Deep Learning AMI

Developer Guide

Deep Learning AMI: Developer Guide

Table of Contents

What Is the AWS Deep Learning AMI?

Welcome to the User Guide for the AWS Deep Learning AMI.

The AWS Deep Learning AMI (DLAMI) is your one-stop shop for deep learning in the cloud. This customized machine instance is available in most Amazon EC2 regions for a variety of instance types, from a small CPU-only instance to the latest high-powered multi-GPU instances. It comes preconfigured with NVIDIA CUDA and NVIDIA cuDNN, as well as the latest releases of the most popular deep learning frameworks.

About This Guide

This guide will help you launch and use the DLAMI. It covers several use cases that are common for deep learning, for both training and inference. Choosing the right AMI for your purpose and the kind of instances you may prefer is also covered. The DLAMI comes with several tutorials for each of the frameworks. It also has tutorials on distributed training, debugging, using AWS Inferentia, and other key concepts. You will find instructions on how to configure Jupyter to run the tutorials in your browser.

Prerequisites

You should be familiar with command line tools and basic Python to successfully run the DLAMI. Tutorials on how to use each framework are provided by the frameworks themselves, however, this guide can show you how to activate each one and find the appropriate tutorials to get started.

Example DLAMI Uses

Learning about deep learning: The DLAMI is a great choice for learning or teaching machine learning and deep learning frameworks. It takes the headache away from troubleshooting the installations of each framework and getting them to play along on the same computer. The DLAMI comes with a Jupyter notebook and makes it easy to run the tutorials provided by the frameworks for people new to machine learning and deep learning.

App development: If you're an app developer and are interested in using deep learning to make your apps utilize the latest advances in AI, the DLAMI is the perfect test bed for you. Each framework comes with tutorials on how to get started with deep learning, and many of them have model zoos that make it easy to try out deep learning without having to create the neural networks yourself or to do any of the model training. Some examples show you how to build an image detection application in just a few minutes, or how to build a speech recognition app for your own chatbot.

Machine learning and data analytics: If you're a data scientist or interested in processing your data with deep learning, you'll find that many of the frameworks have support for R and Spark. You will find tutorials on how to do simple regressions, all the way up to building scalable data processing systems for personalization and predictions systems.

Research: If you're a researcher and want to try out a new framework, test out a new model, or train new models, the DLAMI and AWS capabilities for scale can alleviate the pain of tedious installations and management of multiple training nodes. You can use EMR and AWS CloudFormation templates to easily launch a full cluster of instances that are ready to go for scalable training.

Note
While your initial choice might be to upgrade your instance type up to a larger instance with more GPUs (up to 8), you can also scale horizontally by creating a cluster of DLAMI instances. To

quickly set up a cluster, you can use the predefined AWS CloudFormation template. Check out Related Information (p. 133) for more information on cluster builds.

Features of the DLAMI

Preinstalled Frameworks

There are currently two primary flavors of the DLAMI with other variations related to the operating system (OS) and software versions:

- Deep Learning AMI with Conda (p. 6) - frameworks installed separately using `conda` packages and separate Python environments
- Deep Learning Base AMI (p. 5) - no frameworks installed; only NVIDIA CUDA and other dependencies

The Deep Learning AMI with Conda uses Anaconda environments to isolate each framework, so you can switch between them at will and not worry about their dependencies conflicting.

For more information on selecting the best DLAMI for you, take a look at .

This is the full list of supported frameworks by Deep Learning AMI with Conda:

- Apache MXNet (Incubating)
- PyTorch
- TensorFlow 2

Note
We no longer include the CNTK, Caffe, Caffe2, Theano, Chainer, or Keras Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

Preinstalled GPU Software

Even if you use a CPU-only instance, the DLAMI will have NVIDIA CUDA and NVIDIA cuDNN. The installed software is the same regardless of the instance type. Keep in mind that GPU-specific tools only work on an instance that has at least one GPU. More information on this is covered in the Selecting the Instance Type for DLAMI (p. 7).

- Latest version of NVIDIA CUDA
- Latest version of NVIDIA cuDNN
- Older versions of CUDA are available as well. See the CUDA Installations and Framework Bindings (p. 5) for more information.

Model Serving and Visualization

Deep Learning AMI with Conda comes preinstalled with two kinds of model servers, one for MXNet and one for TensorFlow, as well as TensorBoard, for model visualizations.

- Model Server for Apache MXNet (MMS) (p. 118)

- TensorFlow Serving (p. 120)
- TensorBoard (p. 43)

Getting Started

How to Get Started with the DLAMI

This guide includes tips about picking the DLAMI that's right for you, selecting an instance type that fits your use case and budget, and Related Information (p. 133) that describes custom setups that may be of interest.

If you're new to using AWS or using Amazon EC2, start with the Deep Learning AMI with Conda (p. 6). If you're familiar with Amazon EC2 and other AWS services like Amazon EMR, Amazon EFS, or Amazon S3, and are interested in integrating those services for projects that need distributed training or inference, then check out Related Information (p. 133) to see if one fits your use case.

We recommend that you check out Choosing Your DLAMI (p. 4) to get an idea of which instance type might be best for your application.

Another option is this quick tutorial: Launch a AWS Deep Learning AMI (in 10 minutes).

Next Step

Choosing Your DLAMI (p. 4)

Choosing Your DLAMI

We offer a range of DLAMI options. To help you select the correct DLAMI for your use case, we group images by the hardware type or functionality for which they were developed. Our top level groupings are:

- **DLAMI Type:** CUDA versus Base versus Single-Framework versus Multi-Framework (Conda DLAMI)
- **Compute Architecture:** x86-based versus Arm-based AWS Graviton
- **Processor Type:** GPU versus CPU versus Inferentia versus Habana
- **SDK:** CUDA versus AWS Neuron versus SynapsesAI
- **OS:** Amazon Linux versus Ubuntu

The rest of the topics in this guide help further inform you and go into more details.

Topics

Next Up

Deep Learning AMI with Conda (p. 6)

CUDA Installations and Framework Bindings

While deep learning is all pretty cutting edge, each framework offers "stable" versions. These stable versions may not work with the latest CUDA or cuDNN implementation and features. Your use case and the features you require can help you choose a framework. If you are not sure, then use the latest Deep Learning AMI with Conda. It has official `pip` binaries for all frameworks with CUDA 10, using whichever most recent version is supported by each framework. If you want the latest versions, and to customize your deep learning environment, use the Deep Learning Base AMI.

Look at our guide on Stable Versus Release Candidates (p. 6) for further guidance.

Choose a DLAMI with CUDA

The Deep Learning Base AMI (p. 5) has all available CUDA 11 series, including 11.0, 11.1, and 11.2.

The Deep Learning AMI with Conda (p. 6) has all available CUDA 11 series, including 11.0, 11.1, and 11.2.

> **Note**
> We no longer include the CNTK, Caffe, Caffe2, Theano, Chainer, or Keras Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments continue to be available. However, we only provide updates to these environments if there are security fixes published by the open-source community for these frameworks.

For specific framework version numbers, see the Release Notes for DLAMI (p. 136)

Choose this DLAMI type or learn more about the different DLAMIs with the **Next Up** option.

Choose one of the CUDA versions and review the full list of DLAMIs that have that version in the **Appendix**, or learn more about the different DLAMIs with the **Next Up** option.

Next Up

Deep Learning Base AMI (p. 5)

Related Topics

- For instructions on switching between CUDA versions, refer to the Using the Deep Learning Base AMI (p. 25) tutorial.

Deep Learning Base AMI

The Deep Learning Base AMI is like an empty canvas for deep learning. It comes with everything you need up until the point of the installation of a particular framework, and has your choice of CUDA versions.

