

令和4年度卒業研究報告書

GPUを用いた Deep learning の性能比較, 検証

情報技術科 石川 巧弥

指導教員 菅野 研一

目次

第 1 章	はじめに	3
第 2 章	使用技術	4
2.1	コンテナ	5
2.1.1	メリット	7
2.1.2	デメリット	7
2.2	Docker	8
2.2.1	Docker イメージ	8
2.2.2	nvidia-docker	8
2.3	GPU	9
2.4	ライブラリ	10
2.4.1	TensorFlow	10
2.4.2	Time	10
2.4.3	Torch	10
2.5	ハードウェア	11
第 3 章	環境構築	12
3.1	初期設定	12
3.1.1	NGC サイトにサインインする	12
3.1.2	API KEY を生成する	12
3.1.3	NGC の Docker へログインする	13
3.1.4	利用したい Docker イメージを取得する	14
3.2	コンテナ起動 (TensorFlow)	15
3.3	コンテナにローカルのボリュームをマウントする	16

3.4 X Window System の導入.....	17
3.5 カメラ接続.....	19
3.6 共有メモリ量設定.....	20
3.7 コンテナコミット方法.....	21
第 4 章 処理時間計測.....	22
4.1 計測用サンプルデータ.....	22
4.1.1 手書き数字の画像認識.....	22
4.1.2 物体検知用キャラクター学習.....	23
4.2 計測方法・計測場所.....	24
4.2.1 手書き数字の画像分類.....	25
4.2.2 物体検知用キャラクター学習.....	26
4.3 計測実行.....	28
4.3.1 手書き数字の画像分類.....	29
4.3.2 物体検知用キャラクター学習.....	31
第 5 章 結果・考察.....	34
5.1 計測結果.....	34
5.1.1 手書き数字の画像分類.....	34
5.1.2 物体検知用キャラクター学習.....	36
5.2 考察.....	37
第 6 章 おわりに.....	38
参考文献.....	39

第1章 はじめに

今年度、学校に高性能な GPU を搭載している、AI 専用 PC が導入された。導入されたばかりのため、この PC の利用例はまだ存在しない。そこで、この PC の環境を整え、性能を検証することが、今後の産技短での専用 PC の利用や AI 開発の足掛かりになると考え、この研究に取り組むことにした。

研究の目的は、専用 PC に搭載されている GPU の処理速度を検証し、CPU や他の GPU と比較する。処理させるサンプルデータは Deep Learning を用いたプログラムの学習部分を利用する。サンプルデータは複数用意し、使用する GPU も Google Colaboratory を加えて処理速度を検証していく。

第2章 使用技術

開発環境と Deep learning 専用 PC の情報を表 2-1 と表 2-2 に示す.

OS	Ubuntu 18.04.5 LTS
使用言語	Python 3.8
使用ライブラリ	TensorFlow
コンテナエンジン	Docker

表 2-1 開発環境

GPU	GeForce RTX3090
CPU	Intel Xeon W-2223@3.6Ghz
RAM	32GB
SSD	1×2TB

表 1-2 Deep learning 専用 PC の情報

2.1 コンテナ

コンテナとはアプリケーションと実行環境を1つにまとめ上げる技術である。アプリケーションの動作に必要なホスト OS の基本環境（カーネル）を Docker のようなコンテナエンジンを通して、コンテナ同士が共有できるようにすることで、CPU やメモリなどのハードウェアのリソースと切り離し仮想的な環境を作り出す。^[1]

仮想マシンとコンテナの構成の違いとしては、環境内で使用している OS にある。仮想マシンはハイパーバイザというソフトウェアで立ち上げられるハードウェアの環境にゲスト OS やアプリケーションのインストールを行い、ホスト OS とは別の実行環境を構築している。（図 2-1）

一方、コンテナはコンテナエンジンを通して共有されたホスト OS のリソースに、ゲスト OS 不要でアプリケーションを追加することができる。つまり、ホスト OS で実行環境を構築している。（図 2-2）

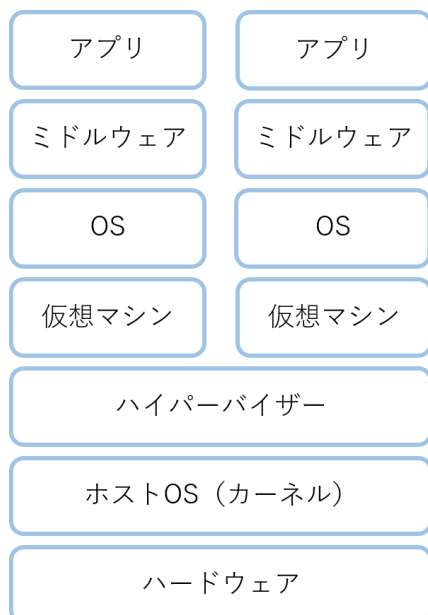


図 2-1 仮想マシン環境

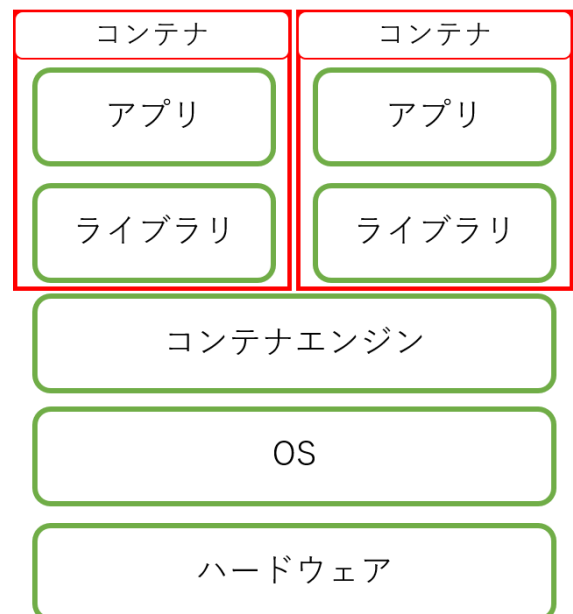


図 2-2 コンテナ環境

本研究環境を図 2-2 に当てはめると以下のようなイメージになる。(図 2-3)

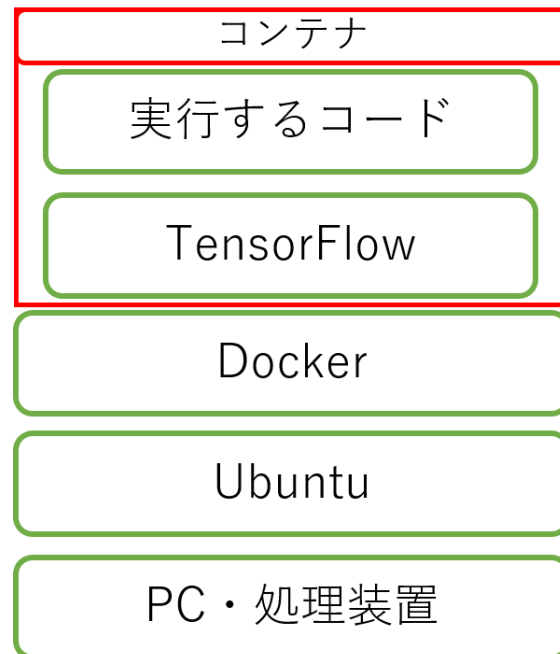


図 2-3 本研究のコンテナ環境

2.1.1 メリット

- (1) コンテナ化されたアプリケーションは OS との依存関係がないため、移行性が高く、あらゆるプラットフォームやクラウドで一貫性のある均一な動作が可能になる。
- (2) コンテナ内に OS が入っていないため、機能が軽量となり、起動時間が短縮される。
- (3) コンテナは独立しているため、アプリケーションをコンテナ化すると、アプリケーションが分離される。そのため、一方に障害が発生しても、他のコンテナには影響しない。

[2]

2.1.2 デメリット

- (1) コンテナを管理、運用するのに複雑化しやすい。
- (2) コンテナ自体はホスト OS に依存するため、同一環境上で異なる OS を動かすことができない。

(本研究環境ではホスト OS は Ubuntu なので、それ以外の OS は動かせない。)

- (3) コンテナ化はセキュリティ面でリスクが高くなる。ホスト OS に障害が起きた際には、生成したコンテナすべてに影響が出るので、セキュリティレベルを高くする必要がある。 [2]

2.2 Docker

Docker とは Docker 社が開発している、コンテナ型の仮想環境を作成、配布、実行するためのプラットフォームのことである。環境やアプリケーションを Docker イメージとして保存し、Docker エンジンにより Docker コンテナとして配備、実行できる。^[3]

