Raport w ramach projektu

realizowanego przez:

Snowdog sp. z o.o.

pn. „**Opracowanie innowacyjnej technologii oraz realizacja prototypu systemu analizującego obraz utrwalony techniką cyfrową przy pomocy smartfonu”**

w ramach

Wielkopolskiego Regionalnego Programu Operacyjnego na lata 2014-2020,

Oś Priorytetowa 1. Innowacyjna i konkurencyjna gospodarka,

Działanie 1.2. „Wzmocnienie potencjału innowacyjnego przedsiębiorstw Wielkopolski”.

Nr zadania	1
Poziom TRL	TRL in: 4 TRL out: 5
Numer projektu	RPWP.01.02.00-30-0175/17
Autorzy raportu	Radosław Zaworski, Bartłomiej Kwiatkowski
Czas pracy i zadania wykonywane przez pracowników na rzecz projektu	1.1. Przetestowanie istniejących architektur sieci neuronowych (styczeń - marzec 2019) 1.2. Zbieranie i przygotowanie danych (styczeń - marzec 2019) 1.3. Projekt i implementacja modułu wstępnego przetwarzania danych (styczeń - marzec 2019)
Kamienie milowe:	<u>Zakładany efekt końcowy (kamień milowy):</u> 1. Raport podsumowujący wyniki badań i rekomendacje dla proponowanego prototypu (do realizacji w etapie 2). 2. Baza obrazów potrzebna do tworzenia i testowania modeli. 3. Implementacja testowanych modeli oraz modułu wstępnego przetwarzania danych. 4. Wskaźnik rezultatu: Liczba przetestowanych architektur sieci = min. 5

	5. Wskaźnik rezultatu: Liczba przeprowadzonych eksperymentów obliczeniowych = 16 6. Wskaźnik rezultatu: Liczba pozyskanych obrazów uczących = 20000
Zgodność z Planem w zakresie prac B+R (TAK/NIE)	TAK

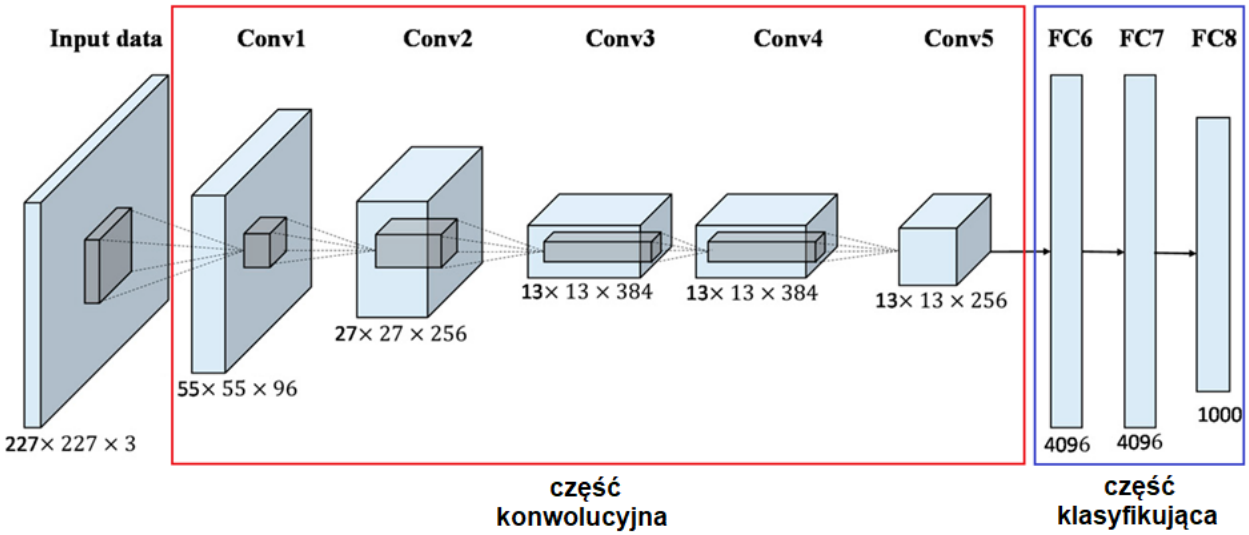
2. Przedmiot badania

Przedmiotem badania były wybrane architektury sztucznych sieci neuronowych służących do zadań:

- klasyfikacji obrazu (architektury A)
- detekcji obiektów (architektury B)
- segmentacji obrazu (architektury C)

W przypadku architektur A wykorzystano metodę "transfer learning". Metoda ta opiera się na wykorzystaniu wiedzy zdobytej podczas nauki rozwiązywania jednego problemu do nowego problemu.[1] Jest to bardzo popularna technika w analizie obrazu z wykorzystaniem sieci neuronowych. Z zasady sieci neuronowe wymagają dużej ilości danych do nauki, a im bardziej rozbudowana jest architektura tym więcej danych jest potrzebne by nie doszło do zjawiska przeuczenia (ang. overfitting).[2] Większość popularnych architektur sieci służących do klasyfikacji można podzielić na dwie części:

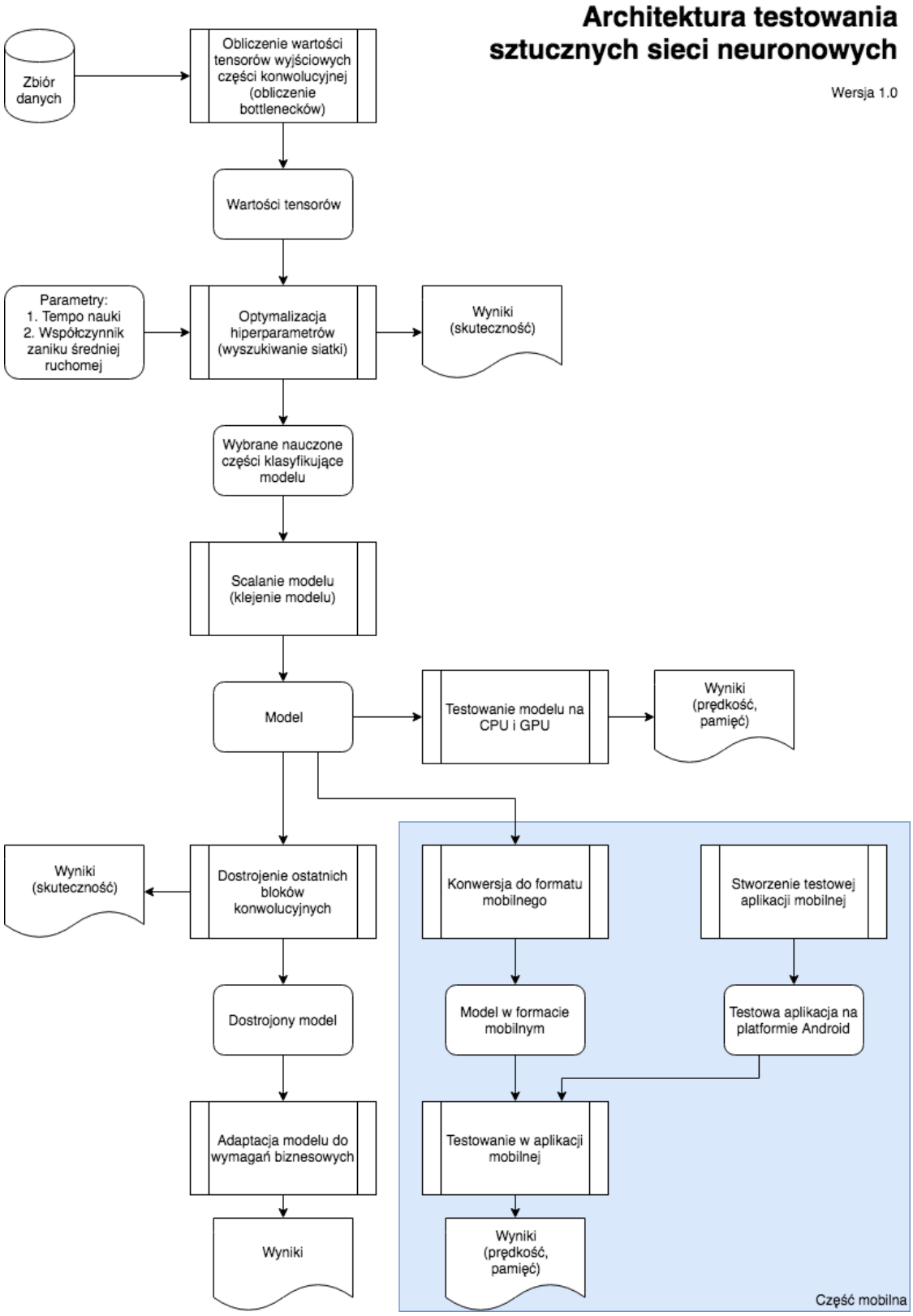
- część konwolucyjna
- część klasyfikująca



Rysunek 1. Podział architektury na części na przykładzie modelu AlexNet

Część konwolucyjna odpowiedzialna jest za rozpoznawanie cech, kolejne warstwy "uczą się postrzegać" coraz bardziej skomplikowane właściwości obrazu, zazwyczaj obrazuje się to w taki sposób, że pierwsza warstwa może identyfikować pojedyncze krawędzie, następna krzywe, kolejna kształty itd. Z kolei w ostatnich warstwach może to być np. oko, liść, sylwetka pojazdu. Tą właściwość można wykorzystać w transfer learningu, ponieważ wiele cech nauczonych przez warstwy w sieci neuronowej można wykorzystać z powodzeniem w innym zadaniu.

W praktyce najczęściej spotyka się technikę nauki całego modelu na dużym zbiorze danych takim jak ImageNet [3], a następnie pozbyciu się części klasyfikującej i zastąpieniu jej nową, wytrenowaną na danych z nowego problemu. W ten sposób znacznie zmniejszamy liczbę parametrów sieci neuronowej do nauki nowego zagadnienia, np. dla zadania klasyfikacji 1000 klas (tyle klas wykorzystuje się podczas nauki na zbiorze ImageNet) używając modelu InceptionV3 zmniejszamy ją z 23817352 do 2049000, czyli pozostaje niecałe 9% początkowej liczby parametrów. Mniejsza ilość parametrów pozwala nie tylko na użycie mniejszej ilości danych, ale może też istotnie przyspieszyć proces nauki.



Rysunek 2. Architektura testowania modeli A

Do badania wybrano 3 architektury A: VGG16, InceptionV3 oraz ResNet50, posłużono się wagami nauczonymi na zbiorze ImageNet i pozbyto się warstw klasyfikacyjnych. W przypadku VGG16 blok klasyfikujący został znacznie zmniejszony by umożliwić naukę na nowym zbiorze danych, mianowicie w dwóch warstwach w pełni połączonych zmniejszono ilość neuronów z 4096 do 100. Do nauki wykorzystano specjalnie pozyskany na potrzeby tego projektu zbiór danych, który następnie podzielono na 3 podzbiory: trenujący, walidacyjny oraz testowy w proporcjach odpowiednio 70%, 20% 10%. Jako optymalizator wybrany został algorytm RMSprop [4]. Do optymalizacji hiperparametrów użyto techniki wyszukiawania siatki (ang. grid search) dla wartości learning rate kontrolującej tempo nauki (im większy learning rate tym szybsza nauka) oraz parametru ρ (rho), czyli współczynnika zaniku średniej ruchomej, który stopniowo zmniejsza tempo nauki (im większy tym tempo spada wolniej). Wybrano 4 wartości learning rate oraz ρ w otoczeniu wartości domyślnych i zalecanych:

- learning rate: 0,0001, 0,001, 0,01, 0,1
- ρ (rho): 0,8, 0,9, 0,95, 0,99

(Domyślna wartość learning rate dla RMSprop to 0,001, a ρ to 0,9 [5])

Długość procesu nauki ograniczono do 50 epok. W celu przyspieszenia badań, wartości tensorów wyjściowych części konwolucyjnej sieci zostały obliczone tylko raz dla każdej architektury i zapisane, a następnie wykorzystane jako wejście do nauki nowej części klasyfikującej.

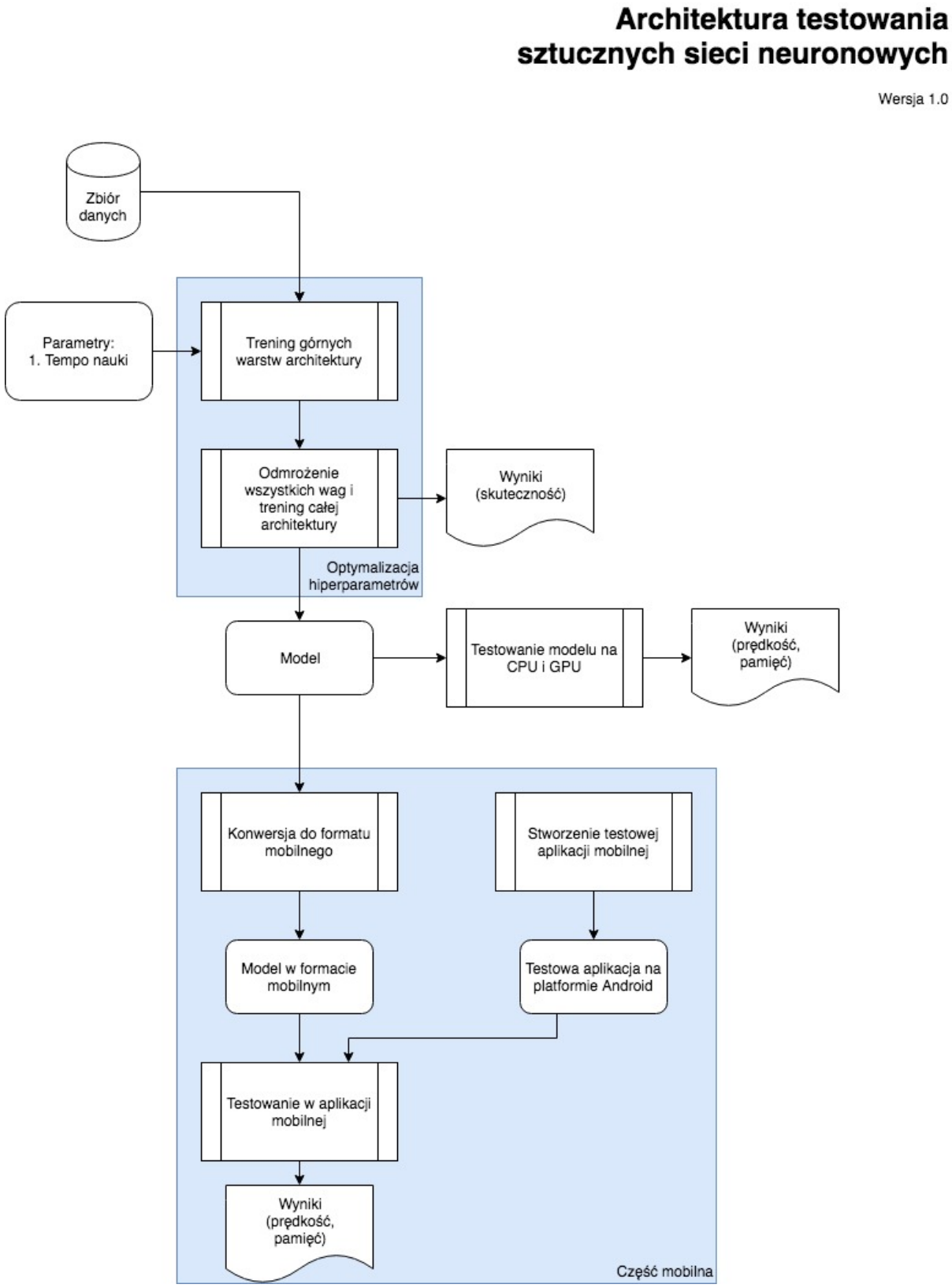
Dodatkowo podczas badania przeprowadzono dodatkowo tzw. fine-tuning, czyli dostrojenie ostatnich bloków konwolucyjnych sieci do nowego zadania. Jak wcześniej wspomniano, im dalej od wejścia sieci, tym bardziej rozbudowane cechy są identyfikowane. Kiedy podstawowe cechy takie jak krawędzie, proste kształty czy tekstury materiału są bardzo uniwersalne, tak bardziej zaawansowane są mocniej związane z pierwotnym zadaniem, więc strojąc je można poprawić jakość klasyfikacji. Strojenie takie polega na uczeniu głębszych warstw architektury po uprzednim wytrenowaniu warstw występujących wyżej z bardzo małym learning rate i wykorzystaniem optymalizatora nieadaptacyjnego takiego jak np. Stochastic Gradient Descent.

Następnym etapem badania było douczenie modelu do zmienionych wymagań klienta biznesowego - poszerzenie asortymentu produktów o dodatkowe X kategorii. Wykonano to analogicznie do pierwszego punktu badania za pomocą transfer learningu warstw klasyfikacyjnych.

Aplikacja do testowania wydajności wykorzystuje bibliotekę TensorFlow Mobile do sprawdzenia statystyk podczas działania sieci. Dla danej architektury model jest wgrywany na pamięć zewnętrzną urządzenia, zamiast do assetów aplikacji, by przyśpieszyć instalację. By przetestować szybkość działania sieci na jej wejście jest podawany biały dwuwymiarowy szum o rozmiarze równym wymaganemu rozmiarowi warstwy wejściowej. Wygenerowany szum można zobaczyć w oknie po prawej na dole.

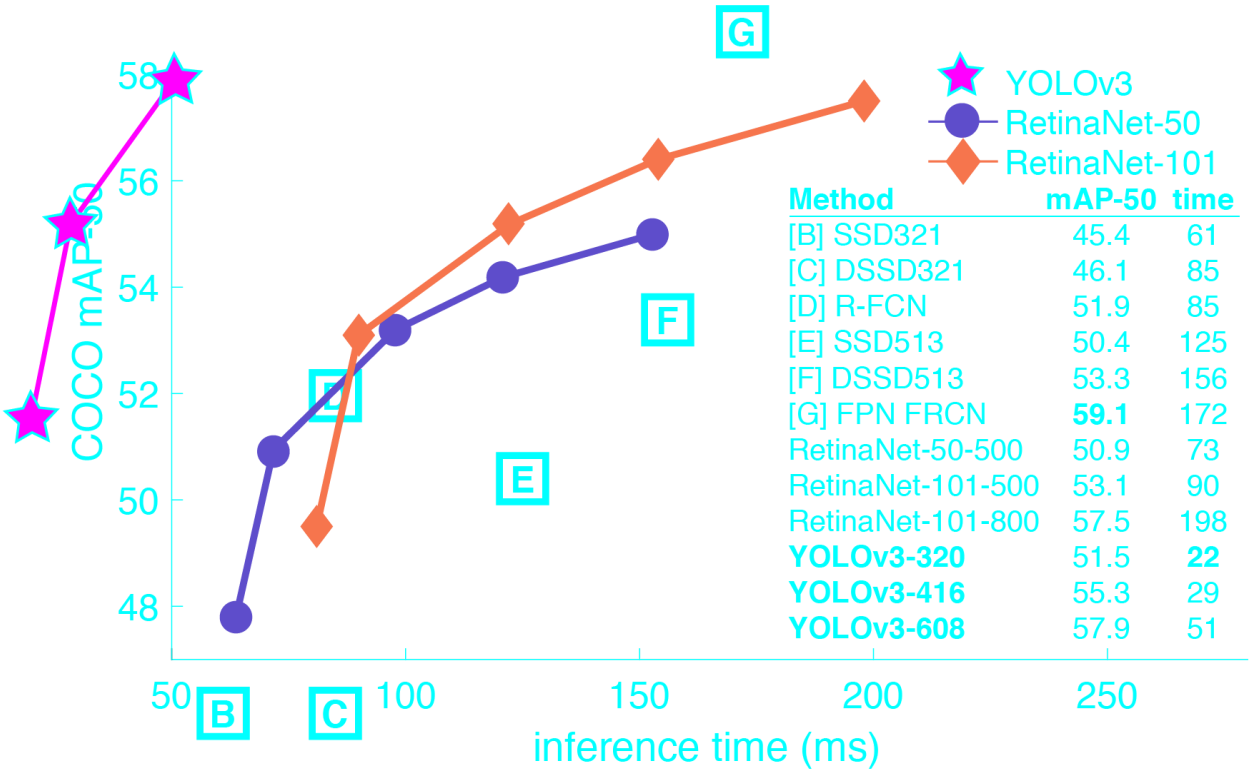
Rysunek 3. Zrzut ekranu z aplikacji mobilnej

Rozmiar warstwy wyjściowej jest pobierany z architektury. W kolejnym kroku uruchomiona sieć przetwarza wprowadzone dane. Rezulatat sieci jest wyświetlany wraz ze statystykami w aplikacji. Podczas działania biblioteki są obliczane statystyki dotyczące działania sieci, które po każdym uruchomieniu są pobierane z biblioteki i zapisywane w pamięci zewnętrznej urządzenia w formie pliku tekstowego. Statystyki zawierają takie informacje jak: czas działania każdej warstwy, czas działania sieci, wykorzystywaną pamięć, ilość odwołań do każdej warstwy, liczbę użytych warstw czy 10 najdłużej działających operacji. Aplikacja umożliwia także wielokrotne powtarzanie testów, aby uzyskać większą pewność wyników.



Rysunek 3. Architektura testowania modeli B

Do zadania detekcji obiektów zostało wybrane YoloV3 , ponieważ ta architektura sieci neuronowych należy do jednych z najszybszych systemów detekcji obiektów o bardzo wysokiej dokładności.



Rysunek 4. Porównanie architektur do detekcji obiektów, źródło: https://pjreddie.com/darknet/yolo/

Model ten występuje w kilku podstawowych wersjach różniących się rozmiarem wejścia oraz ilością parametrów:

- YOLOv3-320
- YOLOv3-416
- YOLOv3-608
- YOLOv3-tiny
- YOLOv3-spp

W eksperymencie zdecydowano użyć się wariantu YOLOv3-320 jako kompromisu między kosztownością obliczeniową a skutecznością. W przyszłości w ramach potrzeb można wybrać szybszy lub dokładniejszy podtyp.

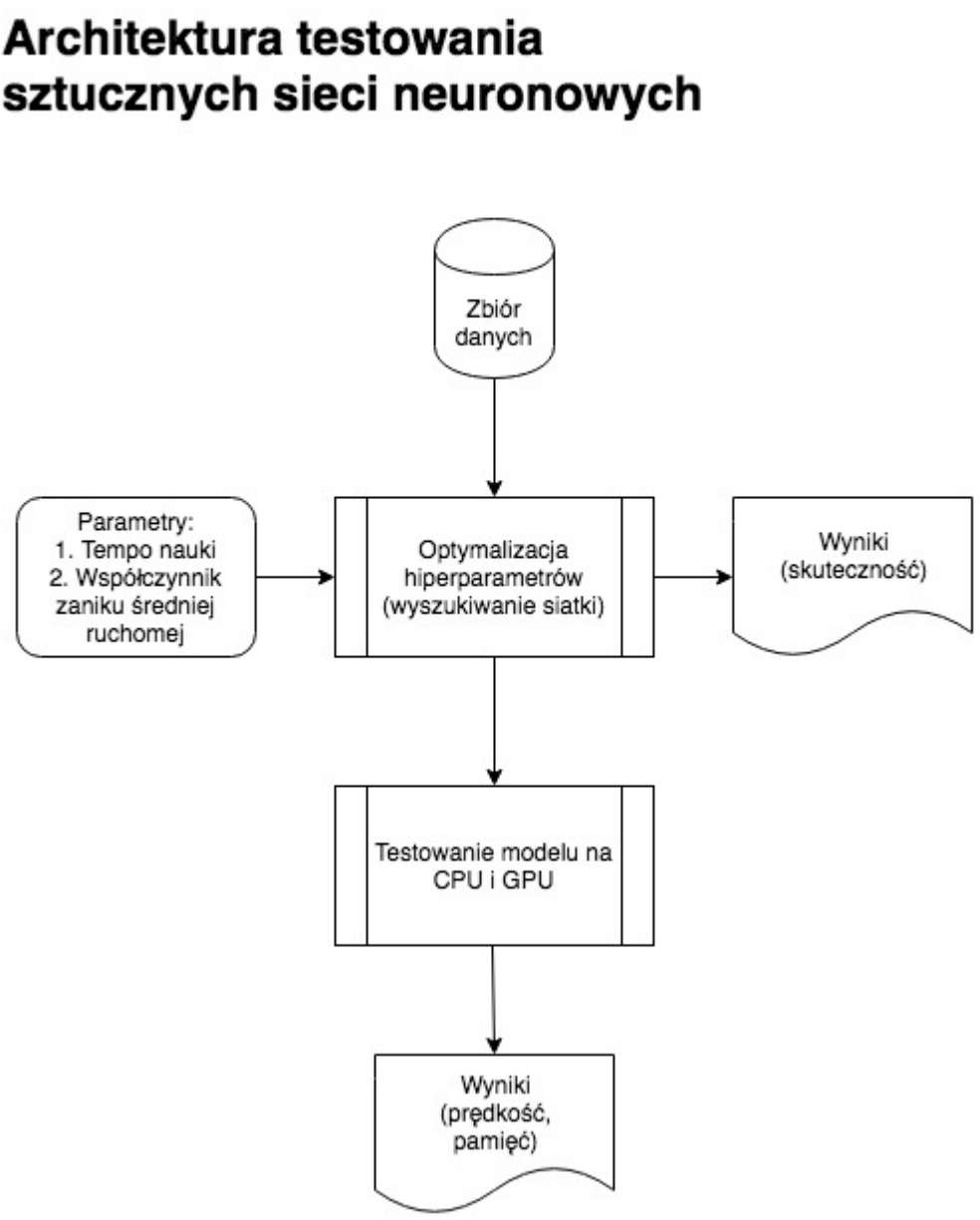
Zbiór Open Images Dataset V4 składa się z około 9 milionów zdjęć na licencji CC BY 2.0 wraz z ramkami ograniczającymi (ang. bounding boxes) które licencjonowane są na podstawie CC BY 4.0. Licencje te pozwalają wykorzystać te dane w zastosowaniach komerycjnych. Do nauki wykorzystano klasę 'watch' na którą składa się 1903 ramek ograniczających na 1434 zdjęciach.

Strategia nauki była bardzo podobna do tej przy architekturach A - zastosowano technikę transfer learning. Na potrzeby tej metody wykorzystano wagi pobrane z oficjalnej strony projektu nauczone na

zbiorze COCO[7]. W początkowej fazie 50 epok (I faza) poza warstawmi: conv2d_59, conv2d_67, conv2d_75, yolo_loss wszystkie wagi zostały zamrożone. Kolejny etap nauki to dostrajanie wszystkich warstw z użyciem tego samego optymalizatora z odpowiednio mniejszymi wartościami learning rate (II faza). Jako optymalizator wybrano algorytm Adam dla którego zbadano 4 arbitralnie dobrane pary wartości parametrów learning rate:

- I faza: 0.01, II faza: 0.001
- I faza: 0.001, II faza: 0.0001
- I faza: 0.0001, II faza: 0.00001
- I faza: 0.00001, II faza: 0.000001

Testowanie na platformie mobilnej odbyło się podobnie jak w poprzednim przypadku.



Rysunek 5. Architektura testowania modeli C

Do badania możliwości segmentacji obrazu wybrano głęboki konwolucyjny enkoder-dekoder SegNet. Ze względu na kosztowność utworzenia odpowiedniego datasetu, oraz mniejszy wybór dostępnym specjalistycznych datasetów do wstępnych badań wykorzystano zbiór CamVid na który składają się materiały wideo z perspektywy jadącego pojazdu i posiada 32 klasy semantyczne. Tutaj również wykorzystano wariant transfer learningu, wagi modelu zostały zainicjowane wartościami z modelu VGG16 wytrenowanego na zbiorze ImageNet, następnie cała sieć została poddana nauce z zastosowaniem optymalizatora Stochastic Gradient Descent z 4 różnymi wartościami learning rate.

W przypadku tej architektury testowanie na platformie mobilnej okazało się niemożliwe. Wynika

to z tego, że TensorFlow Mobile jest dostosowana do potrzeb systemu Android, mniej wydajnych urządzeń z mniejszą ilością pamięci operacyjnej. Dodatkowo urządzenia mobilne zazwyczaj, w odróżnieniu od komputerów osobistych, wykorzystują procesory operate na architekturze ARM. By sprostać tym wymaganiom TensorFlow Mobile został zoptymalizowany poprzez ograniczenie zbioru dostępnych operacji. W przypadku SegNet problematyczna okazała się warstwa 'MaxPoolingWithArgmax2D', która do pracy potrzebuje operacji 'ScatterNd' nieznajdującej się w wersji mobilnej biblioteki TensorFlow. Środkiem zaradczym może być przeniesienie obliczeń na serwer, wybranie innej architektury lub samodzielna kompilacja biblioteki.

