

令和 4 年度 卒業論文

Kubernetes と JupyterHub で構築した

GPU クラスタにおける

深層学習モデル学習時の

学習時間予測

九州工業大学

宇宙システム工学科

191A5017 小野山 翔大

指導教員：花沢 明俊 准教授



# 概要

本論文は、7 章構成である。

第 1 章では、本研究を行うに当たっての背景や目的を紹介する。九州工業大学の AI 学習が抱える問題に注目し、機械学習の観点から解決手法を提案する。また、本論文を読み進めるための前提知識もまとめていく。

第 2 章では、本研究で実施する実験の前準備として、機械学習の実行環境を整える。具体的には、コンテナ管理技術である Kubernetes を複数のサーバーに適用し、動的に計算資源を割り当てられる状態で機械学習環境を構築していく。

第 3 章では、本研究で実施した実験について述べる。第 2 章にて構築した環境の中で機械学習タスクを実行し、実行時間とその他のパラメータを収集する。さらに、得られたデータを使って、実際に学習時間を予測していく。

第 4 章では、第 3 章で述べた実験の結果を解析する。実際に得られたデータや予測結果は CSV 形式であるが、可読性には劣っている。そのため、より可読性の高い形式に解析していく。また、それらのデータを学習データとして学習させた予測精度も共有する。

第 5 章では、得られた結果から読み取れること・考えられることを述べていく。またこれらの結果を踏まえたうえで、今後の洞察や将来性にも言及していく。

第 6 章は結論である。これらの取り組みから得られた結論をまとめていく。

第 7 章では、参考文献を記述していく。本研究を行うに当たって参考した資料をリスト形式で記述していく。

第 8 章では、実際に実験に使用した Python のソースコードを掲載する。実験の具体的な処理については、本章を参考にしてもらいたい。



## 内容

概要 .....	3
1. 序論 .....	1
1.1. 研究背景 .....	1
1.2. 研究目的 .....	1
1.3. 先行研究 .....	2
1.4. 前提知識 .....	2
1.4.1. GPU .....	2
1.4.2. Kubernetes .....	3
1.4.3. JupyterHub .....	4
2. Kubernetes を用いた GPU クラスタの構築 .....	5
2.1. ハードウェアの準備 .....	5
2.2. Swap の無効化 .....	5
2.3. Containerd のインストール .....	8
2.4. CUDA Toolkit のインストール .....	11
2.5. kubeadm, kubectl, kubelet のインストール .....	17

2.6.	Helm のインストール .....	18
2.7.	NFS サーバーの構築.....	19
2.8.	Kubernetes クラスタの立ち上げ.....	21
2.9.	Flannel のインストール .....	23
2.10.	GPU Operator のインストール .....	24
2.11.	永続ストレージ(PV)の設定.....	26
2.12.	JupyterHub のインストール .....	28
3.	学習時間予測実験：方法.....	33
3.1.	学習時間の計測 .....	33
3.2.	学習時間の予測・精度検証・重み計測.....	39
4.	学習時間予測実験：結果.....	41
4.1.	データ収集の結果.....	41
4.2.	予測結果 .....	42
5.	学習時間予測実験：考察.....	44
5.1.	データ収集の結果の考察 .....	44
5.1.1.	バッチサイズ、エポック数、学習時間の関係 .....	44

5.1.2.	バッチサイズ、エポック数、平均メモリ使用率の関係.....	44
5.1.3.	バッチサイズ、エポック数、平均 GPU 使用率の関係.....	45
5.2.	予測結果の考察 .....	45
5.3.	重みの考察.....	46
5.4.	実用上の考察 .....	46
6.	結論.....	47
7.	参考資料.....	48
8.	補足資料.....	50
8.1.	MNIST 手書き画像のクラス分類を行う Python のコード .....	50
8.2.	CIFAR-10 画像のクラス分類を行う Python のコード.....	52
8.3.	深層学習版の CIFAR-10 分類タスクの Python コード .....	55
8.4.	2 層パーセプトロンの Python コード .....	58
8.5.	3 層ニューラルネットワークの Python コード.....	59





# 1. 序論

## 1.1. 研究背景

近年、九州工業大学は機械学習をはじめとした AI 教育に力を入れている。本学のプログラミング学習では、従来は 1,2 年次に C 言語などの組み込み系言語を扱っていた。しかし、ここ数年で状況は大きく変化した。AI の開発を担える人材を育成するというテーマのもと、Python を使った機械学習がカリキュラムに組み込まれている。

機械学習を扱う上で特徴的なのは、開発を通して学習というプロセスを踏むことである。特に深層学習では、人間の脳の神経細胞を模したネットワークモデルをプログラムに記述し、教師データを入力とした学習を行う。学習を終えたモデルは、与えられた教師データから法則を見出す。この法則をもとに、モデルは未知のデータに対して予測を行う。

現在、九州工業大学では、学生に対して個別の機械学習環境を提供できる JupyterHub というプラットフォームを運用している。JupyterHub では、学生は自身のユーザーページにログインできる。各ユーザーページに入れば、機械学習関連のライブラリがあらかじめインストールされた Python, Jupyter 環境を使用できる。

## 1.2. 研究目的

機械学習における学習プロセスは、一般に完了するまで多くの時間を要する。そのため、学習処理を行う際には従来のコンピュータに内蔵されている CPU ではなく、GPU を使用することが多い。GPU はコンピュータの描画処理を高速化するために設計されたデバイスであり、複数のスレッドを並列に実行することができる。そのため、GPU の並列な計算資源を機械学習の行列計算に利用すると、学習時間を大幅に短縮することができる。

九州工業大学で学生が利用できる JupyterHub アカウントにおいても、同様に GPU を利用することができる。しかし、GPU 自体の数は決まっており、学生に一人一人に対して GPU を提供できるとは限らない。一般的には、共通の GPU を複数人で交代しながら使うことになる。ここで問題となるのが、GPU の占有時間である。機械学習においては、学習プロセスに数時間かかることは珍しくない。そのため、GPU が空くのを待っているユーザーにとって、あとどれくらいで GPU が使えるようになるかが重要になってくる。JupyterHub には、そういった GPU 占有時間を予測する機能は実装されていない。そのため現状では、GPU を利用できるかどうかは、逐一確認しないと把握できない。

この問題を解決する手法として、本論文では機会学習を用いた機械学習実行時間の予測

を提案する。実験環境では、動的に GPU の割り当てを行う Kubernetes 環境を構築し、機械学習タスクを実行する。この学習の際に計測されたエポック数やバッチサイズから、学習プロセスの実行時間を予測する。

### 1.3. 先行研究

ここで、現在までのプロセス実行時間の予測について、先行研究にいくつか言及していく。丹野らは、CPU の実行時間を予測した[6]。この際、予測に使用するプロセス情報の組み合わせをいくつか用意し、どの組み合わせが高い信頼度となるのか調べた。信頼度の高い組み合わせを用いることで、精度の高い予測が可能となることが示されている。また、菅谷[7]らはこれらの研究を分散処理に応用した。予測対象プロセスが指定されると、実行履歴から予測対象プロセスに似ているプロセスを複数選択する。それらの実行時間の統計量が予測値となる。堀井らは、ある計算機環境での予測のために生成したデータを、異なる計算機環境での予測にも利用することが可能な MPI プログラム実行時間予測手法を提案している。川口らは、アセンブリコードに対してメモリアクセスを解析し、GPU スケジューリングにかかる時間を短縮した[9]。さらに浜野らは、GPU を搭載したヘテロ型クラスタシステムの運用に焦点を当て、システムのモデリング手法とモデルに基づいたタスクスケジューリング手法を提案した[10]。

以上の先行研究を発展させ、本研究では Kubernetes クラスタ上にデプロイしたアプリケーションから、予測対象のプロセスを実行する。予測対象プロセスの実行環境をアプリケーションという上位のレイヤに移すことで、実行時間予測の新しい手法を提案する。

### 1.4. 前提知識

本研究を紹介するにあたって、いくつか前提知識が必要となる。これらを以下の項目で説明する。

#### 1.4.1. GPU

GPU (Graphics Processing Unit) は、コンピュータグラフィックの処理に特化して設計されたプロセッサである。通常の CPU (Central Processing Unit) には、自身の内部に数個のコアが搭載されている。例えば、CPU の説明にデュアルコアという記載があれば、内部に 2 つのコアが存在する。また、クアッドコアなら 4 個、オクタコアなら 8 個といった具合である。これに対し、GPU が保持しているコアは数千個にもものぼる。本研究室で使用されている NVIDIA GeForce RTX 3060 は、内部に 3584 個ものコアを備えている。この膨大

な量のコアによって、大量の計算を並列に処理することが可能となる。例えば 3D ゲームでは、GPU を使用することでスムーズな映像を描写できる。3D ゲームなどの映像処理には膨大な計算が必要となるため、GPU の計算能力を使って、繊細な映像を表示したり、滑らかな映像を映し出すことが可能となる。

さらに近年では、汎用的な計算に対して GPU が使われることも多い。これは GPGPU (General-purpose computing on graphics processing units) と呼ばれる。GPU は、上述の通り非常に高い並列計算能力を有しているため、画像処理以外の用途にも応用されている。その一つが機械学習である。機械学習における学習プロセスは、一般的に完了するまで多くの時間を要する。Karras らは、GAN[1]の発展形である PGGAN の学習に 96 時間費やしている[2]。また Style GAN の Tensorflow 実装[4]では、実際の学習に数日から数週間かかることが示されている[3]。こういった学習プロセスに対して GPU を使用することで、学習プロセスにかかる時間を短縮することができる。

#### 1.4.2. Kubernetes

Kubernetes は、コンテナ化されたアプリケーションのデプロイやスケジューリングを自動化するプラットフォーム(コンテナオーケストレーションエンジン)である。コンテナとは、ホスト OS から隔離された環境でアプリケーションやミドルウェアを実行できる技術である。従来では、アプリケーションの実行環境を OS から隔離するには、OS ごと仮想化ソフトウェアで仮想化することが多かった。一方、コンテナ技術では OS ではなくプロセスを隔離する。そのため、仮想 OS と比べて非常に軽量であり、構築も簡単に行うことができる。こういった利点があって、コンテナ技術は現代のアプリケーション開発に欠かせない技術となっている。

このように、コンテナ技術には隔離環境でアプリケーションを実行する機能を有する一方で、コンテナ自体を管理したり、複数のコンテナと連携したりする機能はない。そこで、複数のコンテナを管理するために Kubernetes が用いられる。Kubernetes を使えば、複数のコンテナのネットワークやデータを管理したり、コンテナが何らかの理由で停止した際にコンテナを自動で再起動したりできる。さらに、新しいコンテナを追加した時に、他のコンテナの実行状況を判断して自動で適切なリソースを割り当てるスケジューリングの機能も持ち合わせている。このように、Kubernetes は複数のコンテナを「あるべき姿」に維持して管理することができる。

この Kubernetes の自動管理機能を利用して、近年では GPU を搭載した機械学習基盤を Kubernetes 上に構築する動きがみられる。先に説明したように、Kubernetes には充実した

スケジューリング機能が搭載されている。使用されていない計算資源を検知して、優先的にプログラムタスクを割り当てることができる。また Kubernetes は拡張性にも優れており、GPU を簡単に追加できるほか、ストレージを増築することも容易である。こうすることで、ユーザーの増加などに伴い、機械学習基盤も簡単にスケールアップできる。以上の例にならい、本研究では GPU サーバーを計算資源とした Kubernetes クラスタを構築し、このクラスタ上で機械学習を実行する。

#### 1.4.3. JupyterHub

JupyterHub は、ユーザーに JupyterLab 環境を提供するプラットフォームである。ユーザーは、JupyterHub のログイン画面にブラウザから入る。そして、ユーザーネームとパスワードを入力する。適切なユーザーネーム・パスワードが入力されると、ログイン処理が実行され、JupyterLab 環境が使えるようになる。一般的な JupyterHub の処理では、1つのユーザーにつき1つの Docker コンテナが起動する。JupyterLab で作成したファイルやデータは Docker コンテナに保存され、Python や Tensorflow などの処理もコンテナ内で行われる。このように、JupyterHub はバックエンドでコンテナを実行しているため、この仕組みを Kubernetes クラスタ上で処理することもできる。ユーザーコンテナは、Kubernetes 上では Pod として処理される。データやファイルなども、ユーザーごとの Pod に格納される。このように、JupyterHub を使えば、複数のユーザーが自分の JupyterLab 環境を利用できるようになる。

## 2. Kubernetes を用いた GPU クラスタの構築

本研究では、機械学習タスクを実行した時の実行時間を、機械学習を用いて予測する。したがって、機械学習タスクの実行時間を学習データとして収集する必要がある。今回は、GPU を 2 台搭載した Kubernetes クラスタを構築する。本クラスタ上で機械学習タスクを実行し、GPU の割り当てから機械学習タスク終了までに要した時間を、学習データとして収集する。その後、機械学習モデルに学習データを入力して学習を回し、実行時間を予測する。