2.2.1 Docker イメージ

Docker イメージとは、Docker コンテナの動作環境となるテンプレートファイルである。Docker コンテナを実行するためには、Docker イメージが必要となる。今回の環境では NVIDIA の NGC サイトから Docker イメージを取得している。

2.2.2 nvidia-docker

今回の環境では nvidia-docker を利用している。nvidia-docker とは、NVIDIA の GPU とドライバがインストールされたマシン上で稼働するように開発された、Docker コンテナプラグインである。コンテナ内に CUDA や cuDNN などライブラリがインストールされており、機能拡張でコンテナ内から GPU を使用することが可能になる。^[4]

2.3 GPU

GPU とは「Graphics Processing Unit」の略でリアルタイム画像処理に特化した演算装置あるいはプロセッサのことである。並列処理能力に優れており、複数のタスクを一度にこなすことができる。CPU との違いとしては処理をする対象と、スピードに違いがある。CPU は汎用的な処理を行うことが目的であるが、GPU は高速な画像処理を行うことが目的である。また、GPU は単純な処理をするのが得意であるため、機械学習のトレーニング期間の短縮が可能である。^[5]

今回使用した GPU は NVIDIA の GeForce RTX3090 である。CUDA コア数は 10752、メモリサイズは 24GB ある。また、AI 処理に特化した Tensor コアと、よりリアルなグラフィックスを表示するレイトレーシングコアを搭載している。(図 2-4)^[6]

今回の研究では Google Colaboratory の GPU も用いている。Google Colaboratory は無料版を利用しており、GPU は NVIDIA Tesla T4 を用いている。

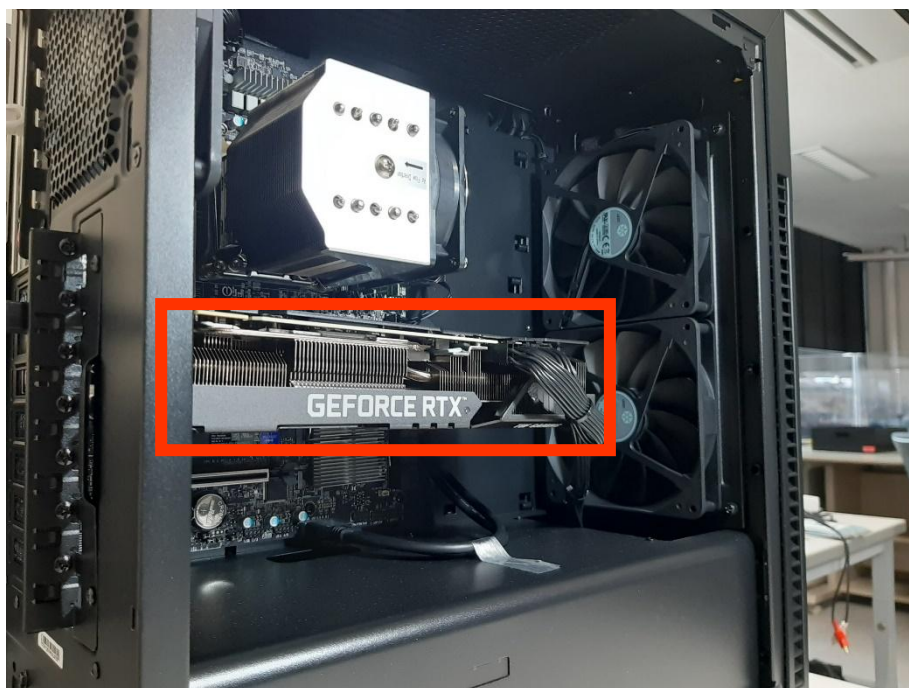


図 2-4 GeForce RTX3090 の画像

2.4 ライブラリ

2.4.1 TensorFlow

TensorFlow とは, Google が開発しオープンソースで公開している, 機械学習に用いるためのソフトウェアライブラリである. 機械学習や数値解析だけでなく, Deep Learning にも対応している. 本研究では Deep Learning を扱うため, 環境のベースとなるライブラリとして利用している. ^[7]

2.4.2 Time

time は時間を扱うためのライブラリで, システム時間の取得, 時刻のフォーマット変換などが可能である. 本研究では現在のシステムの時間を取得する `time.perf_counter` 関数を利用している. ^[8]

2.4.3 Torch

Torch は, 「機械学習ライブラリ」「科学計算フレームワーク」である. GPU を活用する機械学習アルゴリズムを幅広くサポートしている. 本研究では, 処理時間計測の際に GPU と CPU の処理を同期させる `torch.cuda.synchronize` を利用している. ^[9]

2.5 ハードウェア

CPU

今回使用したのは Intel Xeon W-2223@3.6Ghz である。この CPU はコア数が4、スレッド数が8、メモリも最大で1TB まで使用することができる。

RAM

今回の環境のメモリ量は32GB である。

SSD

今回の環境では1TB のメモリを2枚利用している。

第3章 環境構築

3.1 初期設定

3.1.1 NGC サイトにサインインする

NGC のサイトにアクセスする

URL <https://catalog.ngc.nvidia.com/>

右上の Sign in メニューを選択するとメールアドレスを入れるウィンドウが表示されるのでサインインする。

3.1.2 API KEY を生成する

Sign in に成功したら、右上のユーザー名からメニューを開き、Setup を選択する。

API の説明画面が出たら右上の Generate API Key ボタンを押す。

確認のダイアログが表示されるので、Confirm ボタンを押す。

APIKEY が生成されるのでコピーして保存しておく。

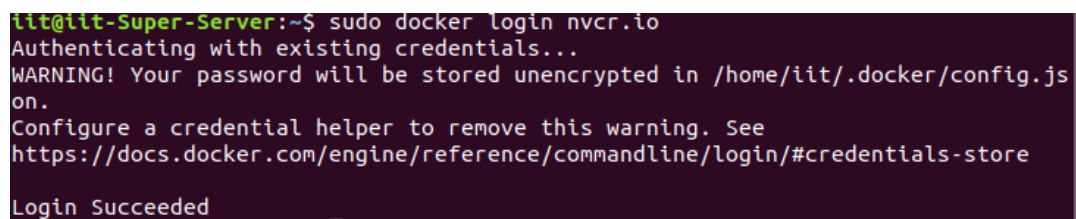
3.1.3 NGC の Docker へログインする

端末を開き，以下のコマンドを実行する．

```
sudo docker login nvcr.io
```

Username には `$oauthtoken` と入力し，Password には先ほど生成した API KEY を入力する．

成功すると以下の画面が出る．(図 3-1)

A terminal window with a dark background and light-colored text. The text shows the command 'sudo docker login nvcr.io' being executed. It then shows 'Authenticating with existing credentials...' followed by a warning message: 'WARNING! Your password will be stored unencrypted in /home/iit/.docker/config.json. Configure a credential helper to remove this warning. See https://docs.docker.com/engine/reference/commandline/login/#credentials-store'. Finally, it displays 'Login Succeeded' in green text.

```
iit@iit-Super-Server:~$ sudo docker login nvcr.io
Authenticating with existing credentials...
WARNING! Your password will be stored unencrypted in /home/iit/.docker/config.js
on.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
Login Succeeded
```

図 3-1 ログイン成功画面

3.1.4 利用したい Docker イメージを取得する

画面左上の CATALOG メニューを選択し、CONTAINERS, MODELS 等から利用したい種類の Docker イメージを選択する。(図 3-2)

そして、Overview タグに記載されている指示に従い、Docker イメージを取得して利用する。

今回は TensorFlow を選択して、指示に従いコンテナを起動した。

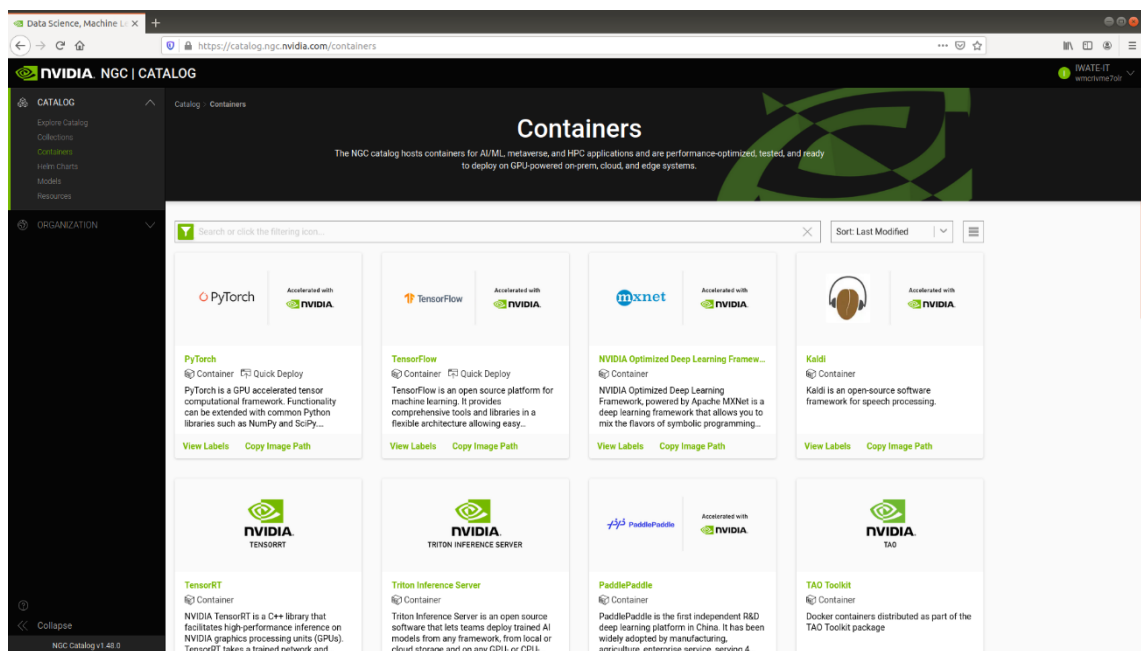


図 3-2 Docker イメージ選択画面

3.2 コンテナ起動 (TensorFlow)

