

Programowanie równoległe i rozproszone

Politechnika Krakowska

Projekt

Temat

*Porównanie czasów działania sieci neuronowych oraz
głębokich sieci w środowiskach TensorFlow, PyTorch i Theano*

Paweł Suchanicz,
Rafał Niemczyk

18 stycznia 2020

Spis treści

1	Wstęp	2
1.1	Opis projektu	2
1.2	Opis zbiorów danych	2
1.2.1	MNIST	2
1.2.2	CIFAR10	2
1.2.3	CIFAR100	3
1.2.4	Letter Recognition	4
1.3	Opis użytych frameworków	5
1.3.1	Keras	5
1.3.2	Tensorflow	5
1.3.3	PyTorch	5
1.3.4	Theano	6
1.4	Specyfikacja sprzętowa	6
2	Wyniki	6
2.1	Implementacja	6
2.1.1	Wczytanie danych	6
2.1.2	Architektura sieci	7
2.1.3	Parametry uczenia	13
2.1.4	Podział danych	14
2.1.5	Walidacja krzyżowa	14
2.2	Porównanie wyników - CPU	14
2.2.1	Walidacja krzyżowa	14
2.2.2	Podział danych 70/30	15
2.3	Porównanie wyników - GPU	15
2.3.1	Walidacja krzyżowa	15
2.3.2	Podział danych 70/30	15
2.4	Wykresy porównawcze według zbiorów	17
2.5	Wnioski	20

1 Wstęp

1.1 Opis projektu

Celem projektu było porównanie czasów działania sieci neuronowych oraz głębokich sieci neuronowych w środowiskach: TensorFlow, PyTorch oraz Theano. Czasy należało zmierzyć na CPU oraz GPU, jeśli była taka możliwość.

Pomiary czasów należało dokonać dla każdego z 4 zbiorów danych. Dlatego zbudowano 4 różne architektury sieci neuronowych. W każdym ze środowisk zbudowano te same architektury oraz zachowano te same parametry uczenia czy też funkcje aktywacji.

Dodatkowo należało dokonać eksperyment w 2 wariantach:

- Podział na zbiór uczący i testowy (70
- 10 krotna walidacja krzyżowa

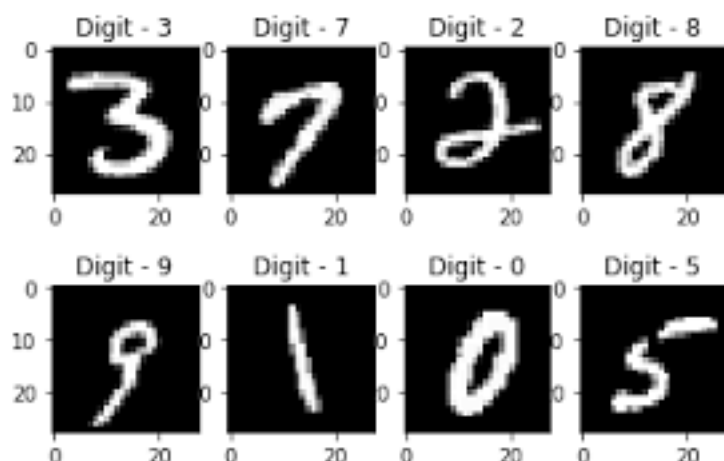
Aby porównanie czasów było jak najbardziej obiektywne użyto domyślnego timera udostępnionego przez Python - timeit.

Aby uruchomić obliczenia sieci neuronowych na GPU należało użyć technologii CUDA. CUDA (Compute Unified Device Architecture) to architektura kart graficznych opracowana przez firmę Nvidia. Pozwala na wykorzystanie mocy obliczeniowej na kartach graficznych.

1.2 Opis zbiorów danych

1.2.1 MNIST

Jest to zestaw odręcznie napisanych cyfr (od 0 do 9), który powszechnie używany jest do szkolenia różnych systemów przetwarzania obrazu oraz w dziedzinie uczenia maszynowego. Baza danych MNIST zawiera 60 000 obrazów szkoleniowych i 10 000 obrazów testowych. Wielkość każdego obrazu to 28 x 28 pikseli w skali szarości.

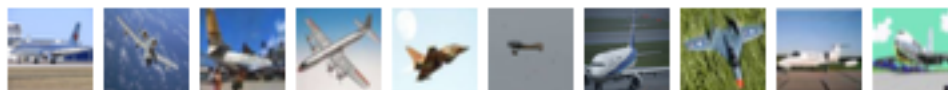


1.2.2 CIFAR10

Jest to zestaw zawierający 60 000 kolorowych obrazów o wielkości 32 x 32 pikseli. Obrazy przedstawiają jedną z 10 klas. Na poniższym obrazku znajdują się klasy wraz z przykładowymi

obrazami. Każda klasa zawiera 6000 obrazów. Dane treningowe składają się z 50 000 obrazów, a testowe z 10 000, czyli w danych treningowych znajdują się 5 000 obrazów jednej klasy, a w testowe 1 000. Klasy całkowicie się wykluczają.

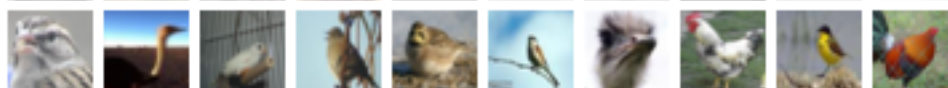
airplane



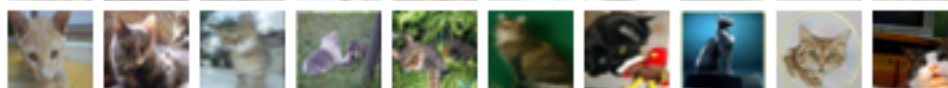
automobile



bird



cat



deer



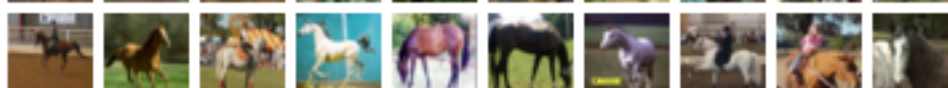
dog



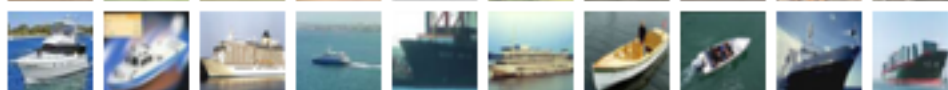
frog



horse



ship



truck



1.2.3 CIFAR100

Ten zestaw danych zawiera 60 000 kolorowych obrazów. Obrazy reprezentują 100 różnych klas, co oznacza, że na każdą klasę przypada 600 obrazów. Z każdej klasy wyodrębnionych zostaje 500 obrazów treningowych oraz 100 testowych. Każda z 100 klas jest przypisana do jednej z 20 superklas, czy też nadklas. Każdy obraz ma dwie etykiety. Jedną jako etykietę nadklasy i drugą jako własną. Poniżej znajdują się tabelami przedstawiające nadklasy i klasy do nich należące.

