NVIDIA コンテナーおよびディープラーニング フレームワーク

DU-08518-001_v001 | 2018 年 3 月

ユーザー ガイド

目次

第 1 章
Docker コンテナー

ここ数年、大規模なデータセンター アプリケーションの展開を簡略化するために、ソフトウェア コンテナーの使用が劇的に増加しています。コンテナーは、アプリケーションをライブラリなどの依存関係と共にカプセル化したものであり、仮想マシン全体のオーバーヘッドなしでアプリケーションとサービスの再現性と信頼性を実現できます。

Docker® Engine Utility for NVIDIA® GPU は nvidia-docker (英語) とも呼ばれ、Docker® を使用して複数のマシンに CPU ベースのアプリケーションを展開するのと同様に、Docker コンテナーを使用して複数のマシンに GPU ベースのアプリケーションを移植できます。

このガイドでは、Docker® Engine Utility for NVIDIA® GPU を nvidia-docker と省略して記載します。

Docker コンテナー

Docker コンテナーは、Docker イメージのインスタンスです。Docker は、1 つのコンテナーに対して 1 つのアプリケーションまたはサービスを展開します。

Docker イメージ

Docker イメージは、nvidia-docker コンテナー内で実行されるソフトウェアです (ファイルシステムとパラメーターを含む)。

1.1. Docker コンテナーとは

Docker コンテナーは、Linux システムや同一ホスト上のインスタンスを問わず、アプリケーションの実行環境を常に維持できるように、アプリケーションにライブラリ、データ ファイル、環境変数をバンドルするメカニズムです。

VM は独自のカーネルを持ちますが、コンテナーはホストのシステム カーネルを使用します。そのため、コンテナーからのすべてのカーネル呼び出しは、ホスト システムのカーネルによって処理されます。DGX™ システムは、ディープラーニング フレームワークを展開するためのメカニズムとして Docker コンテナーを使用します。

Docker コンテナーは、Docker イメージ (英語) の実行中のインスタンスです。

1.2. コンテナーを使用するメリット

コンテナーを使用するメリットの 1 つは、実行するシステムごとではなく、アプリケーション、依存関係、環境変数をコンテナー イメージにまとめてインストールできることです。さらに、次のようなメリットもあります。

- アプリケーション、依存関係、環境変数をコンテナー イメージにインストールする際、実行するシステムごとではなく一度にインストールできる。
- 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可能性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、同じサーバーで使用できる。
- アプリケーションをコンテナー内に構築すると、ソフトウェアをインストールすることなくサーバーなどのさまざまな場所で実行できる。
- 従来の高速計算アプリケーションをコンテナー化して、オンプレミスまたはクラウドの新しいシステムに展開できる。
- 固有の GPU リソースをコンテナーに割り当てることにより、システムを分離してパフォーマンスを高めることができる。
- アプリケーションを異なる環境で簡単に共有、共同作業、テストできる。
- 特定のディープラーニング フレームワークの各インスタンスに 1 つ以上の固有の GPU を割り当て、同時に実行できる。
- コンテナー起動時に外部に公開される特定のポートにコンテナーポートをマッピングすることで、アプリケーション間のネットワークポートの競合を解決できる。

第 2 章
Docker と nvidia-docker のインストール

GPU を活用した Docker イメージを移植するために、NVIDIA は **nvidia-docker** を開発しました。これは、起動時に NVIDIA ドライバーのユーザー モード コンポーネントと GPU を Docker コンテナーにマウントするオープンソースのコマンド ライン ツールです。

また、NVIDIA DGX システム固有のソフトウェアを含む **nvidia-docker** のコンテナー セットにより、アプリケーションのパフォーマンスを最大限に発揮し、シングル GPU のパフォーマンスとマルチ GPU のスケーリングを最大まで高めることができます。

NVIDIA GPU をフルに活用する NGC コンテナーには、**nvidia-docker** が必要です。HPC 視覚化コンテナーの前提条件は、DGX コンテナーとは異なります。詳しくは「NGC コンテナー ユーザー ガイド (英語)」を参照してください。

- DGX™ ユーザーは、Docker と **nvidia-docker** を「NVIDIA コンテナーの使用準備 (英語)」の手順に従ってインストールします。
- Amazon Web Services (AWS) P3 ユーザーは、「AWS で NGC を使用するためのガイド (英語)」の手順に従ってください。
- AWS 以外のユーザーは、**nvidia-docker** インストール手順 (英語) に従って、お使いの GPU 製品タイプとオペレーティング システム シリーズ用の最新の NVIDIA ディスプレイ ドライバーをインストールします。システムに NVIDIA ドライバーが構成されていない場合は、ドライバーをダウンロードしてインストールします。
- お使いの NVIDIA GPU がコンピューティング機能 6.0.0 以上のバージョンの CUDA® をサポートしていること (Pascal GPU アーキテクチャ世代以降であることなど) を確認します。
- NGC API キーを使用して **nvcr.io** にログインします。アクセスおよび API キー取得の手順については、「NGC 入門 (英語)」を参照してください。

第 3 章
コンテナーをプルする

NVIDIA DGX-1™、NVIDIA DGX Station™、NVIDIA® GPU Cloud™ (NGC) の 3 製品すべてで、NVIDIA の GPU アクセラレーション コンテナーにアクセスできます。DGX-1 または DGX ステーションを所有している場合は、https://compute.nvidia.com (英語) にある NVIDIA® DGX™ コンテナー レジストリを使用する必要があります。これは、Docker ハブ `nvcr.io` (NVIDIA DGX コンテナー レジストリ) のインターフェイスです。ここからコンテナーをプルし、レジストリの自分のアカウントにコンテナーをプッシュできます。

Amazon® Web Services™ (AWS) などのクラウド サービス プロバイダー経由で NVIDIA® GPU Cloud™ (NGC) コンテナー レジストリから NVIDIA コンテナーにアクセスしている場合は、https://ngc.nvidia.com (英語) で NGC コンテナー レジストリを使用する必要があります。これは、DGX-1 および DGX ステーションの場合と同じ Docker リポジトリの Web インターフェイスです。アカウント作成後にコンテナーをプルする際には、データセンターに DGX-1 がある場合と同じコマンドを使用します。ただし、現時点では、NGC コンテナー レジストリにコンテナーを保存することはできません。その代わりに、オンプレミスまたはクラウドにある自分の Docker レジストリにコンテナーを保存します。

NVIDIA DGX コンテナー レジストリと NGC コンテナー レジストリのどちらからプルしても、コンテナーはまったく同じです。

これらの 3 製品 DGX-1、DGX Station、NVIDIA NGC Cloud Services のいずれも、フレームワーク ソースの場所はコンテナーの `/opt/<framework>` です。

NGC コンテナー レジストリからコンテナーをプルする前に、Docker と nvidia-docker を「NVIDIA コンテナーの使用準備 (英語)」に記載されている場所にインストールする必要があります。また、「NGC 入門 (英語)」に従い、NGC コンテナー レジストリにアクセスしてログインする必要があります。

3.1. 主な概念

pull コマンドと run コマンドを実行するには、次の概念を理解しておく必要があります。これらの概念は、DGX-1、DGX Station、NVIDIA NGC Cloud Services の 3 製品すべてに適用されます。

pull コマンドは次のように使用します。

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

run コマンドは次のように使用します。

```
nvidia-docker run -it --rm -v local _ dir:container _ dir nvcr.io/nvidia/
caffe2:<xx.xx>
```

両方のコマンドを構成する各属性の概念は、次のとおりです。

nvcr.io
コンテナー レジストリの名前。NGC コンテナー レジストリと NVIDIA DGX コンテナー レジストリでは **nvcr.io** です。

nvidia
レジストリ内でコンテナーが含まれる空間の名前。NVIDIA が提供するコンテナーでは、レジストリ空間は nvidia です。

-it
コンテナーをインタラクティブ モードで実行します。

--rm
完了時にコンテナーを削除します。

-v
ディレクトリをマウントします。

local _ dir
コンテナー内からアクセスするホスト システムのディレクトリまたはファイル (絶対パス)。たとえば、次のパスの **local_dir** は、**/ home/jsmith/data/mnist** です。

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

たとえば、コンテナー内でコマンド **ls /data/mnist** を使用すると、コンテナー外部から **ls /home/jsmith/data/mnist** コマンドを実行した場合と同じファイルが表示されます。

container _ dir
コンテナー内の場合のターゲット ディレクトリ。たとえば、次の例では **/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>
タグ。17.10 など。

3.2. NVIDIA DGX コンテナー レジストリからアクセスしてプルする

DGX-1 または DGX Station を所有している場合は、NVIDIA DGX コンテナー レジストリ リポジトリ **nvcr.io** (https://compute.nvidia.com、英語) を使用する必要があります。ここからコンテナーをプルし、自分のアカウントにコンテナーをプッシュできます。

NGC コンテナー レジストリでは、リポジトリ **https://ngc.nvidia.com （英語）** を使用する必要があります。最終的には、NGC コンテナー レジストリで使用されるコンテナーは完全に同じで、同じ NVIDIA DGX コンテナー レジストリ **nvcr.io** を使用します。ただし、この場合、**nvcr.io** からはコンテナーのプルのみで、プッシュはできません。

nvidia-docker コンテナーをプルする前に、次の前提条件が満たされていることを確認してください。

- コンテナーを含むレジストリ空間に対する読み取りアクセス権を持っている。
- NVIDIA DGX コンテナー レジストリまたは NGC コンテナー レジストリにログインしている。
- Docker コマンドを使用できる **docker** グループのメンバーである。