Why to Choose the Base DLAMI

This AMI group is useful for project contributors who want to fork a deep learning project and build the latest. It's for someone who wants to roll their own environment with the confidence that the latest NVIDIA software is installed and working so they can focus on picking which frameworks and versions they want to install.

Choose this DLAMI type or learn more about the different DLAMIs with the **Next Up** option.

Next Up

DLAMI with Conda

Related Topics

- Using the Deep Learning Base AMI

Deep Learning AMI with Conda

The Conda DLAMI uses Anaconda virtual environments. These environments are configured to keep the different framework installations separate and streamline switching between frameworks. This is great for learning and experimenting with all of the frameworks the DLAMI has to offer. Most users find that the new Deep Learning AMI with Conda is perfect for them.

These AMIs are the primary DLAMIs. They are updated often with the latest versions from the frameworks, and have the latest GPU drivers and software. They are generally referred to as the AWS Deep Learning AMI in most documents.

- The Ubuntu 18.04 DLAMI has the following frameworks: Apache MXNet (Incubating), PyTorch, and TensorFlow 2.
- The Amazon Linux 2 DLAMI has the following frameworks: Apache MXNet (Incubating), PyTorch, and TensorFlow 2.

Note
We no longer include the CNTK, Caffe, Caffe2, Theano, Chainer, and Keras Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments continue to be available. However, we only provide updates to these environments if there are security fixes published by the open-source community for these frameworks.

Stable Versus Release Candidates

The Conda AMIs use optimized binaries of the most recent formal releases from each framework. Release candidates and experimental features are not to be expected. The optimizations depend on the framework's support for acceleration technologies like Intel's MKL DNN, which speeds up training and inference on C5 and C4 CPU instance types. The binaries are also compiled to support advanced Intel instruction sets including but not limited to AVX, AVX-2, SSE4.1, and SSE4.2. These accelerate vector and floating point operations on Intel CPU architectures. Additionally, for GPU instance types, the CUDA and cuDNN are updated with whichever version the latest official release supports.

The Deep Learning AMI with Conda automatically installs the most optimized version of the framework for your Amazon EC2 instance upon the framework's first activation. For more information, refer to Using the Deep Learning AMI with Conda (p. 22).

If you want to install from source, using custom or optimized build options, the Deep Learning Base AMI (p. 5)s might be a better option for you.

Python 2 Deprecation

The Python open source community has officially ended support for Python 2 on January 1, 2020. The TensorFlow and PyTorch community have announced that the TensorFlow 2.1 and PyTorch 1.4 releases are the last ones supporting Python 2. Previous releases of the DLAMI (v26, v25, etc) that contain Python 2 Conda environments continue to be available. However, we provide updates to the Python 2 Conda environments on previously published DLAMI versions only if there are security fixes published by the open-source community for those versions. DLAMI releases with the latest versions of the TensorFlow and PyTorch frameworks do not contain the Python 2 Conda environments.

CUDA Support

Specific CUDA version numbers can be found in the GPU DLAMI release notes.

Next Up

DLAMI CPU Architecture Options (p. 7)

Related Topics

- For a tutorial on using a Deep Learning AMI with Conda, see the Using the Deep Learning AMI with Conda (p. 22) tutorial.

DLAMI CPU Architecture Options

AWS Deep Learning AMIs are offered with either x86-based or Arm-based AWS Graviton2 CPU architectures.

Choose one of the Graviton GPU DLAMIs to work with an Arm-based CPU architecture. All other GPU DLAMIs are currently x86-based.

- AWS Deep Learning AMI Graviton GPU CUDA 11.4 (Ubuntu 20.04)
- AWS Deep Learning AMI Graviton GPU TensorFlow 2.6 (Ubuntu 20.04)
- AWS Deep Learning AMI Graviton GPU PyTorch 1.10 (Ubuntu 20.04)

For information about getting started with the Graviton GPU DLAMI, see The Graviton DLAMI (p. 97). For more details on available instance types, see Selecting the Instance Type for DLAMI (p. 7).

Next Up

DLAMI Operating System Options (p. 7)

DLAMI Operating System Options

DLAMIs are offered in the following operating systems.

- Amazon Linux 2
- Ubuntu 20.04
- Ubuntu 18.04

Older versions of operating systems are available on deprecated DLAMIs. For more information on DLAMI deprecation, see Deprecations for DLAMI

Before choosing a DLAMI, assess what instance type you need and identify your AWS Region.

Next Up

Selecting the Instance Type for DLAMI (p. 7)

Selecting the Instance Type for DLAMI

See the AWS Deep Learning AMI Catalog for recommended Amazon EC2 instance families that are compatible with specific DLAMIs.

More generally, consider the following when selecting an instance type for a DLAMI.

- If you're new to deep learning, then an instance with a single GPU might suit your needs.
- If you're budget conscious, then you can use CPU-only instances.
- If you're looking to optimize high performance and cost efficiency for deep learning model inference, then you can use instances with AWS Inferentia chips.
- If you're looking to optimize high performance and cost efficiency for deep learning model training, then you can use instances with Habana accelerators.
- If you're looking for a high performance GPU instance with an Arm-based CPU architecture, then you can use the G5g instance type.
- If you're interested in running a pretrained model for inference and predictions, then you can attach an Amazon Elastic Inference to your Amazon EC2 instance. Amazon Elastic Inference gives you access to an accelerator with a fraction of a GPU.
- For high-volume inference services, a single CPU instance with a lot of memory, or a cluster of such instances, might be a better solution.
- If you're using a large model with a lot of data or a high batch size, then you need a larger instance with more memory. You can also distribute your model to a cluster of GPUs. You may find that using an instance with less memory is a better solution for you if you decrease your batch size. This may impact your accuracy and training speed.
- If you’re interested in running machine learning applications using NVIDIA Collective Communications Library (NCCL) requiring high levels of inter-node communications at scale, you might want to use Elastic Fabric Adapter (EFA).

For more detail on instances, see EC2 Instance Types.

The following topics provide information about instance type considerations.

Important
The Deep Learning AMIs include drivers, software, or toolkits developed, owned, or provided by NVIDIA Corporation. You agree to use these NVIDIA drivers, software, or toolkits only on Amazon EC2 instances that include NVIDIA hardware.

Topics

Pricing for the DLAMI

The deep learning frameworks included in the DLAMI are free, and each has its own open-source licenses. Although the software included in the DLAMI is free, you still have to pay for the underlying Amazon EC2 instance hardware.

Some Amazon EC2 instance types are labeled as free. It is possible to run the DLAMI on one of these free instances. This means that using the DLAMI is entirely free when you only use that instance's capacity. If you need a more powerful instance with more CPU cores, more disk space, more RAM, or one or more GPUs, then you need an instance that is not in the free-tier instance class.

For more information about instance selection and pricing, see Amazon EC2 pricing.

DLAMI Region Availability

Each Region supports a different range of instance types and often an instance type has a slightly different cost in different Regions. DLAMIs are not available in every Region, but it is possible to copy DLAMIs to the Region of your choice. See Copying an AMI for more information. Note the Region selection list and be sure you pick a Region that's close to you or your customers. If you plan to use more than one DLAMI and potentially create a cluster, be sure to use the same Region for all of nodes in the cluster.

For a more info on Regions, visit EC2 Regions.

Next Up

Recommended GPU Instances (p. 9)

Recommended GPU Instances

We recommend a GPU instance for most deep learning purposes. Training new models is faster on a GPU instance than a CPU instance. You can scale sub-linearly when you have multi-GPU instances or if you use distributed training across many instances with GPUs. To set up distributed training, see Distributed Training (p. 45).

The following instance types support the DLAMI. For information about GPU instance type options and their uses, see EC2 Instance Types and select **Accelerated Computing**.

> **Note**
> The size of your model should be a factor in selecting an instance. If your model exceeds an instance's available RAM, select a different instance type with enough memory for your application.