Nazwa nadklasy	Nazwy klas zawartych w nadklasie
Ssaki wodne	bóbr, delfin, wydra, foka, wieloryb
Ryby	ryba akwariowa, rekin, pstrąg, płastugi, płaszcza
Kwiaty	storczyk, mak, róża, słonecznik, tulipan
Pojemniki na żywność	butelka, miska, puszka, kubek, talerz
Warzywa i owoce	jabłko, grzyb, pomarańcza, gruszka, papryka
Urządzenia elektryczne	zegar, klawiatura, lampa, telefon, telewizja
Meble domowe	łóżko, krzesło, kanapa, stół, szafa
Owady	pszczoła, chrząszcz, motyl, gąsienica, karaluch
Duże drapieżniki	niedźwiedź, lampart, lew, tygrys, wilk
Wszystkożerne i roślinożerne	wielbłąd, szympan, słoń, kangur, bydło
Średniej wielkości ssaki	lis, jeżozwierz, szop, skunks, opos
Bezkregowce	krab, homar, ślimak, pająk, robak
Człowiek	niemowlę, chłopak, dziewczyna, mężczyzna, kobieta
Gady	krokodyl, dinozaur, jaszczurka, wąż, żółw
Małe ssaki	chomik, mysz, królik, wiewiórka, ryjówka
Drzewa	klon, dąb, palma, sosna, wierzba
Pojazdy 1	rower, autobus, motocykl, furgonetka, pociąg
Pojazdy 2	kosiarka, rakietka, tramwaj, czołg, ciągnik

1.2.4 Letter Recognition

Dane stworzone po to, aby móc na podstawie wyświetlanych pikseli zidentyfikować jedną z 26 wielkich liter alfabetu angielskiego. Obrazy znaków zostały oparte na 20 różnych czcionkach, a każda litera w tych 20 czcionkach została losowo zniekształcona, aby utworzyć plik zawierający 20 000 unikalnych cech. Każda litera składa się z 16 cech ją opisujących. Cechy, które tworzą poszczególną literę to:

- letter - wielka litera (od A do Z)
- x-box - pozioma pozycja x (liczba całkowita)
- y-box - pionowa pozycja y (liczba całkowita)
- width - szerokość (liczba całkowita)
- high - wysokość (liczba całkowita)
- onpix - całkowita liczba pikseli (liczba całkowita)
- x-bar - średnia liczba pikseli x (liczba całkowita)
- y-bar - średnia liczba pikseli y (liczba całkowita)
- x2bar - średnia wariancja x (liczba całkowita)
- y2bar - średnia wariancja y (liczba całkowita)
- xybar - średnia korelacja x y (liczba całkowita)
- x2ybr - średnia $x * x * y$ (liczba całkowita)
- xy2br - średnia $x * y * y$ (liczba całkowita)

- x-ege- średnia liczba krawędzi od lewej do prawej (liczba całkowita)
- xegvy - korelacja x krawędzi z y (liczba całkowita)
- y-ege- średnia liczba krawędzi od dołu do góry (liczba całkowita)
- yegvx - korelacja y krawędzi z x (liczba całkowita)

1.3 Opis użytych frameworków

1.3.1 Keras

Keras to biblioteka open-source do sieci neuronowych napisana w pythonie. zaprojektowana, aby umożliwić szybkie eksperymentowanie z głębokimi sieciami neuronowymi. Koncentruje się na byciu przyjazną dla użytkownika. W 2017 roku, zespół Google TensorFlow postanowił wesprzeć Keras w podstawowej bibliotece Tensorflow. Keras został mianowany jako przyjazny interfejs niż samodzielna platforma uczenia maszynowego. Oferuje bardziej intuicyjny zestaw narzędzi, które ułatwiają opracowywanie modeli głębokiego uczenia niezależnie od zastosowanego zaplecza obliczeniowego.

Keras był początkowo zawarty w temacie projektu, jednak po przeanalizowaniu i wglębień w temat frameworków do uczenia maszynowego w pythonie, zrezygnowano z tej opcji. Jak już wspomniano, biblioteka świetnie spisuje się do tworzenia architektury sieci neuronowych oraz do tworzenia całych modeli wraz z parametrami uczenia i funkcjami aktywacji. Jednak obliczenia wykonywane podczas treningu sieci, wykonywane domyślnie są za pomocą biblioteki TensorFlow. Aktualnie oprócz biblioteki obliczeniowej TensorFlow, są dostępne dwie inne biblioteki: Theano oraz CNTK.

1.3.2 Tensorflow

Tensorflow jest biblioteką open-source. Została napisana przez Google Brain Team w celu zastąpienia Theano. Wykorzystywana w uczeniu maszynowym i głębokich sieciach neuronowych. Została wydana 9 listopada 2015 roku. Biblioteka może do działania wykorzystywać zarówno karty graficzne, procesory jak i wyspecjalizowane mikroprocesory. Biblioteka składa się z kilku modułów. W jej najniższej warstwie znajduje się rozproszony silnik wykonawczy (ang. distributed execution engine), który w celu podniesienia wydajności został zaimplementowany w języku programowania C++. Nad nią znajdują się frontendy napisane w kilku językach programowania m.in. w Pythonie oraz C++. Powyżej umieszczona została warstwa API, która zapewnia prostszy interfejs dla powszechnie używanych warstw w modelach głębokiego uczenia. Na następną warstwę składają się wysokopoziomowe API, m.in. Keras oraz Estimator API, które ułatwiają tworzenie modeli i ich ocenę. Ponad tym znajdują się przygotowane przez twórców biblioteki gotowe do użycia modele.

Uważa się, że TensorFlow jest szybszy niż Theano. Jednak jest wolniejszy niż inne frameworki. Dodatkowo nie posiada zbyt dużej liczby wyszkolonych modeli.

