



NVIDIA ディープラーニング フレームワーク コンテナ

ユーザー ガイド

目次

第1章 Docker コンテナ	01
1.1. Docker コンテナとは	01
1.2. コンテナを使う理由	02
1.3. コンテナの Hello World	02
1.4. Docker にログインする	02
1.5. Docker イメージの一覧を表示する	03
第2章 Docker と NVIDIA Container Toolkit のインストール	04
2.1. Docker のベスト プラクティス	04
2.2. docker exec	04
2.3. nvcrio	05
2.4. コンテナをビルドする	05
2.5. ファイル システムの使用とマウント	09
第3章 コンテナをプルする	11
3.1. 重要な概念	11
3.2. NGC コンテナ レジストリへのアクセスとプル	12
3.2.1. Docker CLI を使ってコンテナを NGC コンテナ レジストリからプルする	14
3.2.2. NGC Web インターフェイスを使ってコンテナをプルする	14
3.3. 検証する	15
第4章 NGC イメージ	17
4.1. NGC イメージのバージョン	18
第5章 コンテナを実行する	20
5.1. 使用例：コンテナを実行する	20
5.2. ユーザーを指定する	21
5.3. 削除フラグを設定する	21
5.4. 対話フラグを設定する	21
5.5. ボリューム フラグを設定する	22
5.6. ポート マッピング フラグを設定する	22
5.7. 共有メモリ フラグを設定する	23
5.8. GPU 公開制限フラグを設定する	23
5.9. コンテナの有効期間	23
第6章 NVIDIA ディープラーニング ソフトウェア スタック	25
6.1. OS レイヤー	25
6.2. CUDA レイヤー	25
6.2.1. CUDA ランタイム	26
6.2.2. CUDA Toolkit	26

6.3. ディープラーニング ライブラリ レイヤー	26
6.3.1. NCCL	26
6.3.2. cuDNN レイヤー	27
6.4. フレームワーク コンテナ	27
第 7 章 NVIDIA ディープラーニング フレームワーク コンテナ	29
7.1. DL/ML ソフトウェア フレームワークを使う理由	29
7.2. Kaldi	30
7.3. Apache MXNet を基盤とする NVIDIA の最適化ディープラーニング	30
7.4. TensorFlow	30
7.4.1. TensorFlow コンテナを実行する	31
7.5. PyTorch	31
7.6. DIGITS	31
7.6.1. DIGITS をセットアップする	32
7.6.2. DIGITS を実行する	32
第 8 章 フレームワークの一般的なベスト プラクティス	35
8.1. コンテナを拡張する	35
8.2. データセットとコンテナ	35
8.3. Keras とコンテナ化フレームワーク	36
8.3.1. Keras をコンテナに追加する	37
8.3.2. Keras 仮想 Python 環境を作成する	37
8.3.3. Keras 仮想 Python 環境とコンテナ化フレームワークを使用する	39
8.3.4. コンテナ化 VNC デスクトップ環境を操作する	42
第 9 章 HPC と HPC 可視化コンテナ	43
第 10 章 コンテナとフレームワークのカスタマイズと拡張	44
10.1. コンテナをカスタマイズする	44
10.1.1. コンテナをカスタマイズするメリットと制限事項	45
10.1.2. 使用例 1：コンテナをゼロからビルドする	45
10.1.3. 使用例 2：Dockerfile を使用してコンテナをカスタマイズする	47
10.1.4. 使用例 3：docker commit を使用してコンテナをカスタマイズする	48
10.1.5. 使用例 4：Docker を使用してコンテナを開発する	51
10.1.5.1. 使用例 4.1：ソースをコンテナにパッケージ化する	53
10.2. フレームワークをカスタマイズする	53
10.2.1. フレームワークをカスタマイズするメリットと制限事項	53
10.2.2. 使用例 1：コマンド ラインを使用してフレームワークをカスタマイズする	53
10.2.3. 使用例 2：フレームワークをカスタマイズし、コンテナを再ビルドする	54
10.3. Docker コンテナのサイズを最適化する	56
10.3.1. 各 RUN コマンドを 1 行に記述する	56
10.3.2. エクスポート、インポート、フラット化	58

10.3.3. docker-squash	58
10.3.4. ビルド中にスカッシュする	59
10.3.5. その他のオプション	59
第 11 章 スクリプト	62
11.1. DIGITS.....	62
11.1.1. run_digits.sh.....	62
11.1.2. digits_config_env.sh.....	62
11.2. TensorFlow.....	62
11.2.1. run_tf_cifar10.sh	62
11.3. Keras	63
11.3.1. venvfns.sh	63
11.3.2. setup_keras.sh	64
11.3.3. run_kerastf_mnist.....	64
11.3.4. run_kerasth_mnist.....	65
11.3.5. run_kerastf_cifar10.sh.....	65
11.3.6. run_keras_script	66
11.3.7. cifar10_cnn_filesystem.py	68
第 12 章 トラブルシューティング	72

第 1 章 Docker コンテナ

この数年間で、データセンター アプリケーションの大規模開発を簡素化するためにソフトウェア コンテナを利用する動きが急激に広がりました。コンテナは、アプリケーションや、アプリケーションで使われるライブラリなどの依存関係を 1 つにまとめることで、完全な仮想マシンのオーバーヘッドを回避し、アプリケーションとサービスの実行に再現可能性と信頼性をもたらします。

NVIDIA Container Runtime for Docker、別名 [nvidia-docker2](#) は、複数のマシンに移植できる GPU ベースのアプリケーションを実現します。これは、Docker® が CPU ベースのアプリケーションを複数のマシンに展開できるのと同様に似ています。これを実現するために使用するのが Docker コンテナです。

Docker イメージ

Docker イメージとは、Docker コンテナ内で実行するソフトウェアのことで、ファイルシステムとパラメーターもそれに含まれます。

Docker コンテナ

Docker コンテナは、Docker イメージのインスタンスです。Docker コンテナでは、1 つのアプリケーションまたはサービスが 1 つのコンテナで展開されます。

1.1. Docker コンテナとは

Docker コンテナとは、Linux アプリケーションとそのすべてのライブラリ、データ ファイル、環境変数をひとまとめにすることで、どの Linux システムでも、同じホスト上にある複数のインスタンス間で実行環境が常に同じになるようにするメカニズムです。

仮想マシンにはそれぞれに固有のカーネルがありますが、コンテナはこれとは違い、ホスト システムのカーネルを使います。つまり、コンテナからのカーネル呼び出しは、常にホスト システムのカーネルで処理されます。DGX™ システムでは、ディープ ラーニング フレームワークを展開するメカニズムとして Docker コンテナを使います。

Docker コンテナは、複数のレイヤーで構成されます。複数のレイヤーが結合して 1 つのコンテナとなります。レイヤーとは、一定の機能を全体のコンテナに追加する中間イメージであると考えられます。Dockerfile でレイヤーの変更を指定すると（「[コンテナのビルド](#)」を参照）、Docker によってそのレイヤーとすべての後続レイヤーが再ビルドされますが、このビルドに影響されないレイヤーは再ビルドされません。こうすることでコンテナを作成する時間が短縮され、コンテナをモジュール式で維持できます。

Docker は、システムにレイヤーを 1 組だけ維持するのもにも便利です。スペースを節約できるだけでなく、「バージョンのスキュー（差異）」が発生する可能性も大きく低下するため、同一のレイヤーが複製されることはありません。

Docker コンテナは、[Docker イメージ](#)の実行時インスタンスです。

1.2. コンテナを使う理由

コンテナの使用には、メリットがたくさんあります。その1つは、アプリケーション、依存関係、環境変数をコンテナ イメージに一度にインストールできることです。利用するシステムのすべてに、個別にインストールする必要はありません。それ以外にも、コンテナには以下のようなメリットがあります。

- ▶ コンテナ イメージへのアプリケーション、依存関係、環境変数のインストールは一度のみ。利用するシステムのすべてに、個別にインストールする必要はない。
- ▶ 他の人がインストールしたライブラリと競合するリスクがない。
- ▶ ソフトウェアの依存関係が競合する恐れがある、複数の異なるディープ ラーニング フレームワークを同じサーバー上で使用できる。
- ▶ アプリケーションをコンテナにビルドした後で、特にソフトウェアをインストールしなくても、そのコンテナを他の場所（特にサーバー上）で実行できる。
- ▶ 従来的高速化された計算アプリケーションをコンテナ化して、新しいシステム上、オンプレミス、またはクラウド内に展開できる。
- ▶ 分離とパフォーマンス向上を狙って、特定の GPU リソースをコンテナに割り当てることができる。
- ▶ アプリケーションの共有、共同開発、テストを複数の環境で簡単に行える。
- ▶ 特定のディープ ラーニング フレームワークから複数のインスタンスを作成し、特定の GPU を各インスタンスに個別に割り当てて、同時に実行できる。
- ▶ コンテナの起動時に、外部に公開された特定のポートにコンテナ ポートをマッピングすることで、ネットワークポートの競合を解決できる。

1.3. コンテナの Hello World

NVIDIA コンテナにアクセスできることを確認するために、まずは、あの由緒ある「hello world」を Docker コマンドで試してみましょう。

DGX システムをお使いの場合は、システムにログインします。現在サポートされている DGX システムについては、「[Frameworks Support Matrix](#)」を参照してください。NGC をお使いの場合は、利用中のクラウド プロバイダーの詳細について [NGC ドキュメント](#) でご確認ください。一般には、NVIDIA Deep Learning Image を使って、利用中のクラウド プロバイダーでクラウド インスタンスを起動します。インスタンスがブートされたら、そのインスタンスにログインします。

次に、`docker --version` コマンドを実行して、DGX システムのバージョンを表示します。このコマンドの出力で、システム上の Docker のバージョン（18.06.3-ce, build 89658be）がわかります。

Docker コマンドの使い方がわからないときは、いつでも `docker--help` コマンドで確認できます。

1.4. Docker にログインする

DGX システムをお使いの場合は、初回のログイン時に NVIDIA NGC コンテナ レジストリ (<https://ngc.nvidia.com>) へのアクセスを設定する必要があります。詳細については、「[NGC Getting Started Guide](#)」を参照してください。

1.5. Docker イメージの一覧を表示する

一般的に、最初に行うのは、ローカル コンピューターで現在使用可能な Docker イメージの一覧を表示することでしょう。docker pull コマンドを実行すると、Docker イメージがリポジトリからローカル システムにダウンロードされます。

docker images コマンドは、サーバーにあるイメージの一覧を出力します。画面には次のように表示されます。

REPOSITORY	TAG	IMAGE ID
mxnet-dec	latest	65a48e11da96
<none>	<none>	bfc4512ca5f2
nvcr.io/nvidian_general/adlr_	resumes	a134a09668a8
pytorch	<none>	0f4ab6d62241
<none>	<none>	97274da5c898
nvcr.io/nvidian_sas/games-	cuda10	3dc13f8347f9
libcchem	latest	614dcdafa05c
ubuntu	latest	d355ed3537e9
deeper_photo	latest	932634514d5a
nvcr.io/nvidia/caffe	19.03	b7b62dacdeb1
nvidia/cuda	10.0-devel-centos7	6e3e5b71176e
nvcr.io/nvidia/tensorflow	19.03	56f2980b1e37
nvidia/cuda	10.0-cudnn7-devel-ubuntu16.04	22afb0578249
nvidia/cuda	10.0-devel	a760a0cfca82
mxnet/python	gpu	7e7c9176319c
nvcr.io/nvidia_sas/chainer	latest	2ea707c58bea
deep_photo	latest	ef4510510506
<none>	<none>	9124236672fe
nvcr.io/nvidia/cuda	10.0-cudnn7-devel-ubuntu18.04	02910409eb5d
nvcr.io/nvidia/tensorflow	19.05	9dda0d5c344f

システムにプルされた Docker コンテナが、いくつかあることがわかります。各イメージには固有のタグとイメージ ID があり、この ID はコンテナ バージョンとも呼ばれます。さらに 2 つの列があり、そこにはコンテナの作成日（だいたいの日時）とおおよそのサイズ（GB 単位）が示されます。この表示例では読みやすさを考慮して、この 2 列は省きました。



メモ: このコマンドの出力は変動します。

ヘルプが必要であれば、いつでも docker images --help コマンドを使用できます。

第2章 Docker と NVIDIA Container Toolkit のインストール

このタスクの概要

GPU を利用する Docker イメージに移植性を与えるため、Docker コンテナに GPU サポートを提供する方法が2つ開発されました。

- ▶ ネイティブの GPU サポート
- ▶ nvidia-docker2

どちらの方法でも、コマンド ライン ツールを使って、NVIDIA ドライバーのユーザーモード コンポーネントと GPU を起動時に Docker コンテナにマウントします。

NGC コンテナは、NVIDIA GPU の能力をフルに活用します。詳細については、「[NGC Container User Guide for NGC Catalog](#)」を参照してください。

インストールの手順については、NVIDIA Container Toolkit の [Installation Guide](#) をご覧ください。

2.1. Docker のベスト プラクティス

Docker コンテナは、Docker と互換性があるプラットフォームならどこでも実行できるので、ユーザーはアプリケーションを必要な場所に移動できます。コンテナが特定のプラットフォームに縛られないことは、特定のハードウェアに縛られないことでもあります。最大限のパフォーマンスを実現し、NVIDIA GPU の途方もないパフォーマンスをフルに活用するには、特定のカーネル モジュールとユーザーレベルのライブラリが必要です。NVIDIA GPU を動かすにはカーネル モジュールとユーザーレベルのライブラリが必要ですが、これらによって処理が複雑になります。

コンテナの使用時にこの複雑さを解消する対策の1つは、NVIDIA ドライバーをコンテナにインストールし、キャラクター デバイスを NVIDIA GPU (`/dev/nvidia0` など) にマッピングすることです。これを行うには、ホスト (コンテナを実行するシステム) 上のドライバーと、コンテナにインストールされたドライバーのバージョンが一致している必要があります。この対策を採用すると、コンテナの移植性が大幅に低下します。

2.2. docker exec

実行中のコンテナへの接続が必要となる場合があります。`docker exec` コマンドを使うと、実行中のコンテナに接続してコマンドを実行できます。`bash` コマンドを使って、対話形式のコマンド ライン ターミナルまたは `bash` シェルを起動できます。


```
$ docker exec -it <CONTAINER_ID_OR_NAME> bash
```

たとえば、次のコマンドで Deep Learning GPU Training System™ (DIGITS) コンテナを起動したとします。

```
docker run --gpus all -d --name test-digits \
-u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
nvcr.io/nvidia/digits:17.05
```

コンテナが実行中になったら、コンテナ インスタンスに接続できます。

```
$ docker exec -it test-digits bash
```



メモ: test-digits はコンテナの名前です。コンテナに固有の名前を付けない場合は、コンテナ ID を使う必要があります。



重要: docker exec を使うと、コードのスニペット (スクリプト) を実行できます。コンテナに対話形式で接続すれば、docker exec コマンドをとっても便利に使うことができます。

docker exec コマンドの詳細な使い方については、「[docker exec](#)」を参照してください。

2.3. nvcr.io

ディープラーニングフレームワークのビルドは非常に手間のかかる作業で、かなりの時間を要します。しかも、そういったフレームワークは、毎日ではないとしても、毎週更新されるのです。そのうえ、フレームワークを GPU 向けに最適化し、調整することも必要です。NVIDIA は、nvcr.io という名前の Docker リポジトリを作成しました。ディープラーニングフレームワークのユーザー向けの調整や最適化、テスト、コンテナ化は、このリポジトリで行います。

NVIDIA は、このフレームワーク向けの Docker コンテナ セットを毎月更新します。コンテナの内部には、ソース (オープンソースのフレームワークであるため)、フレームワークビルド用のスクリプト、これらのコンテナに基づいてコンテナを作成するための Dockerfile、特定のコンテナに関するテキストが含まれるマークダウン ファイル、テストや学習に利用できるデータセットをプルするためのツールとスクリプトがあります。DGX システムを購入されたお客様は、このリポジトリにアクセスしてコンテナをプッシュ (保存) することができます。

DGX システムを使うには、nvcr.io へのアクセスに使うシステム管理者アカウントを作成する必要があります。このアカウントは管理者アカウントとして扱われるので、一般ユーザーはアクセスできません。このアカウントを作成した後で、システム管理者は、このアカウントに所属するプロジェクト用のアカウントを作成できます。システム管理者は、プロジェクトへのアクセス権をユーザーに与え、ユーザーがコンテナを作成して保存または共有できるようにします。

2.4. コンテナをビルドする

このタスクの概要

DGX システムをお使いの場合は、コンテナをビルドして、nvcr.io リポジトリに自分のアカウントのプロ

ジェクトとして保存できます（たとえば、自分でアクセスを許可しない限り、他の人はこのコンテナにアクセスできません）。

このセクションの説明は、Docker コンテナ全般に当てはまります。この方法は個人用の Docker リポジトリにも使えますが、その際は細かい部分に注意を払ってください。

DGX システムでは、次のいずれかを行えます。

1. 新しいコンテナをゼロから作成する
2. 既存の Docker コンテナをもとに新しいコンテナを作成する
3. `nvcr.io` にあるコンテナをもとに新しいコンテナを作成する

どの方法でもコンテナを作成できますが、ここでの目標は、GPU があるシステムでコンテナを実行することなので、当然ながらアプリケーションは GPU を使うと仮定できます。しかも、これらのコンテナは GPU 向けにすでに調整済みです。必要な GPU ライブラリ、構成ファイル、コンテナ再ビルド ツールもすでに含まれています。



重要：このような前提があることから、コンテナの作成に `nvcr.io` を使うことを推奨します。

`nvcr.io` にある既存のコンテナを土台にして、新しいコンテナの作成を始めてください。ここでは作成例として TensorFlow 17.06 コンテナを使い、[Octave](#) をこのコンテナに追加して、結果に対して後処理を実行できるようにします。

手順

1. NGC コンテナ レジストリからコンテナをサーバーにプルします（「[コンテナをプルする](#)」を参照）。
2. サーバー上に `mydocker` というサブディレクトリを作成します。



メモ：これは任意に決められるディレクトリ名です。

3. このディレクトリの下に、「`Dockerfile`」というファイルを作成します（1文字目を大文字にしてください）。これは Docker がコンテナの作成時にデフォルトで探す名前です。一般に `Dockerfile` の内容は次のようなものです。

```
[username ~]$ mkdir mydocker
[username ~]$ cd mydocker
[username mydocker]$ vi Dockerfile
[username mydocker]$ more Dockerfile
FROM nvcr.io/nvidia/tensorflow:19.03
RUN apt-get update

RUN apt-get install -y octave
[username mydocker]$
```

`Dockerfile` には 3 つの行があります。

- ▶ `Dockerfile` の最初の行は、コンテナ `nvcr.io/nvidia/tensorflow:17.06` から始めることを Docker に指示します。これは新しいコンテナの基盤とするコンテナです。
- ▶ `Dockerfile` の 2 行目では、コンテナのパッケージアップデートを実行します。コンテナ内のアプリケーションは更新されませんが、`apt-get` データベースが更新されます。新しいアプリケーション

ンをコンテナにインストールする前に、このアップデートが必要です。

- ▶ Dockerfile の3行目、つまり最後の行では、`apt-get` を使って `octave` パッケージをコンテナにインストールすることを Docker に指示します。

コンテナを作成するには、次の Docker コマンドを使います。

```
$ docker build -t nvcr.io/nvidian_sas/tensorflow_octave:17.06_with_octave
```



メモ: このコマンドでは、コンテナの作成にデフォルト ファイルの Dockerfile を使っています。

コマンドは `docker build` で始まります。-t オプションは、この新しいコンテナに対応するタグを作成することを意味します。このタグで、コンテナの保存先である `nvcr.io` リポジトリ内のプロジェクトが特定されます。この例では、`nvidian_sas` というプロジェクトが `nvcr.io` リポジトリで使われています。