3. Wynik działania

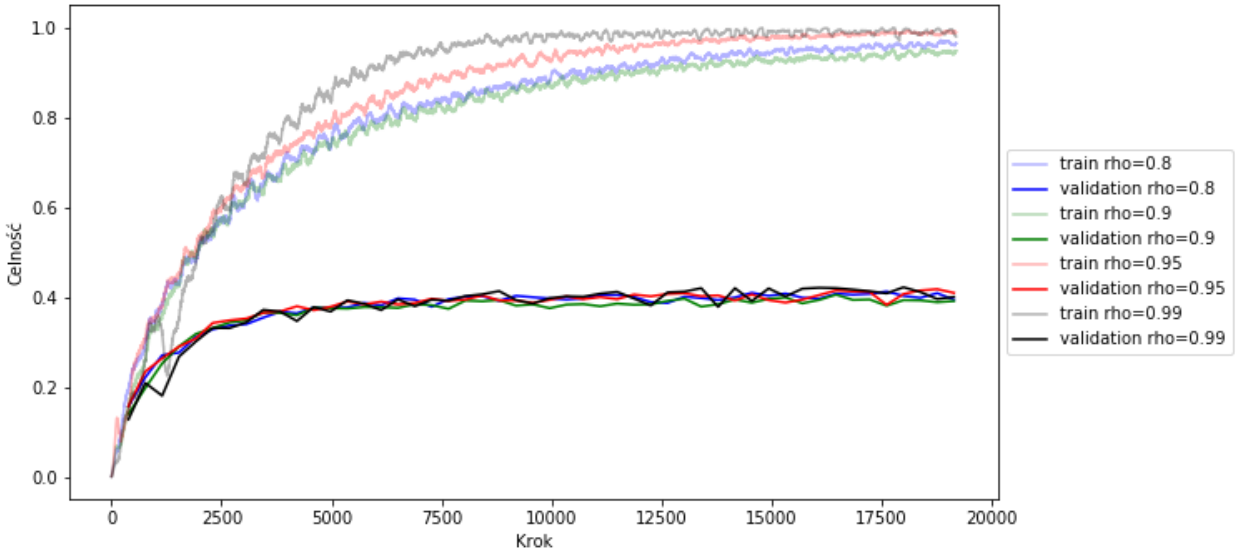
3.1. Badania architektur A

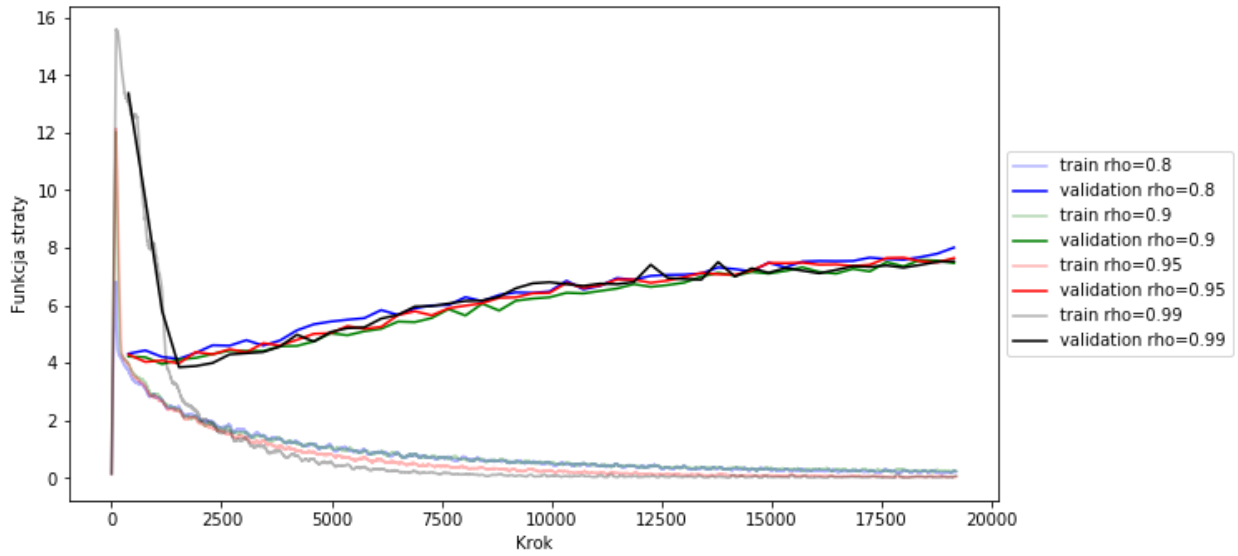
3.1.1. Optymalizacja hiperparametrów

Wykresy poniżej przedstawiają wyniki sieci podczas procesu nauki. Punkty pomiarowe dla danych treningowych są pobierane po każdym kroku nauki, na który składa się partia 32 próbek i zostały wykreślone bladym odcieniem koloru. Dodatkowo dla czytelności krzywa została wygładzona za pomocą filtra z pakietu SciPy - `scipy.signal.lfilter`. Walidacja odbywała się na koniec każdej epoki i z wykorzystaniem całego podzbioru do tego celu przeznaczonego.

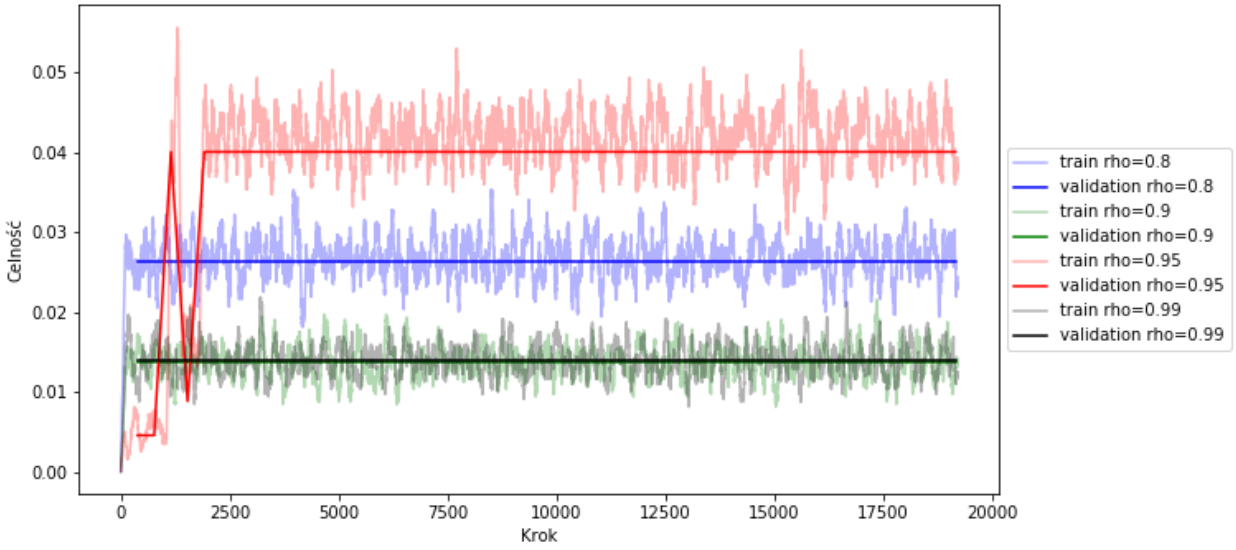
3.1.1.1. VGG16

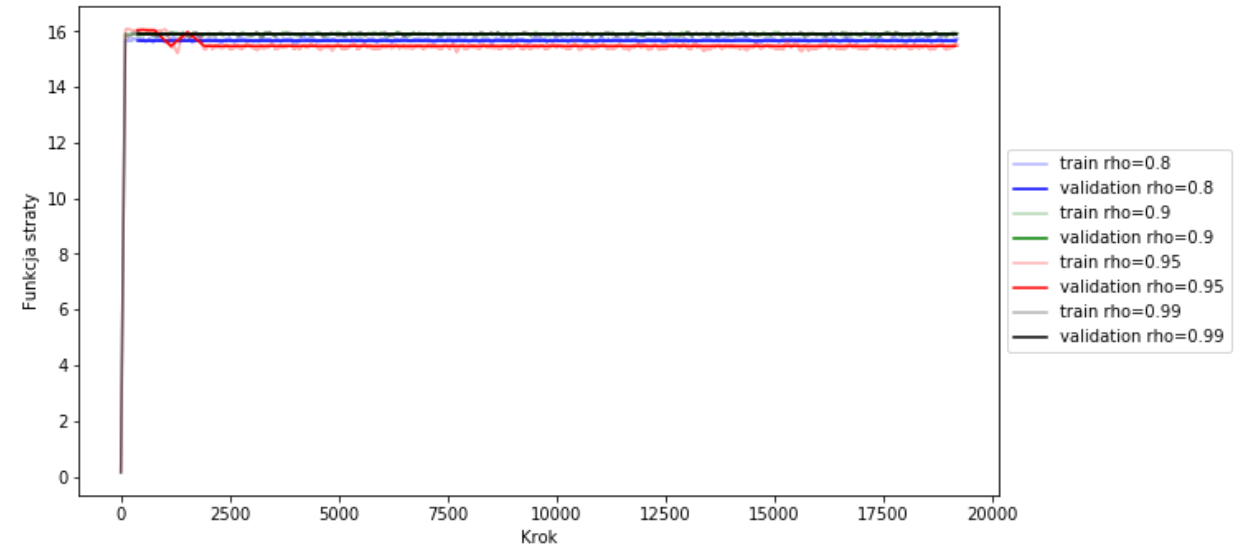
- lr = 0.0001



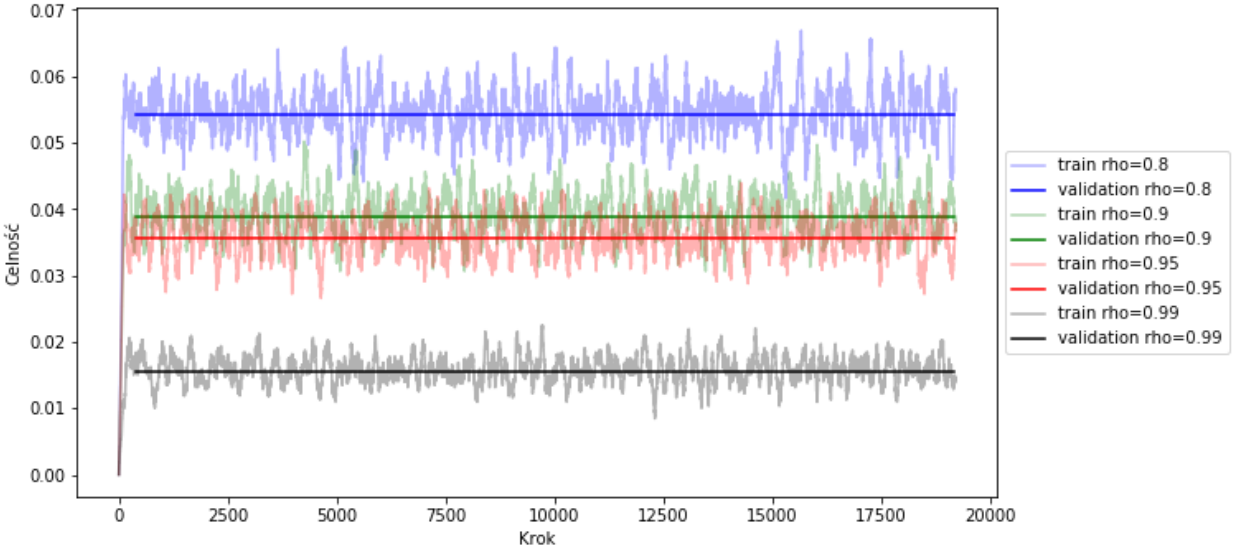


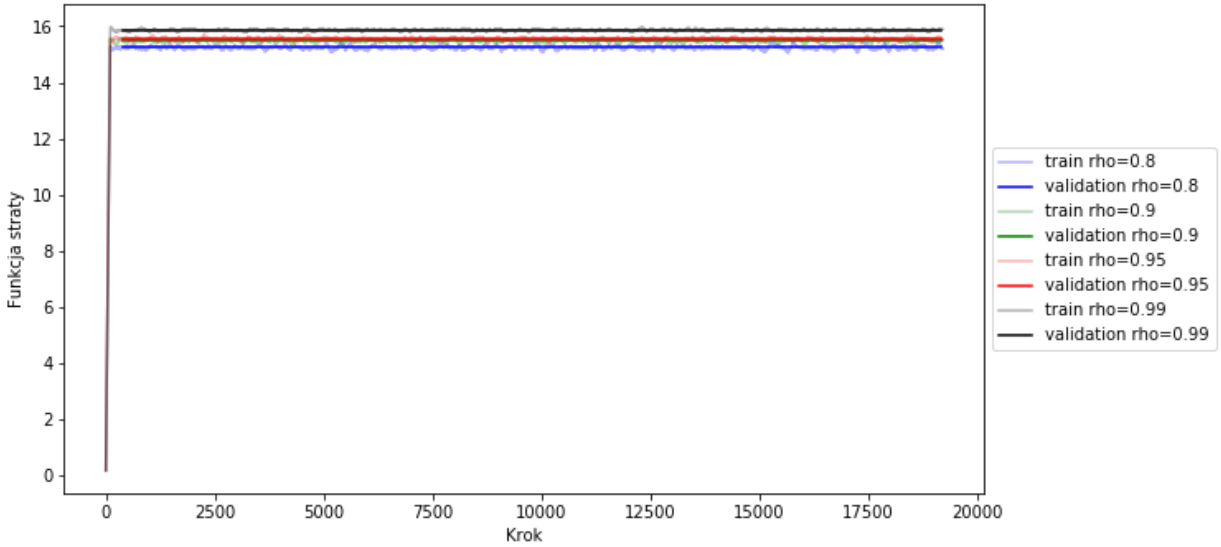
- lr = 0.001


Celność
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

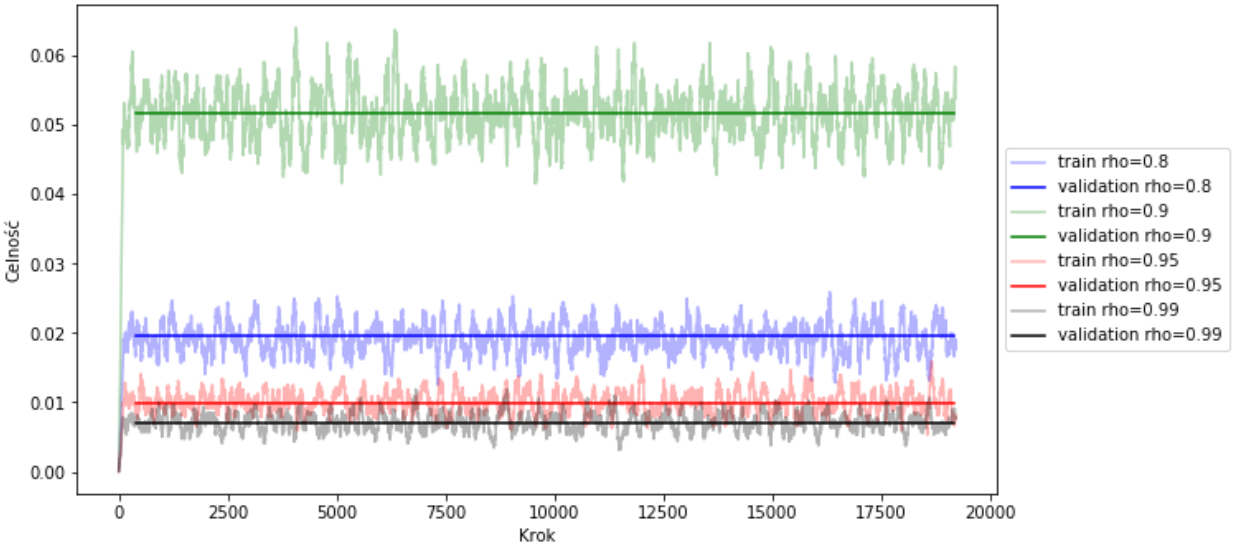

Funkcja straty
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

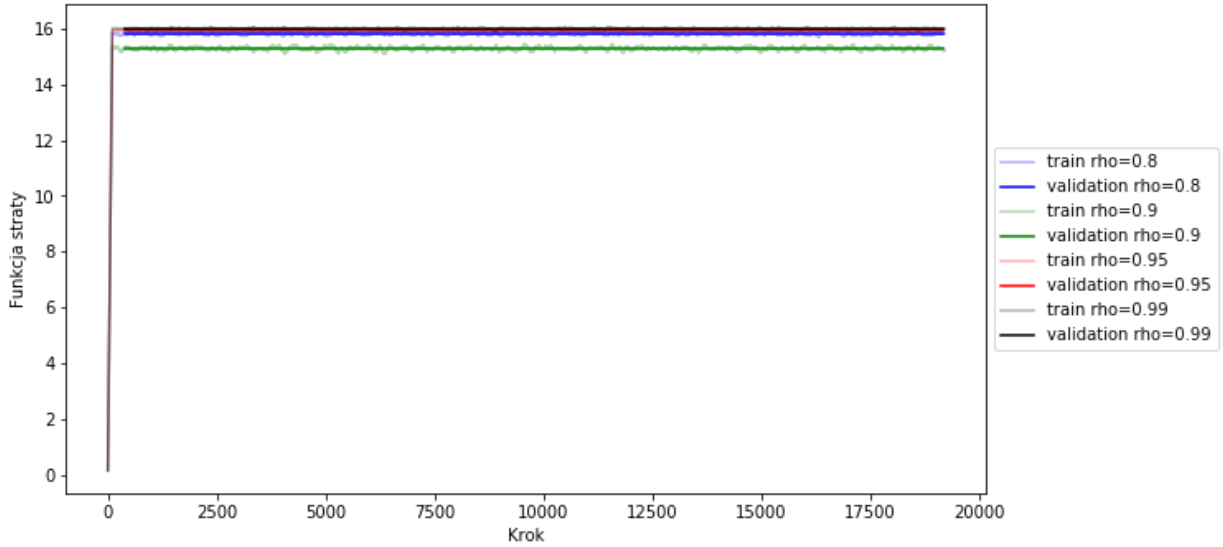
- lr = 0.01


Celność
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99


Funkcja straty
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

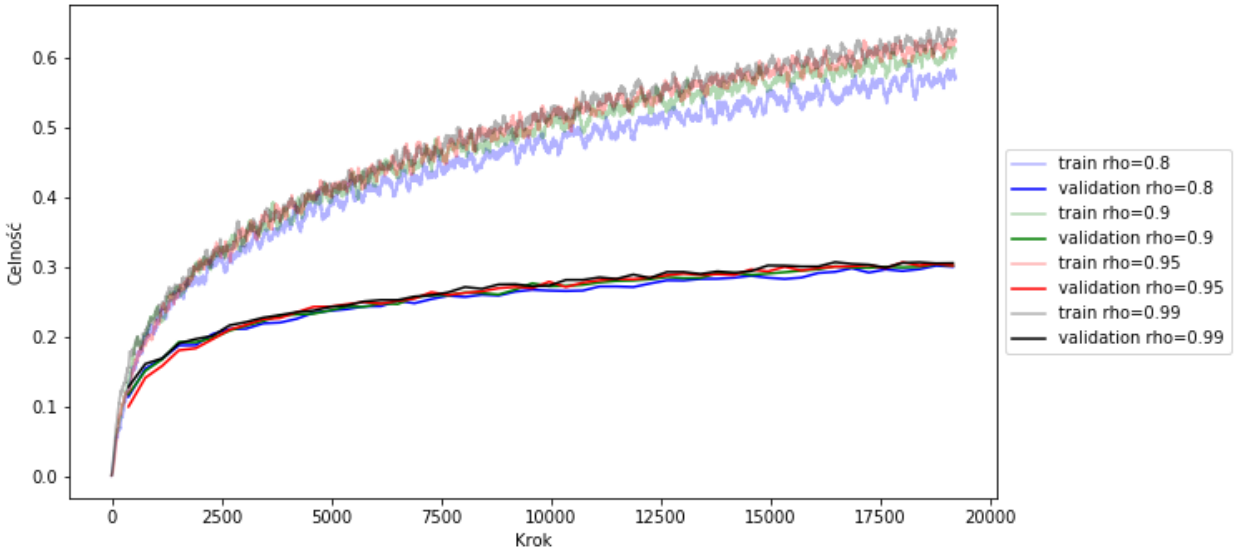
- lr = 0.1

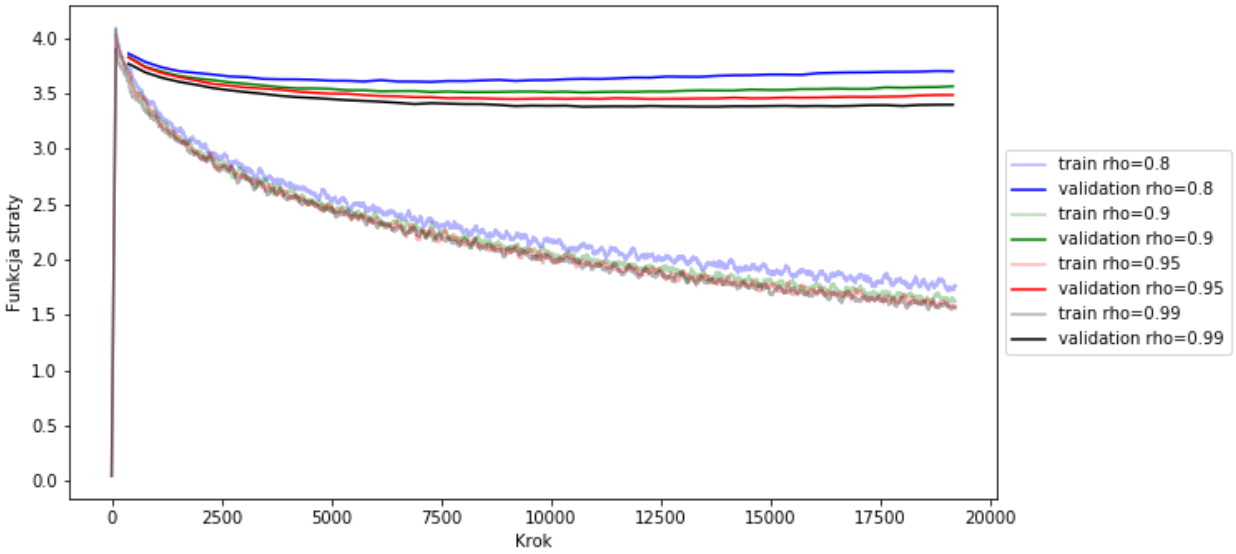

Celność
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99


Funkcja straty
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

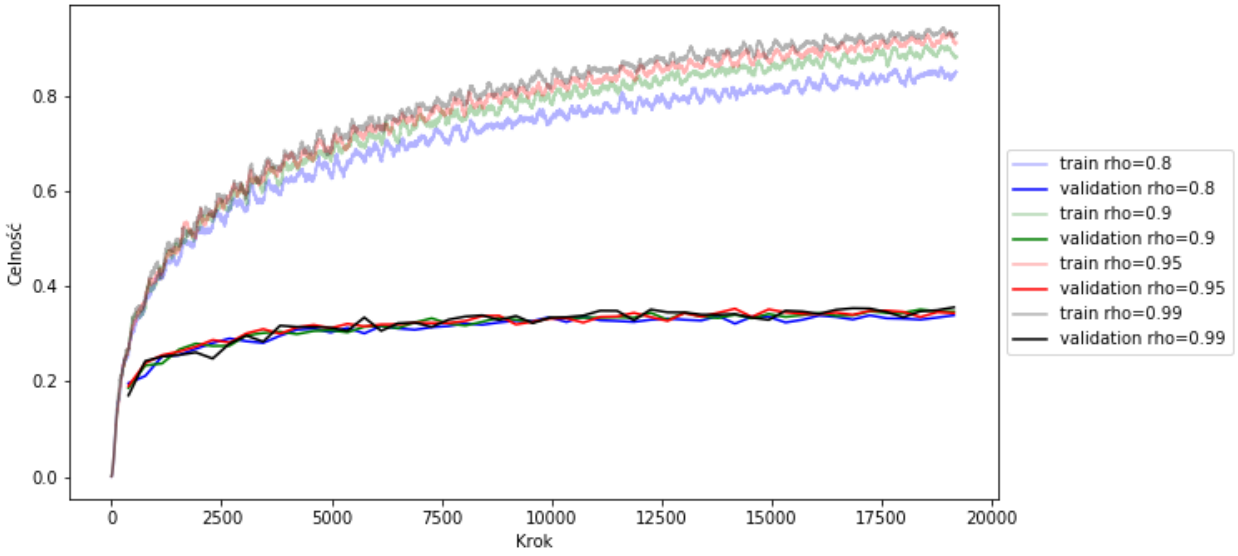
3.1.1.2. InceptionV3

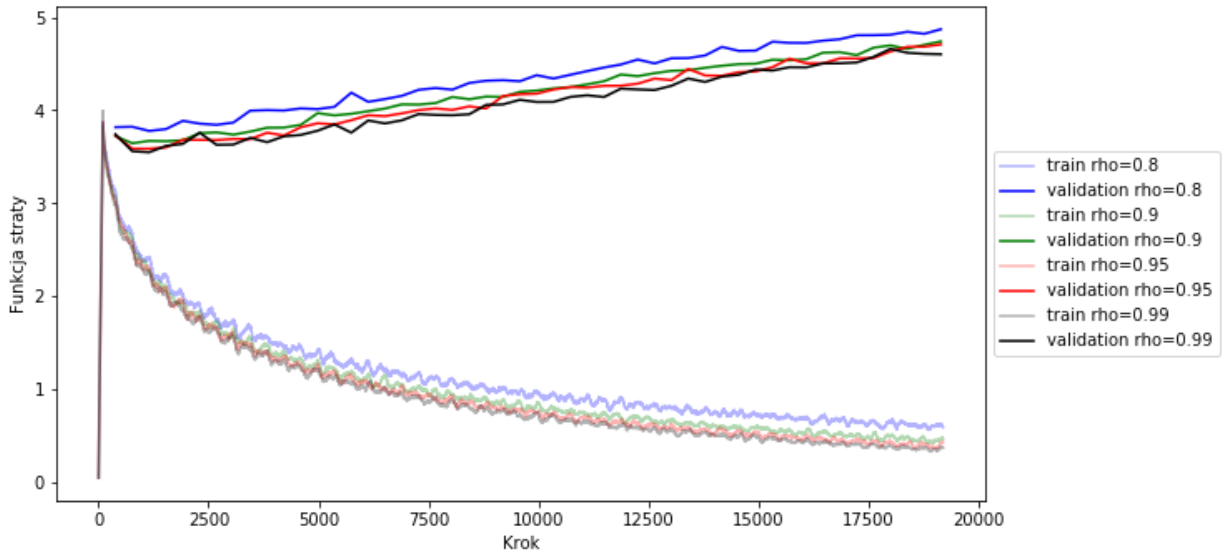
- lr = 0.0001



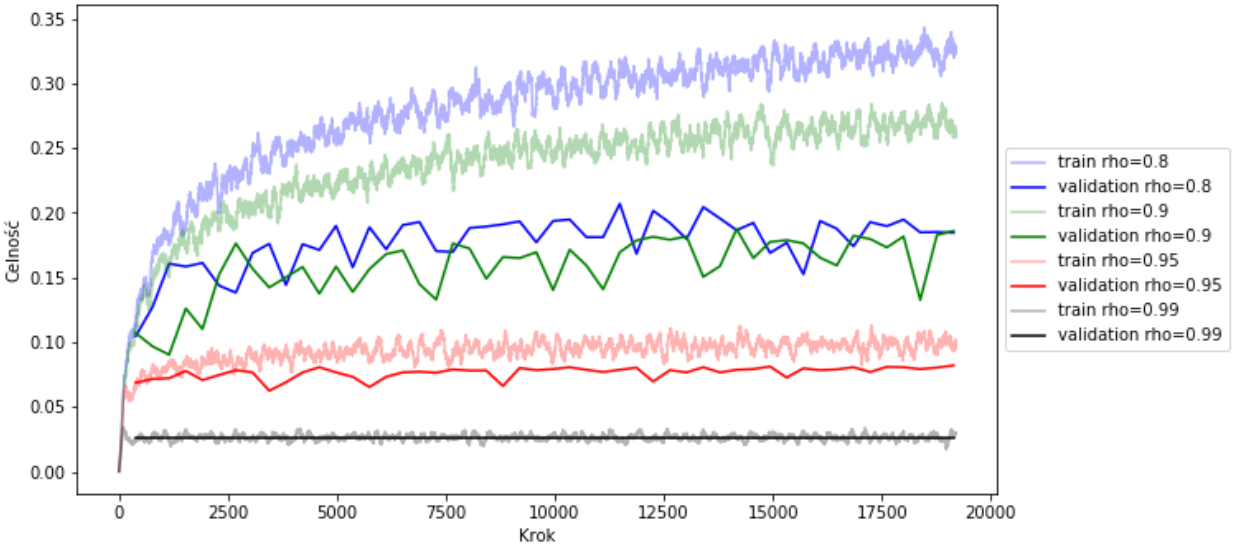


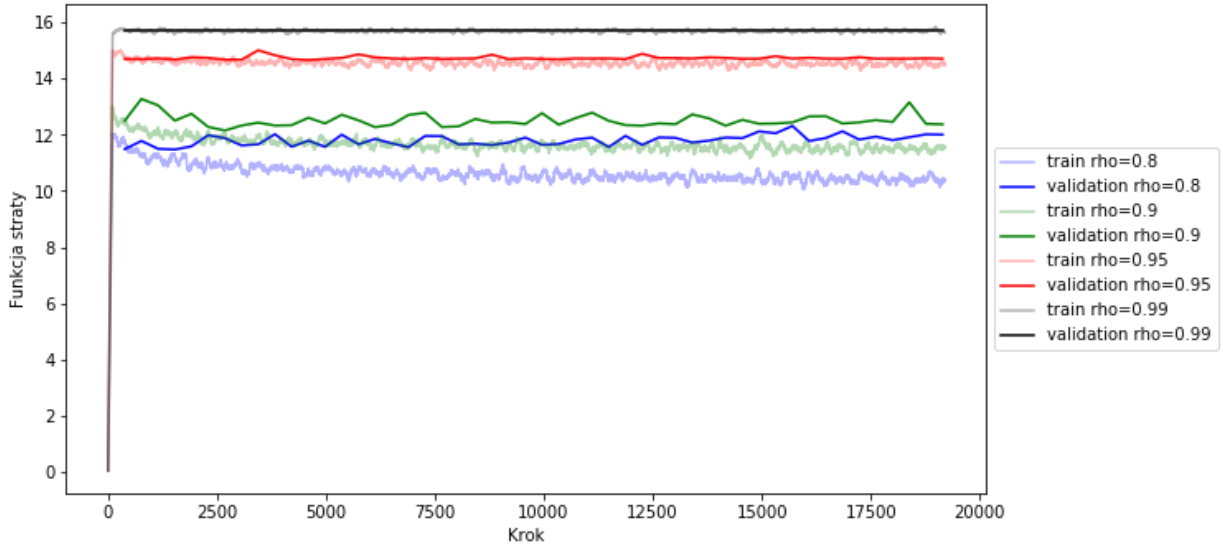
- lr = 0.001


Celność
Krok
0.0
0.2
0.4
0.6
0.8
0
2500
5000
7500
10000
12500
15000
17500
20000
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

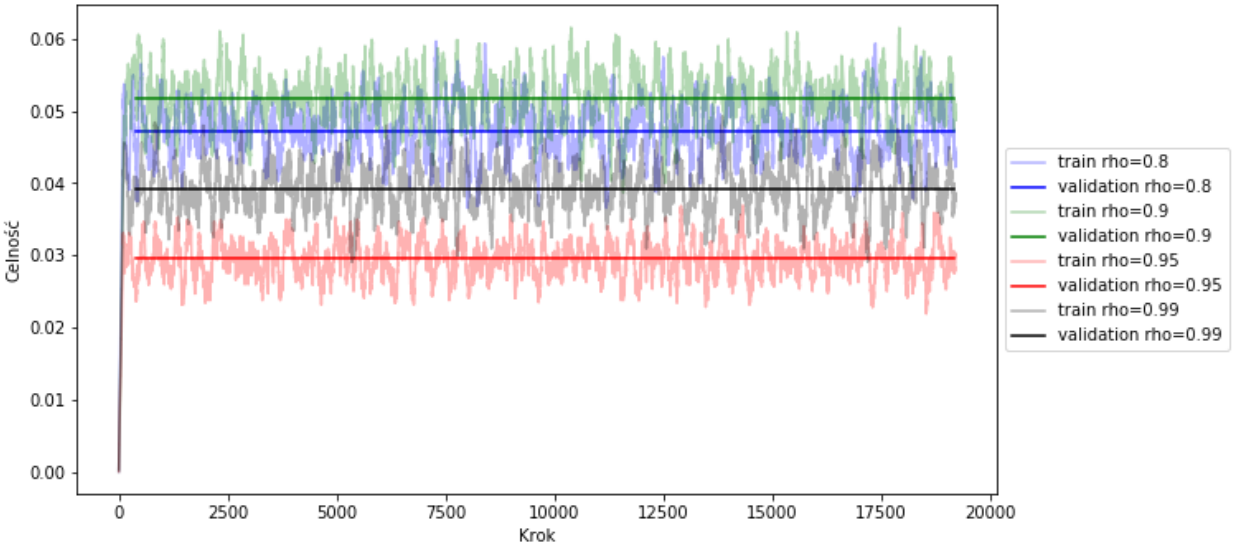

Funkcja straty
Krok
0
1
2
3
4
5
0
2500
5000
7500
10000
12500
15000
17500
20000
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