本クラスタを構築する際には、九州工業大学が保有する物理サーバー及び物理 GPU を採用した。本学は、大学の AI 講義の一環として、希望する学生に対して機械学習を実行できる Jupyter 環境を作成・配布している。これは本学の情報基盤センターで物理サーバーとして管理されているものである。この Jupyter 環境は、一般には GPU をはじめとする高度な計算資源を有したものが望ましい。将来本学サーバーに GPU クラスタを導入することを想定し、本研究では物理サーバーと物理 GPU を用いる。

### 2.1. ハードウェアの準備

Kubernetes GPU クラスタを構築するにあたっては、複数台の PC が必要となる。具体的には、計算を行うノードを 1 台、それらを制御するコントロールプレーンノードを 1 台、そして、Kubernetes 上でのデータを動的に保存する NFS(Network File System)サーバーを 1 台用意する。今回は、クラスタを構築するマシンとして以下の OS を用意する。

- コントロールプレーンノード：Ubuntu 20.04.5 LTS (Focal Fossa)
- ワーカーノード：Ubuntu 20.04.5 LTS (Focal Fossa)
- NFS サーバー：Ubuntu 20.04.5 LTS (Focal Fossa)

また、各ワーカーノードに接続する GPU も用意する。本研究で用いる GPU の仕様は以下の通りである。

- GPU：NVIDIA GeForce RTX 2060

### 2.2. Swap の無効化

Kubernetes クラスタを構築する上で必要になるのが、OS の Swap 機能の無効化である。Swap 機能は、メモリの管理方式の一つである。具体的には、メモリで処理しているデータ

が容量を超えそうになったとき、優先度の低いデータをメモリからストレージに退避する機能を指す。この Swap 機能は、Kubernetes クラスタ構築では無効にする必要がある。

スワップの無効化は、コントロールプレーンノード及びワーカーノードの両方で実施する。実際に使用されている Swap 領域は、以下のコマンドで確認できる。

```
$ free -h
```

	total	used	free	shared	buff/cache	available
Mem:	62Gi	2.8Gi	56Gi	837Mi	3.2Gi	58Gi
Swap:	2.0Gi	0B	2.0Gi			

この出力から、Swap 領域が 2GiB 確保されていることが分かる。ここで、`/etc/fstab` を開き、Swap 領域に関する設定項目をコメントアウトする。

```
$ cat /etc/fstab
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point> <type> <options>          <dump> <pass>
# / was on /dev/nvme0n1p2 during installation
UUID=f597444d-b1ce-448c-9c89-6d89412141b4 /          ext4    errors=remount-ro 0      1
# /boot/efi was on /dev/nvme0n1p1 during installation
UUID=AF2E-4A9C /boot/efi    vfat    umask=0077        0      1
# この部分をコメントアウトする
# /swapfile                                none    swap    sw                0      0
```

次に、`systemctl --type swap` コマンドを実行し、現在のスワップ情報を確認する。

```
$ systemctl --type swap
```

UNIT	LOAD	ACTIVE	SUB	DESCRIPTION
swapfile.swap	loaded	active	active	/swapfile

LOAD = Reflects whether the unit definition was properly loaded.

ACTIVE = The high-level unit activation state, i.e. generalization of SUB.

SUB = The low-level unit activation state, values depend on unit type.

1 loaded units listed. Pass --all to see loaded but inactive units, too.

To show all installed unit files use 'systemctl list-unit-files'.

現時点では、`swapfile.swap` の部分にはなにも処理がなされていない。この `swapfile.swap` の部分を無効化していく

まず、Swap 領域のパーティションを確認する。

```
$ swapon --show
```

NAME	TYPE	SIZE	USED	PRIO
/swapfile	file	2G	0B	-2

今回の場合、`/swapfile` に Swap 領域のパーティションが割り当てられている。この `/swapfile` が OS に読み込まれないように設定する。具体的には、以下のように `/swapfile` のスワップをマスクする。

```
$ sudo systemctl mask "swapfile.swap"
```

Created symlink /etc/systemd/system/swapfile.swap → /dev/null.

これが完了したら、再度 Swap 用パーティションを確認する。

```
$ systemctl --type swap
```

UNIT	LOAD	ACTIVE	SUB	DESCRIPTION
● swapfile.swap	masked	active	active	/swapfile

LOAD = Reflects whether the unit definition was properly loaded.

ACTIVE = The high-level unit activation state, i.e. generalization of SUB.

SUB = The low-level unit activation state, values depend on unit type.

1 loaded units listed. Pass --all to see loaded but inactive units, too.

To show all installed unit files use 'systemctl list-unit-files'.

このように、`/swapfile` の部分に●がマークされ、マスクされていることが確認できる。

以上の設定が終了したら、ノードを再起動する。そして、再度 Swap 領域を確認する。

```
$ free -h
```

	total	used	free	shared	buff/cache	available
Mem:	62Gi	1.2Gi	60Gi	228Mi	1.1Gi	60Gi
Swap:	0B	0B	0B			

この出力から、Swap 領域の容量が 0 になっていることが分かる。これで Swap が無効化され、Kubernetes クラスタを構築することが可能となる。

## 2.3. Containerd のインストール

Kubernetes のコンテナランタイムインターフェース(CRI)には、さまざまな種類が存在する。今回は、CRI として Containerd を使用する。Containerd は、現在のコンテナ技術の主流となっている Docker の内部で使用されている技術である。もともと Docker の一部分であったが、Docker から切り離されてオープンソースプロジェクトとして公開されている。

Containerd のインストールはコントロールプレーンノード・ワーカーノード両方で実施する。まず、Containerd 自体のネットワークを設定する。具体的には、Containerd の設定ファイルである `containerd.conf` にネットワーク設定を記述し、システムに適応させていく。

```
$ cat > /etc/modules-load.d/containerd.conf <<EOF
overlay
br_netfilter
EOF
```

```
$ modprobe overlay
$ modprobe br_netfilter
```

また、必要なカーネルパラメータを設定する。これも同様に、設定ファイルにカーネルのパラメータを記述し、システムに適用させていく。

```
$ cat > /etc/sysctl.d/99-kubernetes-cri.conf <<EOF
net.bridge.bridge-nf-call-iptables = 1
```



```
net.ipv4.ip_forward          = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
```

```
$ sysctl --system
* Applying /etc/sysctl.d/10-console-messages.conf ...
kernel.printk = 4 4 1 7
* Applying /etc/sysctl.d/10-ipv6-privacy.conf ...
...
fs.protected_symlinks = 1
* Applying /etc/sysctl.conf ...
```

これらの設定が完了したら、両ノードに Containerd をインストールする。

```
$ apt-get update && apt-get install -y apt-transport-https ca-certificates curl
software-properties-common
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
...
apt-transport-https is already the newest version (2.0.9).
0 upgraded, 0 newly installed, 0 to remove and 7 not upgraded.
```

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
OK
```

```
$ add-apt-repository ¥
"deb [arch=amd64] https://download.docker.com/linux/ubuntu ¥
$(lsb_release -cs) ¥
stable"
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
...
Hit:11 http://jp.archive.ubuntu.com/ubuntu focal-backports InRelease
Reading package lists... Done
```

```
$ apt-get update && apt-get install -y containerd.io
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
...
Processing triggers for man-db (2.9.1-1) ...
```

```
$ mkdir -p /etc/containerd
```

```
$ containerd config default | sudo tee /etc/containerd/config.toml
disabled_plugins = []
imports = []
...
gid = 0
uid = 0
```

```
$ systemctl restart containerd
```

また、Kubernetes クラスタにのコンテナランタイムとして Containerd を使用する際は、`/etc/containerd/config.toml` 内の `SystemdCgroup` という項目を `True` に設定する。この設定があれば、Kubernetes クラスタを正常に立ち上げられるようになる。

```
$cat /etc/containerd/config.toml
[plugins]

[plugins."io.containerd.gc.v1.scheduler"]
  deletion_threshold = 0
  ...

[plugins."io.containerd.grpc.v1.cri"]
  device_ownership_from_security_context = false
  ...
```

```

[plugins."io.containerd.grpc.v1.cri".cni]
    bin_dir = "/opt/cni/bin"
    ...

[plugins."io.containerd.grpc.v1.cri".containerd]
    default_runtime_name = "runc"
    ...

[plugins."io.containerd.grpc.v1.cri".containerd.default_runtime]
    base_runtime_spec = ""
    ...

[plugins."io.containerd.grpc.v1.cri".containerd.default_runtime.options]

[plugins."io.containerd.grpc.v1.cri".containerd.runtimes]

[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
    base_runtime_spec = ""
    ...

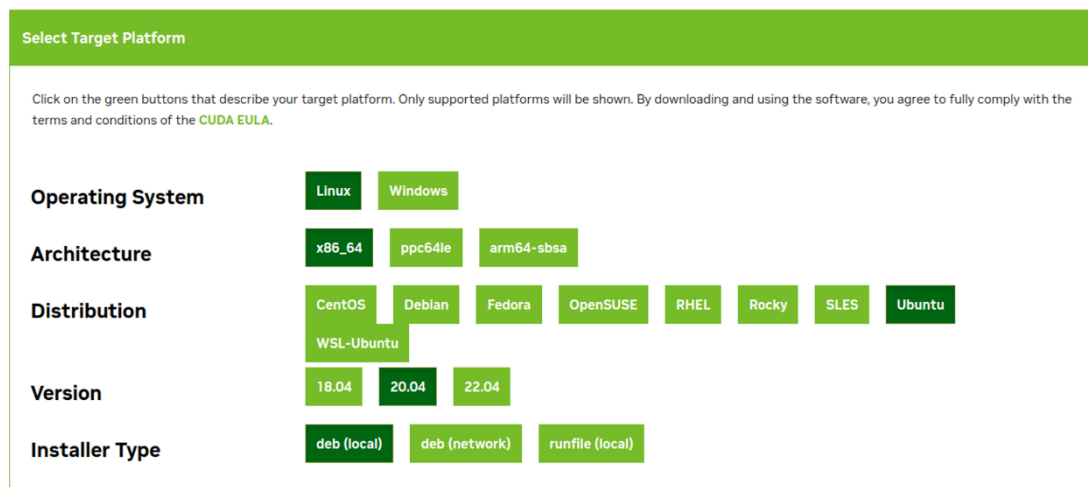
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
    BinaryName = ""
    ...
    SystemdCgroup = true

```

## 2.4. CUDA Toolkit のインストール

次に、コントロールプレーンノード・ワーカーノード両方に、CUDA Toolkit をインストールする。CUDA Toolkit は、CUDA C アプリケーションを GPU 向けにコンパイルするコンパイラである。Ubuntu システムから GPU の計算資源を利用する際に、必ず必要となるものである。

CUDA Toolkit をインストールする際には、画像の CUDA Toolkit のインストールページを参照する。具体的には、インストールページは画像のようなページになる。



Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown. By downloading and using the software, you agree to fully comply with the terms and conditions of the [CUDA EULA](#).

**Operating System**

Linux Windows

**Architecture**

x86\_64 ppc64le arm64-sbsa

**Distribution**

CentOS Debian Fedora OpenSUSE RHEL Rocky SLES Ubuntu

WSL-Ubuntu

**Version**

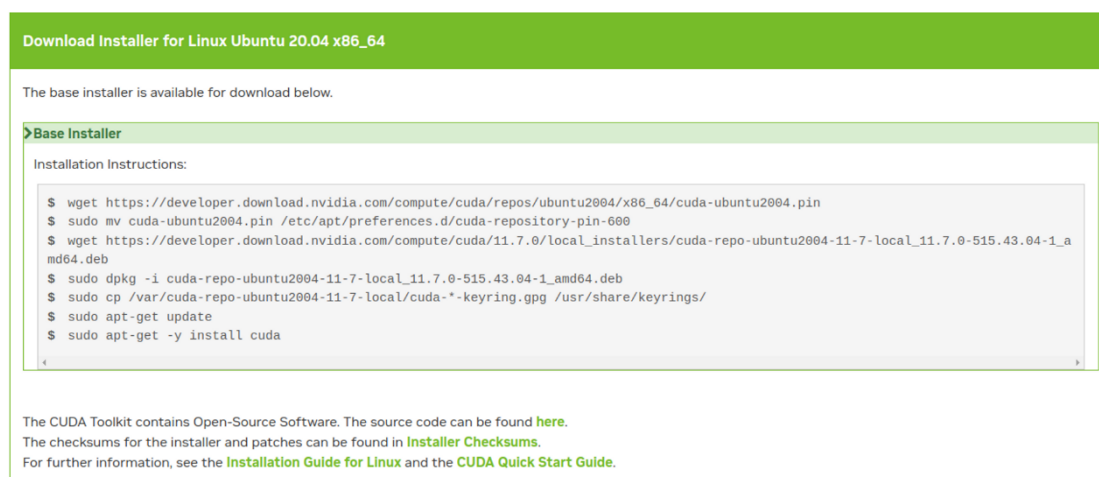
18.04 20.04 22.04

**Installer Type**

deb (local) deb (network) runfile (local)

図 1 : CUDA Toolkit のインストールページ ([https://developer.nvidia.com/cuda-11-7-0-download-archive?target\\_os=Linux&target\\_arch=x86\\_64&Distribution=Ubuntu&target\\_version=20.04&target\\_type=deb\\_local](https://developer.nvidia.com/cuda-11-7-0-download-archive?target_os=Linux&target_arch=x86_64&Distribution=Ubuntu&target_version=20.04&target_type=deb_local))

インストールページには、CUDA Toolkit をインストールする環境の情報を入力していく。本研究では、OS は Ubuntu 20.04 を使用しているため、画像のようなオプションを入力した。このページに必要な情報が入力されると、インストールコマンドが表示される。これらを上から順に実行していくと、CUDA Toolkit がインストールされる。



Download Installer for Linux Ubuntu 20.04 x86\_64

The base installer is available for download below.

>Base Installer

Installation Instructions:

```
$ wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/cuda-ubuntu2004.pin
$ sudo mv cuda-ubuntu2004.pin /etc/apt/preferences.d/cuda-repository-pin-600
$ wget https://developer.download.nvidia.com/compute/cuda/11.7.0/local_installers/cuda-repo-ubuntu2004-11-7-local_11.7.0-515.43.04-1_a
md64.deb
$ sudo dpkg -i cuda-repo-ubuntu2004-11-7-local_11.7.0-515.43.04-1_amd64.deb
$ sudo cp /var/cuda-repo-ubuntu2004-11-7-local/cuda-*-keyring.gpg /usr/share/keyrings/
$ sudo apt-get update
$ sudo apt-get -y install cuda
```

The CUDA Toolkit contains Open-Source Software. The source code can be found [here](#).  
The checksums for the installer and patches can be found in [Installer Checksums](#).  
For further information, see the [Installation Guide for Linux](#) and the [CUDA Quick Start Guide](#).

図 2 : CUDA Toolkit のインストールコマンド

CUDA Toolkit をインストールした際のコマンド及びログは、以下の通りである。

```
$ wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/cuda-ubuntu2004.pin
```

```
--2023-02-03 18:55:50--
https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/cuda-ubuntu2004.pin
```

```
...
```

```
Saving to: 'cuda-ubuntu2004.pin'
```

```
cuda-ubuntu2004.pin
```

```
100%[=====
=====>] 190 --KB/s in 0s
```

```
2023-02-03 18:55:55 (1.57 MB/s) - 'cuda-ubuntu2004.pin' saved [190/190]
```

```
$ sudo mv cuda-ubuntu2004.pin /etc/apt/preferences.d/cuda-repository-pin-600
```

```
$ wget https://developer.download.nvidia.com/compute/cuda/11.7.0/local_installers/cuda-repo-ubuntu2004-11-7-local_11.7.0-515.43.04-1_amd64.deb
```

```
--2023-02-03 18:56:28--
```

```
https://developer.download.nvidia.com/compute/cuda/11.7.0/local_installers/cuda-repo-ubuntu2004-11-7-local_11.7.0-515.43.04-1_amd64.deb
```

```
Resolving developer.download.nvidia.com (developer.download.nvidia.com)... 152.199.39.144
```

```
Connecting to developer.download.nvidia.com
```

```
(developer.download.nvidia.com)|152.199.39.144|:443... connected.
```

```
HTTP request sent, awaiting response... 200 OK
```

```
Length: 2699460270 (2.5G) [application/x-deb]
```

```
Saving to: 'cuda-repo-ubuntu2004-11-7-local_11.7.0-515.43.04-1_amd64.deb.1'
```

```
cuda-repo-ubuntu2004-11-7-local_11.7.0-515.43.
```

```
100%[=====
=====>] 2.51G 11.5MB/s in 3m 49s
```

```
2023-02-03 19:00:16 (11.3 MB/s) - 'cuda-repo-ubuntu2004-11-7-local_11.7.0-515.43.04-1_amd64.deb.1' saved [2699460270/2699460270]
```

```
$ sudo dpkg -i cuda-repo-ubuntu2004-11-7-local_11.7.0-515.43.04-1_amd64.deb
Selecting previously unselected package cuda-repo-ubuntu2004-11-7-local.
...
To install the key, run this command:
sudo cp /var/cuda-repo-ubuntu2004-11-7-local/cuda-15CCF53C-keyring.gpg /usr/share/keyrings/

$ sudo cp /var/cuda-repo-ubuntu2004-11-7-local/cuda-*-keyring.gpg /usr/share/keyrings/

$ sudo apt-get update
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
...
Fetched 336 kB in 4s (90.1 kB/s)
Reading package lists... Done

$ sudo apt-get -y install cuda
Reading package lists...
Building dependency tree...
...
Processing triggers for initramfs-tools (0.136ubuntu6.7) ...
update-initramfs: Generating /boot/initrd.img-5.15.0-58-generic
```

なお、`sudo apt-get -y install cuda`を実行中に、以下のような画面が表示されることがある。

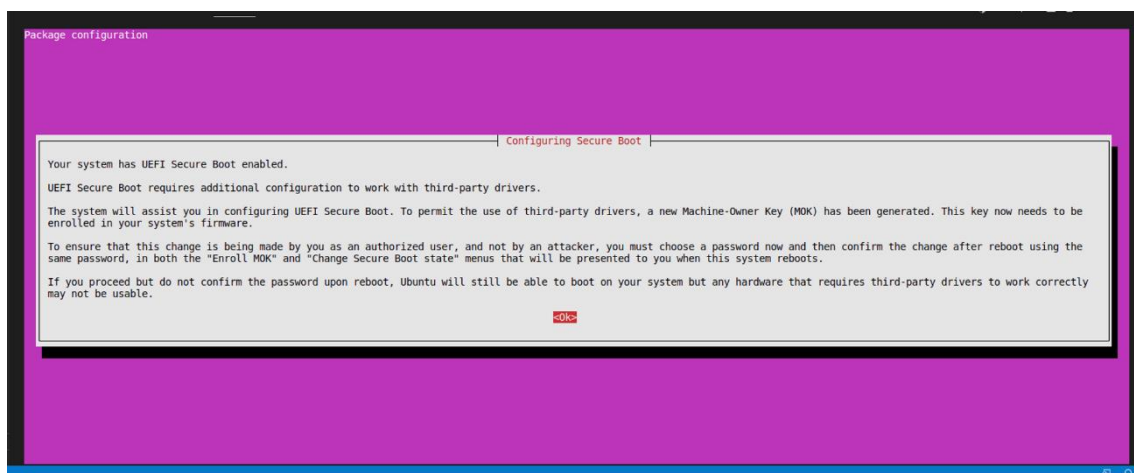


図 3：Secure Boot の確認画面。システムの設定を書き換えるためにパスワードを要求するという案内が書かれている。

この際は、十字キーを操作して<Ok>にカーソルを合わせ、Enter を押す。そうすれば、このような画面が表示に移る。

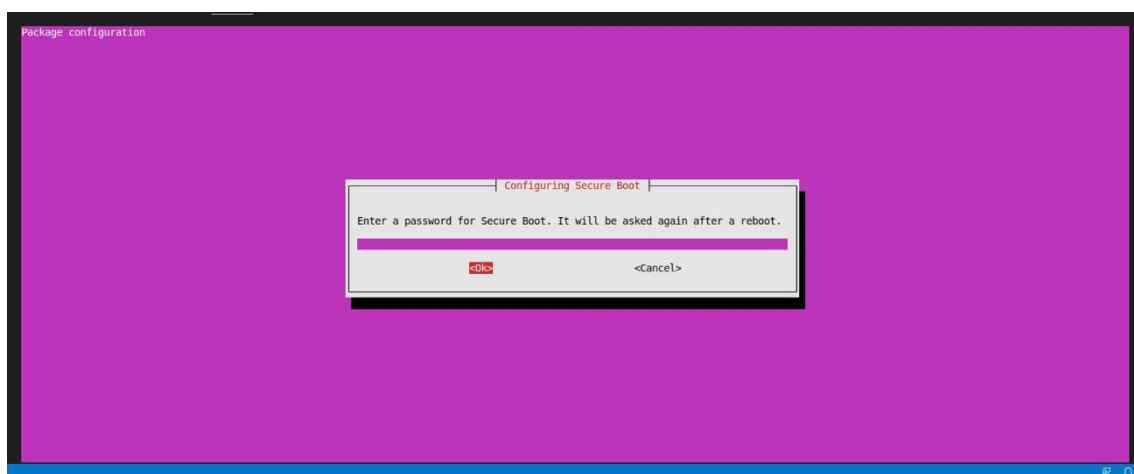


図 4：Secure Boot のためのパスワード入力画面

ここでは、Ubuntu のパスワードを入力して、Enter を押す。再度パスワードを求められたら、再びパスワードを入力して Enter を押す。正しいパスワードが入力されたらインストールが再開される。

CUDA Toolkit のインストールが正常に終了すると、Ubuntu のアプリ一覧画面に、NVIDIA のアプリ群が追加されているのが確認できる。

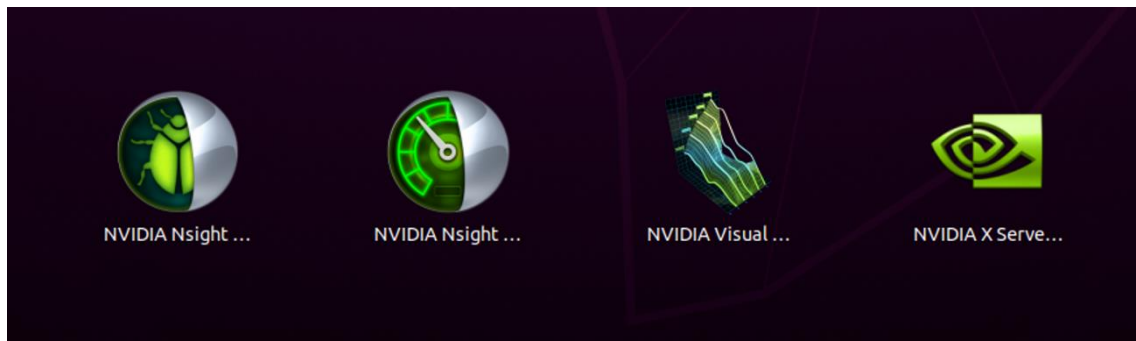


図 5 : CUDA Toolkit をインストールする際に Ubuntu に追加される NVIDIA アプリケーション

これが終わったら、Ubuntu を再起動する。そして、`nvidia-smi` コマンドを入力する。すると、GPU が認識されているのが確認できる。

```
$ nvidia-smi
```

```
Fri Feb  3 19:56:00 2023
```

```
+-----+
| NVIDIA-SMI 515.43.04    Driver Version: 515.43.04    CUDA Version: 11.7    |
|-----+-----+-----+-----+
| GPU   Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|                                           MIG M. |
|=====+=====+=====+=====|
|   0   NVIDIA GeForce ...   On         | 00000000:04:00.0 Off  |                 N/A |
|  0%   26C    P8      6W / 170W |  10MiB / 12288MiB |      0%      Default |
|                                           |                 N/A |
+-----+-----+-----+-----+
```

```
+-----+
| Processes:                                |
| GPU   GI    CI          PID    Type    Process name          GPU Memory |
|          ID    ID                                   Usage      |
|=====+=====+=====+=====|
|   0   N/A   N/A         1115     G   /usr/lib/xorg/Xorg        4MiB |
|   0   N/A   N/A         2207     G   /usr/lib/xorg/Xorg        4MiB |
+-----+-----+-----+-----+
```



## 2.5. kubeadm, kubectl, kubelet のインストール

ここでは、Kubernetes のクラスタを操作する `kubeadm`, `kubectl`, `kubelet` コマンドをインストールする。`kubeadm` は、初期のクラスタの立ち上げの際に、ネットワークなどの設定から各ノードの接続まで、人の手を介さずに自動で行ってくれる。`kubectl` は、コントロールプレーンとの通信をコマンド化し、クラスタの情報を取得できるほか、クラスタに変更を加えることを可能にする。`kubelet` は、各ノードの状態を監視し、コントロールプレーンに定期的にレポートを送信する。これらのアプリケーションは、Kubernetes クラスタを立ち上げて管理するのに必要不可欠となる。

`kubeadm`, `kubectl`, `kubelet` は、コントロールプレーンノードとワーカーノードの両方にインストールする。まず、Linux のファイアウォール設定をまとめた `iptables` を設定する。

```
$ cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF

$ sudo sysctl --system
* Applying /etc/sysctl.d/10-console-messages.conf ...
kernel.printk = 4 4 1 7
...
fs.protected_symlinks = 1
* Applying /etc/sysctl.conf ...
```

`Iptables` の設定が完了したら、`kubeadm`, `kubectl`, `kubelet` をインストールする。

```
$ sudo apt-get update && sudo apt-get install -y apt-transport-https curl
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
...
Unpacking apt-transport-https (2.0.9) ...
Setting up apt-transport-https (2.0.9) ...
```

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
OK
```

```
$ cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
> deb https://apt.kubernetes.io/ kubernetes-xenial main
> EOF
deb https://apt.kubernetes.io/ kubernetes-xenial main
```

```
$ sudo apt-get update
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
Get:1 file:/var/cuda-repo-ubuntu2004-11-7-local InRelease [1,575 B]
...
Fetched 906 kB in 3s (353 kB/s)
Reading package lists... Done
```

```
$ sudo apt-get install -y kubelet=1.25.5-00 kubeadm=1.25.5-00 kubectl=1.25.5-00
Reading package lists... Done
Building dependency tree
...
Setting up kubeadm (1.26.1-00) ...
Processing triggers for man-db (2.9.1-1) ...
```

```
$ sudo apt-mark hold kubelet kubeadm kubectl
kubelet set on hold.
kubeadm set on hold.
kubectl set on hold.
```