プロジェクトは、`nvcr.io` へのアクセスを管理するローカル管理者が作成できます。または、この管理者がプロジェクトの作成権限を一般ユーザーに与えることもできます。プロジェクトでは、新しいコンテナを保存でき、さらにはそれを同僚と共有することもできます。

```
[username mydocker]$ docker build -t nvcr.io/nvidian_sas/
tensorflow_octave:19.03_with_octave .
Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM nvcr.io/nvidia/tensorflow:1903
--> 56f2980ble37
Step 2/3 : RUN apt-get update
--> Running in 69cffa7bbadd
Get:1 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:2 http://ppa.launchpad.net/openjdk-r/ppa/ubuntu xenial InRelease [17.5 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:4 http://ppa.launchpad.net/openjdk-r/ppa/ubuntu xenial/main amd64 Packages [7096 B]
Get:5 http://security.ubuntu.com/ubuntu xenial-security/universe Sources [42.0 kB]
Get:6 http://security.ubuntu.com/ubuntu xenial-security/main amd64 Packages [380 kB]
Get:7 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Get:8 http://security.ubuntu.com/ubuntu xenial-security/restricted amd64 Packages [12.8 kB]
Get:9 http://security.ubuntu.com/ubuntu xenial-security/universe amd64 Packages [178 kB]
Get:10 http://security.ubuntu.com/ubuntu xenial-security/multiverse amd64 Packages [2931 B]
Get:11 http://archive.ubuntu.com/ubuntu xenial-backports InRelease [102 kB]
Get:12 http://archive.ubuntu.com/ubuntu xenial/universe Sources [9802 kB]
Get:13 http://archive.ubuntu.com/ubuntu xenial/main amd64 Packages [1558 kB]
Get:14 http://archive.ubuntu.com/ubuntu xenial/restricted amd64 Packages [14.1 kB]
Get:15 http://archive.ubuntu.com/ubuntu xenial/universe amd64 Packages [9827 kB]
```

この `docker build ...` コマンドからの簡易出力では、Dockerfile の各行がステップとして現れます。この画面キャプチャには、1つ目と2つ目のステップ（コマンド）が示されています。Docker では、こうしたコマンドの実行が標準出力（`stdout`）にエコーされるので、コマンドの実行を目で確認したり、出力をドキュメント用にキャプチャしたりできます。

イメージはビルドされましたが、まだリポジトリに保存されていません。これが `docker image` なのは、そのためです。Docker では、終盤にイメージ ID が `stdout` に出力されます。これは、イメージが正常に作成され、タグ付けされたかどうかを示すものでもあります。

出力の最後に「`Successfully ...`」と表示されない場合は、Dockerfile にエラーがないかを確認する（場合によっては Dockerfile を簡略化する）か、シンプルな Dockerfile を使って Docker が正常に動作するかを確認してください。

4. イメージが正常に作成されたことを確認します。

```
$ docker images
```

例：

```
[username mydocker]$ docker images
```

REPOSITORY CREATED	TAG	IMAGE ID	
nvcr.io/nvidian_sas/tensorflow_octave	19.03_with_octave	67c448c6fe37	About a minute ago
nvcr.io/nvidian_general/adlr_pytorch	resumes	17f2398a629e	47 hours ago
<none>	<none>	0c0f174e3bbc	9days ago
nvcr.io/nvidian_sas/pushed-hshin	latest	c026c5260844	9days ago
torch-caffe	latest	a5cdc9173d02	11days ago
<none>	<none>	al34a09668a8	2weeks ago
<none>	<none>	0f4ab6d62241	2weeks ago
nvcr.io/nvidia/cuda	10.0-cudnn7-devel-ubuntu16.04	a995ceb5f5782	2weeks ago
mxnet-dec-abcd	latest	8bceaf5e58de	2weeks ago
keras_ae	latest	92ab2bed8348	3weeks ago
nvidia/cuda	latest	614dcdafa05c	3weeks ago
ubuntu	latest	d355ed3537e9	3weeks ago
deeper_photo	latest	f4e395972368	4weeks ago
<none>	<none>	0e8208a5e440	4weeks ago
nvcr.io/nvidia/digits	19.03	c4e87f2alebe	5weeks ago
nvcr.io/nvidia/tensorflow	19.03	56f2980ble37	5weeks ago
mxnet/python	gpu	7e7c9176319c	6weeks ago
nvcr.io/nvidian_sas/chainer	latest	2ea707c58bea	6weeks ago
deep_photo	latest	ef4510510506	7weeks ago
<none>	<none>	9124236672fe	8weeks ago
nvcr.io/nvidia/cuda	10.0-cudnn7-devel-ubuntu18.04	02910409eb5d	8weeks ago
nvcr.io/nvidia/digits	19.03	c14438dc0277	2months ago
nvcr.io/nvidia/tensorflow	19.03	9dda0d5c344f	2months ago
nvcr.io/nvidia/caffe	19.03	87c288427f2d	2months ago
nvcr.io/nvidia/tensorflow	19.03	121558cb5849	3months ago

最初の項目が新しいイメージです（約1分前に作成）。

- イメージをリポジトリにプッシュしてコンテナを作成します。

```
docker push <name of image>
```

例：

```
[username mydocker]$ docker push nvcr.io/nvidian_sas/tensorflow_octave:19.03_with_octave
The push refers to a repository [nvcr.io/nvidian_sas/tensorflow_octave]
1b81f494d27d: Image successfully pushed
023cdba2c5b6: Image successfully pushed
8dd41145979c: Image successfully pushed
7cb16b9b8d56: Image already pushed, skipping
bd5775db0720: Image already pushed, skipping
bc0c86a33aa4: Image already pushed, skipping
cc73913099f7: Image already pushed, skipping
d49f214775fb: Image already pushed, skipping
5d6703088aa0: Image already pushed, skipping
7822424b3bee: Image already pushed, skipping
e999e9a30273: Image already pushed, skipping
e33eae9b4a84: Image already pushed, skipping
4a2ad165539f: Image already pushed, skipping
7efc092a9b04: Image already pushed, skipping
```

```

914009c26729: Image already pushed, skipping
4a7ea614f0c0: Image already pushed, skipping
550043e76f4a: Image already pushed, skipping
9327bc01581d: Image already pushed, skipping
6ceab726bc9c: Image already pushed, skipping
362a53cd605a: Image already pushed, skipping
4b74ed8a0e09: Image already pushed, skipping
1f926986fb96: Image already pushed, skipping
832ac06c43e0: Image already pushed, skipping
4c3abd56389f: Image already pushed, skipping
d8b353eb3025: Image already pushed, skipping
f2e85bc0b7b1: Image already pushed, skipping
fc9e1e5e38f7: Image already pushed, skipping
f39a3f9c4559: Image already pushed, skipping
6a8bf8c8edbd: Image already pushed, skipping
Pushing tag for rev [67c448c6fe37] on [https://nvcro.io/v1/repositories/nvidian_sas/
tensorflow_octave

```

このサンプル コードは、`docker push ...` コマンドでイメージをリポジトリにプッシュしてコンテナを作成した後の内容です。この時点で、<https://ngc.nvidia.com> から NGC コンテナ レジストリにログインし、プロジェクトを表示してコンテナがそこにあるかを確認します。

プロジェクト内にコンテナがない場合は、イメージのタグがリポジトリ内の場所と一致するかを確認してください。何らかの理由でプッシュが失敗した場合は、システムとコンテナ レジストリ (`nvcro.io`) との通信に問題があるかもしれないので、もう一度プッシュを試みてください。

コンテナがリポジトリにあることを確認するには、そのコンテナをサーバーにプルして実行します。テストとして、まず `docker rmi ...` コマンドでイメージを DGX システムから削除します。次に、`docker pull ...` を使ってコンテナをサーバーにプルします。octave プロンプトが表示されたら、Octave がインストールされ、このテストの範囲内で正常に機能することがわかります。

2.5. ファイル システムの使用とマウント

ファイル システムを Docker コンテナの内部にマウントすることは、Docker の基本的な使い方の 1 つです。このようなファイル システムには、フレームワークへの入力データだけでなく、コンテナで実行するコードも格納できます。

Docker コンテナには、独自の内部ファイル システムがあり、ホスト上のそれ以外の部分で使われるファイル システムとは分離されます。



重要: 必要であれば、コンテナのファイル システムに外部からデータをコピーできます。ただし、外部のファイル システムをコンテナにマウントする方が、はるかに簡単です。

外部のファイル システムをマウントするには、`docker run --gpus all --rm -ti ... -v $HOME:$HOME \` コマンドに `-v` オプションを指定して実行します。たとえば、次のコマンドは 2 つのファイル システムをマウントします。

```

$ docker run --gpus all --rm -ti ... -v $HOME:$HOME \
-v /datasets:/digits_data:ro \
...

```

ボリュームを扱う部分を除き、コマンドの大半は割愛しました。このコマンドは、外部のファイル システムにあるユーザーのホーム ディレクトリを、コンテナのホーム ディレクトリ (`-v $HOME:$HOME`) にマウント

します。さらに、ホストの `/datasets` ディレクトリをコンテナ内部の `/digits_data` にマウントします (`-v /datasets:/digits_data:ro`)。



再確認: ユーザーは Docker に対して root 権限を持つため、ほぼすべてのものをホスト システムからコンテナの任意の場所にマウントできます。

このコマンドの場合、ボリューム コマンドは次の形式です。

```
-v <External FS Path>:<Container FS Path>(options) \
```

オプションの前半は、外部ファイル システムへのパスです。これを確実にマウントするため、完全修飾パス (FQP) を使うことをおすすめします。これは、コンテナ内部のマウント ポイント `<Container FS Path>` についても当てはまります。

最後のパスの後に、各種のオプションをカッコ () の中に入れて指定できます。前の例では、2 番目のファイル システムが読み取り専用 (ro) でコンテナ内部にマウントされます。-v オプション用の各種オプションについては、[こちら](#)で詳しい説明されています。

DGX™ システムと Docker コンテナでは、[Overlay2](#) ストレージ ドライバーを使って外部ファイル システムをコンテナのファイル システムにマウントします。Overlay2 はユニオンマウントのファイル システムであり、複数のファイル システムを結び付けることで、すべてのコンテンツがあたかも 1 つのファイル システムに存在するかのように見せます。ファイル システムの積集合ではなく、和集合が作成されます。

第3章 コンテナをプルする

このタスクの概要

NGC コンテナ レジストリからコンテナをプルするには、Docker がインストールされている必要があります。DGX をお使いの場合は、「[Preparing To Use NVIDIA Containers Getting Started Guide](#)」の説明を参照してください。

DGX 以外をお使いの場合は、そのプラットフォームに基づいて、[NVIDIA® GPU Cloud™ \(NGC\) コンテナ レジストリのインストール ドキュメント](#)に従ってください。

また、NGC コンテナ レジストリへのアクセスとログインは、「[NGC Getting Started Guide](#)」の説明に従う必要があります。

NGC Docker コンテナが格納されるリポジトリは4つあります。

nvcr.io/nvidia

ディープラーニングフレームワークのコンテナは、nvcr.io/nvidia/ リポジトリに保存されます。

nvcr.io/hpc

HPC コンテナは、nvcr.io/hpc リポジトリに保存されます。

nvcr.io/nvidia-hpcvis

HPC 可視化コンテナは、nvcr.io/nvidia-hpcvis リポジトリに保存されます。

nvcr.io/partner

パートナー コンテナは、nvcr.io/partner リポジトリに保存されます。現在、パートナー コンテナは、ディープラーニングや機械学習に関連するものが中心ですが、そのタイプのコンテナに限定されるわけではありません。

3.1. 重要な概念

pull コマンドと run コマンドの実行に関連して理解すべき概念をここで説明します。

pull コマンドは次のように記述します。

```
docker pull nvcr.io/nvidia/caffe2:17.10
```

run コマンドは次のように記述します。

```
docker run --gpus all -it --rm -v local_dir:container_dir nvcr.io/nvidia/tensorflow:<xx.xx>
```



メモ: 基本コマンドの `docker run --gpu all` を使うには、システムに Docker 19.03-CE と NVIDIA ランタイム パッケージがインストールされている必要があります。Docker の以前のバージョンで使うコマンドについては、「[NGC コンテナの GPU サポートを有効にする](#)」を参照してください。

両方のコマンドを構成する属性について、個別に説明します。

nvcr.io

コンテナ レジストリの名前。NGC コンテナ レジストリの名前は `nvcr.io` です。

nvidia

ディープラーニング コンテナが保存されるレジストリ空間の名前。NVIDIA が提供するコンテナの場合、レジストリ空間は `nvidia` です。

-it

コンテナを対話モードで実行します。

--rm

実行後にコンテナを削除します。

-v

ディレクトリをマウントします。

local_dir

コンテナ内部でアクセスするホスト システム側のディレクトリまたはファイル（絶対パス）。たとえば、以下のパスの `local_dir` は `/home/jsmith/data/mnist` です。

```
-v /home/jsmith/data/mnist:/data/mnist
```

たとえば、`ls /data/mnist` コマンドでコンテナの内部を表示すると、コンテナの外部から「`ls /home/jsmith/data/mnist`」コマンドを実行したときと同じファイルが見えます。

container_dir

コンテナ内部でのターゲット ディレクトリ。たとえば、以下のコマンドでは `/data/mnist` がターゲット ディレクトリです。

```
-v /home/jsmith/data/mnist:/data/mnist
```

<xx.xx>

コンテナのバージョン。たとえば、`19.01`。

py<x>

Python のバージョン。たとえば、`py3`。

3.2. NGC コンテナ レジストリへのアクセスとプル

前提条件

NGC コンテナ レジストリにアクセスするには、次の前提条件が満たされている必要があります。これらの要件の詳細については、「[NGC Getting Started Guide](#)」を参照してください。

- ▶ NGC コンテナ レジストリ <https://ngc.nvidia.com> でアカウントを作成する。API キーを安全な場所に保管する（後で必要になるため）。アカウントを作成した後は、自社データセンターと同じコマンドで DGX システムのコンテナをプルすることができる。



メモ：NGC コンテナ レジストリには、クライアント コンピューターから Docker コマンドを実行するとアクセスできます。DGX プラットフォームを使わないと NGC コンテナ レジストリにアクセスできないわけではありません。インターネットにアクセスできる Linux コンピューターなら、Docker をインストールして使用できます。サポート対象のプラットフォームについては、<https://docs.nvidia.com/ngc/index.html> を参照してください。

- ▶ NGC アカウントがアクティブになっている。
- ▶ NGC コンテナ レジストリへのアクセスの認証に使う NGC API キーがある。
- ▶ Docker コンテナの実行に必要な権限のあるクライアント コンピューターにログインしている。

NGC アカウントがアクティブになった後で、次の方法のどちらかで NGC コンテナ レジストリにアクセスできます。

- ▶ [Docker CLI を使ってコンテナを NGC コンテナ レジストリからプルする](#)
- ▶ [NGC Web インターフェイスを使ってコンテナをプルする](#)

このタスクの概要

Docker レジストリとは、Docker イメージを保存するサービスです。このサービスは、インターネット上、企業のイントラネット上、またはローカル マシン上に配置できます。たとえば、`nvcr.io` は Docker イメージの NGC コンテナ レジストリが保存される場所です。

すべての `nvcr.io` Docker イメージでは、明示的にコンテナ バージョンのタグを使うことで、latest（最新）タグの使用により発生するタグ付けの問題を回避します。たとえば、イメージにローカルでタグ付けされた「latest」バージョンが、実際にはレジストリ内で別の「latest」バージョンを上書きすることがあります。

手順

1. NGC コンテナ レジストリにログインします。

```
$ docker login nvcr.io
```

2. ユーザー名の入力を求められたら、次のテキストを入力します。

```
$oauthtoken
```

`$oauthtoken` とは、認証にユーザー名とパスワードではなく API キーを使うことを示す特別なユーザー名です。

3. パスワードの入力を求められたら、NGC API キーを入力します。

```
Username: $oauthtoken
Password: k7cqFTUvKKdiwGsPnWnyQFYGn1A1sCIRmlP67Qxa
```



ヒント：API キーを入手したときにクリップボードにコピーしておくと、パスワードの入力時に API キーをコマンド シェルに貼り付けることができます。また、後で必要になる可能性があるため、安全な場所に記録することをおすすめします。

3.2.1. Docker CLI を使ってコンテナを NGC コンテナ レジストリからプルする

前提条件

コンテナをプルするには、次の前提条件が満たされている必要があります。

- ▶ コンテナが保存されるレジストリ空間の読み取り権限がある。
- ▶ 「[NGC コンテナ レジストリへのアクセスとプル](#)」の説明に従って NGC コンテナ レジストリにログインする。アクセス可能な安全な場所に API キーを保存する。
- ▶ 使用するアカウントが docker グループのメンバーで、Docker コマンドの使用が許可されている。



ヒント: NGC コンテナ レジストリ内で使用可能なコンテナを参照するには、Web ブラウザーを使って NGC Web サイトから NGC コンテナ レジストリ アカウントにログインします。

手順

1. 目的のコンテナをレジストリからプルします。たとえば、PyTorch™ 21.02 コンテナをプルするとします。

```
$ docker pull nvcr.io/nvidia/pytorch:21.02-py3
```

2. プルされたことを確認するため、システムにある Docker イメージのリストを表示します。

```
$ docker images
```

次のステップ

コンテナをプルした後で、コンテナ内のジョブを実行すると、科学的なワークロードの実行、ニューラルネットワークのトレーニング、ディープラーニングモデルの展開、AI分析の実行などを行えます。

3.2.2. NGC Web インターフェイスを使ってコンテナをプルする

前提条件

NGC コンテナ レジストリからコンテナをプルするには、「[Preparing To Use NVIDIA Containers Getting Started Guide](#)」の説明に従って、Docker と nvidia-docker2 をあらかじめインストールする必要があります。また、NGC コンテナ レジストリへのアクセスとログインは、「[NGC Getting Started Guide](#)」の説明に従う必要があります。

このタスクの概要

このタスクには、以下の前提条件があります。

1. クラウド インスタンス システムがあり、インターネットに接続されている。
2. そのインスタンスに Docker と nvidia-docker2 がインストールされている。
3. ブラウザーで NGC コンテナ レジストリ (<https://ngc.nvidia.com>) にアクセスでき、NGC アカウントがアクティブになっている。
4. コンテナをクラウド インスタンスにプルしたい。

手順

1. NGC コンテナ レジストリ (<https://ngc.nvidia.com>) にログインします。
2. 左側のナビゲーションで (**Registry**) をクリックします。NGC コンテナ レジストリ ページで、使用が許可されている Docker リポジトリとタグを確認します。
3. リポジトリの1つをクリックすると、そのコンテナ イメージに関する情報と、コンテナの実行時に使えるタグが表示されます。
4. (**Pull**) 列で、アイコンをクリックして `docker pull` コマンドをコピーします。
5. コマンド プロンプトを開き、`docker pull` コマンドを貼り付けます。コンテナ イメージのプルが開始されます。プルが正常に完了したことを確認します。
6. Docker コンテナ ファイルをローカル システムに取得した後で、コンテナをローカル Docker レジストリに読み込みます。
7. イメージがローカル Docker レジストリに読み込まれたことを確認します。

```
$ docker images
```


使用するコンテナの詳細については、コンテナ内の `/workspace/README.md` ファイルを参照してください。

3.3. 検証する

Docker イメージの実行後に、*nix 伝統のオプションである `ps` を使って検証できます。たとえば、`$ docker ps -a` コマンドを実行します。

```
[username ~]$ docker ps -a
CONTAINER ID   IMAGE                                COMMAND                                     CREATED
12a4854ba738   nvcr.io/nvidia/tensorflow:21.02    "/usr/local/bin/nv..."                 35 seconds ago
```

`-a` オプションを使わないと、実行中のインスタンスのみが表示されます。

 **重要:** ハングしたジョブが実行中のまま残っている場合や、パフォーマンスの問題がある場合は、`-a` オプションを指定することをおすすめします。

必要であれば、実行中のコンテナを停止することもできます。以下に、例を示します。

```
[username ~]$ docker ps -a
CONTAINER ID   IMAGE                                COMMAND                                     PORTS      NAMES
12a4854ba738   nvcr.io/nvidia/
tensorflow:21.02    "/usr/local/bin/nv..." 6006/tcp  brave_neumann
```

```
[username ~]$  
[username ~]$ docker stop 12a4854ba738  
12a4854ba738  
[username ~]$ docker ps -a  
CONTAINER ID    IMAGE                                COMMAND                                CREATED        NAMES
```

停止するイメージの Container ID が必要なことに注意してください。この ID を確認するには、`$ docker ps -a` コマンドを使用します。

もう1つの便利なコマンド、つまり「Docker オプション」は、イメージをサーバーから除外するコマンドです。イメージを除外または削除すると、サーバーのスペースが節約されます。たとえば、次のコマンドを実行します。

```
$ docker rmi nvcr.io/nvidia.tensorflow:21.02
```