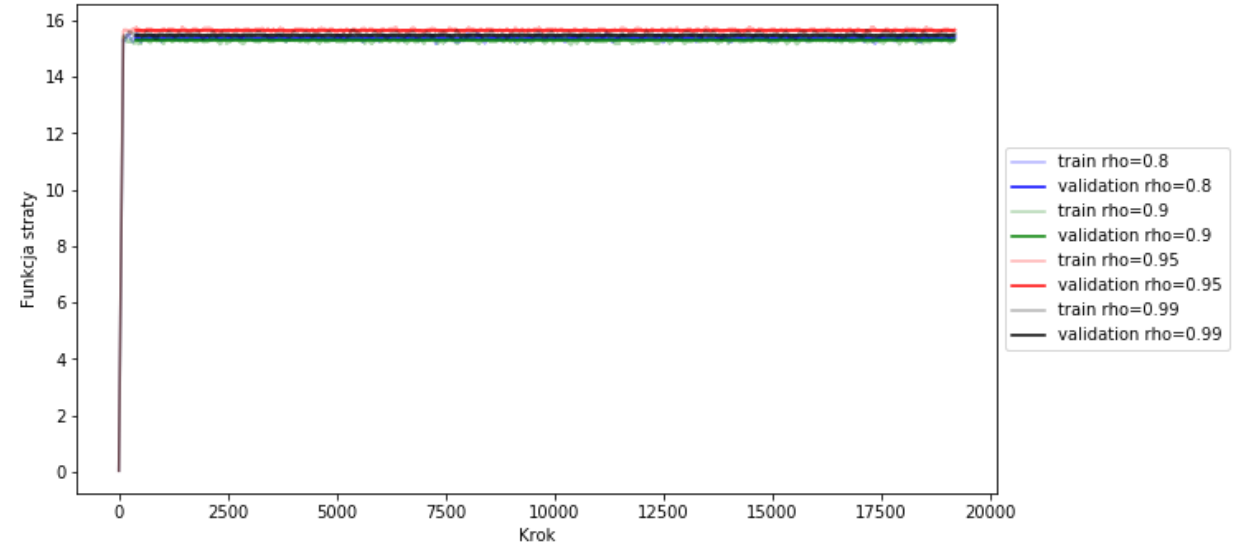
- lr = 0.01


Celność
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99


Funkcja straty
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

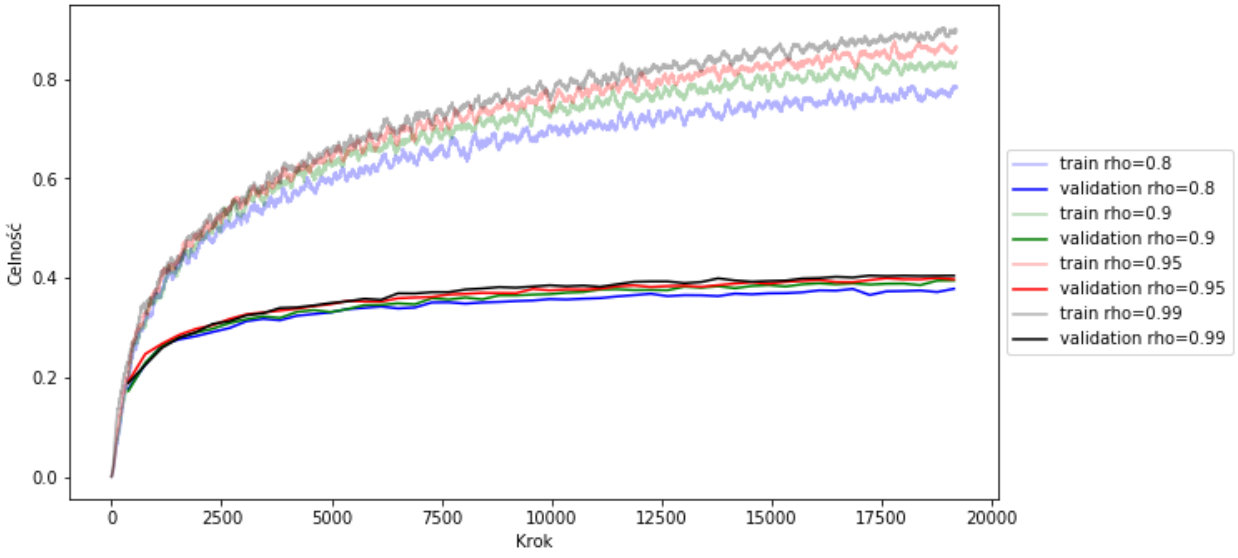
- lr = 0.1

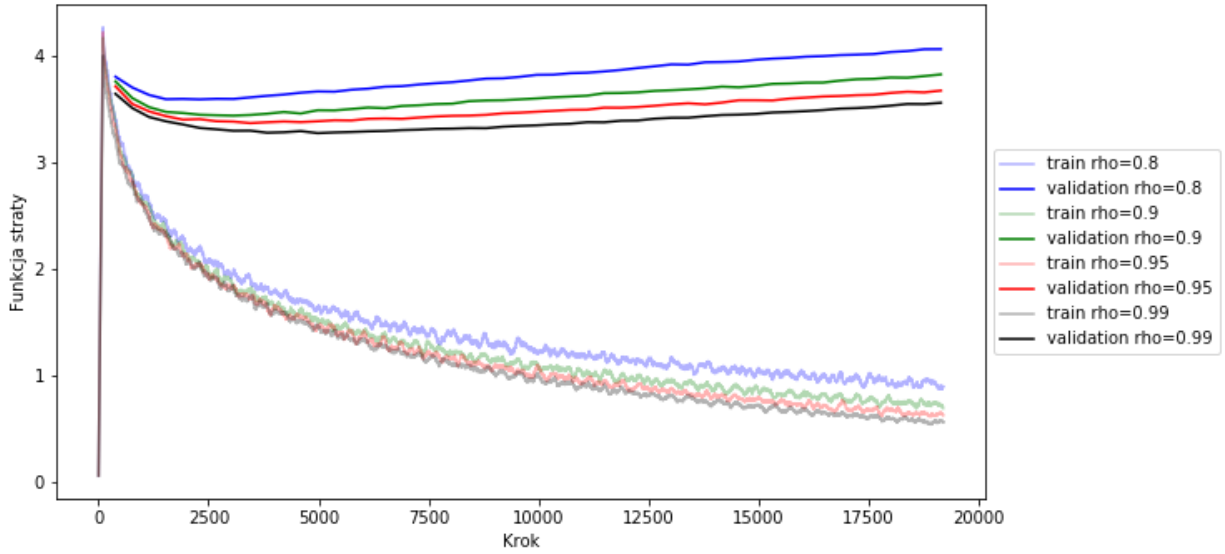

Celność
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99


Funkcja straty
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

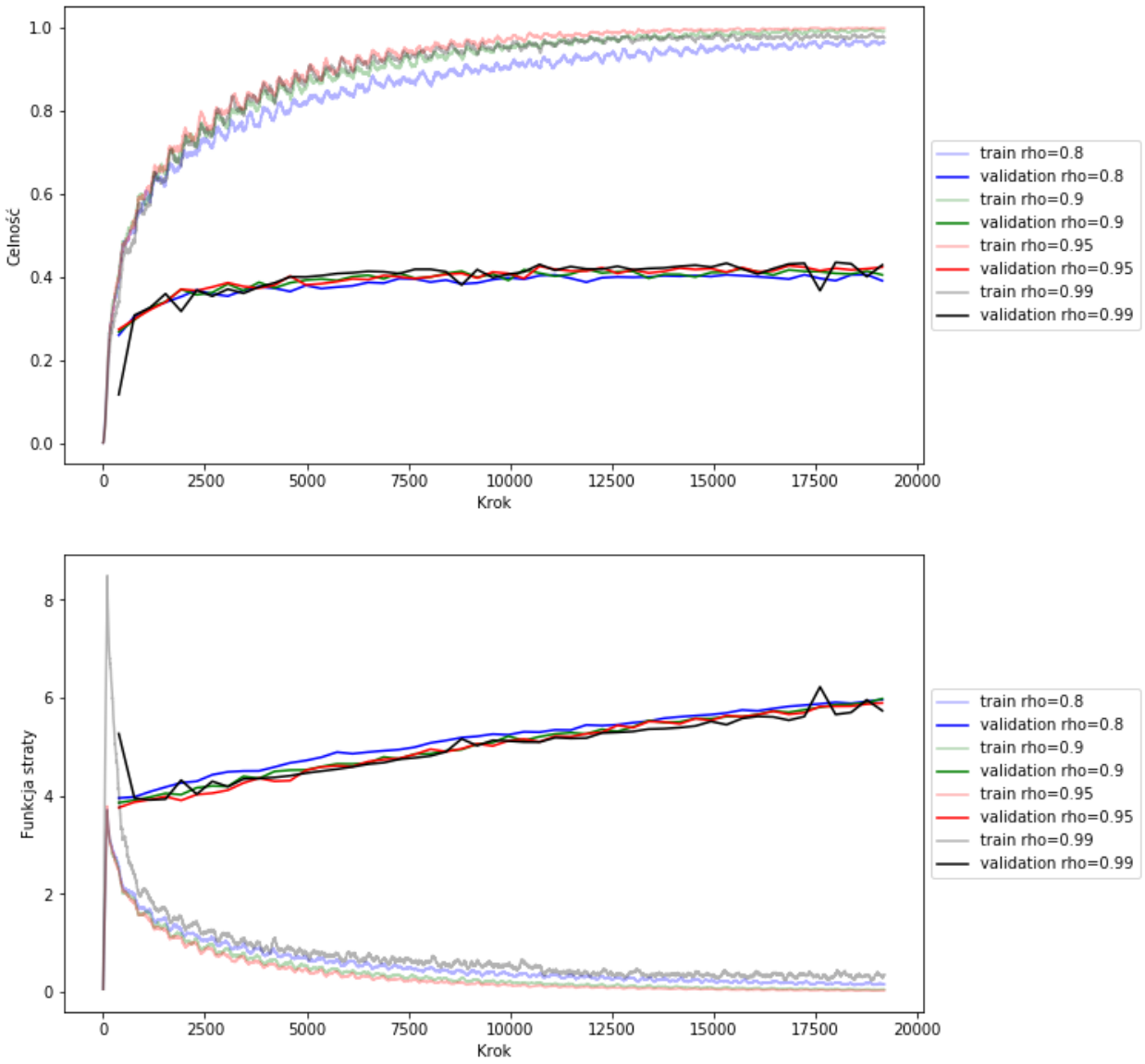
3.1.1.3. ResNet50

- lr = 0.0001

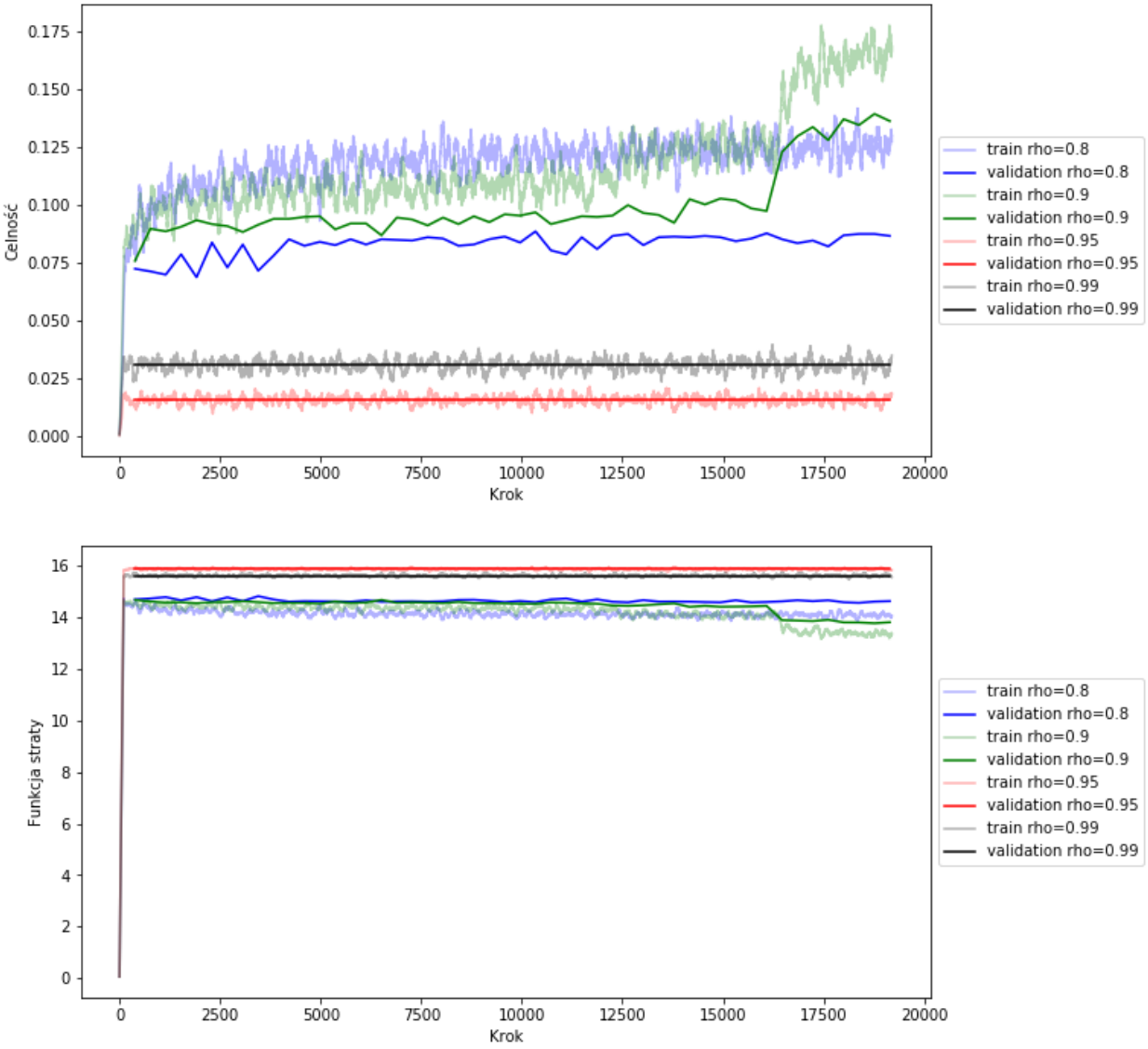




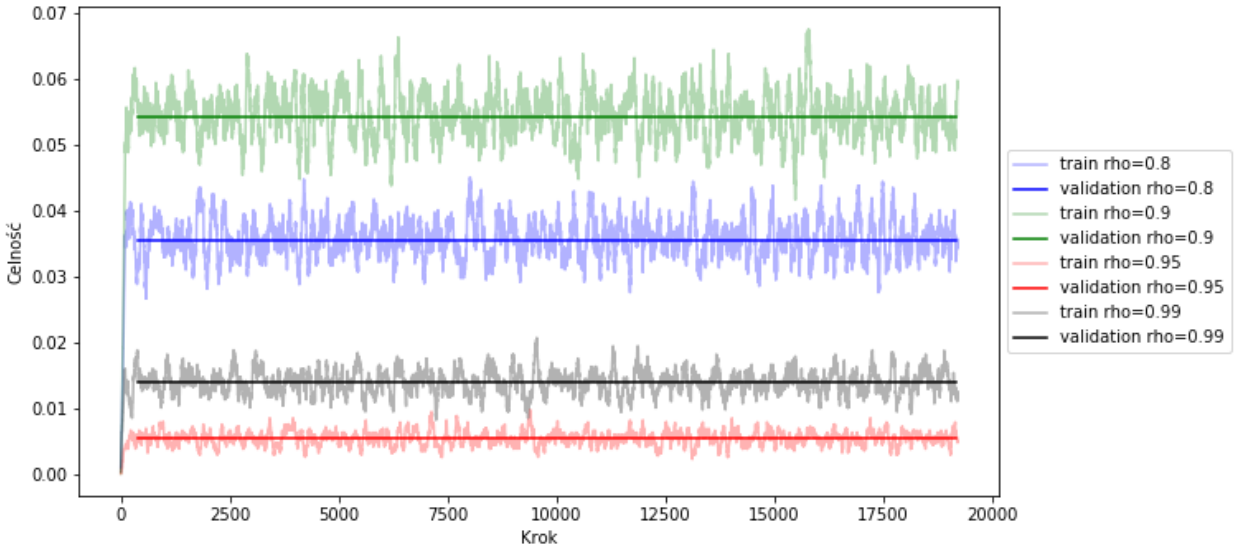
- lr = 0.001

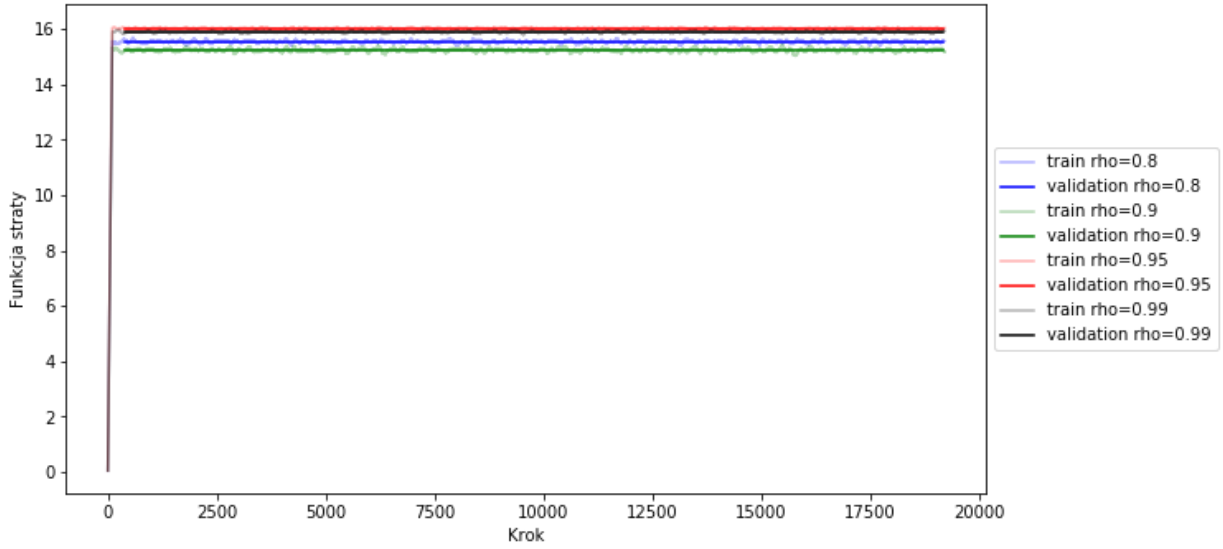

Celność
Krok
Funkcja straty
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

- lr = 0.01


Celność
Krok
Funkcja straty
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

- lr = 0.1


Celność
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99


Funkcja straty
Krok
train rho=0.8
validation rho=0.8
train rho=0.9
validation rho=0.9
train rho=0.95
validation rho=0.95
train rho=0.99
validation rho=0.99

3.1.1.3. Podsumowanie

Nazwa architektury	**learning rate**	**ρ**	**Najlepsza celność podczas walidacji**
VGG16	0,0001	0,99	42,1%
InceptionV3	0,001	0,99	35,5%
ResNet50	0,001	0,99	**43,5%**

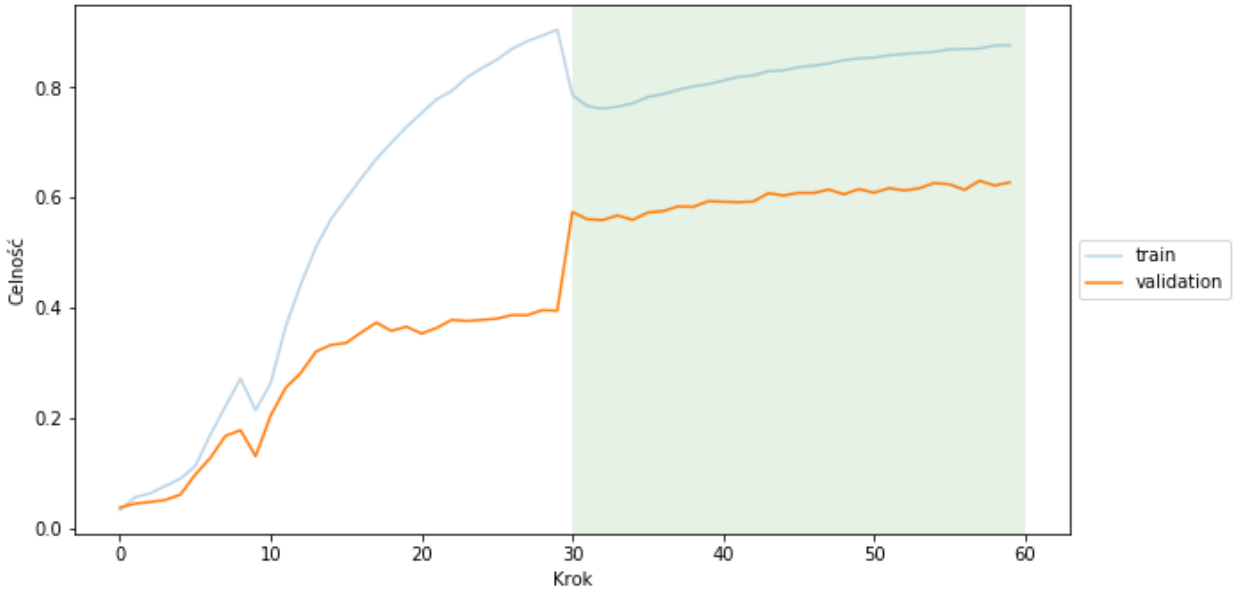
Dla wszystkich architektur bardziej skuteczna okazała się wyższa wartość parametru ρ niż zalecana, ponadto w przypadku VGG16 lepsza okazała się również mniejsza wartość learning rate. Modele które osiągnęły najwyższą celność zostały przekazane do kolejnej kroku eksperymentu - dostrajania ostatniego bloku konwolucyjnego.

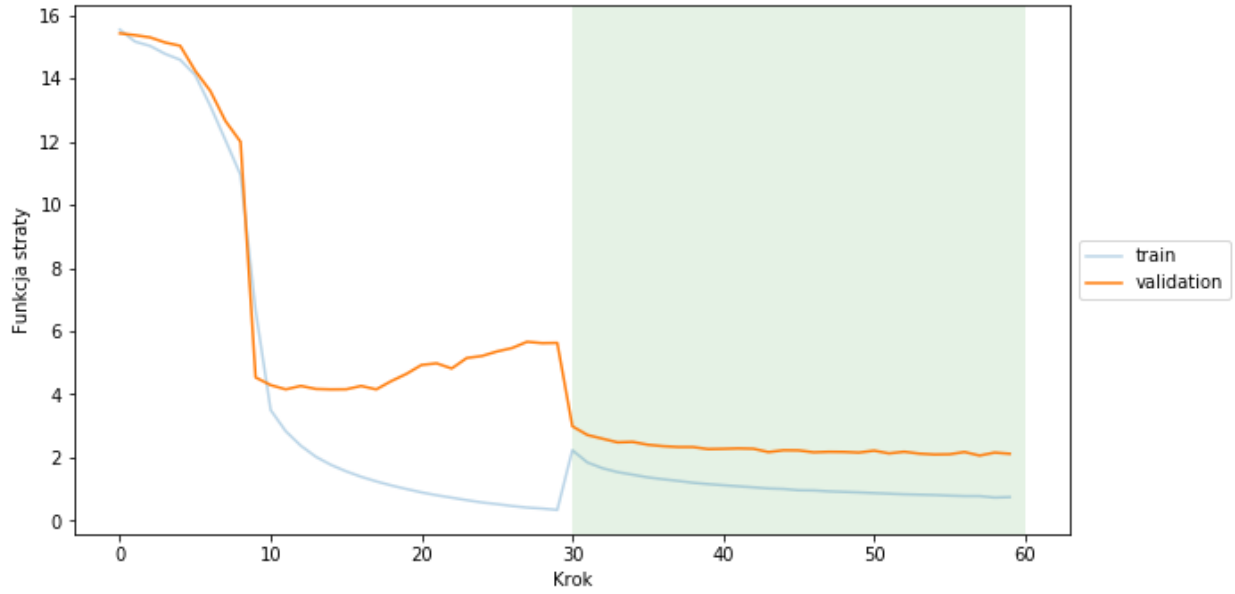
3.1.2. Dostrajanie (ang. fine-tuning)

Ze względu na występujący problem z implementacją warstwy BatchNormalization w bilbiotece Keras[9][10][11] badanie dostrajania musiało zostać zmodyfikowane. Poza scaleniem modeli wymagane było odmrożenie wag w warstwach BatchNormalization i ich douczenie. Wpłynęło to bardzo pozytywnie na jakość klasyfikacji dla InceptionV3 oraz ResNet50. Nie wpłynęło natomiast na VGG16 który nie posiada w swojej budowie wspomnianych warstw. Za duży skok celności w przypadku tej architektury odpowiedzialne jest połączenie wartsw klasyfikujących z warstwami konwolucyjnymi. Podczas samego procesu dostrajania VGG16 również osiągnęło największy zysk wśród testowanych architektur. Przyczyną może być to, że VGG16 w pierwszej fazie miało znacznie gorsze wyniki niż dwie pozostałe architektury, oraz to, że jeden blok konwolucyjny ma procentowo znacznie większy udział we wszystkich wagach sieci niż w Inception oraz ResNet.

Faza dostrajania modelu została wyróżniona na poniższych wykresach za pomocą seledynowego tła.