1.3.3 PyTorch

Jest to open-source biblioteka do uczenia maszynowego, bazująca na bibliotece Torch. Używana do aplikacji jak przetwarzanie języka naturalnego. Opracowana głównie przez Facebook AI Research lab. Oprócz interfejsu pythona, posiada interfejs C++. Na PyTorch zbudowanych

jest wiele elementów oprogramowania do głębokiego uczenia, jak Uber Pyro. Zapewnia dwie funkcje wysokiego poziomu:

- obliczenia tensorowe z silnym przyspieszeniem za pośrednictwem procesorów graficznych GPU
- głębokie sieci neuronowe zbudowane na systemie autodiff

Do plusów biblioteki można zaliczyć np: wiele modułów, które można łatwo połączyć; łatwe pisanie własnych typów warstw i uruchamianie na GPU, wiele wstępnie przeszkolonych modeli. Minusami tej biblioteki są: pisanie własnego kodu treningowego, brak wsparcia.

1.3.4 Theano

Jest to biblioteka i kompilator optymalizujący wyrażenia matematyczne. W Theano obliczenia są wyrażane przy użyciu składni w stylu NumPy i kompilowane w celu wydajnego działania na architekturze CPU lub GPU. Projekt typu open-source, opracowany przede wszystkim przez Montreal Institute for Learning na Uniwersytecie w Montrealu. W dniu 28 września 2017 roku, Yoshua Bengio ogłosił: "Główny rozwój zostanie zakończony po wydaniu wersji 1.0, z powodu konkurencyjnych ofert silnych graczy przemysłowych. Wersja 1.0.0 została wydana 15 listopada 2017r. Na Theano zbudowano biblioteki głębokiego uczenia jak Keras, Lagane i Blocks.

1.4 Specyfikacja sprzętowa

Przy pomiarach szybkości wykonywania algorytmów wykorzystany był sprzęt w konfiguracji - do implementacji z wykorzystaniem CUDA.
Specyfikacja sprzętu:

- Procesor: Intel Core i7-4712MQ 4 x 2.30GHz
- Ram: 8GB DDR3
- System: Windows 10
- GPU: NVIDIA GeForce 840N

2 Wyniki

2.1 Implementacja

W projekcie należało użyć trzech frameworków: TensorFlow, PyTorch oraz Theano. Dwa z nich, TensorFlow oraz Theano, można użyć jako backend do biblioteki Keras. Dzięki czemu dla tych dwóch frameworków stworzono cały model uczenia używając dokładnie tych samych funkcji dostępnych dzięki Keras.

2.1.1 Wczytanie danych

Trzy z 4 zbiorów danych zawarte są w paczkach danych udostępnionych przez Keras. Należało jedynie poprawnie wywołać odpowiednią funkcję. Podobnie sytuacja wyglądała w PyTorch, gdzie zbiory były dostępne i również należało wykonać poszczególną funkcję.

Aby wczytać zbiór LetterRecognition należało użyć funkcji read-csv dostępnej w paczce Pandas.

Dane nie wymagały zbyt dużej obróbki, ponieważ nie brakowało żadnych wartości oraz były zapisane w postaci numerycznej. Podczas wczytania danych poprzez funkcję udostępnioną przez Keras, dane były domyślnie dzielone na treningowe i testowe. Jednak proporcje tych danych nie spełniały wymagań projektowych. Dlatego łączono dane i dopiero w następnym kroku dzielono je na odpowiednie proporcje lub używano do walidacji krzyżowej.

2.1.2 Architektura sieci

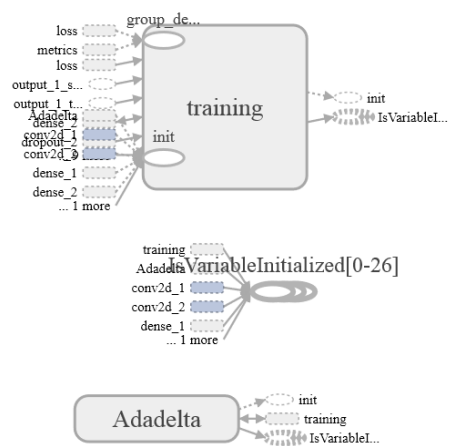
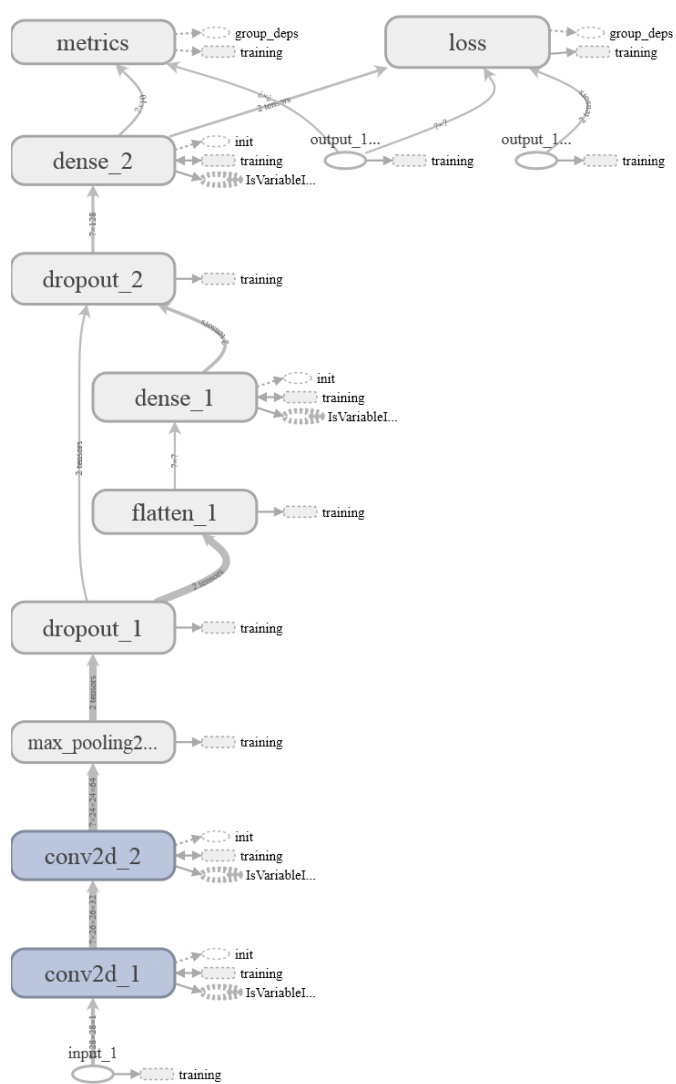
Stworzenie architektury sieci w bibliotece Keras polega głównie na funkcjach udostępnianych przez bibliotekę. Jest to bardzo proste zajęcie, należy tylko odpowiednio dodawać kolejne warstwy sieci do modelu.