要件の詳細については、DGX-1 と DGX ステーションの場合は「DGX コンテナー レジストリ ユーザー ガイド (英語)」、NVIDIA NGC Cloud Services の場合は、「NGC 入門 (英語)」を参照してください。

使用できるコンテナーを参照するには、Web ブラウザーを使用して適切なコンテナー レジストリアカウントにログインします。DGX-1 および DGX Station では https://compute.nvidia.com (英語)、NVIDIA NGC Cloud Services では https://ngc.nvidia.com (英語) です。

NVIDIA DGX コンテナー レジストリで使用できるコンテナーを参照するには、Web ブラウザーを使用して NVIDIA® DGX Cloud Services™ Web サイトの NVIDIA DGX コンテナー レジストリ アカウントにログインします。

いずれかのレジストリからコンテナーをプルするには、次の手順に従います。

1. コマンドを実行して、レジストリからコンテナーをダウンロードします。

```
$ docker pull nvcr.io/nvidia/<repository>:<tag>
```

nvcr.io は nvidia-docker リポジトリの名前です。

nvidia-docker リポジトリは、DGX-1、DGX Station、NVIDIA NGC Cloud Services の 3 製品すべてで同じです。たとえば、次のコマンドを実行します。

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

この場合、コンテナーは「caffe」リポジトリからプルされ、バージョンは 17.03 (タグは「17.03」) です。

2. コンテナーがダウンロードされたことを確認するために、システム上の Docker イメージを一覧表示します。

```
$ docker images
```

コンテナーのプルが完了すると、ニューラル ネットワークの実行、ディープラーニングモデルの展開、AI 分析の実行などのコンテナー ジョブを実行できます。

3.2.1. NVIDIA DGX コンテナー レジストリからコンテナーをプルする

このセクションは、DGX-1 または DGX Station を所有している場合に該当します。

Docker レジストリは、Docker イメージを保存するサービスです。サービスは、インターネット、企業イントラネット、ローカル マシンに配置できます。DGX-1 および DGX Station では、**nvcr.io** は nvidia-docker イメージの NVIDIA DGX コンテナー レジストリの場所です。

すべての **nvcr.io** Docker イメージでは、「latest (最新)」タグを使用した場合に発生するバージョンのあいまいさを回避するために、明示的なバージョン タグを使用します。たとえば、ローカルで latest とタグ付けされたバージョンのイメージによって、レジストリにある別の **latest** バージョンが上書きされる可能性があります。

特定のコンテナーに関する情報については、コンテナー内の **/workspace/ README.md** ファイルを参照してください。

NVIDIA DGX コンテナー レジストリをプルする前に、Docker をインストールする必要があります。Docker と nvidia-docker がインストールされていることを確認してください。詳細については、「NVIDIA コンテナーの使用準備 (英語)」を参照してください。

次のタスクの前提条件:

1. DGX-1 または DGX Station がネットワークに接続されている。
2. DGX-1 または DGX Station に Docker がインストールされている。
3. ブラウザーを使用して https://compute.nvidia.com (英語) にアクセスでき、NVIDIA NGC Cloud Services アカウントが有効になっている。
4. コンテナーを DGX-1 または DGX Station にプルする必要がある。
5. コンテナーをプライベート レジストリにプッシュする必要がある。
6. コンテナーを DGX-1 または DGX Station にプルして実行する必要がある。この手順を完了するには、システムのターミナル ウィンドウで SSH セッションを開く必要があります。

1. Web ブラウザーを開き、NVIDIA DGX コンテナー レジストリ (https://compute.nvidia.com、英語) にログインします。
2. 左のナビゲーションから、プルするコンテナーを選択します。

   例: **caffe** をクリック。
3. **[タグ]** セクションで、実行するリリースを探します。

   例: リリース 17.03 にカーソルを合わせる。
4. **[アクション]** 列で [ダウンロード] アイコンにカーソルを合わせます。[ダウンロード] アイコンをクリックして、`docker pull` コマンドを表示します。
5. `docker pull` コマンドをコピーし、**[閉じる]** をクリックします。
6. コマンド プロンプトを開き、次のコマンドを貼り付けます。

   ```
   docker pull
   ```

   コンテナー 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。

7. ローカル システムに Docker コンテナー ファイルを置いてから、コンテナーをローカルの Docker レジストリにロードします。

8. イメージがローカルの Docker レジストリにロードされていることを確認します。

3.3. NGC コンテナー レジストリからアクセスしてプルする

このセクションは、クラウド プロバイダー経由で NVIDIA NGC Cloud Services を使用している場合に該当します。

Amazon Web Services (AWS) などのクラウド サービス プロバイダーから NVIDIA コンテナーにアクセスしている場合は、最初に https://ngc.nvidia.com (英語) の NGC コンテナー レジストリでアカウントを作成する必要があります。アカウントの作成後にコンテナーをプルするコマンドは、データセンターに DGX-1 がある場合と同じです。ただし、現時点では、NGC コンテナー レジストリにコンテナーを保存することはできません。その代わりに、オンプレミスまたはクラウドにある自分の Docker レジストリにコンテナーを保存します。

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、NGC コンテナー レジストリにアクセスできます。NGC コンテナー レジストリにアクセスする場合は、DGX プラットフォーム以外も使用できます。インターネットにアクセスでき、Docker がインストールされている Linux コンピューターであれば、すべて使用できます。

NGC コンテナー レジストリにアクセスする前に、次の前提条件を満たしていることを確認します。要件の詳細については、「NGC 入門 (英語)」を参照してください。

- NVIDIA NGC Cloud Services アカウントがアクティブになっている。
- NGC コンテナー レジストリへのアクセスを認証するための NVIDIA NGC Cloud Services API キーを持っている。
- nvidia-docker コンテナーを実行可能な権限でクライアント コンピューターにログインしている。

NVIDIA NGC Cloud Services アカウントを有効にした後に、次の 2 つの方法のいずれかで NGC コンテナー レジストリにアクセスできます。

- Docker CLI を使用して NGC コンテナー レジストリからコンテナーをプルする
- NGC コンテナー レジストリ Web インターフェイスを使用してコンテナーをプルする

Docker レジストリは、Docker イメージを保存するサービスです。サービスは、インターネット、企業イントラネット、ローカル マシンに配置できます。たとえば、**nvcr.io** は nvidia-docker イメージの NGC コンテナー レジストリの場所です。

すべての **nvcr.io** Docker イメージでは、「latest (最新)」タグを使用した場合に発生するバージョンのあいまいさを回避するために、明示的なバージョン タグを使用します。たとえば、ローカルで「latest」とタグ付けされたバージョンのイメージによって、レジストリにある別の「latest」バージョンが上書きされる可能性があります。

1. NGC コンテナー レジストリにログインします。

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

2. ユーザー名のプロンプトが表示されたら、次のテキストを入力します。

```
$oauthtoken
```

3. **$oauthtoken** ユーザー名は、通常のユーザー名とパスワードではなく API キーで認証するための特殊なユーザー名です。

4. パスワードの入力を求められたら、次の例のように NVIDIA NGC Cloud Services API キーを入力します。

```
Username: $oauthtoken
Password: k7cqFTUvKKdiwGsPnWnyQFYGnlAlsCIRmlP67Qxa
```

ヒント API キーを取得したら、クリップボードにコピーします。これにより、パスワードの入力を求められたときにコマンド シェルに API キーを貼り付けることができます。

3.3.1. Docker CLI を使用して NGC コンテナー レジストリからコンテナーをプルする

このセクションは、クラウド プロバイダー経由で NVIDIA NGC Cloud Services を使用している場合に該当します。

nvidia-docker コンテナーをプルする前に、次の前提条件が満たされていることを確認してください。

- クラウドインスタンスシステムが「NVIDIA コンテナーの使用準備 (英語)」に従ってコンテナーを実行するようにセットアップされているか、「AWS で NGC を使用するためのガイド (英語)」に従って NGC AWS AMI を使用している。
- コンテナーを含むレジストリ空間に対する読み取りアクセス権を持っている。
- 「NGC コンテナー レジストリにアクセスしてプルする」の手順で NGC コンテナー レジストリにログインしている。
- Docker コマンドを使用できる **docker** グループのメンバーである。

ヒント NGC コンテナー レジストリで使用できるコンテナーを参照するには、Web ブラウザーを使用して NVIDIA NGC Cloud Services Web サイトの NGC コンテナー レジストリ アカウントにログインします。

1. レジストリからコンテナーをプルします。たとえば、NVCaffe™ 17.10 コンテナーをプルするには、次のようにします。

```
$ docker pull nvcr.io/nvidia/caffe:17.10
```

2. システム上の Docker イメージを一覧表示して、コンテナーがプルされていることを確認します。

```
$ docker images
```

コンテナーのプルが完了すると、科学技術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 分析などのジョブをコンテナー内で実行することができます。

3.3.2. NGC コンテナー レジストリ Web インターフェイスを使用してコンテナーをプルする

このセクションは、クラウド プロバイダー経由で NVIDIA NGC Cloud Services を使用している場合に該当します。

NGC コンテナー レジストリからコンテナーをプルする前に、Docker と **nvidia-docker** を「NVIDIA コンテナーの使用準備 (英語)」に記載されている場所にインストールする必要があります。また、「NGC 入門 (英語)」に記載されているとおりに、NGC コンテナー レジストリにアクセスしてログインする必要があります。

このタスクの前提条件:

1. クラウド インスタンス システムがインターネットに接続されている。
2. インスタンスに Docker と nvidia-docker がインストールされている。

3. ブラウザーを使用して https://ngc.nvidia.com (英語) の NGC コンテナー レジストリにアクセスでき、NVIDIA NGC Cloud Services アカウントが有効になっている。
4. コンテナーをクラウド インスタンスにプルする必要がある。

1. https://ngc.nvidia.com (英語) の NGC コンテナー レジストリにログインします。
2. 左ナビゲーションで、**[登録]** をクリックします。NGC コンテナー レジストリ ページを参照して、使用できる Docker リポジトリとタグを確認します。
3. リポジトリのいずれかをクリックして、そのコンテナー イメージの情報と、コンテナーを実行するときに使用できるタグを表示します。
4. **[プル]** 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 プロンプトを開いて、Docker pull コマンドを貼り付けます。コンテナー 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. ローカル システムに Docker コンテナー ファイルを置いてから、コンテナーをローカルの Docker レジストリにロードします。
7. イメージがローカルの Docker レジストリにロードされていることを確認します。

```
$ docker images
```

特定のコンテナーに関する情報については、コンテナー内の **/ workspace/README.md** ファイルを参照してください。

第 4 章
nvidia-docker イメージ

DGX-1 および NGC コンテナーは、`nvcr.io` と呼ばれる nvidia-docker リポジトリでホストされます。前のセクションで説明したように、これらのコンテナーはリポジトリから「プル」され、科学技術ワークロード、視覚化、ディープラーニングなどの GPU アクセラレーション アプリケーションに使用されます。

Docker イメージとは、開発者が構築するファイルシステムのことです。nvidia-docker イメージは、コンテナーのテンプレートとして、複数の層で構成されるソフトウェア スタックです。各階層は、スタック内の下の層に依存します。

Docker イメージからコンテナーが形成されます。コンテナーを作成するときは、書き込み可能な層をスタックの上に追加します。書き込み可能な層が追加された Docker イメージがコンテナーです。コンテナーとは、イメージの実行中のインスタンスのことを指します。コンテナーに対する変更や修正は、書き込み可能な層に対して加えられます。コンテナーは削除できますが、Docker イメージはそのまま残ります。

図 1 は、DGX-1 の nvidia-docker スタックです。nvidia-docker ツールがホスト OS と NVIDIA Driver の上にあることがわかります。このツールを使用して、nvidia-docker 層の上に NVIDIA コンテナーを作成します。コンテナーには、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-docker 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。

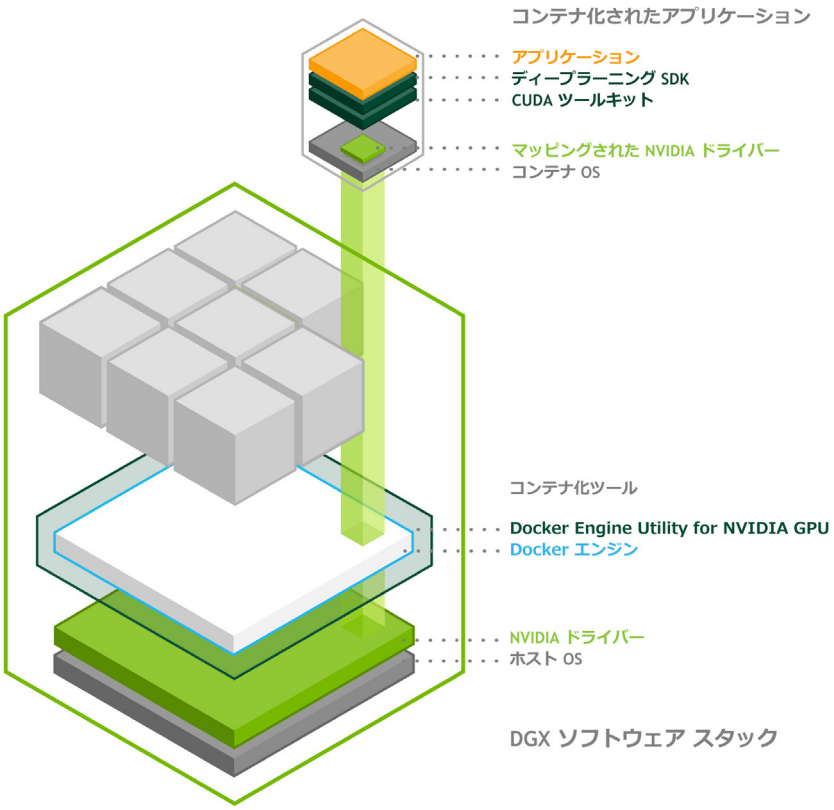

図 1: Docker コンテナーは、アプリケーションの依存関係をカプセル化して、実行の再現性と信頼性を提供します。nvidia-docker ユーティリティは、起動時に NVIDIA ドライバーのユーザーモード コンポーネントと GPU を Docker コンテナーにマウントします。

4.1. nvidia-docker イメージのバージョン

nvidia-docker イメージの各リリースは、バージョン「タグ」で特定されます。単純なイメージの場合、このバージョン タグには通常、イメージの主要なソフトウェア パッケージのバージョンが含まれます。複数のソフトウェア パッケージまたはバージョンが含まれる複雑なイメージの場合は、コンテナー化された単一のソフトウェア構成を表す個別のバージョンが使用されることがあります。一般的なスキームでは、イメージ リリースの年と月でバージョンを示します。たとえば、17.01 は、2017 年 1 月にリリースされたという意味です。

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。前の部分はリポジトリ内のコンテナー名で、後の部分はコンテナーに関連付けられている「タグ」です。この 2 つの情報を図 2 に示します。これは、**`docker images`** コマンドを実行して出力できます。

図 2: **docker images** コマンドの出力

図 2 は、次のようなイメージ名の簡単な例を示しています。

- **nvidia-cuda:8.0-devel**
- **ubuntu:latest**
- **nvcr.io/nvidia/tensorflow:17.01**

イメージにタグを追加しない場合、既定で「latest」というタグが追加されますが、すべての NGC コンテナーには明示的なバージョン タグが付けられます。

次のセクションでは、これらのイメージ名を使用してコンテナーを実行します。後のセクションでは、独自のコンテナー作成のほか、既存コンテナーのカスタマイズおよび拡張方法を紹介します。

第 5 章
コンテナーの実行

コンテナーを実行するには、レジストリ、リポジトリ、タグを指定して **nvidia-docker run** コマンドを実行します。

nvidia-docker ディープラーニング フレームワーク コンテナーを実行する前に、**nvidia-docker** をインストールする必要があります。詳細については、「NVIDIA コンテナーの使用準備 (英語)」を参照してください。

1. ユーザーとして、コンテナーをインタラクティブに実行します。

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

次の例では、インタラクティブ モードの NVCaffe コンテナーの 2016 年 12 月のリリース (16.12) を実行します。ユーザーがコンテナーを終了すると、コンテナーは自動的に削除されます。

```
$ nvidia-docker run --rm -ti nvcr.io/nvidia/caffe:16.12

============
== Caffe ==
============

NVIDIA Release 16.12 (build 6217)

Container image Copyright (c) 2016, 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.
root@df57eb8e0100:/workspace#
```

2. 実行するジョブをコンテナー内から起動します。

実行する正確なコマンドは、実行しているコンテナーの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。詳細については、コンテナーの **/ workspace/README.md** ファイルを参照してください。

次の例は、1 つの GPU で **caffe time** コマンドを実行して、**deploy.prototxt** モデルの実行時間を測定します。

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

3. オプション: 同じ NVCaffe コンテナーで、非インタラクティブ モードで 2016 年 12 月リリース (16.12) を実行します。

```
% nvidia-docker run --rm nvcr.io/nvidia/caffe:16.12 caffe time -model
    /workspace/models/bvlc _ alexnet -solver /workspace/deploy.prototxt -
gpu=0
```

5.1. nvidia-docker run

nvidia-docker run コマンドを実行する場合:

- Docker Engine は、ソフトウェアを実行するコンテナーにイメージをロードします。
- コマンドで使用する追加のフラグと設定を含めることによって、ランタイムのコンテナー リソースを定義します。フラグと設定については、次のセクションで説明します。
- GPU は、Docker コンテナーで明示的に定義されます (既定ですべての GPU を対象に、NV_GPU 環境変数で指定できます)。

5.2. ユーザーの指定

指定されない限り、コンテナー内のユーザーはルート ユーザーとなります。

コンテナー内で実行する場合、ルート ユーザーはホスト オペレーティング システムまたはネットワーク ボリューム上に作成されたファイルにアクセスできます。許可されていないユーザーは、コンテナー内のユーザーの ID を設定する必要があります。たとえば、コンテナー内のユーザーを現在実行しているユーザーとして設定するには、次のコマンドを実行します。

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

通常は、指定されたユーザーとグループがコンテナー内に存在しないため、警告が発生します。次のようなメッセージが表示されます。

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

この警告は原則として無視できます。

5.3. 削除フラグの設定

既定では、Docker コンテナーは実行後もシステム上に残ります。繰り返されるプル操作または実行操作は、コンテナーを終了した後でもローカル ディスク上の多くの領域を使用します。そのため、終了後に nvidia-docker コンテナーをクリーン アップすることが必要です。

コンテナーへの変更を保存する場合や実行後にジョブ ログにアクセスする場合は、**--rm** フラグを使用しないでください。

終了時にコンテナーを自動的に削除するには、**--rm** フラグを追加してコマンドを実行します。

```
% nvidia-docker run --rm nvcr.io/nvidia/<repository>:<tag>
```