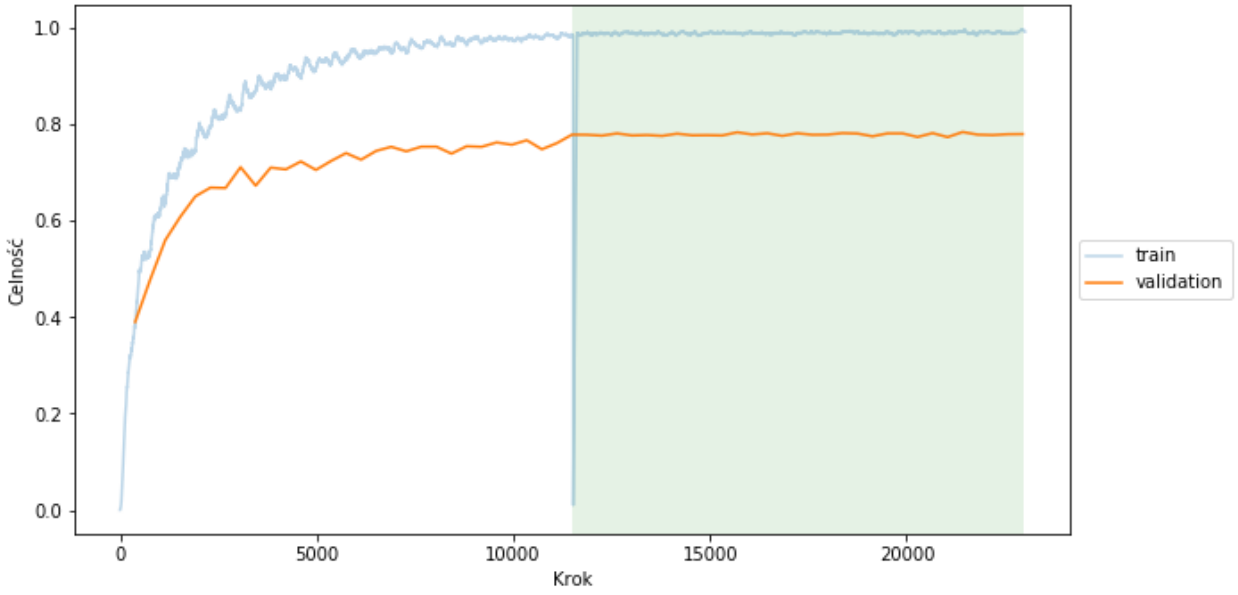
3.1.2.1. VGG16

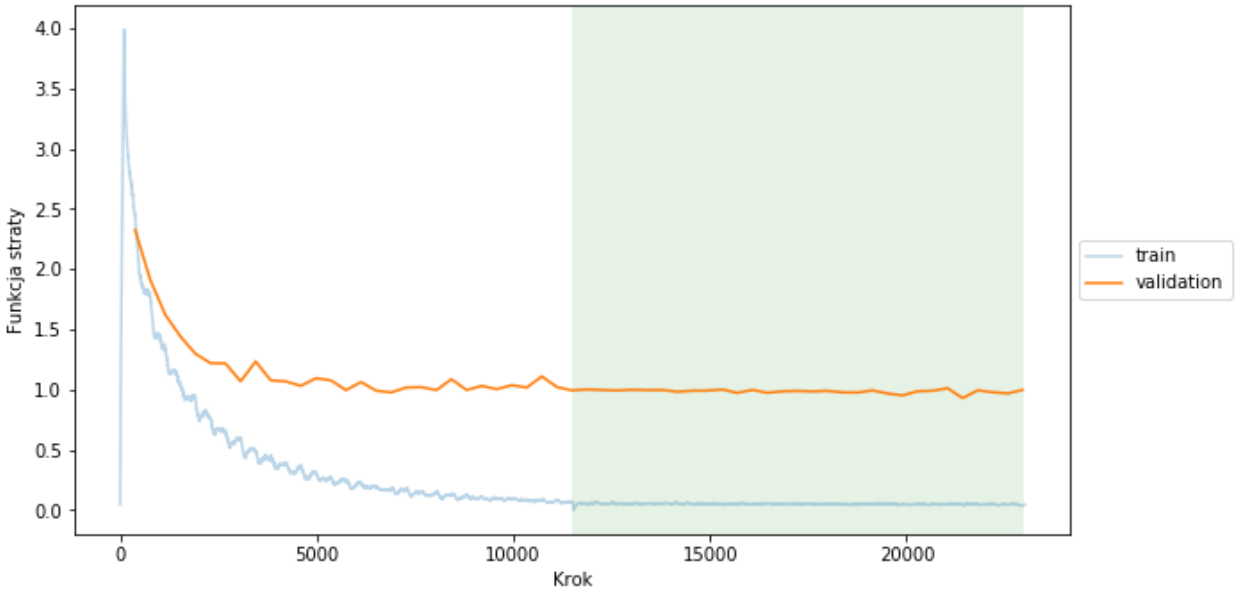




Najlepszy wynik na zbiorze walidacyjnym: 63,1%

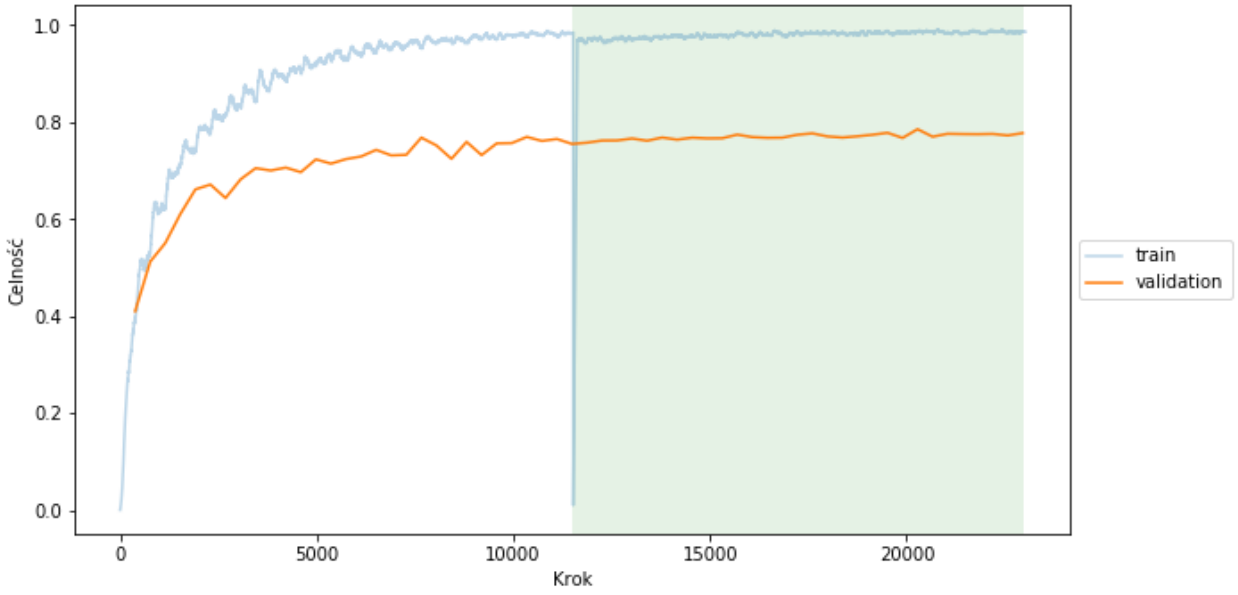
3.1.2.2. InceptionV3

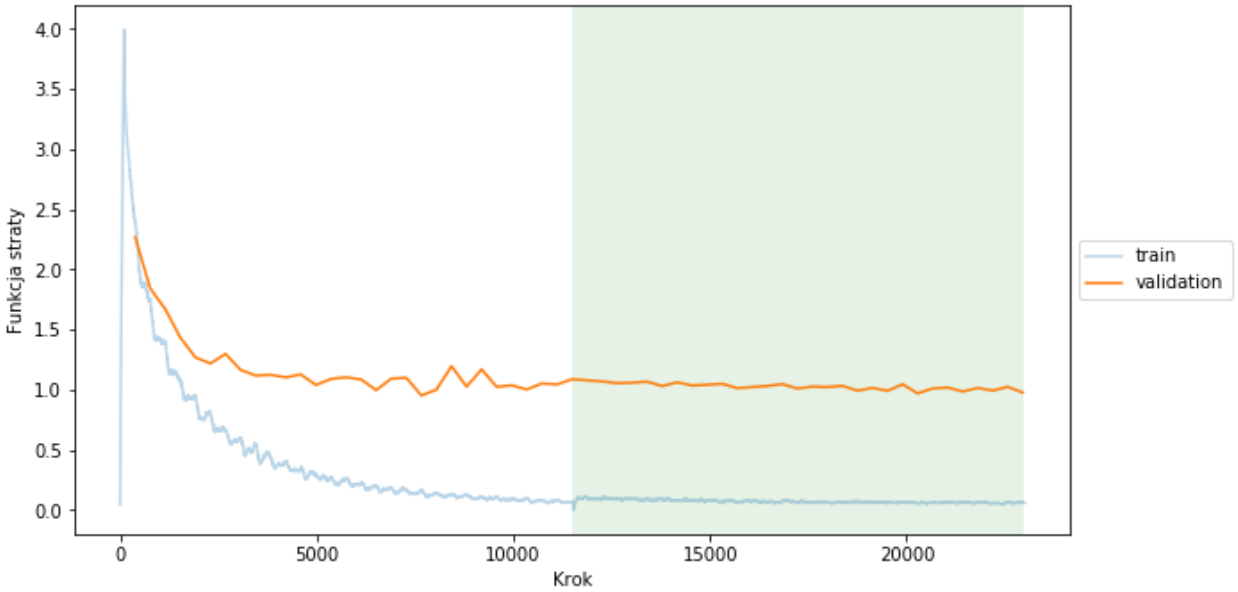




Najlepszy wynik na zbiorze walidacyjnym: 78,4%

3.1.2.3. ResNet50

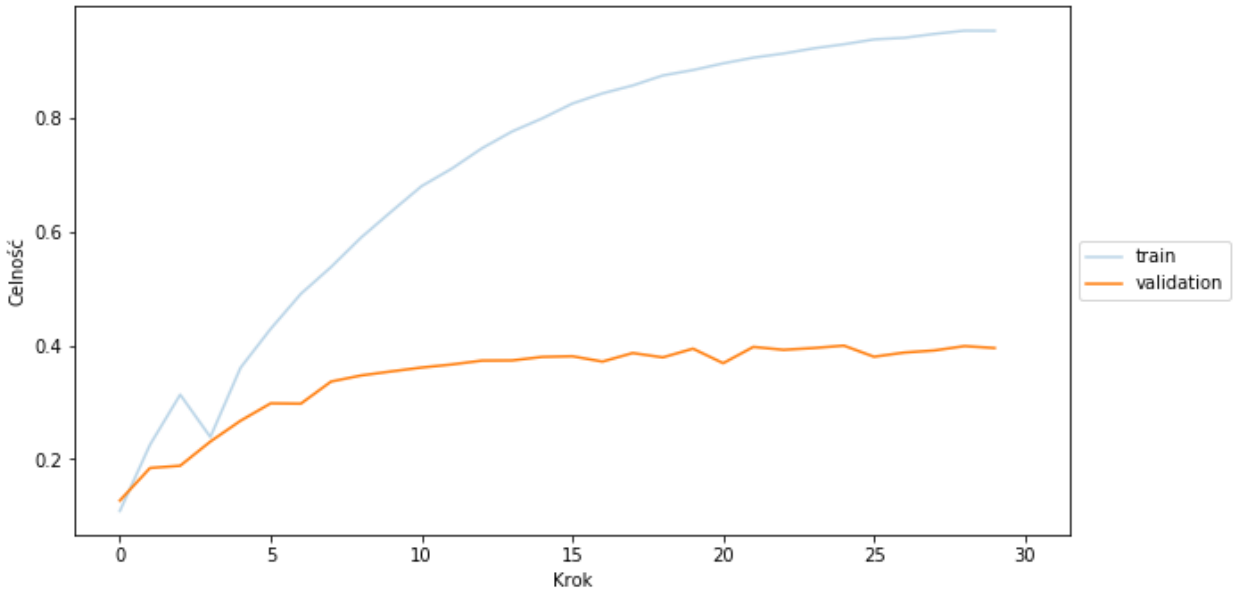


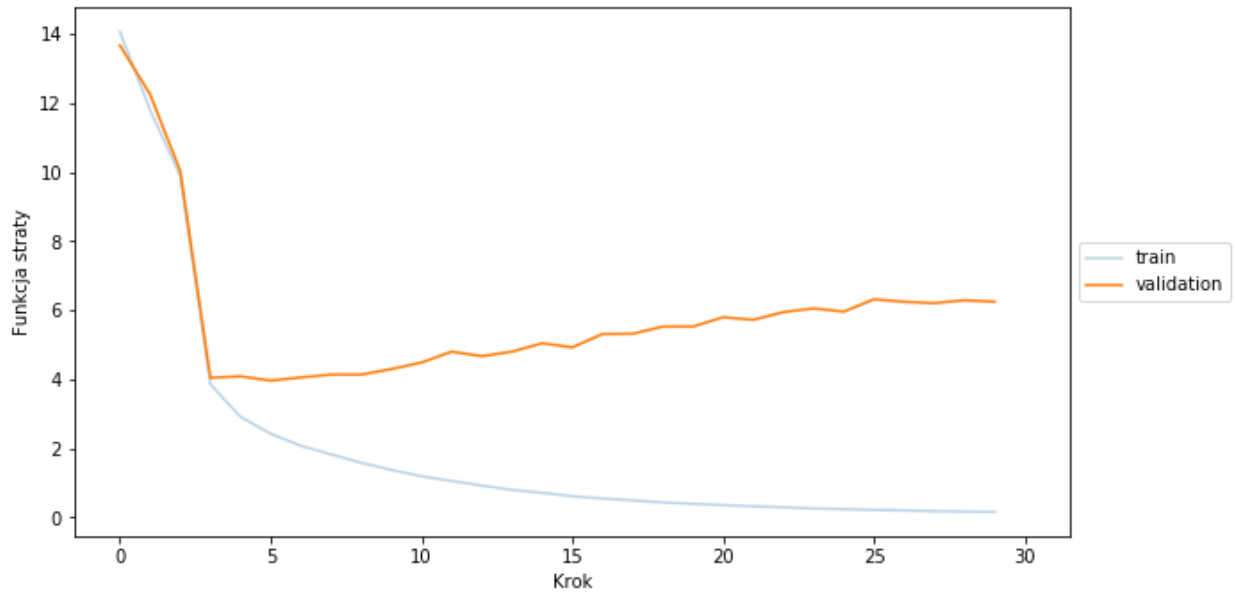


Najlepszy wynik na zbiorze walidacyjnym: 79,8%

3.1.3. Doszkalanie do rozszerzonego zadania

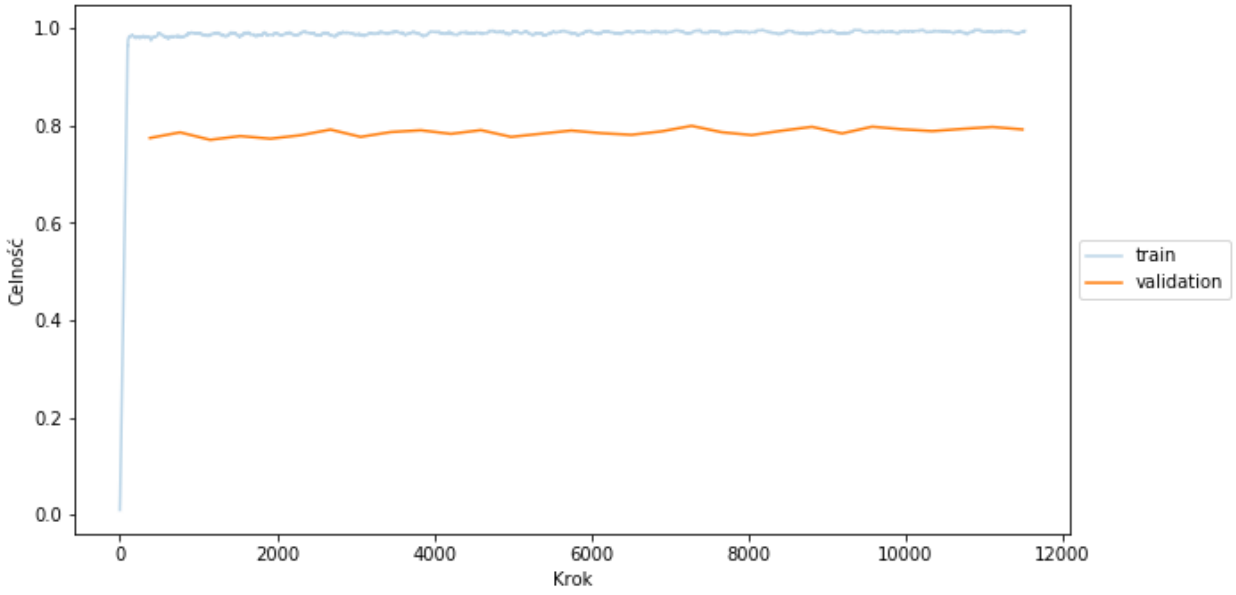
3.1.3.1 VGG16

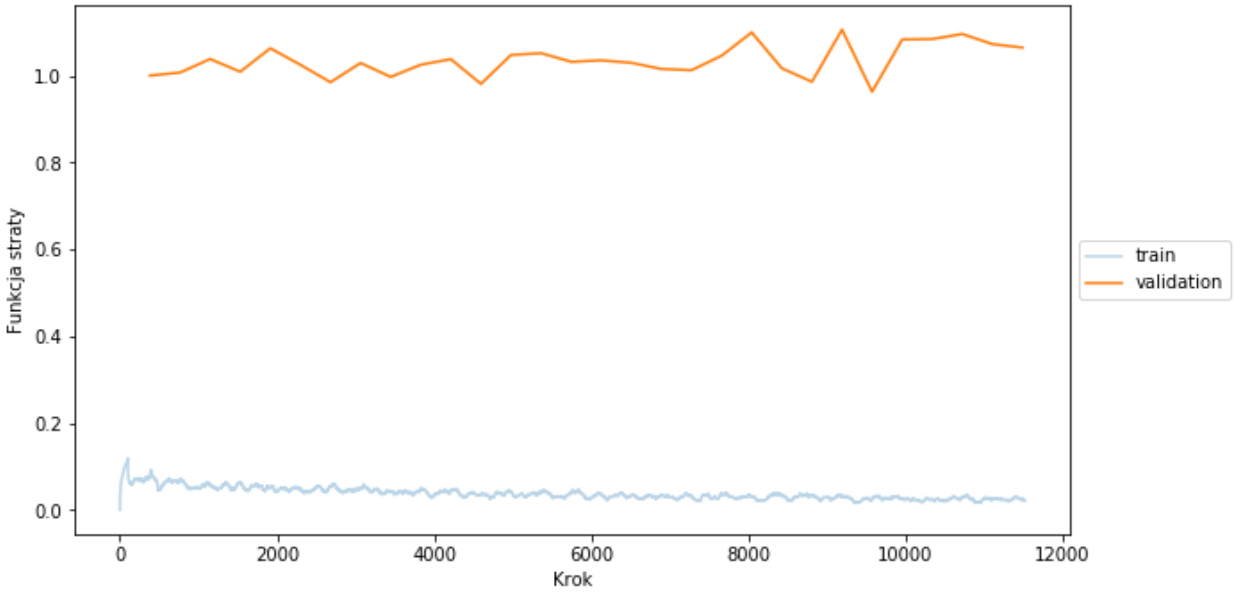




Najlepszy wynik na zbiorze walidacyjnym: 40%

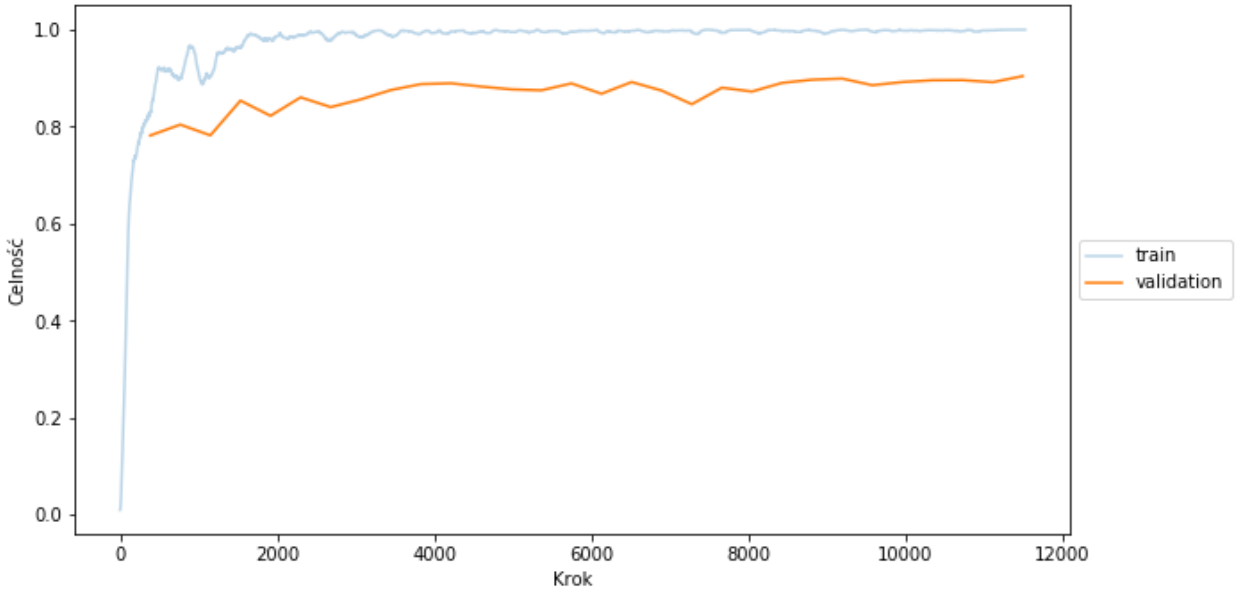
3.1.3.2 InceptionV3

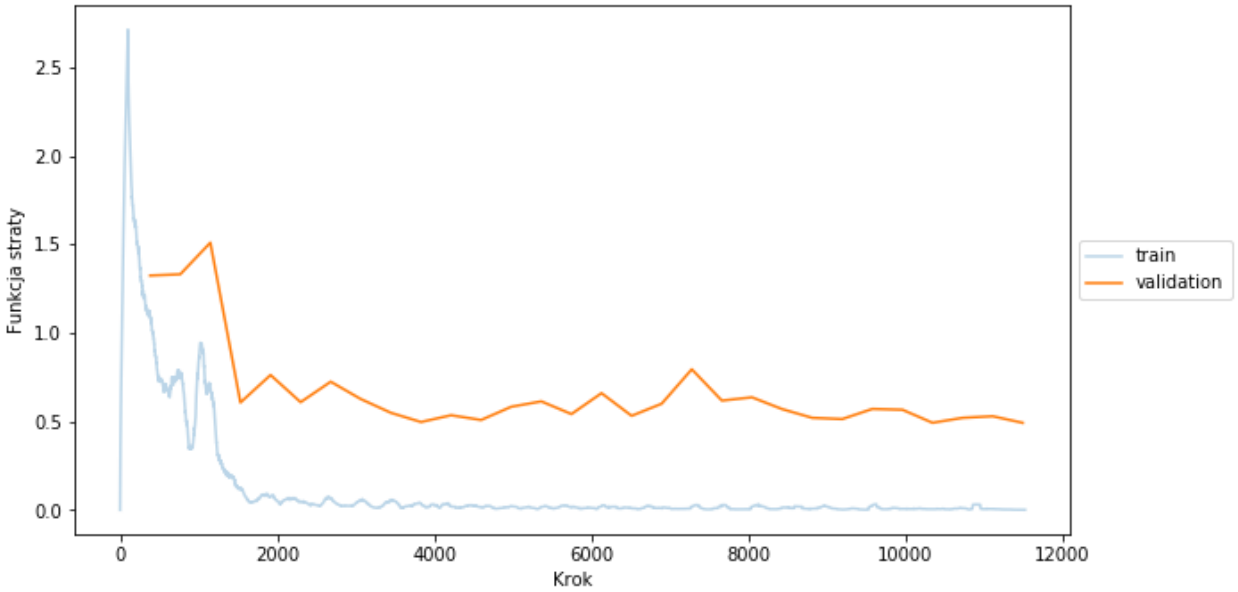




Najlepszy wynik na zbiorze walidacyjnym: 79,9%

3.1.3.3 ResNet50





Najlepszy wynik na zbiorze walidacyjnym: 90,4%

3.1.4. Wydajność

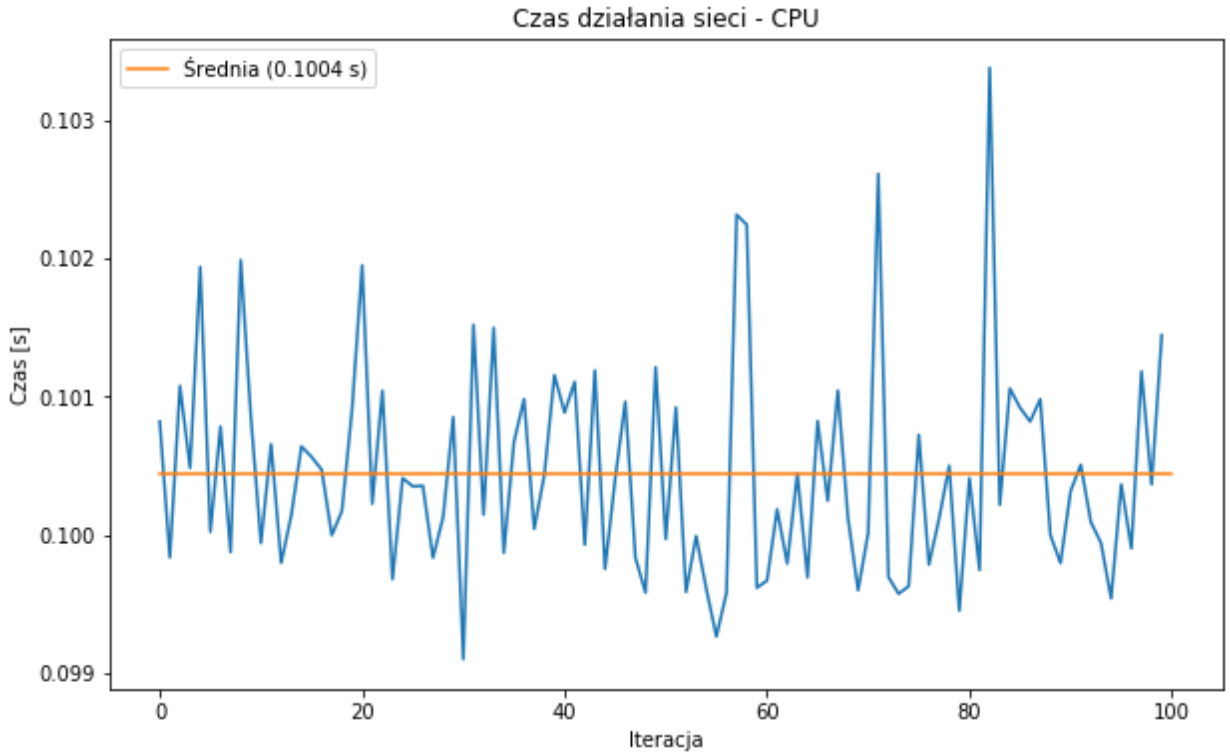
W tym badaniu zmierzono czas działania oraz zapotrzebowanie na pamięć na trzech różnych platformach:

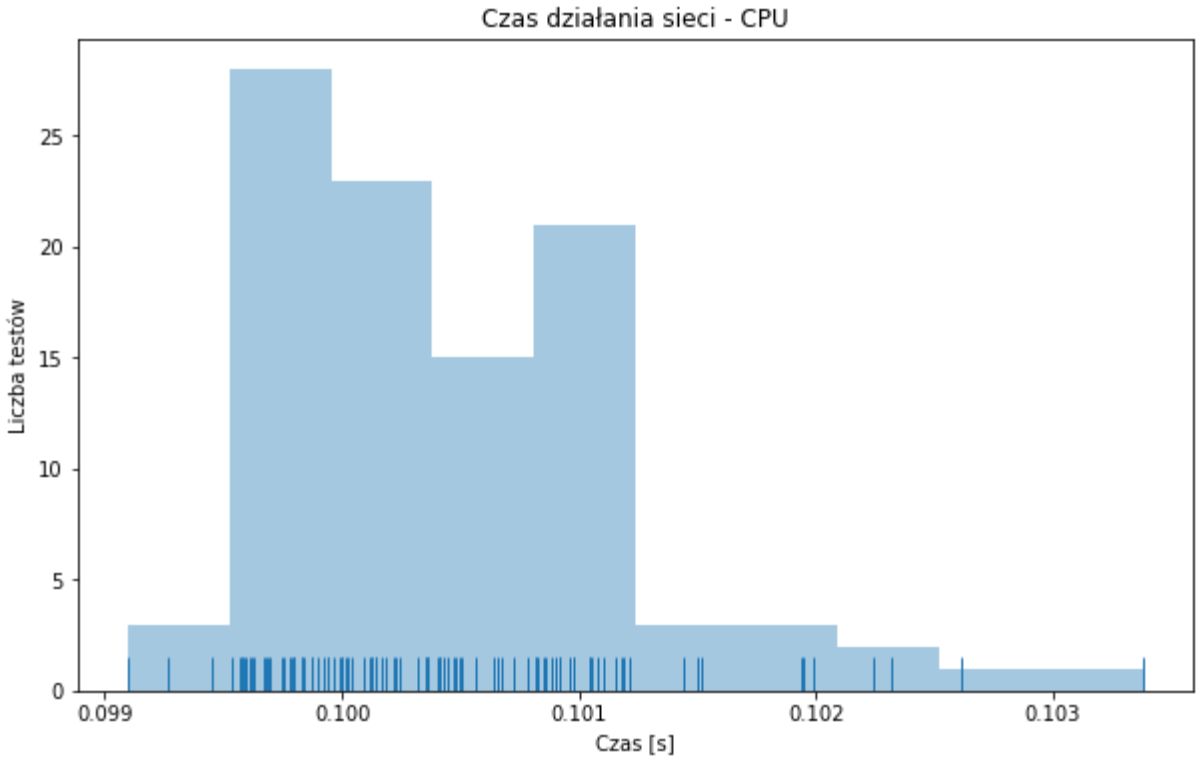
1. CPU - Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz; 4x8GB 2133MHz RAM, Ubuntu 16.04 LTS
2. GPU - Nvidia GeForce GTX 1080 Ti 11GB, Ubuntu 16.04 LTS
3. Urządzenie mobilne - Nexus 9, Android 7.1.1, NVIDIA Tegra K1 2300MHz, 2GB RAM

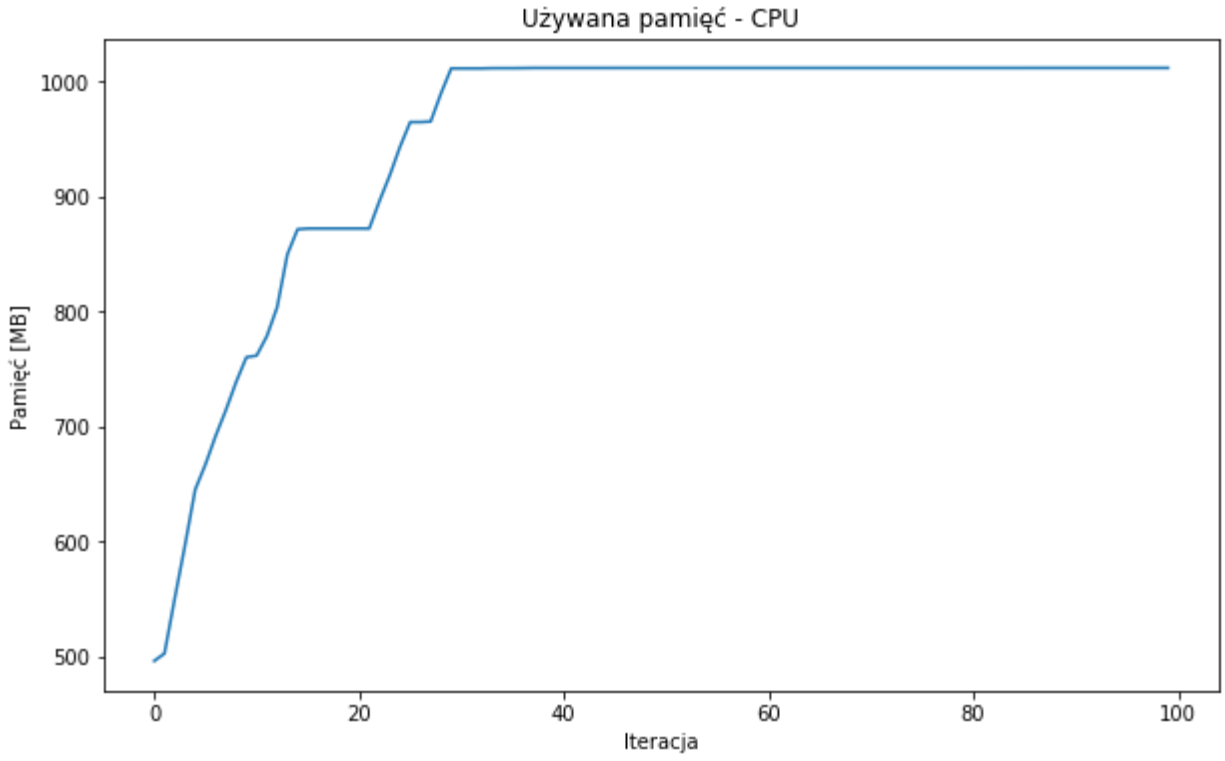
3.1.4.1. CPU

Podczas badania wydajności sieci na CPU mierzono czas predykcji dla 100 przykładowych obrazów w rozmiarach odpowiednich dla danej architektury. Do uzyskania informacji o zużytej pamięci operacyjnej wykorzystano moduł resource należący do biblioteki standardowej języka Python.