W przypadku PyTorch sytuacja wygląda bardzo podobnie. Należy odpowiednio ułożyć warstwy sieci. Główne różnice można zauważyć w nazewnictwie.

Poniżej znajdują się utworzone architektury sieci dla każdego ze zbiorów danych.

- MNIST

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	multiple	320
conv2d_3 (Conv2D)	multiple	18496
max_pooling2d_1 (MaxPooling2D)	multiple	0
dropout_2 (Dropout)	multiple	0
dense_2 (Dense)	multiple	1179776
flatten_1 (Flatten)	multiple	0
dropout_3 (Dropout)	multiple	0
dense_3 (Dense)	multiple	1290
Total params: 1,199,882		
Trainable params: 1,199,882		
Non-trainable params: 0		



- CIFAR10

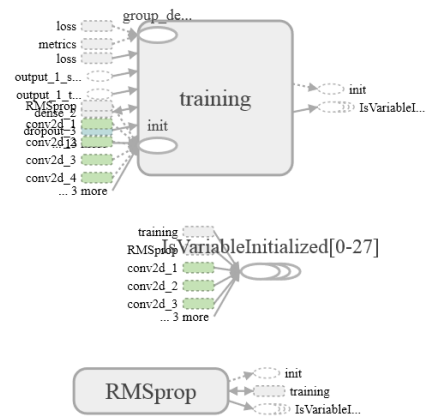
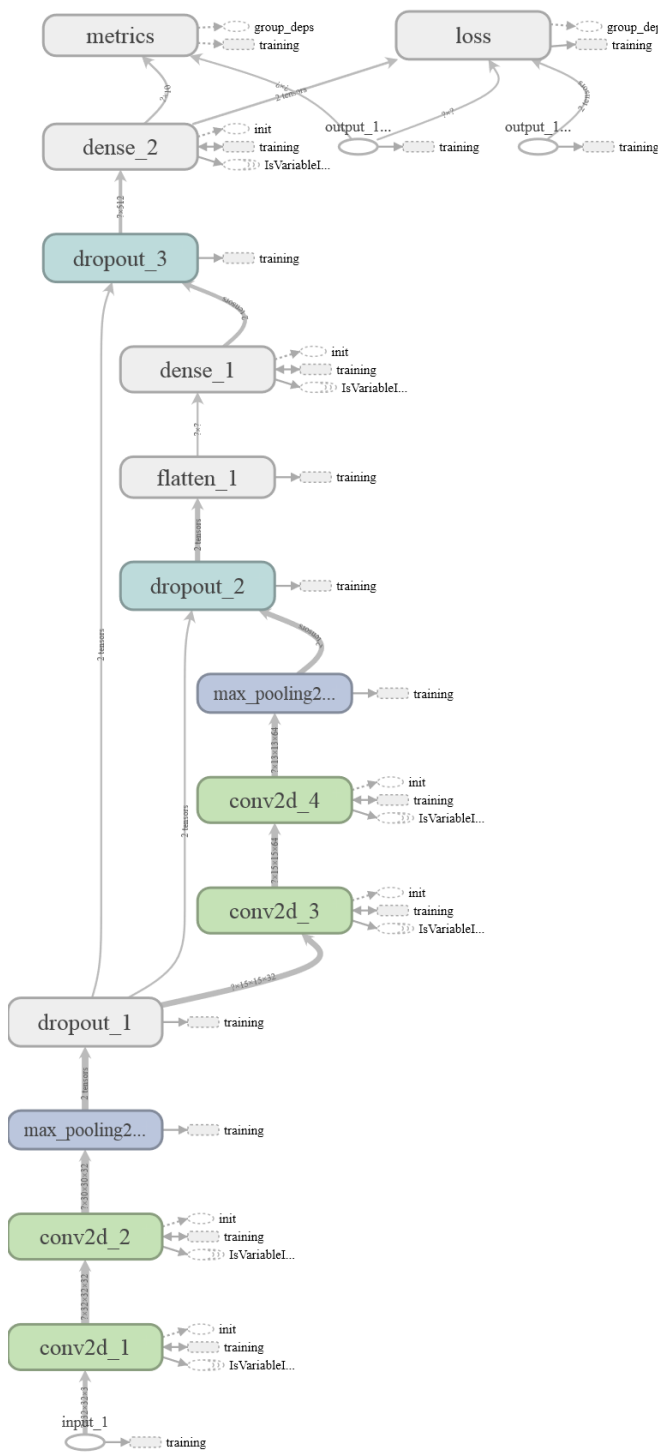
```
Model: "cifar10_model"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	multiple	896
conv2d_1 (Conv2D)	multiple	9248
max_pooling2d (MaxPooling2D)	multiple	0
dropout (Dropout)	multiple	0
conv2d_2 (Conv2D)	multiple	18496
conv2d_3 (Conv2D)	multiple	36928
max_pooling2d_1 (MaxPooling2D)	multiple	0
dropout_1 (Dropout)	multiple	0
flatten (Flatten)	multiple	0
dense (Dense)	multiple	1180160
dropout_2 (Dropout)	multiple	0
dense_1 (Dense)	multiple	5130