以下のコマンドを実行して TensorFlow を起動する。 [10]

コンテナを起動すると root ユーザーに切り替わる。(図 3-3)

```
sudo docker run --gpus all -it --rm nvcr.io/nvidia/tensorflow:22.09-tf2-py3
```

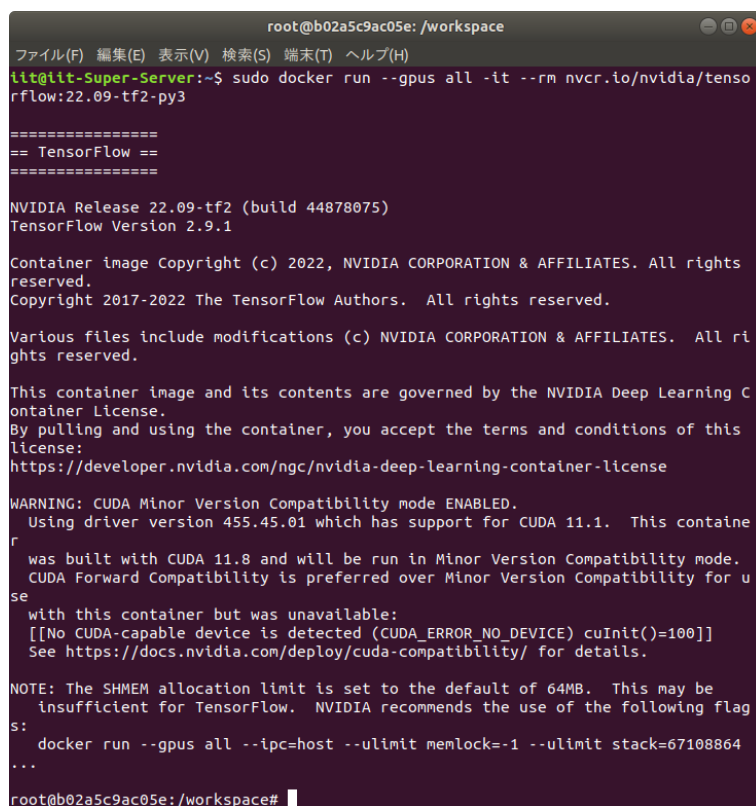
コマンド説明

--gpus all GPU と連携する。

-it ホストの入力をコンテナの標準出力と繋げる

コンテナの標準出力とホストの出力を繋げる。

--rm コンテナ終了時に自動で削除する。



```
root@b02a5c9ac05e: /workspace
ファイル(F) 編集(E) 表示(V) 検索(S) 端末(T) ヘルプ(H)
tit@tit-Super-Server:~$ sudo docker run --gpus all -it --rm nvcr.io/nvidia/tensorflow:22.09-tf2-py3
=====
== TensorFlow ==
=====

NVIDIA Release 22.09-tf2 (build 44878075)
TensorFlow Version 2.9.1

Container image Copyright (c) 2022, NVIDIA CORPORATION & AFFILIATES. All rights reserved.
Copyright 2017-2022 The TensorFlow Authors. All rights reserved.

Various files include modifications (c) NVIDIA CORPORATION & AFFILIATES. All rights reserved.

This container image and its contents are governed by the NVIDIA Deep Learning Container License.
By pulling and using the container, you accept the terms and conditions of this license:
https://developer.nvidia.com/ngc/nvidia-deep-learning-container-license

WARNING: CUDA Minor Version Compatibility mode ENABLED.
Using driver version 455.45.01 which has support for CUDA 11.1. This container was built with CUDA 11.8 and will be run in Minor Version Compatibility mode.
CUDA Forward Compatibility is preferred over Minor Version Compatibility for use with this container but was unavailable:
[[No CUDA-capable device is detected (CUDA_ERROR_NO_DEVICE) cuInit()=100]]
See https://docs.nvidia.com/deploy/cuda-compatibility/ for details.

NOTE: The SHMEM allocation limit is set to the default of 64MB. This may be insufficient for TensorFlow. NVIDIA recommends the use of the following flags:
docker run --gpus all --ipc=host --ulimit memlock=-1 --ulimit stack=67108864
...
root@b02a5c9ac05e: /workspace#
```

図 3-3 コンテナ起動画面

3.3 コンテナにローカルのボリュームをマウントする

以下のようにコマンドを追加すると、ローカルのフォルダーをコンテナのフォルダーとして繋げることができる。^[11]

-v /ホストの任意のパス:/コンテナの任意のパス

コマンド説明

-v ホストのディレクトリを任意の名前でコンテナにマウントすることができる。

実際に実行して、指定したパスでマウントできている様子を図 3-4~図 3-6 に示す。

```
iit@iit-Super-Server:~/Document/AI_basic_ichinoseki_2021$ pwd  
/home/iit/Document/AI_basic_ichinoseki_2021
```

図 3-4 マウントするファイル

```
iit@iit-Super-Server:~$ sudo docker run --gpus all -it --rm -v /home/iit/Document/AI_basic_ichinoseki_2021:/workspace/AI_basic_ichinoseki_2021 tensorflow-plus
```

図 3-5 マウントを実行するコード

```
root@eabbac0d5b35:/workspace/AI_basic_ichinoseki_2021# pwd  
/workspace/AI_basic_ichinoseki_2021
```

図 3-6 コンテナ内にマウントしたファイル

3.4 X Window System の導入

最初の環境では、コンテナ内からウィンドウを表示することができていなかった。そこで X Window System を用いることでウィンドウの表示を可能にした。

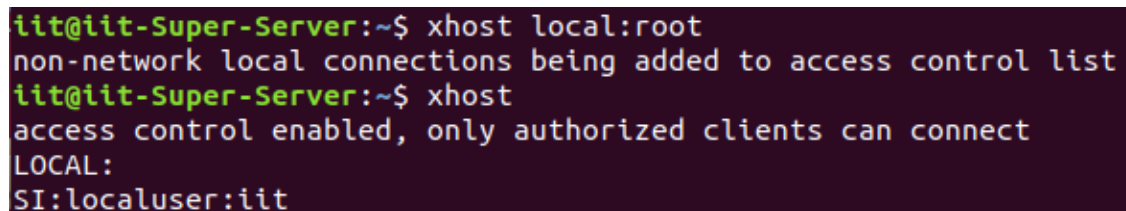
X Window System とは UNIX 系 OS で用いられるウィンドウシステムである。ソフトウェアを実行するコンピュータと入出力するコンピュータを分離することができる。本環境ではデスクトップをホスト、ソフトウェアを実行する側をコンテナ内部として繋げている。^[12]

まずは、どのホストを認証するか許可をする設定をホストで行う必要がある。

そこで以下のコマンドをホストユーザーで実行する。

```
xhost +local:root
```

xhost コマンドで確認すると図のように接続されているのが確認できる。(図 3-7)



```
iit@iit-Super-Server:~$ xhost local:root
non-network local connections being added to access control list
iit@iit-Super-Server:~$ xhost
access control enabled, only authorized clients can connect
LOCAL:
SI:localuser:iit
```

図 3-7 X Window System 接続確認

その後以下のコマンドを追加することで X Window System を利用することができる。

```
-e DISPLAY=$DISPLAY -v /tmp/.X11-unix/:/tmp/.X11-unix mino/debian-xterm
```

コマンド説明

-e 実行中のコンテナで環境変数を指定できる。今回はホストの DISPLAY 変数をコンテナの DISPLAY 変数に入れている。

これは X Window System を用いてウィンドウを表示している様子である。(図 3-8)