3.1.4.1.1. VGG16

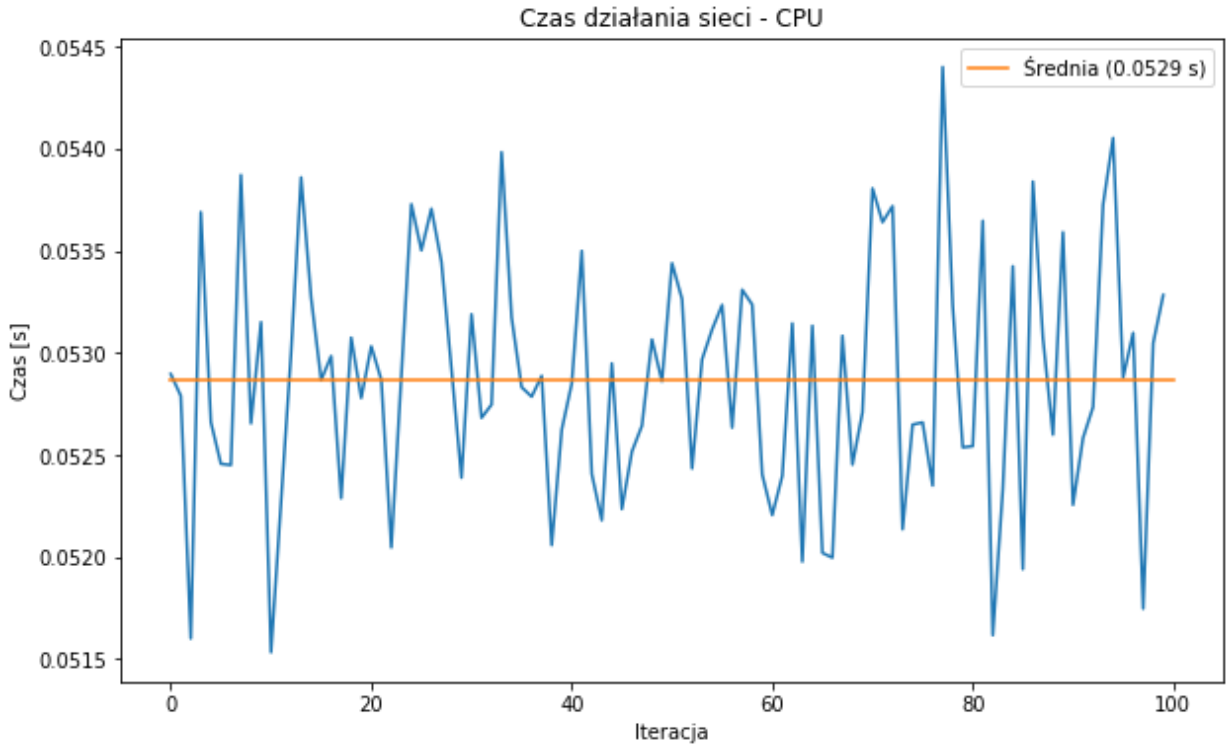


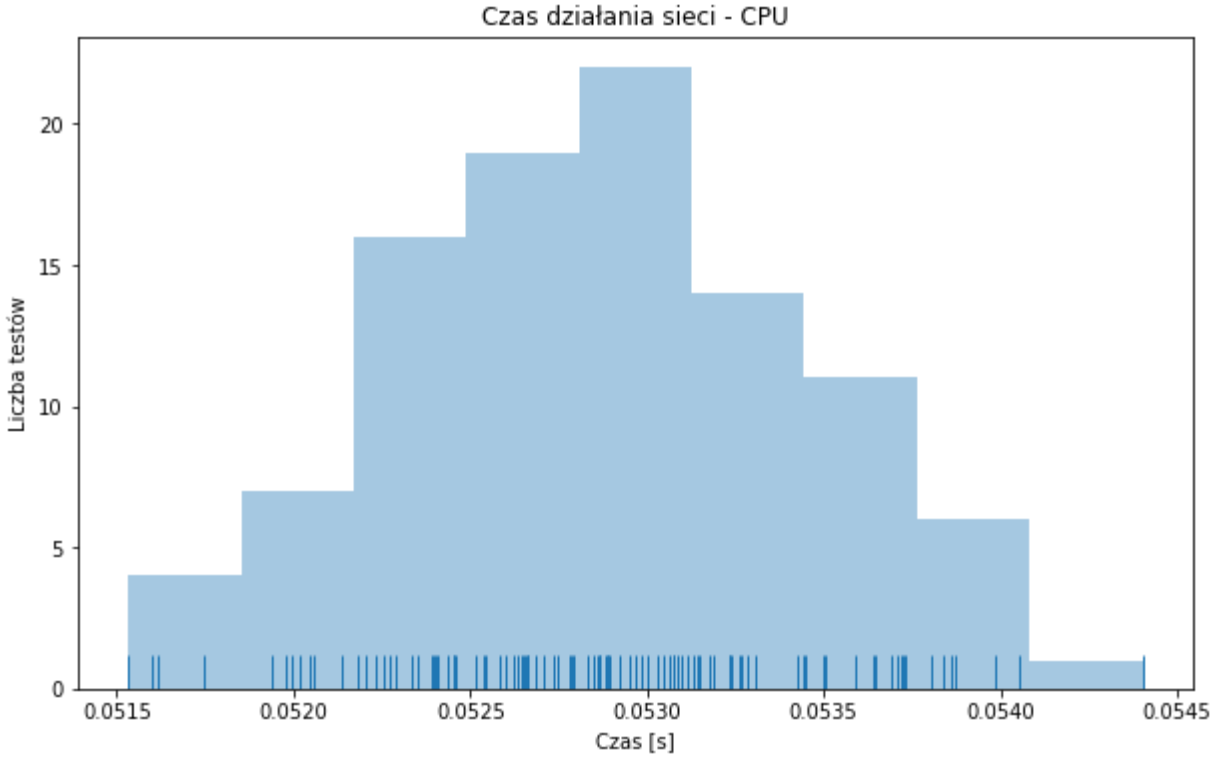


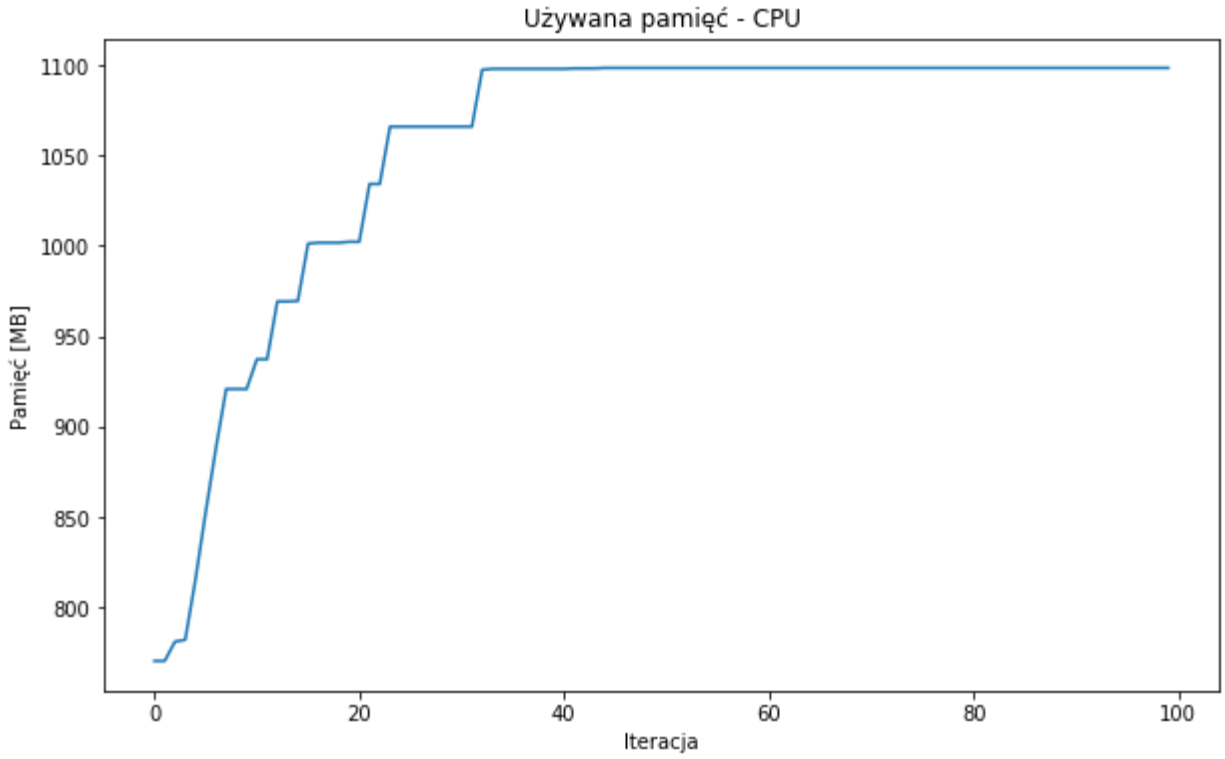


Maksymalne zużycie pamięci: 1011,57MB

3.1.4.1.2 InceptionV3

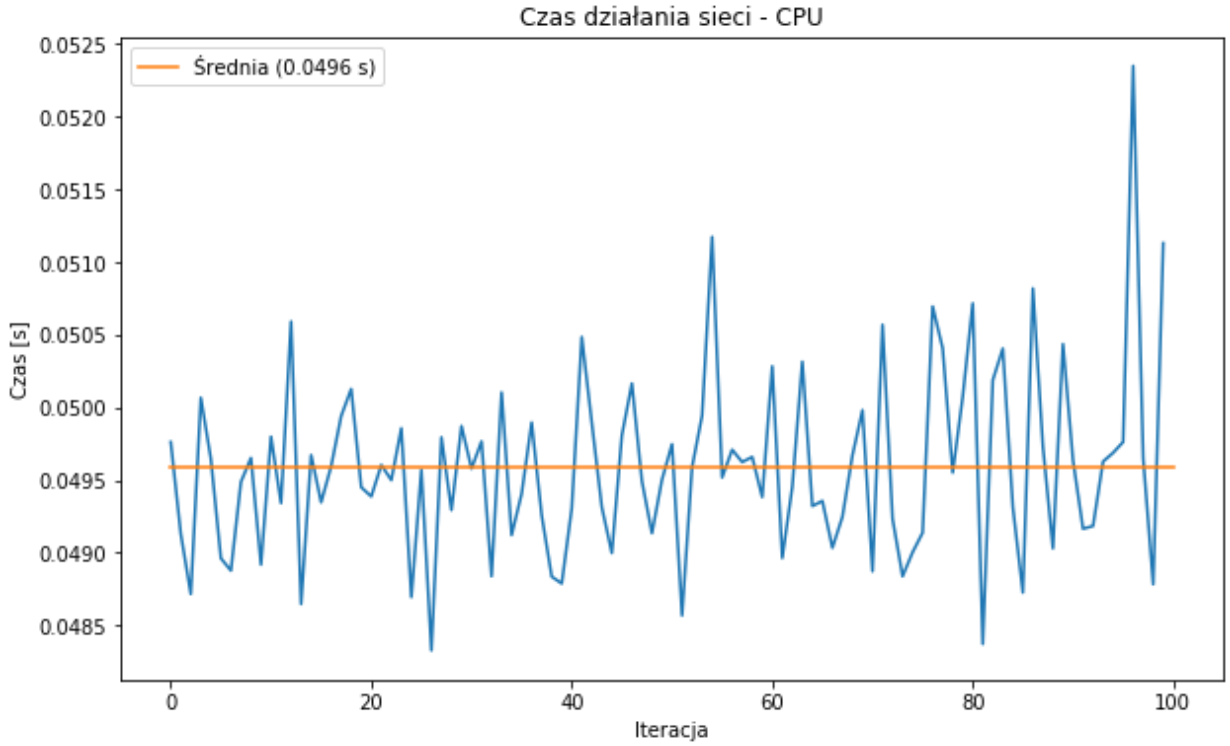


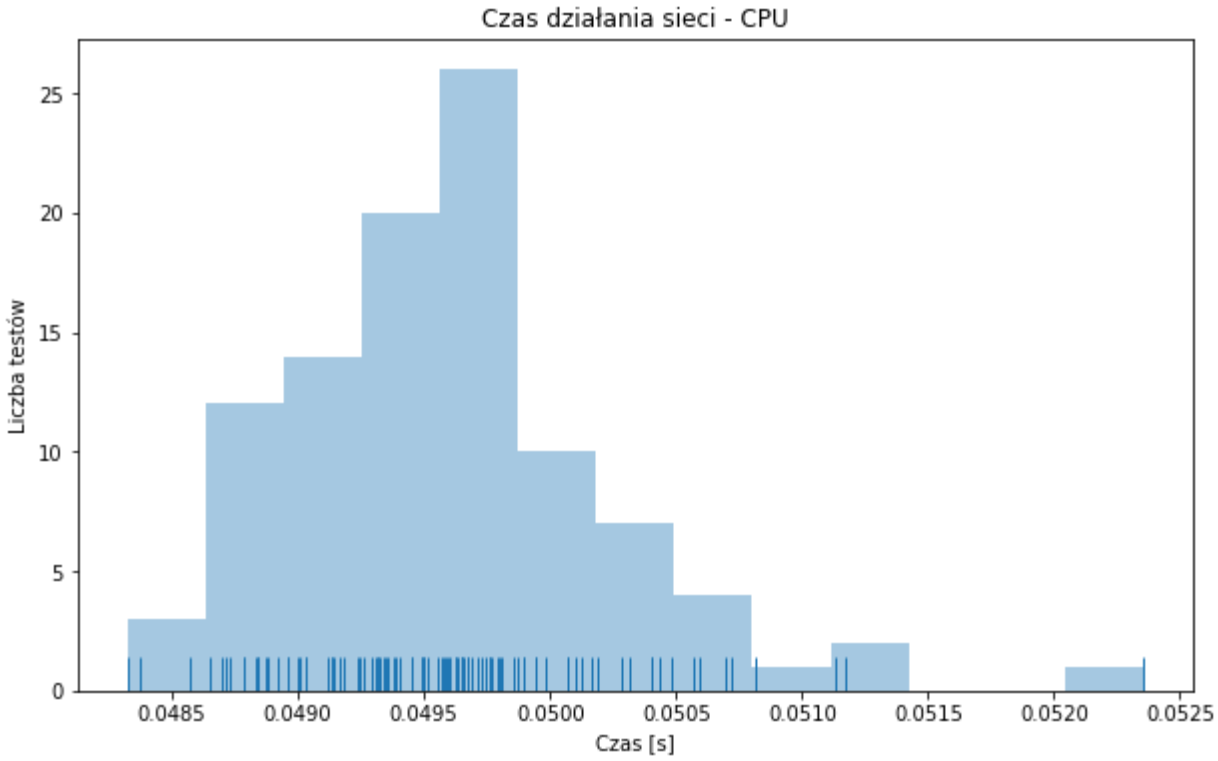


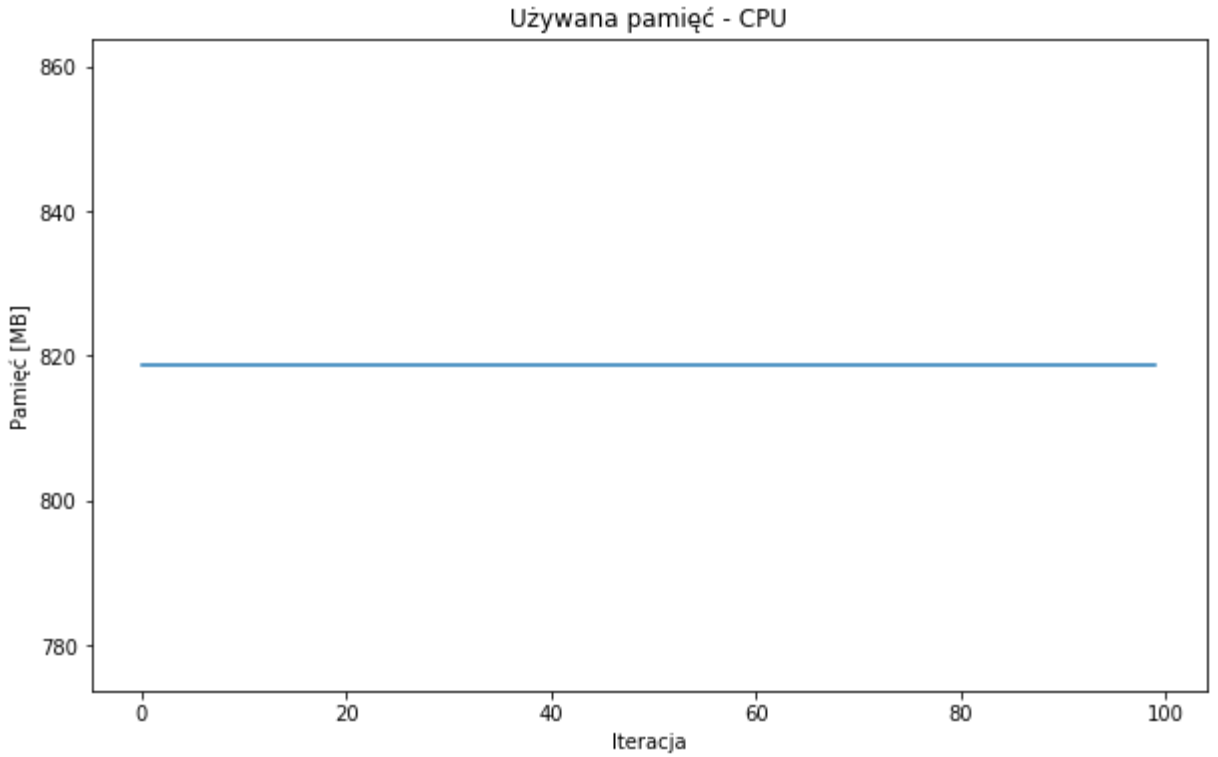


Maksymalne zużycie pamięci: 1103,47MB

3.1.4.1.3. ResNet50







Maksymalne zużycie pamięci: 818,83MB

3.1.4.1.4. Podsumowanie

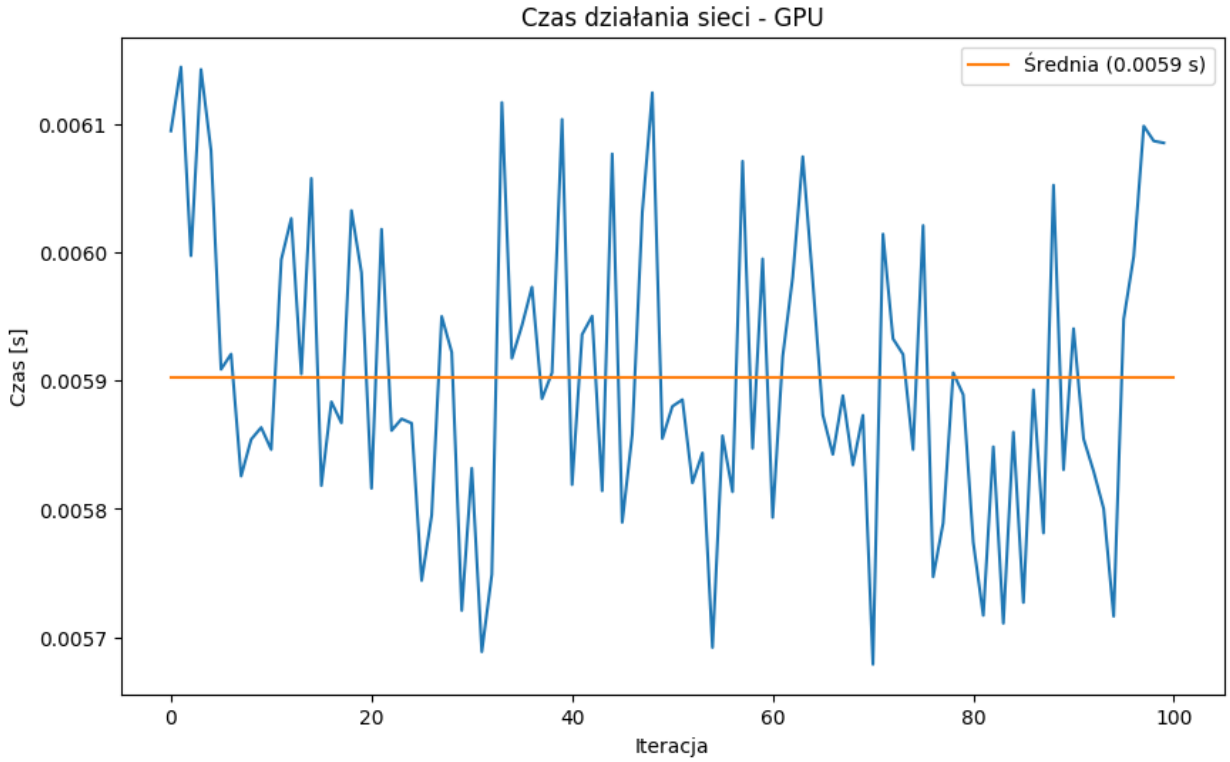
Nazwa architektury	**Średni czas działania sieci [s]**	**Maksymalne zużycie pamięci [MB]**
VGG16	0,1004	1011,57
InceptionV3	0,0529	1103,47
ResNet50	**0,0496**	**818,83**

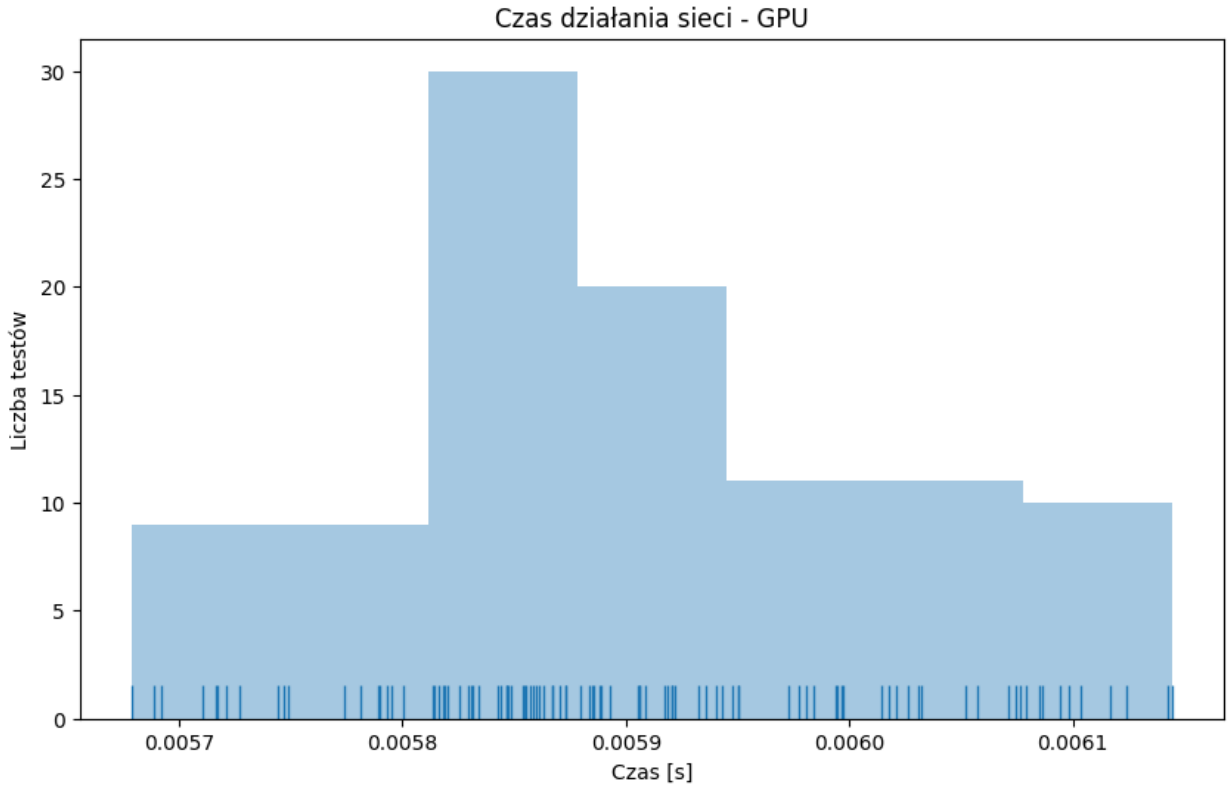
Podczas działania na CPU najwydajnieszym okazał się model ResNet50, będąc nieznacznie szybszym od InceptionV3, ale zajmując przy tym 25% mniej pamięci operacyjnej. Z perspektywy wymagań projektu najgorzej wypadła architektura VGG16, która do klasyfikacji potrzebowała dwukrotnie więcej czasu niż pozostałe.

3.1.4.2. Wydajność z użyciem procesora graficznego

W przypadku badania wydajności na procesorze graficznym czas działania sieci był badany podobnie jak w przypadku CPU, jednak by otrzymać wartość wykorzystanej pamięci zostały wprowadzone pewne modyfikacje. Domyślnie framework TensorFlow który w tym przypadku działa jako backend biblioteki Keras, mapuje praktycznie całą dostępną pamięć procesora graficznego. Dzieje się tak ponieważ umożliwia to bardziej wydajną pracę redukując fragmentację pamięci[6]. Aby to zmienić należy zmienić odpowiednia flagę (`tf.ConfigProto.gpu_options.allow_growth`) w ustawieniach sesji TensorFlow. Po tej modyfikacji odczytano z narzędzia nvidia-smi maksymalną wartość użytej pamięci przez proces.

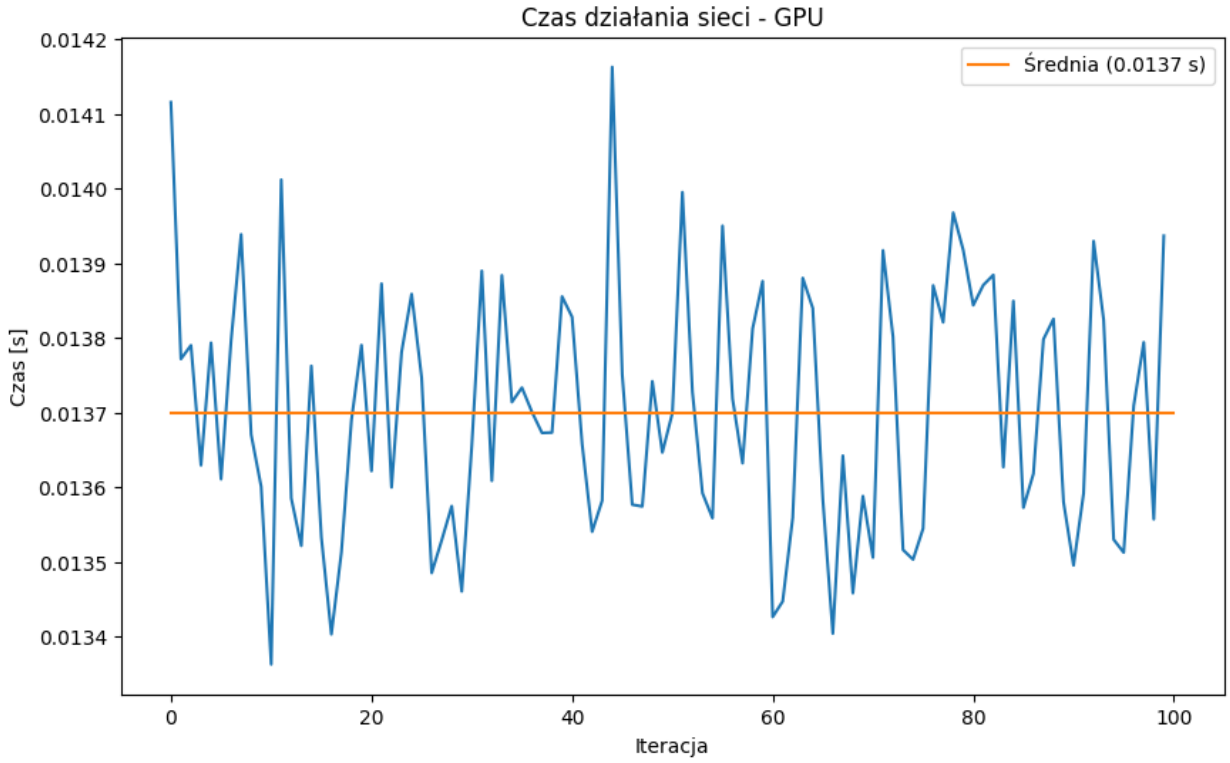
3.1.4.2.1. VGG16

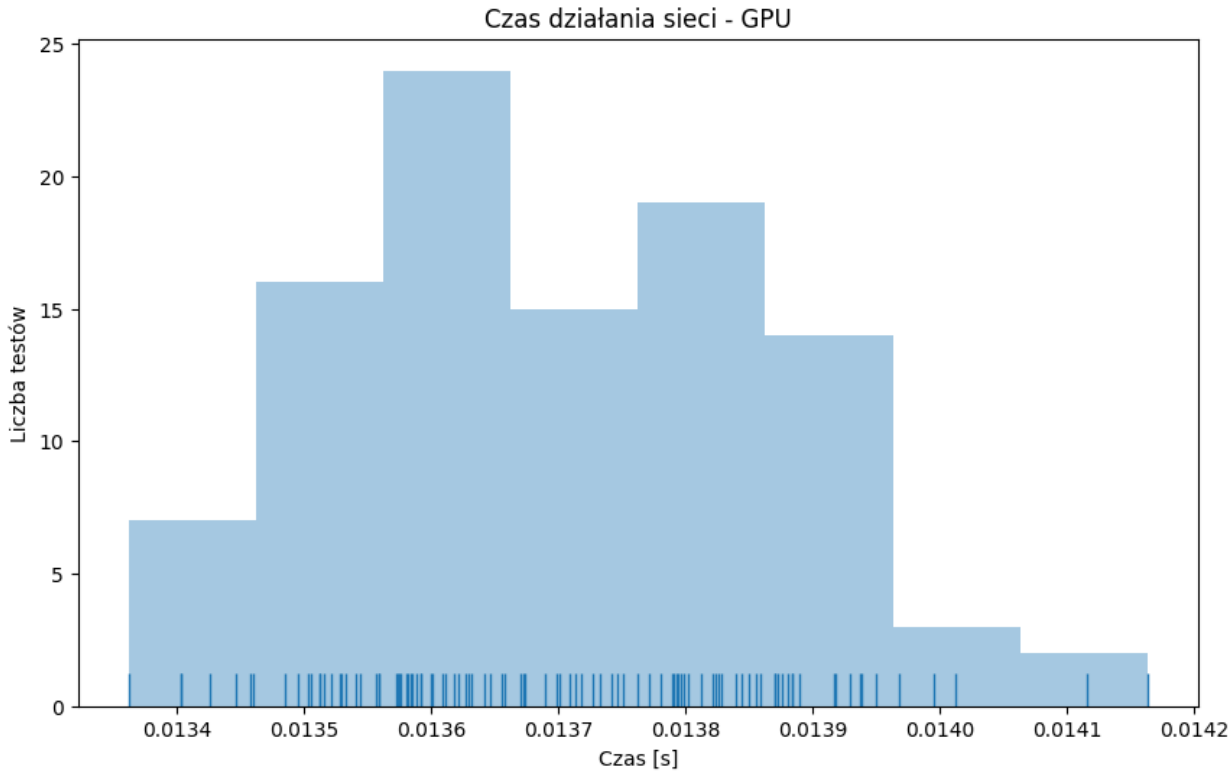

Czas działania sieci - GPU
Średnia (0.0059 s)
0.0061
0.0060
0.0059
0.0058
0.0057
Czas [s]
0
20
40
60
80
100
Iteracja