5.4. インタラクティブ フラグの設定

既定では、コンテナーはバッチ モードで実行されます。このため、実行されたコンテナーはユーザーの操作なしで終了します。コンテナーは、サービスとしてインタラクティブ モードで実行することもできます。

インタラクティブ モードで実行するには、**-ti** フラグを実行コマンドに追加します。

```
% nvidia-docker run -ti --rm nvcr.io/nvidia/<repository>:<tag>
```

5.5. ボリューム フラグの設定

コンテナーにはデータセットは含まれないため、データセットを使用する場合は、ホスト オペレーティング システムからコンテナーにボリュームをマウントする必要があります。詳細については、「コンテナーでデータを管理する (英語)」を参照してください。

通常は、Docker ボリュームまたはホスト データ ボリュームを使用します。Docker ボリュームとの主な違いは、Docker ボリュームは Docker にプライベートで、Docker コンテナー間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 システムからは見えず、データ ストレージは Docker が管理します。ホスト データ ボリュームは、ホスト オペレーティング システムから使用できる任意のディレクトリです。ローカル ディスクまたはネットワーク ボリュームを使用できます。

例 1

ホスト オペレーティング システムのディレクトリ **/raid/imagedata** を **/images** としてコンテナーにマウントします。

```
% nvidia-docker run -ti --rm -v /raid/imagedata:/images
           nvcr.io/nvidia/<repository>:<tag>
```

例 2

コンテナーのローカルの Docker ボリューム **data** (存在しない場合は作成する必要があります) を **/imagedata** としてマウントします。

```
% nvidia-docker run -ti --rm -v data:/imagedata nvcr.io/nvidia/
<repository>:<tag>
```

5.6. マッピング ポート フラグの設定

ディープラーニング GPU トレーニング システム™ (DIGITS) などのアプリケーションは、通信用のポートを使用します。ローカル システム専用のポートを開く、またはローカル システム以外のネットワークに属する他のコンピューターが使用できるようにするよう制御できます。

たとえば、DIGITS では、コンテナー イメージ 16.12 の DIGITS 5.0 を起動すると、既定では DIGITS サーバーはポート 5000 を使用します。ただし、コンテナーを起動した後、コンテナーの IP アドレスがすぐにわからない場合があります。コンテナーの IP アドレスを確認するには、次の方法のいずれかを選択できます。

- ローカル システム ネットワーク スタックを使用してポートを公開する (**--net=host**)。ここでコンテナーのポート 5000 は、ローカル システムのポート 5000 として使用できます。

または、

- ポート (`-p 8080:5000`) をマッピングする。コンテナーのポート 5000 は、ローカル システムのポート 8080 として使用できます。

どちらの場合も、ローカル システム以外のユーザーには、DIGITS がコンテナー内で実行されていることが見えません。ポートを公開しないと、ホストからポートを使用することはできますが、外部からは使用できません。

5.7. 共有メモリ フラグの設定

PyTorch や Microsoft® Cognitive Toolkit™ などの特定のアプリケーションでは、共有メモリ バッファーを使用してプロセス間で通信します。共有メモリは、NVIDIA® Collective Communications Library™ (NCCL) を使用する MXNet™や TensorFlow™ などのシングルプロセスアプリケーションでも必要です。

既定では、Docker コンテナーには 64 MB の共有メモリが割り当てられています。これは、特に 8 個の GPU すべてを使用する場合、不十分です。1GB など、特定のサイズに共有メモリ制限を上げるには、`--shm-size=1g` フラグを `docker run` コマンドに含めます。

または、`--ipc=host` フラグを指定すると、コンテナー内部のホストの共有メモリを再利用できます。この後者の方法は、共有メモリ バッファーのデータが他のコンテナーから見えてしまうため、セキュリティの問題となります。

5.8. GPU フラグの公開制限の設定

コンテナーの中で、使用可能なすべての GPU を利用するためのスクリプトとソフトウェアを設定します。GPU の使用率を高いレベルで調整するには、このフラグを使用して、ホストからコンテナーへの GPU 公開を制限します。たとえば、GPU 0 と GPU 1 のみがコンテナーから見えるようにするには、次のコマンドを実行します。

```
$ NV _ GPU=0,1 nvidia-docker run...
```

このフラグは、使用する GPU を制限するための一時的な環境変数です。

Linux `cgroups` を基礎とした Docker のデバイスマッピング機能を使用すると、コンテナーごとに特定の GPU を定義できます。

5.9. コンテナーの寿命

`--rm` フラグを `nvidia-docker run` コマンドに渡さなければ、終了したコンテナーの状態は無期限に保存されます。次のコマンドを使用して、ディスクに保存されているすべての終了したコンテナーとサイズを一覧表示できます。

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

ディスク上のコンテナーのサイズは、コンテナーの実行中に作成されるファイルによって決まるため、終了したコンテナーのディスク容量は小さくなります。

次のコマンドを実行することで、終了したコンテナーを完全に削除できます。

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

終了後のコンテナーの状態を保存しておくと、次のような標準の Docker コマンドを使用して引き続き操作できます。

- **docker logs** コマンドを実行して、過去の実行のログを検証します。

```
$ docker logs 9489d47a054e
```

- **docker cp** コマンドを使用してファイルを抽出します。

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

- **docker restart** コマンドを使用して、停止しているコンテナーを再起動します。

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

NVCaffe™ コンテナーの場合は、次のコマンドを実行します。

```
$ docker restart caffe
```

- **docker commit** コマンドでは、新しいイメージを作成して変更を保存できます。詳細については、「例 3: docker commit を使用したコンテナーのカスタマイズ」を参照してください。

コンテナー使用中に作成されるデータ ファイルが最終イメージに追加されるため、Docker コンテナーを変更する際には注意が必要です。特に、コア ダンプ ファイルとログ ファイルによって最終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) (英語) は、DGX-1、DGX Station、NVIDIA NGC Cloud Services の 3 製品すべての NVIDIA Registry 領域を占める要素で、CUDA ツールキット、DIGITS ワークフロー、およびすべてのディープラーニング フレームワークが含まれています。

NVIDIA ディープラーニング SDK は、NVCaffe、Caffe2™、Microsoft Cognitive Toolkit、MXNet、PyTorch、TensorFlow、Theano™、Torch™ など、広く使用されるディープラーニング フレームワークを高速化します。

ソフトウェア スタックは、このようなフレームワークをシステムに最適化してコンテナー化したバージョンです。これらのフレームワークはすべての依存関係を含んでおり、事前構成、テスト、調整が実施されているため、そのまま実行できます。カスタムのディープラーニング ソリューションを構築する柔軟性を求めるユーザー向けに、各フレームワークのコンテナー イメージには、完全なソフトウェア開発スタックと共に変更および拡張が可能なソース コードが含まれています。

GX-1、DGX Station、NVIDIA NGC Cloud Services の 3 製品では、プラットフォーム ソフトウェアがサーバーにインストールされる最小限の OS とドライバーを中心に設計されており、nvidia-docker コンテナー内のすべてのアプリケーションと SDK ソフトウェアが Container Registry を通じてプロビジョニングされます。

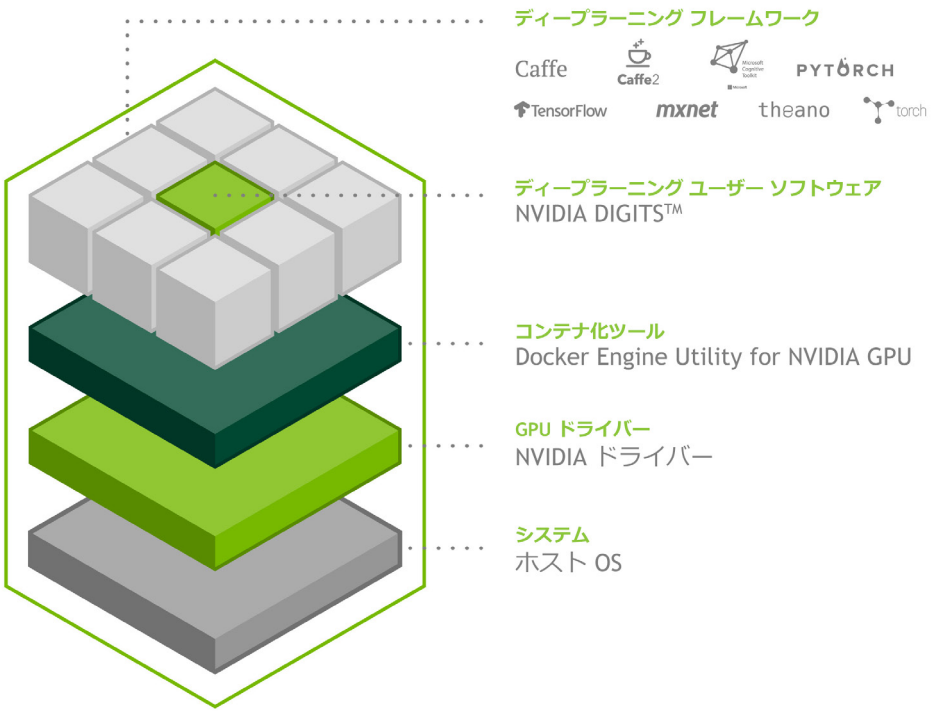

図 3: DGX-1 ディープラーニングソフトウェアスタック

すべての NGC コンテナー イメージは、プラットフォーム層 (**nvcr.io/nvidia/ cuda**) を基礎としています。この図は、他のすべての NGC コンテナーを支えるコンテナー化されたソフトウェア開発スタックのバージョンを表しており、カスタム アプリケーションでコンテナーを柔軟に構築したいユーザーに最適です。