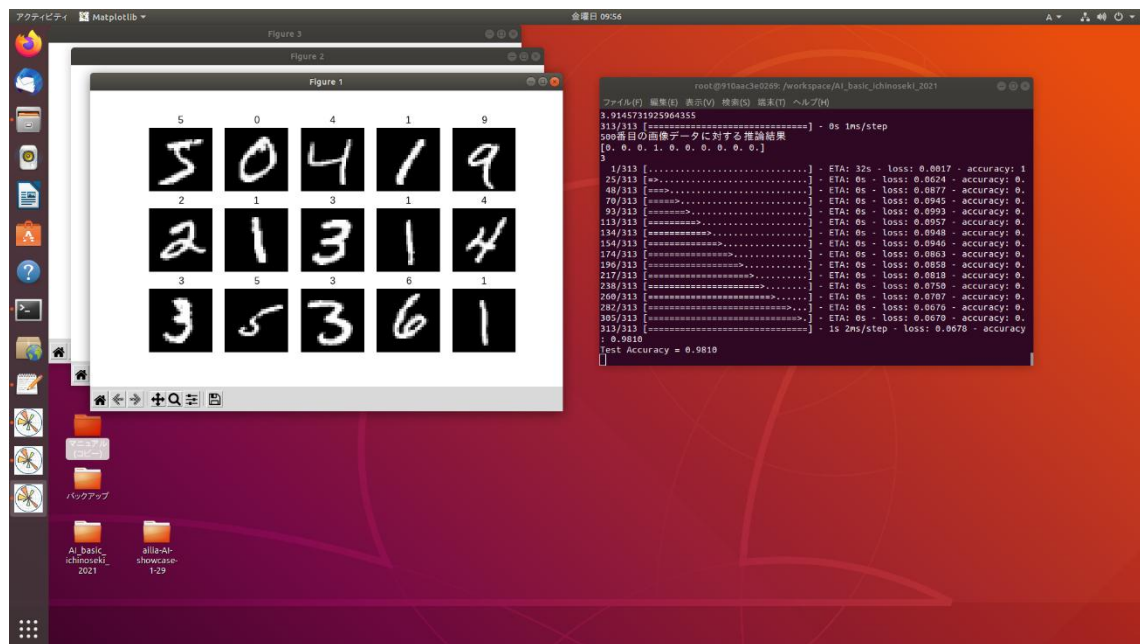


図 3-8 window 表示画面

X Window System の導入方法は以上だが、python のプログラム上でウィンドウ表示を行う記述をしなければ表示されることはないので注意が必要である。(plt.show()を入れる)

3.5 カメラ接続

接続した USB カメラをコンテナ内で使用するためには、以下のコマンドを追加する^[13]。

今回は接続したカメラが video0 に接続されているので video0 を参照する。

```
--device=/dev/video0:/dev/video0
```

コマンド説明

--device デバイスに許可を与える

今回はデモアプリでローカルのカメラを使用したいと思い、接続を行った。(図 3-9)

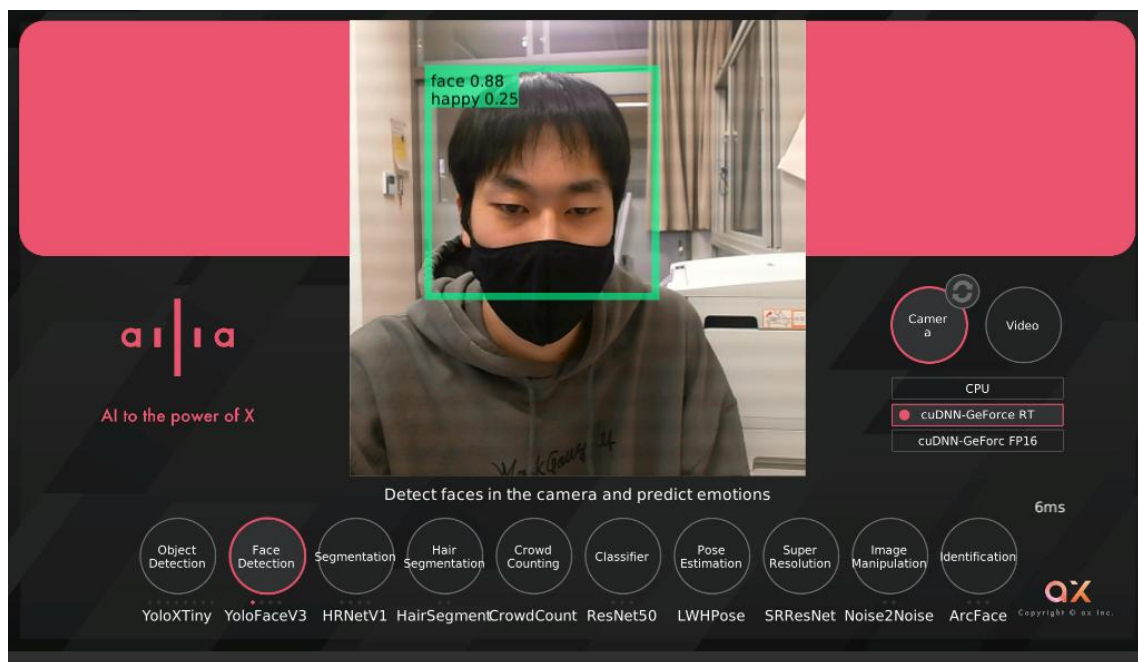


図 3-9 デモアプリによるカメラ利用

3.6 共有メモリ量設定

今回コンテナを起動する際に以下のエラーが出た。

RuntimeError: DataLoader worker (pid 1067) is killed by signal: Bus error. It is possible that dataloader's workers are out of shared memory. Please try to raise your shared memory limit.

df -h コマンドで確認すると shm (POSIX 共有メモリ) が 64M しか割当てられていなく、コンテナを起動できない。

そこで、以下のコマンドを追加して shm の容量を指定した。

```
--shm-size=2G
```

コマンドを実行したことで、shm の容量が 2G となり、コンテナを起動することができた。(図 3-10)

```
root@ad6dbbcbdbc6b:/workspace# df -h
Filesystem      Size  Used Avail Use% Mounted on
overlay          916G   61G  809G   7% /
tmpfs            64M    0    64M   0% /dev
tmpfs           16G    0   16G   0% /sys/fs/cgroup
shm             2.0G    0   2.0G   0% /dev/shm
/dev/sda2        916G   61G  809G   7% /tmp/.X11-unix
tmpfs           16G    0   16G   0% /proc/asound
tmpfs           16G    0   16G   0% /proc/acpi
tmpfs           16G    0   16G   0% /proc/scsi
tmpfs           16G    0   16G   0% /sys/firmware
root@ad6dbbcbdbc6b:/workspace#
```

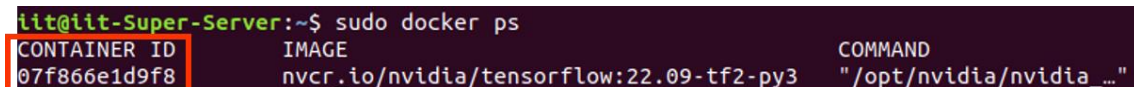
図 3-10 shm の容量

3.7 コンテナコミット方法

コンテナに変更を加えたものから、新たに Docker イメージを作成することができる。これにより変更を加えたコンテナ環境を保存している。^[14]

以下のコマンドで起動しているコンテナ一覧を表示し、CONTAINER ID を確認する。(図 3-11)

```
sudo docker ps
```



```
iit@iit-Super-Server:~$ sudo docker ps
```

CONTAINER ID	IMAGE	COMMAND
07f866e1d9f8	nvcrl.io/nvidia/tensorflow:22.09-tf2-py3	"/opt/nvidia/nvidia_..."