サーバーで `$ docker images` を実行してイメージの一覧を表示すると、イメージが削除されたことがわかります。

第 4 章 NGC イメージ

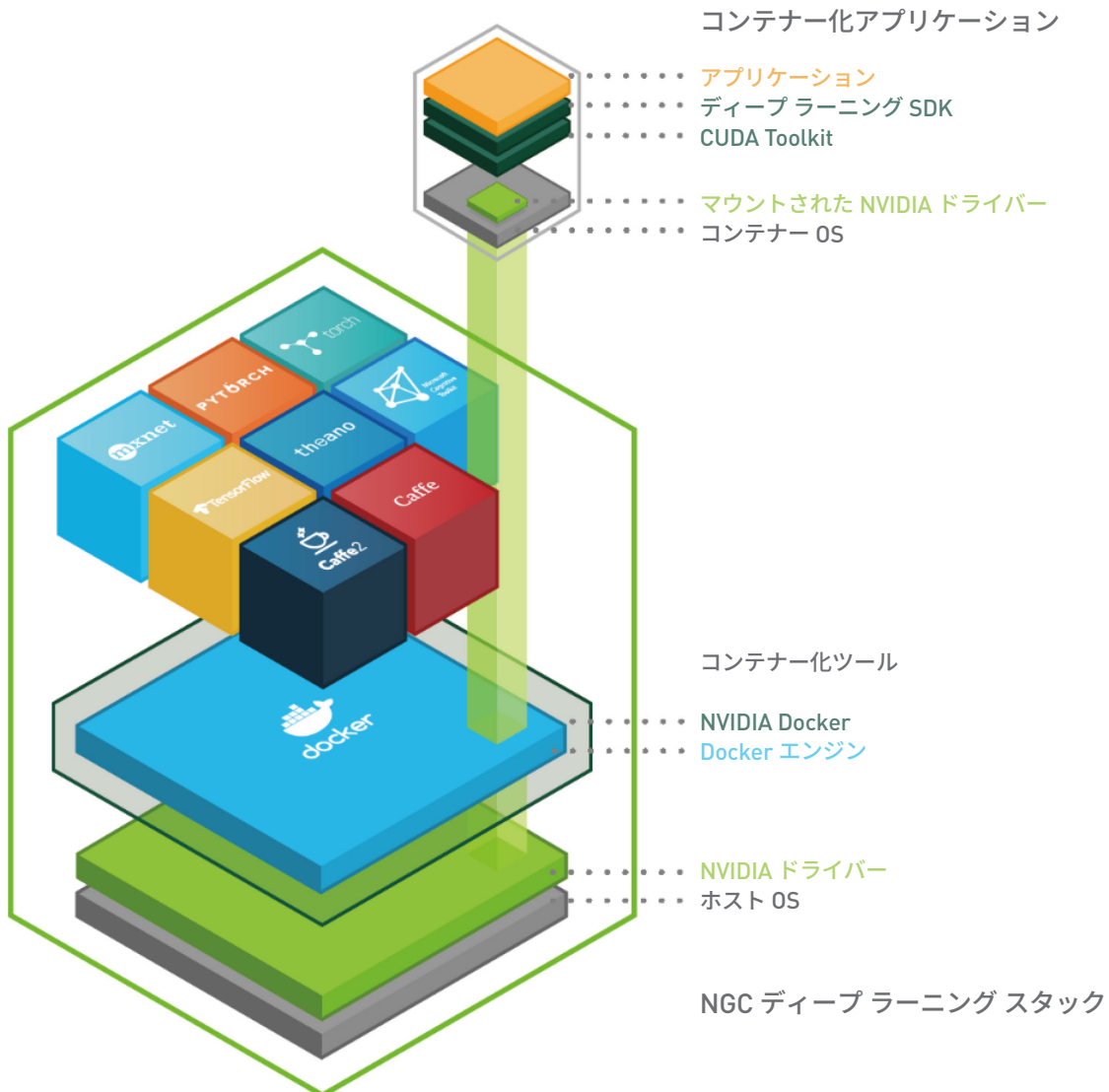
NGC コンテナは、`nvcr.io` というリポジトリでホストされます。前のセクションでも説明したとおり、このリポジトリからコンテナを「プル」し、科学的なワークロード、可視化、ディープラーニングなどの GPU アクセラレーテッド アプリケーションの実行に使用できます。

Docker イメージとは、要するに、開発者がビルドするファイルシステムです。レイヤーは、スタック内ですぐ下にあるレイヤーに依存します。

Docker イメージが「実行」、つまりインスタンス化されると、コンテナが作成されます。コンテナの作成時に、書き込み可能なレイヤーがスタックの最上部に追加されます。書き込み可能なコンテナ レイヤーが追加された Docker イメージが、コンテナです。コンテナとは、このイメージの実行中インスタンスです。コンテナへの変更と修正は、書き込み可能なレイヤーに対して行われます。コンテナを削除することはできますが、Docker イメージはそのまま残ります。

[図 1](#) に、DGX システム ファミリのスタックを図示します。このように、NVIDIA Container Toolkit はホスト OS と NVIDIA ドライバーの上に位置します。このツール群は、NVIDIA コンテナの作成、管理、操作に使うもので、`nvidia-docker` レイヤーの上にあるレイヤーです。これらのコンテナには、アプリケーション、ディープラーニング SDK、CUDA Toolkit が含まれます。NVIDIA コンテナ化ツールによって、適切な NVIDIA ドライバーがマウントされます。

図 1. Docker コンテナは、アプリケーションの依存関係を 1 つにまとめることで、アプリケーションの実行に再現可能性と信頼性をもたらす。nvidia-docker ユーティリティは、NVIDIA ドライバーのユーザー モード コンポーネントと GPU を起動時に Docker コンテナにマウントする。



4.1. NGC イメージのバージョン

Docker イメージのリリースは、バージョン「タグ」で区別されます。シンプルなイメージであれば、このバージョン タグには通常、イメージに含まれるメインのソフトウェア パッケージのバージョンが設定されます。複数のソフトウェア パッケージやバージョンが含まれる複雑なイメージであれば、コンテナ化されたソフトウェアの構成を表す個別のバージョンが設定されることがあります。よく使われるのは、イメージがリリースされた年と月を組み合わせたタグです。たとえば、イメージの 21.02 リリースは、2021 年 2 月にリリースされたことを意味します。

イメージ名は、コロンで区切られた 2 つの部分で構成されます。1 つ目の部分はリポジトリ内のコンテナの名前で、2 つ目の部分はコンテナに関連付けられた「タグ」です。この 2 種類の情報を図 2 に示します。ここでは、`docker images` コマンドを実行したときの出力を使用しました。

図 2. docker images コマンドの出力

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nvidia/cuda	8.0-devel	5094464ddfe8	2 weeks ago	1.62 GB
ubuntu	latest	f49eec89601e	2 weeks ago	129 MB
nvcr.io/nvidia/tensorflow	17.01	4352527009ae	2 weeks ago	2.77 GB

Image Name = Repository:Tag ImageID = Unique Hash

図 2 は、次のようなシンプルなイメージ名の例です。

- ▶ nvidia-cuda:8.0-devel
- ▶ ubuntu:latest
- ▶ nvcr.io/nvidia/tensorflow:21.01

イメージにタグを追加しないことにした場合は、デフォルトで「latest」がタグとして追加されますが、NGC コンテナでは常に明示的なバージョン タグが与えられます。

次のセクションでは、こうしたイメージ名をコンテナの実行に使う方法を説明します。また、その後のセクションでは、独自のコンテナを作成する方法や、既存のコンテナをカスタマイズして拡張する方法も説明します。

第5章 コンテナを実行する

前提条件

NGC ディープ ラーニング フレームワークを実行するには、Docker 環境で NVIDIA GPU がサポートされている必要があります。コンテナを実行するには、この章の説明に沿って、適切なコマンドにレジストリ、リポジトリ、タグを指定して実行します。

5.1. 使用例：コンテナを実行する

手順

1. ユーザーとして、コンテナを対話形式で実行します。

```
$ docker run --gpus all -it --rm -v local_dir:container_dir  
nvcr.io/nvidia/<repository>:<xx.xx>
```

使用例：次の例では、NVIDIA PyTorch コンテナの 2021 年 2 月リリース (21.02) を対話モードで実行します。ユーザーがコンテナを終了すると、そのコンテナは自動的に削除されます。

```
$ docker run --gpus all --rm -ti nvcr.io/nvidia/pytorch:21.02-py3  
  
===== NVIDIA PyTorch =====  
===== NVIDIA PyTorch =====  
NVIDIA Release 21.02 (build 11032)  
  
Container image Copyright (c) 2021, NVIDIA CORPORATION. All rights reserved.  
Copyright (c) 2014 - 2019, The Regents of the University of California (Regents)  
All rights reserved.  
  
Various files include modifications (c) NVIDIA CORPORATION. All rights reserved.  
NVIDIA modifications are covered by the license terms that apply to the underlying  
project or file.  
root@df57eb8e0100:/workspace#
```

2. コンテナ内で、実行したいジョブを開始します。正確なコマンドは、実行しているコンテナにあるディープ ラーニング フレームワークと、実行するジョブによって異なります。コンテナの詳細については、`/workspace/README.md` ファイルを参照してください。

使用例：次の例では、`pytorch time` コマンドを 1 つの GPU に実行して、`deploy.prototxt` モデルの実行時間を測定します。


```
# pytorch time -model models/bvlc_alexnet/ -solver deploy.prototxt -gpu=0
```

3. **オプション**: 同じ NVIDIA PyTorch コンテナの 2021 年 2 月リリース (21.02) を、対話ではないモードでも実行してみます。

```
% docker run --gpus all -it --rm -v local_dir:container_dir nvcr.io/nvidia/  
pytorch:<xx.xx>-py3 <command>
```

5.2. ユーザーを指定する

特に指定しない限り、コンテナ内のユーザーは root ユーザーです。

コンテナ内で実行する場合、ホスト オペレーティング システムまたはネットワーク ボリュームに作成されたファイルには root ユーザーがアクセスできます。このようなアクセスを許可しない場合は、コンテナ内でユーザー ID を設定する必要があります。たとえば、コンテナ内のユーザーを現在実行中のユーザーに設定するには、次のコマンドを実行します。

```
% docker run --gpus all -ti --rm -u $(id -u):$(id -g) nvcr.io/nvidia/<repository>:  
<container version>
```

通常、指定したユーザーとグループがコンテナ内に存在しないため、警告が表示されます。これは次のようなメッセージです。

```
groups: cannot find name for group ID 1000I have no name! @c177b61e5a93:/workspace$
```

通常は、この警告を無視してかまいません。

5.3. 削除フラグを設定する

デフォルトで、Docker コンテナは実行後もシステムに残ります。プルと実行を繰り返すと、コンテナが終了した後も、ローカル ディスクのスペースが次第に減っていきます。このため、終了後にコンテナを消去する必要があります。

メモ: 残したい変更をコンテナに加えた場合や、実行の終了後にジョブのログにアクセスする場合は、`--rm` フラグを使わないでください。

終了後にコンテナを自動的に削除するには、`run` コマンドに `--rm` フラグを追加します。

```
% docker run --gpus all --rm nvcr.io/nvidia/<repository>:<container version>
```

5.4. 対話フラグを設定する

デフォルトで、コンテナはバッチ モードで実行されます。つまり、コンテナが終了するまで、実行中にユーザーとの対話が一切ありません。コンテナは、サービスとして対話モードで実行することもできます。

対話モードを実行するには、`run` コマンドに `-ti` フラグを追加します。

```
% docker run --gpus all -ti --rm nvcr.io/nvidia/<repository>:<container version>
```

5.5. ボリューム フラグを設定する

コンテナにデータセットは含まれないので、データセットを使うには、ホスト オペレーティング システムからボリュームをコンテナにマウントする必要があります。詳細については、「[Volumes](#)」を参照してください。

Docker ボリュームまたはホスト データ ボリュームを使うのが一般的です。ホスト データ ボリュームと Docker ボリュームの最大の違いは、Docker ボリュームは Docker のプライベートなボリュームであり、Docker コンテナのみで共有されることです。Docker ボリュームはホスト オペレーティング システムから見え、そのデータ ストレージは Docker で管理されます。ホスト データ ボリュームは、ホスト オペレーティング システムで使用できる任意のディレクトリです。ローカル ディスクやネットワーク ボリュームをホスト データ ボリュームとして使うことができます。

使用例 1

ホスト オペレーティング システムから /raid/imagdata ディレクトリをコンテナに /images としてマウントします。

```
% docker run --gpus all -ti --rm -v /raid/imagdata:/images nvcr.io/nvidia/
<repository>:<container version>
```

使用例 2

data という名前のローカル Docker ボリューム（なければ必ず事前に作成する）をコンテナ内に /imagdata としてマウントします。

```
% docker run --gpus all -ti --rm -v data:/imagdata nvcr.io/
nvidia/<repository>:<container version>
```

5.6. ポート マッピング フラグを設定する

Deep Learning GPU Training System™ (DIGITS) などのアプリケーションでは、通信するためにポートが開かれます。このポートをローカル システムにのみ開くのか、それともローカル システムの外部にあるネットワーク上で他のコンピューターにもポートへのアクセスを許可するのかを制御できます。

たとえば DIGITS なら、コンテナ イメージ 16.12 で起動された DIGITS 5.0 では、デフォルトで DIGITS サーバーがポート 5000 で開かれます。ただし、コンテナの起動後に、そのコンテナの IP アドレスを確認するのは簡単ではないかもしれません。コンテナの IP アドレスは、次のいずれかの方法で確認できます。

- ▶ ローカル システム ネットワーク スタックを使ってポートを表示する（`--net=host`）。この場合、コンテナのポート 5000 が、ローカル システムのポート 5000 として使用可能となっている。

または、

- ▶ ポートをマッピングする（`-p 8080:5000`）。この場合、コンテナのポート 5000 が、ローカル システムのポート 8080 として使用可能となっている。

どちらの方法でも、ローカル システムの外側にいるユーザーには、DIGITS がコンテナの中で実行されていることはわかりません。ポートを公開しない場合、そのポートはホストから引き続き使用できますが、外部からは使用できません。

5.7. 共有メモリ フラグを設定する

PyTorch™などの一部のアプリケーションでは、プロセス間通信に共有メモリ バッファが使用されます。共有メモリは、Apache MXNet™とTensorFlow™を基盤とする NVIDIA の最適化ディープラーニング フレームワークのようなシングル プロセスのアプリケーションでも必要な場合があります。このフレームワークでは、NVIDIA® Collective Communications Library™ (NCCL) ライブラリが使用されます。

デフォルトで、Docker コンテナには 64MB の共有メモリが割り当てられますが、それでは足りないことがあります。特に、8 つの GPU をすべて使う場合は足りません。共有メモリの上限を特定のサイズ、たとえば 1GB に増やすには、`--shm-size=1g` フラグを `docker run` コマンドに追加します。

または、`--ipc=host` フラグを指定して、ホストの共有メモリ スペースをコンテナ内で再利用します。この方法を使う場合、共有メモリ バッファにあるデータが他のコンテナにも見える可能性があるため、セキュリティ上の懸念があります。

5.8. GPU 公開制限フラグを設定する

コンテナの内部では、スクリプトやソフトウェアは使用可能なすべての GPU をフルに活用しようとします。GPU の使用量をおおまかに調整するには、このフラグを使って、ホストからコンテナへの GPU の公開を制限します。たとえば、GPU0 と GPU1 のみをコンテナに公開するには、次のコマンドを実行します。

```
$ docker run --gpus "device=0,1" ...
```

指定した GPU は、Docker のデバイスマッピング機能によりコンテナごとに定義されます。現在、これには Linux の `cgroups` 機能が利用されています。

5.9. コンテナの有効期間

`--rm` フラグを `docker run --gpus` コマンドに指定しない場合、終了したコンテナの状態は永久に保持されます。終了した後も保存されているすべてのコンテナとそれがディスクに占めるサイズは、次のコマンドで表示できます。

```
$ docker ps --all --size --filter Status=exited
```

ディスク上のコンテナのサイズは、実行中に作成されたファイルによって異なりますが、終了したコンテナがディスクに占めるスペースは少量です。

終了したコンテナを永続的に削除するには、次のコマンドを実行します。

```
docker rm [CONTAINER ID]
```

コンテナの終了後に状態を保存すると、標準の Docker コマンドを使ってそれらの状態を操作できます。以下に、例を示します。

- ▶ 過去の実行に関するログを調べるには、`docker logs` コマンドを実行します。

```
$ docker logs 9489d47a054e
```

- ▶ このファイルは、`docker cp` コマンドを使って抽出できます。

```
$ docker cp 9489d47a054e:/log.txt .
```

- ▶ 停止したコンテナは、`docker restart` コマンドを使って再開できます。

```
$ docker restart <container name>
```

PyTorch コンテナの場合は、次のコマンドを実行します。

```
$ docker restart pytorch
```

- ▶ 変更を保存するには、`docker commit` コマンドを使って新しいイメージを作成します。詳細については、「[使用例 3：docker commit](#)」を使用してコンテナをカスタマイズする」を参照してください。



メモ： Docker コンテナの変更をコミットする際は、コンテナの使用中に作成されたデータ ファイルを含めてイメージが生成されることに注意してください。コア ダンプ ファイルとログは特にサイズが大きいため、これらによってイメージのサイズが大幅に増える可能性があります。

第 6 章 NVIDIA ディープ ラーニング ソフトウェア スタック

[NVIDIA Deep Learning Software Developer Kit \(SDK\)](#) には、DGX システム用の NVIDIA レジストリ領域に存在するものすべて (CUDA Toolkit、DIGITS、すべてのディープ ラーニング フレームワークなど) が含まれます。

NVIDIA Deep Learning SDK は、Apache MXNet、PyTorch、TensorFlow を基盤とする NVIDIA の最適化ディープ ラーニング フレームワークのような、一般的なディープ ラーニング フレームワークを高速化します。



メモ:18.09 コンテナのリリース以降、Caffe2、Microsoft Cognitive Toolkit、Theano™、Torch™の各フレームワークはコンテナ イメージ内で提供されません。

ソフトウェア スタックは、システムに最適化されたコンテナ バージョンとして、これらのフレームワークを提供します。これらのフレームワークは、すべての依存関係を含め、事前にビルド、テスト、調整が行われており、すぐに実行できる状態にあります。カスタムのディープ ラーニング ソリューションを柔軟にビルドしたいユーザーのために、各フレームワーク コンテナ イメージには、フレームワークのソース コードと完全なソフトウェア開発スタックも含まれているので、これを利用して独自の変更や強化を加えることができます。

このプラットフォーム ソフトウェアの設計の中心にあるのは、サーバーにインストールされた最小構成の OS とドライバー、そして、DGX システムでの NGC コンテナ レジストリによる、コンテナ内のアプリケーションと SDK ソフトウェアのプロビジョニングです。

すべての NGC コンテナ イメージは、プラットフォーム レイヤー (nvcr.io/nvidia/cuda) を基盤とします。このイメージは、他のすべての NGC コンテナを下支えするソフトウェア開発スタックのコンテナ化バージョンとなるものです。カスタム アプリケーションを追加してコンテナを柔軟にビルドしたいユーザーは、このイメージを使用できます。

6.1. OS レイヤー

ソフトウェア スタックの最下位レイヤー (基本レイヤー) が、OS のユーザー空間です。このレイヤーには、リリース月に使用可能となったすべてのセキュリティ パッチなどのソフトウェアが含まれます。

6.2. CUDA レイヤー

CUDA® は、NVIDIA が開発した並列コンピューティングのプラットフォームであり、プログラミング モデルです。アプリケーション開発者は、CUDA を使って GPU の強力な並列処理能力を利用できます。CUDA は、ディープ ラーニングだけでなく、天文学や分子動力学シミュレーション、金融工学など、演算能力とメモリを集中的

に消費する多様なアプリケーションを GPU で高速化するための基盤です。CUDA の詳細については、[CUDA 関連ドキュメント](#)を参照してください。

6.2.1. CUDA ランタイム

CUDA ランタイム レイヤーは、CUDA アプリケーションを展開環境で実行するために必要なコンポーネントを提供します。CUDA ランタイムは CUDA Toolkit と共にパッケージ化され、すべての共有ライブラリを含みますが、CUDA コンパイラー コンポーネントは含みません。

6.2.2. CUDA Toolkit

CUDA Toolkit は、最適化された GPU アクセラレーテッド アプリケーションを開発するための開発環境を提供します。CUDA Toolkit を使うと、GPU アクセラレーテッドの組み込みシステム、デスクトップワークステーション、エンタープライズ データセンター、クラウド向けにアプリケーションの開発、最適化、展開を行えます。CUDA Toolkit には、ライブラリ、デバッグ ツールと最適化ツール、アプリケーションを展開するためのコンパイラーとランタイム ライブラリが含まれます。