## 2.6. Helm のインストール

ここでは、Kubernetes へのアプリケーションの適用を簡単に行える Helm をインストールしていく。本来は、Kubernetes クラスタに複雑な設定のアプリケーションをインストー

ルする際には、`kubectl apply` コマンドを大量に打ち込むことになる。Helm を使うことで、これらの複雑なインストール作業をコマンドひとつで実行できる。設定ファイルが数百にも及ぶアプリケーションも簡単にインストールできるため、Helm は事実上 Kubernetes の標準パッケージマネージャとなっている。

Helm は、コントロールプレーンノードにインストールする。次のコマンドを実行し、Helm をインストールする。

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
```

```
$ chmod 700 get_helm.sh
```

```
$ ./get_helm.sh
```

```
Downloading https://get.helm.sh/helm-v3.11.0-linux-amd64.tar.gz
```

```
Verifying checksum... Done.
```

```
Preparing to install helm into /usr/local/bin
```

```
helm installed into /usr/local/bin/helm
```

## 2.7. NFS サーバーの構築

ここでは、NFS サーバーを設定していく。NFS とは Network File System の頭文字を取った言葉である。ネットワーク上に NFS を共有ストレージとして設定することで、複数のマシンとデータを共有することができる。Kubernetes においては、JupyterHub 上のユーザーデータはクラウドストレージや NFS に保存される。今回は物理サーバーを使ったオンプレミス構成であるため、共有ストレージとして NFS を使用する。

まず、Ubuntu に NFS サーバー用のパッケージをインストールする。

```
$ sudo apt install nfs-kernel-server
```

```
Reading package lists... Done
```

```
Building dependency tree
```

```
...
```

```
Processing triggers for man-db (2.9.1-1) ...
```

```
Processing triggers for systemd (245.4-4ubuntu3.19) ...
```

NFS に接続できる IP アドレスは、`/etc/exports` というファイルで定義することになっている。本研究では、`/nfs` というディレクトリをマウント NFS ディレクトリとして設定する。

```
$ cat << EOF > /etc/exports
```

```
/nfs 192.168.10.0/24(rw,sync,no_root_squash,no_subtree_check)
```

```
EOF
```

これらが完了したら、NFS サーバーを再起動して、`/etc/exports` の設定を反映させる。

```
$ sudo systemctl restart nfs-kernel-server
```

次に、NFS サーバーに接続する側の設定を行う。コントロールプレーンノードとワーカーノードの両方で、NFS クライアントをインストールする。

```
$ sudo apt-get install nfs-common
```

```
Reading package lists... Done
```

```
Building dependency tree
```

```
...
```

```
Processing triggers for man-db (2.9.1-1) ...
```

```
Processing triggers for libc-bin (2.31-0ubuntu9.9) ...
```

そして、コントロールプレーンノードとワーカーノードの両方で、NFS サーバーマウント用のディレクトリを作成し、権限を設定する。

```
$ sudo mkdir /nfs
```

```
$ sudo chmod 777 /nfs
```

これらが完了したら、実際に NFS サーバーをマウントできるか確認する。

```
$ sudo mount -t nfs -v 192.168.10.51:/nfs /nfs
```

```
mount.nfs: timeout set for Sat Feb  4 10:44:49 2023
```

```
mount.nfs: trying text-based options 'vers=4.2,addr=192.168.10.51,clientaddr=192.168.10.28'
```

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            32G   0    32G   0% /dev
tmpfs           6.3G  2.8M  6.3G   1% /run
...
tmpfs           6.3G   32K  6.3G   1% /run/user/1000
192.168.10.51:/nfs 109G  11G   93G  11% /nfs
```

最後の行に NFS サーバーの情報が表示されている。これより、NFS サーバーが正常にマウントされていることが分かる。

これが確認できたら、マウントを解除しておく。実際の Kubernetes 上では、NFS サーバーにマウントするのは管理者ではなく Pod であるため、可能な限りマウントしているクライアントを減らし、NFS サーバーへの負荷を減らしておく。

```
$ sudo umount /nfs
```

## 2.8. Kubernetes クラスタの立ち上げ

以上の操作が完了して初めて、Kubernetes クラスタを立ち上げることができる。今回は、`kubeadm` コマンドを使用してクラスタを立ち上げることとする。

コントロールプレーンノード上で、以下のように `kubeadm init` コマンドを実行する。

```
$ sudo kubeadm init --pod-network-cidr=10.244.0.0/16
I0204 12:13:53.598457    11068 version.go:256] remote version is much newer: v1.26.1; falling
back to: stable-1.25
[init] Using Kubernetes version: v1.25.6
[preflight] Running pre-flight checks
...
Your Kubernetes control-plane has initialized successfully!
```

To start using your cluster, you need to run the following as a regular user:

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Alternatively, if you are the root user, you can run:

```
export KUBECONFIG=/etc/kubernetes/admin.conf
```

You should now deploy a pod network to the cluster.

Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:

<https://kubernetes.io/docs/concepts/cluster-administration/addons/>

Then you can join any number of worker nodes by running the following on each as root:

```
kubeadm join 192.168.10.47:6443 --token u8d00a.rtf02d1ibvucbly ¥
--discovery-token-ca-cert-hash
sha256:f0d618c896876f6158b9f76a387b1086ec8424c329833e02c2cb6d85026651bc
```

**kubeadm init** コマンドが正常に実行されると、そのログ出力に 2 種類のコマンドが表示される。1 つ目の指示は、クラスタの設定をシステムに移動させるコマンドである。これらをコントロールプレーンノード上で順番に実行していけば、**kubectl** コマンドが使えるようになる。

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

```
$ kubectl get pods -A
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	coredns-565d847f94-kdvcc	0/1	Pending	0	33m
kube-system	coredns-565d847f94-vvvzx	0/1	Pending	0	33m
kube-system	etcd-onoyama-controller	1/1	Running	0	33m
kube-system	kube-apiserver-onoyama-controller	1/1	Running	0	33m
kube-system	kube-controller-manager-onoyama-controller	1/1	Running	0	33m

kube-system	kube-proxy-7h4fn	1/1	Running	0	33m
kube-system	kube-scheduler-onoyama-controller	1/1	Running	0	33m

2つ目の `kubeadm join` コマンドは、クラスタにノードを追加するためのコマンドである。このコマンドはワーカーノードで実行する。

```
$ sudo kubeadm join 192.168.10.47:6443 --token u8d00a.rtf02d1ibvuccbly --discovery-token-ca-cert-hash sha256:f0d618c896876f6158b9f76a387b1086ec8424c329833e02c2cb6d85026651bc
```

```
[preflight] Running pre-flight checks
```

```
...
```

```
This node has joined the cluster:
```

- \* Certificate signing request was sent to apiserer and a response was received.
- \* The Kubelet was informed of the new secure connection details.

```
Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

`kubeadm join` コマンドが正常に実行されれば、ノードとしてクラスタに追加される。このノードの情報も、`kubectl` コマンドによって確認できる。

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
onoyama-controller	Ready	control-plane	62m	v1.26.1
onoyama-worker	Ready	<none>	4m27s	v1.26.1

## 2.9. Flannel のインストール

先程のクラスタ構築の際、`kubectl` コマンドを用いて Pod を確認した。このうち、CoreDNS の Pod は Pending 状態になっていた。これは、Pod 同士の通信が行えず、CoreDNS が常に待機状態になっているということである。Pod 同士のネットワークを構築するアプリケーションとして、本研究では Flannel をインストールする。

Flannel は、GitHub 上のリソースを指定した状態で、`kubectl` コマンドからインストールする。なお、`kubectl` コマンドでクラスタの設定を行う際には、`yaml` ファイルに設定項目を記述した上で、`kubectl apply -f [yaml file]` というコマンドを実行するのが通例となっている。

```
$ kubectl apply -f https://github.com/flannel-io/flannel/releases/latest/download/kube-
flannel.yml
namespace/kube-flannel created
serviceaccount/flannel created
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
configmap/kube-flannel-cfg created
daemonset.apps/kube-flannel-ds created
```

正常にインストールが完了すると、Pod 一覧に Flannel の Pod が追加されていることが確認できる。また、CoreDNS の Pod が Pending から Running へ変化していることも確認できる。

```
$ kubectl get pods -A
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-flannel	kube-flannel-ds-88f8x	1/1	Running	0	71s
kube-system	coredns-565d847f94-kdvcc	1/1	Running	0	42m
kube-system	coredns-565d847f94-vvvzx	1/1	Running	0	42m
kube-system	etcd-onoyama-controller	1/1	Running	0	43m
kube-system	kube-apiserver-onoyama-controller	1/1	Running	0	43m
kube-system	kube-controller-manager-onoyama-controller	1/1	Running	0	43m
kube-system	kube-proxy-7h4fn	1/1	Running	0	42m
kube-system	kube-scheduler-onoyama-controller	1/1	Running	0	43m

## 2.10. GPU Operator のインストール

次に、Kubernetes クラスタ上に GPU Operator をインストールする。GPU Operator は、Kubernetes クラスタに GPU を適用させる操作を自動で行えるツールである。コンテナエンジンを GPU に対応させたり、GPU の情報を Kubernetes に受け渡したりできるようになる。GPU Operator は、Helm の形式で配布されているため、Helm を使ってインストールする。

```
$ cat deploy-gpu-operator.sh
helm repo add nvidia https://nvidia.github.io/gpu-operator
helm repo update
```



```
helm install --wait --generate-name ¥
  -n gpu-operator --create-namespace ¥
nvidia/gpu-operator ¥
--set driver.enabled=false
```

```
$ sh deploy-gpu-operator.sh
"nvidia" has been added to your repositories
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "nvidia" chart repository
Update Complete. #Happy Helming!#
NAME: gpu-operator-1672113357
LAST DEPLOYED: Tue Dec 27 12:55:59 2022
NAMESPACE: gpu-operator
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

インストールが正常に終了すれば、`kubectl` コマンドで Pod がデプロイされていることが確認できる。

```
$ kubectl get pods -n gpu-operator
```

NAME	READY	STATUS	RESTARTS	AGE
gpu-feature-discovery-sk9vr	1/1	Running	0	7m25s
gpu-operator-1675483366-node-feature-discovery-master-5b78p86mp	1/1	Running	0	18m
gpu-operator-1675483366-node-feature-discovery-worker-pv5n8	1/1	Running	0	8m
gpu-operator-856595ff85-5zfrx	1/1	Running	0	18m
nvidia-container-toolkit-daemonset-z99s9	1/1	Running	0	7m25s
nvidia-cuda-validator-k5s6b	0/1	Completed	0	6m42s
nvidia-dcgm-exporter-kbfwm	1/1	Running	0	7m25s
nvidia-device-plugin-daemonset-nvkw	1/1	Running	0	7m25s
nvidia-device-plugin-validator-zrvsf	0/1	Completed	0	3m55s
nvidia-operator-validator-hh452	1/1	Running	0	7m25s

## 2.11. 永続ストレージ(PV)の設定

Kubernetes に JupyterHub をインストールする際には、永続ストレージ(Persistent Volume: PV)が必要となる。JupyterHub 上で作成したユーザーの名前やパスワード、ファイルなどは、Kubernetes の PV に保存する設計になっている。そのため、Kubernetes クラスタ上で、JupyterHub が使える PV を設定する必要がある。

そのために、まず Kubernetes クラスタに Kubernetes NFS Subdir External Provisioner をインストールする。NFS Subdir External Provisioner は、外部の NFS ストレージを Kubernetes 上に設定できるツールである。Kubernetes には、動的プロビジョニングという機能がある。これは、アプリケーションからのリクエストを受けて、Kubernetes が動的にストレージを提供できるというものである。JupyterHub において、各ユーザーアカウントを管理するときには、ユーザーの新規作成などのために動的プロビジョニングが必須となる。動的プロビジョニングを有効にするには、利用可能なストレージを基に StorageClass を作成する必要がある。外部の NFS を参照して StorageClass を作成するために、NFS Subdir External Provisioner を使用する。

```
$ cat deploy-nfs-provisioner.sh

helm repo add nfs-subdir-external-provisioner https://kubernetes-sigs.github.io/nfs-subdir-external-provisioner/

helm upgrade --cleanup-on-fail ¥
  --install    nfs-subdir-external-provisioner    nfs-subdir-external-provisioner/nfs-subdir-external-provisioner ¥
  --namespace nfs ¥
  --create-namespace ¥
  --set nfs.server=192.168.10.51 ¥
  --set nfs.path=/nfs ¥
  --set storageClass.defaultClass=true
  --set storageClass.name=nfs

$ sh deploy-nfs-provisioner.sh

"nfs-subdir-external-provisioner" has been added to your repositories
Release "nfs-subdir-external-provisioner" does not exist. Installing it now.
NAME: nfs-subdir-external-provisioner
```

LAST DEPLOYED: Sat Feb 4 13:24:58 2023

NAMESPACE: nfs

STATUS: deployed

REVISION: 1

TEST SUITE: None

Kubernetes NFS Subdir External Provisioner を正常にインストールできれば、`kubectl` コマンドでその Pod を確認できる。

```
$ kubectl get pods -n nfs
```