図 3-11 CONTAINER ID 確認

そして、下記のコマンドでコンテナをコミットする。

```
sudo docker commit CONTAINERID コミット後のコンテナ名
```

コミットしたコンテナを使いたい場合は実行コマンドの最後にコンテナの名前を入れることで実行できる。今回は TensorFlow の環境を tensorflow-plus という名前でコミットして利用した。また, yolo v5 のパッケージをインストールした, コンテナを tensorflow-tiikawa という名前でコミットして利用した。

コード実行例

```
iit@iit-Super-Server:~$ sudo docker commit 421d292c222b tensorflow-plus
```

第4章 処理時間計測

4.1 計測用サンプルデータ

処理時間計測では2種類のサンプルデータを用いた。

4.1.1 手書き数字の画像認識

データセットとして「MNIST」を使用した。「MNIST」とは「0」～「9」の手書き数字の画像データセットである。(図4-1)

主に画像認識を目的としたディープラーニング/機械学習の初心者向けチュートリアルでよく使われる。データがきれいに整形されているので、高い精度が出やすいといわれている。

データセット全体で6万枚の学習用データと1万枚のテストデータがある。各画像にはラベル(正解を示す教師データ)がついている。^[15]

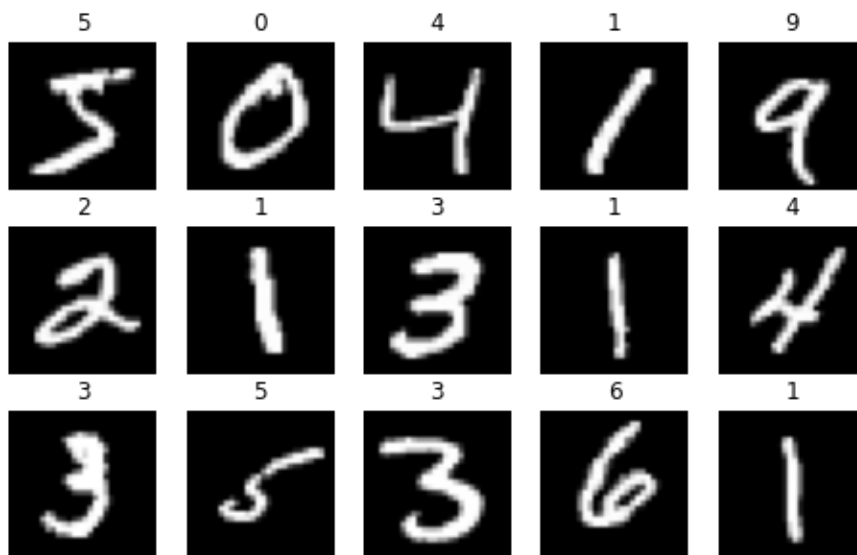


図 4-1 MNIST のデータセット

4.1.2 物体検知用キャラクター学習

他の卒業研究^[16]で yolo5 を用いたキャラクターの物体検知を行っていた。そのデータセットを使わせていただき処理時間の計測を行った。画像から学習させたいキャラクターの座標を、xml ファイルで記述し学習させる。学習用データとして 29 枚の画像が用意されている。(図 4-2)

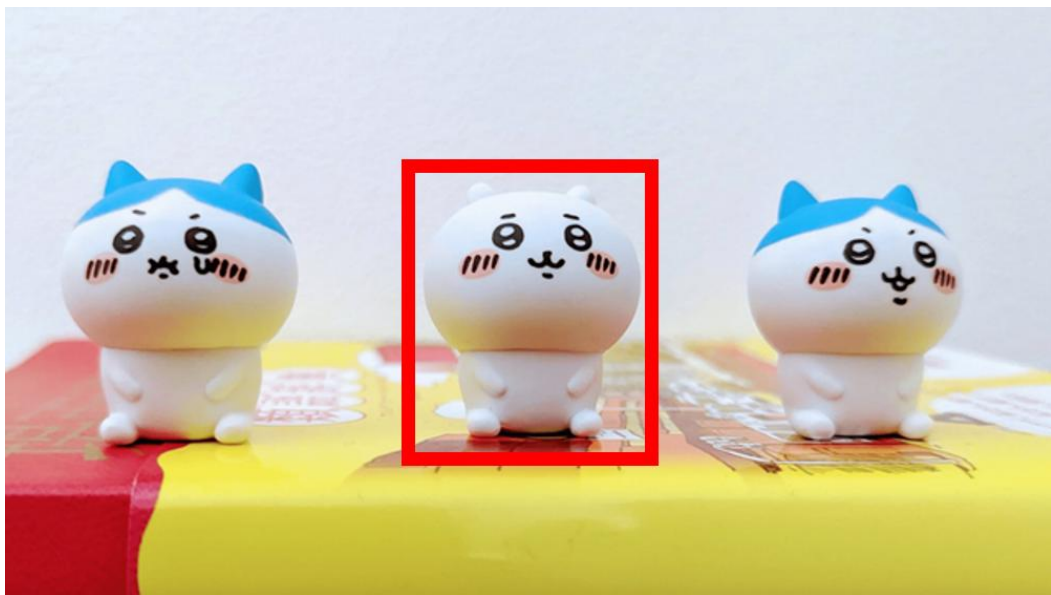


図 4-2 キャラクター学習用データ

4.2 計測方法・計測場所

計測するのはサンプルが学習を行う部分である。その際、学習開始位置と学習終了位置に変数として `start` と `end` を作成し、それぞれ `time.perf_counter` 関数を代入することで、現在の時間を取得する。

そして、開始位置 `start` と終了位置 `end` の差を処理時間とする。

終了時間 (`end`) - 開始時間 (`start`) = 処理時間

また、GPU を利用する際には `torch.cuda.synchronize` を `time.perf_counter` 関数の前に入れて処理を同期させる。

使用関数説明

`time.perf_counter()`

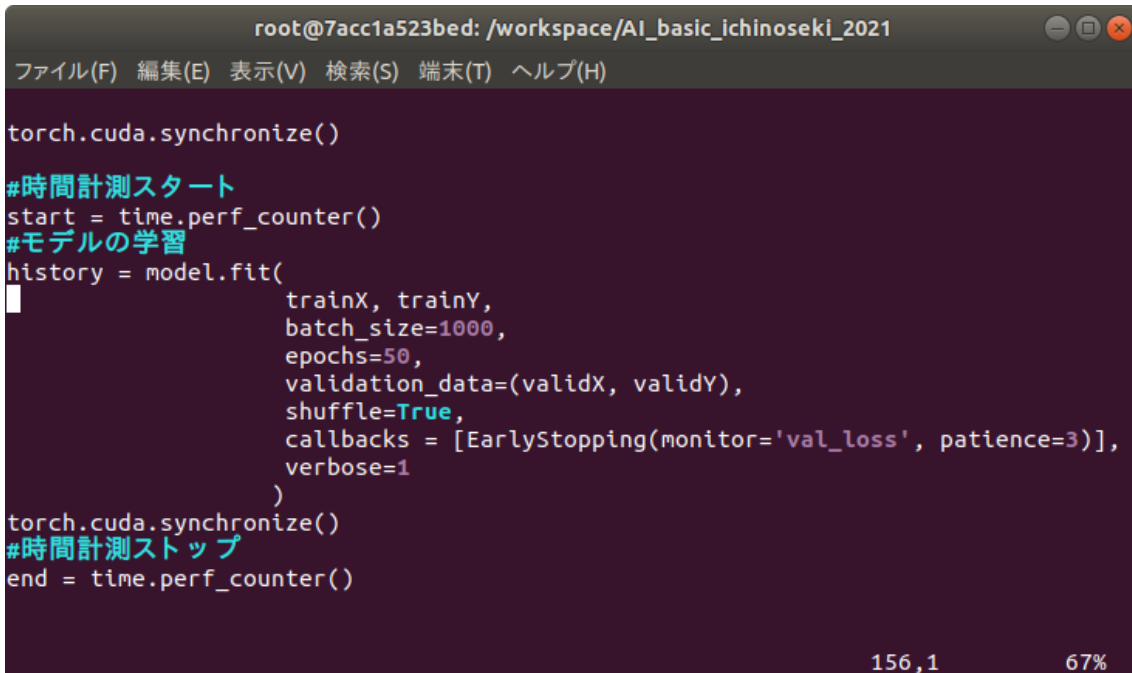
パフォーマンスカウンターの値（小数点以下がミリ秒）を返す。クロックは短期間の計測が行えるよう、可能な限り高い分解能を持つ（今回は小数点第 16 位まで表示された）。

`torch.cuda.synchronize()`

CUDA デバイス上のすべてのストリームのすべてのカーネルが完了するまで待機する。つまり、CPU と GPU を同期することで、処理終了時間を一致させることができる。CUDA を利用している GPU でしか利用することができない。

4.2.1 手書き数字の画像分類

計測する位置は `model.fit()` の中である。(図 4-3)



```
root@7acc1a523bed: /workspace/AI_basic_ichinoseki_2021
ファイル(F) 編集(E) 表示(V) 検索(S) 端末(T) ヘルプ(H)

torch.cuda.synchronize()
#時間計測スタート
start = time.perf_counter()
#モデルの学習
history = model.fit(
    trainX, trainY,
    batch_size=1000,
    epochs=50,
    validation_data=(validX, validY),
    shuffle=True,
    callbacks = [EarlyStopping(monitor='val_loss', patience=3)],
    verbose=1
)
torch.cuda.synchronize()
#時間計測ストップ
end = time.perf_counter()

156,1 67%
```