次のライブラリは、ディープ ニューラル ネットワーク向けの GPU アクセラレーテッド プリミティブを提供します。

CUDA[®] Basic Linear Algebra Subroutines library™ (cuBLAS) cuBLAS は、完全な標準 BLAS ライブラリを GPU 向けに高速化したバージョンです。GPU での実行が著しく高速化されています。cuBLAS 行列乗算 (GEMM) ルーチンは、ディープ ニューラル ネットワークにおいて、完全に接続されたレイヤーの演算などに使われる重要な演算法です。cuBLAS の詳細については、[cuBLAS 関連ドキュメント](#)を参照してください。

6.3. ディープラーニング ライブラリ レイヤー

次のライブラリは、NVIDIA の GPU でディープ ラーニングを実行するのに不可欠です。NVIDIA Deep Learning Software Development Kit (SDK) の一部として提供されます。

6.3.1. NCCL

NVIDIA[®] Collective Communications Library™ (NCCL) (発音は「ニッケル」) は、アプリケーションに簡単に統合できる、トポロジ対応型のマルチ GPU 集合通信プリミティブのライブラリです。

集合通信アルゴリズムは、多数のプロセッサをいっせいに動かしてデータを集計します。NCCL は汎用の並列プログラミング フレームワークではなく、集合通信プリミティブの高速化に特化したライブラリです。現在、次の集合操作がサポートされています。

- ▶ AllReduce
- ▶ Broadcast
- ▶ Reduce
- ▶ AllGather
- ▶ ReduceScatter

プロセッサが正確に同期して通信することが、集合通信の大きな特徴です。CUDA ベースの集合は、ローカルリダクションを実現するための CUDA メモリ コピー操作と CUDA カーネルの組み合わせだというのが、従来

の認識でした。これとは異なり、NCCL は各集合体を 1 つのカーネルに実装して、通信と演算の両方を処理します。こうすることで、迅速な同期が可能になり、帯域幅消費のピーク時に必要なリソースを最小限に減らすことができます。

NCCL があれば、開発者はアプリケーションを特定のマシン向けに最適化する必要はありません。NCCL は、ノード内部とノード間の両方で、複数の GPU に高速の集合を提供します。PCIe、NVLink™、InfiniBand Verbs、IP ソケットなど、さまざまな相互接続テクノロジーがサポートされています。また、NCCL では、システムが基盤とする GPU 相互接続トポロジに合わせて通信戦略のパターンが自動的に調整されます。

NCCL の設計においてパフォーマンスの次に重視されたのが、プログラミングを容易にすることでした。NCCL ではシンプルな C API を使うので、さまざまなプログラミング言語から簡単に利用できます。NCCL は、MPI (Message Passing Interface) で定義され、広く使われている集合 API に忠実に従います。MPI の経験があれば、NCCL の API はすぐに使えるでしょう。NCCL の集合は、MPI の定義とはやや違い、CUDA プログラミング モデルと直接連携するために「stream」引数を受け取ります。また、NCCL は事実上すべてのマルチ GPU 並列モデルと互換性があり、以下のような構成で使用できます。

- ▶ シングルスレッド
- ▶ マルチスレッド (GPU ごとに 1 スレッドなど)
- ▶ マルチプロセス (GPU 上でのマルチスレッド演算と結び付けられた MPI)

NCCL はディープラーニングフレームワークに適していることがわかっています。AllReduce の集合が、ニューラルネットワークのトレーニングに盛んに使われています。NCCL が提供するマルチモード通信をマルチ GPU で使うと、ニューラルネットワークのトレーニングを柔軟にスケールできます。

NCCL の詳細については、[NCCL 関連ドキュメント](#)を参照してください。

6.3.2. cuDNN レイヤー

CUDA® Deep Neural Network library™ (cuDNN) は、順畳み込みと逆畳み込み、プーリング、正規化、アクティベーションのレイヤーなどの標準ルーチン向けに高度に調整された実装を提供します。

フレームワーク開発の足並みが揃わないこと、そして、cuDNN ライブラリに後方互換性がないことから、cuDNN は単独でコンテナに収めるしかありません。つまり、使用できる CUDA と cuDNN コンテナが複数あることとなりますが、各コンテナには固有のタグがあるので、フレームワークでは Dockerfile にこのタグを指定する必要があります。

cuDNN の詳細については、[cuDNN 関連ドキュメント](#)を参照してください。

6.4. フレームワーク コンテナ

フレームワーク レイヤーには、特定のディープラーニングフレームワークに必要なものがすべて揃っています。このレイヤーの目的は、基本的なワーキングフレームワークを提供することです。このフレームワークは、プラットフォーム コンテナ レイヤーで細かくカスタマイズできます。

フレームワーク レイヤーでは、次の操作を行えます。

- ▶ NVIDIA から提供されたフレームワークをそのまま実行する。フレームワークはビルド済みで、コンテナイメージの内部で実行できる状態にある。
- ▶ NVIDIA から提供されたフレームワークをやや変更する。NVIDIA のコンテナ イメージに変更を加えてから、コンテナ内部で再コンパイルする。

- ▶ NVIDIA から提供された CUDA、cuDNN、NCCL レイヤーの上で実行したいアプリケーションをゼロからビルドする。

次のセクションでは、NVIDIA ディープラーニング フレームワーク コンテナについて説明します。

フレームワークの詳細については、[フレームワーク関連ドキュメント](#)を参照してください。

第7章 NVIDIA ディープラーニング フレームワーク コンテナ

ディープラーニングフレームワークは、複数のレイヤーで構成されるソフトウェアスタックの一部です。レイヤーは、スタック内ですぐ下にあるレイヤーに依存します。このようなソフトウェアアーキテクチャには、多くのメリットがあります。

- ▶ 各ディープラーニングフレームワークは個別のコンテナに収められるので、フレームワークによって異なるバージョンでC標準ライブラリ (libc) や cuDNN などのライブラリを使用することができ、互いに干渉しません。
- ▶ レイヤー化コンテナを使う最大の理由は、ユーザーが求めているエクスペリエンスを提供できることです。
- ▶ ディープラーニングフレームワークがパフォーマンス向上やバグ修正のために更新されると、コンテナの新しいバージョンがレジストリで使える状態になります。
- ▶ システムの保守は簡単で、アプリケーションがOSに直接インストールされないことから、OSイメージはクリーンに保たれます。
- ▶ セキュリティアップデート、ドライバーアップデート、OSパッチをシームレスに提供できます。

以降のセクションでは、nvr.ioのフレームワークコンテナについて説明します。

7.1. DL/ML ソフトウェアフレームワークを使う理由

フレームワークは、ディープラーニングの研究と応用をより身近で効率的なものにするために作られました。フレームワークには、以下のようなメリットがあります。

- ▶ フレームワークは、ディープニューラルネットワーク (DNN) のトレーニングに必要な演算に高度に最適化されたGPU対応コードを提供する。
- ▶ NVIDIAのフレームワークは、GPUのパフォーマンスをできるだけ強化するために調整され、テストされている。
- ▶ フレームワークを使うと、シンプルなコマンドラインまたはPythonなどのスクリプト言語インターフェイスを使ってコードにアクセスできる。
- ▶ 多くの強力なDNNが、このようなフレームワークを使ってトレーニングされ、展開されている。GPUコードのような複雑なコンパイルコードを書かなくても、GPUアクセラレーションでトレーニングを高速化できる。

7.2. Kaldi

Kaldi Speech Recognition Toolkit プロジェクトは、2009年にジョンズ ホプキンス大学で発足しました。その目的は、音声認識システムの構築に必要なコストと時間を削減することでした。当初は新しい言語とドメインに ASR サポートを提供することが主な活動でしたが、着実に規模と能力を拡大し、数百人の研究者がこの分野の進歩に参加するようになりました。Kaldi は現在、事実上の標準の音声認識ツールキットとなり、毎日数百万人が利用する音声サービスの提供に役立っています。

Kaldi の最適化と変更に関する最新情報は、「[Deep Learning Frameworks Release Notes](#)」で確認できます。

7.3. Apache MXNet を基盤とする NVIDIA の最適化ディープラーニング

フレームワーク Apache MXNet を基盤とする NVIDIA の最適化ディープラーニング フレームワークは、効率と柔軟性を重視して設計されたディープラーニングフレームワークです。シンボリックプログラミングと命令型プログラミングを混在させて、効率と生産性を最大限に高めることができます。MXNet は、Apache Incubator プロジェクトの一部です。MXNet ライブラリは移植性があり、複数の GPU とマシンにスケールリングできます。MXNet は、AWS や Azure など、大手のパブリッククラウドプロバイダーでサポートされ、Amazon は AWS で利用するディープラーニングフレームワークとして MXNet を選びました。MXNet は、C++、Julia、MATLAB、JavaScript、Go、R、Scala、Perl、Wolfram Language などの言語をサポートしています。

Python Apache MXNet を基盤とする NVIDIA の最適化ディープラーニングフレームワークの中核にあるのは、シンボリック演算と命令型演算をその場で自動的に並列処理する動的な依存関係スケジューラーです。スケジューラーの上にはグラフ最適化レイヤーがあり、ここでシンボリックの実行が高速化され、メモリの使用が効率化されます。Apache MXNet を基盤とする NVIDIA の最適化ディープラーニングフレームワークは軽量で移植性があり、複数の GPU とマシンにスケールリングされます。

Apache MXNet を基盤とする NVIDIA の最適化ディープラーニングフレームワークに加えられた最適化と変更について、「[Deep Learning Frameworks Release Notes](#)」に詳しく説明されています。

7.4. TensorFlow

TensorFlow™ は、データフローグラフを使って数値計算を行うためのオープンソースのソフトウェアライブラリです。グラフ内のノードが数値演算を表し、グラフのエッジがそれらの間を流れる多次元データ配列（テンソル）を表します。このようにアーキテクチャが柔軟なので、コードを書き直さなくても、デスクトップ、サーバー、またはモバイルデバイスで1つ以上の CPU または GPU に計算を展開できます。

TensorFlow は、Google の機械知能研究組織の Google Brain チームで活動する研究者とエンジニアによって、機械学習とディープニューラルネットワークの研究を深めるために開発されました。それらに限らず、幅広い分野に応用できるシステムです。

TensorFlow の結果を可視化するため、この特定の Docker イメージには TensorBoard も含まれています。TensorBoard は可視化ツールのスイートです。たとえば、トレーニングの履歴やモデルの外観を表示できます。

TensorFlow の最適化と変更に関する最新情報は、「[Deep Learning Frameworks Release Notes](#)」で確認できます。

7.4.1. TensorFlow コンテナを実行する

[TensorFlow](#) を GPU システムで効率よく実行するには、TensorFlow Docker コンテナを使ってコードを実行する起動スクリプトを設定します。

TensorFlow をスクリプトで実行したい場合は、「[Scripts Best Practices](#)」セクションの `run_tf_cifar10.sh` スクリプトを参照してください。これはシステム上で実行できる bash スクリプトです。Docker コンテナを `nvcr.io` リポジトリからシステムにプルしたと仮定します。また、CIFAR-10 データをシステム上の `/datasets/cifar` に保存し、それをコンテナ内の `/datasets/cifar` にマッピングしたと仮定します。スクリプトに次のような引数を渡すこともできます。

```
$. /run_tf_cifar10.sh --data_dir=/datasets/cifar --num_gpus=8
```

`run_tf_cifar10.sh` スクリプトで使うパラメーターの詳細については、このドキュメントの Keras セクション（「[Keras とコンテナ化フレームワーク](#)」）を参照してください。スクリプトの `/datasets/cifar` パスは、サイトの CIFAR データがある場所に変更できます。TensorFlow 用の CIFAR-10 データセットを入手できない場合は、スクリプトを書き込み可能なボリューム `-v /datasets/cifar:/datasets/cifar (ro を含めない)` で実行すると、初回の実行で、データセットが自動的にダウンロードされます。

CIFAR-10 のトレーニングを並列処理したい場合は、Keras を使う TensorFlow の基本的なデータ並列処理も実行できます。GitHub 上で [cifar10_cnn_mgpu.py](#) の使用例を参照してください。

Python スクリプトを Docker コンテナに結合する方法については、「[run_tf_cifar10.sh](#)」スクリプトの説明を参照してください。

7.5. PyTorch

[PyTorch](#) は、Python をフロントエンドとする、GPU アクセラレーテッド テンソル計算フレームワークです。[PyTorch](#) は、設計段階で Python と固く統合されています。[NumPy](#)、[SciPy](#)、[scikit-learn](#)、またはその他の Python エクステンションを使う場合は、PyTorch を使うのが自然です。[Cython](#) や [Numba](#) のようなライブラリを使って、ニューラル ネットワーク レイヤーを Python で書くこともできます。NVIDIA の [cuDNN](#) や [NCCL](#) のような高速化ライブラリのほかに、[Intel の MKL](#) が、パフォーマンスの最大化のために含まれています。

また、PyTorch には、標準の定義済みニューラル ネットワーク レイヤー、ディープ ラーニング オプティマイザー、データ読み込みユーティリティ、マルチ GPU とマルチノードのサポートも含まれています。関数がただちに実行され、静的なグラフにキューイングされないため、使いやすさが向上し、高度なデバッグも可能になります。

PyTorch の最適化と変更に関する最新情報は、「[Deep Learning Frameworks Release Notes](#)」で確認できます。

7.6. DIGITS

[Deep Learning GPU Training System™ \(DIGITS\)](#) は、ディープ ラーニングをエンジニアやデータサイエンティストが活用できるようにするものです。

DIGITS は、NVIDIA が提供している人気のトレーニング ワークフロー マネージャーであり、NVCaffe、Torch、または TensorFlow フレームワーク向けの簡単な Web インターフェイスを使って、イメージ データセットの管理とトレーニングを行えます。

DIGITS はフレームワークではなく、ラッパーとして NVCaffe、Torch、または TensorFlow 向けにグラフィカ

ル Web インターフェイスを提供します。コマンドラインで直接フレームワークを操作する必要はありません。

DIGITS を利用すれば、DNN をトレーニングして、イメージの分類、セグメント化、オブジェクト検出タスクの精度を短時間で高めることができます。DIGITS は、一般的なディープラーニング タスクを簡素化するので、データ管理、マルチ GPU システム向けニューラル ネットワークの設計とトレーニング、高度な可視化によるリアルタイムのパフォーマンス監視、結果ブラウザからの最適な展開実行モデルの選択などを簡単に行えます。完全な対話形式のツールなので、データサイエンティストはニューラル ネットワークのプログラミングやデバッグではなく、設計とトレーニングに専念できます。

DIGITS の最適化と変更に関する最新情報は、「[DIGITS Release Notes](#)」で確認できます。

7.6.1. DIGITS をセットアップする

DIGITS コンテナを実行する場合、次のディレクトリ、ファイル、ポートがあると便利です。

表 2. DIGITS コンテナ実行時の詳細

説明	値	メモ
DIGITS ワーキング ディレクトリ	\$HOME/digits_workdir	作成する必要があるディレクトリ
DIGITS ジョブ ディレクトリ	\$HOME/digits_workdir/jobs	作成する必要があるディレクトリ
DIGITS 構成ファイル	\$HOME/digits_workdir/ digits_config_env.sh	ジョブ ディレクトリとログ ファイルを渡すためのファイル
DIGITS ポート	5000	マルチユーザー環境では一意のポートを選ぶ

重要：一連の環境変数を 1 つのファイルにまとめて指定し、それを `docker run --gpus` コマンドに `--env-file` オプションで渡すことをおすすめします。

`digits_config_env.sh` スクリプトは、DIGITS のジョブ ディレクトリとログ ファイルの場所を宣言します。DIGITS を実行するときは、このスクリプトがよく使われます。

```
# DIGITS Configuration File
DIGITS_JOB_DIR=$HOME/digits_workdir/jobs
DIGITS_LOGFILE_FILENAME=$HOME/digits_workdir/digits.log
```

これらの 2 つの変数をシンプルな bash スクリプトに定義したのが、次の使用例です。DIGITS の構成については、「[Configuration.md](#)」を参照してください。

7.6.2. DIGITS を実行する

DIGITS を実行するには、`run_digits.sh` スクリプトを使用します。ただし、DIGITS をコマンド ラインから実行したい場合は、DIGITS を効率よく実行するのに必要な詳細の大半が含まれるシンプルなコマンドを使用できます。


メモ：まだ jobs ディレクトリがない場合は、作成する必要があります。

```
$ mkdir -p $HOME/digits_workdir/jobs

$ docker run --gpus all --rm -ti --name=${USER}_digits -p 5000:5000 \
-u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
--env-file=${HOME}/digits_workdir/digits_config_env.sh \
-v /datasets:/digits_data:ro \
--shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 \
nvcr.io/nvidia/digits:17.05
```

このコマンドでは、オプションの一部を使う必要があるでしょうが、すべてを使う必要はありません。以下の表に、各パラメーターと説明を記載します。

表 3. docker run --gpus コマンドのオプション

パラメーター	説明
--name	Docker コンテナ インスタンスに関連付ける名前。
--rm	終了後にコンテナ インスタンスを削除することを Docker に指示する。
-ti	対話モードで実行し、tty をインスタンスに関連付けることを Docker に指示する。
-d	デーモン モードで実行することを Docker に指示する。tty を使わず、バックグラウンドで実行する（コマンドに表示されない。このモードで DIGITS を実行することは非推奨）。
-p p1:p2	外部アクセス用にホスト ポート p1 をコンテナ ポート p2 にマッピングすることを Docker に指示する。このオプションは、DIGITS の出力をファイアウォールの背後にプッシュする場合に役に立つ。
-u id:gid	ファイルへのアクセス権限を得るため、ユーザー id とグループ id を使ってコンテナを実行することを Docker に指示する。
-v d1:d2	ホスト ディレクトリ d1 をコンテナのディレクトリ d2 にマッピングすることを Docker に指示する。  重要： データをコンテナの外部に保存できるので、とても便利なオプションです。
--env-file	どの環境変数をコンテナに設定するかを Docker に指示する。
--shm-size ...	DIGITS のマルチ GPU エラーを回避するための一時的なオプション。
container	実行するコンテナ インスタンスを Docker に指示する (nvcr.io/nvidia/digits:17.05 など)。
command	コンテナが起動した後で実行を開始する、オプションのコマンド。この例では、このオプションを使いません。

DIGITS が実行を開始した後で、システムの IP アドレスとポートを使ってブラウザを開きます。たとえば、URL が `http://dgxip:5000/` だとします。ポートがブロックされている場合、SSH トンネルが設定されていれば、「[DIGX Best Practices](#)」を参照)、URL `http://localhost:5000/` を使用できます。

この例では、オプション `-v /datasets:/digits_data:ro` によって、データセットが `/digits_data` (コネクタ内部) にマウントされます。コネクタの外部では、データセットは `/datasets` (システム上の任意のパスを使用できる) に保存されています。コンテナ内部では、データが `/digits_data` にマッピング

されます。また、オプション `:ro` を指定して、読み取り専用で (`ro`) マウントしています。



重要：コンテナの外部と内部のパスには、完全修飾パス名を使うことを強く推奨します。

システムとコンテナの使い方を学ぶためにデータセットが必要であれば、DIGITS からダウンロードできる [標準データセット](#) を使用してください。

DIGITS コンテナに含まれる Python スクリプトを使って、特定のサンプル データセットをダウンロードできます。 `digits.download_data` と呼ばれるこのツールで、 [MNIST](#)、 [CIFAR-10](#)、 [CIFAR-100](#) の各データセットをダウンロードできます。また、このスクリプトをコマンド ラインで使って DIGITS を実行すると、サンプル データセットがプルされます。次の例では、MNIST データセットを使用します。

```
docker run --gpus all --rm -ti \  
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \  
  --env-file=${HOME}/digits_workdir/digits_config_env.sh \  
  -v /datasets:/digits_data \  
  --entrypoint=bash \  
  nvcr.io/nvidia/digits:17.05 \  
  -c 'python -m digits.download_data mnist /digits_data/digits_mnist'
```

ここでは、コンテナへのエントリ ポイントがオーバーライドされ、データセットをダウンロードする `bash` コマンドが実行されます (`-c` オプション)。必要に応じて、データセットのパスを調整してください。

MNIST データを使った DIGITS の実行例は、 [こちら](#) でご覧になれます。

さらに多くの DIGITS 使用例は、 [こちら](#) で参照できます。

第 8 章 フレームワークの 一般的なベスト プラクティス

NVIDIA は、DGX システムの一部として、主要なディープ ラーニング フレームワーク向けに調整、最適化、テストされた、すぐに実行できる状態にある Docker コンテナを提供しています。これらのコンテナは、[NGC コンテナ レジストリ](https://ngc.nvidia.com/catalog/containers) (nvcr.io) から利用できるため、そのまま使ったり、独自のコンテナを作成するための基盤として使ったりできます。

