**Projekt Dyplomowy**
**na kierunku Informatyka Stosowana**

**Politechnika Warszawska**
**Wydział Elektryczny**

# *Porównanie wydajności języków programowania w przetwarzaniu obrazów z użyciem głębokich sieci neuronowych*

Michał Banaszczak, michal.banaszczak.stud@pw.edu.pl
Semestr VI studiów I stopnia, Informatyka Stosowana, Inżynieria Danych i Multimedia
Wersja i data wykonania sprawozdania: v1, 14.06.2023
Prowadzący: dr inż. Witold Czajewski

# Spis treści

# 1 Wprowadzenie

Sieci neuronowe i uczenie głębokie stają się coraz popularniejszymi tematami, zwłaszcza w obliczu niedawnych osiągnięć w tej dziedzinie. Zjawisku pomaga fakt, że dzięki postępującemu rozwojowi technicznemu wielkie modele, których parametry liczy się w setkach milionów, można trenować na sprzęcie klasy konsumenckiej. Współcześnie rozwijanych jest wiele frameworków służących do tworzenia sieci neuronowych opartych na różnych technologiach i językach.

Różnica w wydajności języków kompilowanych i interpretowanych jest znana i dobrze udokumentowana, natomiast frameworki uczenia głębokiego często udostępniają frontendy dla różnych języków programowania połączone ze wspólnym backendem, który dodatkowo korzysta z przyspieszenia sprzętowego GPU. Celem tego projektu jest sprawdzenie wydajności frameworków do uczenia głębokiego opartych na różnych językach programowania do przetwarzania obrazów. Testy prowadzone będą zarówno na prostych przykładowych architekturach sieci, jak i na współczesnych, zaawansowanych modelach.

## 2 Możliwe rozwiązania

Python jest jednym z ważniejszych języków dla data science oraz uczenia maszynowego i posiada obszerny ekosystem bibliotek z tym związanych. Jednym z najpopularniejszych frameworków do uczenia głębokiego jest PyTorch, otwartoźródłowy framework pierwotnie stworzony przez Meta AI, a teraz prowadzony przez Linux Foundation. Jego pierwsza produkcyjna wersja, wydana w 2018r., została zintegrowana z backendem Caffe [13], niewspieranego już frameworku, który również posiadał API dla Pythona i C++. PyTorch jest technologią o bardzo imperatywnym podejściu, które pozwala na łatwiejsze zrozumienie zachodzących procesów dla początkujących oraz na większą dowolność dla specjalistów, w niektórych jednak przypadkach może to prowadzić do mniejszej wydajności względem rozwiązań konkurencyjnych. Takim właśnie rozwiązaniem, zorientowanym przede wszystkim na szybkość i wydajność, jest TensorFlow stworzony przez zespół Google Brain. W 2019r. została wydana jego 2. wersja zintegrowana z biblioteką Keras [14], która zawiera wiele udogodnień do uczenia maszynowego i głębokiego. MATLAB, jako środowisko do obliczeń naukowych i inżynierskich posiada wiele toolboxów i innych narzędzi do tworzenia i testowania sieci neuronowych. Wszystkie wymienione wyżej frameworki wspierają oczywiście technologię CUDA i dzięki temu oferują przyspieszenie sprzętowe obliczeń z wykorzystaniem kart graficznych nVidii.

C++ posiada dużo bibliotek do sieci neuronowych (OpenNN, FANN, MiniDNN), natomiast mało która wspiera CUDA, co dyskwalifikuje je z tego projektu. PyTorch i TensorFlow mają również frontend dla języka C++ - pierwszy nazywa się LibTorch, a drugi funkcjonuje pod tą samą nazwą. Miałem jednak wątpliwości czy wybór któregoś z nich miały tak naprawdę sens – byłoby to w końcu testowanie tej samej technologii dwa razy.

PyTorch, TensorFlow i MATLAB posiadają zbiory predefiniowanych modeli na podstawie różnych prac naukowych. Zbiory te nie do końca się jednak pokrywają, więc do testów zostanie wybranych parę modeli dostępnych w każdej technologii, jak i parę prostszych modeli napisanych przeze mnie.

## 2.1 Wybór rozwiązania

PyTorch, TensorFlow i MATLAB tak jak pisałem wyżej są popularnymi i często wykorzystywanymi technologiami, naturalnie więc zostaną poddane testom wydajnościowym w tym projekcie.

Jeśli chodzi o C++, zdecydowałem się przetestować czystą bibliotekę cuDNN. Ma ona znacznie okrojoną funkcjonalność względem wyżej wymienionych frameworków, natomiast klasyfikacja zbioru MNIST jest wykorzystywana jako przykład do weryfikacji poprawnej instalacji CUDA i cuDNN. Zacząłem się przez to zastanawiać, czy w jakichś zastosowaniach miałoby sens pisanie wszystkich funkcjonalności w cuDNN od zera dla dodatkowej wydajności względem gotowych frameworków, nawet jeśli nie jest to popularne rozwiązanie. Skorzystałem w tym celu z projektu autorstwa Jacka Hana [5], który zmodyfikowałem do własnych potrzeb.

Wybór technologii podyktował wybór modeli, jakie zostaną użyte w projekcie. Są to:

1. FCNet – prosta sieć MLP na podstawie pierwszego rozdziału książki „Neural Networks and Deep Learning" autorstwa Michaela Nielsona [11]
2. SimpleConvNet – prosta sieć CNN, jaką można znaleźć w dowolnym poradniku do sieci neuronowych dla początkujących [6]
3. ResNet-50 [1]
4. DenseNet-121 [2]
5. MobileNet v2 [4] – model stworzony z myślą o urządzeniach mobilnych, stosunkowo mała liczba parametrów w odpowiedniej architekturze miała zapewnić wysoką celność klasyfikacji przy niskim czasie inferencji
6. ConvNeXt [3] – najnowszy z dostępnych predefiniowanych modeli

Wykorzystane zostaną dwa zbiory danych – MNIST [9] oraz CIFAR-10 [8]. W projektach badawczych powinno się już odchodzić od pierwszego z nich na rzecz bardziej skomplikowanych, nawet zamienników 1:1 jak np. Fashion-MNIST [16], który został stworzony przez Zalando Research, natomiast zdałem sobie z tego sprawę już w trakcie rozwoju projektu i bałem się, że zmiana zbioru może przysporzyć dodatkowych kłopotów, na które nie miałem czasu. ImageNet, inny popularny zbiór do testowania współczesnych modeli, jest zbiorem obszernym (pod względem ilości klas, próbek, jak i rozmiaru obrazów). Wątpię by moja karta graficzna była w stanie go udźwignąć, dlatego nie został wykorzystany w tym projekcie.

# 3 Założenia projektu

Projekt polega na utworzeniu benchmarków dla różnych technologii, na podstawie których porównana zostanie ich wydajność. Nie zakłada się, że będą one uruchamiane przez osoby trzecie – produktem projektu są rezultaty testów i ich analiza.