```

Total params: 1,250,858
Trainable params: 1,250,858
Non-trainable params: 0

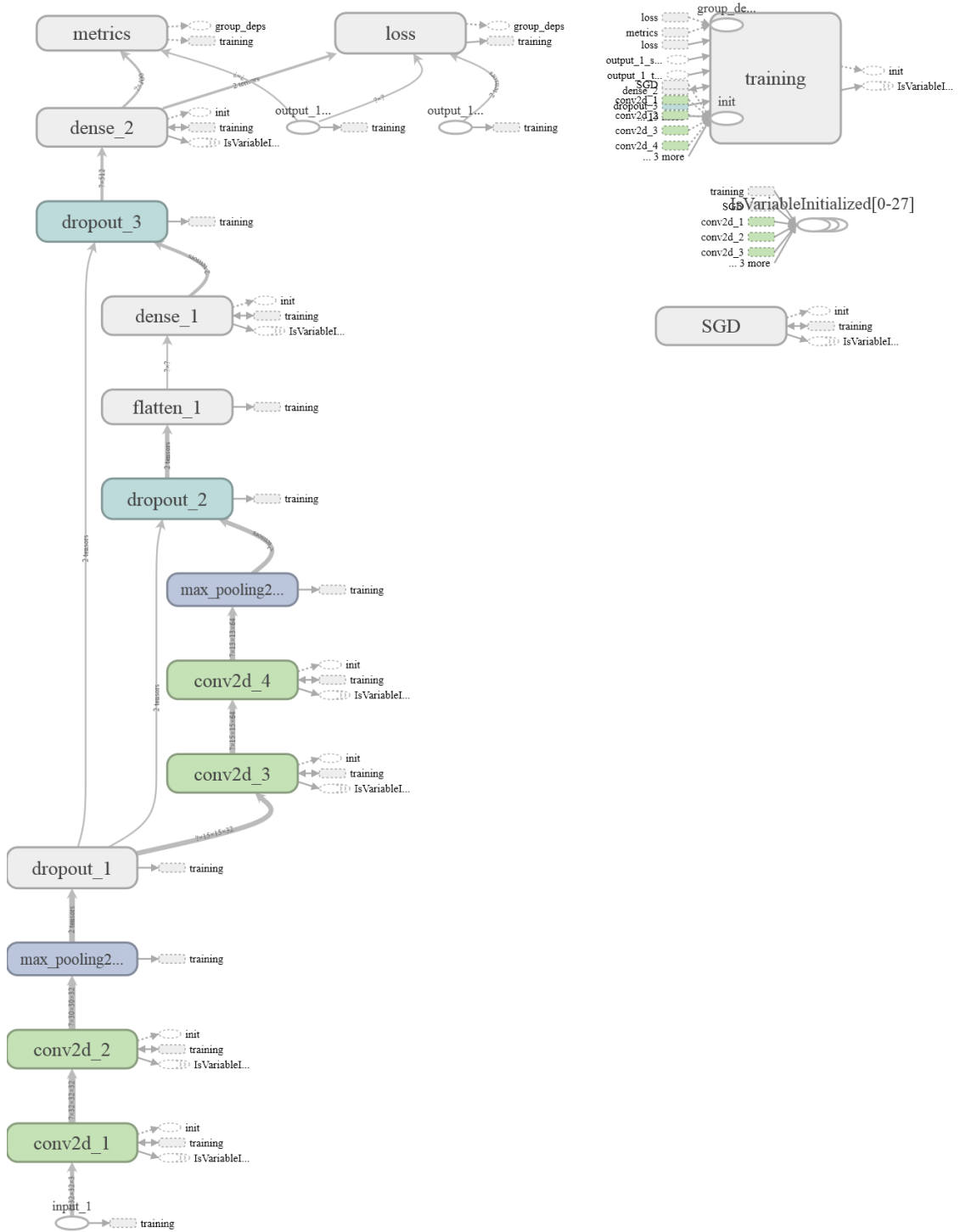
```



- CIFAR100

```
Model: "cifar100_model"

Layer (type)                 Output Shape              Param #
=====
conv2d (Conv2D)              multiple                  896
conv2d_1 (Conv2D)            multiple                  9248
max_pooling2d (MaxPooling2D) multiple                  0
dropout (Dropout)            multiple                  0
conv2d_2 (Conv2D)            multiple                 18496
conv2d_3 (Conv2D)            multiple                 36928
max_pooling2d_1 (MaxPooling2 multiple                  0
dropout_1 (Dropout)          multiple                  0
flatten (Flatten)            multiple                  0
dense (Dense)                multiple                1180160
dropout_2 (Dropout)          multiple                  0
dense_1 (Dense)              multiple                 51300
=====
Total params: 1,297,028
Trainable params: 1,297,028
Non-trainable params: 0
```



- LetterRecognition

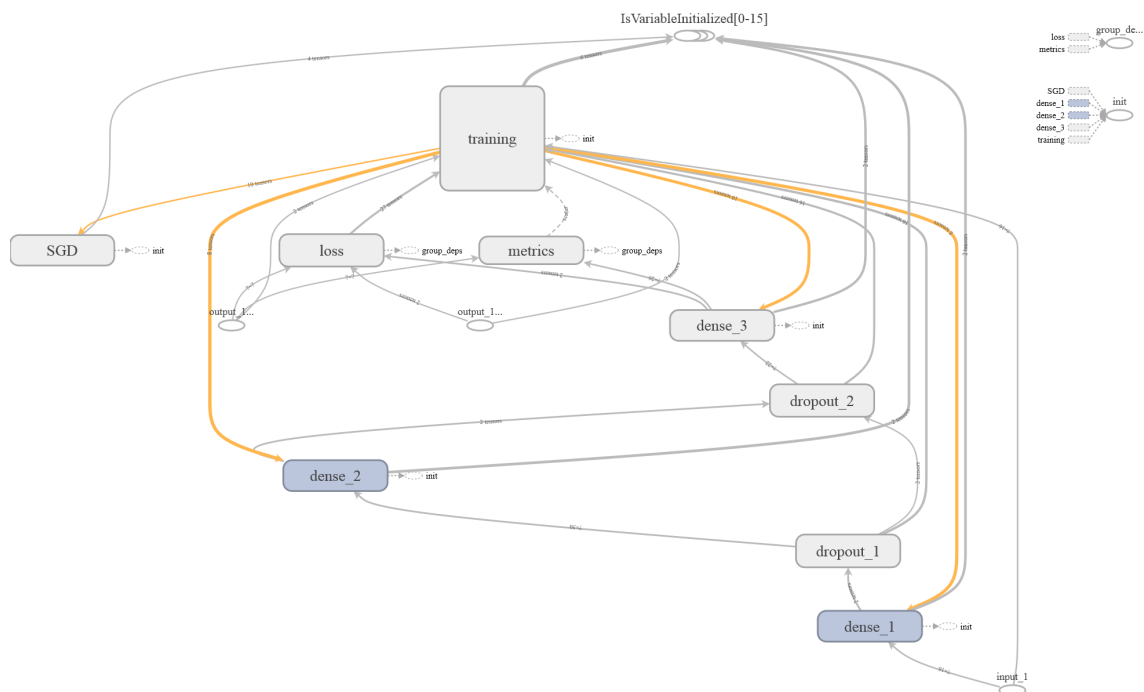
```
Model: "letter_recognition_model"
```

Layer (type)	Output Shape	Param #
dense (Dense)	multiple	850
dropout (Dropout)	multiple	0
dense_1 (Dense)	multiple	1122
dropout_1 (Dropout)	multiple	0
dense_2 (Dense)	multiple	598