- Amazon EC2 P3 Instances have up to 8 NVIDIA Tesla V100 GPUs.
- Amazon EC2 P4 Instances have up to 8 NVIDIA Tesla A100 GPUs.
- Amazon EC2 G3 Instances have up to 4 NVIDIA Tesla M60 GPUs.
- Amazon EC2 G4 Instances have up to 4 NVIDIA T4 GPUs.
- Amazon EC2 G5 Instances have up to 8 NVIDIA A10G GPUs.
- Amazon EC2 G5g Instances have Arm-based AWS Graviton2 processors.

DLAMI instances provide tooling to monitor and optimize your GPU processes. For more information about monitoring your GPU processes, see GPU Monitoring and Optimization (p. 74).

For specific tutorials on working with G5g instances, see The Graviton DLAMI (p. 97).

Next Up

Recommended CPU Instances (p. 9)

Recommended CPU Instances

Whether you're on a budget, learning about deep learning, or just want to run a prediction service, you have many affordable options in the CPU category. Some frameworks take advantage of Intel's MKL DNN, which speeds up training and inference on C5 (not available in all Regions), C4, and C3 CPU instance types. For information about CPU instance types, see EC2 Instance Types and select **Compute Optimized**.

Note
The size of your model should be a factor in selecting an instance. If your model exceeds an instance's available RAM, select a different instance type with enough memory for your application.

- Amazon EC2 C5 Instances have up to 72 Intel vCPUs. C5 instances excel at scientific modeling, batch processing, distributed analytics, high-performance computing (HPC), and machine and deep learning inference.
- Amazon EC2 C4 Instances have up to 36 Intel vCPUs.

Next Up

Recommended Inferentia Instances (p. 10)

Recommended Inferentia Instances

AWS Inferentia instances are designed to provide high performance and cost efficiency for deep learning model inference workloads. Specifically, Inf1 instance types use AWS Inferentia chips and the AWS Neuron SDK, which is integrated with popular machine learning frameworks such as TensorFlow, PyTorch, and MXNet.

Customers can use Inf1 instances to run large scale machine learning inference applications such as search, recommendation engines, computer vision, speech recognition, natural language processing, personalization, and fraud detection, at the lowest cost in the cloud.

Note
The size of your model should be a factor in selecting an instance. If your model exceeds an instance's available RAM, select a different instance type with enough memory for your application.

- Amazon EC2 Inf1 Instances have up to up to 16 AWS Inferentia chips and 100 Gbps of networking throughput.

For more information about getting started with AWS Inferentia DLAMIs, see The AWS Inferentia Chip With DLAMI (p. 82).

Next Up

Recommended Habana Instances (p. 10)

Recommended Habana Instances

Instances with Habana accelerators are designed to provide high performance and cost efficiency for deep learning model training workloads. Specifically, DL1 instance types use Habana Gaudi accelerators from Habana Labs, an Intel company. DL1 instances are ideal for training machine learning models used in applications such as natural language processing, object detection and classification, recommendation engines, and autonomous vehicle perception.

Instances with Habana accelerators are configured with Habana SynapseAI software and pre-integrated with popular machine learning frameworks such as TensorFlow and PyTorch. If you are looking for an optimal combination of performance and price for training deep learning models, consider instances with Habana accelerators for the lowest cost to train.

Note
The size of your model should be a factor in selecting an instance. If your model exceeds an instance's available RAM, select a different instance type with enough memory for your application.

- Amazon EC2 DL1 Instances have up to eight Habana Gaudi accelerators, 256GB of accelerator memory, 4TB of local NVMe storage, and 400 Gbps of networking throughput.

For more information about getting started with Habana DLAMIs, see The Habana DLAMI (p. 104).

Launching and Configuring a DLAMI

If you're here you should already have a good idea of which AMI you want to launch. If not, choose a DLAMI using the AMI selection guidelines found throughout or use the full listing of AMIs in the Appendix section, .

You should also know which instance type and region you're going to choose. If not, browse Selecting the Instance Type for DLAMI (p. 7).

> **Note**
> We will use p2.xlarge as the default instance type in the examples. Just replace this with whichever instance type you have in mind.
>
> **Important**
> If you plan to use Elastic Inference, you have Elastic Inference Setup that must be completed prior to launching your DLAMI.

Topics

Step 1: Launch a DLAMI

> **Note**
> For this walkthrough, we might make references specific to the Deep Learning AMI (Ubuntu 16.04). Even if you select a different DLAMI, you should be able to follow this guide.

Launch the instance

1. You have a couple routes for launching DLAMI. Choose one:

    - EC2 Console (p. 13)
    - Marketplace Search (p. 14)

    > **Tip**
    > *CLI Option:* If you choose to spin up a DLAMI using the AWS CLI, you will need the AMI's ID, the region and instance type, and your security token information. Be sure you have your AMI and instance IDs ready. If you haven't set up the AWS CLI yet, do that first using the guide for Installing the AWS Command Line Interface.

2. After you have completed the steps of one of those options, wait for the instance to be ready. This usually takes only a few minutes. You can verify the status of the instance in the EC2 Console.

DLAMI ID

Find the ID for the DLAMI of your choice with the AWS Command Line Interface (AWS CLI). If you do not already have the AWS CLI installed, see Getting started with the AWS CLI.

1. Make sure that your AWS credentials are configured.

   ```
   aws configure
   ```

2. Choose a DLAMI and check the details in the release notes. Use the following command to get the ID for the DLAMI of your choice:

   ```
   aws ec2 describe-images --region us-east-1 --owners amazon \
   --filters 'Name=name,Values=Deep Learning AMI (Ubuntu 18.04) Version ??.?'
    'Name=state,Values=available' \
   --query 'reverse(sort_by(Images, &CreationDate))[:1].ImageId' --output text
   ```

   Note
   You can specify a release version for a given framework or get the latest release by replacing the version number with a question mark.

3. The output should look similar to the following:

   ```
   ami-094c089c38ed069f2
   ```

   Copy this DLAMI ID and press `q` to exit the prompt.

Next Step

EC2 Console (p. 13)

EC2 Console

Note
To launch an instance with Elastic Fabric Adapter (EFA), refer to these steps.

1. Open the EC2 Console.
2. Note your current region in the top-most navigation. If this isn't your desired AWS Region, change this option before proceeding. For more information, see EC2 Regions.
3. Choose **Launch Instance**.
4. Search for the desired instance by name:
   a. Select the DLAMI that is right for you. Find the DLAMI name as listed in the release notes or find the DLAMI ID using the AWS CLI.
   b. Choose **Community AMIs**.
      i. To view a selection of the latest DLAMIs, choose **Quick Start.**
      ii. Choose **AWS Marketplace** to browse additional DLAMIs. Only a subset of available DLAMIs will be listed here.
   c. Enter the DLAMI name or search the DLAMI ID. Browse the options and then click **Select** on your choice.
5. Review the details, and then choose **Continue**.
6. Choose an instance type. For recommendations on DLAMI instance types, see Instance Selection.

> **Note**
> If you want to use Elastic Inference (EI), click **Configure Instance Details**, select **Add an Amazon EI accelerator**, then select the size of the Amazon EI accelerator.

7. Choose **Review and Launch**.
8. Review the details and pricing. Choose **Launch**.

> **Tip**
> Check out Get Started with Deep Learning Using the AWS Deep Learning AMI for a walk-through with screenshots!

Next Step

Step 2: Connect to the DLAMI (p. 14)

Marketplace Search

1. Browse the AWS Marketplace and search for AWS Deep Learning AMI.
2. Browse the options, and then click **Select** on your choice.
3. Review the details, and then choose **Continue**.
4. Review the details and make note of the **Region**. If this isn't your desired AWS Region, change this option before proceeding. For more information, see EC2 Regions.
5. Choose an instance type.
6. Choose a key pair, use your default one, or create a new one.
7. Review the details and pricing.
8. Choose **Launch with 1-Click**.