図 4-3 手書き数字の学習計測位置

コードの解説 (`model.fit`)

`trainX` と `trainY` は学習用データを保持する変数である。

`batch_size` は学習用データをわけた際の一つのサイズを指定している。

`epochs` はデータ全体を何回学習させるかを指定している。

`validation_data` は検証用データを保持する変数である。

`shuffle` は読み込ませるデータをランダムにすることができる。

`EarlyStopping` はある程度学習の精度が良くなると、自動で学習を終了する。

`val_loss` が 0 に近づけば、学習をもとに正解を導けていることになる。

`verbose` は処理の過程を詳細に表示することができる。

4.2.2 物体検知用キャラクター学習

計測する位置は yolov5 のファイルにある train.py 内の学習部分で行う。(図 4-4, 図 4-5)

変数として processing_time を用意しておくことで, epoch が一回終了するタイミングの時間を記録している。

```
# Start training
torch.cuda.synchronize()
#計測スタート
start=time.perf_counter()

#学習スタート
processing_time = []
processing_time.append(0)
t0 = time.time()
nb = len(train_loader) # number of batches
nw = max(round(hyp['warmup_epochs'] * nb), 100) # number of warmup iterations, max(3 epochs, 100 iterations)
# nw = min(nw, (epochs - start_epoch) / 2 * nb) # limit warmup to < 1/2 of training
last_opt_step = -1
maps = np.zeros(nc) # mAP per class
results = (0, 0, 0, 0, 0, 0, 0) # P, R, mAP@.5, mAP@.5-.95, val_loss(box, obj, cls)
scheduler.last_epoch = start_epoch - 1 # do not move
scaler = torch.cuda.amp.GradScaler(enabled=amp)
stopper, stop = EarlyStopping(patience=opt.patience), False
compute_loss = ComputeLoss(model) # init loss class
callbacks.run('on_train_start')
LOGGER.info(f'Image sizes {imgsz} train, {imgsz} val\n'
            f'Using {train_loader.num_workers * WORLD_SIZE} dataloader workers\n'
            f'Logging results to {colorstr('bold', save_dir)}\n'
            f'Starting training for {epochs} epochs...')
for epoch in range(start_epoch, epochs): # epoch -----
    callbacks.run('on_train_epoch_start')
    model.train()#怪しい::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::~
```

図 4-4 キャラクター学習の学習開始位置

```

# end epoch -----
protime = time.perf_counter()
processing_time.append(protime-start)

torch.cuda.synchronize()
#計測ストップ
end=time.perf_counter()

print("処理時間計測結果")
print(end-start)

print("経過時間")
for l in processing_time:
    print(l)

plt.title("GPU Performance")
plt.xlabel("time(s)")
plt.ylabel("frequency")

count = [0,1,2,3,4,5,6,7,8,9,10]
plt.plot(processing_time,count)

plt.grid()
plt.ylim(0,10)
plt.xlim(0,500)
plt.savefig("GPUグラフ.pdf")
plt.show()

# end training -----

```

図 4-5 キャラクター学習の学習終了位置

4.3 計測実行

Google Colaboratory の GPU を使う場合は、設定を行い GPU と繋げる必要がある。

- ①Google Colaboratory の画面の右上にある接続メニューを開き、「リソースを表示」を選択する。
- ②「ランタイムのタイプを変更」を選択する。(図 4-6)

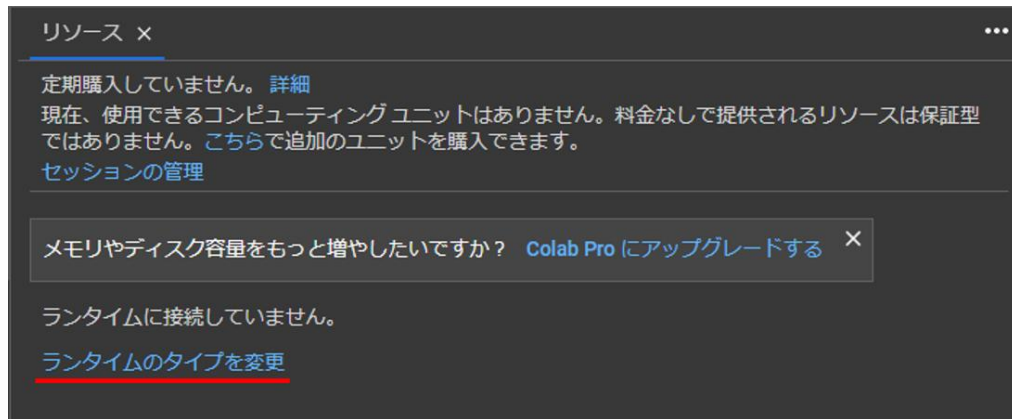


図 4-6 リソース表示画面

- ③「ハードウェアアクセラレータ」から GPU を選択することで設定完了である。(図 4-7)



図 4-7 ランタイムのタイプ変更画面

なお、今回は繋げていないが、ローカルの処理装置と繋げることも可能である。

4.3.1 手書き数字の画像分類

GPU を用いて処理する場合は以下のコードでコンテナを作成する。

```
sudo docker run --gpus all -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -v  
/home/iit/Document/AI_basic_ichinoseki_2021:/workspace/AI_basic_ichinoseki_2021 -it --rm  
tensorflow-plus
```

CPU を用いて処理する場合は以下のコードでコンテナを作成する。

```
sudo docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -v  
/home/iit/Document/AI_basic_ichinoseki_2021:/workspace/AI_basic_ichinoseki_2021 -it --rm  
tensorflow-plus
```

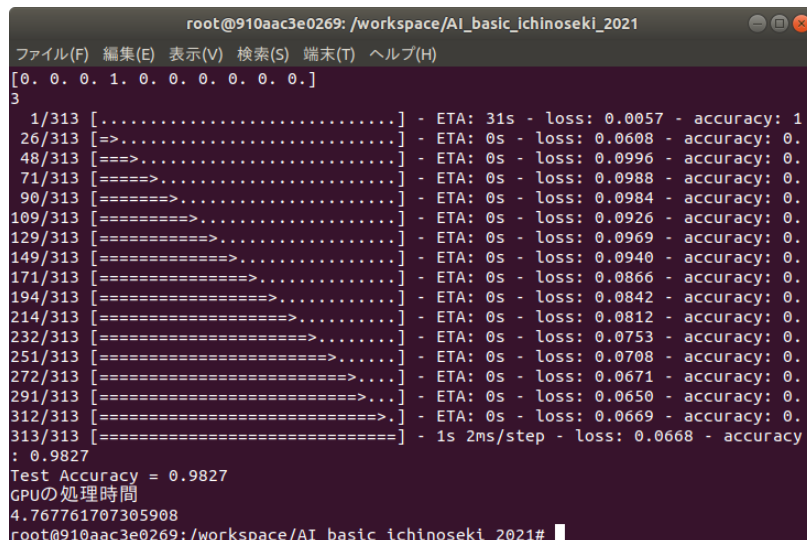
ディレクトリを移動する。

```
cd AI_basic_ichinoseki_2021
```

プログラムを実行する。(図 4-8)

```
python Chapter05.py
```

Google Colaboratory で実行する場合はソースコードを直接書き込み実行する。



```
root@910aac3e0269: /workspace/AI_basic_ichinoseki_2021  
ファイル(F) 編集(E) 表示(V) 検索(S) 端末(T) ヘルプ(H)  
[0. 0. 0. 1. 0. 0. 0. 0. 0.]  
3  
1/313 [.....] - ETA: 31s - loss: 0.0057 - accuracy: 1  
26/313 [=>.....] - ETA: 0s - loss: 0.0608 - accuracy: 0.  
48/313 [====>.....] - ETA: 0s - loss: 0.0996 - accuracy: 0.  
71/313 [=====>.....] - ETA: 0s - loss: 0.0988 - accuracy: 0.  
90/313 [=====>.....] - ETA: 0s - loss: 0.0984 - accuracy: 0.  
109/313 [=====>.....] - ETA: 0s - loss: 0.0926 - accuracy: 0.  
129/313 [=====>.....] - ETA: 0s - loss: 0.0969 - accuracy: 0.  
149/313 [=====>.....] - ETA: 0s - loss: 0.0940 - accuracy: 0.  
171/313 [=====>.....] - ETA: 0s - loss: 0.0866 - accuracy: 0.  
194/313 [=====>.....] - ETA: 0s - loss: 0.0842 - accuracy: 0.  
214/313 [=====>.....] - ETA: 0s - loss: 0.0812 - accuracy: 0.  
232/313 [=====>.....] - ETA: 0s - loss: 0.0753 - accuracy: 0.  
251/313 [=====>.....] - ETA: 0s - loss: 0.0708 - accuracy: 0.  
272/313 [=====>.....] - ETA: 0s - loss: 0.0671 - accuracy: 0.  
291/313 [=====>.....] - ETA: 0s - loss: 0.0650 - accuracy: 0.  
312/313 [=====>.....] - ETA: 0s - loss: 0.0669 - accuracy: 0.  
313/313 [=====>.....] - 1s 2ms/step - loss: 0.0668 - accuracy  
: 0.9827  
Test Accuracy = 0.9827  
GPUの処理時間  
4.767761707305908  
root@910aac3e0269: /workspace/AI_basic_ichinoseki_2021#
```

