSender().uploadFile(filePath, directoryPath,
                            iteration, resources);
                }
            }
        }
    }
    return null;
}
```

Listing 3.2. Asynchroniczne zadanie konwersji

1. Czytanie obecnych zasobów przy pomocy ResourceReadera - Listing 3.3.
2. Wybranie uruchomienia lokalnego lub zdalnego:

(a) W trybie treningu wybieramy losowo.

(b) W trybie finalnym model decyduje o wyborze zadania i uczy się ónline".

3. Wykonanie zadania kompresji przy użyciu Convertera - Listing 3.5 - lub wysłanie zapytania na serwer przy użyciu RequestSendera - Listing 3.6.

ResourceReader - Listing 3.3 jest odpowiedzialny za czytanie zasobów - Listing 3.4 z urządzenia mobilnego, a są to:

- Liczba rdzeni.
- Częstotliwość rdzeni wyrażona w Hz.
- Ilość pamięci wyrażona w kB.
- Ilość baterii wyrażona w procentach.
- Typ sieci.
- Prędkość pobierania i wysyłania wyrażona w kb/s.

```
1  public Resources readResources() {
2
3      long memory = readTotalRam();
4      int processorCores = getProcessorCores();
5      double processorFrequency = getProcessorFrequency();
6      int battery = getBatteryLevel();
7      String network = getNetwork();
8      boolean hasWifi = hasWifi();
9      int downloadSpeed = getDownloadSpeed();
10     int uploadSpeed = getUploadSpeed();
11
12     return new Resources(processorCores, processorFrequency, memory, battery,
           hasWifi, downloadSpeed, uploadSpeed, network);
13 }
```

Listing 3.3. Narzędzie do czytania zasobów

```
public Resources(int processorCores, double processorFrequency, long memory, int
    battery, boolean hasWifi, int downloadSpeed, int uploadSpeed, String network) {
    this.processorCores = processorCores;
    this.processorFrequency = processorFrequency;
    this.memory = memory;
    this.battery = battery;
    this.hasWifi = hasWifi;
    this.downloadSpeed = downloadSpeed;
    this.uploadSpeed = uploadSpeed;
    this.network = network;
}
```

Listing 3.4. Zasoby

Converter - Listing 3.5 - jest odpowiedzialny za wykonanie lokalnej konwersji pliku PNG do pliku JPEG, a następnie za zapisanie go do folderu wskazanego przez użytkownika.

```
public class Converter {
    public boolean convertImage(String filePath, String directoryPath, int iteration
        ){
        boolean success;
        try {
            File file = new File(filePath);
            String fileName = FilenameUtils.removeExtension(file.getName());
            Bitmap bmp = BitmapFactory.decodeFile(filePath);
            File convertedImage = new File(directoryPath + "/" + fileName + "
                _converted.jpg");
            convertedImage.createNewFile();
            FileOutputStream outStream = new FileOutputStream(convertedImage);
            success = bmp.compress(Bitmap.CompressFormat.PNG, 100, outStream);

            outStream.flush();
            outStream.close();
        } catch (IOException e) {
            e.printStackTrace();
            success = false;
        }
        return success;
    }
}
```

Listing 3.5. Konwerter

RequestSender - Listing 3.6 - jest odpowiedzialny za synchroniczne wysłanie zdjęcia w celu przeprowadzeni konwersji zdalnej oraz zapisanie otrzymanego wyniku do folderu wskazanego przez użytkownika.