6.1. OS 層

ソフトウェア スタック内で、最下部の層 (ベース層) は OS のユーザー スペースです。この層のソフトウェアには、リリース月に提供されるすべてのセキュリティ パッチが含まれます。

6.2. CUDA 層

CUDA® は、NVIDIA が開発したパラレル コンピューティング プラットフォームとプログラミング モデルで、アプリケーション開発者は GPU の大規模な並列処理機能を利用できます。CUDA は、ディープラーニングの GPU アクセラレーションをはじめ、天文学、分子動力学シミュレーション、金融工学など、さまざまな演算処理やメモリ負荷の高いアプリケーションの基盤となります。

6.2.1. CUDA ランタイム

CUDA ランタイム層は、展開環境での CUDA アプリケーション実行に必要なコンポーネントを提供します。CUDA ランタイムは、すべての共有ライブラリが CUDA ツールキットと共にパッケージ化されていますが、CUDA コンパイラ コンポーネントは含まれません。

6.2.2. CUDA ツールキット

CUDA ツールキットは、GPU アクセラレーション アプリケーションを最適化するための開発環境を提供します。CUDA ツールキットを使用すると、GPU が組み込まれたシステム、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、クラウドなどを対象としたアプリケーションを開発、最適化、展開できます。CUDA ツールキットには、アプリケーションを展開するためのデバッグと最適化のツール、コンパイラ、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 ネットワークの GPU アクセラレーション プリミティブを提供します。

CUDA® Basic Linear Algebra Subroutines ライブラリ™ (cuBLAS)

cuBLAS は、GPU での実行を大幅に高速化する完全な BLAS 標準ライブラリの GPU アクセラレーション バージョンです。cuBLAS GEMM (General Matrix-matrix Multiplication) ルーチンは、全面的に接続されている層のコンピューティングなど、ディープ ニューラル ネットワークで使用される主要なコンピューティングです。

6.3. ディープラーニング ライブラリ層

ここで紹介するライブラリは、NVIDIA GPU のディープラーニングに欠かせないものです。これらは、NVIDIA ディープラーニング SDK の一部です。

6.3.1. NCCL

NVIDIA® Collective Communications Library ™ (NCCL、発音「ニッケル」) は、トポロジ対応で簡単にアプリケーションに統合できるマルチ 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、協調して機能する多数のプロセッサによってデータを集計します。NCCL は、本格的な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重点を置いたライブラリです。現時点では、次の集合操作がサポートされています。

- `AllReduce`
- `Broadcast`
- `Reduce`
- `AllGather`
- `ReduceScatter`

通信プロセッサ間の緊密な同期は、集合通信の主要な側面です。従来、CUDA ベースの集合は、ローカル リダクション用の CUDA メモリ コピー操作と CUDA カーネルによって実現していました。NCCL はこれとは異なり、通信と演算の両方の処理を扱う 1 つのカーネルに各集合を実装します。これにより、同期が高速化し、ピーク帯域幅に到達するまでに必要なリソースを最小限に抑えられます。

NCCL は、開発時に特定のマシン向けにアプリケーションを最適化する必要がない点が便利です。NCCL は、ノード内、ノード間の両方の複数の GPU を短時間で収集します。PCIe、NVLink™、InfiniBand Verbs、IP ソケットなどの相互接続技術をサポートしています。NCCL は、通信戦略を自動的にパターン化して、システムの基盤である GPU の相互接続技トポロジと一致させます。

NCCL の設計では、パフォーマンスだけでなくプログラミングが容易であることも重視されています。NCCL はシンプルな C API を採用しています。これは、さまざまなプログラミング言語で容易に利用できます。NCCL は、MPI (Message Passing Interface) によって定義される一般的な集合 API に厳密に従っています。MPI に慣れているユーザーならば、NCCL の API をごく自然に使用できます。MPI との小さな違いは、NCCL 集合は「ストリーム」引数を受け取り CUDA プログラミング モデルと直接統合できるという点です。

最後に、NCCL は、次の例のようなほぼすべてのマルチ GPU 並列化モデルと互換性があります。

- シングルスレッド
- マルチスレッド。例: GPU ごとに 1 つのスレッドを使用
- マルチプロセス。例: GPU のマルチスレッド操作と組み合わせた MPI

NCCL にとって、ディープラーニング フレームワークは大きな応用領域です。ディープラーニング フレームワークでは、`AllReduce` 集合がニューラル ネットワーク トレーニングのために多用されます。ニューラル ネットワーク トレーニングは、NCCL が提供するマルチ GPU とマルチ ノード通信によって効率的にスケーリングできます。

6.3.2. cuDNN

CUDA® Deep Neural Network library™ (cuDNN) には、フォワード畳み込み、バックワード畳み込み、プール、正規化、アクティベーション層などの標準ルーチンが高度に調整されて実装されています。

フレームワークの速度はそれぞれ異なります。また、cuDNN ライブラリは旧バージョンと互換性がないため、固有のコンテナー内のみで有効です。そのため、複数の CUDA および cuDNN コンテナーを使用できますが、フレームワーク固有のタグを Dockerfile 内で指定する必要があります。

6.4. フレームワーク コンテナー

フレームワーク層には、個々のディープラーニング フレームワークのすべての要件が含まれます。この層の主な目標は、基本的な作業フレームワークを提供することです。フレームワークは、プラットフォーム コンテナー層の仕様でさらにカスタマイズできます。

フレームワーク層内では、次のような操作を選択できます。

- NVIDIA が提供するフレームワークをそのまま実行する。これは、フレームワークが構築され、コンテナー イメージ内で実行する準備ができている場合です。
- NVIDIA が提供するフレームワークに基づき、変更を加える。これは、コンテナー内で NVIDIA のコンテナー イメージから開始し、変更を適用して、再コンパイルする場合です。
- 目的のアプリケーションを NVIDIA が提供する CUDA、cuDNN、NCCL の各層の上にゼロから構築する。

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 コンテナーについて説明します。

第 7 章
NVIDIA ディープラーニング フレームワーク コンテナー

ディープラーニング フレームワークは、複数の層で構成されるソフトウェア スタックの一部です。各階層は、スタック内でその下にある層に依存します。このソフトウェア アーキテクチャには、次のような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ーに格納されるため、libc、cuDNN など、さまざまなバージョンのライブラリを使用でき、互いに干渉することはありません。
- ユーザーに応じたエクスペリエンスをターゲットにできる。
- ディープラーニング フレームワークのパフォーマンスの向上や、バグ修正があれば、コンテナーの新しいバージョンをレジストリに提供する。
- システムの維持が容易で、アプリケーションが OS に直接インストールされないため、OS のイメージをクリーンに保つことができる。
- セキュリティ更新、ドライバー更新、および OS のパッチをシームレスに提供する。

次のセクションでは、`nvcr.io` に含まれるフレームワークのコンテナーについて説明します。

7.1. フレームワークを使用するメリット

フレームワークは、ディープラーニングの研究と応用のアクセス性と効率を高めることを目的に作成されました。フレームワークを使用する主なメリットは次のとおりです。

- フレームワークで提供される高度に最適化された GPU 対応コードは、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計算に特化している。
- NVIDIA フレームワークは、最大限の GPU パフォーマンスが得られるように調整、テストされている。
- 簡単なコマンド ラインや Python などのスクリプト言語インターフェイスを使用してコードにアクセスできる。
- GPU 用コードや複雑なコンパイル済みコードを作成しなくても強力な DNN を多数トレーニングして実装できるだけでなく、GPU アクセラレーションによってトレーニングの高速化できる。

7.2. NVCaffe

Caffe™ (英語) は、柔軟性、速度、およびモジュール性を念頭に置いて作成されたディープラーニング フレームワークです。Caffe は当初、Berkeley Vision and Learning Center (BVLC) およびコミュニティの参加者によって開発されました。

NVCaffe は、NVIDIA が管理している BVLC Caffe のフォークで、NVIDIA GPU (特にマルチ GPU 構成) 向けに調整されています。

NVCaffe に対する最適化と変更の詳細については、「ディープラーニング フレームワークのリリース ノート (英語)」を参照してください。

7.3. Caffe2

Caffe2 (英語) は、畳み込みニューラル ネットワーク (CNN)、リカレント ニューラル ネットワーク (RNN) などのモデル タイプを、使いやすい Python ベースの API で簡単に表現し、高度に効率的な C++ と CUDA バックエンドを使用して実行するように設計されたディープラーニング フレームワークです。

Caffe2 の柔軟な API により、ユーザーは表現力の高いハイレベルな操作で、推論やトレーニング用のモデルを定義できます。Python インターフェイスを使用すると、推論やトレーニングのプロセスを簡単に制御して視覚化できます。

Caffe2 は、シングル GPU とマルチ GPU の実行をサポートし、マルチノードの実行もサポートします。

Caffe2 に対する最適化と変更の詳細については、「ディープラーニング フレームワークのリリース ノート (英語)」を参照してください。

7.4. Microsoft Cognitive Toolkit

Microsoft Cognitive Toolkit (英語) (旧称 CNTK) は、統合型ディープラーニング ツールキットで、フィードフォワード ディープ ニューラル ネットワーク (DNN)、CNN、RNN などの一般的なモデル タイプを簡単に実現して組み合わせることができます。