```

Total params: 2,570
Trainable params: 2,570
Non-trainable params: 0

```



2.1.3 Parametry uczenia

Dla każdej architektury dobrane inne parametry uczenia, co wyszczególniono poniżej:

- MNIST:
 - funkcja błędu - categorical-crossentropy
 - optymalizator - Adadelata()
 - batch-size = 128

- liczba epok = 12
- CIFAR10:
 - funkcja błędu - categorical-crossentropy
 - optymalizator - RMSprop(learning-rate=0.0001, decay=1e-6)
 - batch-size = 128
 - liczba epok = 12
- CIFAR100:
 - funkcja błędu - categorical-crossentropy
 - optymalizator - SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
 - batch-size = 128
 - liczba epok = 12
- LetterRecognition:
 - funkcja błędu - categorical-crossentropy
 - optymalizator - SGD(lr=0.001, decay=1e-7, momentum=0.)
 - batch-size = 128
 - liczba epok = 300

Aby zainicjalizować wagi w Keras należy użyć metody `compile`, natomiast w PyTorch należy przekazać funkcję modyfikującą wagi. Tworzenie optymalizatorów jest bardzo podobne w obu frameworkach.

2.1.4 Podział danych

Wykonując zadanie zawarte w poleceniu, tzn aby podzielić dane na testowe i treningowe w proporcji 70%/30% użyto funkcji `train-test-split` z pakietu `sklearn`.

2.1.5 Walidacja krzyżowa

Walidacja krzyżowa jest metodą statystyczną, polegająca na podziale próby statystycznej na podzbiory, a następnie przeprowadzaniu wszelkich analiz na niektórych z nich (zbiór uczący), podczas gdy pozostałe służą do potwierdzenia wiarygodności jej wyników (zbiór testowy). K-Krotna walidacja to metoda w której oryginalna próba jest dzielona na K podzbiorów. Następnie kolejno każdy z nich bierze się jako zbiór testowy, a pozostałe razem jako zbiór uczący i wykonuje analizę. Analiza jest więc wykonywana K razy. K rezultatów jest następnie uśrednianych (lub łączonych w inny sposób) w celu uzyskania jednego wyniku. Aby skorzystać z walidacji krzyżowej użyto funkcji `KFold` z pakietu `sklearn`.