Etapy realizacji projektu:

1. Instalacja Ubuntu (podyktowana brakiem wsparcia dla Windowsa przez niektóre technologie), wymaganych sterowników i bibliotek oraz innego oprogramowania
2. Konfiguracja technologii, przygotowanie modeli i zbiorów danych
   2.1. PyTorch
   2.2. TensorFlow
   2.3. MATLAB
   2.4. cuDNN
3. Przygotowanie i przeprowadzenie benchmarków
4. Analiza wyników

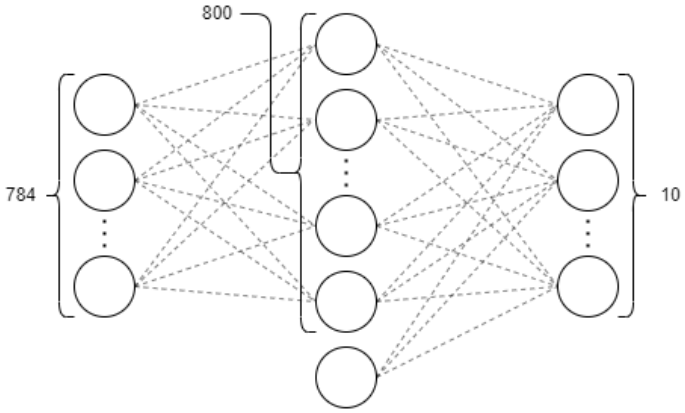
# 4 Realizacja projektu

## *4.1 Szczegółowy opis realizacji*

Pierwszym etapem projektu było przygotowanie maszyny. W związku z zakończeniem wsparcia dla Windowsa przez TensorFlow począwszy od wersji 2.11 zdecydowałem postawić maszynę linuxową. Jak się okazało, nie zajęło mi to wcale mniej czasu niż być może konteneryzacja TensorFlowa na Windowsie, gdyż ostatecznie przeinstalowywałem Ubuntu sześciokrotnie. Powodem były pomyłki takie jak:

- przy próbie instalacji MATLABa nadałem prawa do zapisu złemu katalogowi, efektywnie odbierając sobie prawo do korzystania z polecenia sudo,
- próba zmiany wersji Pythona z domyślnej na nowszą spowodowała odinstalowanie dziesiątek pakietów zależnych od tej wersji, w tym wszystkich odpowiedzialnych za wyświetlanie interfejsu graficznego,
- instalacja złej wersji cuDNN przez co TensorFlow nie chciał się nawet uruchomić, nie pomagało przeinstalowanie kompatybilnych wersji bibliotek.

Kiedy uporałem się z instalacją Ubuntu przeszedłem do implementacji pierwszego modelu w PyTorchu. Mój pierwotny pomysł na FullyConnectedNet zakładał odwzorowanie którejś z przykładowych architektur sieci wymienionych na oficjalnej stronie MNIST. Problem jest z nimi taki, że są to wiekowe rozwiązania, które nawet nie korzystają z optymalizacji takich jak SGD. Spędziłem tak więc za dużo czasu na wymyśleniu jak w PyTorchu aktualizować parametry modelu bez wykorzystania optymalizatora. Tutaj udało mi się to osiągnąć dzięki imperatywnemu podejściu biblioteki, ale coś takiego nie byłoby możliwe w TensorFlowie ani w MATLABie. W porę na szczęście dotarła do mnie głupota tego podejścia. W końcu z jakiegoś powodu nie da się domyślnie aktualizować parametrów bez wykorzystania optymalizatora – może to prowadzić do przeuczenia sieci i jest nieoptymalne, spowalnia proces uczenia. Ostatecznie więc FullyConnectedNet bazowany jest na modelu opisywanym w pierwszym rozdziale książki „Neural Networks and Deep Learning” autorstwa Michaela Nielsona. W mojej implementacji można oczywiście zmieniać ilość warstw ukrytych jak i zawartych w nich neuronów, jednak jedna warstwa ukryta w zupełności wystarcza do osiągnięcia ponaddziewięćdziesięcioprocentowej skuteczności w klasyfikacji.




Rysunek 1. Diagram sieci FullyConnectedNet

```
class FullyConnectedNet(nn.Module):

  def __init__(self, layers=[784, 800, 10]):
    super(FullyConnectedNet, self).__init__()
    self.layers = nn.ModuleList([nn.Linear(a, b, dtype=torch.float64)
                                 for a, b in zip(layers[:-1], layers[1:])])

  def forward(self, x):
    for layer in self.layers[:-1]:
      x = F.relu(layer(x))
    x = self.layers[-1](x)
    return F.log_softmax(x, dim=1)
```

Kod 1. Definicja FullyConnectedNet w PyTorchu

Następnie przeszedłem do implementacji tego modelu w TensorFlow. Próbowałem jak najbliżej odwzorować funkcjonalność PyTorcha również korzystając z subclassingu przy definicji modelu, a i funkcja `get_mnist_loaders` wczytująca dane dla obu technologii korzystała z tego samego skryptu dekodującego oryginalne pliki binarne. Poniżej definicja tego modelu.

```
class FullyConnectedNet(tf.keras.Model):

  def __init__(self, hidden_layers=[800], num_classes=10):
    super().__init__()
    self.hidden_layers = tf.keras.Sequential([
      tf.keras.layers.Dense(n, activation='relu')
      for n in hidden_layers
    ])
    self.output_layer = tf.keras.layers.Dense(num_classes, activation='softmax')

  def call(self, x):
    x = self.hidden_layers(x)
    return self.output_layer(x)
```

Kod 2. Definicja FullyConnectedNet w TensorFlow

W trakcie tej implementacji doszedłem do dwóch wniosków:

- nie ma sensu tworzenie wspólnych funkcji do wczytywania danych dla wszystkich technologii jak zrobiłem ze zbiorem MNIST, gdyż każdy z frameworków ma swoje typy, domyśle struktury i sposoby przetwarzania danych, więc zamiast odejmować sobie roboty to jej potrójnie dokładam,
- nie mają sensu próby definicji modeli w podobny sposób – oba pythonowe frameworki mają zupełnie inne i wzajemnie niekompatybilne podejście do definicji sieci i warstw.

Od tego momentu przy implementacji następnych modeli przestałem ograniczać się do jakiegoś „złotego środka”, a zacząłem kierować się filozofią technologii, w której akurat pracowałem. Znacząco przyspieszyło to pracę i zmniejszyło liczbę błędów wynikającą właśnie z prób robienia czegoś na około.