Microsoft Cognitive Toolkit は、自動の差別化と並列化によって、複数の GPU とサーバーに確率的勾配降下法 (SGD: Stochastic Gradient Descent) ラーニングを実装します。Microsoft Cognitive Toolkit は、Python アプリケーションまたは C++ アプリケーションからライブラリとして呼び出すか、または BrainScript モデルの記述言語を使用してスタンドアロン ツールとして実行できます。

NVIDIA と Microsoft は、DGX システム、Azure N シリーズの仮想マシンなどの GPU ベースのシステム上で Microsoft Cognitive Toolkit の使用を加速するために、緊密に連携してきました。この組み合わせは、1 つのフレームワークの中で、まず DGX-1 を使用してオンプレミスでモデルのトレーニングを行い、その後 Microsoft Azure クラウドでモデルをさらに大規模に展開できるため、スタートアップまたは大企業を問わずきわめて優れた操作性と拡張性が得られます。

Microsoft Cognitive Toolkit に対する最適化と変更の詳細については、「ディープラーニング フレームワークのリリース ノート (英語)」を参照してください。

7.5. MXNet

MXNet (英語) は、効率と柔軟性の両方を考慮して設計されたディープラーニング フレームワークであり、記号プログラミングと命令型プログラミングを組み合わせて効率と生産性を最大化します。

MXNet の中核には、記号と命令の両方の演算を自動的に並列化して迅速に処理する動的な依存関係スケジューラがあります。スケジューラの上に構築されるグラフ最適化層が記号演算の実行を高速化し、メモリを効率化します。MXNet は移植性を備え、軽量で、複数の GPU と複数のマシンにスケーリングできます。

MXNet に対する最適化と変更の詳細については、「ディープラーニング フレームワークのリリース ノート (英語)」を参照してください。

7.6. TensorFlow

TensorFlow™ (英語) は、データフローグラフを使用する数値演算のためのオープンソース ソフトウェア ライブラリです。グラフのノードは数学的演算を表し、グラフのエッジはその間を流れる多次元データ配列 (テンソル) を表します。この柔軟なアーキテクチャにより、コードを書き直すことなく、デスクトップ、サーバー、またはモバイル デバイスの 1 つ以上の CPU または GPU に計算を展開できます。

TensorFlow は当初、機械学習とディープ ニューラル ネットワークの研究を行うために、Google の Machine Intelligence 研究組織内の Google Brain チームによって開発されました。このシステムは、他のさまざまなドメインにも適用できるよう一般化されています。

TensorFlow の結果を視覚化するために、この Docker イメージにも TensorBoard (英語) が含まれます。TensorBoard は、視覚化ツールのスイートです。たとえば、トレーニング履歴やモデルの外観を表示できます。

TensorFlow に対する最適化と変更の詳細については、「ディープラーニング フレームワークのリリース ノート (英語)」を参照してください。

7.7. Theano

Theano (英語) は、多次元配列を含む数式を効率的に定義、最適化、および評価するための Python ライブラリです。Theano は、2007 年から大規模で計算量が膨大な科学調査に利用されています。

Theano に対する最適化と変更の詳細については、「ディープラーニング フレームワークのリリース ノート (英語)」を参照してください。

7.8. Torch

Torch (英語) は、幅広いディープラーニング アルゴリズムをサポートする科学計算フレームワークです。Torch は、Lua という簡単で高速なスクリプト言語と、その基盤である C/CUDA の実装のために優れた使いやすさと効率性を備えています。

Torch に含まれるニューラル ネットワークと最適化の一般的なライブラリは、簡単に使用できるにもかかわらず、複雑なニューラル ネットワーク トポロジの構築にきわめて柔軟に対応できます。

Torch に対する最適化と変更の詳細については、「ディープラーニング フレームワークのリリース ノート (英語)」を参照してください。

7.9. PyTorch

PyTorch (英語) は、次の 2 つのハイレベルな機能を提供する Python パッケージです。

- 強力な GPU アクセラレーションによるテンソル計算 (NumPy など)
- テープベースの Autograd システムに基づいたディープ ニューラル ネットワーク

必要に応じて、NumPy、SciPy、Cython などの使い慣れた Python パッケージを再利用して PyTorch を拡張できます。

PyTorch に対する最適化と変更の詳細については、「ディープラーニング フレームワークのリリース ノート (英語)」を参照してください。

7.10. DIGITS

Deep Learning GPU Training System™ (DIGITS、英語) は、ディープラーニング機能を技術者とデータ サイエンティストに提供します。

DIGITS はフレームワークではなく、NVCaffe、Torch、および TensorFlow のラッパーであり、コマンド ラインで直接処理する代わりに、グラフィカル Web インターフェイスを提供します。

DIGITS を使用すると、イメージ分類、セグメンテーション、オブジェクト検出の各タスクにおいて、高精度なディープ ニューラル ネットワーク (DNN) を高速でトレーニングすることができます。DIGITS により、データの管理、マルチ GPU システムにおけるニューラル ネットワークの設計とトレーニング、高度な視覚化によるパフォーマンスのリアルタイム監視、展開のために結果ブラウザーから最高のパフォーマンスのモデルを選択するなどの一般的なディープラーニング タスクを簡素化できます。DIGITS は完全にインタラクティブなシステムであるため、データ サイエンティストはプログラミングやデバッグではなく、ネットワークの設計とトレーニングに専念できます。

DIGITS の最適化と変更の詳細については、「DIGITS のリリース ノート (英語)」を参照してください。

第 8 章
HPC および HPC 視覚化コンテナー

HPC 視覚化コンテナー

NVIDIA に最適化されたフレームワーク (英語)と HPC コンテナーへのアクセスに加えて、NVIDIA GPU Cloud (NGC) コンテナー レジストリは、HPC 用の科学技術視覚化コンテナーもホストします。これらのコンテナーは、ParaView (英語) という一般的な科学技術視覚化ツールに依存します。

HPC 環境においては、一般的にリモート視覚化が必要です。データはリモート HPC システムまたはクラウドに配置されて処理され、ユーザーはワークステーションからこのアプリケーションとグラフィカルに対話します。一部の視覚化コンテナーには特別なクライアント アプリケーションが必要になるため、HPC 視覚化コンテナーは次の 2 つのコンポーネントで構成されます。

サーバー コンテナー

サーバー コンテナーは、サーバー システム上のファイルにアクセスできる必要があります。このアクセス許可の方法は、この後で説明しています。サーバー コンテナーは、シリアル モードとパラレル モードのどちらでも実行できます。このアルファ リリースでは、シリアル ノード構成が中心となっています。

パラレル構成が必要な場合は、hpcviscontainer@nvidia.com にお問い合わせください。

クライアント コンテナー

クライアント アプリケーションとサーバー コンテナーのバージョンを一致させるために、NVIDIA はクライアント アプリケーションをコンテナーとして提供します。サーバー コンテナーと同様に、サーバー コンテナーとの接続を確立するためにポートにアクセスする必要があります。

さらに、クライアント コンテナーは、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 サーバーへのアクセスも必要です。

視覚化製品または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 コンテナーにマッピングすることをお勧めします。さらに、クライアントとサーバー コンテナーの接続はオープンである必要があります。

利用可能な HPC 視覚化コンテナーのリストおよびその使用方法の手順については、「NGC コンテナー ユーザー ガイド (英語)」を参照してください。

第 9 章
コンテナー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、調整済みでパッケージ化されているため、すぐに実行することができます。さらに、企業のインフラに合わせて新しいイメージを構築したり、カスタム コード、ライブラリ、データ、設定を使用して既存のイメージを補強したりすることもできます。このセクションでは、新規コンテナーの作成、コンテナーのカスタマイズ、ディープラーニング フレームワークの拡張による機能追加、開発環境の拡張フレームワークを使用したコード開発、バージョン リリース用のパッケージ化などについて、演習を交えながら順に説明します。

既定では、コンテナー構築の必要はありません。DGX と NGC のどちらのコンテナー レジストリ (`nvcr.io`) にも、すぐに使用できる一連のコンテナーがあります。これらは、ディープラーニング、科学計算、視覚化などの専用コンテナーと、CUDA Toolkit のみを含むコンテナーです。

コンテナーの優れている点は、新規コンテナーを作成時の土台として使用できることです。これをコンテナーの「カスタマイズ」または「拡張」と呼びます。完全にゼロから作成することもできますが、これらのコンテナーは通常、GPU システムで動作するため、少なくとも OS と CUDA を含む `nvcr.io` コンテナーから開始することをお勧めします。ただし、これに制限されることなく、GPU の代わりに CPU で動作するコンテナーを作成することもできます。その場合は、Docker の OS のみを含むコンテナーから開始できます。または、開発を簡素化するために、CUDA を含むコンテナーで CUDA を使用しないという方法もあります。

DGX-1 と DGX Station の場合は、変更または拡張されたコンテナーを NVIDIA DGX コンテナー レジストリ (`nvcr.io`) にプッシュまたは保存できます。これらを DGX システムの他のユーザーと共有することもできますが、管理者の協力が必要です。

現時点では、カスタマイズされたコンテナーを NGC コンテナー レジストリ (クラウドベース) ソリューションから `nvcr.io` に保存することはできません。カスタマイズまたは拡張されたコンテナーは、ユーザーのプライベートコンテナーリポジトリに保存できます。カスタマイズまたは拡張されたコンテナーは、ユーザーのプライベートコンテナーリポジトリに保存できます。