このセクションでは、フレームワークを効果的に使うためのヒントを紹介します。Docker の使用方法に関するベスト プラクティスは、「[Docker And Container Best Practices](#)」に記載されています。NVIDIA コンテナを使い始めるには、「[Preparing To Use NVIDIA Containers](#)」を参照してください。

8.1. コンテナを拡張する

nvcr.io のコンテナ（フレームワーク）の使い方には、いくつかのベスト プラクティスがあります。先ほども触れたとおり、コンテナの 1 つを選び、それを土台にする（拡張する）ことができます。こうすると、ある意味で、新しいコンテナを特定のフレームワークやコンテナのバージョンに固定することになります。この方法を使えるのは、フレームワークの派生物を作成したり、フレームワークまたはコンテナに新たな機能を追加したりする場合です。

ただし、フレームワークを拡張する場合は、数か月以内にそのフレームワークが変更される可能性が高いことを理解しておいてください。ディープ ラーニングとディープ ラーニング フレームワークは、非常に早いペースで開発が進められているからです。特定のフレームワークを拡張することで、拡張した部分はフレームワークの特定のバージョンに縛られます。フレームワークが変更されると、拡張した部分を新しいバージョンに追加しなくてはならず、作業負担が増えます。可能であれば、拡張部分を特定のコンテナに関連付けずに、コンテナの外部に置くことを強くおすすめします。拡張部分を内部に配置する場合は、取り込み用のパッチについてフレームワークの開発チームと相談してください。

8.2. データセットとコンテナ

データセットをコンテナに含めたいと思う人もいるかもしれませんが、しかし、そうすると、そのコンテナは特定のバージョンに依存することになります。フレームワークを新しいバージョンにアップデートする場合、または別のフレームワークに移行する場合、データもコピーしなければなりません。これでは、ハイペースで開発されるフレームワークに追従するのは困難です。

ベスト プラクティスは、「データセットをコンテナに収めない」ことです。できれば、ビジネス ロジック コードをコンテナに収めることも避けるべきです。なぜなら、データセットやビジネス ロジック コードをコンテナに収めると、コンテナの使い方を一般化するのが難しくなるからです。

そうする代わりに、必要なデータセットと、実行するビジネス ロジック コードを収めたディレクトリのみがあるファイル システムをコンテナにマウントします。特定のデータセット、ビジネス ロジックをコンテナから分離することで、コンテナ（フレームワーク、コンテナの 1 バージョンなど）の変更が容易になり、データやコードが含まれるコンテナを再ビルドする必要もありません。

以降のセクションでは、コンテナ レジストリ (nvcr.io) のコンテナに含まれる主要なフレームワークについて、ベスト プラクティスを簡単に紹介します。また、それらのコンテナのいくつかを用いて Keras 利用する方法についても、1 セクションを使って説明します。Keras は、高レベルに抽象化されたディープ ラーニング フレームワークであり、多くのユーザーに利用されています。

8.3. Keras とコンテナ化フレームワーク

[Keras](#) は、TensorFlow、Theano、Microsoft Cognitive Toolkit v2.x リリースに対応した Python フロントエンドで、広く利用されています。これらのフレームワークに使用できる、高レベルのニューラル ネットワーク API が Keras に実装されています。Keras は開発のペースが非常に速いため、nvcr.io 内のコンテナには含まれません。Keras を任意のコンテナに追加することはできますが、nvcr.io からコンテナを起動し、起動プロセス中に Keras をインストールする方法がいくつかあります。ここでは、Keras を仮想 Python 環境で使用するためのスクリプトも紹介します。

Keras とその使い方に関するベスト プラクティスを学ぶ前に、背景知識として [virtualenv](#) と [virtualenvwrapper](#) について理解することをおすすめします。

Keras を実行するときは、必要なフレームワーク バックエンドを指定する必要があります。これを行うには、`$HOME/.keras/keras.json` ファイルか、環境変数 `KERAS_BACKEND=<backend>` を使います（ここで backend には `theano`、`tensorflow`、`cntk` のいずれかを選択できます）。Python コードの最小限の変更でフレームワークを選べるのが、Keras の人気の理由です。

Keras をコンテナ化フレームワークで使えるように構成する方法はいくつかあります。



重要：信頼性が特に高いのは、Keras を含めてコンテナを作成する方法、または Keras をコンテナ内にインストールする方法です。

コンテナ化サービスが展開済みであれば、Keras を含めてコンテナを設定するとよいでしょう。



重要：開発環境では、Keras を含めて仮想 Python 環境を設定することをおすすめします。

この仮想環境をコンテナにマッピングすれば、Keras コードを目的のフレームワークによるバックエンドで実行できます。

Python 環境をコンテナ化フレームワークから分離することで得られるメリットは、M 個のコンテナと N 個の環境があるすると、作成するコンテナの数が $M * N$ 個ではなく、 $M + N$ 個で済むことです。この設定では、目的のコンテナを起動し、Keras Python 環境をそのコンテナ内にアクティブ化する起動スクリプトまたはオーケストレーション スクリプトを使用します。この設定のデメリットは、仮想 Python 環境とフレームワーク バックエンドとの互換性を保証するためにテストが必要になることです。環境に互換性がなければ、コンテナの内部で仮想 Python 環境を再作成して互換性を与える必要があるでしょう。

8.3.1. Keras をコンテナに追加する

Keras を既存のコンテナに追加することもできます。フレームワークと同様に Keras でも変更が頻繁にあるため、Keras の変更について最新情報を得る必要があります。

Keras を既存のコンテナにインストールする方法は 2 つあります。どちらかの方法で作業を始める前に、「[Docker And Containers Best Practices](#)」のガイドをよく読み、既存のコンテナをもとにしたビルド方法を理解してください。

1 つ目の方法では、Python の OS バージョンで Python ツールの `pip` を使って Keras をインストールします。

```
# sudo pip install keras
```

インストールされた Keras のバージョンを確認してください。システム OS バージョンに対応した古いバージョンであり、期待したバージョンや、必要とされるバージョンではないかもしれません。その場合は、次の説明に従って、Keras をソース コードからインストールします。

もう 1 つの方法は、Keras を ソース からビルドするというものです。メイン ブランチからダウンロードするのではなく、リリース の 1 つをダウンロードすることをおすすめします。手順をシンプルに示すと、次のとおりです。

1. 1 つのリリースを `.tar.gz` 形式でダウンロードします (`.zip` 形式も可)。
2. コンテナを `TensorFlow` で起動します。
3. ホーム ディレクトリをボリュームとしてコンテナにマウントします（「[Using And Mounting File Systems](#)」を参照）。
4. コンテナに移動し、シェル プロンプトを開きます。
5. Keras リリースを解凍 (`untar`) します (`.zip` ファイルの場合は `unzip`)。
6. `cd` でディレクトリを切り替えます。

```
# cd keras  
# sudo python setup.py install
```

Keras を仮想 Python 環境の一部として使う場合は、次のセクションでその手順を確認してください。

8.3.2. Keras 仮想 Python 環境を作成する

仮想 Python 環境で Keras を使う方法を学ぶ前に、Keras の インストール依存関係を確認することをおすすめします。これらの依存関係は、データサイエンスを扱う Python 環境である NumPy、SciPy、YAML、h5py に共通するものです。cuDNN も使用できますが、これはフレームワーク コンテナにすでに含まれています。

Keras を仮想 Python 環境で実行するためのスクリプトも提供され、詳細がドキュメントに記載されています。手動で操作する場合と比べて、簡単に Keras を実行できます。

`venvfnsh` スクリプトは、すべてのユーザーがアクセス可能な、システム上のディレクトリに配置する必要があります。たとえば、`/usr/share/virtualenvwrapper/` に置きます。管理者は、ユーザー全員がアクセスできるディレクトリに、このスクリプトを配置してください。

`setup_keras.sh` スクリプトは、`py-keras` 仮想 Python 環境を `~/.virtualenvs` ディレクトリ (ユーザーのホーム ディレクトリ内にある) に作成します。ユーザーは次のようにスクリプトを実行できます。

```
$. /setup_keras.sh
```

このスクリプトを使うと、ローカル ユーザーとして、ホーム ディレクトリがマウントされた `nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04` コンテナを起動できます。スクリプトの重要な部分を以下に抜粋します。

```
dname=${USER}_keras

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04
```

重要: Keras ファイルを作成するときは、`-u` オプションまたは `--user` オプションを使ってアクセス権を正しく設定してください。`-d` オプションと `-t` オプションは、コンテナ プロセスをデーモン化します。つまり、コンテナはバックグラウンドでデーモン サービスとして動作し、コードの実行を受け付けます。

`docker exec` を使って、コードのスニペット (スクリプト) を実行したり、対話形式でコンテナに接続したりすることができます。以下は、Keras 仮想 Python 環境を設定するスクリプトの一部です。

```
docker exec -it $dname \
  bash -c 'source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
  mkvirtualenv py-keras
  pip install --upgrade pip
  pip install keras --no-deps
  pip install PyYaml
  # pip install -r /path/to/requirements.txt
  pip install numpy
  pip install scipy
  pip install ipython'
```

Python パッケージの数が多い場合は、それらを `requirements.txt` ファイルにまとめて記載して、次のようにインストールすることができます。

```
pip install -r /path/to/requirements.txt --no-deps
```

メモ: この行は上記のコマンドにも含まれていますが、不要なのでコメントアウトされています。

`--no-deps` オプションは、パッケージの依存関係をインストールしないことを指示します。このオプションを指定するのは、Keras をインストールするとデフォルトで TensorFlow もインストールされるからです。

重要: TensorFlow のような最適化されていないフレームワークをインストールしたくない場合は、`--no-deps` オプションを使うとインストールが阻止されます。

スクリプトで「`bash -c ...`」から始まる行に注目してください。これは、先ほど触れたスクリプト (`venvfns.sh`) を指します。このスクリプトは、システム上の共通の場所に配置されている必要があります。後から別のパッケージが必要になった場合は、コンテナを再起動し、それらの新しいパッケージを上記の手順または対話形式の手順で追加できます。次のコード スニペットは、対話形式の手順です。

```
dname=${USER}_keras
```

```
docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04

sleep 2 # wait for above container to come up

docker exec -it $dname bash
```

これで仮想 Python 環境がアクティブになるので、次に対話セッションにログインし、必要なものをインストールします。次の例では、h5py をインストールします。h5py は、モデルを HDF5 形式で保存するために Keras で使用されます。

```
source ~/.virtualenvs/py-keras/bin/activate
pip install h5py
deactivate
exit
```

基盤として使うライブラリの不足でインストールに失敗した場合は、コンテナに root で接続し、不足しているライブラリをインストールすることができます。

次の例では、python-dev パッケージをインストールしています。Python.h が不足している場合は、このパッケージによってインストールされます。

```
$ docker exec -it -u root $dname \
  bash -c 'apt-get update && apt-get install -y python-dev # anything else...'
```

使い終わったコンテナは、次のコマンドを使って停止または削除することができます。

```
$ docker stop $dname && docker rm $dname
```

8.3.3. Keras 仮想 Python 環境とコンテナ化 フレームワークを使用する

次の例では、前のセクションで説明した手順で py-keras venv (Python 仮想環境) が作成済みであると仮定します。このセクションで使うすべてのスクリプトは、「[スクリプト](#)」セクションに掲載されています。

[run_kerastf_mnist.sh](#) スクリプトでは、Keras venv をアクティブにしてから、それを使って Keras MNIST コードの mnist_cnn.py をデフォルト バックエンド TensorFlow で実行します。Keras の標準的な使用例は、[こちら](#)で参照できます。

[run_kerastf_mnist.sh](#) スクリプトを、Theano を使う [run_kerasth_mnist.sh](#) スクリプトと比較してみてください。違いは大きく 2 つあります。

1. バックエンド コンテナ nvcr.io/nvidia/theano:17.05 が、nvcr.io/nvidia/tensorflow:17.05 の代わりに使用される。
2. スクリプトのコード起動セクションで、KERAS_BACKEND=theano が指定される。スクリプトは次のコマンドで実行できます。

```
$/run_kerasth_mnist.sh # Ctrl^C to stop running
$/run_kerastf_mnist.sh
```

`run_kerastf_cifar10.sh` スクリプトは、パラメーターを受け取るように変更されています。CIFAR-10 データが収められた外部データ ディレクトリを指定する方法は、このスクリプトでわかります。`cifar10_cnn_filesystem.py` スクリプトは、元の `cifar10_cnn.py` から変更されています。このコードをシステム上でコマンドラインから実行するには、次のようなコマンドを使います。

```
$/run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
```

この例では、ストレージがシステムの `/datasets/cifar` にマウントされていると仮定します。



重要：ここで理解しておいてほしいのは、コンテナの内部でコードを実行するには、起動スクリプトを設定する必要があります。

起動スクリプトは一般化してパラメーター化すると便利です。このようなスクリプトをカスタムのアプリケーションやワークフロー向けに作成することは、エンドユーザーまたは開発者の仕事です。

例：

1. このスクリプトでは、パラメーターが次のように一時変数に結合されます。

```
function join { local IFS="$1"; shift; echo "$*"; }
script_args=$(join : "$@")
```

2. パラメーターは、次のオプションでコンテナに渡されます。

```
-e script_args="$script_args"
```

3. コンテナ内では、次の行でパラメーターが分割され、計算コードに渡されます。

```
python $cifarcode ${script_args//:/ }
```

4. 起動スクリプトへの以下のオプションを通じて、外部ストレージが読み取り専用として渡されます。

```
-v /datasets/cifar:/datasets/cifar:ro
```

次に、以下を実行します。

```
--datadir=/datasets/cifar
```

パラメーターの解析によって起動ロジックを一般化し、重複を回避することで、`run_kerastf_cifar10.sh` スクリプトを改良できます。getopts またはカスタムのパーサーを使って bash でパラメーターを解析するには、いくつかの方法があります。非 bash 起動スクリプトを独自に記述して、Python、Perl などを使うことができます。

`run_keras_script` スクリプトは、高レベルでパラメーター化された bash 起動スクリプトです。次の例では、このスクリプトを使って、前述の MNIST と CIFAR の使用例を実行しています。

```
# running Tensorflow MNIST
./run_keras_script.sh \
  --container=nvcr.io/nvidia/tensorflow:17.05 \
  --script=examples/keras/mnist_cnn.py

# running Theano MNIST
./run_keras_script.sh \
  --container=nvcr.io/nvidia/theano:17.05 --backend=theano \
```

```

--script=examples/keras/mnist_cnn.py

# running Tensorflow Cifar10
./run_keras_script.sh \
  --container=nvcr.io/nvidia/tensorflow:17.05 --backend=tensorflow \
  --script=examples/keras/cifar10_cnn_filesystem.py \
  --epochs=3 --datadir=/datasets/cifar

# running Theano Cifar10
./run_keras_script.sh \
  --container=nvcr.io/nvidia/theano:17.05 --backend=theano \
  --datamnt=/datasets/cifar \
  --script=examples/keras/cifar10_cnn_filesystem.py \
  --epochs=3 --datadir=/datasets/cifar

```



重要：コンテナの停止後にファイルシステムに書き込んで永続化する必要がある出力が、コードから生成される場合は、そのロジックを追加する必要があります。

この例では、ホームディレクトリがコンテナに「書き込み可能」でマウントされます。これで、コードから結果をユーザーのホームパス内のどこかに書き込めます。ファイルシステムパスをコンテナにマウントし、計算コードに指定するか渡す必要があります。

これらの例は、計算コードのオーケストレーションに Keras を使う方法と使わない方法を具体的に示しています。



重要：実際には、多くの場合で、コンテナの起動、コンテナへの接続、コード実行をすべて対話形式で行うのが便利です。

これらの対話セッションでは、コードのデバッグと開発が（ヘルパー スクリプトによる自動化を利用して）より簡単に行えます。対話セッションでは、次のように一連のコマンドを手動でターミナルに入力します。

```

# in bash terminal
dname=mykerastf

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -v /datasets/cifar:/datasets/cifar:ro -w $workdir \
  nvcr.io/nvidia/tensorflow:17.05

docker exec -it $dname bash
# now interactively in the container.
source ~/.virtualenvs/py-keras/bin/activate
source ~/venvfnsh
enablevenvglobalsitepackages
./run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
# change some parameters or code in cifar10_cnn_filesystem.py and run again
./run_kerastf_cifar10.sh --aug --epochs=2 --datadir=/datasets/cifar
disablevenvglobalsitepackages
exit # exit interactive session in container

docker stop $dname && docker rm $dname # stop and remove container

```

8.3.4. コンテナ化 VNC デスクトップ環境を操作する

コンテナ化デスクトップに求められるものは、データセンターのセットアップによって異なります。システムがログイン ノードまたはヘッド ノードの背後でオンプレミス システム向けにセットアップされる場合、一般にデータセンターでは、VNC ログイン ノードを提供するか、X Window System をログイン ノードで実行して、テキスト エディターや IDE (統合開発環境) のようなビジュアル ツールを使えるようにします。

クラウドベース システム (NGC) には、すでにファイアウォールとセキュリティ ルールが提供されていることがあります。その場合は、VNC などに適切なポートが開かれていることを確認してください。

このシステムで開発とコンピューティングの大半を行う場合は、コンテナ化デスクトップを使って、デスクトップに似た環境をセットアップすることができます。この手順と Dockerfile は、[こちら](#)でご覧になれます。

コンテナの最新リリースをこのシステムにダウンロードできます。次のステップでは、Dockerfile の FROM フィールドを次のように変更します。

```
FROM nvcr.io/nvidia/cuda:11.0-cudnn6-devel-ubuntu20.04
```

これは、NVIDIA DGX 製品チームが公式にサポートするコンテナではありません。つまり、このコンテナは nvcr.io に含まれません。ここでは、Eclipse や Sublime Text (Sublime Text とよく似ているが無料で使える Visual Studio Code の代用を推奨) などの便利な GUI ドリブン ツールを開発に使うため、システムにデスクトップに似た環境をセットアップする方法の例として、このコンテナを提供しています。

build_run_dgxdesk.sh サンプル スクリプトは、GitHub サイトで入手して、コンテナ化デスクトップのビルドと実行に利用できます ([「スクリプト」](#) セクションを参照)。DGX Station や NGC のような他のシステムでも、手順は似ています。

システムに接続するには、そのシステムに対応した VNC クライアントを RealVnc からダウンロードするか、Web ブラウザーを使います。

```
=> connect via VNC viewer hostip:5901, default password: vncpassword
=> connect via noVNC HTML5 client: http://hostip:6901/?password=vncpassword
```

第 9 章 HPC と HPC 可視化コンテナ

HPC 可視化コンテナ

NGC コンテナ レジストリは、[NVIDIA の最適化フレームワーク](#)と HPC コンテナへのアクセスだけでなく、HPC 用の科学分野向け可視化コンテナもホストします。これらのコンテナは、科学分野向け可視化ツールとして広く使われている [ParaView](#) を利用して動作します。

一般に、HPC 環境での可視化には、リモートの可視化が必要とされます。つまり、データはリモートの HPC システムまたはクラウドに存在し、そこで処理されます。ユーザーは、ワークステーションからグラフィカルなインターフェイスを使って、このアプリケーションを操作します。一部の可視化コンテナは、特別なクライアント アプリケーションを必要とすることから、HPC 可視化コンテナは 2 つのコンポーネントで構成されます。

サーバー コンテナ

サーバー コンテナは、サーバー システム上のファイルにアクセスする必要があります。このアクセスを許可する方法が説明されている参照先については、後述します。サーバー コンテナは、シリアル モードまたはパラレル モードで実行できます。現在のアルファ リリースでは、シリアル モード構成の提供に重点が置かれています。パラレル構成についての問い合わせは、hpcviscontainer@nvidia.com までお寄せください。

クライアント コンテナ

クライアント アプリケーションとサーバー コンテナのバージョンを確実に一致させるため、NVIDIA はクライアント アプリケーションをコンテナに入れて提供しています。サーバー コンテナと同じように、クライアント コンテナの側でも、サーバー コンテナとの接続を確立するために一部のポートにアクセスする必要があります。

また、クライアント コンテナは、グラフィカル ユーザー インターフェイスを表示するためにユーザーの X サーバーにアクセスする必要があります。

NVIDIA では、可視化製品やその他のデータを保存するためにホスト ファイル システムをクライアント コンテナにマッピングすることを推奨しています。そのうえで、クライアントとサーバー コンテナの接続が開かれている必要もあります。