Next Step

Step 2: Connect to the DLAMI (p. 14)

Step 2: Connect to the DLAMI

Connect to the DLAMI that you launched from a client (Windows, MacOS, or Linux). For more information, see Connect to Your Linux Instance in the *Amazon EC2 User Guide for Linux Instances*.

Keep a copy of the SSH login command handy if you want to do the Jupyter setup after logging in. You will use a variation of it to connect to the Jupyter webpage.

Next Step

Step 3: Secure Your DLAMI Instance (p. 14)

Step 3: Secure Your DLAMI Instance

Always keep your operating system and other installed software up to date by applying patches and updates as soon as they become available.

If you are using Amazon Linux or Ubuntu, when you login to your DLAMI, you are notified if updates are available and see instructions for updating. For further information on Amazon Linux maintenance, see Updating Instance Software. For Ubuntu instances, refer to the official Ubuntu documentation.

On Windows, check Windows Update regularly for software and security updates. If you prefer, have updates applied automatically.

> **Important**
> For information about the Meltdown and Spectre vulnerabilities and how to patch your operating system to address them, see Security Bulletin AWS-2018-013.

Step 4: Test Your DLAMI

Depending on your DLAMI version, you have different testing options:

- Deep Learning AMI with Conda (p. 6) – go to Using the Deep Learning AMI with Conda (p. 22).
- Deep Learning Base AMI (p. 5) – refer to your desired framework's installation documentation.

You can also create a Jupyter notebook, try out tutorials, or start coding in Python. For more information, see Set up a Jupyter Notebook Server (p. 15).

Clean Up

When you no longer need the DLAMI, you can stop it or terminate it to avoid incurring continuing charges. Stopping an instance will keep it around so you can resume it later. Your configurations, files, and other non-volatile information is being stored in a volume on Amazon S3. You will be charged the small S3 fee to retain the volume while the instance is stopped, but you will no longer be charged for the compute resources while it is in the stopped state. When your start the instance again, it will mount that volume and your data will be there. If you terminate an instance, it is gone, and you cannot start it again. Your data actually still resides on S3, so to prevent any further charges you need to delete the volume as well. For more instructions, see Terminate Your Instance in the *Amazon EC2 User Guide for Linux Instances*.

Set up a Jupyter Notebook Server

A Jupyter notebook server enables you to create and run Jupyter notebooks from your DLAMI instance. With Jupyter notebooks, you can conduct machine learning (ML) experiments for training and inference while using the AWS infrastructure and accessing packages built into the DLAMI. For more information about Jupyter notebooks, see the Jupyter Notebook documentation.

To set up a Jupyter notebook server, you must:

- Configure the Jupyter notebook server on your Amazon EC2 DLAMI instance.
- Configure your client so that you can connect to the Jupyter notebook server. We provide configuration instructions for Windows, macOS, and Linux clients.
- Test the setup by logging in to the Jupyter notebook server.

To complete the steps to set up a Jupyter, follow the instructions in the following topics. Once you've set up a Jupyter notebook server, see Running Jupyter Notebook Tutorials (p. 26) for information on running the example notebooks that ship in the DLAMI.

Topics

- Test by Logging in to the Jupyter notebook server (p. 18)

Secure Your Jupyter Server

Here we set up Jupyter with SSL and a custom password.

Connect to the Amazon EC2 instance, and then complete the following procedure.

Configure the Jupyter server

1. Jupyter provides a password utility. Run the following command and enter your preferred password at the prompt.

```
$ jupyter notebook password
```

   The output will look something like this:

```
Enter password:
Verify password:
[NotebookPasswordApp] Wrote hashed password to /home/ubuntu/.jupyter/
jupyter_notebook_config.json
```

2. Create a self-signed SSL certificate. Follow the prompts to fill out your locality as you see fit. You must enter `.` if you wish to leave a prompt blank. Your answers will not impact the functionality of the certificate.

```
$ cd ~
$ mkdir ssl
$ cd ssl
$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout mykey.key -out mycert.pem
```

   Note
   You might be interested in creating a regular SSL certificate that is third party signed and does not cause the browser to give you a security warning. This process is much more involved. Visit Jupyter's documention for more information.

Next Step

Start the Jupyter notebook server (p. 16)

Start the Jupyter notebook server

Now you can fire up the Jupyter server by logging in to the instance and running the following command that uses the SSL certificate you created in the previous step.

```
$ jupyter notebook --certfile=~/ssl/mycert.pem --keyfile ~/ssl/mykey.key
```

With the server started, you can now connect to it via an SSH tunnel from your client computer. When the server runs, you will see some output from Jupyter confirming that the server is running. At this point, ignore the callout that you can access the server via a localhost URL, because that won't work until you create the tunnel.

Note
Jupyter will handle switching environments for you when you switch frameworks using the Jupyter web interface. More info on this can be found in Switching Environments with Jupyter (p. 27).

Next Step

Configure the Client to Connect to the Jupyter Server (p. 17)

Configure the Client to Connect to the Jupyter Server

After configuring your client to connect to the Jupyter notebook server, you can create and access notebooks on the server in your workspace and run your deep learning code on the server.

For configuration information, choose one of the following links.

Topics

Configure a Windows Client

Prepare

Be sure you have the following information, which you need to set up the SSH tunnel:

- The public DNS name of your Amazon EC2 instance. You can find the public DNS name in the EC2 console.
- The key pair for the private key file. For more information about accessing your key pair, see Amazon EC2 Key Pairs in the *Amazon EC2 User Guide for Linux Instances*.

Using Jupyter Notebooks from a Windows Client

Refer to these guides on connecting to your Amazon EC2 instance from a Windows client.

1. Troubleshooting Connecting to Your Instance
2. Connecting to Your Linux Instance from Windows Using PuTTY

To create a tunnel to a running Jupyter server, a recommended approach is to install Git Bash on your Windows client, then follow the Linux/macOS client instructions. However, you may use whatever approach you want for opening an SSH tunnel with port mapping. Refer to Jupyter's documentation for further information.

Next Step

Configure a Linux or macOS Client (p. 17)

Configure a Linux or macOS Client

1. Open a terminal.
2. Run the following command to forward all requests on local port 8888 to port 8888 on your remote Amazon EC2 instance. Update the command by replacing the location of your key to access the Amazon EC2 instance and the public DNS name of your Amazon EC2 instance. Note, for an Amazon Linux AMI, the user name is `ec2-user` instead of `ubuntu`.

```
$ ssh -i ~/mykeypair.pem -N -f -L 8888:localhost:8888 ubuntu@ec2-###-##-##-
###.compute-1.amazonaws.com
```

This command opens a tunnel between your client and the remote Amazon EC2 instance that is running the Jupyter notebook server.