すべての nvidia-docker ディープラーニング フレームワーク イメージには、フレームワーク自体を構築するためのソースおよびすべての前提条件が含まれていることに注意してください。

注意: Docker の構築時に 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。nvidia-docker は本質的に、コードを GPU で実行するために必要なコンポーネントを使用してコンテナーを透過的にプロビジョニングする docker のラッパーです。

9.1. コンテナーのカスタマイズ

NVIDIA は、テストと調整が済み、すぐに実行できる多数のイメージを NGC コンテナー レジストリを通じて提供しています。任意のイメージをプルしてコンテナーを作成し、選択したソフトウェアやデータを追加できます。

ベスト プラクティスは、新規 docker イメージの開発に **`docker commit`** を使用せず、代わりに Dockerfile を使用することです。この方法では、docker イメージの開発中に行われる変更を効率的にバージョン管理するための可視性と機能が得られます。**`docker commit`** は、短期の破棄可能なイメージのみに適しています (例として、「例 3: docker を使用したコンテナーのカスタマイズ」を参照)。

すべての nvidia-docker ディープラーニング フレームワーク イメージには、フレームワーク自体を構築するためのソースおよびすべての前提条件が含まれていることに注意してください。

注意: Docker の構築時に 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。nvidia-docker は本質的に、コードを GPU で実行するために必要なコンポーネントを使用してコンテナーを透過的にプロビジョニングする docker のラッパーです。

Docker ファイルの記述の詳細については、「Dockerfile の記述のベスト プラクティス (英語)」を参照してください。

9.1.1. コンテナーをカスタマイズするメリットと制約

コンテナーは、個々のニーズに合わせてカスタマイズできます。たとえば、依存する特定のソフトウェアが NVIDIA の提供しているコンテナーに含まれない場合などです。どのような理由でも、コンテナーをカスタマイズすることは可能です。

フレームワークのソースに含まれている場合を除いて、コンテナー イメージにはサンプル データセットやサンプル モデル定義は含まれません。コンテナーにサンプル データセットやモデルが含まれているかどうかを必ず確認してください。

9.1.2. 例 1: 新規コンテナーを構築する

Docker は、Dockerfile を使用して Docker イメージを作成または構築します。Dockerfile は、新規 docker イメージを作成するために Docker が順に実行するコマンドを含むスクリプトです。コンテナー イメージのソース コードとも言えます。Dockerfile は、ベース イメージを基に作成します。

詳細については、「Dockerfile の記述のベスト プラクティス (英語)」を参照してください。

1. ローカル ハード ドライブに作業ディレクトリを作成します。
2. そのディレクトリ内で、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。ファイルを作業ディレクトリに保存します。
3. **`Dockerfile`** を開き、次の内容を記述します。

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install -y curl
CMD echo "hello from inside a container"
```

最後の行の **`CMD`** で、コンテナーを作成するときに指定したコマンドが実行されます。これは、コンテナーが正しく構築されたかどうかを確認する 1 つの方法です。

この例では、DGX™ システム リポジトリではなく Docker リポジトリからもコンテナーをプルしています。この後に、NVIDIA® リポジトリを使用するいくつかの例を紹介します。

4. **Dockerfile** を保存して閉じます。

5. イメージを構築します。イメージを構築してタグを作成するために、次のコマンドを実行します。

```
$ docker build -t <new _ image _ name>:<new _ tag> .
```

このコマンドは、Dockerfile と同じディレクトリで実行されました。

各行ごとに docker 構築プロセスの「ステップ」のリストが出力されます。

たとえば、コンテナーに **test1** という名前を付け、**latest** とタグ付けします。説明用にプライベート DGX システム リポジトリの名前を **nvidian_sas** としています。次のコマンドでコンテナーが構築され、出力内容を確認できます。

```
$ docker build -t test1:latest .
Sending build context to Docker daemon 3.072 kB
Step 1/3 : FROM ubuntu:14.04
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 1f491b9235d8
---> 934785072daf
Removing intermediate container 1f491b9235d8
Successfully built 934785072daf
```

イメージの構築の詳細については、「docker build」を参照してください。イメージのタグ付けの詳細については、「docker tag」を参照してください。

6. 正常に構築されたことを確認するために、メッセージが表示されます。

```
Successfully built 934785072daf
```

このメッセージは、正常に構築されたことを示します。これ以外のメッセージの場合は、正常に構築されなかったことを示します。

イメージが構築され、ランダムの場合は番号 **934785072daf** が割り当てられます。

7. イメージを表示できることを確認します。次のコマンドを実行して、コンテナーを表示します。

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

これで、新しいコンテナーが利用できるようになりました。

コンテナーは、この DGX システムにローカルです。コンテナーをプライベート リポジトリに格納する場合は、次の手順に従います。

8. コンテナーをプッシュしてプライベート Docker リポジトリに格納します。

これは、DGX-1™ と DGX Station のみで動作します。

a) プッシュの最初の手順は、タグ付けです。

```
$ docker tag test1 nvcr.io/nvidian _ sas/test1:latest
```

b) イメージにタグが付けられます。ここでは、**nvcr.io** の **nvidian_sas** というプライベート プロジェクトにプッシュします。

```
$ docker push nvcr.io/nvidian _ sas/test1:latest
The push refers to a repository [nvcr.io/nvidian _ sas/test1]
…
```

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

9.1.3. 例 2: Dockerfile を使用したコンテナーのカスタマイズ

この例では、Dockerfile を使用して **nvcr.io** の NVCaffe コンテナーをカスタマイズします。コンテナーをカスタマイズする前に、**docker pull** コマンドを使用して NVCaffe 17.03 コンテナーをレジストリにロードしておく必要があります。

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

本書の最初に説明したように、**nvcr.io** の Docker コンテナーにもサンプル Dockerfile があります。このサンプルでは、フレームワークを変更して Docker イメージを再構築する方法を説明しています。**/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、コンテナーをカスタマイズするためのテンプレートとして、**Dockerfile.customcaffe** ファイルを使用します。

1. ローカル ハード ドライブに **my_docker_images** という作業ディレクトリを作成します。
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。ファイルを作業ディレクトリに保存します。
3. 再度 **Dockerfile** を開き、ファイルに次の行を含めます。

```
FROM nvcr.io/nvidia/caffe:17.03
# APPLY CUSTOMER PATCHES TO CAFFE
# Bring in changes from outside container to /tmp
# (assumes my-caffe-modifications.patch is in same directory as
Dockerfile)
#COPY my-caffe-modifications.patch /tmp

# Change working directory to NVCaffe source path
WORKDIR /opt/caffe

# Apply modifications
#RUN patch -p1 < /tmp/my-caffe-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
```

ファイルを保存します。

4. **docker build** コマンドを使用してイメージを構築し、リポジトリ名とタグを指定します。次の例では、リポジトリ名は *corp/caffe* で、タグは *17.03.1PlusChanges* です。この場合のコマンドは次のとおりです。

```
$ docker build -t corp/caffe:17.03.1PlusChanges .
```

5. **nvidia-docker run** コマンドを使用して、Docker イメージを実行します。たとえば、次のようにします。

```
$ nvidia-docker run -ti --rm corp/caffe:17.03.1PlusChanges .
```

9.1.4. 例 3: docker commit を使用したコンテナーのカスタマイズ

この例では、`docker commit` コマンドを使用して、コンテナーの現在の状態を 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、実行中のコンテナーを変更して保存する必要がある場合に役立ちます。この例では、**apt-get** タグを使用して、ユーザーがルートで実行する必要があるパッケージをインストールします。

- 説明のために、サンプルの手順では NVCaffe イメージのリリース 17.04 を使用しています。
- コンテナーを実行する際に、**--rm** フラグは使用しないでください。コンテナーを実行する際に **--rm** フラグを使用すると、コンテナーの終了と共に変更内容が失われます。

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

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

2. **nvidia-docker** を使用して、DGX システムでコンテナーを実行します。

```
$ nvidia-docker run -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 64 MB. This may be
 insufficient for NVIDIA Caffe. NVIDIA recommends the use of the following
 flags:
   nvidia-docker run --shm-size=1g --ulimit memlock=-1 --ulimit
 stack=67108864...

root@1fe228556a97:/workspace#
```

3. これで、コンテナーのルート ユーザーになります (プロンプトを確認してください)。**apt** コマンドを使用すると、パッケージをプルしてコンテナーに格納できます。

NVIDIA コンテナーは、Ubuntu で **apt-get** パッケージ マネージャーを使用することによって構築されます。使用している個々のコンテナーの詳細については、コンテナーのリリース ノート「ディープラーニングのドキュメント (英語)」を確認してください。

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

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

apt を使用して Octave をインストールする前に、まず **apt-get update** を実行する必要があります。

4. ワークスペースを終了します。

```
# exit
```

5. **docker ps -a** を使用してコンテナーのリストを表示します。例として、次に **docker ps -a** コマンドの出力を示します。

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

6. これで、Octave をインストールした場所で実行されるコンテナーから、新しいイメージを作成できます。次のコマンドを使用してコンテナーをコミットします。

```
$ docker commit 1fe228556a97 nvcr.io/nvidian _ sas/caffe _ octave:17.04
sha256:0248470f46e22af7e6cd90b65fdee6b4c6362d08779a0bc84f45de53a6ce9294
```

7. 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   ...
```