図 4-8 手書き数字学習実行画面 (専用 PC)

学習データを利用して、テスト用データを推論する。(図 4-9)

結果を見ると正しく推論されていることが確認できる。(図 4-10)

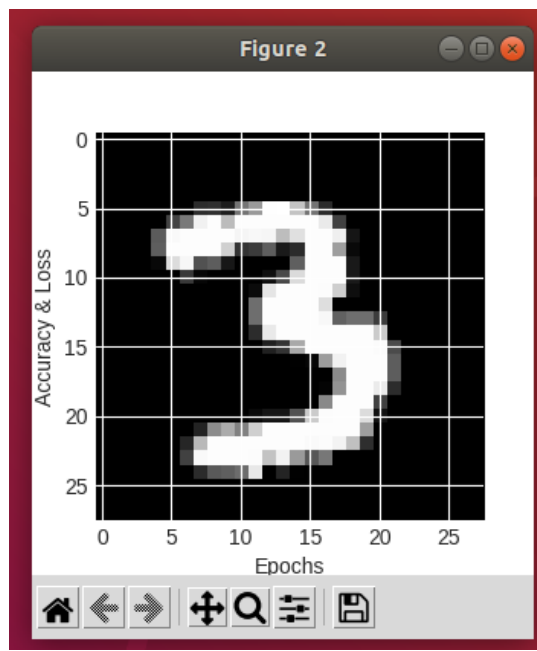


図 4-9 推論対象データ

```
500番目の画像データに対する推論結果  
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]  
3
```

図 4-10 推論結果

4.3.2 物体検知用キャラクター学習

GPU を用いて処理する場合は以下のコードでコンテナを作成する。

```
sudo docker run --gpus all -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -v  
/home/iit/Document/tiikawa:/workspace/tiikawa --shm-size=2G -it --rm tensorflow-tiikawa
```

CPU を用いて処理する場合は以下のコードでコンテナを作成する。

```
sudo docker run -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -v  
/home/iit/Document/tiikawacpu:/workspace/tiikawacpu --shm-size=2G -it --rm tensorflow-tiikawa
```

ディレクトリを移動する。

```
cd tiikawa/yolov5
```

プログラムを実行する。Batch-size と epoch 数は実行する際に指定する。

```
python train.py --batch 20 --epochs 10 --data '/workspace/tiikawa/yolov5/tiikawa.yaml' --name tiikawa  
--name 保存するファイルの名前を指定する。
```

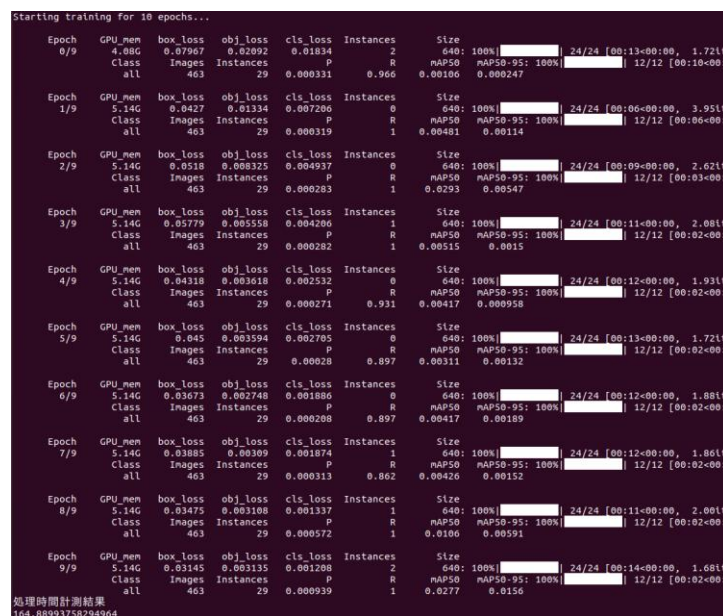


図 4-11 キャラクター学習実行画面

Google Colaboratory を用いるときは、プログラムを実行する前に、データセットが入っているドライブと繋げる必要がある。

以下のコードを実行し、ドライブをマウントする。

```
from google.colab import drive  
drive.mount('/content/drive')
```

以下のコードを実行し、yolov5 のパッケージを一括でインストールする。

```
!pip install -r /content/drive/MyDrive/tiikawa/yolov5/requirements.txt
```

ディレクトリを移行し、プログラムを実行する。

```
%cd /content/drive/MyDrive/tiikawa/yolov5
```

```
!python train.py --batch 20 --epochs 10 --data '/content/drive/My Drive/tiikawa/tiikawa.yaml' --name  
tiikawa
```