Nie mam przez to jednak na myśli, że implementacja predefiniowanych modeli obyła się bez żadnych problemów. Sugerując się dokumentacją [10][12] uznałem, że minimalnym rozmiarem tensora wejściowego jest (224x224x3), co byłoby znacznym ograniczeniem przy pracy z jedną kartą graficzną - 4 to największy rozmiar batcha jaki mieścił się w VRAMie mojej karty, co wydłużyło znacząco czas uczenia, a też może prowadzić do przeuczenia. Zdecydowanie za dużo czasu spędziłem na próbach różnych ustawień CUDA i różnych podejść do przetwarzania wstępnego, aż znalazłem w dokumentacji Keras zapis o tym, że można zmienić rozmiar tensora wejściowego, jeśli podłączy się do modelu własny klasyfikator [7]. Podobne podejście (zmiana różnych warstw predefiniowanego modelu) zastosowałem dla MATLABa i PyTorcha i dla nich rozwiązało to problem.

```
new_inputs = imageInputLayer([32 32 3], ...
    Name = "new_input_1", ...
    Normalization = "zscore" ...
);

new_logits = fullyConnectedLayer(10, ...
    Name = "NewLogits", ...
    BiasLearnRateFactor = 10, ...
    WeightLearnRateFactor = 10 ...
);

model = mobilenetv2(Weights="none");
model = replaceLayer(model, "Logits", new_logits);
model = replaceLayer(model, "input_1", new_inputs);
```

Kod 3. Dostosowanie modelu w MATLABie

```
model = mobilenet_v2()
model.classifier[1] = nn.Linear(1280, num_classes, bias=True)
```

Kod 4. Dostosowanie modelu w PyTorchu

```
def classifier_overlay(inputs):
  x = tf.keras.layers.GlobalAveragePooling2D()(inputs)
  x = tf.keras.layers.Flatten()(x)
  x = tf.keras.layers.Dense(1024, activation="relu")(x)
  x = tf.keras.layers.Dense(512, activation="relu")(x)
  x = tf.keras.layers.Dense(10, activation="softmax", name="classification")(x)
  return x

def combine_model(inputs, predef_model, classifier, image_size=32):

  predef_model_materialised = predef_model(
    input_shape=(image_size, image_size, 3),
    include_top=False,
    weights=None
  )

  resize = tf.keras.layers.Resizing(image_size, image_size)(inputs)

  feature_extractor = predef_model_materialised(resize)
  classification_output = classifier(feature_extractor)
  return tf.keras.Model(inputs=inputs, outputs=classification_output)

model = combine_model(
  tf.keras.layers.Input(shape=(32,32,3)),
  tf.keras.applications.MobileNetV2,
  classifier_overlay
)
```

Kod 5. Dołączanie własnego klasyfikatora do modelu w TensorFlow

TensorFlow dalej miał jednak problem z uczeniem paru modeli w tym samym notebooku, należało zmienić parametr dostępu do pamięci jak poniżej.

```
gpus = tf.config.experimental.list_physical_devices('GPU')
  if gpus:
    for gpu in gpus:
      tf.config.experimental.set_memory_growth(gpu, True)
```

Kod 6. Zmiania ustawień TensorFlow

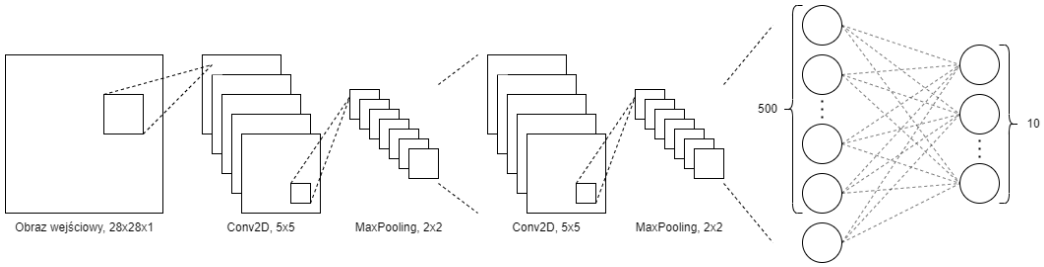
Rozwiązanie to okazało się działać jedynie w Jupyter Notebooku. Przy uruchomieniu normalnego skryptu TensorFlow nie zwalnia pamięci VRAM na karcie graficznej nawet po usunięciu wszystkich referencji do modelu jak i po restarcie karty. TensorFlow generalnie nigdy nie zwalnia pamięci w obrębie jednego procesu i jest to zachowanie oczekiwane. Jedynym więc wyjściem było spawnowanie nowych podprocesów do uczenia każdego kolejnego modelu.

Implementacja benchmarków w MATLABie obyła się bez większych problemów, poza tym, że dokumentacja bardzo niejasno stanowi o tym w jaki sposób do sieci mogą być ładowane dane trzymane w całości w pamięci. ImageDatastore mianowicie można utworzyć wyłącznie dla obrazów przechowywanych w systemie plików, które są wczytywane dopiero w miarę przetwarzania. Na początku próbowałem zastosować workaround z AugumentedImageDatastore, skalując obrazy do aktualnego rozmiaru (czyli efektywnie nie skalując wcale). Ostatecznie okazało się, że można normalnie podać czterowymiarowy tensor z danymi, a funkcja trainNetwork automatycznie podzieli go na batche rozmiaru określonego przez parametr MiniBatchSize.

Adaptacja sieci w cuDNN do potrzeb tego projektu również przeszła płynnie. Wprowadzone przeze mnie modyfikacje:

- dodanie uczenia w wielu epochach, gdyż początkowo program iterował po prostu przez zbiór danych jeden raz,
- pomiary czasu procesu uczenia i inferencji z wykorzystaniem eventów CUDA,
- dodanie fabryki modeli, które można podmieniać bez potrzeby przekompilowania projektu (w pliku `c++/src/builders.cpp`).

Implementacja zaawansowanego modelu, nawet MobileNet, który ma stosunkowo mało warstw, byłaby ciężka w tej bibliotece, a mi powoli kończył się czas. Zdecydowałem więc dodać kolejny prosty model, SimpleConvNet, by oprócz sieci w pełni połączonych porównać jeszcze wydajność z wykorzystaniem sieci konwolucyjnych.




Rysunek 2. Diagram sieci SimpleConvNet

```
void cudl::SimpleConvNetBuilder(Network &model) {
    model.add_layer(new Conv2D("conv1", 16, 5));
    model.add_layer(new Pooling("pool", 2, 0, 2, CUDNN_POOLING_MAX));
    model.add_layer(new Conv2D("conv2", 32, 5));
    model.add_layer(new Pooling("pool", 2, 0, 2, CUDNN_POOLING_MAX));
    model.add_layer(new Dense("dense1", 500));
    model.add_layer(new Activation("relu", CUDNN_ACTIVATION_RELU));
    model.add_layer(new Dense("dense2", 10));
    model.add_layer(new Softmax("softmax"));
}
```

Kod 7. Definicja modelu SimpleConvNet w C++

Eventy CUDA w bibliotece cuDNN przyjmują wskaźnik typu float, do którego zapisują czas jaki upłynął między eventami w milisekundach. Tracimy przez to rozdzielczość względem typu double, natomiast ostatecznie różnice między wydajnością różnych technologii okazała się na tyle duża, że utrata tej rozdzielczości nie ma znaczenia.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
float milliseconds;