Next Step

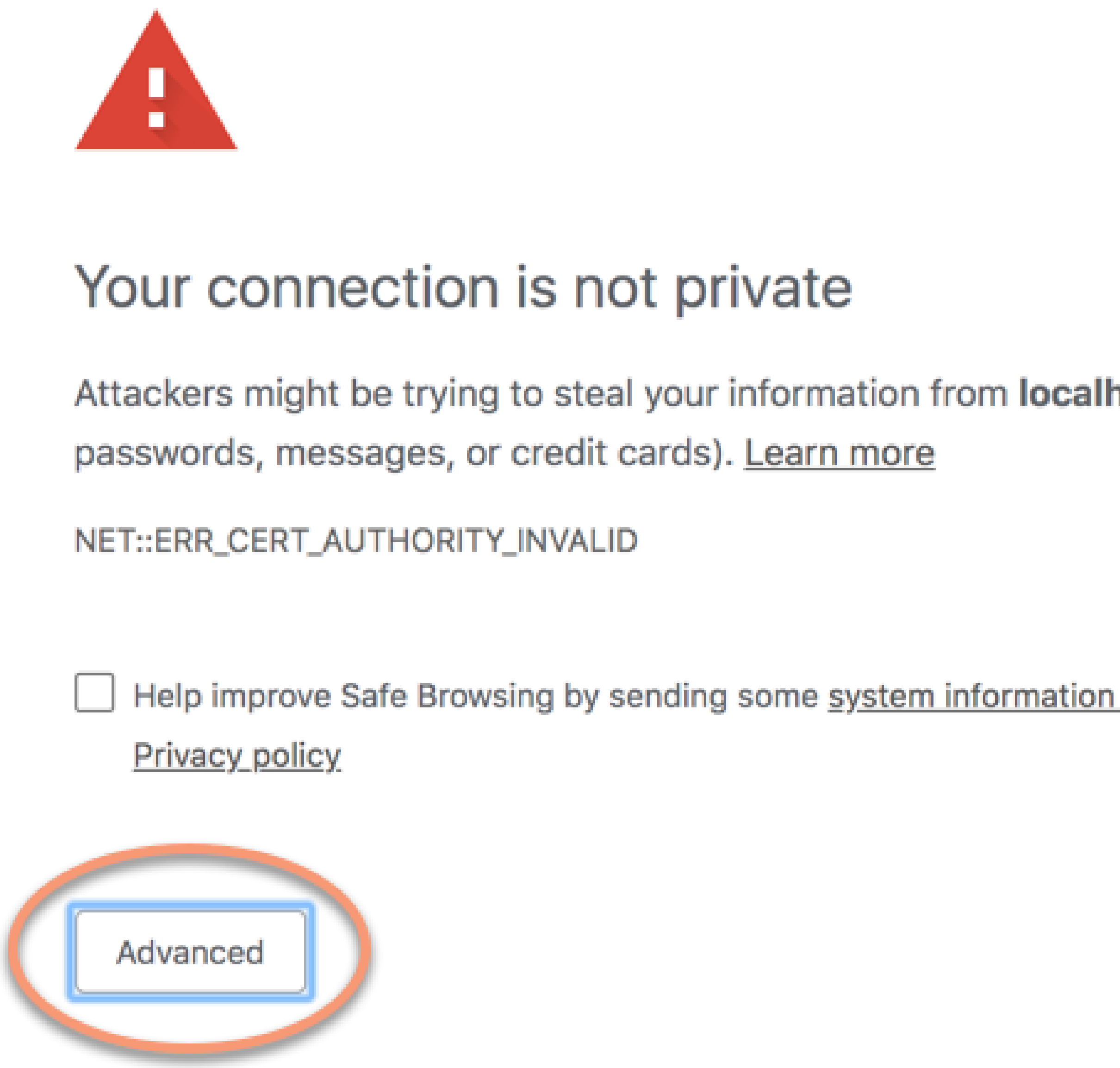
Test by Logging in to the Jupyter notebook server (p. 18)

Test by Logging in to the Jupyter notebook server

Now you are ready to log in to the Jupyter notebook server.

Your next step is to test the connection to the server through your browser.

1. In the address bar of your browser, type the following URL, or click on this link: https://localhost:8888
2. With a self signed SSL certificate, your browser will warn you and prompt you to avoid continuing to visit the website.



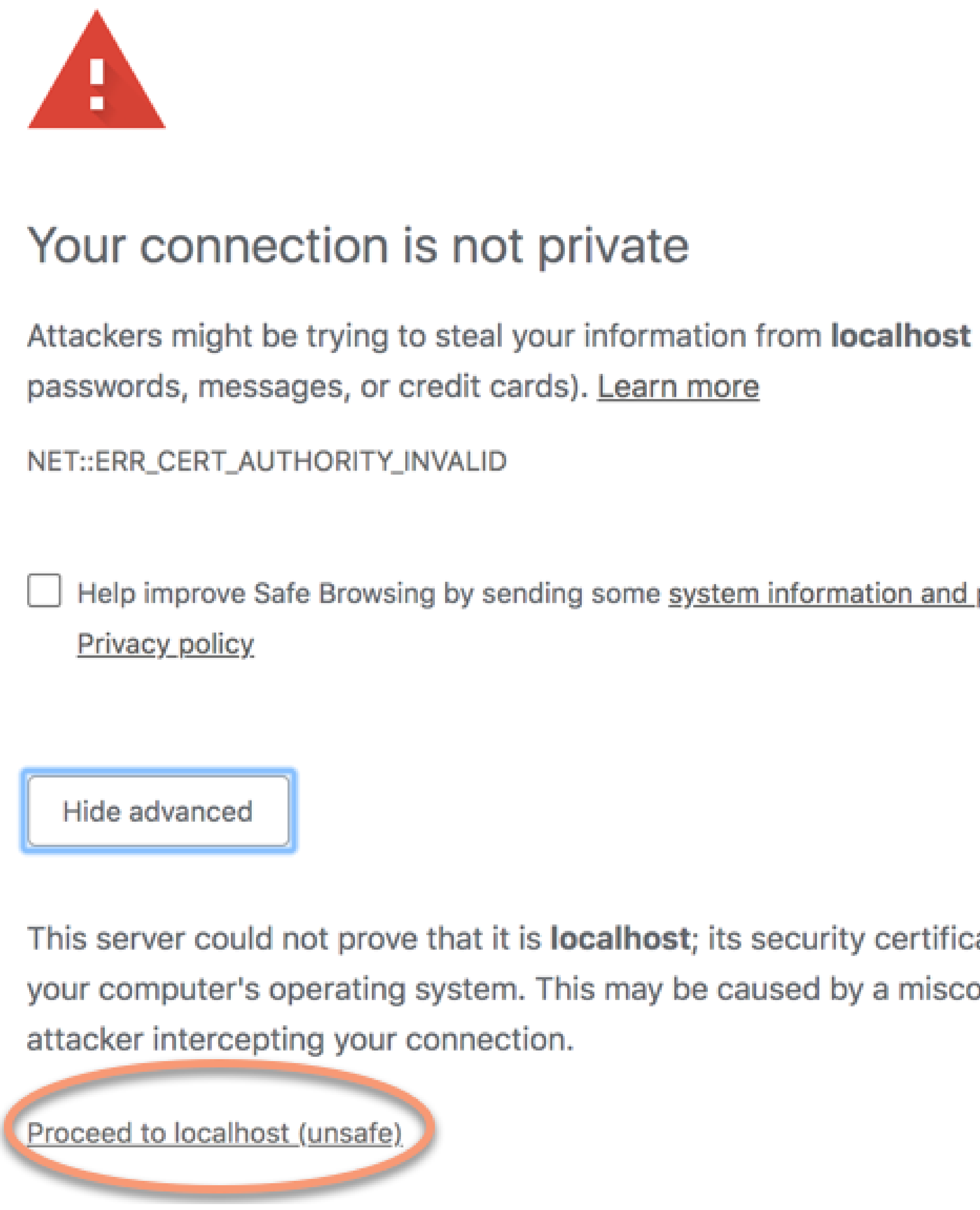
Your connection is not private

Attackers might be trying to steal your information from **localh**
passwords, messages, or credit cards). Learn more

NET::ERR_CERT_AUTHORITY_INVALID

Help improve Safe Browsing by sending some system information
Privacy policy

Since you set this up yourself, it is safe to continue. Depending your browser you will get an "advanced", "show details", or similar button.



Your connection is not private

Attackers might be trying to steal your information from **localhost**
passwords, messages, or credit cards). Learn more

NET::ERR_CERT_AUTHORITY_INVALID

Help improve Safe Browsing by sending some system information and
Privacy policy

Hide advanced

This server could not prove that it is **localhost**; its security certific
your computer's operating system. This may be caused by a misco
attacker intercepting your connection.