```
1  public void uploadFile(final String filePath, final String directoryPath, final int
       iteration, Resources resources) {
2      Retrofit retrofit = new Retrofit.Builder()
3              .baseUrl(URL_STRING)
4              .build();
5
6      UploadService service = retrofit.create(UploadService.class);
7      File file = new File(filePath);
8      MultipartBody.Part filePart = MultipartBody.Part.createFormData("file", file.
           getName(), RequestBody.create(MediaType.parse("image/*"), file));
9      final long startTime = System.nanoTime();
10     final Resources finalResources = resources;
11
12     Call<ResponseBody> callSync = service.calc(filePart);
13     try
14     {
15         Response<ResponseBody> response = callSync.execute();
16         if(response.isSuccessful()) {
17             File fileToSave = new File(filePath);
18             String fileName = FilenameUtils.removeExtension(fileToSave.getName());
19             File convertedImage = new File(directoryPath + "/" + fileName + "
                   _converted.jpg");
20             FileOutputStream fileOutputStream = new FileOutputStream(convertedImage)
                   ;
21             IOUtils.write(response.body().bytes(), fileOutputStream);
22
23             long endTime = System.nanoTime();
24
25             long timeElapsedMillis = (endTime - startTime) / 1_000_000;
26             InferenceResult result = new InferenceResult(
27                     TensorUtils.prepareInput(filePath, finalResources),
28                     (float) timeElapsedMillis,
29                     ImageModel.Execution.CLOUD.ordinal()
30             );
31             updateModel(result);
32         }
33         else {
34             System.out.println(response.errorBody());
35         }
36     }
37     catch (Exception ex)
38     {
39         ex.printStackTrace();
40     }
41 }
```

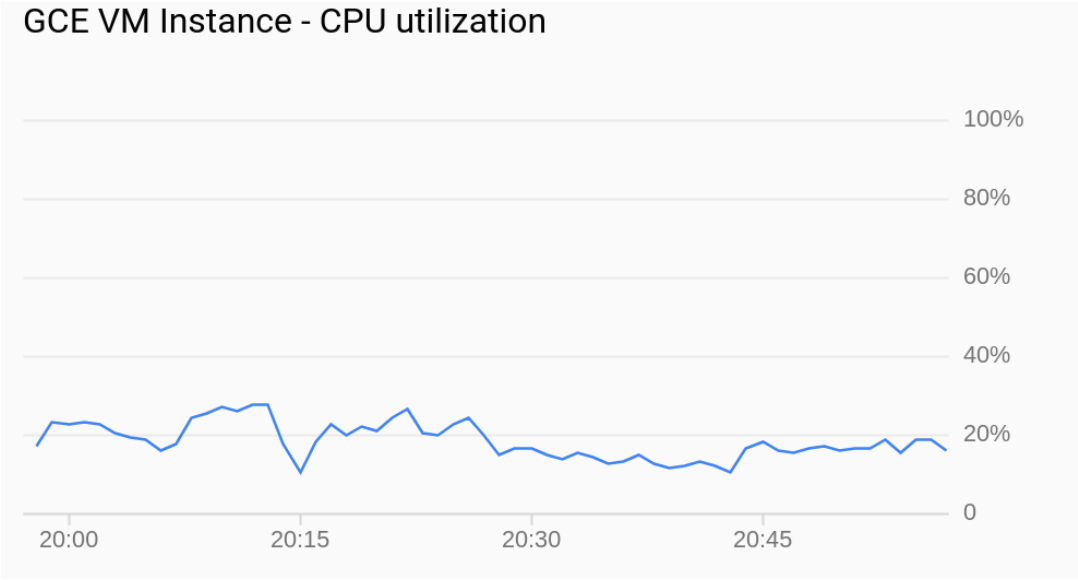
Listing 3.6. RequestSender

4. Obliczenia w chmurze

Istotne informacje odnośnie serwera jak i jego lokacji oraz maszyny na której działał:

1. Do wykonania zadania wybraliśmy platformę Google Cloud. Wybór był podytkowany niekomercyjną dostępnością platformy (tzw. Free Tier opiewa tam na zawrotną kwotę 300$, co daje duże pole do popisu).

2. Jako platforma obliczeniowa służyła nam jedna maszyna wirtualna o parametrach znaczących 4GB pamięci RAM oraz 1 CPU. Uznaliśmy taką konfigurację za wystarczająca po sprawdzeniu wykresu zużycia procesora na maszynie wirtualnej 4.1 podczas gdy równolegle 3 urządzenia wysyłały do serwera zapytania o konwersje/aktualizacje modelu.