// do something

cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);
```

Kod 8. Pomiar czasu w C++ z wykorzystaniem eventów CUDA

Przygotowanie środowisk testowych zostawiłem na koniec, gdyż była to już najprostsza robota po rozwiązaniu wszystkich innych problemów. C++ i MATLAB uruchamiane są tak samo jak były do tej pory testowane. Pythonowe frameworki przepisałem z Jupytera (gdzie wygodniej było mi testować różne podejścia) do skryptów wykonywalnych – dokładniejszy opis ich działania w rozdziale 4.2.1.

Ostatni etap, czyli analiza wyników, została opisana w rozdziale 5.

## *4.2 Opis implementacyjny*

### 4.2.1 Python

PyTorch przez swoje imperatywne podejście nie dostarcza funkcji służących do trenowania i ewaluacji modelu. Ich implementacje można znaleźć w pliku `torch_funcs.py` kolejno pod nazwami `fit` i `test`.

Kod źródłowy dla TensorFlow podzielony jest w taki sam sposób – plik z funkcjami narzędziowymi `tf_funcs.py` i plik z testami `tf_benchmarks.py` – ale przez specyfikę technologii różni się implementacja tych funkcjonalności. Tutaj wymagane jest dołączenie własnych warstw klasyfikujących, jeśli chce się zmienić ilość klas w predefiniowanych modelach – odpowiadają za to funkcje `classifier_overlay` i `combine_model`. TensorFlow nie pozwala na korzystanie z eventów CUDA, więc do mierzenia czasu trenowania i inferencji modelu zaimplementowano własną klasę `PerfCounterCallback`. TensorFlow nie zwalnia

pamięci VRAM w obrębie jednego procesu, dlatego trenowanie paru modeli w pętli (tak jak zrobione to zostało dla PyTorcha) jest niemożliwe. Uczenie modelu odbywa się w funkcji `train_single_model`, która uruchamiana jest w specjalnie spawnowanych podprocesach.

Funkcje zaimplementowane dla obu frameworków w plikach `*_funcs.py` to funkcje odpowiedzialne za ładowanie zbiorów danych: `get_mnist_loaders` i `get_cifar10_loaders`, oraz implementacje modeli FullyConnectedNet i SimpleConvNet. Funkcje z tego pliku importowane są w `*_benchmarks.py`, gdzie w funkcja `env_builder` przygotowuje środowiska testowe dla kolejnych modeli i przeprowadzane są testy wydajnościowe.

### 4.2.2 MATLAB

Funkcje narzędziowe znajdują się w katalogu MATLAB w plikach `loadCIFARData.m`, `downloadCIFARData.m`, `loadMNISTImgsAndLabels.m` oraz `saveTrainingState.m` a ich nazwy są adekwatne do funkcjonalności. W plikach `*_test.m` tworzona jest konfiguracja sieci, wczytywane są dane z wykorzystaniem w razie potrzeby podmieniane są odpowiednie warstwy sieci do kompatybilności z wykorzystywanymi zbiorami danych i przeprowadzane są testy wydajnościowe procesu uczenia i inferencji.

### 4.2.3 cuDNN

Tak jak pisałem w rozdziale 2.1, implementacja benchmarków z wykorzystaniem biblioteki cuDNN została oparta o otwartoźródłowy projekt Jacka Hana[x], który zmodyfikowałem na potrzeby projektu. Wprowadzone przeze mnie modyfikacje:

- dodanie uczenia w wielu epochach, gdyż początkowo program iterował po prostu przez zbiór danych jeden raz,
- pomiary czasu procesu uczenia i inferencji z wykorzystaniem eventów CUDA,
- dodanie fabryki modeli, które można podmieniać bez potrzeby przekompilowania projektu (w pliku `src/builders.cpp`).

Szczegółowa instrukcja kompilacji i uruchomienia w pliku README.md, jak i wszystkie pliki źródłowe, znajdują się w katalogu c++.