NAME	READY	STATUS	RESTARTS	AGE
nfs-subdir-external-provisioner-6444b4cb44-6kxbd	1/1	Running	0	93s

Kubernetes NFS Subdir External Provisioner のインストールが完了したら、Kubernetes クラスタ上に Storage Class が生成されていることが確認できる。Storage Class は、Kubernetes 上でどういったストレージが欲しいのかを指定するために利用されるオブジェクトである。ストレージの名前や種類など、動的にプロビジョニングする際のストレージの条件が設定されている。

```
$ kubectl get sc
```

NAME	PROVISIONER	RECLAIMPOLICY	VOLUMEBINDINGMODE	ALLOWVOLUMEEXPANSION	AGE
nfs (default)	cluster.local/nfs-subdir-external-provisioner	Delete	Immediate	true	3d2h

これらが確認できたら、次に Persistent Volume Claim (PVC)を作成する。PVC は、クラスタに対してストレージの永続化を要求するリソースのことである。Kubernetes クラスタは、PVC を経由して PV を利用することになる。PVC では、参照元の StorageClass や容量などが指定されている。これらの条件をもとに、ストレージが割り当てられることになっている。

PVC を作成するには、`kubectl` コマンドを用いる。

```
$ cat pvc.yaml
```

```
apiVersion: v1
```

```
kind: PersistentVolumeClaim
```

```
metadata:
```

```
  name: pvc-nfs
```

```

  namespace: default
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: nfs

```

```

$ kubectl apply -f pvc.yaml
persistentvolumeclaim/pvc-nfs created

```

pvc.yaml の設定がクラスタに反映されれば、kubectl コマンドからその PVC を確認できる。

```

$ kubectl get pvc

```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
pvc-nfs	Bound	pvc-8f9d9039-e0b7-4f8b-bd69-455825e38f81	5Gi	RWO	nfs	105s

## 2.12. JupyterHub のインストール

最後に、JupyterHub をインストールする。JupyterHub は、ユーザーアカウントごとに個別の JupyterLab 環境を提供できるプラットフォームである。Kubernetes クラスタに JupyterHub をインストールするには、設定項目を config.yaml に記述した上で、helm upgrade コマンドを用いる。

```

$ cat config.yaml
# This file can update the JupyterHub Helm chart's default configuration values.
#
# For reference see the configuration reference and default values, but make
# sure to refer to the Helm chart version of interest to you!
#
# Introduction to YAML:      https://www.youtube.com/watch?v=cdLNKUoMc6c
#      Chart      config      reference:      https://zero-to-
jupyterhub.readthedocs.io/en/stable/resources/reference.html

```

```
# Chart default values: https://github.com/jupyterhub/zero-to-jupyterhub-
k8s/blob/HEAD/jupyterhub/values.yaml
```

```
# Available chart versions: https://jupyterhub.github.io/helm-chart/
```

```
#
```

```
singleuser:
```

```
  image:
```

```
    name: cschranz/gpu-jupyter
```

```
    tag: v1.4_cuda-11.6_ubuntu-20.04
```

```
  storage:
```

```
    capacity: 2Gi
```

```
    dynamic:
```

```
      storageClass: nfs
```

```
$ cat deploy-jupyterhub.sh
```

```
helm repo add jupyterhub https://jupyterhub.github.io/helm-chart/
```

```
helm repo update
```

```
helm upgrade --cleanup-on-fail ¥
```

```
  --install jhub jupyterhub/jupyterhub ¥
```

```
  --namespace jhub ¥
```

```
  --create-namespace ¥
```

```
  --version=2.0.0 ¥
```

```
  --debug ¥
```

```
  --timeout 10m0s ¥
```

```
  --values config.yaml
```

```
$ sh deploy-jupyterhub.sh
```

```
"jupyterhub" has been added to your repositories
```

```
Hang tight while we grab the latest from your chart repositories...
```

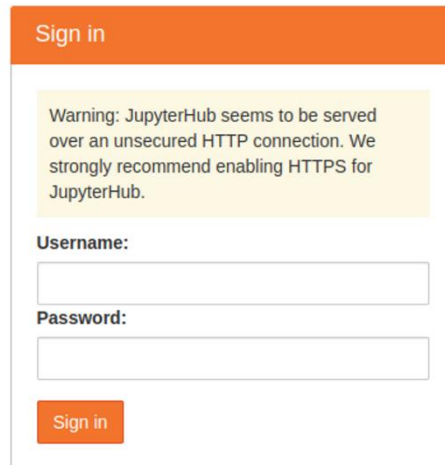
```
...Successfully got an update from the "nfs-subdir-external-provisioner" chart repository
```

```
...Successfully got an update from the "nvidia" chart repository
```

```
...Successfully got an update from the "jupyterhub" chart repository
```

```
Update Complete. *Happy Helming!*
```





The image shows a 'Sign in' form for JupyterHub. It has an orange header with the text 'Sign in'. Below the header is a yellow warning box with the text: 'Warning: JupyterHub seems to be served over an unsecured HTTP connection. We strongly recommend enabling HTTPS for JupyterHub.' Under the warning box are two input fields: 'Username:' and 'Password:'. At the bottom of the form is an orange 'Sign in' button.

図 6：JupyterHub のログイン画面

ここに、任意の Username と Password を入力する。今回の設定では、これらのセキュリティ関係の実装は行っていない。任意の Username と Password を入力すると、ユーザーページにログインできる。各ユーザーページでは、自分の JupyterLab 環境を使うことができる。また、カーネルの言語として Python(Tensorflow, Keras, PyTorch), Julia, R が使用できる。

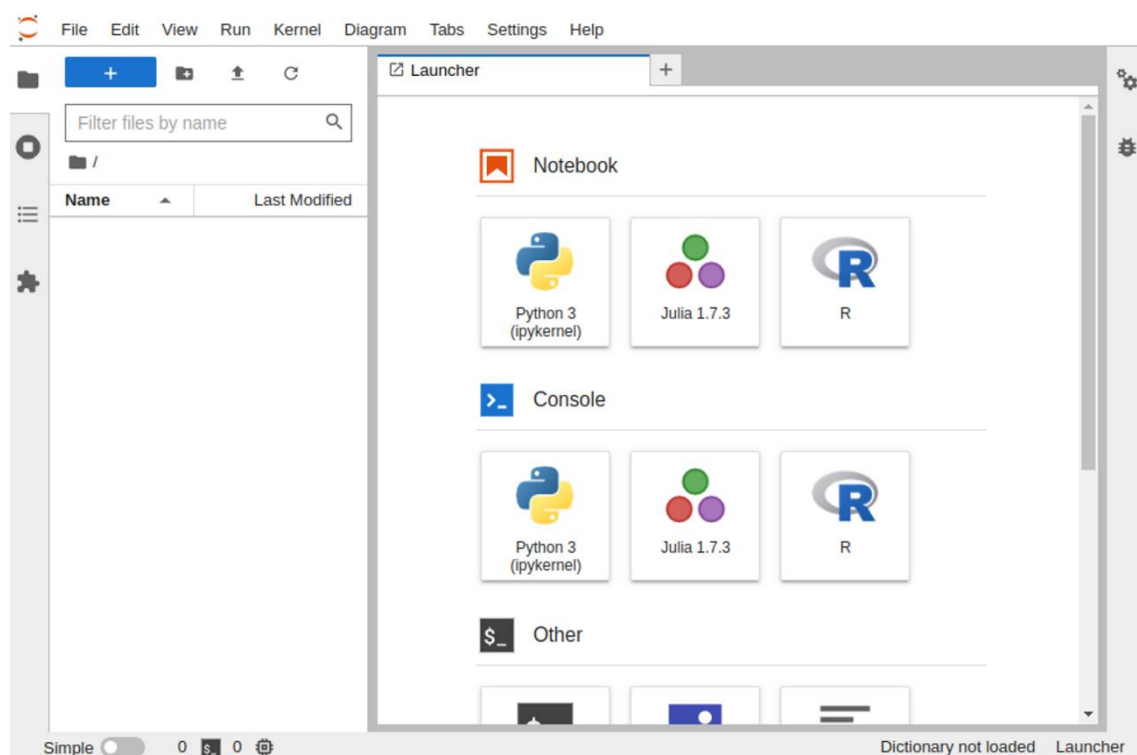


図 7 : JupyterHub のユーザーページ。Python の他にも、Julia や R といったデータ分析言語が使用可能。



## 3. 学習時間予測実験：方法

### 3.1. 学習時間の計測

以降の実験では、この JupyterLab 環境にて機械学習を学習させてデータを収集する。そしてそれらのデータを教師データとして、機械学習の学習時間を予測していく。

実行時間を予測していく上で、今回は以下の 5 つのデータを収集することにする。

- バッチサイズ
- エポック数
- 学習中のメモリ平均使用率
- 学習中の GPU 平均使用率
- 学習時間

バッチサイズは、教師データをどれだけまとめて処理するかを表す。例えば、10000 の教師データがあるとする。バッチサイズが 100 だと、一度に 100 のデータを処理するため、すべての教師データを処理するのに 100 回繰り返す必要がある。バッチサイズが 1000 だと、一度に 1000 のデータを処理するため、すべてのデータを処理するのに 10 回で済む。このように、バッチサイズは一度に学習するデータの束を表す。

エポック数は、同じ教師データを何回学習するかを表す。10000 の教師データをバッチサイズ 1000 で処理するとき、同じ処理を 10 回繰り返してすべての教師データを使ったとき、エポック数が 1 となる。

学習中のメモリ平均使用率・GPU 平均使用率は、エポックごとにメモリ・GPU の使用率を計測して合計し、それらをエポック数で乗算したものである。一般的に、時間のかかる学習というのはデータセットのサイズが大きかったり、学習プロセスの処理数が膨大であったりする。この際、メモリや GPU を多く割り当てることで、複数のデータや処理を並列して実行し、時間を短縮することができる。すなわち、メモリ・GPU の使用率が小さい処理より、メモリ・GPU の使用率が大きい処理の方が「重い」処理だと言える。この「重い」処理は、一般にメモリ・GPU を利用しても一定の時間がかかる。そのため、今回は実行時間予測の際の入力に使用する。

学習時間は、機械学習プロセスが開始してから終了するまで時間を計測したものである。

今回は、データのインポート時間や機械学習モデルの初期化時間は含めず、学習処理にのみ焦点を当てることとする。

今回は実行時間計測の対象として、以下の3つのタスクを実行することにする。

- MNIST の手書き画像データのクラス分類
- CIFAR10 の画像データのクラス分類
- 深層学習を用いた CIFAR-10 の画像データのクラス分類

MNIST データセットは、手書き数字 60000 枚とテスト画像 10000 枚を提供しているデータセットである。各データは 0~9 の数字の手書き画像であり、それぞれ正解ラベルが与えられている。機械学習では、初心者向けチュートリアル của データセットとしてよく使用される。

## 10 MNIST Images



図 8：MNIST データセットの手書き画像

MNIST データセットのクラス分類を行う際には、以下のようなモデルを使用する。

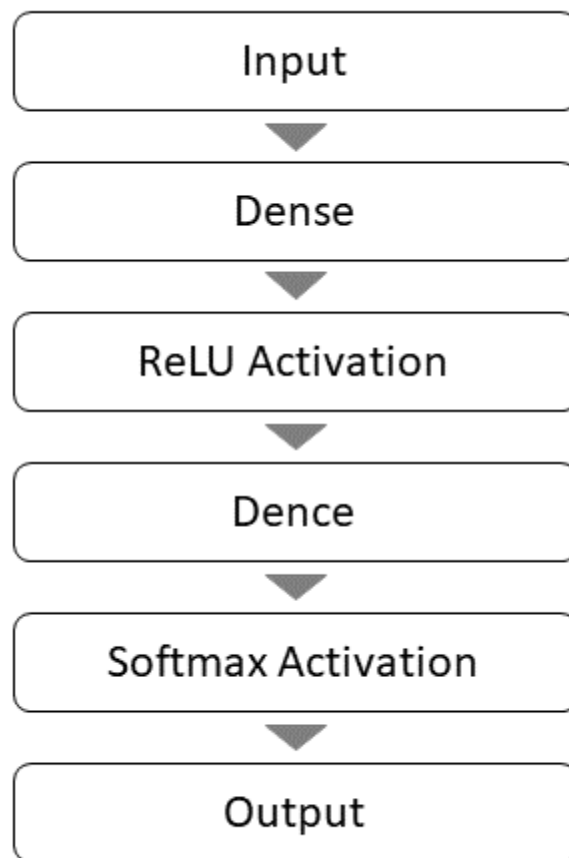


図 9：MNIST データセットのクラス分類を行う際のモデル構造

CIFAR-10 データセットは、以下の 10 種類のカラー画像とクラスを提供する画像データセットである。

- ラベル「0」： airplane（飛行機）
- ラベル「1」： automobile（自動車）
- ラベル「2」： bird（鳥）
- ラベル「3」： cat（猫）
- ラベル「4」： deer（鹿）
- ラベル「5」： dog（犬）
- ラベル「6」： frog（カエル）
- ラベル「7」： horse（馬）
- ラベル「8」： ship（船）
- ラベル「9」： truck（トラック）