2.2 Porównanie wyników - CPU

2.2.1 Walidacja krzyżowa

Porównanie wyników, gdy uczenie sieci uruchomiono na CPU i wykorzystano walidację krzyżową.

Czas wykonania		TensorFlow	Theano	PyTorch
	MNIST	12089.98s	12405.71s	23848.42s
	CIFAR10	15902.45s	19150.98s	27867.25s
	CIFAR100	17127.30s	457762.47s	32609.92s
	LetterRec	1551.61s	1071.60s	2019.43s

Test accuracy		TensorFlow	Theano	PyTorch
	MNIST	82.56%	83.86%	98.97%
	CIFAR10	61.72%	62.95%	62.70%
	CIFAR100	39.52%	46.13%	43.92%
	LetterRec	69.25%	67.60%	66.55%

2.2.2 Podział danych 70/30

Porównanie wyników, gdy uczenie sieci uruchomiono na CPU i wykorzystano podział 70/30.

Czas wykonania		TensorFlow	Theano	PyTorch
	MNIST	1005.69s	1928.22s	1281.81s
	CIFAR10	1480.35s	3112.23s	1524.95s
	CIFAR100	1551.61s	74422.8454s	1890.53s
	LetterRec	90.876s	108.85s	154.82s

Test accuracy		TensorFlow	Theano	PyTorch
	MNIST	81.26%	98.96%	98.61%
	CIFAR10	57.63%	57.27%	48.75%
	CIFAR100	39.54%	44.21%	36.34%
	LetterRec	3.2%	3.68%	48.82%

2.3 Porównanie wyników - GPU

2.3.1 Walidacja krzyżowa

Porównanie wyników, gdy uczenie sieci uruchomiono na GPU i wykorzystano walidację krzyżową.

Czas wykonania		TensorFlow	Theano	PyTorch
	MNIST	3623.60s	7587.83s	4026.90s
	CIFAR10	4558.62s	9226.08s	3876.99s
	CIFAR100	4264.93s	10257.84s	3907.12s
	LetterRec	1905.68s	1451.61s	5090.07s

Test accuracy		TensorFlow	Theano	PyTorch
	MNIST	83.34%	98.94%	99.08%
	CIFAR10	63.10%	62.60%	62.53%
	CIFAR100	37.09%	36.43%	44.41%
	LetterRec	67.46 %	72.25	50.15%

2.3.2 Podział danych 70/30

Porównanie wyników, gdy uczenie sieci uruchomiono na GPU i wykorzystano podział 70/30.

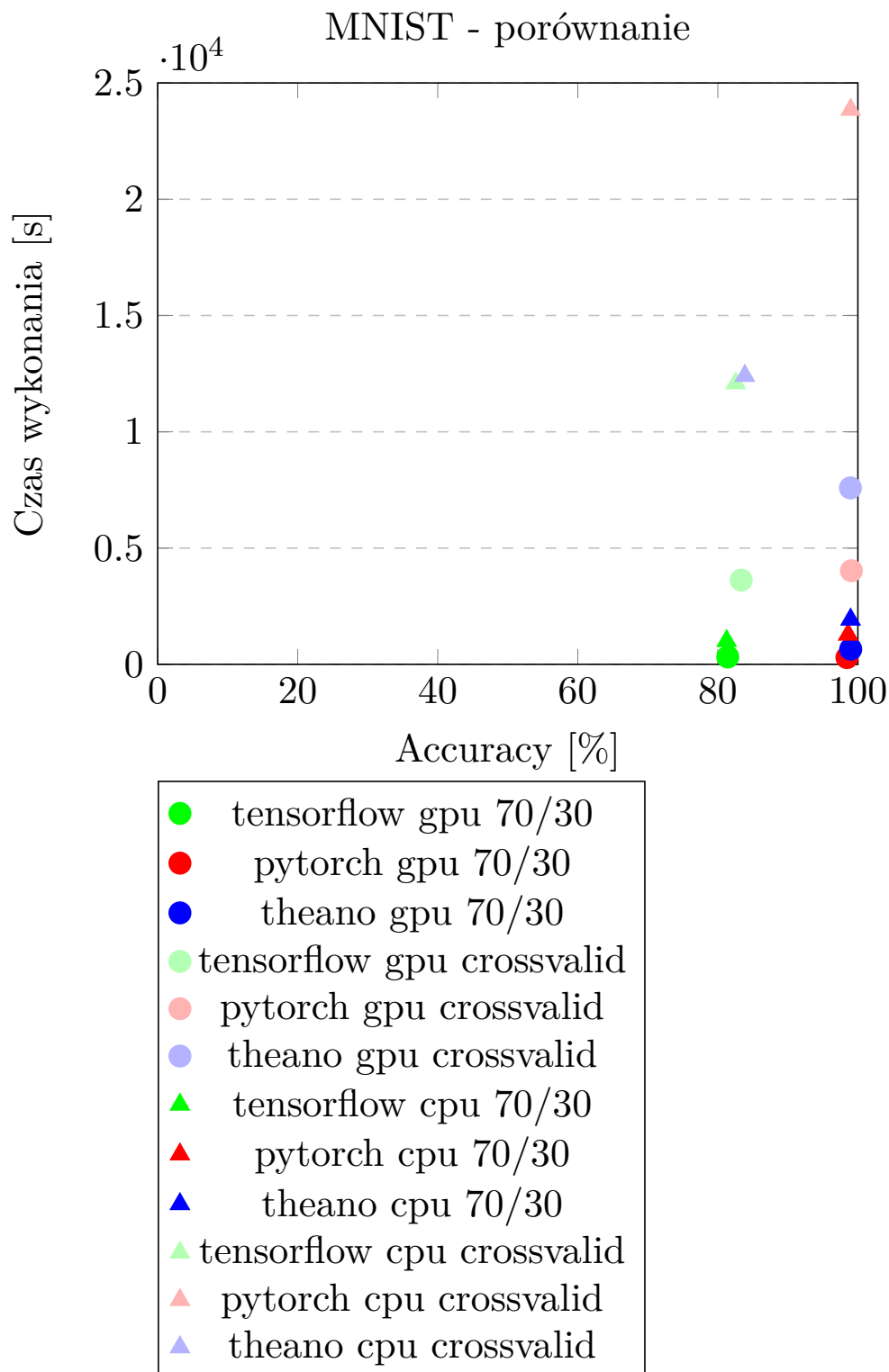
Czas wykonania

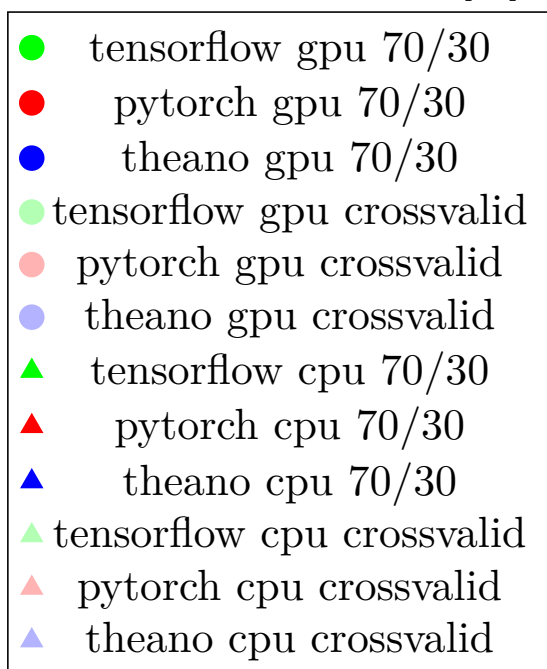
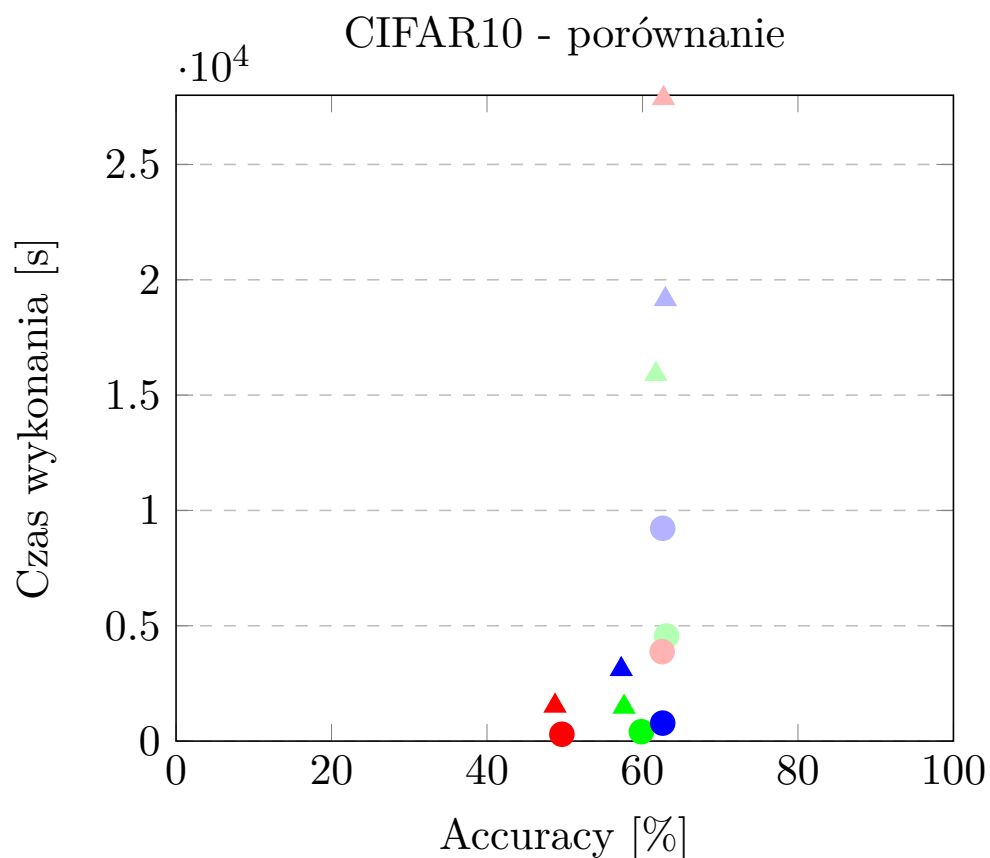