### 4.2.4 Zbiory danych

Zbiór CIFAR-10, jak wspominałem w rozdziale 4.1, popierany jest przez każdy framework z osobna w odpowiednim dla danej technologii formacie. Zachowana jest struktura katalogów, by uniknąć problemów np. z brakiem dostępu do nieistniejącego katalogu.

Zbiór MNIST należy pobrać ręcznie korzystając ze skryptu `download_mnist.sh`, który znajduje się w katalogu datasets/mnist-digits. Pobieranie następuje również automatycznie przy wykonaniu skryptu uruchamiającego wszystkie benchmarki. Pliki są w formacie bajtowym proponowanym przez autorów zbioru, bez wstępnej zmiany na faktyczny obraz.

## *4.3 Opis uruchomieniowy*

Benchmarki do uruchomienia wymagają karty graficznej nVidii – korzystałem z GeForce GTX 1050. Poniżej znajduje się tabela z wersjami wykorzystywanego oprogramowania. Większość wersji nie jest od siebie zależna, natomiast te oznaczone gwiazdką zostały odgórnie narzucone przez wymóg kompatybilności z TensorFlow.

Tabela 1. Wymagane wersje oprogramowania

| Oprogramowanie | Wersja |
|---|---|
| Sterownik GPU nVidii* | 520.61.05 |
| CUDA Toolkit* | 11.8 |
| cuDNN* | 8.6.0.163 |
| Python | 3.11.3 |
| TensorFlow | 2.12.0 |
| PyTorch | 2.0.0+cu118 |
| MATLAB | R2023a |
| Kompilator C++ | Dowolny obsługujący standard C++14 (g++ 11.3.0 użyty w projekcie) |

Benchmarki uruchomić można zbiorczo za pomocą skryptu `runtests.sh`, który znajduje się w katalogu głównym. Nie posiada on żadnych argumentów wejściowych. Możliwe jest również uruchamianie benchmarków z osobna, wykonując pliki źródłowe w podkatalogu każdej z technologii, jednak nie jest to wskazane.

# 5 Analiza wyników

Wszystkie poniżej wypisane czasy przetwarzania zostały podane w milisekundach.

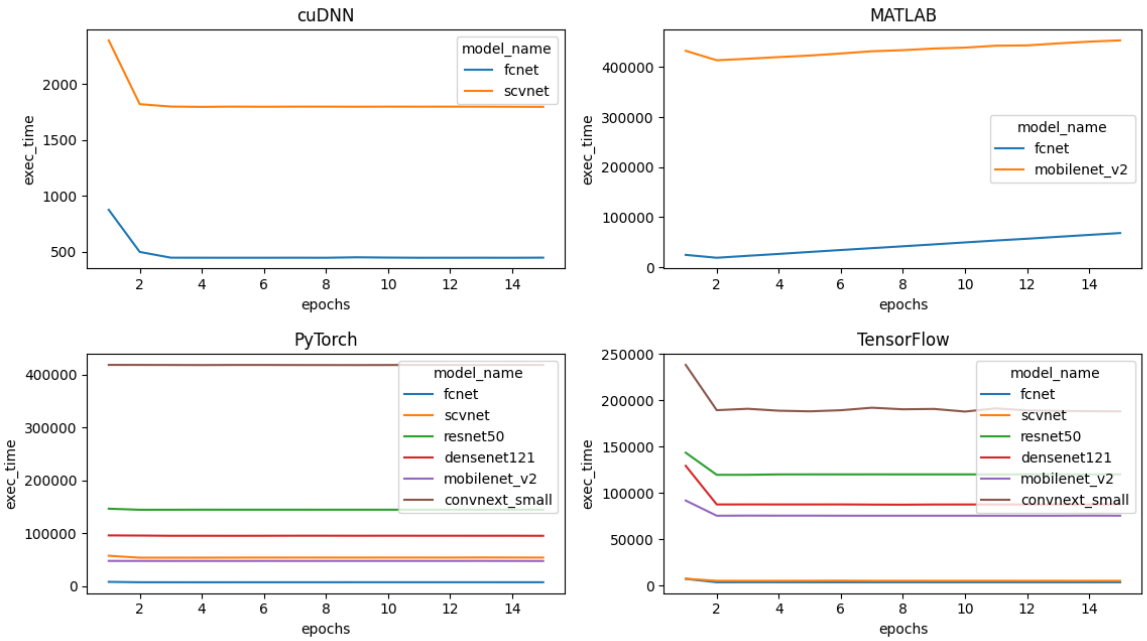
Tabela 2. Średni czas uczenia jednego epocha

| Framework | FCNet | SCVNet | ResNet50 | DenseNet121 | MobileNet v2 | ConvNeXt Small |
|---|---|---|---|---|---|---|
| MATLAB | 42621,3 | X | X | X | 433933,3 | X |
| PyTorch | 7012,5 | 53857,0 | 144477,5 | 95096,9 | 47289,0 | 418458,7 |
| TensorFlow | 3702,1 | 5202,9 | 121504,6 | 90265,3 | 76482,2 | 192714,5 |
| cuDNN | 479,5 | 1837,1 | X | X | X | X |

Tabela 3. Średni czas inferencji dla całego zbioru testowego

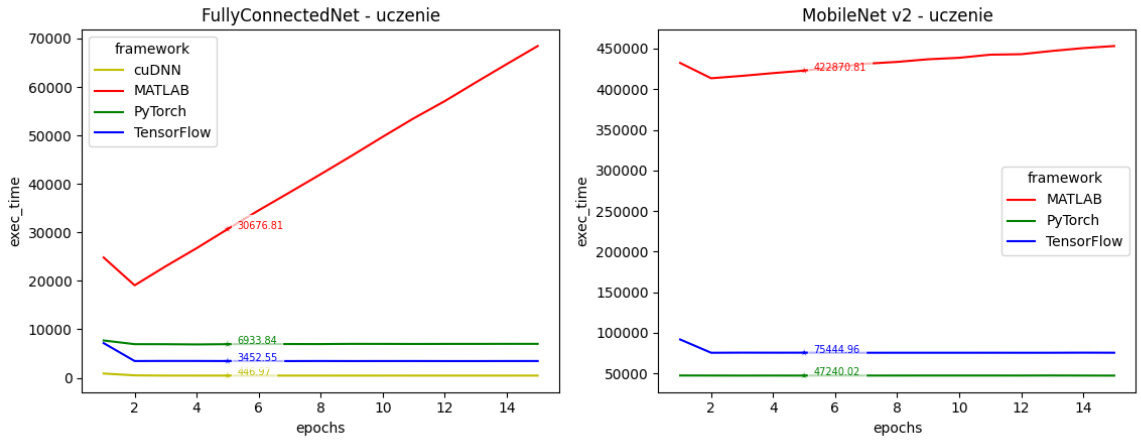
| Framework | FCNet | SCVNet | ResNet50 | DenseNet121 | MobileNet v2 | ConvNeXt Small |
|---|---|---|---|---|---|---|
| MATLAB | 179,3 | X | X | X | 2539,0 | X |
| PyTorch | 414,9 | 5090,2 | 5200,4 | 4669,2 | 1944,2 | 5264,3 |
| TensorFlow | 232,5 | 290,0 | 3704,5 | 3388,4 | 2495,8 | 12363,4 |
| cuDNN | 9,5 | 10,3 | X | X | X | X |

Na pierwszy rzut oka widać to czego można było spodziewać się od początku – C++ nie ma sobie równych pod względem szybkości wykonywania kodu, a przynajmniej nie jeśli porównujemy go z językami interpretowanymi jakimi są Python i MATLAB. Czas inferencji mniejszy o dwa rzędy wielkości naprawdę robi wrażenie. Niewiele więcej można jednak wyczytać z surowych danych – przyjrzyjmy się jeszcze wykresom.

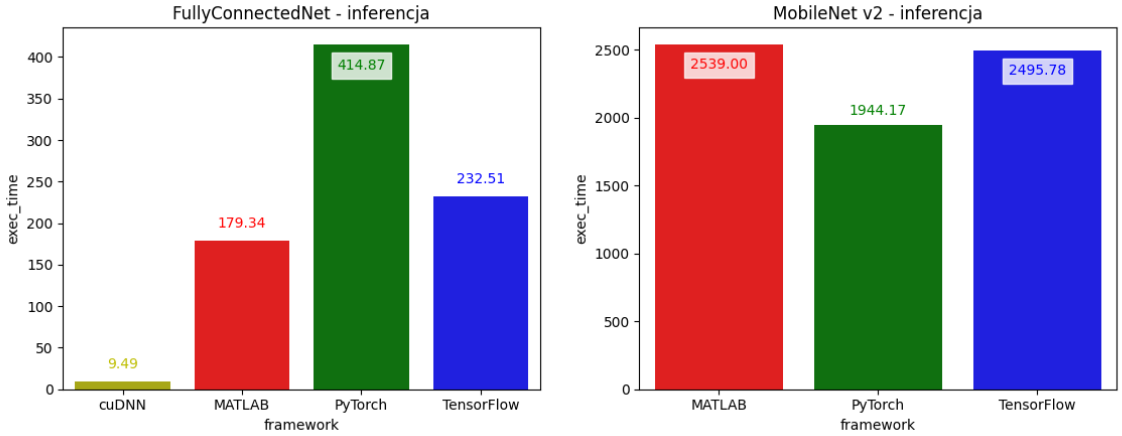



Wykres 1. Zbiorcze przedstawienie rezultatów benchmarków z etapu uczenia

Na zbiorczym porównaniu  technologii na wykresie 1. zauważyć można, że PyTorch jako jedyny nie ma problemu z wydłużającym się pierwszym epochem. Trzeba by było sprawdzić, czy stosuje on jakieś ciekawe rozwiązanie tego problemu, czy np. po prostu dane ładowane są na kartę od razu po inicjalizacji modelu, a nie dopiero w trakcie uczenia, jak ma to miejsce u innych.




Wykres 2. Porównanie MATLABa – proces uczenia



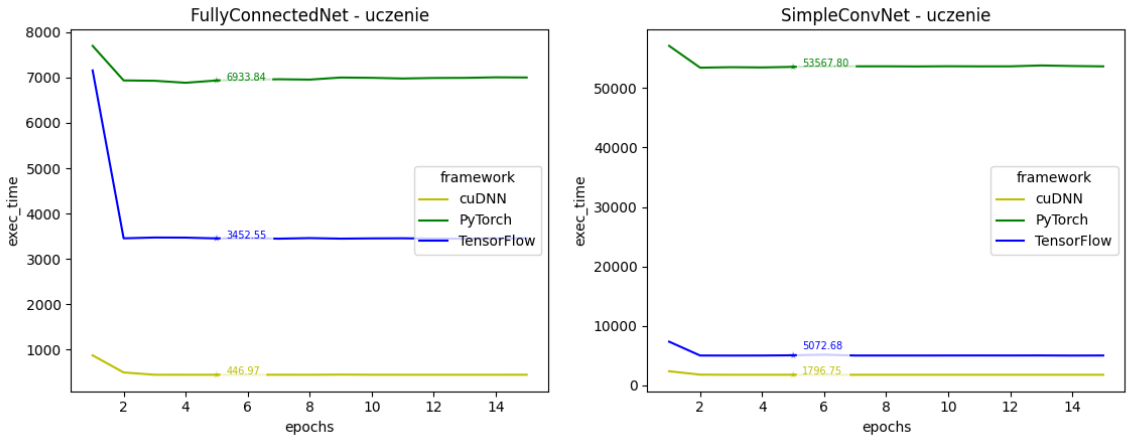