Proceed to localhost (unsafe)

Click on this, then click on the "proceed to localhost" link. If the connection is successful, you see the Jupyter notebook server webpage. At this point, you will be asked for the password you previously setup.

Now you have access to the Jupyter notebook server that is running on the DLAMI. You can create new notebooks or run the provided Tutorials (p. 27).

Using a DLAMI

Topics

The following sections describe how the Deep Learning AMI with Conda can be used to switch environments, run sample code from each of the frameworks, and run Jupyter so you can try out different notebook tutorials.

Using the Deep Learning AMI with Conda

Topics

Introduction to the Deep Learning AMI with Conda

Conda is an open source package management system and environment management system that runs on Windows, macOS, and Linux. Conda quickly installs, runs, and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer.

The Deep Learning AMI with Conda has been configured for you to easily switch between deep learning environments. The following instructions guide you on some basic commands with `conda`. They also help you verify that the basic import of the framework is functioning, and that you can run a couple simple operations with the framework. You can then move on to more thorough tutorials provided with the DLAMI or the frameworks' examples found on each frameworks' project site.

Log in to Your DLAMI

After you log in to your server, you will see a server "message of the day" (MOTD) describing various Conda commands that you can use to switch between the different deep learning frameworks. Below is an example MOTD. Your specific MOTD may vary as new versions of the DLAMI are released.

Note
We no longer include the CNTK, Caffe, Caffe2, Theano, Chainer, and Keras Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

```
=============================================================================
       __|  __|_  )
       _|  (      /   Deep Learning AMI (Ubuntu 18.04) Version 40.0
      ___|\___|___|
=============================================================================

Welcome to Ubuntu 18.04.5 LTS (GNU/Linux 5.4.0-1037-aws x86_64v)

Please use one of the following commands to start the required environment with the
 framework of your choice:
for AWS MX 1.7 (+Keras2) with Python3 (CUDA 10.1 and Intel MKL-DNN)
 _______________________________ source activate mxnet_p36
for AWS MX 1.8 (+Keras2) with Python3 (CUDA + and Intel MKL-DNN)
 ___________________________ source activate mxnet_latest_p37
for AWS MX(+AWS Neuron) with Python3 ___________________________________________________
 source activate aws_neuron_mxnet_p36
for AWS MX(+Amazon Elastic Inference) with Python3 _______________________________________
 source activate amazonei_mxnet_p36
for TensorFlow(+Keras2) with Python3 (CUDA + and Intel MKL-DNN)
 _____________________________ source activate tensorflow_p37
for Tensorflow(+AWS Neuron) with Python3 _________________________________________ source
 activate aws_neuron_tensorflow_p36
for TensorFlow 2(+Keras2) with Python3 (CUDA 10.1 and Intel MKL-DNN)
 _______________________ source activate tensorflow2_p36
for TensorFlow 2.3 with Python3.7 (CUDA + and Intel MKL-DNN) ________________________
 source activate tensorflow2_latest_p37
for PyTorch 1.4 with Python3 (CUDA 10.1 and Intel MKL)
 _________________________________________ source activate pytorch_p36
for PyTorch 1.7.1 with Python3.7 (CUDA 11.0 and Intel MKL) ________________________________
 source activate pytorch_latest_p37
for PyTorch (+AWS Neuron) with Python3 ______________________________________________
 source activate aws_neuron_pytorch_p36
for base Python3 (CUDA 10.0)
 _______________________________________________________________________ source activate
 python3
```

Each Conda command has the following pattern:

`source activate framework_python-version`

For example, you may see `for MXNet(+Keras1) with Python3 (CUDA 10.1) ________________________ source activate mxnet_p36`, which signifies that the environment has MXNet, Keras 1, Python 3, and CUDA 10.1. And to activate this environment, the command you would use is:

```
$ source activate mxnet_p36
```

Start the TensorFlow Environment

Note
When you launch your first Conda environment, please be patient while it loads. The Deep Learning AMI with Conda automatically installs the most optimized version of the framework for your EC2 instance upon the framework's first activation. You should not expect subsequent delays.

1. Activate the TensorFlow virtual environment for Python 3.

   ```
   $ source activate tensorflow_p37
   ```

2. Start the iPython terminal.

   ```
   (tensorflow_37)$ ipython
   ```

3. Run a quick TensorFlow program.

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
```

You should see "Hello, Tensorflow!"

Next Up

Running Jupyter Notebook Tutorials (p. 26)

Switch to the PyTorch Python 3 Environment

If you're still in the iPython console, use `quit()`, then get ready to switch environments.

- Activate the PyTorch virtual environment for Python 3.

```
$ source activate pytorch_p36
```

Test Some PyTorch Code

To test your installation, use Python to write PyTorch code that creates and prints an array.

1. Start the iPython terminal.

```
(pytorch_p36)$ ipython
```

2. Import PyTorch.

```
import torch
```

   You might see a warning message about a third-party package. You can ignore it.

3. Create a 5x3 matrix with the elements initialized randomly. Print the array.

```
x = torch.rand(5, 3)
print(x)
```

   Verify the result.

```
tensor([[0.3105, 0.5983, 0.5410],
        [0.0234, 0.0934, 0.0371],
        [0.9740, 0.1439, 0.3107],
        [0.6461, 0.9035, 0.5715],
        [0.4401, 0.7990, 0.8913]])
```

Switch to the MXNet Python 3 Environment

If you're still in the iPython console, use `quit()`, then get ready to switch environments.

- Activate the MXNet virtual environment for Python 3.

```
$ source activate mxnet_p36
```

Test Some MXNet Code

To test your installation, use Python to write MXNet code that creates and prints an array using the `NDArray` API. For more information, see NDArray API.

1. Start the iPython terminal.

```
(mxnet_p36)$ ipython
```

2. Import MXNet.

```
import mxnet as mx
```

   You might see a warning message about a third-party package. You can ignore it.
3. Create a 5x5 matrix, an instance of the `NDArray`, with elements initialized to 0. Print the array.

```
mx.ndarray.zeros((5,5)).asnumpy()
```

   Verify the result.

```
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]], dtype=float32)
```

   You can find more examples of MXNet in the MXNet tutorials section.

Removing Environments

If you run out of space on the DLAMI, you can choose to uninstall Conda packages that you are not using:

```
conda env list
conda env remove --name <env_name>
```

Using the Deep Learning Base AMI

Using the Deep Learning Base AMI

The Base AMI comes with a foundational platform of GPU drivers and acceleration libraries to deploy your own customized deep learning environment. By default the AMI is configured with the NVIDIA CUDA 11.0 environment. You can also switch between different versions of CUDA. Refer to the following instructions for how to do this.

Configuring CUDA Versions

You can verify the CUDA version by running NVIDIA's `nvcc` program.