Rys. 4.1. Wykres zużycia CPU w czasie dla maszyny wirtualnej podczas gdy 3 urządzenia równolegle wysyłają zapytania do serwera

3. Maszyna wirtualna podczas pierwszych testów, była fizycznie umieszczona w regionie Stanów Zjednoczonych, co miało niestety znaczący wpływ na latencje przy wysyłaniu zadań. Obecnie maszyna jest umieszczona na terenie Europy zachodniej, co daje mniejszą latencje, ale wciąż taką która ma znaczący wpływ przy wysyłaniu czegokolwiek na serwer.

4. Serwer został zaimplementowany w języku Python za pomocą frameworka Flask, po to aby możliwe było korzystanie z bibliotek ML w chmurze.

5. Aby zachować zgodność co do technologii, w której wykonywane jest zadanie konwersji, serwer dla każdego zapytania o konwersje wykonywał zadanie uruchamiając plik *.jar* w nowej instancji JVM.

6. Sam serwer udostępniał trzy endpointy:

   (a) */updateModel* - służacy do aktualizowania parametrów modelu

   (b) */calc* - służacy do inwokacji obliczeń na w chmurze. Przyjmował plik w formacie .png a zwracał .jpg

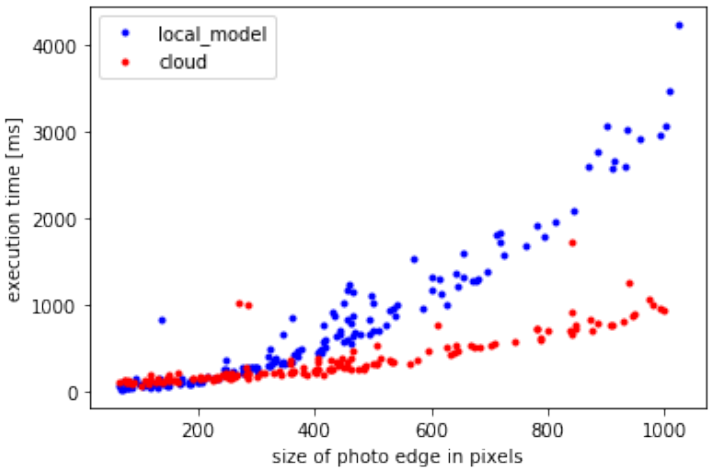
   (c) */fetchModel* - służący do pobrania aktualnej wersji modelu

5. Wyniki

Przeprowadziliśmy eksperymenty w dwóch środowiskach: w środowisku chmurowym oraz w sieci lokalnej. Zadanie, które należało wykonać, to konwersja z formatu png na jpg wykonana 10-krotnie w celu zmniejszenia wpływu przesyłania obrazów przez sieć w przypadku wykonania w chmurze.

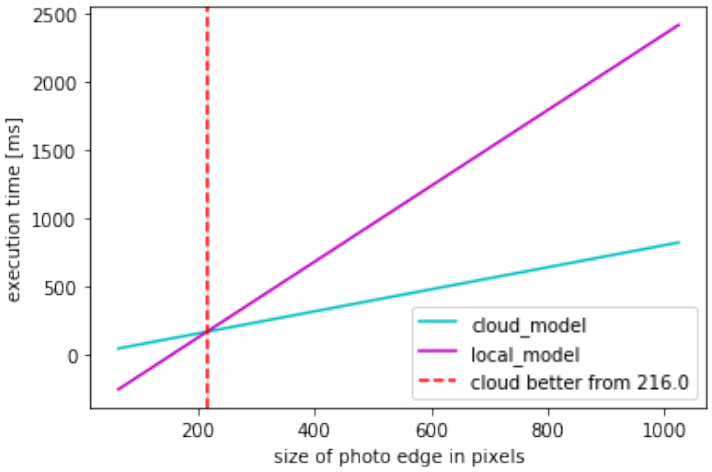
5.1. Sieć lokalna

Do tej pory testowaliśmy obrazy wielkości od 64x64 piksele do 1024x1024. Rozkład czasu wykonania w zależności od wielkości obrazu przedstawia Rys.5.1



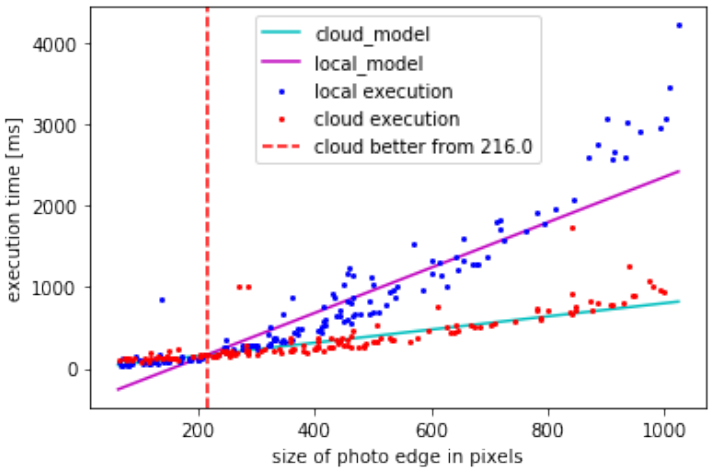
Rys. 5.1. Wykres pokazujący czas wykonania lokalnie i na serwerze w zależności od wielkości pliku