	TensorFlow	Theano	PyTorch
MNIST	318.34s	654.99s	293.51s
CIFAR10	407.71s	779.47s	297.83s
CIFAR100	377.30s	876.40s	298.36s
LetterRec	182.15s	141.96s	288.27s

Test accuracy

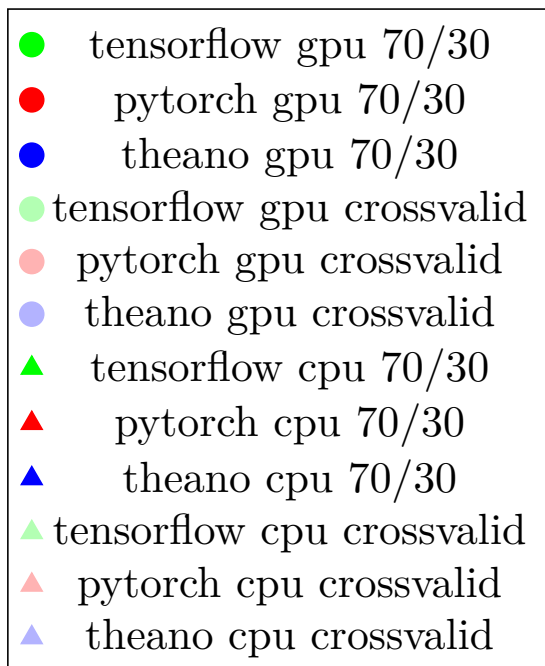
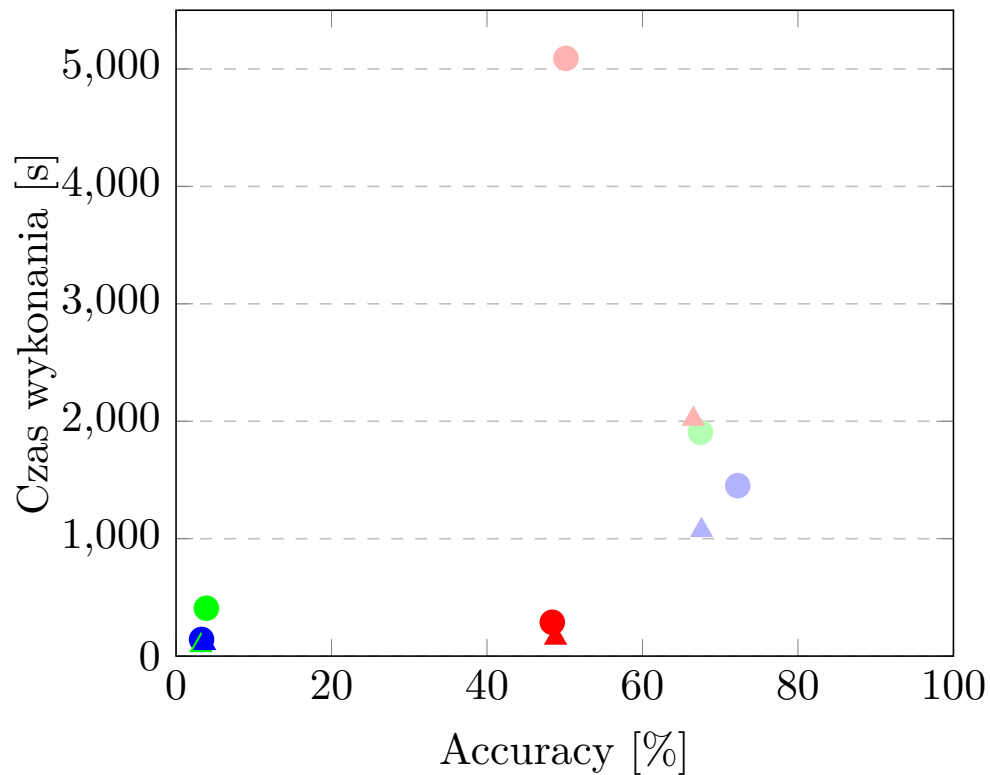
	TensorFlow	Theano	PyTorch
MNIST	81,42%	98.98%	98.44%
CIFAR10	59,86%	62.60%	49.63%
CIFAR100	38,69%	36.43%	33.16%
LetterRec	3,9%	3.28%	48.40%

2.4 Wykresy porównawcze według zbiorów





LetterRecognition - porównanie



2.5 Wnioski

Z powyższych porównań przedstawionych w tabelach i na wykresach można wyciągnąć następujące wnioski:

- Sieci neuronowe konwolucyjne, które były wykorzystane do klasyfikacji obrazów (tj. dla zbiorów MNIST, CIFAR10, CIFAR100) zdecydowanie szybciej działają jeżeli do obliczeń wykorzystywany jest GPU.
- Dla sieci wykorzystanej dla zbioru Letter-Recognition lepiej użyć CPU do obliczeń. Użycie GPU zwiększyło znacznie długość uczenia sieci. Efekt był najbardziej widoczny w przypadku użycia bibliotek pytorch i tensorflow.
- 10-krotna walidacja krzyżowa zgodnie z oczekiwaniami zwiększała czas uczenia około 10-krotnie oraz sprawiała że wynik accuracy był mniej obciążony błędami związanymi z przeuceniem sieci. Walidacja krzyżowa zazwyczaj nie zwiększa dokładności, ale dla zbioru Letter-Recognition zauważyliśmy znaczną poprawę w wynikach po jej użyciu (z około 3 do około 70 procent).
- Biblioteka Theano okazała się najwolniejsza dla sieci konwolucyjnych, natomiast najszybciej ze wszystkich testowanych bibliotek poradziła sobie z siecią dla zbioru Letter-Recognition.
- Implementacje oraz domyślne parametry niektórych funkcji, optymalizatorów, inicjalizatorów wag, klas do czytania danych itp. różnią się dla każdej biblioteki. Stąd wynikają różnice w otrzymywanych wynikach Accuracy dla każdej z bibliotek.