```
nvcc --version
```

You can select and verify a particular CUDA version with the following bash command:

```
sudo rm /usr/local/cuda
sudo ln -s /usr/local/cuda-11.0 /usr/local/cuda
```

For more information, see the Base DLAMI release notes.

Running Jupyter Notebook Tutorials

Tutorials and examples ship with each of the deep learning projects' source and in most cases they will run on any DLAMI. If you chose the Deep Learning AMI with Conda (p. 6), you get the added benefit of a few hand-picked tutorials already set up and ready to try out.

Important
To run the Jupyter notebook tutorials installed on the DLAMI, you will need to Set up a Jupyter Notebook Server (p. 15).

Once the Jupyter server is running, you can run the tutorials through your web browser. If you are running the Deep Learning AMI with Conda or if you have set up Python environments, you can switch Python kernels from the Jupyter notebook interface. Select the appropriate kernel before trying to run a framework-specific tutorial. Further examples of this are provided for users of the Deep Learning AMI with Conda.

Note
Many tutorials require additional Python modules that may not be set up on your DLAMI. If you get an error like `"xyz module not found"`, log in to the DLAMI, activate the environment as described above, then install the necessary modules.

Tip
Deep learning tutorials and examples often rely on one or more GPUs. If your instance type doesn't have a GPU, you may need to change some of the example's code to get it to run.

Navigating the Installed Tutorials

Once you're logged in to the Jupyter server and can see the tutorials directory (on Deep Learning AMI with Conda only), you will be presented with folders of tutorials by each framework name. If you don't see a framework listed, then tutorials are not available for that framework on your current DLAMI. Click on the name of the framework to see the listed tutorials, then click a tutorial to launch it.

The first time you run a notebook on the Deep Learning AMI with Conda, it will want to know which environment you would like to use. It will prompt you to select from a list. Each environment is named according to this pattern:

```
Environment (conda_framework_python-version)
```

For example, you might see `Environment (conda_mxnet_p36)`, which signifies that the environment has MXNet and Python 3. The other variation of this would be `Environment (conda_mxnet_p27)`, which signifies that the environment has MXNet and Python 2.

Tip
If you're concerned about which version of CUDA is active, one way to see it is in the MOTD when you first log in to the DLAMI.

Switching Environments with Jupyter

If you decide to try a tutorial for a different framework, be sure to verify the currently running kernel. This info can be seen in the upper right of the Jupyter interface, just below the logout button. You can change the kernel on any open notebook by clicking the Jupyter menu item **Kernel**, then **Change Kernel**, and then clicking the environment that suits the notebook you're running.

At this point you'll need to rerun any cells because a change in the kernel will erase the state of anything you've run previously.

Tip
Switching between frameworks can be fun and educational, however you can run out of memory. If you start running into errors, look at the terminal window that has the Jupyter server running. There are helpful messages and error logging here, and you may see an out-of-memory error. To fix this, you can go to the home page of your Jupyter server, click the **Running** tab, then click **Shutdown** for each of the tutorials that are probably still running in the background and eating up all of your memory.

Next Up

For more examples and sample code from each framework, click **Next** or continue to Apache MXNet (Incubating) (p. 28).

Tutorials

The following are tutorials on how to use the Deep Learning AMI with Conda's software.

Topics

10 Minute Tutorials

- Launch a AWS Deep Learning AMI (in 10 minutes)
- Train a Deep Learning model with AWS Deep Learning Containers on Amazon EC2 (in 10 minutes)

Activating Frameworks

The following are the deep learning frameworks installed on the Deep Learning AMI with Conda. Click on a framework to learn how to activate it.

Topics

Apache MXNet (Incubating)

Activating Apache MXNet (Incubating)

This tutorial shows how to activate MXNet on an instance running the Deep Learning AMI with Conda (DLAMI on Conda) and run a MXNet program.

When a stable Conda package of a framework is released, it's tested and pre-installed on the DLAMI. If you want to run the latest, untested nightly build, you can Installing MXNet's Nightly Build (experimental) (p. 29) manually.

To run MXNet on the DLAMI with Conda

1. To activate the framework, open an Amazon Elastic Compute Cloud (Amazon EC2) instance of the DLAMI with Conda.
   - For MXNet and Keras 2 on Python 3 with CUDA 9.0 and MKL-DNN, run this command:

     ```
     $ source activate mxnet_p36
     ```

   - For MXNet and Keras 2 on Python 2 with CUDA 9.0 and MKL-DNN, run this command:

     ```
     $ source activate mxnet_p27
     ```

2. Start the iPython terminal.

   ```
   (mxnet_p36)$ ipython
   ```

3. Run a quick MXNet program. Create a 5x5 matrix, an instance of the `NDArray`, with elements initialized to 0. Print the array.

   ```
   import mxnet as mx
   mx.ndarray.zeros((5,5)).asnumpy()
   ```

4. Verify the result.

```
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]], dtype=float32)
```

Installing MXNet's Nightly Build (experimental)

You can install the latest MXNet build into either or both of the MXNet Conda environments on your Deep Learning AMI with Conda.

To install MXNet from a nightly build

1. - For the Python 3 MXNet environment, run this command:

     ```
     $ source activate mxnet_p36
     ```

   - For the Python 2 MXNet environment, run this command:

     ```
     $ source activate mxnet_p27
     ```

2. Remove the currently installed MXNet.

   Note
   The remaining steps assume you are using the `mxnet_p36` environment.

   ```
   (mxnet_p36)$ pip uninstall mxnet-cu90mkl
   ```

3. Install the latest nightly build of MXNet.

   ```
   (mxnet_p36)$ pip install --pre mxnet-cu90mkl
   ```

4. To verify you have successfully installed latest nightly build, start the IPython terminal and check the version of MXNet.

   ```
   (mxnet_p36)$ ipython
   ```

   ```
   import mxnet
   print (mxnet.__version__)
   ```

   The output should print the latest stable version of MXNet.

More Tutorials

You can find more tutorials in the Deep Learning AMI with Conda tutorials folder, which is in the home directory of the DLAMI.

1. Use Apache MXNet (Incubating) for Inference with a ResNet 50 Model (p. 107)
2. Use Apache MXNet (Incubating) for Inference with an ONNX Model (p. 106)
3. Model Server for Apache MXNet (MMS) (p. 118)

For more tutorials and examples, see the framework's official Python documentation, the Python API for MXNet, or the Apache MXNet website.

Caffe2

Note
We no longer include the CNTK, Caffe, Caffe2 and Theano Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

Caffe2 Tutorial

To activate the framework, follow these instructions on your Deep Learning AMI with Conda.

There is only the Python 2 with CUDA 9 with cuDNN 7 option:

```
$ source activate caffe2_p27
```

Start the iPython terminal.

```
(caffe2_p27)$ ipython
```

Run a quick Caffe2 program.

```

from caffe2.python import workspace, model_helper
import numpy as np
# Create random tensor of three dimensions
x = np.random.rand(4, 3, 2)
print(x)
print(x.shape)
workspace.FeedBlob("my_x", x)
x2 = workspace.FetchBlob("my_x")
print(x2)
```

You should see the initial numpy random arrays printed and then those loaded into a Caffe2 blob. Note that after loading they are the same.

More Tutorials

For more tutorials and examples refer to the framework's official Python docs, Python API for Caffe2, and the Caffe2 website.

Chainer

Note
We no longer include Chainer Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

Chainer is a flexible Python-based framework for easily and intuitively writing complex neural network architectures. Chainer makes it easy to use multi-GPU instances for training. Chainer also automatically logs results, graph loss and accuracy, and produces output for visualizing the neural network with a computational graph. It is included with the Deep Learning AMI with Conda (DLAMI with Conda).

Activate Chainer

1. Connect to the instance running Deep Learning AMI with Conda. Refer to the the section called "Instance Selection" (p. 7) or the Amazon EC2 documentation on how to select or connect to an instance.
2. • Activate the Python 3 Chainer environment:

   ```
   $ source activate chainer_p36
   ```

   • Activate the Python 2 Chainer environment:

   ```
   $ source activate chainer_p27
   ```
3. Start the iPython terminal:

```
(chainer_p36)$ ipython
```
4. Test importing Chainer to verify that it is working properly:

```

import chainer
```

   You may see a few warning messages, but no error.

More Info

- Try the tutorials for Chainer (p. 45).
- The `Chainer` examples folder inside the source you downloaded earlier contains more examples. Try them to see how they perform.
- To learn more about Chainer, see the Chainer documentation website.

CNTK

Note
We no longer include the CNTK, Caffe, Caffe2 and Theano Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

Activating CNTK

This tutorial shows how to activate CNTK on an instance running the Deep Learning AMI with Conda (DLAMI on Conda) and run a CNTK program.

When a stable Conda package of a framework is released, it's tested and pre-installed on the DLAMI. If you want to run the latest, untested nightly build, you can Install the CNTK Nightly Build (experimental) (p. 32) manually.

To run CNTK on the DLAMI with Conda

1. To activate CNTK, open an Amazon Elastic Compute Cloud (Amazon EC2) instance of the DLAMI with Conda.
   • For Python 3 with CUDA 9 with cuDNN 7:

```
$ source activate cntk_p36
```

- For Python 2 with CUDA 9 with cuDNN 7:

```
$ source activate cntk_p27
```

2. Start the iPython terminal.