データ数は8000万であるため、比較的大きいデータを扱う画像認識タスクで用いられる。このデータ数の多さを考慮すると、これらのデータセットを使うことで、GPU の使用率の振れ幅を大きくすることができる。

## 10 CIFAR-10 Images

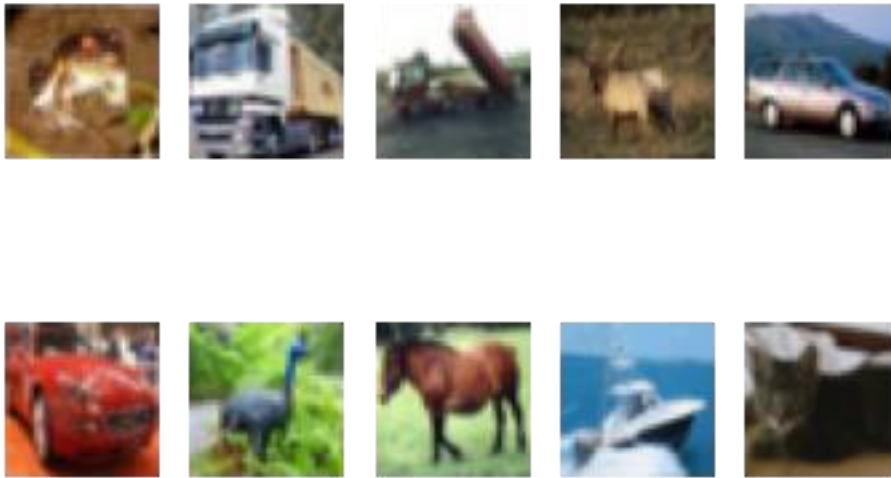


図 10：CIFAR-10 データセットの画像

CIFAR-10 データセットのクラス分類を行う際には、以下のようなモデルを使用する。

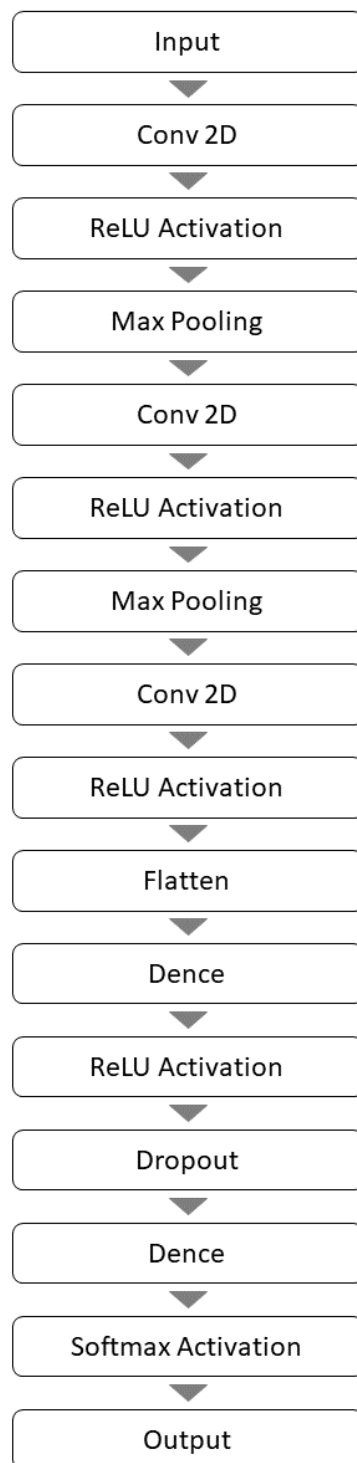


図 11 : CIFAR-10 データセットのクラス分類を行う際のモデル構造

また、GPU を最大限使用した状況のデータも収集するために、深層学習を用いた CIFAR-10 データセットのクラス分類も行う。

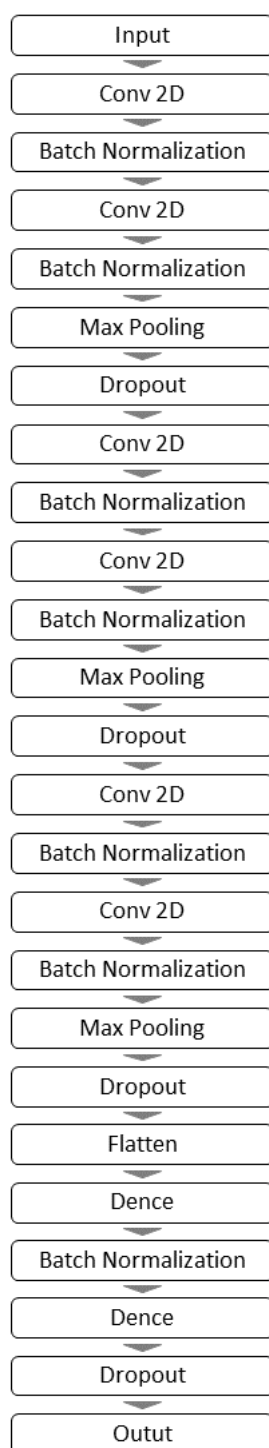


図 12：深層学習を使った CIFAR-10 画像のクラス分類を行う際のモデル画像

これらのクラス分類の学習処理を実行する際は、バッチサイズ、エポック数、メモリ・GPU 平均使用率、学習時間を計測する。計測したデータは、CSV ファイルに保存する。ここで、各カラムは左列からバッチサイズ、エポック数、メモリ・GPU 平均使用率、学習時間の順番である。バッチサイズは 100~ 800 の範囲で 10 刻み、エポック数も同様に 10, 20, 30, 40 の範囲で、条件を変えながら学習する。

以上の MNIST 手書き画像のクラス分類、CIFAR-10 画像のクラス分類、深層学習を使った CIFAR-10 画像のクラス分類の 3 つが、今回の実験の対象である。

### 3.2. 学習時間の予測・精度検証・重み計測

バッチサイズ、エポック数、平均メモリ・GPU 使用率、学習時間のデータが取得できたら、それらを教師データとして学習を行い、学習時間予測の精度を計測する。今回は、予測に 2 層パーセプトロンと 3 層ニューラルネットワークの 2 モデルを用いる。2 層パーセプトロンと 3 層ニューラルネットワークのモデル構造は、以下のようになる

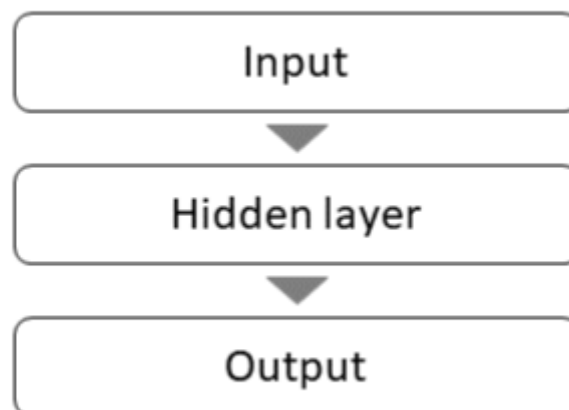


図 13：2 層パーセプトロンのモデル構造

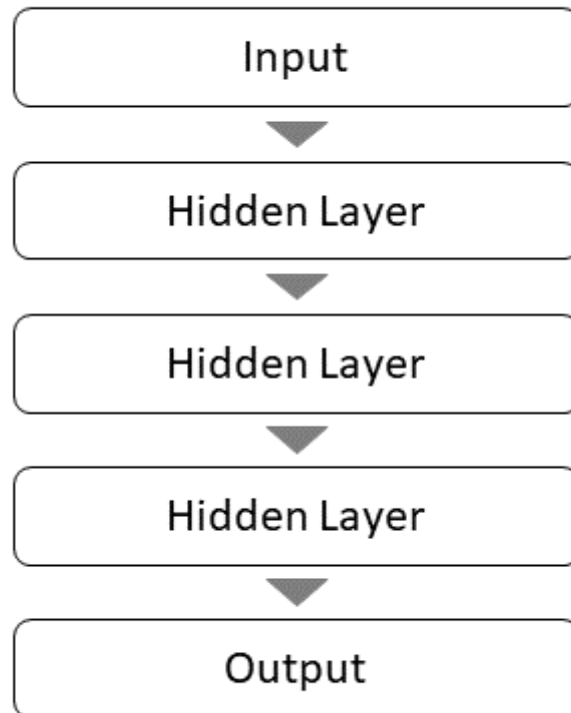


図 14：3 層ニューラルネットワークのモデル構造

3つの画像分類タスクで収集したデータ加えて、それらを全て結合したデータも用意する。そしてこれらのデータのうち、80%を学習データとして2モデルを学習させる。学習が完了すれば、モデルはバッチサイズ、エポック数、平均メモリ・GPU使用率から学習時間の予測値が出力される。ここに残りの20%のデータを入力し、実際の学習時間と予測値の学習時間との間にどれくらいの差があるのかを確認する。ここで、誤差の確認には決定係数(R<sup>2</sup>値)を用いる。R<sup>2</sup>値は、実際の値と予測値の差が小さくなるほど1に近づき、差が大きくなるほど0に近づく。そのため、このR<sup>2</sup>値を算出することで予測の精度を検証できる。今回はより精度を正確なものにするため、各データで1000回精度を出力し、それらを平均して最終的な精度としている。

また、エポック数がほぼ確実に学習時間に影響を与えることが予測されるため、2層パーセプトロンで学習させた際の重みを算出する。そして、各データをエポック数ごとに抽出し、そのデータを学習データとして2層パーセプトロンを学習させ、重みを確認する。この重みから、各パラメータが予測にどれくらい影響を与えるのかを検証する。



## 4. 学習時間予測実験：結果

### 4.1. データ収集の結果

MNIST 手書き画像分類、CIFAR10 画像分類、深層学習を使った CIFAR10 画像分類の 3 つのタスクにおいて、それぞれバッチサイズ、エポック数、学習時間の関係をプロットすると、以下のようになった。

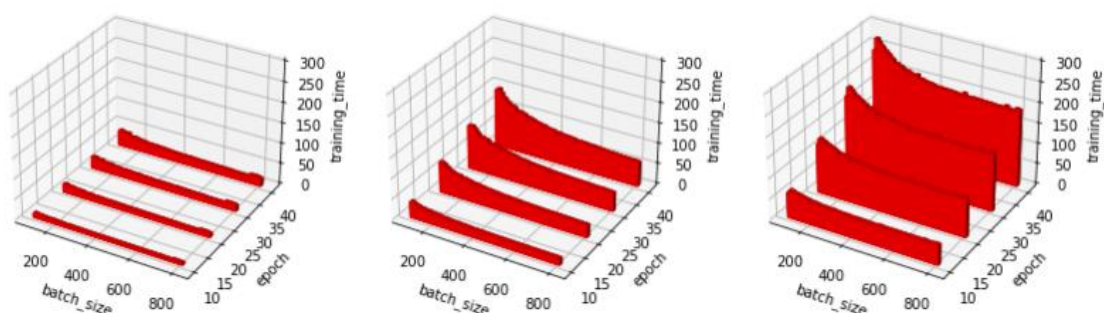


図 15：左から順に、MNIST 手書き画像分類、CIFAR10 画像分類、  
深層学習を使った CIFAR10 画像分類を行った際の、  
バッチサイズ、エポック数、学習時間

3 つのタスクにおいて、それぞれバッチサイズ、エポック数、平均メモリ使用率の関係をプロットすると、以下のようになった。

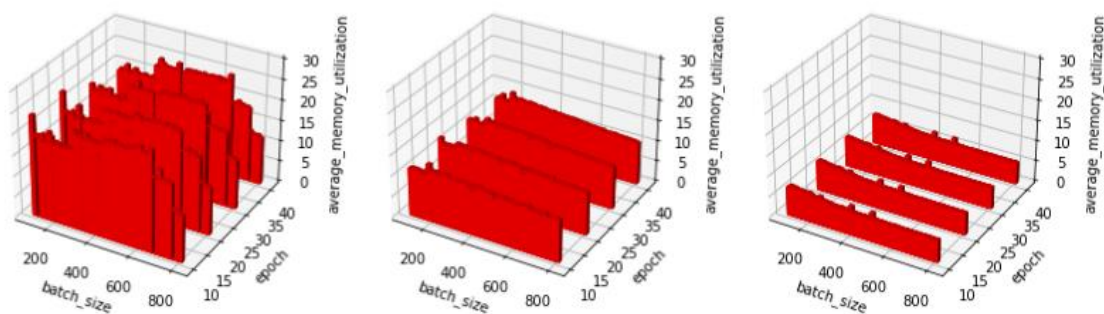


図 16：左から順に、MNIST 手書き画像分類、CIFAR10 画像分類、  
深層学習を使った CIFAR10 画像分類を行った際の、  
バッチサイズ、エポック数、平均メモリ使用率