Wykres 3. Porównanie MATLABa – proces inferencji

Od razu w oczy rzucają się dwa problemu MATLABa:

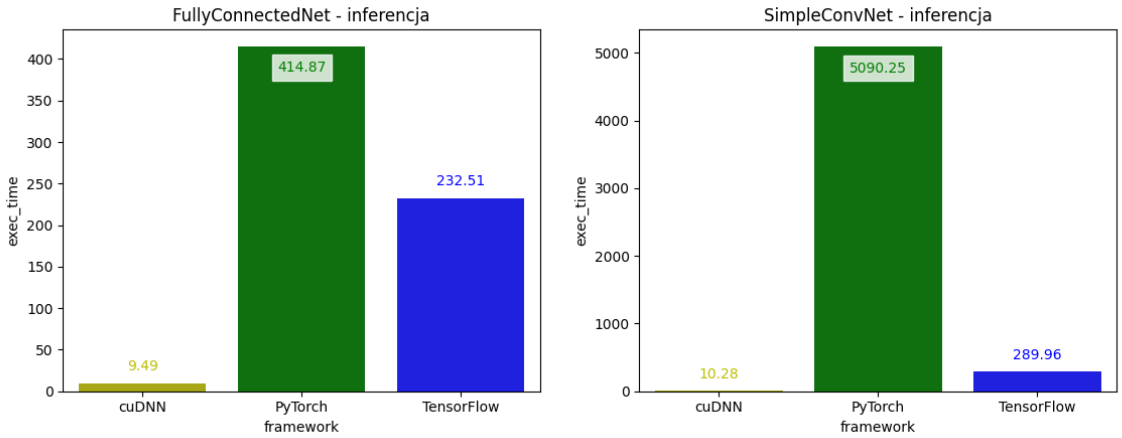
- tragiczna wydajność w porównaniu nawet z innym językiem interpretowanym,
- każdy kolejny epoch trwa dłużej niż poprzedni.

Problem wydłużających się epochów wygląda jakby był zauważalny głównie przy małych modelach. Choć dla MobileNetu bezwzględnie wydłużył się o podobną ilość czasu, to w perspektywie ile trwało przetrenowanie wszystkich epochów nie jest to już aż tak znaczące. Problem ten jednak często pojawia się na forach MathWorks i zdaje się, że nie ma na niego

żadnego rozwiązania. Czasy inferencji natomiast są porównywalne z osiągami PyTorcha oraz TensorFlowa – chociaż tyle.

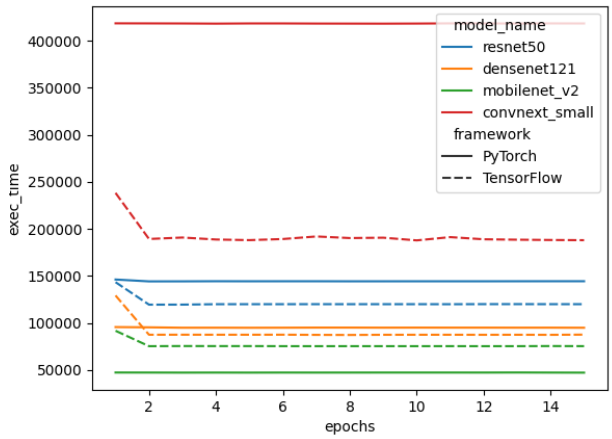



Wykres 4. Porównanie cuDNN – proces uczenia

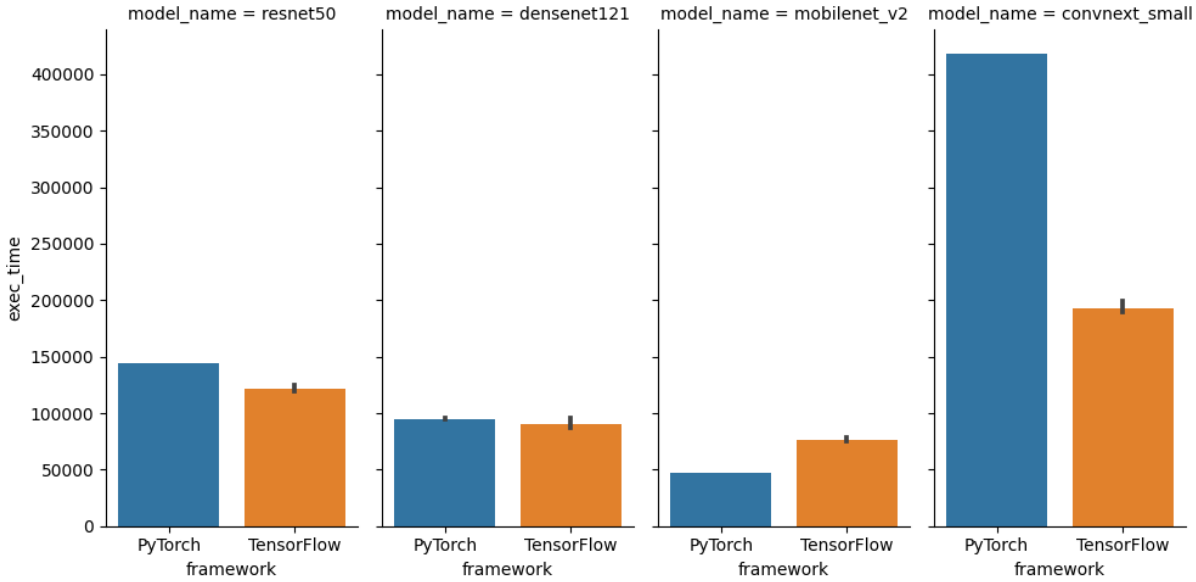



Wykres 5. Porównanie cuDNN – proces inferencji

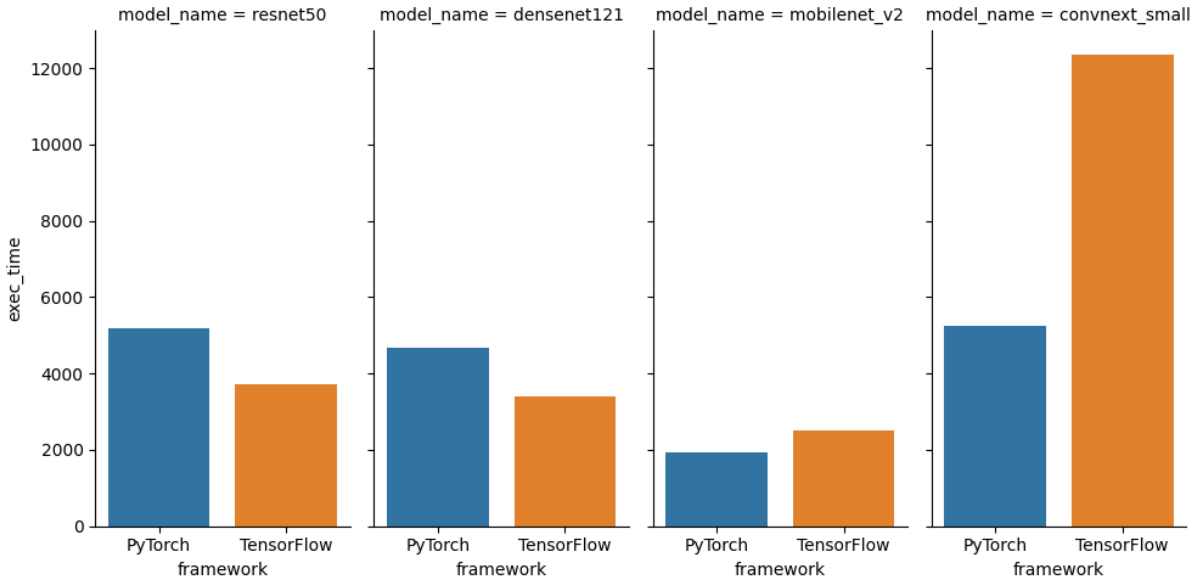
Tak jak już zauważyłem wcześniej – porównanie C++ jest bezkonkurencyjny w porównaniu do języków interpretowanych, nawet jeśli większość kosztownych obliczeń przeniesiemy z CPU na kartę graficzną. Ciekawą rzeczą jaką można tu natomiast zaobserwować jest tragiczna wydajność PyTorcha na tak prostym modelu jakim jest SimpleConvNet, nawet porównując tylko do TensorFlowa.




Wykres 6. Porównanie czasu uczenia predefiniowanych modeli




Wykres 7. Średni czas uczenia predefiniowanych modeli




Wykres 8. Czas inferencji dla predefiniowanych modeli

Słaba wydajność PyTorcha przy uczeniu modelu ConvNeXt potwierdza tezę, że framework ten musi radzić sobie gorzej z sieciami konwolucyjnymi niż TensorFlow. Tym bardziej jednak zaskakuje wynik inferencji dla tego modelu – PyTorch poradził sobie podobnie jak np. dla DenseNetu, ale TensorFlow był tutaj ponaddwukrotnie wolniejszy. Obserwujemy jeszcze przewagę PyTorcha nad TensorFlowem przy uczeniu i ewaluacji MobileNetu, mimo że dla pozostałych modeli czasy uczenia były zbliżone, a przy ewaluacji wygrywał TensorFlow.

# 6 Podsumowanie i wnioski

## *6.1 Wnioski dot. technologii*

Patrząc na wyniki MATLABa chciałoby się powiedzieć, że to świetny framework do głębokich sieci neuronowych, kiedy warunkiem koniecznym jest korzystanie z MATLABa. To co MATLAB traci przez niską wydajność, nadrabia niesamowitą prostotą instalacji i obszernym ekosystemem. By mieć dostęp do przyspieszenia sprzętowego wystarczy bowiem zainstalować tylko sterowniki karty graficznej i dwa toolboxy. Domyślam się, że podobnie prosto byłoby ustawić przetwarzanie na wielu kartach graficznych równolegle. I właśnie ten ekosystem potężnych toolboxów i innych narzędzi jest największą zaletą MATLABa.

Szybkość, jaką oferuje cuDNN zupełnie nie dziwi. Bolączką tej biblioteki jest jednak bardzo uboga funkcjonalność – sporą część projektu, którą w innych przypadkach implementuje sam framework, tu trzeba napisać własnoręcznie. Dobrze byłoby porównać wydajność samej biblioteki cuDNN z którymś z faktycznych frameworków do sieci neuronowych dla C++, by mieć realne odniesienie – porównywanie do języków interpretowanych po prostu nie jest fair.

TensorFlow jest zorientowany na wydajność i to widać. Szybkość uczenia modeli i inferencji jednak opłacamy nie tak przyjemnym procesem programowania jak w przypadku PyTorcha. Mimo że osobiście interesuje się programowaniem funkcyjnym to funkcyjne API Keras nie raz przyprawiło mnie o ból głowy. Jest to framework rozwijany przez korporację, a dokumentacja świetnie to odzwierciedla.