Czas działania sieci - GPU
30
25
20
15
10
5
0
Liczba testów
0.0057
0.0058
0.0059
0.0060
0.0061
Czas [s]

Maksymalne zużycie pamięci VRAM: 5235MB

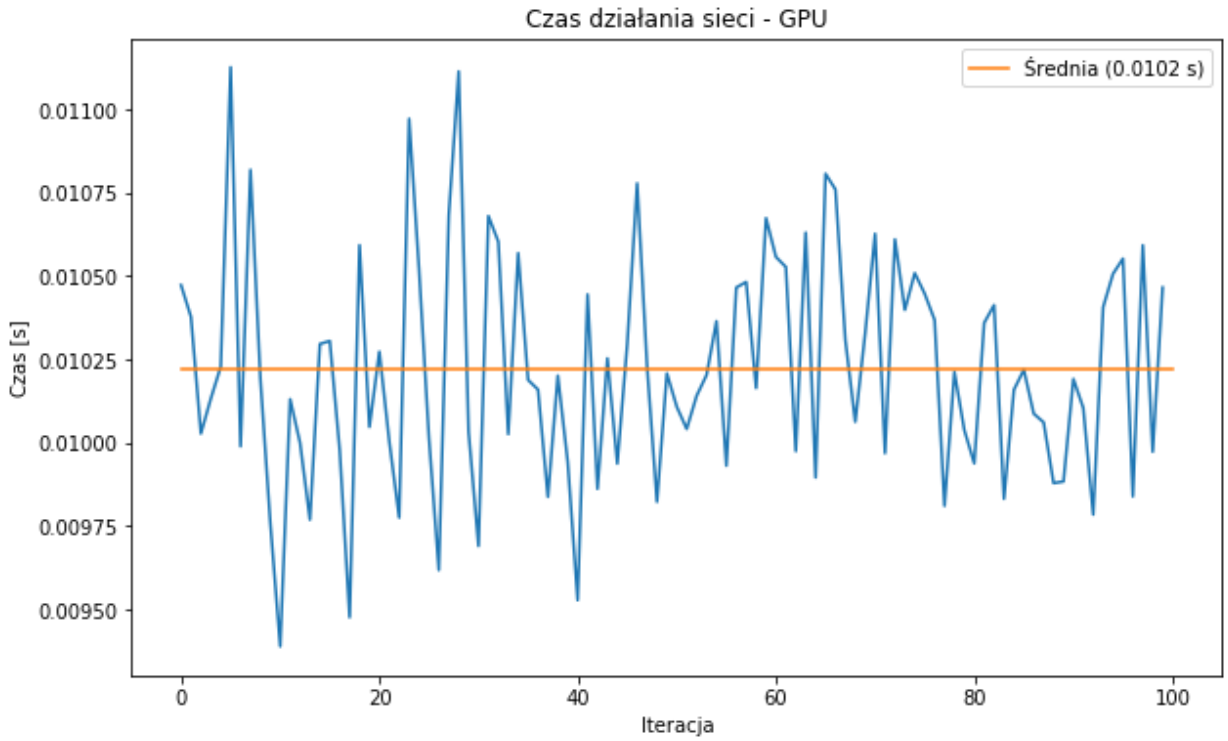
3.1.4.2.2. InceptionV3

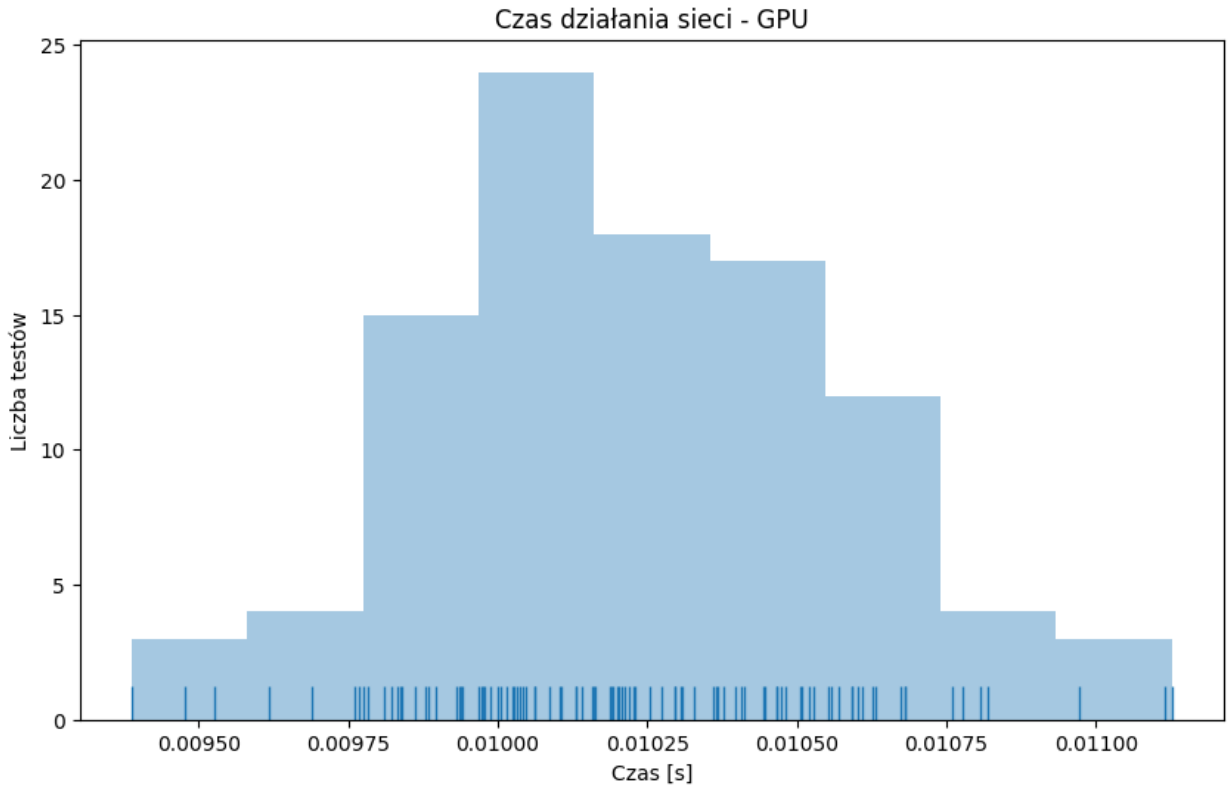




Maksymalne zużycie pamięci: 3571MB

3.1.4.2.3. ResNet50





Maksymalne zużycie pamięci VRAM: 5875MB

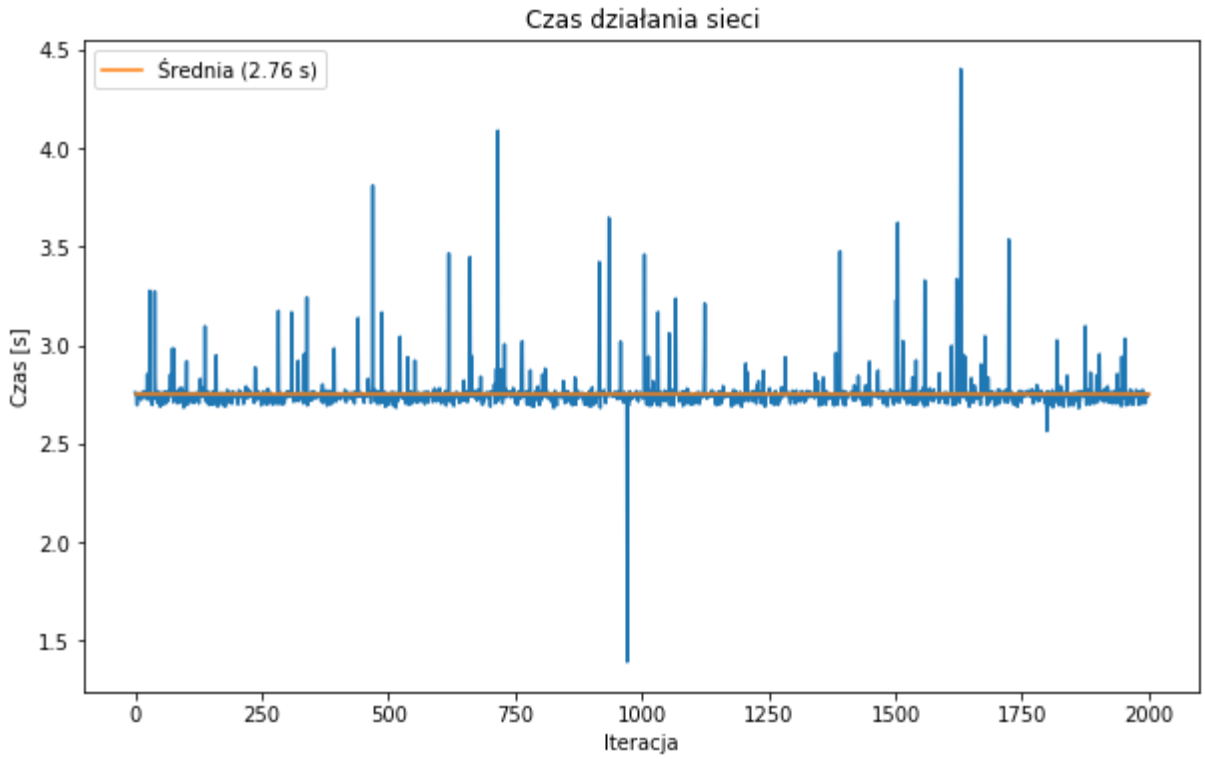
3.1.4.2.4 Podsumowanie

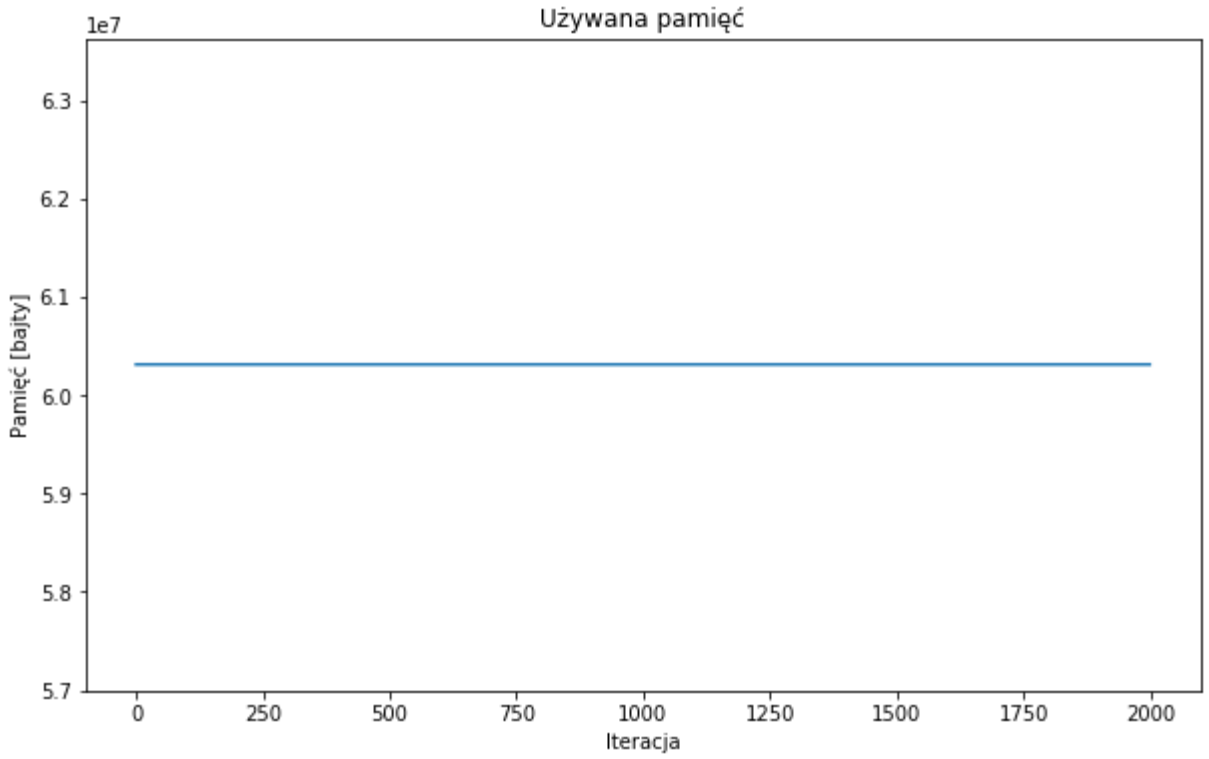
Nazwa architektury	**Średni czas działania sieci [s]**	**Maksymalne zużycie pamięci VRAM [MB]**
VGG16	**0,0059**	5235
InceptionV3	0,0137	**3571**
ResNet50	0,0102	5875

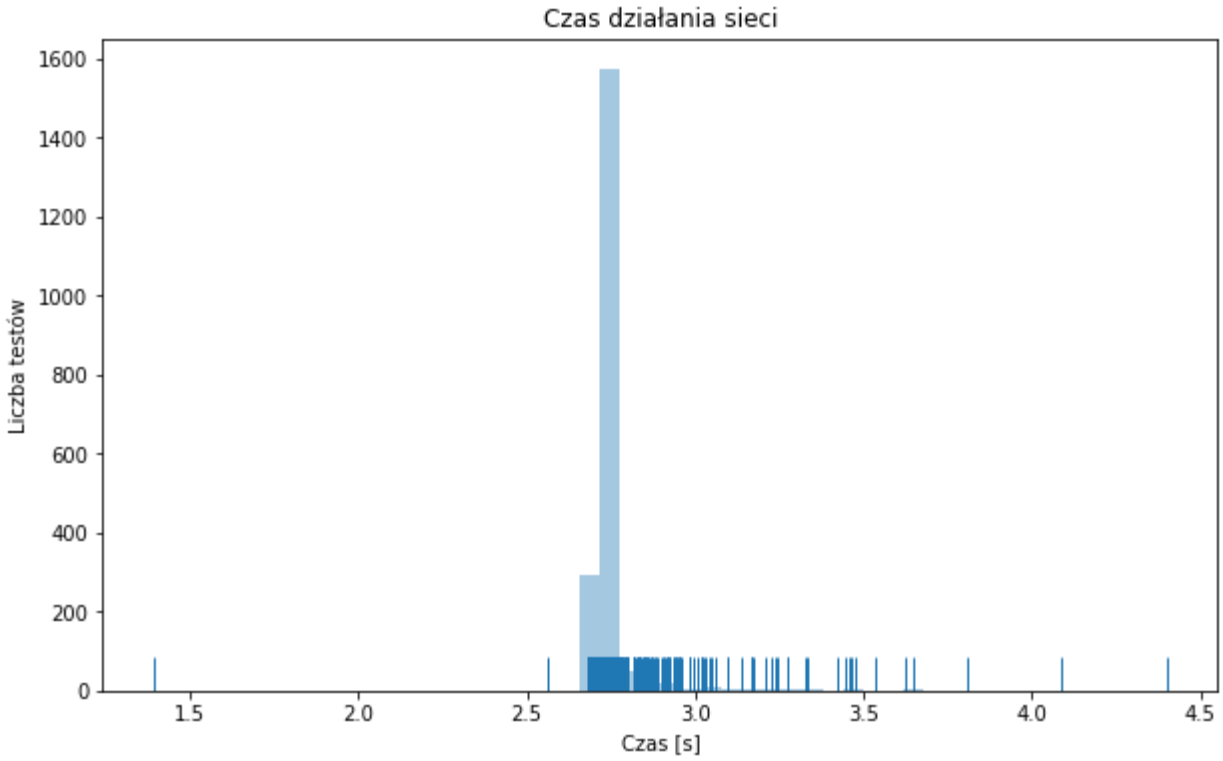
W przypadku klasyfikacji z użyciem GPU VGG16 okazał się być znacznie szybszy nić ResNet, który osiągnął najlepsze wyniki na CPU. Najbardziej oszczędną architekturą pod względem pamięci jest InceptionV3.

3.1.4.3. Wydajność na urządzeniu mobilnym

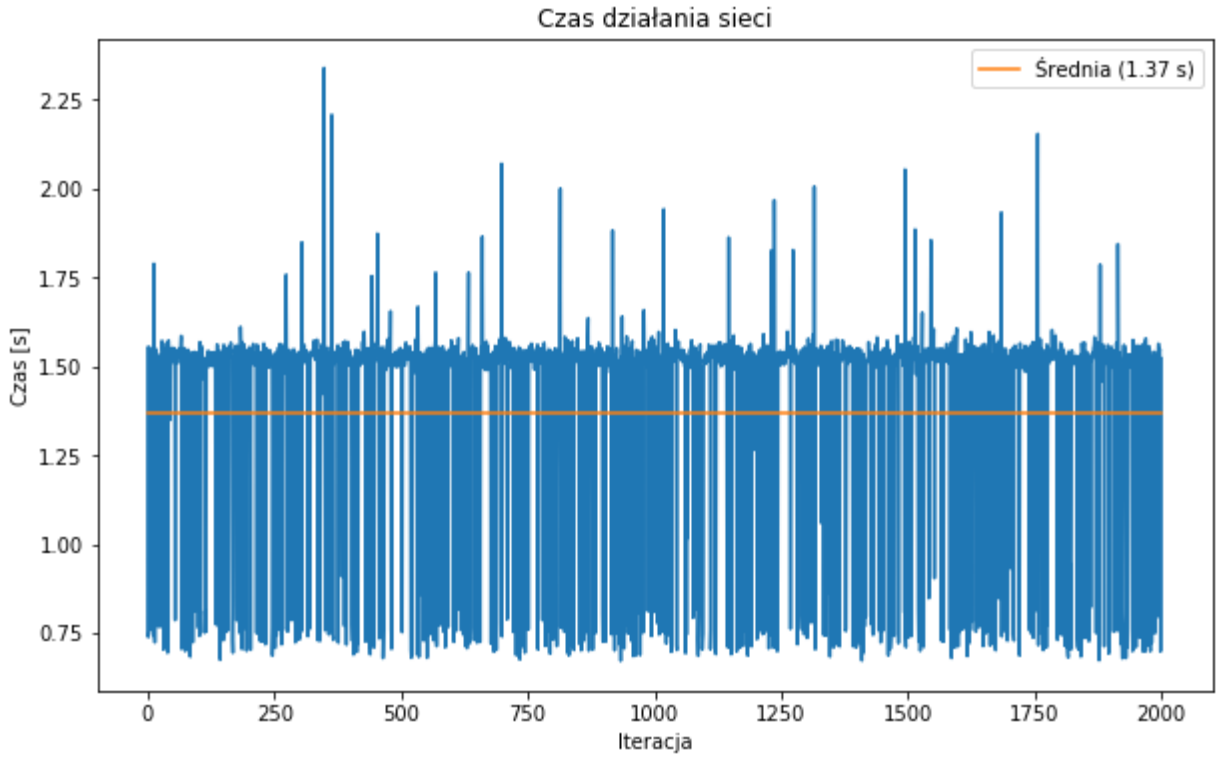
3.1.4.3.1. VGG16

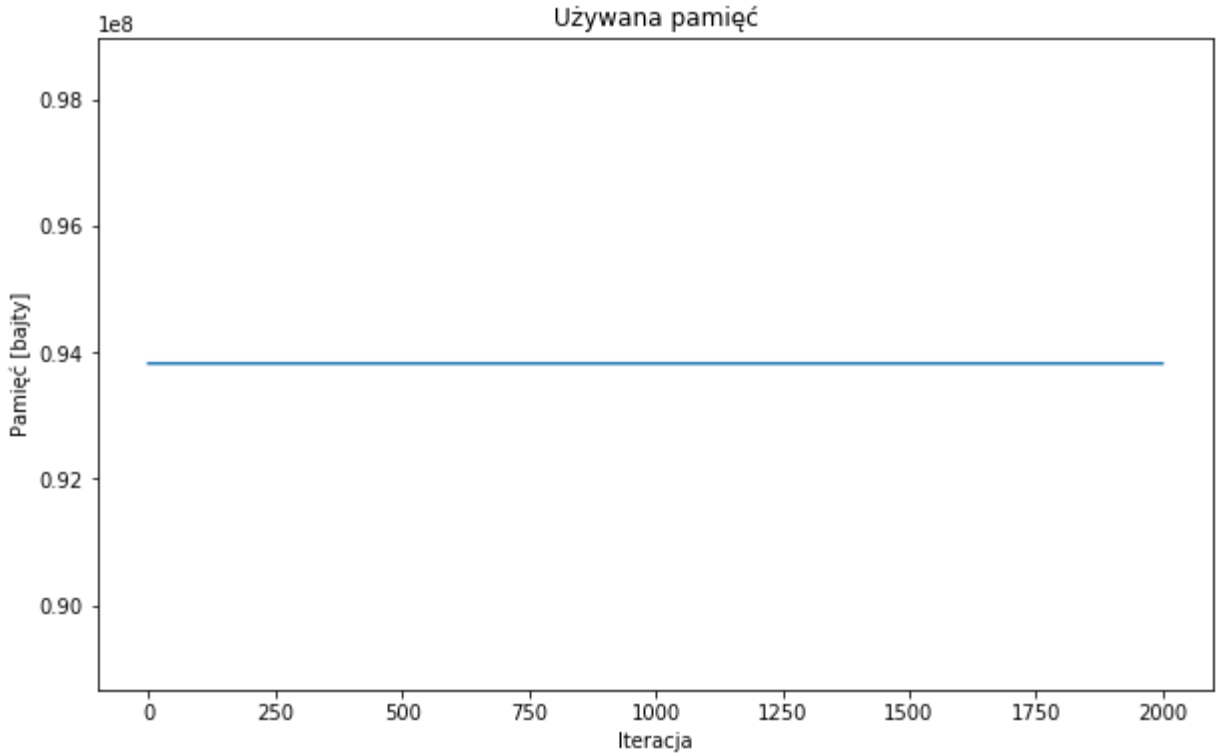


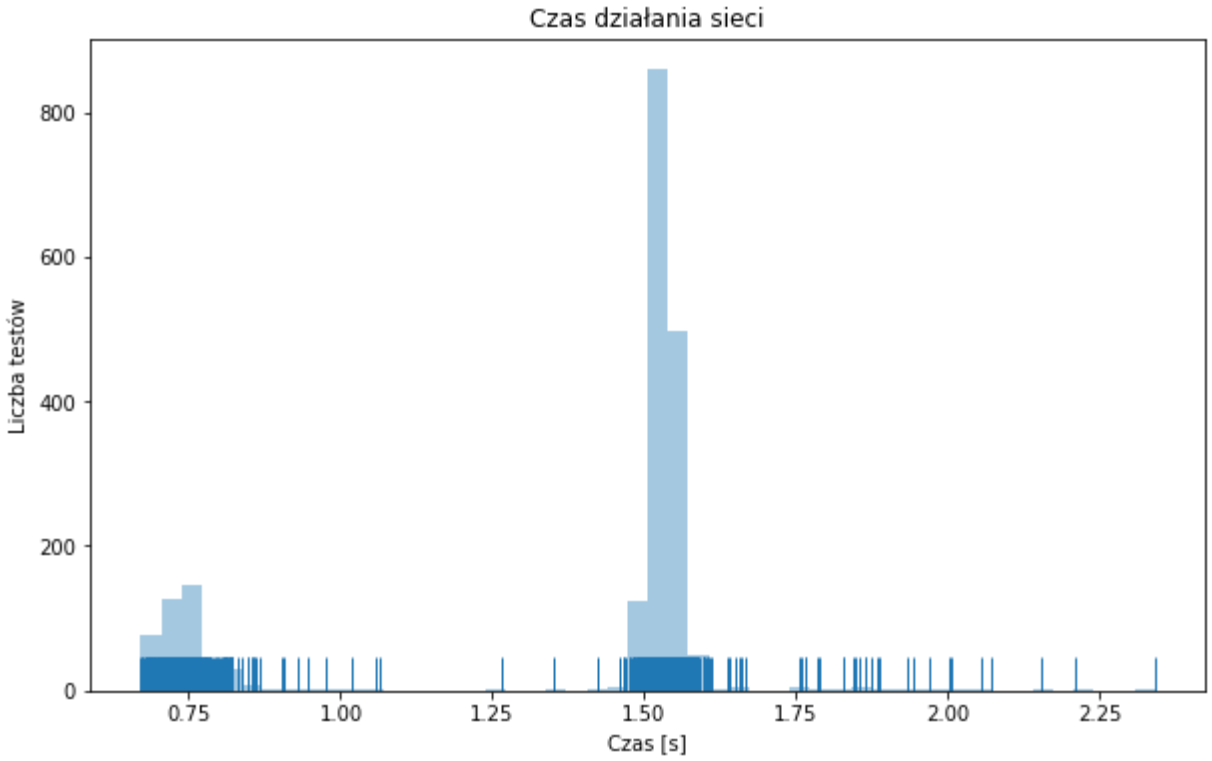

Używana pamięć
1e7
6.3
6.2
6.1
6.0
5.9
5.8
5.7
Pamięć [bajty]
0
250
500
750
1000
1250
1500
1750
2000
Iteracja


Czas działania sieci
1600
1400
1200
1000
800
600
400
200
0
Liczba testów
1.5
2.0
2.5
3.0
3.5
4.0
4.5
Czas [s]

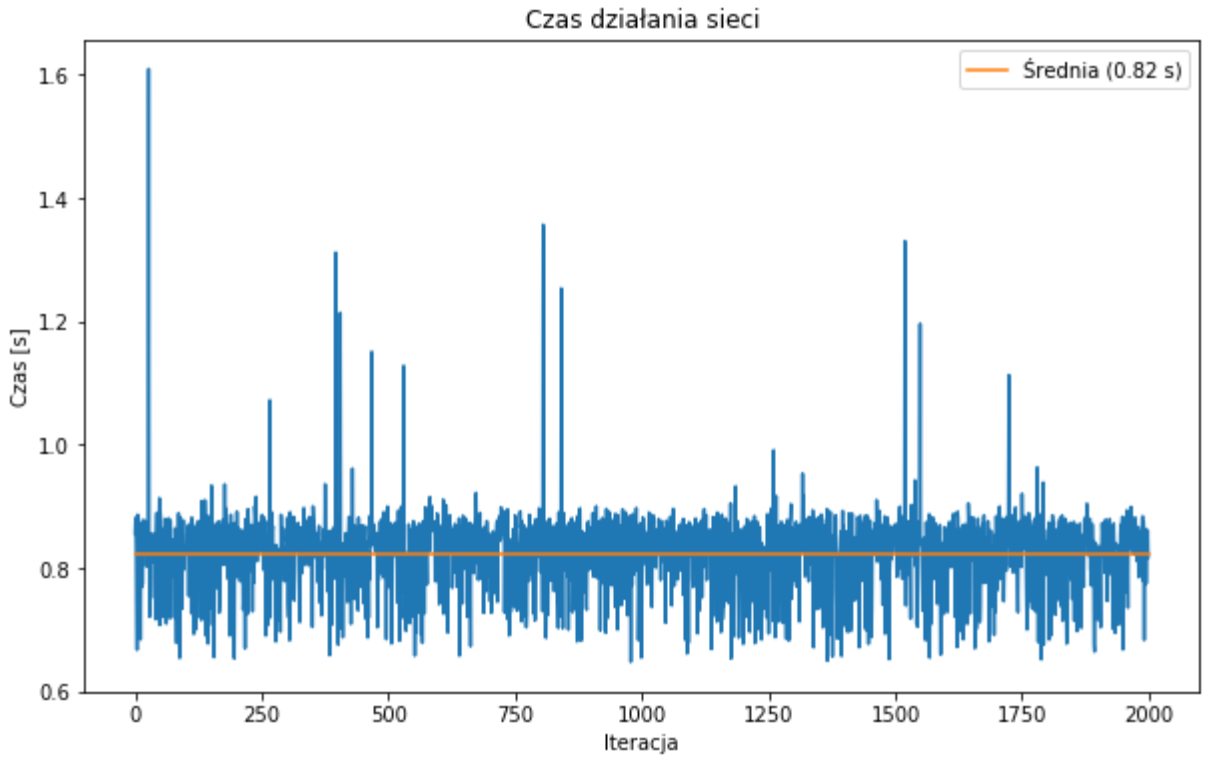
3.1.4.3.2. InceptionV3

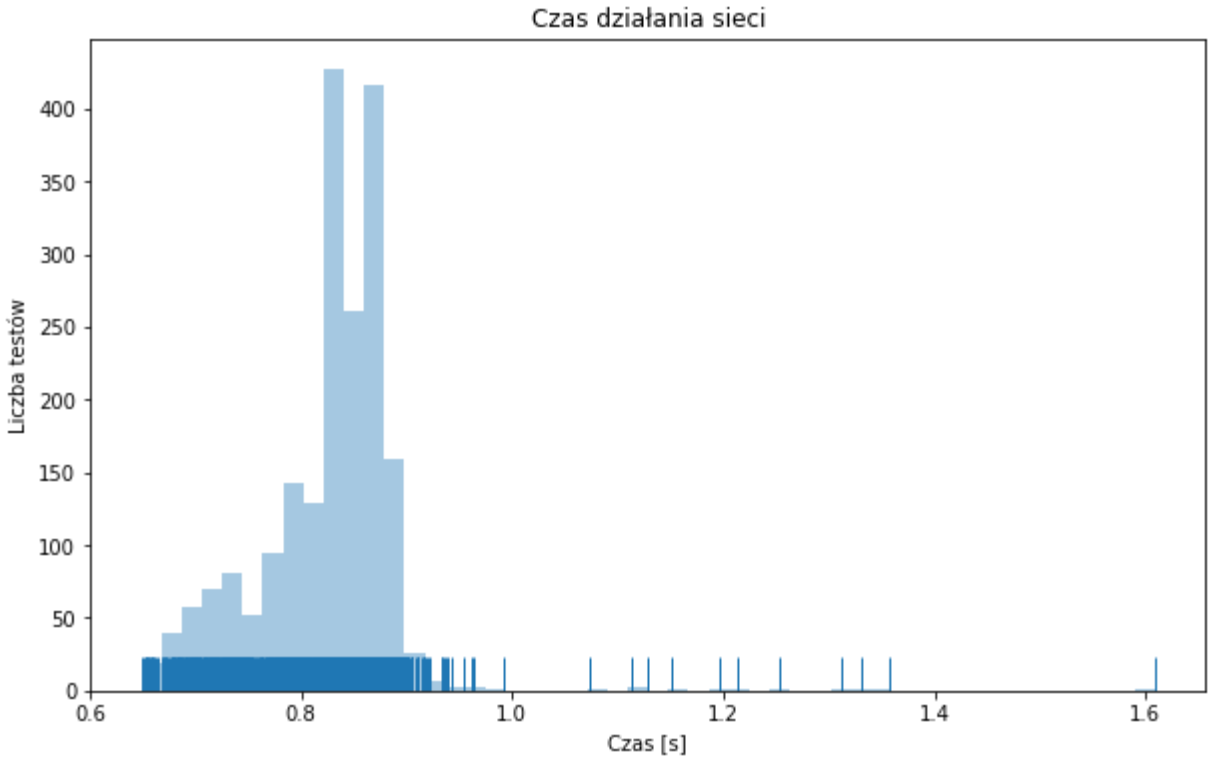


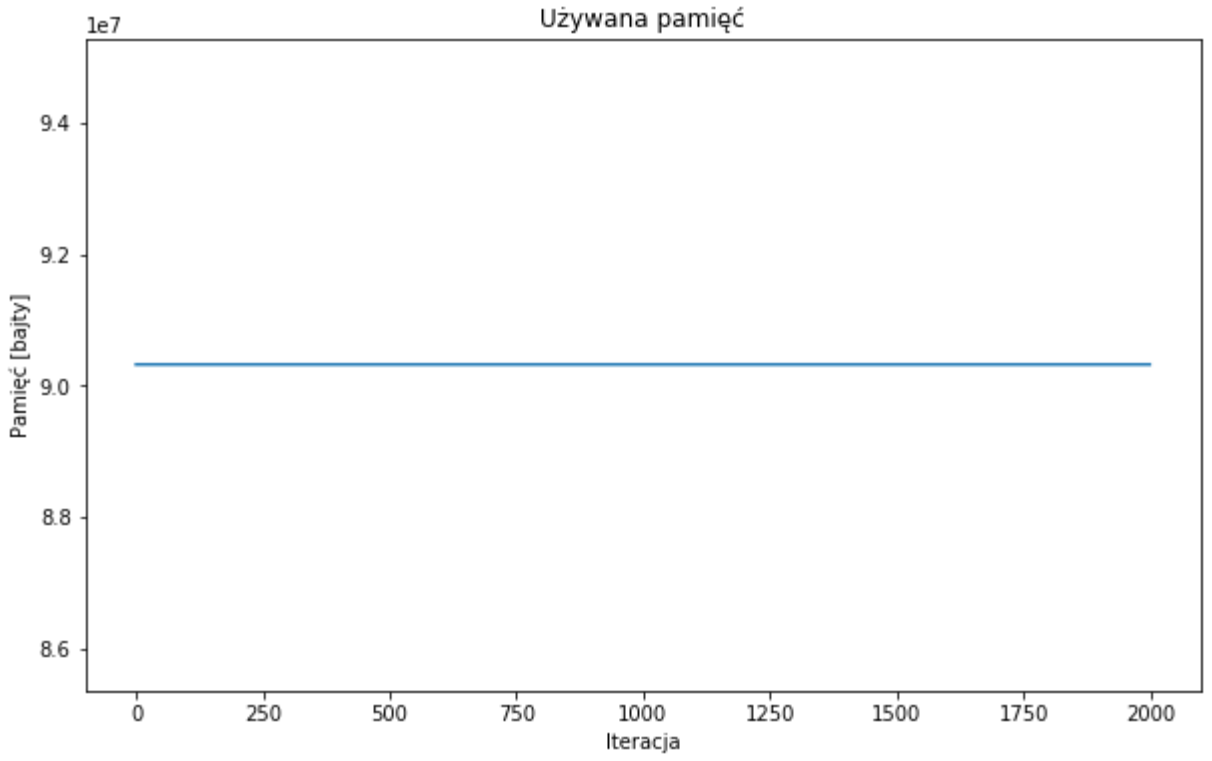




3.1.4.3.3. ResNet50




Czas działania sieci
Liczba testów
Czas [s]
0
50
100
150
200
250
300
350
400
0.6
0.8
1.0
1.2
1.4
1.6