各エポック数を固定した時のデータを以下に示す。

3つのタスクにおいて、それぞれバッチサイズ、エポック数、平均 GPU 使用率の関係をプロットすると、以下のようになった。

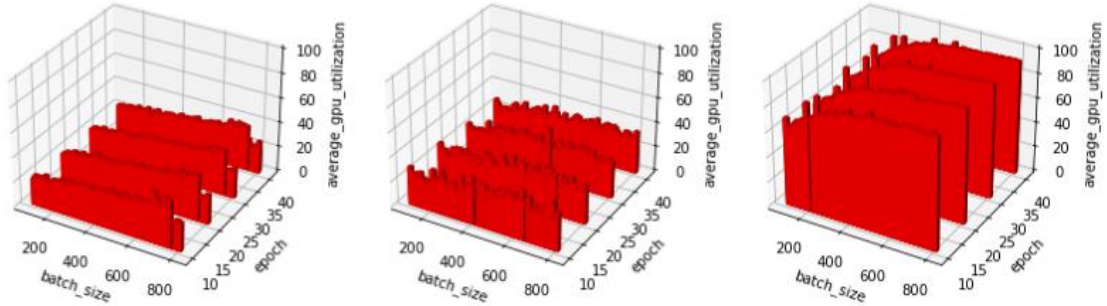


図 17：左から順に、MNIST 手書き画像分類、CIFAR10 画像分類、  
深層学習を使った CIFAR10 画像分類を行った際の、  
バッチサイズ、エポック数、平均 GPU 使用率

#### 4.2. 予測結果

MNIST 手書き画像分類データ、CIFAR10 画像分類データ、深層学習を使った CIFAR10 画像分類データ、全データの合計 4 データを、2 層パーセプトロンと 3 層ニューラルネットワークで学習させたときの精度は、以下の通りであった。

表 1：MNIST 手書き画像分類データ、CIFAR10 画像分類データ、  
深層学習を使った CIFAR10 画像分類データ、全データの合計 4 データを、  
2 層パーセプトロンと 3 層ニューラルネットワークで学習させたときの精度

	2 層パーセプトロン 予測精度	3 層ニューラルネットワーク 予測精度	2 層パーセプトロンの重み [バッチサイズ エポック数 平均メモリ使用率 平均 GPU 使用率]
MNIST 手書き画像分類データ	0.986	0.964	[-0.01285696 0.54008374 -0.25226827 -0.23678224]
CIFAR-10 画像分類データ	0.992	0.990	[-0.06620581 2.05091343 -5.68929198 -0.70034147]
深層学習を使った CIFAR-10 画像分類データ	0.991	0.987	[ 0.00952939 5.01498954 3.46986795 -1.62088255]
全データ	0.969	0.987	[-0.06986106 2.50171748 -3.68002805 0.94597626]

また、各学習のエポックを固定した際の学習精度と、パーセプトロンの重みは、以下の通りになった。

表 2：エポック数ごとに抽出したデータで学習させた際の、予測精度と重み

	エポック数	2 層パーセプトロン予測精度	3 層ニューラルネットワーク予測精度	2 層パーセプトロンの重み [バッチサイズ 平均メモリ使用率 平均 GPU 使用率]
MNIST 手書き画像 分類データ	10	-192.181	0.901	[-0.00475715 -0.07820823 -0.0994388]
	20	-4.799	0.941	[-0.01053435 -0.18731763 -0.19745749]
	30	-26.473	0.0.978	[-0.0127739 -0.24281611 -0.33596834]
	40	0.969	0.0.864	[-0.02101131 -0.44778225 -0.38912936]
CIFAR-10 画像分類 データ	10	-11.234	0.981	[-0.02972231 -2.50962314 -0.09803529]
	20	0.855	-5.252	[-0.0624954 -5.33692334 -0.22682121]
	30	0.880	0.956	[-0.0851214 -7.27338227 -0.91452773]
	40	-4.053	0.973	[ -0.12405639 -12.02518217 -0.93394118]
深層学習を使った CIFAR-10 画像分類 データ	10	0.620	-21.813	[ 0.00466382 1.77655967 -0.64376862]
	20	0.833	0.940	[ 0.00821367 0.91186608 -1.49232724]
	30	0.963	0.752	[ 0.01718635 4.37592832 -2.17672061]
	40	0.830	0.854	[ 0.02001662 4.05381492 -2.98452409]
全データ	10	0.909	0.944	[-0.02801807 -1.53644598 0.38853722]
	20	0.964	0.873	[-0.05448193 -3.01291374 0.75331869]
	30	0.950	0.984	[-0.0821298 -4.42969314 1.16343136]
	40	0.974	0.954	[-0.11922878 -6.29306038 1.44605052]

## 5. 学習時間予測実験：考察

### 5.1. データ収集の結果の考察

#### 5.1.1. バッチサイズ、エポック数、学習時間の関係

まず、バッチサイズ、エポック数、学習時間の関係について考察する。学習時間は、バッチサイズが増えると短くなり、エポック数が増えると長くなると言える。また、タスク間によって学習時間が大きく異なることも分かる。

バッチサイズが増えると学習時間が短くなる理由としては、バッチサイズが一度に処理するデータの数を表しているからであると考えられる。一度に処理するデータ数が大きくなれば、それだけデータを処理する手数は短くなる。さらに GPU は高度な並列計算機能を持つため、一度に処理するデータが増えようとも、それにかかる時間にはそれほど影響しないと考えられる。そのため、全体で見ると学習時間が短くなっていると言えるだろう。

エポック数が増えると学習時間が長くなる理由としては、エポック数が学習の繰り返し回数を表しているからだと考えられる。学習プロセスにおいては、すべてのデータを処理したとき、それを 1 エポックとしている。そのため、エポック数が増えれば学習時間はほぼ比例して増えていく。それが今回の実験結果にも反映されたと言える。

分類タスク間に注目しても、MNIST 分類タスク、CIFAR-10 分類タスク、深層学習版 CIFAR-10 分類タスクの順番で学習時間は単調に学習している。MNIST 画像よりも CIFAR-10 の方が解像度、枚数が多く、これにより CIFAR-10 の方がより処理に時間がかかるであろう。さらに同じ CIFAR-10 でも、層を深くすれば必要な計算量は増加する。これらの理由から、タスク間で学習時間に違いが見られるのは妥当であると言える。

#### 5.1.2. バッチサイズ、エポック数、平均メモリ使用率の関係

平均メモリ使用率は、エポック数にはそれほど関係せず、バッチサイズが増えると小さくなる傾向にあると言える。また、タスク間で見てみると、処理が重くなるほど平均メモリ使用率は小さくなると言える。

エポック数は、メモリの仕様率には直接関係しないと言える。そもそもエポック数は学習処理の繰り返し回数である。そのため、エポック数が増えたからといってメモリを多く使用する処理が実行されるわけではない。データ収集の結果を見ても、エポック数に対応した学

習時間の変化は見られない。以上より、エポック数が学習時間に与える影響は小さいと言って良いであろう。

バッチサイズは、大きくなるほど平均メモリ使用率が小さくなっている。バッチサイズが増えると、一度に処理するデータ数が増える。システムはこれらのデータを効率的に処理しようとするため、GPU に多くのデータを割り当てるようになる。これは後述する平均 GPU 使用率の傾向からも分かる。さらに、MNIST 手書き画像の分類よりも CIFAR-10 の画像分類の方が平均メモリ使用率は小さいのも、同じ理由であると考えられる。一般に、MNIST 画像より CIFAR-10 画像の方が解像度は高く、データ数も多いためである。同じ CIFAR-10 画像でも、層を深くすればより処理は複雑になる。こういった理由でより多くのデータを GPU に割り当てた結果、メモリが担う処理が少なくなり、結果的にメモリ使用量が少なくなったと考えられる。

### 5.1.3. バッチサイズ、エポック数、平均 GPU 使用率の関係

バッチサイズ、エポック数、平均 GPU 使用率の関係について考察する。平均 GPU 使用率は、エポック数にはほとんど影響されないが、バッチサイズが増えると大きくなると読み取れる。また、重いタスクになるにつれて平均 GPU 使用率は増加することもわかる。

エポック数が平均 GPU 使用率に影響しない理由は、平均メモリ使用率の時と同じである。エポック数は同じ処理をどれだけ繰り返すかを決めているだけで、処理の内容が変わることはない。回数が増加することによって変化するのは学習プロセスの所要時間だけで、他のパラメータにはほとんど関係がないと言える。

バッチサイズに関しては、大きくなるほど平均 GPU 使用率が大きくなっている、この理由は、先の平均メモリ使用率の時と逆である。すなわち、バッチサイズ増加によって平行処理するデータが増えた結果、多くの並行処理を GPU が担うようになるためである。タスク間の平均 GPU 使用率の差に関しても、同じ原因であると考えられる。複雑な処理になるほど、システムは GPU に処理を割り当てる傾向にあると言える。

## 5.2. 予測結果の考察

ここでは、予測した際の精度について考察していく。

全体的に、予測精度は高いと言える。R2 値は 0 から 1 の間の値を取る。実際の学習時間と予測された学習時間との差が小さいほど 1 に近い値となる。この R2 値がいずれも 0.96 を超えていることから、2 層パーセプトロンと 3 層ニューラルネットワークのいずれにおい

ても、高い精度で学習できたと言えるだろう。

各エポック数で抽出したデータを学習させたときは、予測精度にある程度の差が見られた。2層パーセプトロンの際は特に顕著で、学習処理が軽く、エポック数が少ない学習だと精度は悪かった。逆に、重い学習を実行させたときや、3層パーセプトロンで学習させたときは、エポック数をパラメータに入れずとも高い予測精度が確認できた。

### 5.3. 重みの考察

ここでは、実際に重みを計測した際の考察について述べる。

各予測における2層パラメータの重みでは、左から2番目の数値が最も高かった。すなわち、実験方法で述べたとおり、学習時間予測に最も影響するパラメータはエポック数であると断言できる。パーセプトロンにおいては、予測に重要なパラメータに大きな重みを付与し、学習における重要度を決定する。今回の実験でも、0.9代の予測精度を出した実験であったため、顕著に重みの差が表れていると考えられる。

### 5.4. 実用上の考察

ここでは、実際に学習時間予測を利用する場面について考察する。

今回の実験結果を踏まえると、画像分類タスクに限定すれば、機械学習の学習時間予測は可能であると言える。MNIST データセットと CIFAR-10 データセットのいずれにおいても、パラメータ間に一定の傾向は見られた。さらに、そこから学習させた結果、どのデータセット、どの学習モデルでも、一定の精度で予測することができた。さらに、異なる条件で実行した画像分類タスクのデータを全て合わせてもなお、一定の予測精度を保っていた。ここから、より多くの画像分類タスクでデータを収集できれば、予測モデルを汎化させることができると考える。画像分類以外の機械学習タスクではデータを収集していないため、考察することはできない。しかし、今回の結果から、画像分類にタスクを限定すれば、学習にかかる時間は予測できると考えられる。さらに、エポック数やバッチサイズ以外の、より多くのパラメータを使うことで、精度のさらなる向上や適用範囲の拡大も考えられるだろう。

今回の研究では、予測モデルを実際にシステムに組み込む方法などには言及していない。これらをベースにさらなる研究を重ね、本学の AI 教育の発展の一助とされれば幸いである。

## 6. 結論

本研究では、九州工業大学内で運用している JupyterHub システムの現状を踏まえて、機械学習タスクの学習時間予測を提案した。その際、新規で Kubernetes クラスタ環境を整え、その上に JupyterHub システムを構築した。この環境にて、MNIST 手書き画像分類、CIFAR10 画像分類、深層学習を使った CIFAR10 画像分類の 3 つのタスクを実行し、バッチサイズ、エポック数、メモリ平均使用率、GPU 平均使用率、学習時間を収集した。そしてそれらのデータを基に、2 層パーセプトロンと 3 層ニューラルネットワークで学習を行い、学習結果を予測した。その結果、いずれの条件においても、高い精度で予測を行うことができた。機械学習タスクの種類を増やしたり、より多くのパラメータを収集したりすれば、さらなる精度改善や汎化性能の向上も見込める。今回は JupyterHub のシステムに組みこむ方法などについては言及していない。今後の研究にて、本研究を参考にしてもらえれば幸いである。

## 7. 参考資料

1. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio. Generative Adversarial Networks. 2017
2. Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen. Progressive Growing of GANs for Improved Quality, Stability, and Variation. 2017
3. Tero Karras, Samuli Laine, Timo Aila. A Style-Based Generator Architecture for Generative Adversarial Networks. 2018
4. NVlabs/stylegan: StyleGAN - Official TensorFlow Implementation. <https://github.com/NVLabs/stylegan>
5. Dror G. Feitelson, Ahuva Mu'alem Weil. Utilization and Predictability in Scheduling the IBM SP2 with Backlling. 2001
6. 丹野 祐樹, 菅谷 至寛, 阿 曾 弘具. プロセス情報を利用した実行時間予測と信頼度による予測選択手法. 2007
7. 菅谷至寛, 丹野祐樹, 大町 真一郎, 阿曾弘具. 分散処理の効率化のための実行時間予測手法. 2011
8. 野村 哲弘, 佐々木 淳, 三浦 信一, 遠藤 敏夫, 松岡 聡. TSUBAME2 におけるジョブスケジューリング効率化への取り組みと検証. 2015
9. 川口 優樹, 津邑 公暁. メモリアクセスパターンを考慮した GPU スケジューリングポリシーの選択手法. 2019
10. 浜 野 智 明, 額 田 彰, 遠 藤 敏 夫, 松 岡 聡. GPU クラスタにおける省電力タスクスケジューリング. 2010
11. 青山真也. Kubernetes 完全ガイド. インプレス. 2020
12. John Arundel, Justin Domingus, 須田 一輝, 渡邊 了介. Kubernetes で実践するクラウドネイティブ DevOps. オライリージャパン. 2020
13. Docker: Accelerated, Containerized Application Development <https://www.docker.com/>
14. Kubernetes <https://kubernetes.io/ja/>
15. Helm <https://helm.sh/ja/>
16. Zero to JupyterHub with Kubernetes — Zero to JupyterHub with Kubernetes documentation <https://z2jh.jupyter.org/en/stable/>
17. flannel-io/flannel: flannel is a network fabric for containers, designed for Kubernetes <https://github.com/flannel-io/flannel>
18. Getting Started — NVIDIA Cloud Native Technologies documentation <https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/getting-started.html>