Najprzyjemniej pracowało mi się z PyTorchem. Jestem miło zaskoczony tym faktem, bo na początku zastanawiałem się w ogóle czy brać go pod uwagę w tym projekcie. Dokumentacja jest dogłębna i widać, że pisana przez ludzi dla ludzi, a składnia jest dużo bardziej *pythonic* niż w przypadku TensorFlowa (za sprawą trzeciej reguły filozofii PyTorcha). Parę rzeczy, które w TensorFlow dzieją się automatycznie, np. wybór GPU, przebieg trenowania i ewaluacji, a w PyTorchu trzeba zadbać o to samemu – wbrew pozorom znacznie ułatwia to pracę z frameworkiem (reguła druga). Mimo że wydajnościowo jest trochę wolniejszy od TensorFlowa, to sam kod pisze się w nim dużo szybciej – flagowym przykładem tego jest dostosowanie predefiniowanego modelu do własnych potrzeb w 4. i 5. fragmencie kodu (reguła pierwsza). A jeśli porównamy koszty czasu pracy programisty a pracy komputera, jasnym będzie, który lepiej optymalizować.

## 6.2 Wnioski dot. projektu

Projekt zakładał zrealizowanie benchmarków wydajnościowych dla różnych języków programowania przy przetwarzaniu obrazów z wykorzystaniem sieci neuronowych. Benchmarki te opierały się o uczenie i ewaluację różnych modeli klasyfikujących obrazy w trzech językach: C++, MATLAB oraz Python – tu sprawdzono dwa wiodące frameworki, którymi są PyTorch i TensorFlow.

Zrealizowano znaczną większość początkowych założeń projektu. PyTorch i TensorFlow przetestowane zostały w pełni, zaimplementowano w nich wszystkie planowane modele klasyfikacyjne. MATLAB z uwagi na niepełny zbiór modeli predefiniowanych został przeanalizowany w okrojonym zakresie, natomiast wyniki testów i tak wystarczająco świadczą o wydajności tej technologii w zastosowaniu przy głębokich sieciach neuronowych. Biblioteka cuDNN została przetestowana w takim zakresie na jaki pozwalała jej okrojona funkcjonalność.

Projekt ten miał jednak całkiem szeroki zakres, zwłaszcza biorąc pod uwagę, że nie miałem wcześniej styczności z żadnym z badanych frameworków, a o głębokich sieciach neuronowych posiadałem jedynie nieugruntowaną i skąpą wiedzę teoretyczną [15]. Największą bolączką tego zagadnienia jest konieczność nauki i konfiguracji wielu technologii, co samo w sobie zajmuje wiele czasu, a co dopiero kiedy mówimy o czterech frameworkach na raz. Projekt ten był dla mnie osobiście całkiem ciekawy, jako że interesuję się szeroko pojętym data science; momentami jednak, implementowanie tych samych nie odkrywczych rzeczy po cztery razy robiło się monotonne.

# 7 Możliwości rozbudowy

Na pewno dobrze byłoby rozwinąć część projektu dotyczącą C++, tylko może już nie w czystym cuDNN. Przeprowadzenie testów dla LibTorcha lub C++ API TensorFlowa pozwoli na porównanie obu frontendów tych technologii, a za razem pozwoli sprawdzić tezę, czy czysty cuDNN faktycznie jest znacząco od nich szybszy, czy nie jest to warte zachodu.

PyTorch 2.0 wprowadziła możliwość kompilacji modeli co podobno optymalizuje ich działanie. Wersja ta została wydana w marcu 2023r. i jest stosunkowo świeżą funkcjonalnością, przez co niewiele poradników (w tym oficjalna dokumentacja) sugeruje jeszcze kompilacje modeli przed trenowaniem – dlatego w tym projekcie zdecydowałem się z niej nie korzystać. Zwłaszcza, że największy zysk dotyczy high-endowych kart graficznych. Porównanie wydajności PyTorcha z i bez kompilacji modeli może być ciekawym tematem do poruszenia.