使用できる HPC 可視化コンテナのリストと使い方については、「[NGC Container User Guide](#)」を参照してください。

第 10 章 コンテナとフレームワークのカスタマイズと拡張

NVIDIA Docker イメージは、調整済みのすぐ使えるパッケージで提供されますが、新しいイメージをゼロからビルドしたり、既存のイメージをカスタムのコード、ライブラリ、データ、または自社インフラストラクチャ向けの設定で強化したりすることもできます。このセクションでは、一連の演習を通じて、コンテナをゼロから作成する手順、コンテナをカスタマイズする手順、ディープ ラーニング フレームワークを拡張して機能を追加する手順、この拡張したフレームワークを開発環境で使ってコードを開発する手順、そのコードをバージョン付きのリリースとしてパッケージ化する手順を説明します。

通常は、ユーザーがコンテナをビルドする必要はありません。NGC コンテナ レジストリ ([nvcr.io](https://ngc.nvidia.com)) には、すぐ使える多数のコンテナが用意されています。ディープ ラーニング、科学分野の計算と可視化に使えるコンテナや、CUDA Toolkit のみが含まれるコンテナなどがあります。

コンテナの特に便利な点は、新しいコンテナを作成するための出発点として使えることです。これを、コンテナを「カスタマイズする」、「拡張する」と言います。コンテナはゼロから作成することもできますが、たいていのコンテナは GPU システムで動作するので、OS と CUDA が含まれる [nvcr.io](https://ngc.nvidia.com) コンテナをもとに作成することをおすすめします。ただし、例外もあります。システムの CPU 上で動作し、GPU を使わないコンテナであれば、ゼロから作成してよいでしょう。その場合、Docker から最小構成の OS コンテナを取得して始めることができます。ただしその場合でも、開発を楽にするために、CUDA が含まれるコンテナから始めてかまいません。コンテナの使用時に CUDA を使わなければよいのです。

DGX システムをお使いの場合は、変更または拡張したコンテナを NGC コンテナ レジストリ ([nvcr.io](https://ngc.nvidia.com)) にプッシュ（保存）できます。このコンテナは、DGX システムの他のユーザーと共有できますが、管理者に相談する必要があります。

すべてのディープ ラーニング フレームワーク イメージに、フレームワークをビルドするためのソースとすべての前提条件が含まれることを確認してください。



注意： Docker のビルド時に NVIDIA ドライバーを Docker イメージにインストールしないでください。

10.1. コンテナをカスタマイズする

NVIDIA は、テスト済み、調整済みのすぐ使える多数のイメージを NGC コンテナ レジストリに収めて提供しています。イメージの 1 つをプルしてコンテナを作成し、任意に選んだソフトウェアまたはデータを追加できます。

ベスト プラクティスは、新しい Docker イメージの開発に `docker commit` を使わず、代わりに Dockerfile を使うことです。Dockerfile を使うと、Docker イメージの開発中に行った変更を効率よくバージョンで管理

できる可視化と機能が得られます。docker commit は、短期間しか存在しない破棄可能なイメージのみに使用してください（「[使用例 3：docker commit を使用してコンテナをカスタマイズする](#)」を参照）。

Dockerfile の書き方については、[ベスト プラクティス関連ドキュメント](#)を参照してください。

10.1.1. コンテナをカスタマイズするメリットと制限事項

特定のニーズを満たすためにコンテナをカスタマイズすることができます。カスタマイズする理由はたくさんありますが、NVIDIA が提供するコンテナに含まれていない特定のソフトウェアを使いたいこともその 1 つです。理由を問わず、コンテナをカスタマイズできます。

サンプルのデータセットまたはモデル定義は、フレームワークのソースによって含まれていない限り、コンテナイメージには含まれません。サンプルのデータセットまたはモデルがあるか、コンテナを確認してください。

10.1.2. 使用例 1：コンテナをゼロからビルドする

このタスクの概要

Docker では、Dockerfile を使って Docker イメージの作成またはビルドを行います。Dockerfile とは、新しい Docker イメージを作成するために Docker で連続して使われるコマンドが含まれるスクリプトです。簡単に言えば、Dockerfile とはコンテナ イメージのソース コードです。基本 OS のみを使う場合でも、Dockerfile の記述は常に、継承元となる基本イメージで始まります。

Dockerfile の書き方に関するベスト プラクティスについては、「[Best practices for writing Dockerfiles](#)」を参照してください。

実習として、Ubuntu 20.04 を基本 OS として使う Dockerfile からコンテナを作成してみましょう。コンテナの作成時に OS のアップデートも実施します。

手順

1. ワーキング ディレクトリをローカル ハードドライブに作成します。
2. このディレクトリで、テキスト エディターを開き、Dockerfile というファイルを作成します。このファイルをワーキング ディレクトリに保存します。
3. この Dockerfile を開き、以下のコードを追加します。

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install -y curl
CMD echo "hello from inside a container"
```

最後の行の CMD に指定したコマンドが、コンテナの作成時に実行されます。これは、コンテナが正常にビルドされたことを確認する 1 つの方法です。

この例では、コンテナを NGC リポジトリではなく、Docker リポジトリからプルしています。NVIDIA® リポジトリを使う例もこの後で紹介します。

4. Dockerfile を保存し、閉じます。
5. イメージをビルドします。次のコマンドを実行して、イメージをビルドし、タグを作成します。

```
$ docker build -t <new_image_name>:<new_tag> .
```

メモ: このコマンドは、Dockerfile があるディレクトリで実行します。

Docker ビルド プロセスからの出力には、Dockerfile 内の 1 行ごとに「Step」と表示されます。

たとえば、コンテナの名前を test1 とし、これに latest というタグを付けます。説明の都合上、このプライベートな DGX システム リポジトリの名前は nvidian_sas としています（実際の名前は、DGX を登録したときの名前になります。これは一般的に、何らかの会社名です）。次のコマンドは、コンテナをビルドします。出力を以下に抜粋したので、参考にしてください。

```
$ docker build -t test1:latest .
Sending build context to Docker daemon 8.012 kB
Step 1/3 : FROM ubuntu:20.04
14.04: Pulling from library/ubuntu
...
Step 2/3 : RUN apt-get update && apt-get install -y curl
...
Step 3/3 : CMD echo "hello from inside a container"
---> Running in 1f391b9285d8
---> 934785072daf
Removing intermediate container 1f391b9285d8
Successfully built 934785072daf
```

イメージをビルドする方法の詳細については、「[docker build](#)」を参照してください。イメージにタグ付ける方法の詳細については、「[docker tag](#)」を参照してください。

6. ビルドが正常に実行されたことを確認します。次のようなメッセージが表示されます。

```
Successfully built 934785072daf
```

このメッセージは、ビルドが正常に実行されたことを示します。これ以外のメッセージが表示された場合は、ビルドが正常に実行されなかったということです。

メモ: イメージがビルドされると、ランダムな番号（ここでは 934785072daf）が割り当てられます。

7. イメージを表示できるか確認します。次のコマンドを実行して、コンテナを表示します。

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
test1               latest             934785072daf       19 minutes ago    222 MB
```

新しいコンテナが使用可能になりました。

メモ: このコンテナは、この DGX システム上にローカルに存在します。コンテナをプライベートなリポジトリに保存するには、次の手順を行ってください。

メモ: この操作を行うには DGX システムが必要です。

8. コンテナをプライベートな Docker リポジトリに保存するには、コンテナをプッシュします。

- a) . コンテナをプッシュする最初の手順として、タグを付けます。

```
$ docker tag test1 nvcr.io/nvidian_sas/test1:latest
```

- b) . タグ付けしたら、nvcr.io 内のたとえば nvidian_sas というプライベートなプロジェクトにコンテ

ナーをプッシュします。

```
$ docker push nvcr.io/nvidian_sas/test1:latest
The push refers to a repository [nvcr.io/nvidian_sas/test1]
...
```

c) . コンテナが `nvidian_sas` リポジトリに表示されることを確認します。

10.1.3. 使用例 2 : Dockerfile を使用してコンテナをカスタマイズする

このタスクの概要

この例では、Dockerfile を使って、`nvcr.io` 内の PyTorch コンテナをカスタマイズします。コンテナをカスタマイズする前に、`docker pull` コマンドを使って、PyTorch 21.02 コンテナがレジストリに読み込まれたことを確認してください。

```
$ docker pull nvcr.io/nvidia/pytorch:21.02-py3
```

すでに説明したとおり、`nvcr.io` の Docker コンテナは、フレームワークにパッチを適用してから Docker イメージを再ビルドする方法を記述したサンプルの Dockerfile も提供します。`/workspace/docker-examples` ディレクトリには、2 つのサンプル Dockerfile があります。この例では、コンテナをカスタマイズするためのテンプレートとして `Dockerfile.customcaffe` ファイルを使います。

手順

1. ワーキング ディレクトリを `my_docker_images` という名前でローカル ハード ドライブに作成します。
2. テキスト エディターを開き、Dockerfile というファイルを作成します。このファイルをワーキング ディレクトリに保存します。
3. Dockerfile を再び開き、次のコード行を追加します。ファイルを保存します。

```
FROM nvcr.io/nvidia/pytorch:21.02
# APPLY CUSTOMER PATCHES TO PYTORCH
# Bring in changes from outside container to /tmp
# (assumes my-pytorch-modifications.patch is in same directory as
Dockerfile)
#COPY my-pytorch-modifications.patch /tmp

# Change working directory to PyTorch source path
WORKDIR /opt/pytorch

# Apply modifications
#RUN patch -p1 < /tmp/my-pytorch-modifications.patch

# Note that the default workspace for caffe is /workspace
RUN mkdir build && cd build && \
    cmake -DCMAKE_INSTALL_PREFIX=PATH=/usr/local -DUSE_NCCL=ON
-DUSE_CUDNN=ON -DCUDA_ARCH_NAME=Manual -DCUDA_ARCH_BIN="35 52 60 61"
-DCUDA_ARCH_PTX="61" .. && \
    make -j"$(nproc)" install && \
```

```
make clean && \
cd .. && rm -rf build

# Reset default working directory
WORKDIR /workspace
```

4. `docker build` コマンドを使ってイメージをビルドし、リポジトリ名とタグを指定します。次の例では、リポジトリの名前は `corp/pytorch` で、タグは `21.02.1PlusChanges` です。コマンドは次のようになります。

```
$ docker build -t corp/pytorch:21.02.1PlusChanges .
```

5. Docker イメージを実行します。

```
docker run --gpus all -ti --rm corp/pytorch:21.02.1PlusChanges .
```

10.1.4. 使用例 3: `docker commit` を使用してコンテナをカスタマイズする

このタスクの概要

この例では、`docker commit` コマンドを使って、コンテナの現在の状態を Docker イメージにフラッシュします。これは推奨されるベスト プラクティスではありませんが、動作中のコンテナに対して変更があり、それを保存したい場合は便利な方法です。この例では、`apt-get` タグを使い、ユーザーが `root` として実行する必要があるパッケージをインストールしています。

メモ:

- ▶ この例では、説明の都合上、`NVCaffe` イメージのリリース 17.04 を使っています。
- ▶ コンテナを実行するときに `--rm` フラグを使わないでください。コンテナを実行するときに `--rm` フラグを使うと、コンテナの終了時に変更内容が失われます。

手順

1. Docker コンテナを `nvcr.io` リポジトリから DGX システムにプルします。たとえば、次のコマンドは `NVCaffe` コンテナをプルします。

```
$ docker pull nvcr.io/nvidia/caffe:17.04
```

2. コンテナを DGX システムで実行します。

```
docker run --gpus all -ti nvcr.io/nvidia/caffe:17.04

=====
== NVIDIA Caffe ==
=====

NVIDIA Release 17.04 (build 26740)

Container image Copyright (c) 2017, NVIDIA CORPORATION. All rights reserved.
Copyright (c) 2014, 2015, The Regents of the University of California (Regents)
All rights reserved.
```

```
Various files include modifications (c) NVIDIA CORPORATION. All rights reserved.
NVIDIA modifications are covered by the license terms that apply to the underlying
project or file.

NOTE: The SHMEM allocation limit is set to the default of 64MB. This may be insufficient
for NVIDIA Caffe. NVIDIA recommends the use of the following flags:
    docker run --gpus all --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 ...

root@1fe228556a97:/workspace#
```

- ここでコンテナの root ユーザーに切り替わるはずですが（プロンプトに注目してください）。apt コマンドを使って、パッケージをプルし、コンテナ内に配置します。

メモ: NVIDIA コンテナは、apt-get パッケージ マネージャーを使う Ubuntu を使ってビルドされます。使用する特定のコンテナの詳細については、「[Deep Learning Documentation](#)」のリリース ノートで確認してください。

この例では、MATLAB の GNU クローンである Octave をコンテナにインストールします。

```
# apt-get update
# apt install octave
```

メモ: まず apt-get update を実行してから、apt を使って Octave をインストールします。

- ワークスペースを終了します。

```
# exit
```

- docker ps -a を使って、コンテナのリストを表示します。docker ps -a コマンドからは次のように出力されます。

```
$ docker ps -a
CONTAINER ID          IMAGE                    CREATED              ...
1fe228556a97         nvcr.io/nvidia/caffe:17.04  3 minutes ago      ...
```

- これで、Octave がインストールされた動作中のコンテナから新しいイメージを作成できます。コンテナを次のコマンドでコミットできます。

```
$ docker commit 1fe228556a97 nvcr.io/nvidian_sas/caffe_octave:17.04
sha256:0248470f46e22af7e6cd90b65fdee6b4c6362d08779a0bc84f45de53a6ce9294
```

- イメージのリストを表示します。

```
$ docker images
REPOSITORY              TAG          IMAGE ID          ...
nvidian_sas/caffe_octave 17.04       75211f8ec225     ...
```

- 確認のため、コンテナを再び実行し、Octave が実際にあるかどうかをチェックします。

メモ: この操作が可能なのは、DGX-1 と DGX Statio のみです。

```

docker run --gpus all -ti nvidian_sas/caffe_octave:17.04

=====
== NVIDIA Caffe ==
=====

NVIDIA Release 17.04 (build 26740)

Container image Copyright (c) 2017, NVIDIA CORPORATION. All rights reserved. Copyright
(c) 2014, 2015, The Regents of the University of California (Regents) All rights
reserved.

Various files include modifications (c) NVIDIA CORPORATION. All rights reserved. NVIDIA
modifications are covered by the license terms that apply to the underlying project
or file.

NOTE: The SHMEM allocation limit is set to the default of 64MB. This may be
insufficient for NVIDIA Caffe. NVIDIA recommends the use of the following flags:
docker run --gpus all --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 ...

root@2fc3608ad9d8:/workspace# octave
octave: X11 DISPLAY environment variable not set
octave: disabling GUI features
GNU Octave, version 4.0.0
Copyright (C) 2015 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1>

```

Octave プロンプトが表示されるので、Octave はインストールされています。

9. コンテナをプライベートなリポジトリに保存 (Docker 用語で「プッシュ」) したい場合は、`docker push ...` コマンドを使います。

```
$ docker push nvcr.io/nvidian_sas/caffe_octave:17.04
```

結果

新しい Docker イメージが使用できる状態になります。ローカルの Docker リポジトリにこのイメージがあることを確認してください。

10.1.5. 使用例 4 : Docker を使用してコンテナを開発する

このタスクの概要

開発者がコンテナを拡張しようとする動機は主に 2 つです。

1. プロジェクトの不変の依存関係をすべて含むが、ソース コードは含まない開発イメージを作成する。
2. ソースの固定バージョンとすべてのソフトウェア依存関係が含まれる本番環境イメージまたはテスト用イメージを作成する。

データセットは、コンテナ イメージにパッケージ化されません。コンテナ イメージの設計段階で、データセットと結果に使うボリューム マウントについて考慮に入れることが理想です。

以降に記載した例では、ローカル データセットをホスト上の `/raid/datasets` からコンテナ内部の `/dataset` に読み取り専用でマウントします。

また、現在の実行から出力をキャプチャするために、ジョブ固有のディレクトリをマウントします。これらの例では、コンテナを起動するたびにタイムスタンプ付きの出力ディレクトリを作成し、それをコンテナの `/output` にマッピングします。

こうすると、コンテナが起動するたびに出力がキャプチャされ、個別に保存されます。モデルの開発と反復のためにソースをコンテナに含めると、さまざまな操作が難しくなり、ワークフロー全体が複雑になりすぎる可能性があります。

たとえば、ソース コードがコンテナに含まれていると、エディター、バージョン管理ソフトウェア、ドットファイルなどもコンテナに含める必要があります。ただし、ソース コードの実行に必要なものをすべて含む開発イメージを作成すれば、ソース コードをコンテナにマッピングすることで、ホスト ワークステーションの開発環境を利用できます。モデルの固定バージョンを共有する場合は、ソース コードのバージョン付きコピーと、開発環境でトレーニングされた重みをパッケージ化するのが最適です。

ここで具体例として、Isola 氏を中心とするグループが提供しているオープン ソース実装 [Image-to-Image Translation with Conditional Adversarial Networks](#) ([pix2pix](#) から入手可能) を使って、開発と配布の手順を実習します。Pix2Pix は、Conditional Adversarial Network を使用して入力イメージから出力イメージをマッピングする方法を学習するための Torch 実装です。オンライン プロジェクトはいずれ変更される可能性があるため、ここではスナップショット バージョン `d7e7b8b557229e75140cbe42b7f5dbf85a67d097` 変更セットを使って説明します。

このセクションではコンテナを仮想環境として使っています。つまり、コンテナにはプロジェクトに必要なすべてのプログラムとライブラリが含まれます。



メモ：ここではネットワーク定義とトレーニング スクリプトをコンテナ イメージから分離しています。これは反復開発には便利なモデルです。作業中のファイルはホスト側に保存され、コンテナには実行時にのみマッピングされるからです。

元プロジェクトとの差分を調べる方法については、「[Comparing changes](#)」を参照してください。

開発に使うマシンが、長期的なトレーニング セッションを実行するマシンと異なる場合、現在の開発状態をコンテナにパッケージ化することもできます。

手順

1. ワーキング ディレクトリをローカル ハードドライブに作成します。

```
mkdir Projects
$ cd ~/Projects
```

2. Git クローンで Pix2Pix Git リポジトリのクローンを作成します。

```
$ git clone https://github.com/phillipi/pix2pix.git
$ cd pix2pix
```

3. git checkout コマンドを実行します。

```
$ git checkout -b devel d7e7b8b557229e75140cbe42b7f5dbf85a67d097
```

4. データセットをダウンロードします。

```
bash ./datasets/download_dataset.sh facades
I want to put the dataset on my fast /raid storage.
$ mkdir -p /raid/datasets
$ mv ./datasets/facades /raid/datasets
```

5. Dockerfile というファイルを作成し、次のコードを追加します。

```
FROM nvcr.io/nvidia/torch:17.03
RUN luarocks install nnggraph
RUN luarocks install
https://raw.githubusercontent.com/szym/display/master/display-scm-0.rockspec
WORKDIR /source
```

6. 開発用 Docker コンテナ イメージをビルドします (build-devel.sh)。

```
docker build -t nv/pix2pix-torch:devel .
```

7. 次の train.sh スクリプトを作成します。

```
#!/bin/bash -x
ROOT="${ROOT:-/source}"
DATASET="${DATASET:-facades}"
DATA_ROOT="${DATA_ROOT:-/datasets/$DATASET}"
DATA_ROOT=$DATA_ROOT name="${DATASET}_generation"
which_direction=BtoA th train.lua
```

実際にこのモデルで開発するとしたら、ホスト側でファイルを変更してから、コンテナ内部で実行されるトレーニング スクリプトを実行することで、開発を反復します。

8. **オプション**: ファイルを編集し、変更が終わるたびに次のステップを実行します。
9. トレーニング スクリプトを実行します (run-devel.sh)。

```
docker run --gpus all --rm -ti -v $PWD:/source -v
/raid/datasets:/datasets nv/pix2pix-torch:devel ./train.sh
```


10.1.5.1. 使用例 4.1：ソースをコンテナにパッケージ化する

このタスクの概要

モデル定義とスクリプトをコンテナにパッケージ化する手順は、かなりシンプルです。COPY ステップを Dockerfile に追加すれば作業は完了です。

ボリュームのマウントを省き、コンテナにパッケージ化されたソースを使うように実行スクリプトを更新します。パッケージ化されたコンテナは、内部コードが固定されるので、`devel` コンテナ イメージと比べて移植性が高くなります。このコンテナ イメージに特定のタグを付けてバージョンを管理し、コンテナレジストリに保存することをおすすめします。

コンテナを実行するスクリプトの更新も、これと同じぐらい簡単です。ローカル ソースのボリュームをコンテナにマウントするコードを省けば完了です。

10.2. フレームワークをカスタマイズする

各 Docker イメージには、フレームワークのビルドに必要なコードが含まれているので、フレームワーク自体に変更を加えることができます。各イメージのフレームワーク ソースは、`/workspace` ディレクトリに格納されます。