19. 岡田 賢治, 小山田 耕二. CUDA 高速 GPU プログラミング入門. 秀和システム. 2010
20. CUDA Toolkit 11.7 Downloads | NVIDIA Developer  
[https://developer.nvidia.com/cuda-11-7-0-download-archive?target\\_os=Linux&target\\_arch=x86\\_64&Distribution=Ubuntu&target\\_version=20.04&target\\_type=deb\\_local](https://developer.nvidia.com/cuda-11-7-0-download-archive?target_os=Linux&target_arch=x86_64&Distribution=Ubuntu&target_version=20.04&target_type=deb_local)
21. Andreas C. Muller, Sarah Guido, 中田 秀基. Python ではじめる機械学習 —scikit-learnで学ぶ特徴量エンジニアリングと機械学習の基礎. オライリージャパン. 2017

## 8. 補足資料

### 8.1. MNIST 手書き画像のクラス分類を行う Python のコード

MNIST 手書き画像のクラス分類を行う Python のコードは、以下の通りである。

```
import csv
import time
import os
import subprocess
import psutil
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

class CustomCallback(keras.callbacks.Callback):
    def __init__(self):
        super(CustomCallback, self).__init__()
        self.training_time = None
        self.average_memory_utilization = None
        self.average_gpu_utilization = None

    def on_train_begin(self, logs={}):
        self.start_time = time.time()
        self.memory_utilizations = []
        self.gpu_utilizations = []

    def on_train_end(self, logs={}):
        self.training_time = time.time() - self.start_time
        self.average_memory_utilization = np.mean(self.memory_utilizations)
```

```

        self.average_gpu_utilization = np.mean(self.gpu_utilizations)
        print('\nTotal training time: {} seconds'.format(self.training_time))
        print('\nAverage memory utilization during training so far:
{}'.format(self.average_memory_utilization))
        print('\nAverage GPU utilization during training so far:
{}'.format(self.average_gpu_utilization))

    def on_epoch_end(self, epoch, logs={}):
        memory_utilization = psutil.cpu_percent()
        gpu_utilization_process = subprocess.Popen(['nvidia-smi', '--query-
gpu=utilization.gpu', '--format=csv,noheader'], stdout=subprocess.PIPE)
        gpu_utilization = int(gpu_utilization_process.stdout.readline().strip().decode('utf-
8')).split()[0])
        self.memory_utilizations.append(memory_utilization)
        self.gpu_utilizations.append(gpu_utilization)

def Classify(batch_size, epochs):
    # Load the MNIST dataset
    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

    # Preprocess the data
    x_train = x_train.reshape(60000, 784)
    x_test = x_test.reshape(10000, 784)
    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    x_train /= 255
    x_test /= 255
    y_train = to_categorical(y_train, 10)
    y_test = to_categorical(y_test, 10)

    # Define the model
    model = Sequential()
    model.add(Dense(512, activation='relu', input_shape=(784,)))
    model.add(Dense(10, activation='softmax'))

```

```

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
callback = CustomCallback()
model.fit(x_train, y_train, epochs=epochs, batch_size=batch_size, callbacks=[callback])

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test accuracy:', test_acc)

with open('mnist_classification.csv', 'a') as f:
    writer = csv.writer(f)
    writer.writerow([batch_size, epochs, callback.average_memory_utilization,
callback.average_gpu_utilization, callback.training_time, test_loss, test_acc])

for batch in range(100, 801, 10):
    for epoch in range(10, 41, 10):
        Classify(batch, epoch)

```

## 8.2. CIFAR-10 画像のクラス分類を行う Python のコード

CIFAR-10 画像のクラス分類を行う Python のコードは、以下の通りである。

```

import csv
import time
import os
import subprocess
import psutil
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential

```

```

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

class CustomCallback(keras.callbacks.Callback):
    def __init__(self):
        super(CustomCallback, self).__init__()
        self.training_time = None
        self.average_memory_utilization = None
        self.average_gpu_utilization = None

    def on_train_begin(self, logs={}):
        self.start_time = time.time()
        self.memory_utilizations = []
        self.gpu_utilizations = []

    def on_train_end(self, logs={}):
        self.training_time = time.time() - self.start_time
        self.average_memory_utilization = np.mean(self.memory_utilizations)
        self.average_gpu_utilization = np.mean(self.gpu_utilizations)
        print('\nTotal training time: {} seconds'.format(self.training_time))
        print('\nAverage      memory      utilization      during      training      so      far:
{}'.format(self.average_memory_utilization))
        print('\nAverage      GPU      utilization      during      training      so      far:
{}'.format(self.average_gpu_utilization))

    def on_epoch_end(self, epoch, logs={}):
        memory_utilization = psutil.cpu_percent()
        gpu_utilization_process = subprocess.Popen(['nvidia-smi', '--query-gpu=utilization.gpu',
        '--format=csv,noheader'], stdout=subprocess.PIPE)
        gpu_utilization = int(gpu_utilization_process.stdout.readline().strip().decode('utf-
8').split()[0])
        self.memory_utilizations.append(memory_utilization)
        self.gpu_utilizations.append(gpu_utilization)

def Classify(batch_size, epochs):

```

```

# Load the CIFAR10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

# Preprocess the data
x_train = x_train / 255.0
x_test = x_test / 255.0

# Build the model
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Train the model
print("\n◆Train:")
callback = CustomCallback()
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
validation_data=(x_test, y_test), callbacks=[callback])
print("Done")

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test Accuracy:", test_acc)

with open('cifar10_classification.csv', 'a') as f:
    writer = csv.writer(f)

```

```

        writer.writerow([batch_size, epochs, callback.average_memory_utilization,
callback.average_gpu_utilization, callback.training_time, test_loss, test_acc])

```

```

for batch in range(100, 801, 100):
    for epoch in range(10, 41, 10):
        Classify(batch, epoch)

```

### 8.3. 深層学習版の CIFAR-10 分類タスクの Python コード

深層学習版の CIFAR-10 分類タスクのコードは以下の通りである。

```

import csv
import time
import os
import subprocess
import psutil
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D,
BatchNormalization

class CustomCallback(keras.callbacks.Callback):
    def __init__(self):
        super(CustomCallback, self).__init__()
        self.training_time = None
        self.average_memory_utilization = None
        self.average_gpu_utilization = None

```

```

def on_train_begin(self, logs={}):
    self.start_time = time.time()
    self.memory_utilizations = []
    self.gpu_utilizations = []

def on_train_end(self, logs={}):
    self.training_time = time.time() - self.start_time
    self.average_memory_utilization = np.mean(self.memory_utilizations)
    self.average_gpu_utilization = np.mean(self.gpu_utilizations)
    print('\nTotal training time: {} seconds'.format(self.training_time))
    print('\nAverage memory utilization during training so far:
{}'.format(self.average_memory_utilization))
    print('\nAverage GPU utilization during training so far:
{}'.format(self.average_gpu_utilization))

def on_epoch_end(self, epoch, logs={}):
    memory_utilization = psutil.cpu_percent()
    gpu_utilization_process = subprocess.Popen(['nvidia-smi', '--query-
gpu=utilization.gpu', '--format=csv,noheader'], stdout=subprocess.PIPE)
    gpu_utilization = int(gpu_utilization_process.stdout.readline().strip().decode('utf-
8').split()[0])
    self.memory_utilizations.append(memory_utilization)
    self.gpu_utilizations.append(gpu_utilization)

def Classify(batch_size, epochs):
    (x_train, y_train), (x_test, y_test) = cifar10.load_data()

    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    x_train /= 255
    x_test /= 255

    y_train = keras.utils.to_categorical(y_train, 10)
    y_test = keras.utils.to_categorical(y_test, 10)

```



```

model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)))
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))

model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.3))

model.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.4))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])

callback = CustomCallback()
model.fit(x_train, y_train, batch_size=batch_size, epochs=epoch, validation_data=(x_test,
y_test), callbacks=[callback], verbose=1)

test_loss, test_acc = model.evaluate(x_test, y_test)

```

```

with open('cifar10_deep_classification.csv', 'a') as f:
    writer = csv.writer(f)
    writer.writerow([batch_size, epochs, callback.average_memory_utilization,
callback.average_gpu_utilization, callback.training_time, test_loss, test_acc])

for batch in range(100, 801, 10):
    for epoch in range(10, 41, 10):
        Classify(batch, epoch)

```

#### 8.4. 2 層パーセプトロンの Python コード

学習時間予測の際に用いた 2 層パーセプトロンの Python コードは、以下の通りである。

```

import pandas as pd
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split

print("2 layer perceptron")
for task in ["mnist_classification", "cifar10_classification", "cifar10_deep_classification",
"all_classification"]:
    print("=" * 40)
    print(task)

    # Load the data into a pandas dataframe
    df = pd.read_csv(f"training_data/{task}.csv")

    # Split the data into input features and target output
    X = df[['batch_size', 'epoch', 'average_memory_utilization', 'average_gpu_utilization']]
    y = df['training_time']

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

    # Train a 2-layer perceptron model

```

```

mlp = MLPRegressor(hidden_layer_sizes=(100,), max_iter=100000)
mlp.fit(X_train, y_train)

# Calculate average score
total_score = 0
for _ in range(100):
    # Evaluate the model on the test data
    total_score += mlp.score(X_test, y_test)
print("Test score: ", total_score / 100)

```

## 8.5. 3層ニューラルネットワークのPythonコード

学習時間予測の際に用いた3層ニューラルネットワークのPythonコードは、以下の通りである。

```

import pandas as pd
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split

print("3layer neural network")
for task in ["mnist_classification", "cifar10_classification", "cifar10_deep_classification",
"all_classification"]:
    print("=" * 40)
    print(task)

    # Load the data into a pandas dataframe
    df = pd.read_csv(f"training_data/{task}.csv")

    # Split the data into input features and target output
    X = df[['batch_size', 'epoch', 'average_memory_utilization', 'average_gpu_utilization']]
    y = df['training_time']

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

```

```

# Train a 3-layer perceptron model
mlp = MLPRegressor(hidden_layer_sizes=(100, 50, 20), max_iter=100000)
mlp.fit(X_train, y_train)

# Calculate average score
total_score = 0
for _ in range(100):
    # Evaluate the model on the test data
    total_score += mlp.score(X_test, y_test)
print("Test score: ", total_score / 100)

```

## 8.6. 重み検証の際に使用した Python コード

通常のデータで重み検証を行った際の Python コードは、以下の通りである。

```

import pandas as pd
from sklearn.linear_model import Perceptron
from sklearn.linear_model import LinearRegression
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error

print("2 layer perceptron")
for task in ["mnist_classification", "cifar10_classification", "cifar10_deep_classification",
"all_classification"]:
    print("=" * 40)
    print(task)

    # Load the data into a pandas dataframe
    df = pd.read_csv(f"training_data/{task}.csv")

    # Split the data into input features (X) and output target (y)
    X = df[['batch_size', 'epoch', 'average_memory_utilization', 'average_gpu_utilization']]

```

```

y = df['training_time']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# sc = StandardScaler()
# sc.fit(X_train)
# X_train = sc.transform(X_train)
# X_test = sc.transform(X_test)

# Train a Perceptron model on the training data
model = LinearRegression()
model.fit(X_train, y_train)

print(model.coef_)

print("2 layer perceptron")
for task in ["mnist_classification", "cifar10_classification", "cifar10_deep_classification",
"all_classification"]:
    print("=" * 40)
    for epoch in [10, 20, 30, 40]:
        print(f"{task} with epoch {epoch}")

    # Load the data into a pandas dataframe
    df = pd.read_csv(f"training_data/{task}_with_epoch_{epoch}.csv")

    # Split the data into input features (X) and output target (y)
    X = df[['batch_size', 'average_memory_utilization', 'average_gpu_utilization']]
    y = df['training_time']

    # Split the data into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

    # sc = StandardScaler()
    # sc.fit(X_train)
    # X_train = sc.transform(X_train)

```

```
# X_test = sc.transform(X_test)

# Train a Perceptron model on the training data
model = LinearRegression()
model.fit(X_train, y_train)

print(model.coef_)
```



## 謝辞

本論文を執筆するにあたり、研究に関するさまざまなタスクをサポートしてくれた花沢研究室のメンバー、そして、本論文に関する助言を下さった花沢明俊先生に、深い感謝の意を表す。