学習データを利用して、物体検知を行う。

専用 PC で実行したデータは/workspace/tiikawa/yolov5/runs/train に保存されている。

Google Colaboratory で実行したデータは/マイドライブ/tiikawa/yolov5/runs/train に保存されている。

テスト用データを見るとキャラクターを検知していることが確認できる。

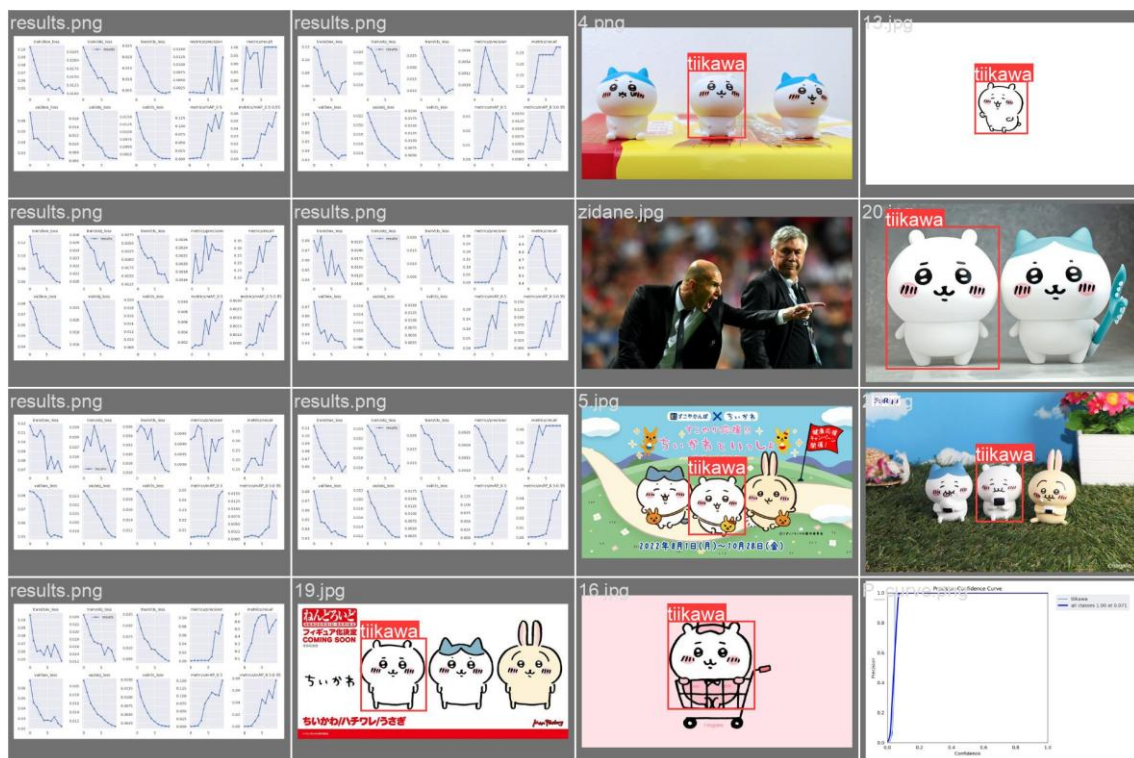


図 4-12 物体検知結果

第5章 結果・考察

5.1 計測結果

5.1.1 手書き数字の画像分類

3つの処理装置でそれぞれ5回ずつ計測し、平均をとった結果を表5-1に示す。

GPU	3.88 秒
CPU	13.16 秒
Google Colaboratory の GPU	4.32 秒

表 5-1 処理装置別手書き数字の学習処理時間

専用 PC の GPU と CPU の処理速度を比較すると、3.4 倍 GPU が速く処理していることが確認できた。

また，学習枚数に応じた処理時間を比較したグラフを以下に示す．（図 5-1）

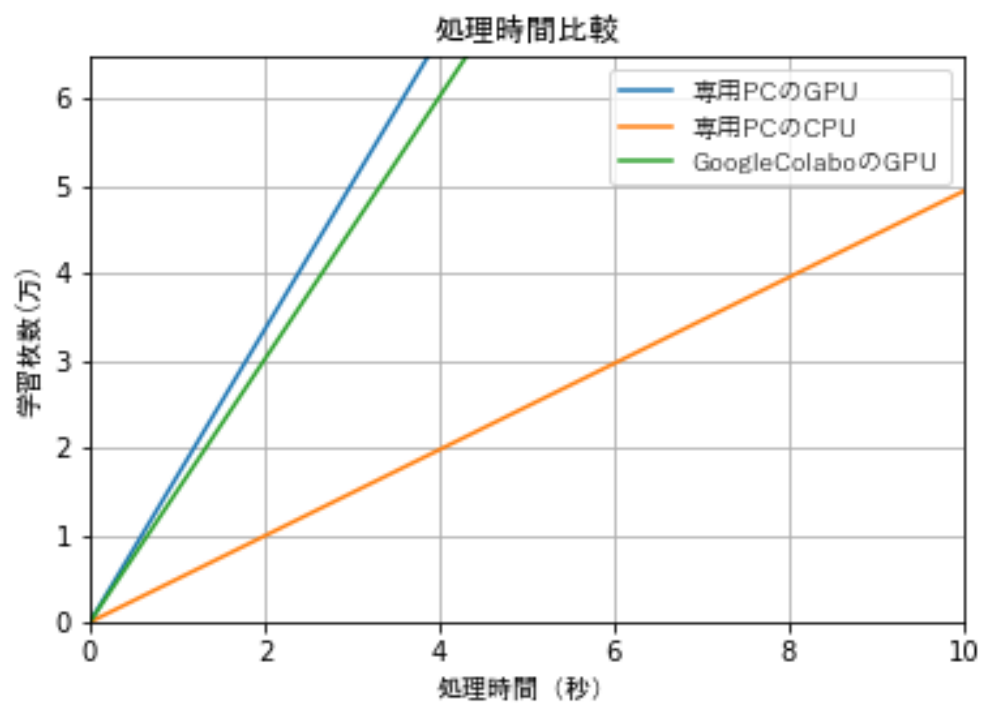


図 5-1 処理時間比較グラフ（手書き）

5.1.2 物体検知用キャラクター学習

3つの処理装置に同様の学習処理を行わせた結果を表5-2に示す。

GPU	157.9 秒
CPU	1222 秒
Google Colaboratory の GPU	330.3 秒

表 5-2 処理装置別キャラクター学習処理時間

専用 PC の GPU と CPU の処理速度を比較すると、7.7 倍 GPU が速く処理していることが確認できた。

また、実行回数に応じた処理時間を比較したグラフを以下に示す。（図 5-2）

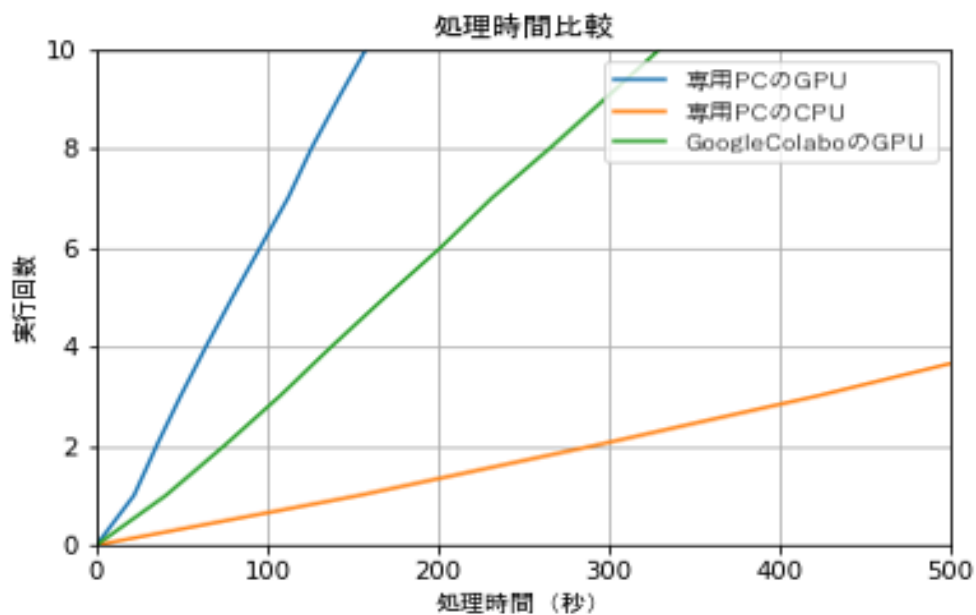


図 5-2 処理時間比較グラフ（キャラクター学習）

5.2 考察

手書き数字の学習時間と、キャラクター学習の時間を見ると、どちらも専用 PC の GPU の処理が一番速いことを確認できた。

また、グラフを見ると手書き数字の学習よりもキャラクター学習の時間のほうが GPU の処理性能に差があった。理由として考えられるのは、手書き数字のデータセットが整いすぎていることだと思われる。学習枚数は手書き数字のほうが多いが、画像のサイズが均一で、ラベルのつけ方が簡単であることから処理に差があまり生まれなかったと推測できる。より高速に Deep Learning を処理するためには、データセットの作り方を工夫する必要があると思われる。

また、複雑な処理であれば GPU の性能の差が顕著に出たので、今後大容量で複雑な処理を行う研究や開発に利用されることを期待している。

第6章 おわりに

今回の研究では、コンテナ環境という未知の環境を扱うことになり大変苦戦した。始めは環境のことをよく理解せずに研究を行っていたが、それにより環境を壊してしまうことがあった。しっかりと環境を理解して使う大切さを学ぶことができた。また、研究を通して Deep Learning の仕組みと GPU の重要性を体験することができた。自分でデータセットを作って学習をさせ、システムを作ることはできなかったが、専用 PC の可能性を十分に感じたので、これから何らかの機能を作り、活用することができればよいと思った。

第7章 参考文献

- [1] 株式会社ピーエスシー 【話題の IT トレンド】 コンテナ技術と仮想マシンの違いとは？

<https://psc-smartwork.com/topics/2021/08/162.html>

- [2] VERITAS コンテナ化とそのメリットについて

<https://www.veritas.com/ja/jp/information-center/containerization>

- [3] Wikipedia Docker

<https://ja.wikipedia.org/wiki/Docker>

- [4] DevOps GPU を利用するための Docker プラグイン「NVIDIA Docker」とは？導入手順は？
(前編)

<https://licensecounter.jp/devops-hub/blog/nvidia-1/>

- [5] TECHCAMP ブログ 【3 分解説】 CPU と GPU の違いをわかりやすく紹介！用途やコア数も

<https://tech-camp.in/note/technology/42935/>

- [6] NVIDIA GEFORCERTX 3090 ファミリ

<https://www.nvidia.com/ja-jp/geforce/graphics-cards/30-series/rtx-3090-3090ti/>

- [7] Wikipedia TensorFlow

<https://ja.wikipedia.org/wiki/TensorFlow>

- [8] python time --- 時刻データへのアクセスと変換

<https://docs.python.org/ja/3.8/library/time.html?highlight=time#module-time>

- [9] OSS×Cloud オープンソースの AI・人工知能／Torch とは

https://www.ossnews.jp/oss_info/Torch

- [10] NVIDIA NGC CATALOG TensorFlow

<https://catalog.ngc.nvidia.com/orgs/nvidia/containers/tensorflow>

[11] Docker-docs-ja docker run ボリュームのマウント

<https://docs.docker.jp/engine/reference/commandline/run.html>

[12] IT 用語辞典 e-Words XWindow System[x11]x.Org

https://e-words.jp/w/X_Window_System.html

[13] ITmedia エンタープライズ 第17回 Dockerで植物が育つ様子を自動録画してみよう——

その1

https://www.itmedia.co.jp/enterprise/articles/1603/02/news031_2.html

[14] Docker-docs-ja docker commit

<https://docs.docker.jp/engine/reference/commandline/commit.html>

[15] @IT MNIST：手書き数字の画像データセット

<https://atmarkit.itmedia.co.jp/ait/articles/2001/22/news012.html>

[16] 遠藤綾李:深層学習を用いたごみ分別支援ツールの作成, 令和4年度卒業研究発表会, 番号 07,

2023