Dla takiego zbioru danych modele wykonują następujące predykcje, jak przedstawiono na Rys.5.2



Rys. 5.2. Wykres pokazujący predykcje czasu wykonania dla chmury i urządzenia mobilnego

Oznacza to, że nasz klasyfikator zapytany w tym momencie o dowolny obraz każe wykonać obliczenia na urządzeniu mobilnym dla obrazów mniejszych niż 216x16, a w chmurze dla większych. jakość predykcji możemy ocenić na podstawie wykresu 5.3.



Rys. 5.3. Wykres pokazujący predykcje czasu wykonania dla chmury i urządzenia mobilnego z nałożonymi faktycznymi pomiarami w środowisku lokalnym

Dla tego zbioru składającego się ze 150 obrazów wielkości od 64x64 do 1024x1024 pikseli wyniki przedstawione w tabeli 5.1 pokazują, że w lokalnej sieci najlepiej działa model, jednak tylko nieco lepiej od liczenia wszystkiego w chmurze - przynajmniej dla tak postawionego problemu.

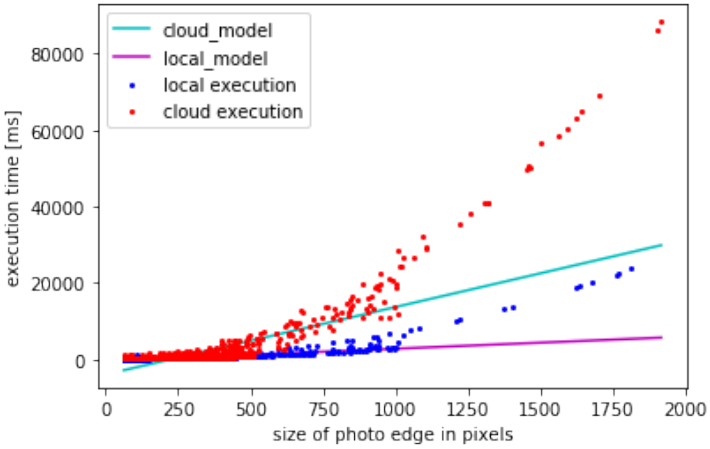
5.2. Chmura

W chmurze wykonaliśmy te same obliczenia, co na własnych komputerach. Wyniki były jednak mocno niezadowalające. Wykonaliśmy w sumie 2699 pomiarów z czego 1440 na 3 różnych urządzeniach

Tabela 5.1. Tabela czasu wykonania obliczeń dla 10-krotnej konwersji 150 obrazów w milisekundach

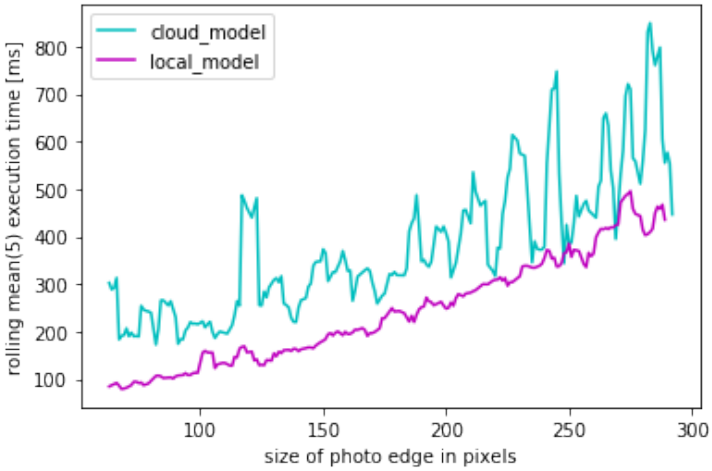
117501	wszystko lokalnie
54171	wszystko w chmurze
53471	model podejmuje decyzje

mobilnych a 1259 w chmurze opisanej w rozdziale 4. Rozkład wyników w zależności od wielkości obrazu oraz predykcje modeli prezentuje Rysunek 5.4



Rys. 5.4. Wykres pokazujący predykcje czasu wykonania dla chmury i urządzenia mobilnego z nałożonymi faktycznymi pomiarami w Google Cloud Engine

Wyraźnie widać, że czas wykonania w chmurze jest na ogół wyższy, niż na urządzeniu mobilnym,co przeczy naszej początkowej hipotezie. Nawet dla niewielkich obrazów da się zaobserwować to samo zjawisko. Dla lepszej czytelności pokazujemy je jako średnią kroczącą z uwzględnieniem 4 poprzednich wyników na Rys.5.5.