Używana pamięć
1e7
Pamięć [bajty]
Iteracja
8.6
8.8
9.0
9.2
9.4
0
250
500
750
1000
1250
1500
1750
2000

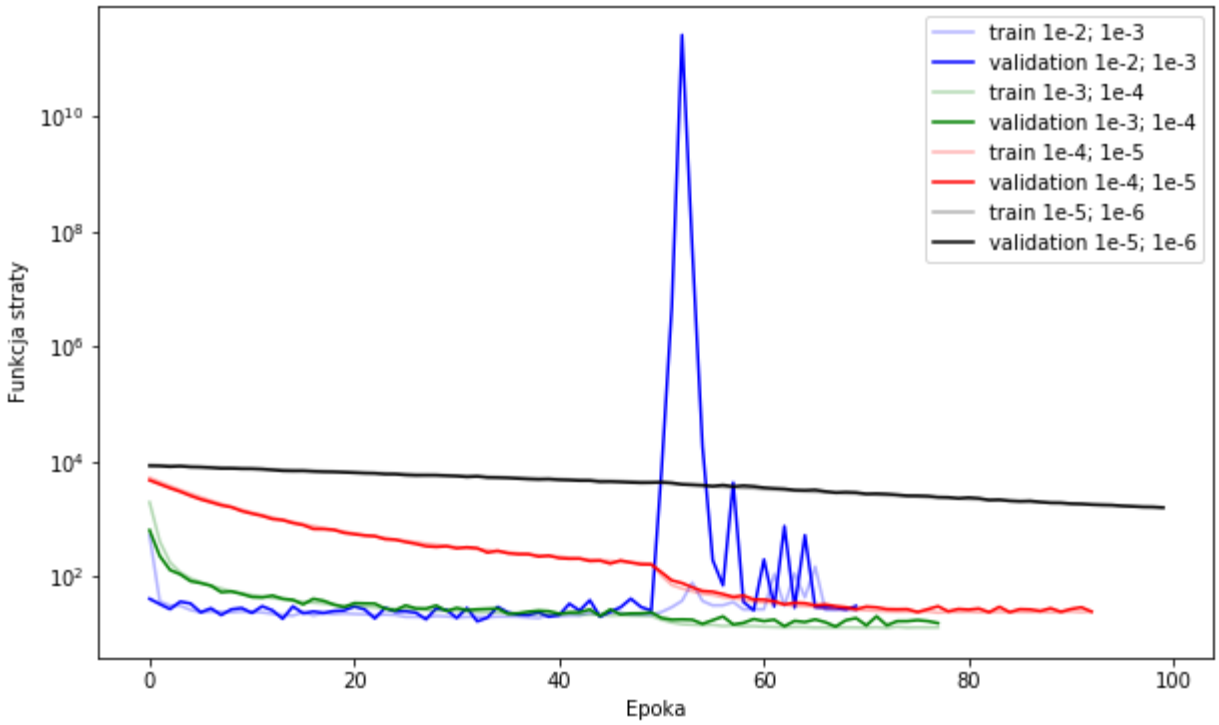
3.1.4.3.4. Podsumowanie

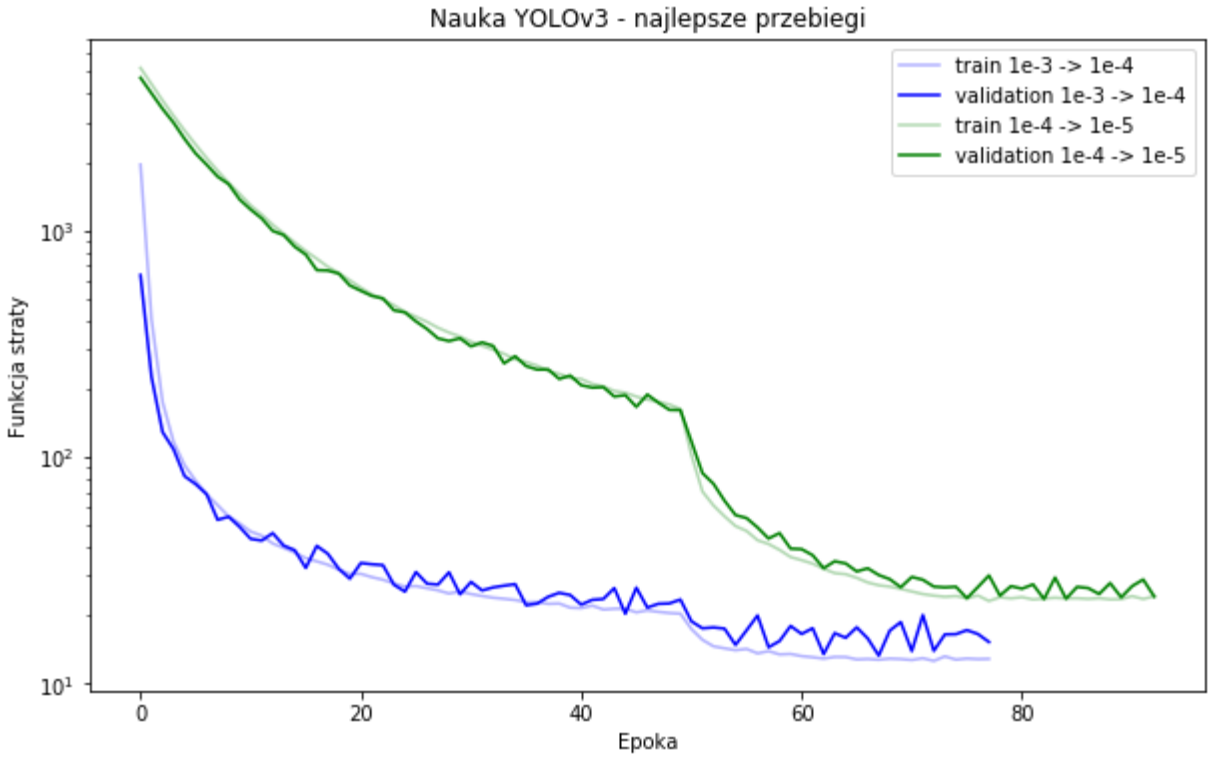
Nazwa architektury	Średni czas działania sieci [s]	Maksymalne zużycie pamięci operacyjnej [MB]
VGG16	2.76	**57,52**
InceptionV3	1.37	89,48
ResNet50	**0,82**	86,14

3.2. Badania architektur B

3.2.1 Nauka

Podczas nauki modeli z rodziny YOLO minimalizowana jest funkcja strat tzw. Yolo Loss, jej dokładny opis można znaleźć w publikacji *You Only Look Once: Unified, Real-Time Object Detection* [8].

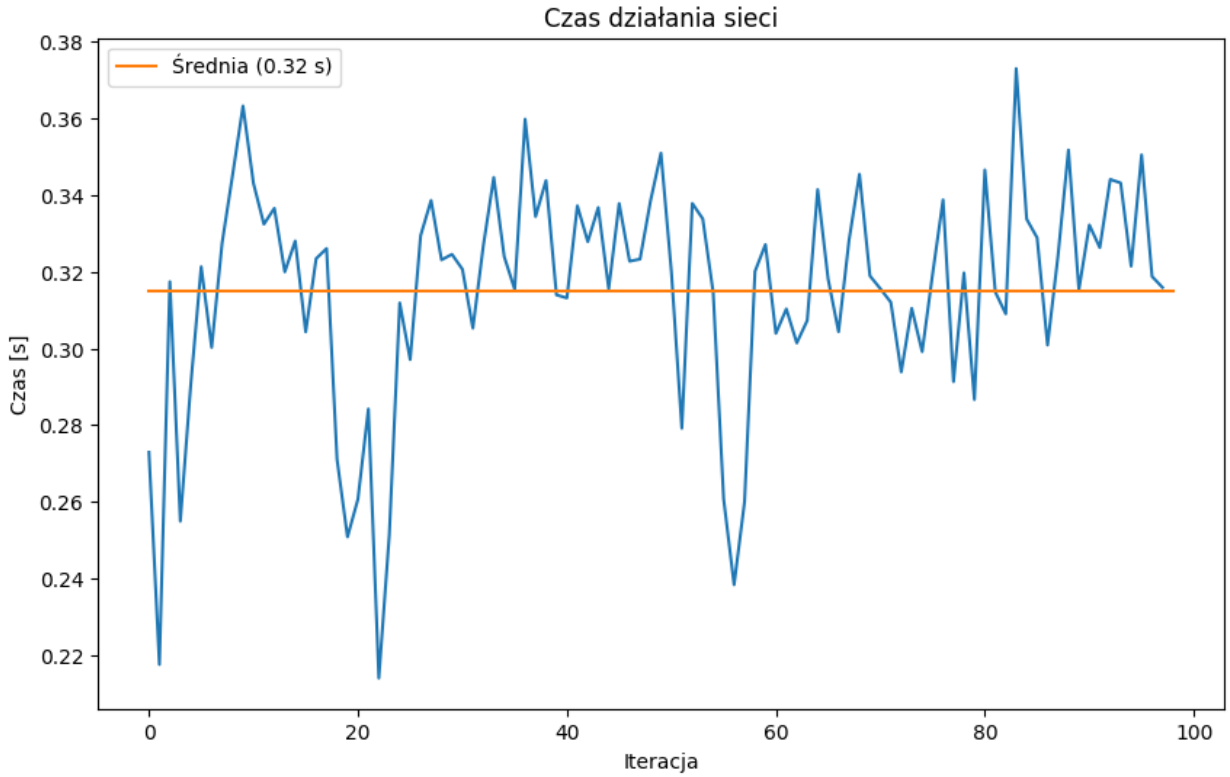


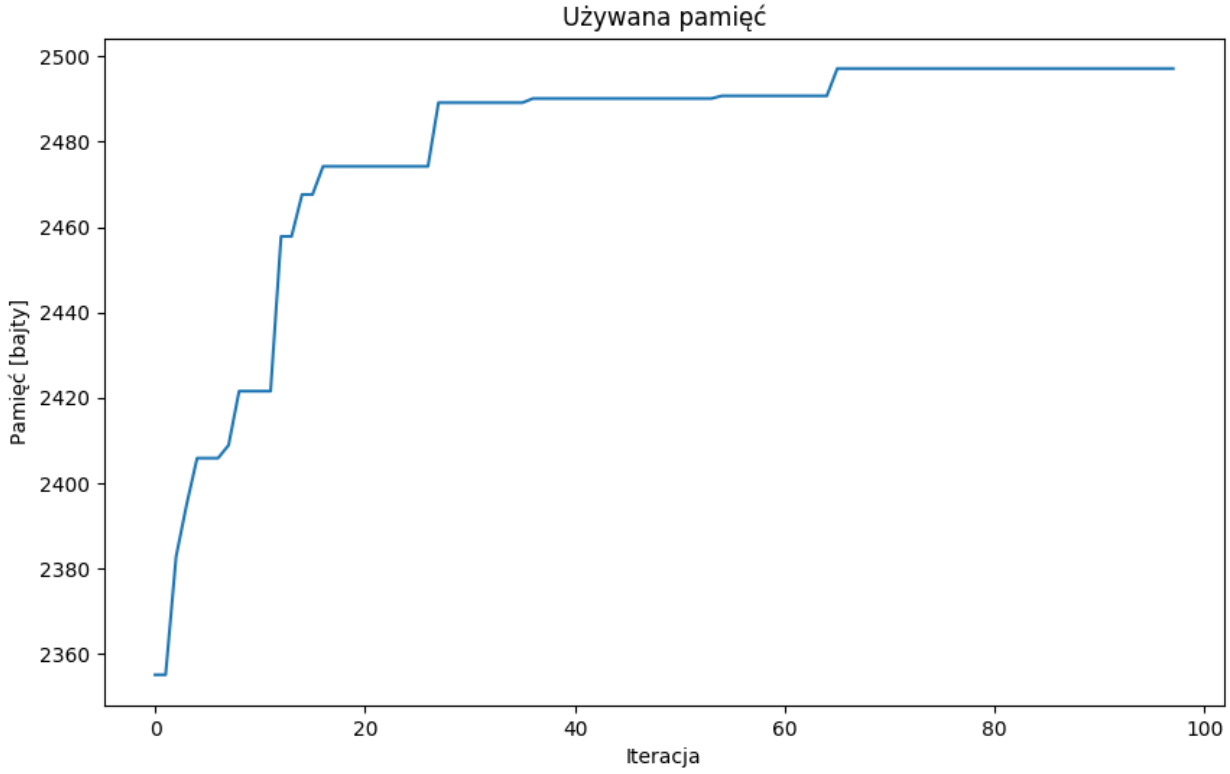


Learning rate	**Minimalna wartość funkcji straty dla zbioru walidacyjnego**
1e-2; 1e-3	16.41
1e-3; 1e-4	**13.31**
1e-4; 1e-5	23.64
1e-5; 1e-6	1570.54

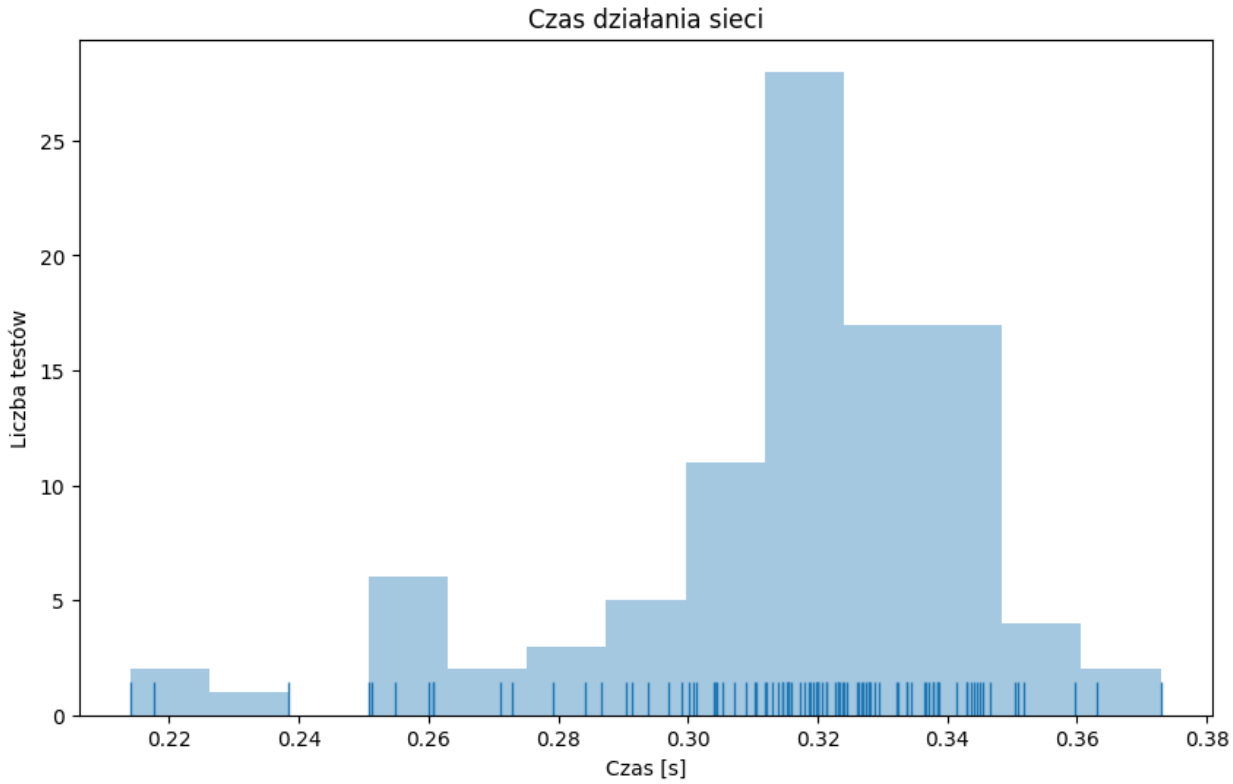
3.2.2. Wydajność

3.2.2.1. CPU

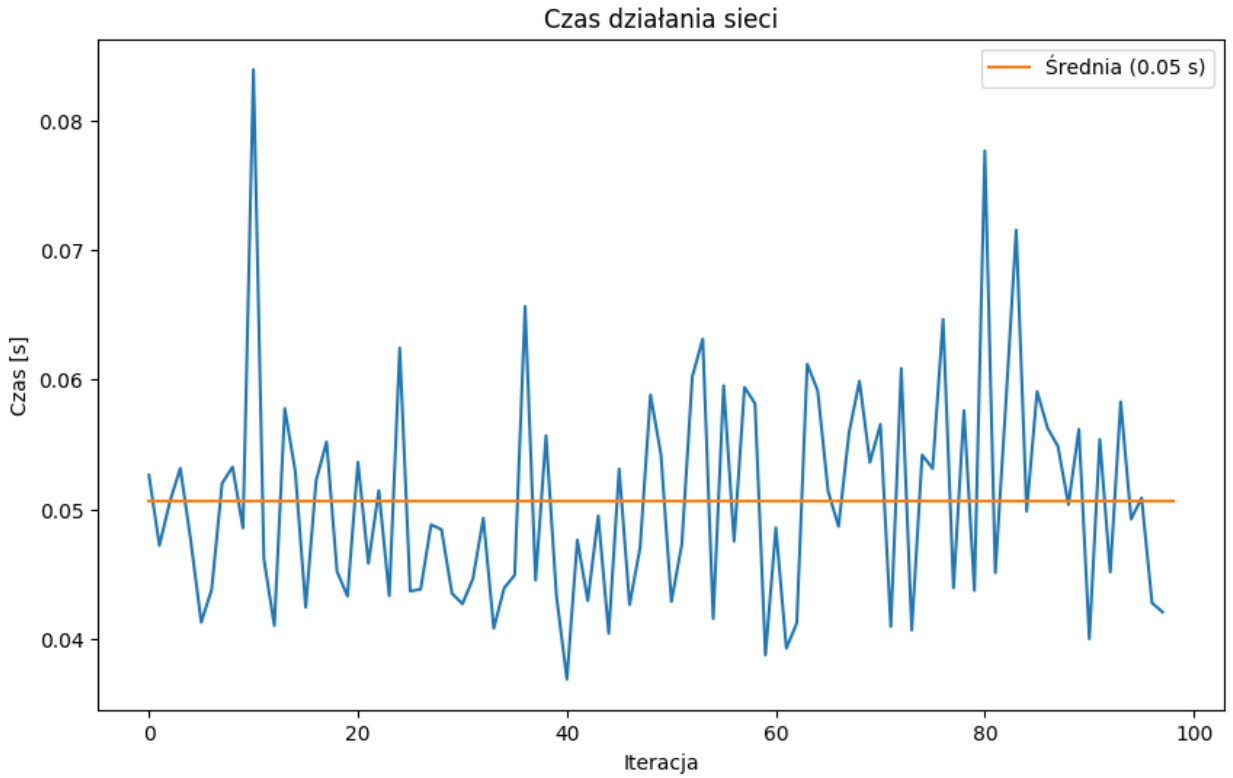


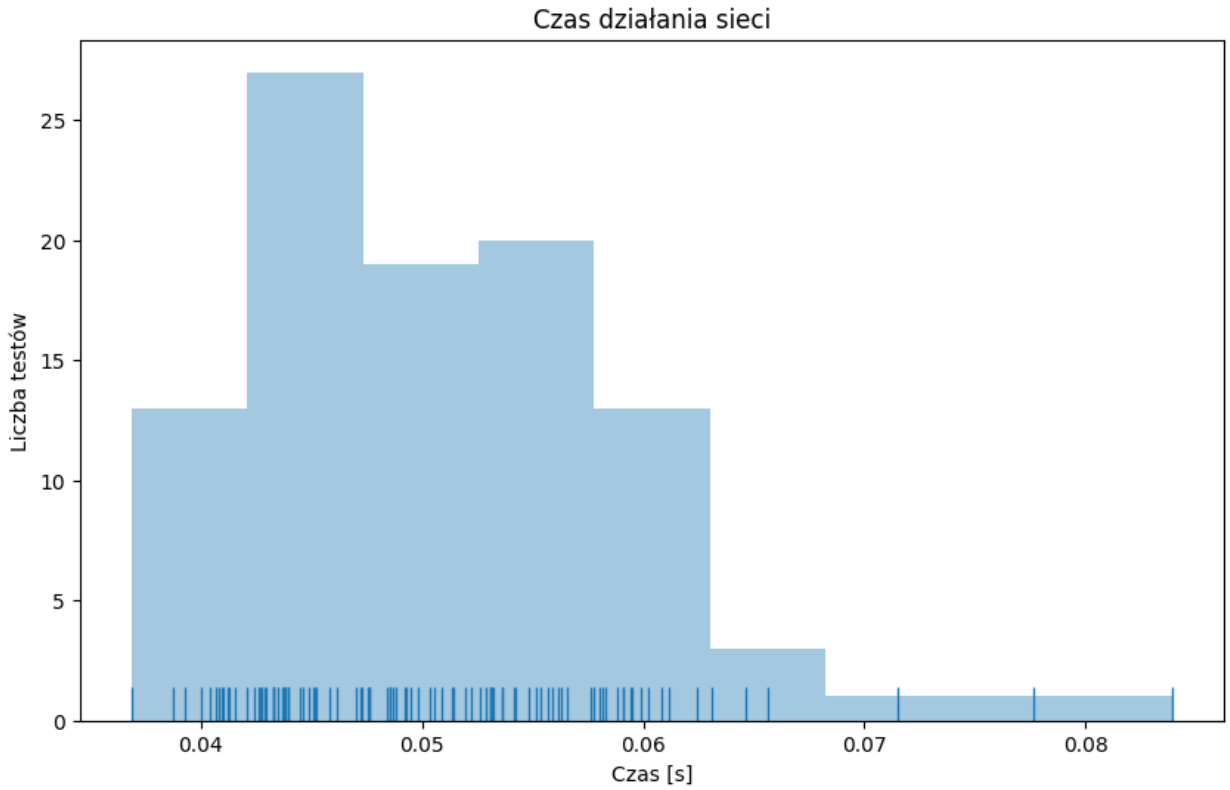


Maksymalne zużycie pamięci: 2497MB


Czas działania sieci
Liczba testów
Czas [s]
0
5
10
15
20
25
0.22
0.24
0.26
0.28
0.30
0.32
0.34
0.36
0.38

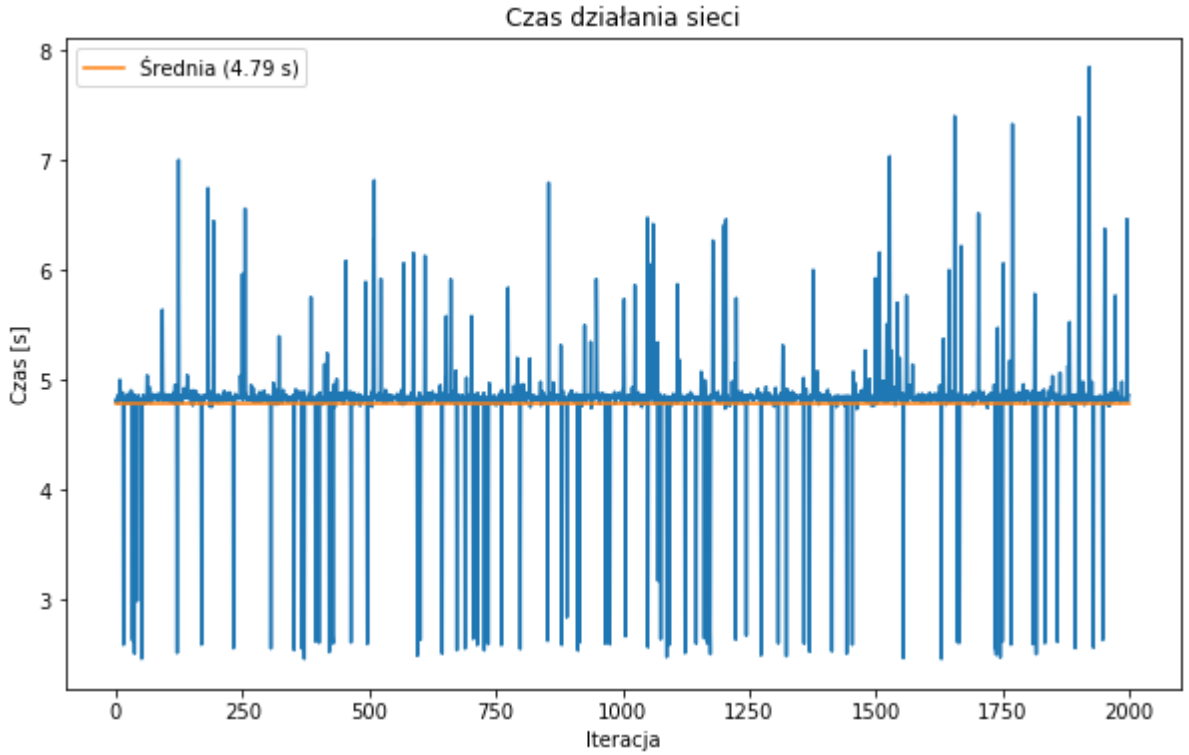
3.2.2.2. GPU

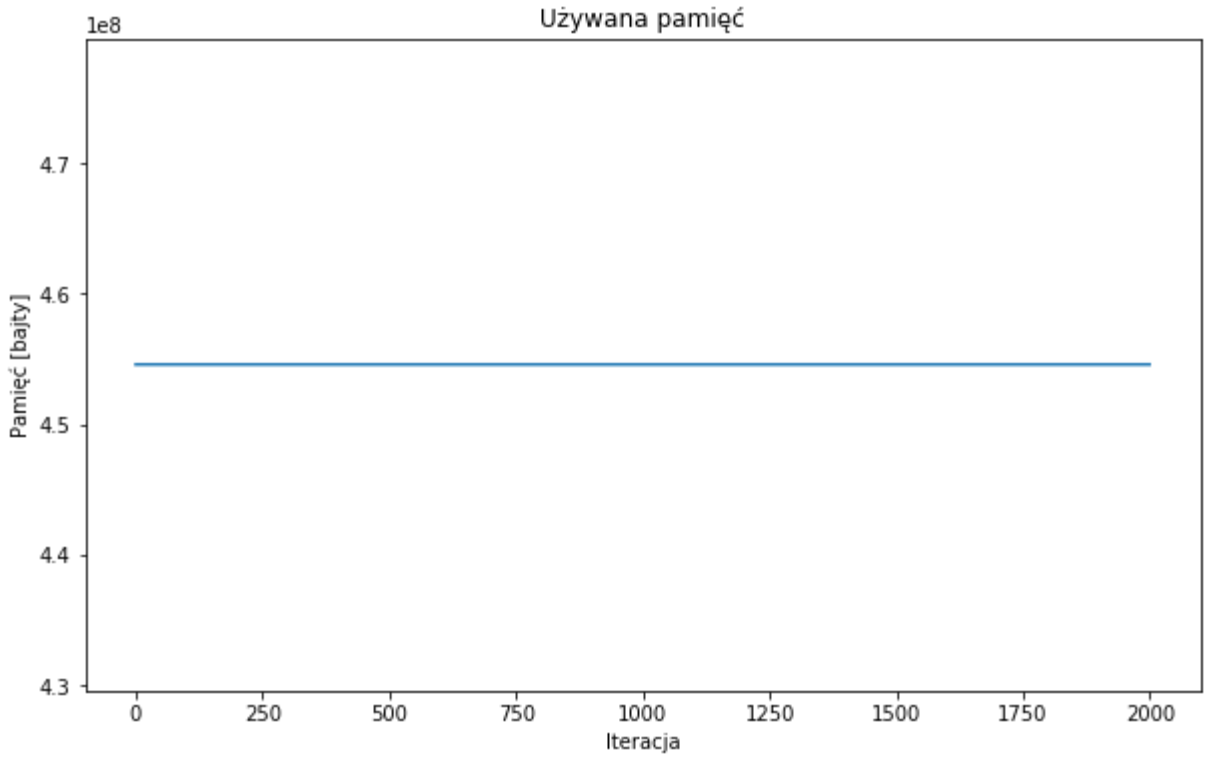

Czas działania sieci
Średnia (0.05 s)
Czas [s]
Iteracja
0.04
0.05
0.06
0.07
0.08
0
20
40
60
80
100