8. 検証のために、コンテナーを再度実行して、Octave が意図した場所に実際にあることを確認します。

これは、DGX-1 と DGX Station のみで動作します。

```
$ nvidia-docker run -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 64 MB. This may be
 insufficient for NVIDIA Caffe. NVIDIA recommends the use of the following
 flags:
   nvidia-docker run --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ているため、Octave はインストールされています。

9. コンテナーを各自のプライベート リポジトリに保存する場合は (Docker では「プッシュ」と呼ばれます)、**docker push...** コマンドを使用できます。

```
$ docker push nvcr.io/nvidian _ sas/caffe _ octave:17.04
```

新しい docker イメージが利用可能になりました。ローカルの Docker リポジトリで確認できます。

9.1.5. 例 4: Docker を使用したコンテナーの開発

開発者がコンテナーを拡張するユース ケースには、主に次の 2 つがあります。

1. プロジェクトのすべての不変の依存関係を含むが、ソース コード自体は含まない開発イメージを作成する。
2. ソースの最終版とソフトウェアのすべての依存関係を含む運用イメージまたはテスト イメージを作成する。

データセットは、コンテナー イメージに含まれていません。コンテナー イメージは、データセットと結果のボリューム マウントを想定して設計するのが理想的です。

これらの例では、ローカル データセットをホストの **/raid/datasets** から **/dataset** に、コンテナー内の読み取り専用ボリュームとしてマウントします。実行中の出力を取得するために、ジョブ固有のディレクトリもマウントします。

これらの例では、コンテナー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 **/output** にマッピングします。この方法では、順に起動したコンテナーの出力を個別に取得できます。

開発やモデルの反復処理のためにコンテナーにソースを格納すると、ワークフロー全体が複雑化します。たとえば、ソース コードをコンテナー内に置く場合、エディター、バージョン管理ソフトウェア、dotfile などもコンテナーに格納する必要があります。

しかし、ソース コードの実行に必要なすべてを含む開発イメージを作成しておけば、ソース コードをコンテナーにマッピングして、ホスト ワークステーションの開発環境を使用できます。モデルの最終版を共有するには、バージョン管理されたソース コードのコピーと、開発環境でトレーニングされた重みとをパッケージ化するのが便利です。

例として、Isola などによる「Conditional Adversarial Networks によるイメージ間の変換 (英語)」の作業をオープンソースに実装するための開発と提供の例を挙げます。これは pix2pix (英語) で利用できます。Pix2Pix は、Conditional Adversarial Network を使用する入力イメージから出力イメージへのマッピングを学習する Torch 実装です。オンライン プロジェクトは時間経過と共に変化しますが、ここでは、スナップショット バージョンの **d7e7b8b557229e75140cbe42b7f5dbf85a67d097 change-set** に合わせます。

このセクションでは、コンテナーを仮想環境として使用し、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をコンテナーに含めます。

ネットワーク定義とトレーニング スクリプトは、コンテナー イメージとは別に保管されています。実際に実行されるファイルはホストに永続的に保存され、実行時のみにコンテナーにマッピングされるため、このモデルは反復型の開発に効果的です。

元のプロジェクトとの違いは、「変更の比較 (英語)」で見ることができます。

開発に使用するマシンが長期間のトレーニング セッションを実行するマシンと異なる場合は、開発中の状態をコンテナーにパッケージ化することもできます。

1. ローカル ハード ドライブに作業ディレクトリを作成します。

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

2. git によって 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** という名前でファイルを作成し、次の行を追加します。

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

6. 開発用 Docker コンテナー イメージ (**build-devel.sh**) を構築します。

```
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
```

実際の開発では、ホストでファイルへの変更を繰り返し、コンテナー内でトレーニング スクリプトを実行します。

8. オプション: ファイルを編集し、変更時に次の手順を実行します。

9. トレーニング スクリプト (**run-devel.sh**) を実行します。

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

9.1.5.1. 例 4.1: ソースをコンテナーにパッケージ化する

モデル定義とスクリプトをコンテナーにパッケージ化するのは非常に簡単で、**COPY** ステップを Dockerfile に追加するだけです。

ボリューム マウントをドロップし、コンテナー内にパッケージ化されているソースを使用するように実行スクリプトを更新しました。内部コードが修正されたため、パッケージ化されたコンテナーは **devel** コンテナー イメージよりもはるかに移植しやすくなりました。このコンテナー イメージのバージョンは、特定のタグを使用して管理し、コンテナー レジストリに格納することをお勧めします。

コンテナーを実行する更新も、ローカル ソースのコンテナーへボリューム マウントをドロップするだけで済みます。

9.2. フレームワークのカスタマイズ

各 Docker イメージには、フレームワーク自体を変更できるようにするためのフレームワークの構築用コードが含まれます。各イメージのフレームワーク ソースは **/workspace** ディレクトリにあります。

9.2.1. フレームワークをカスタマイズするメリットと制約

フレームワークのカスタマイズは、NVIDIA リポジトリの外部でフレームワークに適用するパッチまたは変更がある場合、またはフレームワークに追加する特別なパッチがある場合に役立ちます。

これは、DGX-1 と DGX Station のみに該当します。

9.2.2. 例 1: フレームワークのカスタマイズ

この例では、フレームワークをカスタマイズしてコンテナーを再構築する方法を説明します。

ここでは、NVCaffe 17.03 フレームワークを使用します。

ネットワーク層の作成時に、NVCaffe フレームワークは次の出力メッセージを **stdout** に返すように設定されています。

```
"Creating Layer"
```

これは、NVCaffe 17.03 コンテナーで bash シェルのコマンドを実行した際に出力される結果です。

```
# 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
…
```

次のステップで、NVCaffe の **"Created Layer"** というメッセージを **"Just Created Layer"** に変更する方法を示します。この例では、既存のフレームワークを変更しています。

フレームワーク コンテナーをインタラクティブ モードで実行していることを確認してください。

1. **nvcr.io** リポジトリから NVCaffe 17.03 コンテナーを探します。

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

2. DGX システムでコンテナーを実行します。

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

   

   これにより、コンテナーのルート ユーザーになります。プロンプトが変化されます。

3. NVCaffe のソース ファイル **/opt/caffe/src/caffe/net.cpp** を編集します。変更するのは **162** 行目です。

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

   

   vi を使用して、**"Created Layer"** を **"Just Created Layer"** に変更します。

4. NVCaffe を再構築します。次の手順に従って完了します。

   ```
   # 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
   ```

5. 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にあることを確認します。ここでは **/usr/local/** です。

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

6. NVCaffe を実行し、**stdout** の出力が変わっ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
...
```

7. **nvcr.io** のプライベート DGX リポジトリまたはプライベート Docker リポジトリにコンテナーを保存します (「例 2: Dockerfile を使用したコンテナーのカスタマイズ」を参照)。

これは、DGX-1 と DGX Station のみで動作します。

第 10 章
トラブルシューティング

nvidia-docker のコンテナーの詳細については、GitHub サイト (NVIDIADocker GitHub (英語)) をご覧ください。

ディープラーニング フレームワークのリリース ノートおよびその他の製品資料については、ディープラーニング ドキュメントの Web サイト (「ディープラーニング フレームワークのリリース ノート (英語)」) をご覧ください。

通知

このガイドの情報およびこのガイドで参照する NVIDIA ドキュメントに含まれる他のすべての情報は、「現状有姿」で提供されます。NVIDIA は製品に関する情報について、明示または黙示、あるいは法定または非法定にかかわらず保証しません。さらに、特定の目的に対する黙示的保証、非抵触行為、商品性、および適正すべてに対する責任を明示的に否認します。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 がこのガイド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、本製品の販売に関する NVIDIA の契約条件に従って制限されるものとします。

このガイドで説明されている NVIDIA 製品はフォールト トレラント (耐障害性) ではないため、その使用やシステムの失敗が死亡、重大な身体傷害、または物的損害が生じるような状況 (核、航空電子工学、生命維持、またはその他の生命維持に不可欠な用途など) をもたらすあらゆるシステムの設計、構築、保全、および運用に使用するように設計、製造、または意図されていません。NVIDIA は、そのような危険度の高い使用への適合性に対する明示的または黙示的な保証を明確に否認します。NVIDIA は、そのような危険度の高い使用から生じるいかなる請求または損害賠償にも、その全部または一部に対して、お客様または第三者に責任を負わないものとします。

NVIDIA は、このガイドで説明されている製品が追加的なテストや修正を行わずに特定の用途に適合することを表明するものでも、保証するものでもありません。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。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欠点は、NVIDIA 製品の品質および信頼性に影響する可能性があり、その結果、このガイドには含まれていない追加的あるい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、コスト、あるいは問題に対しても責任を負いません。(i) このガイドに違反する方法で NVIDIA 製品を使用すること (ii) お客様の製品設計。

このガイドの製品に関する情報をお客様が使用する権利を除き、明示的か黙示的かを問わず、このガイドに基づいて、他のいかなるライセンスも NVIDIA によって付与されないものとします。このガイドに含まれる情報を複製することは、複製が NVIDIA によって書面で承認されており、改変なしで複製されており、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。

商標

NVIDIA、NVIDIA のロゴ、cuBLAS、CUDA、cuDNN、cuFFT、cuSPARSE、DIGITS、DGX、DGX-1、DGX Station、GRID、Jetson、Kepler、NVIDIA GPU Cloud、Maxwell、NCCL、NVLink、Pascal、Tegra、TensorRT、Tesla、および Volta は、米国またはその他の国における NVIDIA Corporation の商標または登録商標です。その他の社名ならびに製品名は、関連各社の商標である可能性があります。

Copyright

www.nvidia.co.jp