Na ten moment benchmarki uruchamiane są przez skrypt, ale konfiguracja każdego z nich zapisana jest w plikach źródłowych. Dobrze byłoby przenieść ustawianie konfiguracji do tego wspólnego skryptu, gdyż tak łatwiej sprawdzać działanie dla różnych parametrów, np. jak zmiana wielkości batcha wpływa na wydłużenie pierwszego epocha i generalnie całe uczenie.

# Bibliografia


[1] He, K., Zhang, X., Ren, S. and Sun, J. (2015). *Deep Residual Learning for Image Recognition*. [online] arXiv.org. Available at: https://arxiv.org/abs/1512.03385 [Accessed 13 Jun. 2023].

[2] Huang, G., Liu, Z. and Weinberger, Kilian Q (2018). *Densely Connected Convolutional Networks*. [online] arXiv.org. Available at: https://arxiv.org/abs/1608.06993 [Accessed 13 Jun. 2023].

[3] Liu, Z., Mao, H., Wu, C.-Y., Feichtenhofer, C., Darrell, T. and Xie, S. (2022). *A ConvNet for the 2020s.* [online] Available at: https://arxiv.org/abs/2201.03545 [Accessed 13 Jun. 2023].

[4] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.-C. (2019). *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. [online] arXiv.org. Available at: https://arxiv.org/abs/1801.04381 [Accessed 13 Jun. 2023].

[5] Han, J. (2020). *MNIST with cuDNN*. [online] GitHub. Available at: https://github.com/haanjack/mnist-cudnn [Accessed 13 Jun. 2023].

[6] Howard, J. (2018). *What is torch.nn really? — Switch to CNN*. [online] pytorch.org. Available at: https://pytorch.org/tutorials/beginner/nn_tutorial.html#switch-to-cnn [Accessed 13 Jun. 2023].

[7] Keras Team (2018). *Keras documentation: ResNet and ResNetV2*. [online] keras.io. Available at: https://keras.io/api/applications/resnet/#resnet50-function [Accessed 13 Jun. 2023].

[8] Krizhevsky, A., Nair, V. and Hinton, G. (2009). *CIFAR-10 and CIFAR-100 datasets*. [online] Toronto.edu. Available at: https://www.cs.toronto.edu/~kriz/cifar.html [Accessed 13 Jun. 2023].

[9] LeCun, Y., Burges, C. and Cortes, C. (2009). *MNIST handwritten digit database*. [online] lecun.com. Available at: http://yann.lecun.com/exdb/mnist/ [Accessed 13 Jun. 2023].

[10] MathWorks (2023). *Pretrained Deep Neural Networks*. [online] www.mathworks.com. Available at: https://www.mathworks.com/help/deeplearning/ug/pretrained-convolutional-neural-networks.html [Accessed 13 Jun. 2023].

[11] Nielsen, M.A. (2018). *Neural Networks and Deep Learning*. [online] neuralnetworksanddeeplearning.com. Available at: http://neuralnetworksanddeeplearning.com/chap1.html [Accessed 13 Jun. 2023].

[12] The PyTorch Team (2017). *torchvision.models — Torchvision 0.8.1 documentation*. [online] pytorch.org. Available at: https://pytorch.org/vision/0.8/models.html [Accessed 13 Jun. 2023].

[13] The PyTorch Team (2018). *The road to 1.0: production ready PyTorch*. [online] www.pytorch.org. Available at: https://pytorch.org/blog/the-road-to-1_0/ [Accessed 13 Jun. 2023].

[14] The TensorFlow Team (2019). *TensorFlow 2.0 is now available!* [online] blog.tensorflow.org. Available at: https://blog.tensorflow.org/2019/09/tensorflow-20-is-now-available.html [Accessed 13 Jun. 2023].

[15] Wikipedia Contributors (2019). *Dunning–Kruger effect*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect [Accessed 13 Jun. 2023].

[16] Zalando Research (2020). *zalandoresearch/fashion-mnist*. [online] GitHub. Available at: https://github.com/zalandoresearch/fashion-mnist [Accessed 13 Jun. 2023].