特定のディレクトリの場所については、「[Deep Learning Framework Release Notes](#)」で、特定のフレームワークに関する情報を参照してください。

10.2.1. フレームワークをカスタマイズするメリットと制限事項

フレームワークのカスタマイズが便利なのは、NVIDIA リポジトリの外部でパッチまたは変更をフレームワークに適用したい場合や、フレームワークに適用したい特別なパッチがある場合です。

10.2.2. 使用例 1：コマンドラインを使用してフレームワークをカスタマイズする

このタスクの概要

サンプルの Dockerfile を使って、NVCaffe コンテナ イメージ内のソース コードにパッチを適用し、NVCaffe を再ビルドする手順を示します。以下に示した RUN コマンドでは、元のイメージをビルドしたときと同じ方法で NVCaffe を再ビルドします。

このように、コンテナを対話形式で変更するのではなく、Dockerfile と `dockerbuild` を使ってカスタマイズを適用することで、NVCaffe コンテナ イメージの以降のバージョンに同じ変更を適用する手順が簡単になります。

詳細については、「[Dockerfile reference](#)」を参照してください。

手順

1. Dockerfile 向けにワーキング ディレクトリを作成します。

```
$ mkdir docker
$ cd docker
```

2. テキスト エディターを開き、Dockerfile というファイルを作成してから、次のコードを追加します。

```
FROM nvcr.io/nvidia/caffe:17.04
RUN apt-get update && apt-get install bc
```

3. コンテナ外部からの変更を /tmp に配置します。



メモ: ここでは my-caffe-modifications.patch が Dockerfile と同じディレクトリにあると仮定しています。

```
COPY my-caffe-modifications.patch /tmp
```

4. ワーキング ディレクトリを NVCaffe ソース パスに変更します。

```
WORKDIR /opt/caffe
```

5. 変更内容を適用します。

```
RUN patch -p1 < /tmp/my-caffe-modifications.patch
```

6. NVCaffe を再ビルドします。

```
RUN mkdir build && cd build && \
  cmake -DCMAKE_INSTALL_PREFIX:PATH=/usr/local -DUSE_NCCL=ON -DUSE_CUDNN=ON \
    -DCUDA_ARCH_NAME=Manual -DCUDA_ARCH_BIN="35 52 60 61" -DCUDA_ARCH_PTX="61" .. && \
  make -j"$(nproc)" install && \
  make clean && \
  cd .. && rm -rf build
```

7. デフォルトのワーキング ディレクトリにリセットします。

```
WORKDIR /workspace
```

10.2.3. 使用例 2：フレームワークをカスタマイズし、コンテナを再ビルドする

このタスクの概要

これは、フレームワークをカスタマイズしてコンテナを再ビルドする手順の一例です。ここでは、NVCaffe 17.03 フレームワークを使用します。

現在、NVCaffe フレームワークは、ネットワーク レイヤーが作成されると、次の出力メッセージを stdout に返します。

```
"Creating Layer"
```

たとえば、NVCaffe 17.03 コンテナ内の bash シェルで次のコマンドを実行すると、この出力が表示されます。

```
# which caffe
/usr/local/bin/caffe
# caffe time --model /workspace/models/bvlc_alexnet/deploy.prototxt
--gpu=0
...
I0523 17:57:25.603410 41 net.cpp:161] Created Layer data (0)
I0523 17:57:25.603426 41 net.cpp:501] data -> data
I0523 17:57:25.604748 41 net.cpp:216] Setting up data
...
```

以下では、NVCaffe でメッセージ "Created Layer" を "Just Created Layer" に変更する手順を説明します。この例で、既存のフレームワークの変更方法がわかります。

前提条件

フレームワーク コンテナを対話モードで実行する必要があります。

手順

1. NVCaffe 17.03 コンテナを `nvcr.io` リポジトリで探します。

```
$ docker pull nvcr.io/nvidia/caffe:17.03
```

2. コンテナを DGX システムで実行します。

```
docker run --gpus all --rm -ti nvcr.io/nvidia/caffe:17.03
```



メモ: この操作を行うと、コンテナ内で root ユーザーに切り替わります。プロンプトの表示が変わることに注目してください。

3. NVCaffe ソース ファイル `/opt/caffe/src/caffe/net.cpp` でファイルを編集します。変更する行は、162 行の前後です。

```
# vi /opt/caffe/src/caffe/net.cpp
:162 s/Created Layer/Just Created Layer
```



メモ: ここでは vi を使用しています。メッセージ "Created Layer" を "Just Created Layer" に変更します。

4. NVCaffe を再ビルドします。

```
# cd /opt/caffe
# cmake -DCMAKE_INSTALL_PREFIX=PATH=/usr/local -DUSE_NCCL=ON
-DUSE_CUDNN=ON -DCUDA_ARCH_NAME=Manual -DCUDA_ARCH_BIN="35 52 60
61" -DCUDA_ARCH_PTX="61" ..
# make -j"$(proc)" install
# make install
# ldconfig
```

- 更新された NVcaffe フレームワークを実行する前に、更新された NVcaffe バイナリが正しい場所（`/usr/local/` など）にあることを確認します。

```
# which caffe
/usr/local/bin/caffe
```

- NVcaffe を実行し、`stdout` の出力が変更されていることを確認します。

```
# caffe time --model /workspace/models/bvlc_alexnet/deploy.prototxt
--gpu=0
/usr/local/bin/caffe
...
I0523 18:29:06.942697 7795 net.cpp:161] Just Created Layer data (0)
I0523 18:29:06.942711 7795 net.cpp:501] data -> data
I0523 18:29:06.944180 7795 net.cpp:216] Setting up data
...
```

- コンテナを `nvcr.io` のプライベートな DGX リポジトリまたはプライベートな Docker リポジトリに保存します（「[使用例 2：Dockerfile を使用してコンテナをカスタマイズする](#)」を参照）。

10.3. Docker コンテナのサイズを最適化する

レイヤーを使用する Docker コンテナ形式では、設計の段階で、コンテナ イメージがインスタンス化されるときに転送する必要があるデータの量に制限が課されています。Docker コンテナ イメージがインスタンス化される、つまりリポジトリから「プル」されるときに、Docker がレイヤーをリポジトリからローカル ホストにコピーしなければならないことがあります。Docker では、各レイヤーのハッシュを使ってそのレイヤーがすでにホスト上にあるかどうかをチェックします。レイヤーがすでにローカル ホストにあれば「再ダウンロード」せずに、時間と（わずかな量ですが）ネットワークの使用量を節約します。

これが NVIDIA の NGC にとって特に便利なのは、すべてのコンテナが同じ基本 OS とライブラリを使ってビルドされるからです。NGC からコンテナ イメージを実行し、さらに別のコンテナ イメージを実行する場合、最初のコンテナにある多くのレイヤーが 2 番目のコンテナでも使われる可能性が高いため、2 番目のコンテナ イメージをプルする時間が短縮され、コンテナをすばやく実行できます。

ほとんどあらゆるものをコンテナに収めることができるので、ユーザーまたはコンテナ開発者は非常に大きな（GB 以上）のコンテナを作成できます。データを Docker コンテナ イメージに含めることは推奨されませんが、ユーザーや開発者は（十分な理由があつて）そうすることがあります。データを含めると、コンテナ イメージのサイズはさらに増え、コンテナ イメージや各種レイヤーのダウンロード時間が長くなります。こうなると、ユーザーや開発者は、コンテナ イメージまたは個々のレイヤーのサイズを減らす方法を模索するようになります。

以下では、イメージまたはレイヤーのサイズが大きすぎる場合、つまりサイズを小さくしたい場合に使えるオプションをいくつか紹介します。最適な唯一のオプションというものはないため、これらをコンテナ イメージで試してみてください。

10.3.1. 各 RUN コマンドを 1 行に記述する

Dockerfile では、1 行に 1 つの `RUN` コマンドを記述すると、とても便利です。コマンドが 1 つずつ目に入るので、コードが読みやすくなります。ただし、このように記述すると、Docker ではコマンドごとにレイヤーが作成されます。各レイヤーには作成元や作成時間、含まれるもの、固有のハッシュに関する一定の情報（メタデータ）が保存されます。コマンドがたくさんあると、メタデータの量が増えます。

コンテナ イメージのサイズを減らす簡単な方法は、可能であればすべての RUN コマンドを 1 つの RUN ステートメントにまとめることです。こうすると、RUN コマンドは相当な大きさになりますが、メタデータの量は大幅に減ります。なるべく多くの RUN コマンドをグループ化することをおすすめします。Dockerfile によっては、すべての RUN コマンドを 1 つの RUN ステートメントにまとめられないこともあります。RUN コマンドの数を減らすのにベストを尽くす必要がありますが、合理的に考えることも重要です。

以下に、コンテナ イメージをビルドするためのシンプルな Dockerfile の例を示します。

```
$ cat Dockerfile
FROM ubuntu:20.04

RUN date > /build-info.txt
RUN uname -r >> /build-info.txt

Notice there are two RUN commands in this simple Dockerfile. The container image can be
built using the following command and associated output.
$ docker build -t first-image -f Dockerfile .
...
Step 2/3 : RUN date > /build-info.txt
---> Using cache
---> af12c4b34f91
Step 3/3 : RUN uname -r >> /build-info.txt
---> Running in 0f883f37e3c8
...
```

RUN コマンドごとにコンテナ イメージ内にレイヤーが作成されることに注目してください。

このコンテナ イメージでレイヤーがどのように作成されるのか調べてみましょう。

```
$ docker run --rm -it first-image cat /build-info.txt
Mon Jan 18 10:14:02 UTC 2021
5.5.115-1.el7.elrepo.x86_64

$ docker history first-image
IMAGE          CREATED          CREATED BY                                      SIZE
d2c03aa61290  11 seconds ago  /bin/sh -c uname -r >> /build-info.txt        57B
af12c4b34f91  16 minutes ago  /bin/sh -c date > /build-info.txt             29B
5e8b97a2a082  6 weeks ago     /bin/sh -c #(nop) CMD ["/bin/bash"]          0B
<missing>     6 weeks ago     /bin/sh -c mkdir -p /run/systemd && echo 'do... 7B
<missing>     6 weeks ago     /bin/sh -c sed -i 's/^#\s*\s*(deb.*universe\)$... 2.76kB
<missing>     6 weeks ago     /bin/sh -c rm -rf /var/lib/apt/lists/*        0B
<missing>     6 weeks ago     /bin/sh -c set -xe && echo '#!/bin/sh' > /... 745B
<missing>     6 weeks ago     /bin/sh -c #(nop) ADD file:d37ff24540ea7700d... 114MB
```

このコマンドの出力から、各レイヤーに関する情報が得られます。RUN コマンドごとにレイヤーがあることに注目してください。

ここで、Dockerfile を編集して 2 つの RUN コマンドを結合してみましょう。

```
$ cat Dockerfile
FROM ubuntu:20.04

RUN date > /build-info.txt && uname -r >> /build-info.txt
$ docker build -t one-layer -f Dockerfile .

$ docker history one-layer
IMAGE          CREATED          CREATED BY                                      SIZE
3b1ef5bc19b2  6 seconds ago   /bin/sh -c date > /build-info.txt && uname -... 57B
5e8b97a2a082  6 weeks ago     /bin/sh -c #(nop) CMD ["/bin/bash"]          0B
```

<missing>	6 weeks ago	/bin/sh -c mkdir -p /run/systemd && echo 'do...	7B
<missing>	6 weeks ago	/bin/sh -c sed -i 's/^\s*\s*(deb.*universe\)\$...	2.76kB
<missing>	6 weeks ago	/bin/sh -c rm -rf /var/lib/apt/lists/*	0B
<missing>	6 weeks ago	/bin/sh -c set -xe && echo '#!/bin/sh' > /...	745B
<missing>	6 weeks ago	/bin/sh -c #(nop) ADD file:d37ff24540ea7700d...	114MB

これで、2つの RUN コマンドが含まれる 1つのレイヤーが作成されたことがわかります。

RUN コマンドを結合する理由はもう 1つあります。レイヤーが複数あれば、コンテナ イメージ内の 1つのレイヤーだけを修正すればよく、コンテナ イメージ全体を修正する必要はありません。

10.3.2. エクスポート、インポート、フラット化

スペースが貴重な場合は、既存のコンテナ イメージからすべての履歴を除去することもできます。これができるのは、動作中のコンテナがあるときです。コンテナが起動したら、次の 2つのコマンドを実行します。

```
# export the container to a tarball
docker export <CONTAINER ID> > /home/export.tar

# import it back
cat /home/export.tar | docker import - some-name:<tag>
```

各レイヤーから履歴が除去されますが、レイヤーは保持されます（削除される心配はありません）。

ほかに、イメージを 1つのレイヤーに「フラット化」するというオプションもあります。フラット化でレイヤーからすべての冗長性が除去され、1つのコンテナが作成されます。履歴を除去する場合と同様に、これも実行中のコンテナがあるときに使用できます。コンテナが起動したら、次のコマンドを実行します。

```
docker export <CONTAINER ID> | docker import - some-image-name:<tag>
```

このパイプラインでは、コンテナが import コマンドによってエクスポートされ、レイヤーが 1つしかない新しいコンテナが作成されます。詳しくはこの[ブログ記事](#)をご覧ください。

10.3.3. docker-squash

数年前、Docker がまだリリースされていない頃に、[docker-squash](#) というツールでイメージを「スカッシュ」する機能が追加されました。2年ほどアップデートがないツールですが、Docker コネクタ イメージのサイズ削減のために今もよく使われます。このツールでは、Docker コンテナ イメージを受け取り、それを 1つのレイヤーに「スカッシュ」することでレイヤー間の共通性とレイヤーの履歴を減らし、可能な限り小さなコンテナ イメージを作成します。

このツールでは PORT、ENV などの Docker コマンドが利用され、イメージの動作はスカッシュの前後でまったく同じです。また、スカッシュ プロセス中に削除されたファイルは、実際にイメージから削除されます。

以下は docker-squash のシンプルな実行例です。

```
docker save <ID> | docker-squash -t <TAG> [-from <ID>] | docker load
```

このパイプラインでは現在のイメージを受け取り、保存し、新しいタグでスカッシュして、コンテナを再び読み込みます。生成されるイメージでは、元の FROM レイヤー下のすべてのレイヤーが 1つのレイヤーにスカッシュされています。docker-squash ではデフォルトで基本イメージ レイヤーが保持されるため、アップデー

トをイメージにプッシュしてプルするときに転送を繰り返す必要はありません。

このツールは、コンテナに最終処理が行われていて今後は更新されないことを想定して設計されました。つまり、レイヤーと履歴について詳細な情報を保持する必要はほとんどありません。イメージをスカッシュし、本番環境に配置できます。イメージのサイズを最小限にまで減らすと、ユーザーはイメージを速やかにダウンロードし、実行することができます。

10.3.4. ビルド中にスカッシュする

Docker がリリースされてから、転送に時間がかかる巨大なイメージが作成されるようになるまで、長くはかかりませんでした。その段階で、ユーザーや開発者はコンテナのサイズをどうやって減らすかを考え始めました。ビルド中にイメージをスカッシュする機能を追加するパッチが Docker 向けに提案されたのは、それほど前ではありません。squash オプションは、Docker 1.13 (API 1.25) で追加されました。このときはまだ、Docker は別のバージョン付け規則を使っていました。Docker 17.06-ce 時点で、このオプションはまだ試験的 (experimental) な機能に分類されています。この試験的なオプションを使いたい場合は、Docker にオプションの使用を指示できます (詳しくは Docker のドキュメントを参照)。ただし、NVIDIA ではこのオプションをサポートしていません。

--squash オプションは、コンテナのビルド時に使用します。コマンドは次のように使用します。

```
docker build --squash -t chamilad/testdocker:0.1 .
```

このコマンドでは、コンテナのビルドに「Dockerfile」を Dockerfile として使用します。

--squash オプションは、2 つのレイヤーがあるイメージを作成します。一般に Dockerfile の先頭にある FROM で最初のレイヤーが生成されます。それ以降のレイヤーは、すべて 1 つのレイヤーに「スカッシュ」されます。これで最初のレイヤーを除き、すべてのレイヤーから履歴が除去されます。また、冗長なファイルも削除されます。

まだ試験的な機能なので、イメージのサイズをどれくらい圧縮できるかは場合によりますが、50% 削減できたという報告が複数あります。

10.3.5. その他のオプション

イメージのサイズ削減に使えるオプションは他にもありますが、それらは特に Docker を基盤とするものではありません (いくつかはあります)。それ以外はどれも Linux に昔からあるコマンドです。

Docker のビルド オプションには、Docker コンテナ内でのアプリケーションのビルドを扱うものがあります。コンテナの作成時にアプリケーションをビルドする場合、ビルドに使ったツールを残すとイメージのサイズが大きくなるので、残したくないと思うかもしれません。動作中のコンテナが変更されないことを想定するならば、そのとおりです。Docker コンテナがレイヤーとしてビルドされることを思い出してください。この事実をコンテナのビルド時に利用して、バイナリをレイヤー間でコピーすることができます。

たとえば、Docker ファイルはこうなります。

```
$ cat Dockerfile
FROM ubuntu:20.04

RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
        build-essential \
        gcc && \
```

```
rm -rf /var/lib/apt/lists/*

COPY hello.c /tmp/hello.c
RUN gcc -o /tmp/hello /tmp/hello.c
```

コンテナをビルドし、gcc をインストールし、シンプルな「hello world」アプリケーションをビルドします。コンテナの履歴をチェックすると、レイヤーのサイズがわかります。

```
$ docker history hello
IMAGE          CREATED          CREATED BY                                      SIZE
49fef0e11806  8 minutes ago  /bin/sh -c gcc -o /tmp/hello /tmp/hello.c    8.6kB
44a449445055  8 minutes ago  /bin/sh -c #(nop) COPY file:8f0c1776b2571c38... 63B
c2e5b659a549  8 minutes ago  /bin/sh -c apt-get update -y && apt-get ...    181MB
5e8b97a2a082  6 weeks ago    /bin/sh -c #(nop) CMD ["/bin/bash"]          0B
<missing>     6 weeks ago    /bin/sh -c mkdir -p /run/systemd && echo 'do... 7B
<missing>     6 weeks ago    /bin/sh -c sed -i 's/^#\s*(deb.*universe\)$... 2.76kB
<missing>     6 weeks ago    /bin/sh -c rm -rf /var/lib/apt/lists/*        0B
<missing>     6 weeks ago    /bin/sh -c set -xe && echo '#!/bin/sh' > /... 745B
<missing>     6 weeks ago    /bin/sh -c #(nop) ADD file:d37ff24540ea7700d... 114MB
```

ビルド ツールが含まれるレイヤーのサイズは 181MB なのに、アプリケーション レイヤーのサイズは 8.6KB しかないことに注目してください。ビルド ツールが最終的なコンテナに不要であれば、イメージから除去できます。ただし、apt-get remove ... コマンドを実行しただけでは、ビルド ツールは除去されません。

次の Dockerfile のように、以前のレイヤーからバイナリを新しいレイヤーにコピーすればビルド ツールは除去されます。

```
$ cat Dockerfile
FROM ubuntu:16.04 AS build

RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
        build-essential \
        gcc && \
    rm -rf /var/lib/apt/lists/*

COPY hello.c /tmp/hello.c

RUN gcc -o /tmp/hello /tmp/hello.c

FROM ubuntu:16.04

COPY --from=build /tmp/hello /tmp/hello
```

この方法を「マルチステージ」ビルドと呼びます。この Dockerfile では、最初のステージが OS に「build」と名付けることから始まります。次に、ビルド ツールがインストールされ、ソースがコンテナにコピーされ、バイナリがビルドされます。

次のレイヤーは、新しい OS FROM コマンドで始まります（これを「第 1 ステージ」と呼びます）。Docker ではこのコマンドで始まるレイヤーのみが保存され、それ以降のレイヤー、つまり「第 2 ステージ」は保存されません（言い換えるなら、ビルド ツールをインストールした最初のレイヤーは保存されません）。第 2 ステージでは、バイナリを第 1 ステージからコピーできます。このステージにはビルド ツールが含まれません。コンテナ イメージのビルドは、以前と同じです。

最初の Dockerfile を使ったコンテナのサイズと、2 番目の Dockerfile を使ったコンテナのサイズは、以下のように比較できます。最初の出力は、元の Dockerfile です。


```
$ docker images hello
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
hello               latest      49fef0e11806     21 minutes ago  295MB
$ docker images hello-rt
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
hello-rt            latest      f0cef59a05dd     2 minutes ago   114MB
```