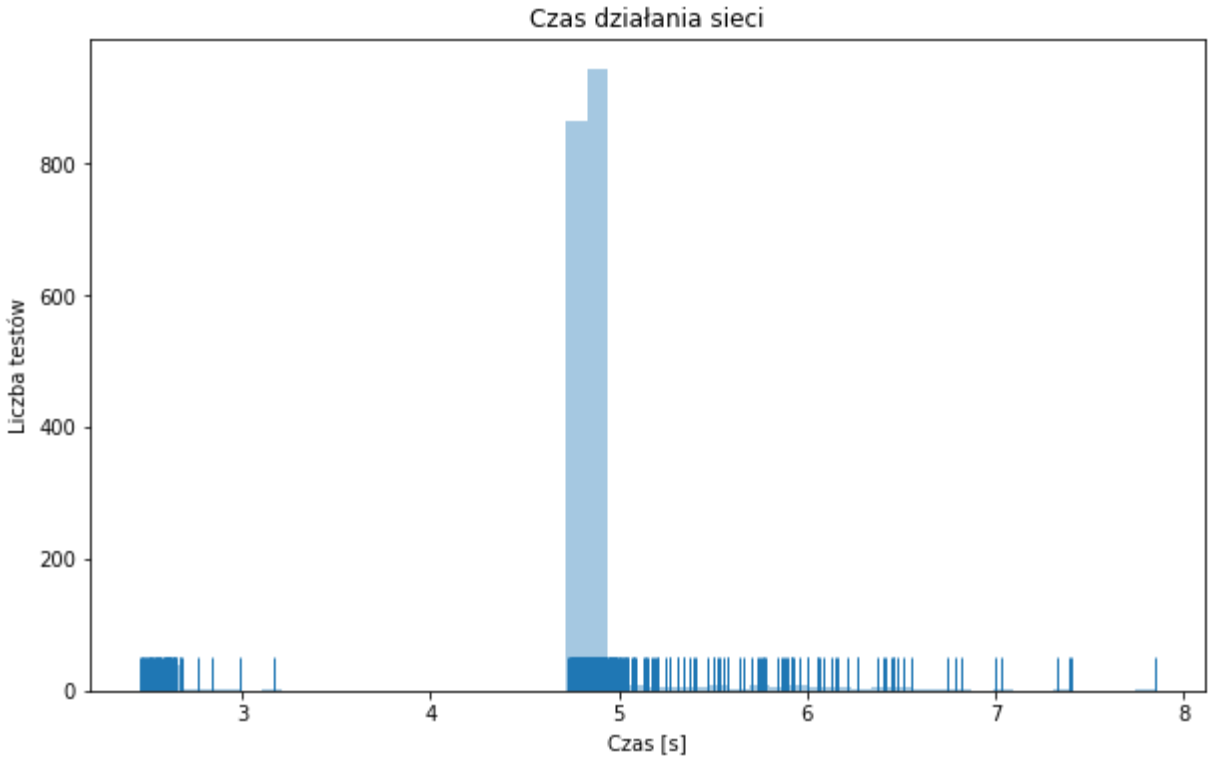


Maksymalne zużycie pamięci VRAM: 7555MB

3.2.2.3. Mobile

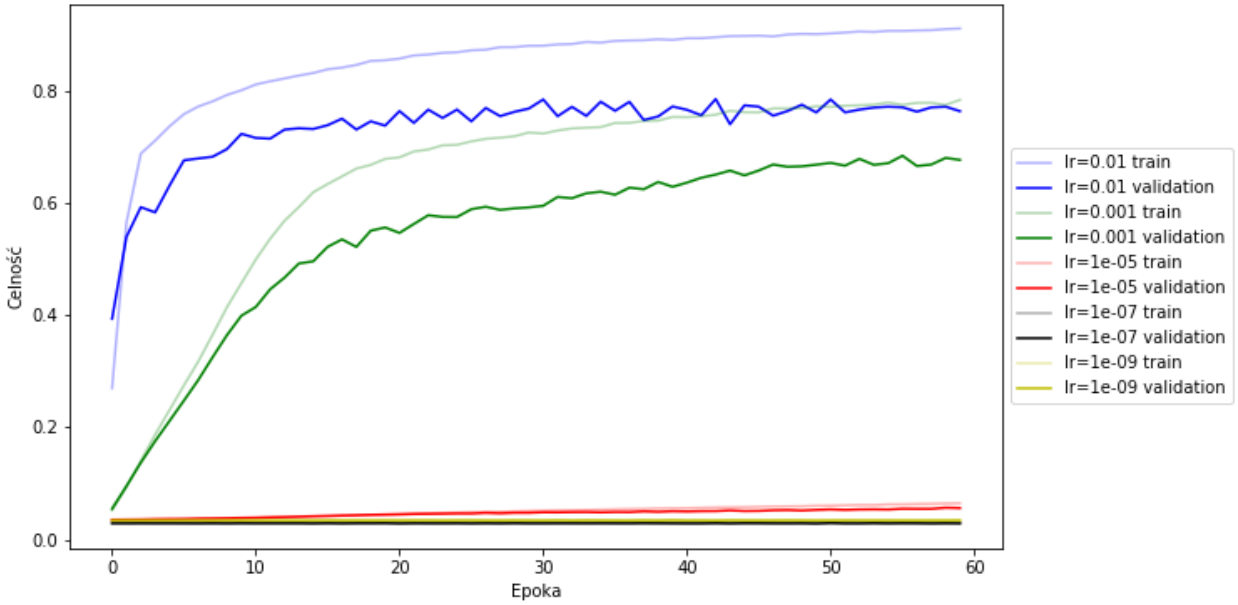


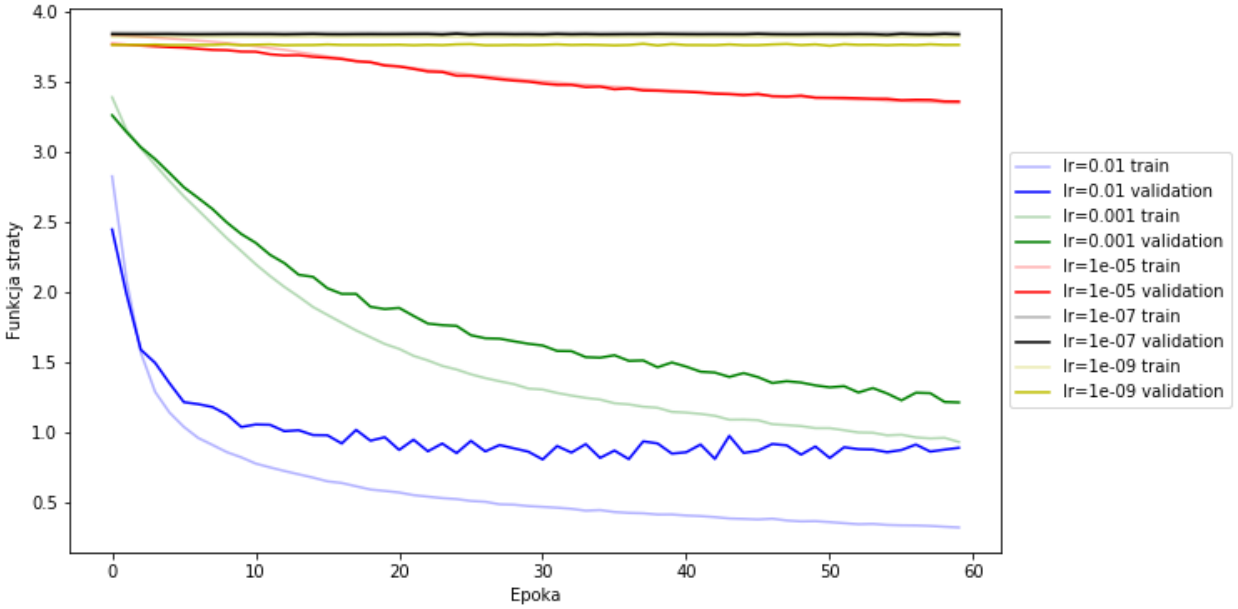

Używana pamięć
1e8
4.7
4.6
4.5
4.4
4.3
Pamięć [bajty]
0
250
500
750
1000
1250
1500
1750
2000
Iteracja


Czas działania sieci
800
600
400
200
0
Liczba testów
3
4
5
6
7
8
Czas [s]

3.3. Badania architektur C

3.3.1. Nauka



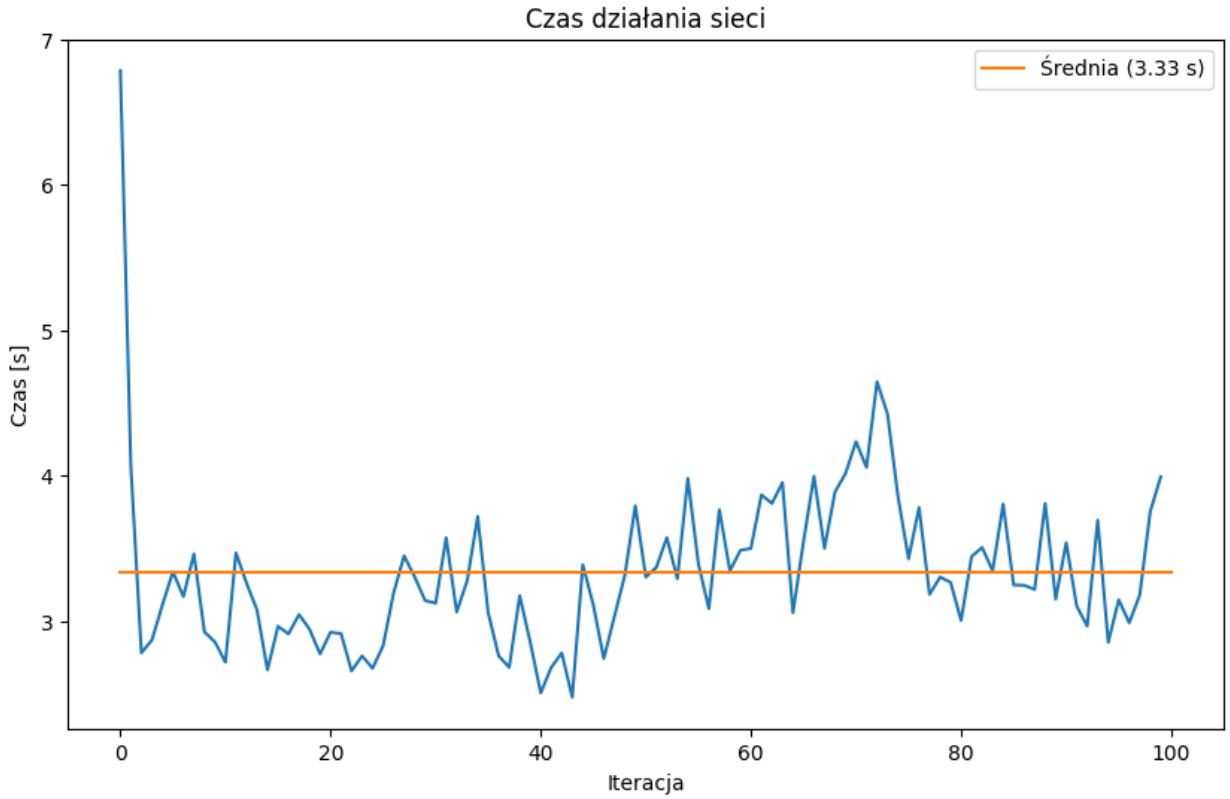


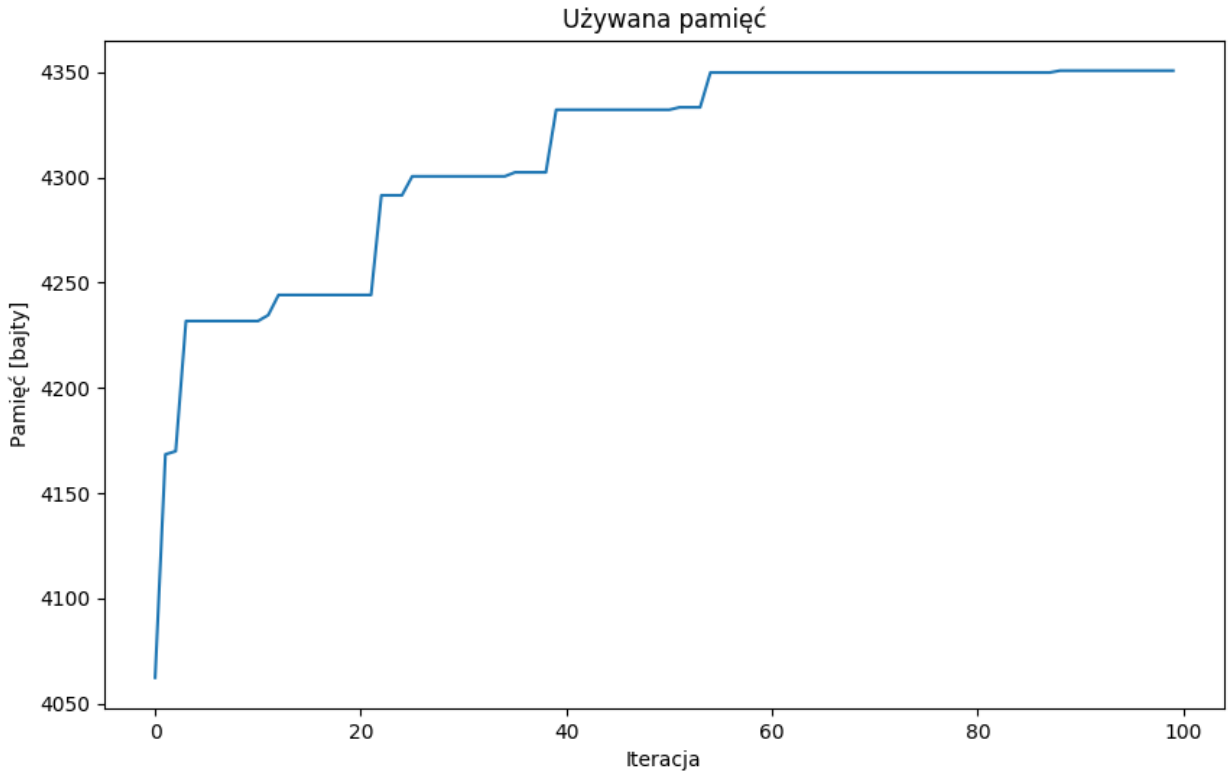
Learning rate	**Celność podczas walidacji**	**IoU***
0,01	0.90887	0.83468
0,001	0.79354	0.66032
0,00001	0.066231	0.03425
0,0000001	0.028684	0.01455

*IoU - (Intersection over Union) część wspólna przez sumę ramek ograniczających

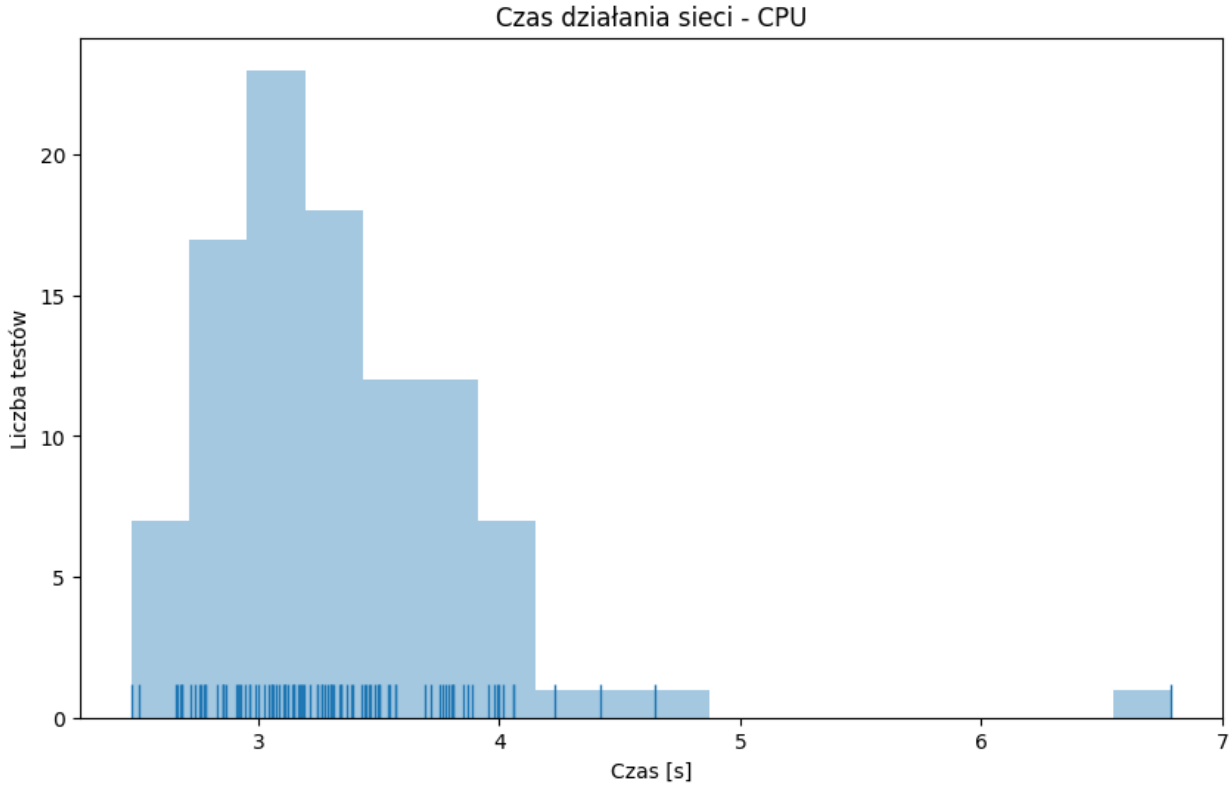
3.3.2. Wydajność

3.3.2.1 CPU

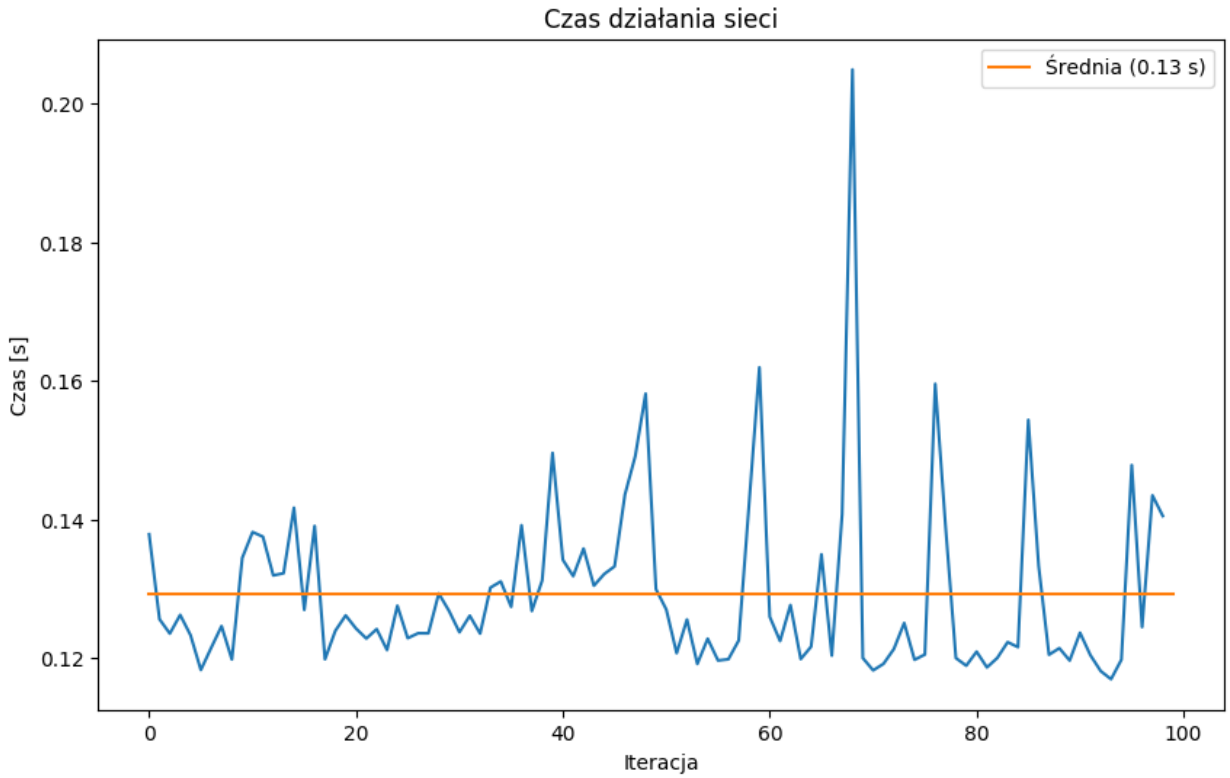


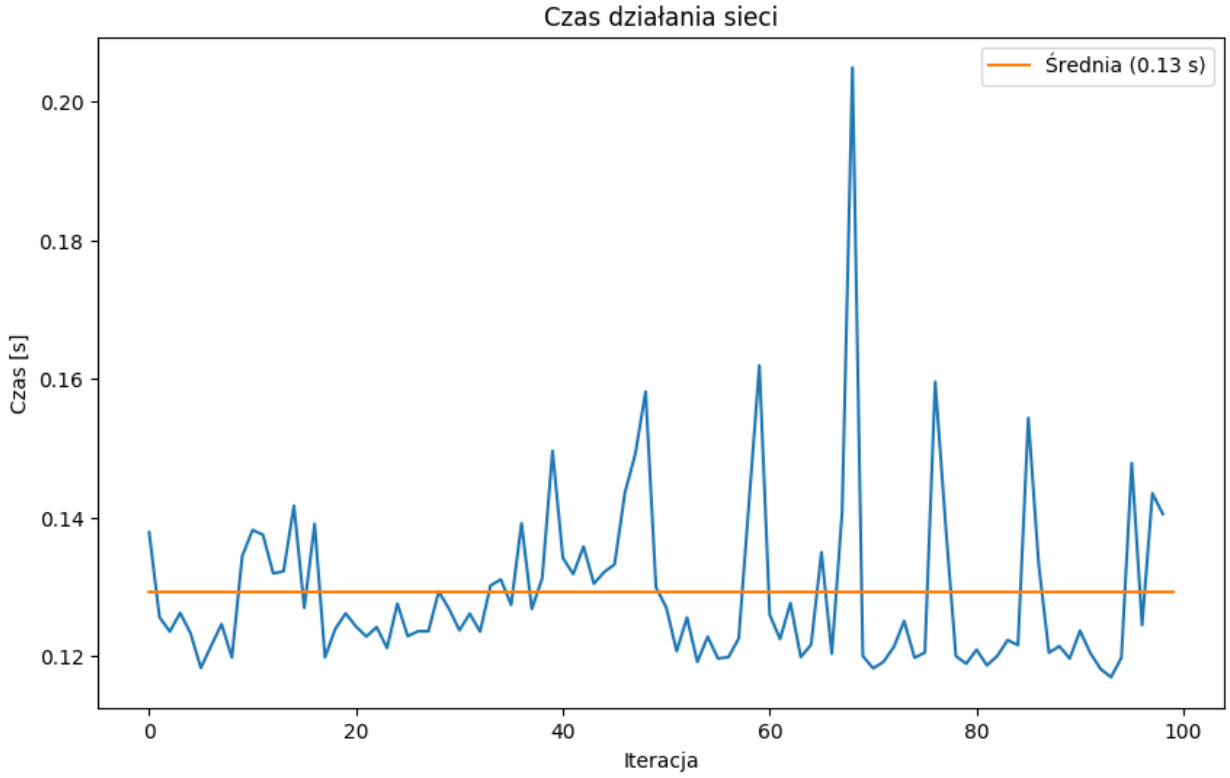


Maksymalne zużycie pamięci: 4351MB



3.3.2.2 GPU





Maksymalne zużycie pamięci VRAM: 4951MB

3.4. Implementacja dedykowanego oprogramowania umożliwiającego automatyczne pozyskanie zasobów internetowych.
Dokonano przeglądu szeregu API serwisów do przechowania zdjęć i zdecydowano się na serwis Flickr ze względu na bardzo szeroki zasób zdjęć oraz na łatwość implementacji z wykorzystaniem oficjalnego API i dostępnej pakietu flickrapi dla języka Python.
Zaimplementowane narzędzie pozwala na pobranie zdjęć dla określonych fraz wyszukiwania, tagów czy licencji zdjęć.

4. Wnioski z przeprowadzonego badania
W przypadku architektur A po procesie transfer learningu najbardziej skuteczna okazała się architektura ResNet50, nie wiele gorsza VGG16, a znacznie gorzej InceptionV3. Podczas dalszych badań jednak na jaw wyszedł problem z implementacją warstwy BatchNormalization w bibliotece Keras[9][10][11], który uniemożliwił dostrojenie modelu. To mogło wpłynąć na jakość klasyfikacji już podczas pierwszego etapu nauki, ponieważ architektury InceptionV3 oraz ResNet50 posiadają właśnie takie warstwy. Eksperyment w dalszej części został zmodyfikowany w ten sposób by ominąć ten problem i spowodowało to znaczną poprawę wyników InceptionV3 i ResNet50 względem VGG16. Doszkalanie modeli do nowego zadania dało bardzo pozytywne wyniki, modele osiągały bardzo wysokie wyniki zaledwie po kilku epokach nauki poza VGG16 którego proces nauki przypominał ten z pierwszego eksperymentu transfer learningu. Oznacza to, że nawet w bardzo dynamicznie zmieniającym się asortymencie sklepu nie będzie kłopotu z dostosowywaniem sieci neuronowej. Ponieważ czas klasyfikacji dla wszystkich modeli niezależnie od tego w jakim środowisku zostały uruchomione (CPU, procesor graficzny i urządzenie mobilne) jest akceptowalny na tym etapie projektu to dalsze badania będą się skupiać głównie na architekturach InceptionV3 oraz ResNet50 z naciskiem na ResNet50 w związku z tym, że zgodnie ze publikacją [12] sprawdza się lepiej w zadaniu rekomendacji obrazów, które jest kluczowe w projekcie.
Badanie YOLOv3, architektury wybranej do zadania detekcji obiektów dowiodło, że już niewielka ilość zdjęć pozwala w zadowalający sposób wytrenować model do nowej klasy. Oznacza to, że dostosowywanie modelu do nowych systemów rekomendacji dla innych sklepów i produktów będzie względnie łatwe. W drugim etapie projektu prace w przypadku tej architektury będą skupiać na zmniejszeniu czasu działania sieci na urządzeniu mobilnym, ponieważ znacznie odstaje od architektur A i mógłby być niezadowalający dla użytkownika końcowego.
SegNet mimo że osiągnął zadowalające wyniki podczas nauki, okazał się trudny do uruchomienia na urządzeniu mobilnym. Biorąc pod uwagę jednak czas działania sieci na CPU i GPU, który był odpowiednio 10 i 2.5 raza dłuższy niż w przypadku YOLOv3, można podejrzewać, że będzie zbyt czasochłonnym procesem by mógł odbywać się na urządzeniu mobilnym. Dodatkową przeszkodą jest czasochłonność sporządzenia zbioru danych do nauki takiego modelu. W następnych etapach projektu rozważana będzie słuszność wykonywania zadania segmentacji w zadaniu rekomendacji oraz poszukiwanie rozwiązań zastępczych takich jak np. segmentacja wododziałowa (ang. watershed segmentation).

5. Zgodność realizacji działania z założeniami z Planu B+R

Powyższe badania oraz inne działania spełniły założenia Planu B+R, zrealizowano założenia kamieni milowych:

- Raport podsumowujący wyniki badań i rekomendacje dla proponowanego prototypu (do realizacji w etapie 2).
- Baza obrazów potrzebna do tworzenia i testowania modeli.
- Implementacja testowanych modeli oraz modułu wstępnego przetwarzania danych.
- Wskaźnik rezultatu: Liczba przetestowanych architektur sieci = min. 5
- Wskaźnik rezultatu: Liczba przeprowadzonych eksperymentów obliczeniowych = 16
- Wskaźnik rezultatu: Liczba pozyskanych obrazów uczących = 20000

6. Bibliografia

[1] West, Jeremy; Ventura, Dan; Warnick, Sean (2007). "Spring Research Presentation: A Theoretical Foundation for Inductive Transfer". Brigham Young University, College of Physical and Mathematical Sciences. Archived from the original on 2007-08-01. Retrieved 2007-08-05.

[2] Leinweber, D. J. (2007). "Stupid data miner tricks". The Journal of Investing. 16: 15–22. doi:10.3905/joi.2007.681820

[3] http://www.image-net.org/

[4] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[5] https://keras.io/optimizers/

[6] https://www.tensorflow.org/guide/using_gpu#allowing_gpu_memory_growth

[7] http://cocodataset.org/

[8] You Only Look Once: Unified, Real-Time Object Detection https://arxiv.org/pdf/1506.02640.pdf

[9] https://github.com/keras-team/keras/issues/6977

[10] The Batch Normalization layer of Keras is broken, http://blog.datumbox.com/the-batch-normalization-layer-of-keras-is-broken/

[11] Strange behaviour of the loss function in keras model, with pretrained convolutional base, https://stackoverflow.com/questions/51123198/strange-behaviour-of-the-loss-function-in-keras-model-with-pretrained-convoluti/

[12] Do Better ImageNet Models Transfer Better... for Image Recommendation?, https://arxiv.org/abs/1807.09870

Poznań, 29.03.2019r.

.....

(miejscowość, data sporządzenia raportu, podpis autorów)

Poznań, 29.03.2019r.

.....

(miejscowość, data sporządzenia raportu, podpis pracodawcy)