Rys. 5.5. Wykres pokazujący średnią kroczącą czasu wykonania dla chmury i urządzenia mobilnego dla obrazów mniejszych niż 300x300 pikseli

Tabelaryczne zestawienie wyników nie ma tutaj sensu, gdyż w zasadzie zawsze opłacało się bardziej skorzystać z obliczeń na urządzeniu mobilnym. Przyczyny takiego stanu rzeczy są następujące:

- Czas przesłania obrazu na chmurę jest znaczący
- Problem, który wybraliśmy, nawet po modyfikacji miał, jak się okazało zbyt duży narzut komunikacyjny

Zestawienie wyników z sieci lokalnej i z chmury pozwala wyciągnąć następujące wnioski

1. Uczenie w chmurze może być wydajne
2. Należy starannie dobierać problem, który chcemy wykonywać zdalnie. Powinniśmy albo mieć pewność, że będzie on wykonany blisko urządzenia mobilnego, albo ilość danych do przesłania jest znikoma
3. Nasz problem nadaje się do zdalnego wykonania tylko w przypadku, gdy zdalne urządzenie znajduje się w tej samej sieci, co mobilne.

Spis rysunków

Listings

Bibliografia

[1] Piotr Nawrocki, Bartłomiej Śnieżyński i Hubert Słojewski. „Adaptable mobile cloud computing environment with code transfer based on machine learning". W: *Pervasive and Mobile Computing* 57 (maj 2019), s. 49–63. DOI: *10.1016/j.pmcj.2019.05.001*.

[2] Piotr Nawrocki i Bartłomiej Śnieżyński. „Autonomous Context-Based Service Optimization in Mobile Cloud Computing". W: *Journal of Grid Computing* 15 (lip. 2017), 343–356. DOI: *10.1007/s10723-017-9406-2*.

[3] Piotr Nawrocki i Bartłomiej Śnieżyński. „Adaptive Service Management in Mobile Cloud Computing by Means of Supervised and Reinforcement Learning". W: *Journal of Network and Systems Management* (lut. 2017). DOI: *10.1007/s10922-017-9405-4*.