2 番目の出力は、マルチステージの Dockerfile です。サイズの違いに注目してください。

Docker コンテナのサイズを減らす方法として、小さな基本イメージから始めることもおすすめです。一般に、配布用の基本イメージはスリムですが、イメージに何がインストールされるかを確認することをおすすめします。不要なものがあれば、それらを省いた独自の基本イメージを作成することができます。

また、`apt-get clean` コマンドを実行して、イメージに含まれているかもしれないパッケージのキャッシュを消去することもできます。

第 11 章 スクリプト

11.1. DIGITS

11.1.1. run_digits.sh

```
#!/bin/bash
# file: run_digits.sh

mkdir -p $HOME/digits_workdir/jobs

cat <<EOF > $HOME/digits_workdir/digits_config_env.sh
# DIGITS Configuration File
DIGITS_JOB_DIR=$HOME/digits_workdir/jobs
DIGITS_LOGFILE_FILENAME=$HOME/digits_workdir/digits.log
EOF

docker run --gpus all --rm -ti --name=${USER}_digits -p 5000:5000 \
-u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
--env-file=${HOME}/digits_workdir/digits_config_env.sh \
-v /datasets:/digits_data:ro \
--shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 \
nvcr.io/nvidia/digits:17.05
```

11.1.2. digits_config_env.sh

```
# DIGITS Configuration File
DIGITS_JOB_DIR=$HOME/digits_workdir/jobs
DIGITS_LOGFILE_FILENAME=$HOME/digits_workdir/digits.log
```

11.2. TensorFlow

11.2.1. run_tf_cifar10.sh

```
#!/bin/bash
# file: run_tf_cifar10.sh

# run example:
```

```

# ./run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
# Get usage help via:
# ./run_kerastf_cifar10.sh --help 2>/dev/null
_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd) "

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir
cifarcodes=${_basedir}/examples/tensorflow/cifar/cifar10_multi_gpu_train.py
# cifarcodes=${_basedir}/examples/tensorflow/cifar/cifar10_train.py

function join { local IFS="$1"; shift; echo "$*"; }

script_args=$(join : "$@")

dname=${USER}_tf

docker run --gpus all --name=$dname -d -t \
  --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -v /datasets/cifar:/datasets/cifar:ro -w $workdir \
  -e cifarcodes=$cifarcodes -e script_args="$script_args" \
  nvcr.io/nvidia/tensorflow:17.0

sleep 1 # wait for container to come up

docker exec -it $dname bash -c 'python $cifarcodes ${script_args//:/ /}'

docker stop $dname && docker rm $dname

```

11.3. Keras

11.3.1. venvfns.sh

```

#!/bin/bash
# file: venvfns.sh
# functions for virtualenv

[[ "${BASH_SOURCE[0]}" == "${0}" ]] && \
  echo Should be run as : source "${0}" && exit 1

enablevenvglobalsitepackages() {
  if ! [ -z ${VIRTUAL_ENV+x} ]; then
    _libpypath=$(dirname $(python -c \
      "from distutils.sysconfig import get_python_lib; print(get_python_lib())"))
    if ! [[ "${_libpypath}" == *"${VIRTUAL_ENV}"* ]]; then
      return # VIRTUAL_ENV path not in the right place
    fi
    no_global_site_packages_file=${_libpypath}/no-global-site-packages.txt
    if [ -f $no_global_site_packages_file ]; then
      rm $no_global_site_packages_file;
      echo "Enabled global site-packages"
    else
      echo "Global site-packages already enabled"
    fi
  fi
}

```

```

disablevenvglobalsitepackages() {
  if ! [ -z ${VIRTUAL_ENV+x} ]; then
    _libppath=$(dirname $(python -c \
      "from distutils.sysconfig import get_python_lib; print(get_python_lib())"))
    if ! [ "${_libppath}" == *"${VIRTUAL_ENV}"* ]; then
      return # VIRTUAL_ENV path not in the right place
    fi
    no_global_site_packages_file=${_libppath}/no-global-site-packages.txt
    if ! [ -f $no_global_site_packages_file ]; then
      touch $no_global_site_packages_file
      echo "Disabled global site-packages"
    else
      echo "Global site-packages were already disabled"
    fi
  fi
}

```

11.3.2. setup_keras.sh

```

#!/bin/bash
# file: setup_keras.sh

dname=${USER}_keras

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04

docker exec -it -u root $dname \
  bash -c 'apt-get update && apt-get install -y virtualenv virtualenvwrapper'

docker exec -it $dname \
  bash -c 'source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
  mkvirtualenv py-keras
  pip install --upgrade pip
  pip install keras --no-deps
  pip install PyYaml
  pip install numpy
  pip install scipy
  pip install ipython'

docker stop $dname && docker rm $dname

```

11.3.3. run_kerastf_mnist.sh

```

#!/bin/bash
# file: run_kerastf_mnist.sh

_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir
mnistcode=${_basedir}/examples/keras/mnist_cnn.py

dname=${USER}_keras

```

```

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -w $workdir -e mnistcode=$mnistcode \
  nvcr.io/nvidia/tensorflow:17.05

sleep 1 # wait for container to come up

docker exec -it $dname \
  bash -c 'source ~/.virtualenvs/py-keras/bin/activate
  source ~/venvfns.sh
  enableenvglobalsitepackages
  python $mnistcode
  disableenvglobalsitepackages'

docker stop $dname && docker rm $dname

```

11.3.4. run_kerasth_mnist.sh

```

#!/bin/bash
# file: run_kerasth_mnist.sh

_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd) "

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir
mnistcode=${_basedir}/examples/keras/mnist_cnn.py

dname=${USER}_keras

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -w $workdir -e mnistcode=$mnistcode \
  nvcr.io/nvidia/theano:17.05

sleep 1 # wait for container to come up

docker exec -it $dname \
  bash -c 'source ~/.virtualenvs/py-keras/bin/activate
  source ~/venvfns.sh
  enableenvglobalsitepackages
  KERAS_BACKEND=theano python $mnistcode
  disableenvglobalsitepackages'

docker stop $dname && docker rm $dname

```

11.3.5. run_kerastf_cifar10.sh

```

#!/bin/bash
# file: run_kerastf_cifar10.sh

# run example:
# ./run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
# Get usage help via:
# ./run_kerastf_cifar10.sh --help 2>/dev/null

_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd) "

```

```

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir
cifarcodes=${_basedir}/examples/keras/cifar10_cnn_filesystem.py

function join { local IFS="$1"; shift; echo "$*"; }

script_args=$(join : "$@")

dname=${USER}_keras

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -v /datasets/cifar:/datasets/cifar:ro -w $workdir \
  -e cifarcodes=$cifarcodes -e script_args="$script_args" \
  nvcr.io/nvidia/tensorflow:17.05

sleep 1 # wait for container to come up

docker exec -it $dname \
  bash -c 'source ~/.virtualenvs/py-keras/bin/activate
  source ~/venvfns.sh
  enablevenvglobalsitepackages
  python $cifarcodes ${script_args//:/ }
  disablevenvglobalsitepackages'

docker stop $dname && docker rm $dname

```

11.3.6. run_keras_script

```

#!/bin/bash
# file: run_keras_script.sh

_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir

function join { local IFS="$1"; shift; echo "$*"; }

container="nvcr.io/nvidia/tensorflow:17.05"
backend="tensorflow"
script=''
datamnt=''

usage() {
cat <<EOF
Usage: $0 [-h|--help] [--container=container] [--script=script]
  [--<remain_args>]

Sets up a keras environment. The keras environment is setup in a
virtualenv and mapped into the docker container with a chosen
--backend. Then runs the specified --script.

--container - Specify desired container. Use "=" equal sign.
  Default: ${container}

--backend - Specify the backend for Keras: tensorflow or theano.
  Default: ${backend}

```

```

--script - Specify a script. Specify scripts with full or relative
          paths (relative to current working directory). Ex.:
          --script=examples/keras/cifar10_cnn_filesystem.py

--datamnt - Data directory to mount into the container.

--<remain_args> - Additional args to pass through to the script.

-h|--help - Displays this help.

EOF
}

remain_args=()

while getopts ":h-" arg; do
  case "${arg}" in
    h ) usage
        exit 2
        ;;
    - ) [ $OPTIND -ge 1 ] && optind=$(expr $OPTIND - 1 ) || optind=$OPTIND
        eval _OPTION="\${optind}"
        OPTARG=$(echo $_OPTION | cut -d=' ' -f2)
        OPTION=$(echo $_OPTION | cut -d=' ' -f1)
        case $OPTION in
          --container ) larguments=yes; container="$OPTARG" ;;
          --script ) larguments=yes; script="$OPTARG" ;;
          --backend ) larguments=yes; backend="$OPTARG" ;;
          --datamnt ) larguments=yes; datamnt="$OPTARG" ;;
          --help ) usage; exit 2 ;;
          --* ) remain_args+=($_OPTION) ;;
        esac
        OPTIND=1
        shift
        ;;
    esac
  done

script_args="$(join : ${remain_args[@]})"

dname=${USER}_keras

# formulate -v option for docker if datamnt is not empty.
mntdata=${([[ ! -z "${datamnt//}" ]] && echo "-v ${datamnt}:${datamnt}:ro" )}

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  $mntdata -w $workdir \
  -e backend=$backend -e script=$script -e script_args="$script_args" \
  $container

sleep 1 # wait for container to come up

docker exec -it $dname \
  bash -c 'source ~/.virtualenvs/py-keras/bin/activate
  source ~/.venvfnsh
  enableenvglobalsitepackages
  KERAS_BACKEND=$backend python $script ${script_args//:/ }
  disableenvglobalsitepackages'

docker stop $dname && docker rm $dname

```

11.3.7. cifar10_cnn_filesystem.py

```
#!/usr/bin/env python
# file: cifar10_cnn_filesystem.py
'''
Train a simple deep CNN on the CIFAR10 small images dataset.
'''

from __future__ import print_function
import sys
import os

from argparse import (ArgumentParser, SUPPRESS)
from textwrap import dedent

import numpy as np

# from keras.utils.data_utils import get_file
from keras.utils import to_categorical
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
import keras.layers as KL
from keras import backend as KB

from keras.optimizers import RMSprop

def parser_(desc):
    parser = ArgumentParser(description=dedent(desc))

    parser.add_argument('--epochs', type=int, default=200,
                        help='Number of epochs to run training for.')

    parser.add_argument('--aug', action='store_true', default=False,
                        help='Perform data augmentation on cifar10 set.\n')

    # parser.add_argument('--datadir', default='/mnt/datasets')
    parser.add_argument('--datadir', default=SUPPRESS,
                        help='Data directory with Cifar10 dataset.')

    args = parser.parse_args()

    return args

def make_model(inshape, num_classes):
    model = Sequential()
    model.add(KL.InputLayer(input_shape=inshape[1:]))
    model.add(KL.Conv2D(32, (3, 3), padding='same'))
    model.add(KL.Activation('relu'))
    model.add(KL.Conv2D(32, (3, 3)))
    model.add(KL.Activation('relu'))
    model.add(KL.MaxPooling2D(pool_size=(2, 2)))
    model.add(KL.Dropout(0.25))

    model.add(KL.Conv2D(64, (3, 3), padding='same'))
    model.add(KL.Activation('relu'))
    model.add(KL.Conv2D(64, (3, 3)))
    model.add(KL.Activation('relu'))
```



```

model.add(KL.MaxPooling2D(pool_size=(2, 2)))
model.add(KL.Dropout(0.25))

model.add(KL.Flatten())
model.add(KL.Dense(512))
model.add(KL.Activation('relu'))
model.add(KL.Dropout(0.5))
model.add(KL.Dense(num_classes))
model.add(KL.Activation('softmax'))

return model

def cifar10_load_data(path):
    """Loads CIFAR10 dataset.

    # Returns
    Tuple of Numpy arrays: `(x_train, y_train), (x_test, y_test)`.
    """
    dirname = 'cifar-10-batches-py'
    # origin = 'http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz'
    # path = get_file(dirname, origin=origin, untar=True)
    path_ = os.path.join(path, dirname)

    num_train_samples = 50000

    x_train = np.zeros((num_train_samples, 3, 32, 32), dtype='uint8')
    y_train = np.zeros((num_train_samples,), dtype='uint8')

    for i in range(1, 6):
        fpath = os.path.join(path_, 'data_batch_' + str(i))
        data, labels = cifar10.load_batch(fpath)
        x_train[(i - 1) * 10000: i * 10000, :, :, :] = data
        y_train[(i - 1) * 10000: i * 10000] = labels

    fpath = os.path.join(path_, 'test_batch')
    x_test, y_test = cifar10.load_batch(fpath)

    y_train = np.reshape(y_train, (7, 1))
    y_test = np.reshape(y_test, (6, 1))

    if KB.image_data_format() == 'channels_last':
        x_train = x_train.transpose(0, 2, 3, 1)
        x_test = x_test.transpose(0, 2, 3, 1)

    return (x_train, y_train), (x_test, y_test)

def main(argv=None):
    """
    """
    main.__doc__ = __doc__
    argv = sys.argv if argv is None else sys.argv.extend(argv)
    desc = main.__doc__
    # CLI parser
    args = parser_(desc)

    batch_size = 32
    num_classes = 10
    epochs = args.epochs
    data_augmentation = args.aug

    datadir = getattr(args, 'datadir', None)

```

```

# The data, shuffled and split between train and test sets:
(x_train, y_train), (x_test, y_test) = cifar10_load_data(datadir) \
    if datadir is not None else cifar10.load_data()
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)

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

callbacks = None

print(x_train.shape, 'train shape')
model = make_model(x_train.shape, num_classes)

print(model.summary())

# initiate RMSprop optimizer
opt = RMSprop(lr=0.0001, decay=1e-6)

# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])

nsamples = x_train.shape[0]
steps_per_epoch = nsamples // batch_size

if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True,
              callbacks=callbacks)
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(
        # set input mean to 0 over the dataset
        featurewise_center=False,
        samplewise_center=False, # set each sample mean to 0
        # divide inputs by std of the dataset
        featurewise_std_normalization=False,
        # divide each input by its std
        samplewise_std_normalization=False,
        zca_whitening=False, # apply ZCA whitening
        # randomly rotate images in the range (degrees, 0 to 180)
        rotation_range=0,
        # randomly shift images horizontally (fraction of total width)
        width_shift_range=0.1,
        # randomly shift images vertically (fraction of total height)
        height_shift_range=0.1,
        horizontal_flip=True, # randomly flip images
        vertical_flip=False) # randomly flip images

```

```
# Compute quantities required for feature-wise normalization
# (std, mean, and principal components if ZCA whitening is applied).
datagen.fit(x_train)

# Fit the model on the batches generated by datagen.flow().
model.fit_generator(datagen.flow(x_train, y_train,
                                batch_size=batch_size),
                    steps_per_epoch=steps_per_epoch,
                    epochs=epochs,
                    validation_data=(x_test, y_test),
                    callbacks=callbacks)

if __name__ == '__main__':
    main()
```

第 12 章 トラブルシューティング

Docker コンテナの詳細については、以下のリソースを参照してください。

- ▶ [NGC User Guide](#)
- ▶ [NVIDIA-Docker GitHub](#)
- ▶ [HPC Visualization Containers User Guide](#)

ディープ ラーニング フレームワークのリリース ノートとその他の製品ドキュメントについては、ディープ ラーニング ドキュメントの Web サイトを参照してください。 [Release Notes for Deep Learning Frameworks.](#)

通知事項

本書は情報提供のみを目的としており、製品の特定の機能、状態、または品質を保証するものではありません。NVIDIA Corporation（以下「NVIDIA」という）は、本書に含まれる情報の正確性または完全性について、明示的か黙示的かを問わず、一切の表明も保証も行わないものではなく、ここに含まれる誤りについて一切の責任を負わないものとします。NVIDIA は、これらの情報の結果または使用について一切責任を負わず、その使用に起因して第三者の特許権またはその他の権利の侵害が発生しても一切責任を負わないものとします。本書は、いかなる資料（以下に定義する）、コード、または機能の開発、リリース、または提供も約束するものではありません。

NVIDIA は、本書に対する訂正、修正、加筆、改訂、その他の変更を通知なしに随時行う権利を留保します。

お客様は、注文を行う前に最新の関連情報を入手し、それらの情報が最新かつ完全であることを確認する必要があります。

NVIDIA 製品は、NVIDIA とお客様のそれぞれの正式の権限を有する代表者が署名した個別の販売契約で別途合意がない限り、注文確認時点で提供される NVIDIA の標準的な販売条件に従って販売されます（「販売規約」）。NVIDIA は、本書で参照される NVIDIA 製品の購入に関連してお客様の一般条件を適用することをここに明示的に拒否します。本書によって直接的または間接的にいかなる契約上の義務も生じません。

NVIDIA 製品は、医療、軍事、航空、宇宙、生命維持の各設備で使用したり、NVIDIA 製品の故障または誤動作の結果、負傷、死亡、物的損害、環境損害などが起こることを合理的に予想できるような用途に使用したりするように設計または認可されておらず、そのような使用への適合性は保証されません。

NVIDIA は、そのような設備または用途に NVIDIA 製品を含めたり使用したりすることに対して一切の法的責任を負いません。そのため、そのような使用はお客様自身の責任において行うものとします。

NVIDIA は、本書に基づく製品が特定の用途に適合することについて、一切の表明または保証を行いません。各製品のすべてのパラメーターのテストが NVIDIA によって実行されるとは限りません。

本書に含まれる情報の適用性を評価および判断し、お客様によって計画された用途への製品の適合性を確認し、用途または製品の不履行を避けるためにその用途に対する必要なテストを実施することは、お客様側の責任です。お客様の製品設計に含まれる弱点が NVIDIA 製品の品質および信頼性に影響を及ぼす可能性があり、その結果、本書に含まれていない追加的または異なる条件や要件が生じる可能性があります。NVIDIA は、次に基づく、またはそれに起因する一切の不履行、損害、費用、または問題に対して責任を負いません。(i) 本書に反する方法での NVIDIA 製品の使用、または (ii) お客様の製品設計。

本書は、明示的か黙示的かを問わず、NVIDIA の特許権、著作権、またはその他の知的財産権が適用されるいかなるライセンスも付与するものではありません。サードパーティ製品またはサービスに関して NVIDIA によって公開される情報は、それらの製品またはサービスを使用するための NVIDIA からのライセンスを構成するものではなく、それらの製品またはサービスを保証または是認するものではありません。これらの情報を使用するには、サードパーティの特許権またはその他の知的財産権に基づいてサードパーティから提供されるライセンスが必要になるか、NVIDIA の特許権またはその他の知的財産権に基づいて NVIDIA から提供されるライセンスが必要になる可能性があります。

本書に含まれる情報を複製することは、複製が NVIDIA によって書面で事前に承認されており、改変なしで複製されており、適用されるあらゆる輸出法および規制に完全に準拠し、かつ関連するあらゆる条件、制限、および通知を伴っている場合に限り許可されます。本書とすべての NVIDIA の設計仕様、リファレンス ボード、ファイル、図、診断、リスト、およびその他のドキュメント（以下、総称して「資料」という）は「現状有姿」で提供されます。

NVIDIA は資料について、明示、黙示、法定、その他の方法を問わず、いかなる保証も行わず、非侵害、商品性、および特定の目的への適合性に関するあらゆる黙示的保証を明示的に否認します。法律で禁止されていない範囲において、NVIDIA はいかなる場合も、本書の使用によっていかなる損害（直接的、間接的、特別的、偶発的、懲罰的、または結果的な損害が含まれますが、これらに限定されません）が生じる可能性について知らされていたとしても、その損害の責任を負わないものとします。お客様が何らかの理由で被るいかなる損害にかかわらず、NVIDIA がここに記載される製品に関してお客様に対して負う累積責任は、本製品の販売規約に従って制限されるものとします。

HDMI

HDMI、HDMI ロゴ、および High-Definition Multimedia Interface は、HDMI Licensing LLC の商標または登録商標です。

OpenCL

OpenCL は Apple Inc. の商標であり、Khronos Group Inc. からのライセンス許諾により使用されます。

商標

NVIDIA、NVIDIA ロゴ、cuBLAS、CUDA、cuDNN、DALI、DIGITS、DGX、DGX-1、DGX-2、DGX Station、DLProf、Jetson、Kepler、Maxwell、NCCL、Nsight Compute、Nsight Systems、NVCAffe、NVIDIA Ampere GPU Architecture、PerfWorks、Pascal、SDK Manager、Tegra、TensorRT、Triton Inference Server、Tesla、TF-TRT、Volta は、NVIDIA Corporation の米国およびその他の国における商標または登録商標です。その他の会社名、商品名は関連各社の商標です。

Copyright

© 2017-2021 NVIDIA Corporation & affiliates. All rights reserved.