```
(cntk_p36)$ ipython
```

3. - If you have a CPU instance, run this quick CNTK program.

```
import cntk as C
C.__version__
c = C.constant(3, shape=(2,3))
c.asarray()
```

  You should see the CNTK version, then the output of a 2x3 array of 3's.

- If you have a GPU instance, you can test it with the following code example. A result of `True` is what you would expect if CNTK can access the GPU.

```
from cntk.device import try_set_default_device, gpu
try_set_default_device(gpu(0))
```

Install the CNTK Nightly Build (experimental)

You can install the latest CNTK build into either or both of the CNTK Conda environments on your Deep Learning AMI with Conda.

To install CNTK from a nightly build

1. - For CNTK and Keras 2 on Python 3 with CUDA 9.0 and MKL-DNN, run this command:

```
$ source activate cntk_p36
```

- For CNTK and Keras 2 on Python 2 with CUDA 9.0 and MKL-DNN, run this command:

```
$ source activate cntk_p27
```

2. The remaining steps assume you are using the `cntk_p36` environment. Remove the currently installed CNTK.

```
(cntk_p36)$ pip uninstall cntk
```

3. To install the CNTK nightly build, you first need to find the version you want to install from the CNTK nightly website.

4. - (Option for GPU instances) - To install the nightly build, you would use the following, substituting in the desired build:

```
(cntk_p36)$ pip install https://cntk.ai/PythonWheel/GPU/latest-nightly-build
```

  Replace the URL in the previous command with the GPU version for your current Python environment.

- (Option for CPU instances) - To install the nightly build, you would use the following, substituting in the desired build:

  ```
  (cntk_p36)$ pip install https://cntk.ai/PythonWheel/CPU-Only/latest-nightly-build
  ```

  Replace the URL in the previous command with the CPU version for your current Python environment.

5. To verify you have successfully installed latest nightly build, start the IPython terminal and check the version of CNTK.

   ```
   (cntk_p36)$ ipython
   ```

   ```
   import cntk
   print (cntk.__version__)
   ```

   The output should print something similar to `2.6-rc0.dev20181015`

More Tutorials

For more tutorials and examples, see the framework's official Python docs, Python API for CNTK, or the CNTK website.

Keras

Keras Tutorial

1. To activate the framework, use these commands on your the section called "Conda DLAMI" (p. 22) CLI.
   - For Keras 2 with an MXNet backend on Python 3 with CUDA 9 with cuDNN 7:

     ```
     $ source activate mxnet_p36
     ```

   - For Keras 2 with an MXNet backend on Python 2 with CUDA 9 with cuDNN 7:

     ```
     $ source activate mxnet_p27
     ```

   - For Keras 2 with a TensorFlow backend on Python 3 with CUDA 9 with cuDNN 7:

     ```
     $ source activate tensorflow_p36
     ```

   - For Keras 2 with a TensorFlow backend on Python 2 with CUDA 9 with cuDNN 7:

     ```
     $ source activate tensorflow_p27
     ```

2. To test importing Keras to verify which backend is activated, use these commands:

   ```
   $ ipython
   import keras as k
   ```

   The following should appear on your screen:

```
Using MXNet backend
```

If Keras is using TensorFlow, the following is displayed:

```
Using TensorFlow backend
```

Note
If you get an error, or if the wrong backend is still being used, you can update your Keras configuration manually. Edit the `~/.keras/keras.json` file and change the backend setting to `mxnet` or `tensorflow`.

More Tutorials

- For a multi-GPU tutorial using Keras with a MXNet backend, try the Keras-MXNet Multi-GPU Training Tutorial (p. 54).
- You can find examples for Keras with a MXNet backend in the Deep Learning AMI with Conda `~/examples/keras-mxnet` directory.
- You can find examples for Keras with a TensorFlow backend in the Deep Learning AMI with Conda `~/examples/keras` directory.
- For additional tutorials and examples, see the Keras website.

PyTorch

Activating PyTorch

When a stable Conda package of a framework is released, it's tested and pre-installed on the DLAMI. If you want to run the latest, untested nightly build, you can Install PyTorch's Nightly Build (experimental) (p. 35) manually.

To activate the currently installed framework, follow these instructions on your Deep Learning AMI with Conda.

For PyTorch on Python 3 with CUDA 10 and MKL-DNN, run this command:

```
$ source activate pytorch_p36
```

For PyTorch on Python 2 with CUDA 10 and MKL-DNN, run this command:

```
$ source activate pytorch_p27
```

Start the iPython terminal.

```
(pytorch_p36)$ ipython
```

Run a quick PyTorch program.

```
import torch
x = torch.rand(5, 3)
print(x)
print(x.size())
y = torch.rand(5, 3)
print(torch.add(x, y))
```

You should see the initial random array printed, then its size, and then the addition of another random array.

Install PyTorch's Nightly Build (experimental)

How to install PyTorch from a nightly build

You can install the latest PyTorch build into either or both of the PyTorch Conda environments on your Deep Learning AMI with Conda.

1. - (Option for Python 3) - Activate the Python 3 PyTorch environment:

     ```
     $ source activate pytorch_p36
     ```

   - (Option for Python 2) - Activate the Python 2 PyTorch environment:

     ```
     $ source activate pytorch_p27
     ```

2. The remaining steps assume you are using the `pytorch_p36` environment. Remove the currently installed PyTorch:

   ```
   (pytorch_p36)$ pip uninstall torch
   ```

3. - (Option for GPU instances) - Install the latest nightly build of PyTorch with CUDA 10.0:

     ```
     (pytorch_p36)$ pip install torch_nightly -f https://download.pytorch.org/whl/
     nightly/cu100/torch_nightly.html
     ```

   - (Option for CPU instances) - Install the latest nightly build of PyTorch for instances with no GPUs:

     ```
     (pytorch_p36)$ pip install torch_nightly -f https://download.pytorch.org/whl/
     nightly/cpu/torch_nightly.html
     ```

4. To verify you have successfully installed latest nightly build, start the IPython terminal and check the version of PyTorch.

   ```
   (pytorch_p36)$ ipython
   ```

   ```
   import torch
   print (torch.__version__)
   ```

   The output should print something similar to `1.0.0.dev20180922`

5. To verify that the PyTorch nightly build works well with the MNIST example, you can run a test script from PyTorch's examples repository:

   ```
   (pytorch_p36)$ cd ~
   (pytorch_p36)$ git clone https://github.com/pytorch/examples.git pytorch_examples
   (pytorch_p36)$ cd pytorch_examples/mnist
   (pytorch_p36)$ python main.py || exit 1
   ```

More Tutorials

You can find more tutorials in the Deep Learning AMI with Conda tutorials folder in the home directory of the DLAMI. For further tutorials and examples refer to the framework's official docs, PyTorch documentation, and the PyTorch website.

- PyTorch to ONNX to MXNet Tutorial (p. 116)
- PyTorch to ONNX to CNTK Tutorial (p. 114)

TensorFlow

Activating TensorFlow

This tutorial shows how to activate TensorFlow on an instance running the Deep Learning AMI with Conda (DLAMI on Conda) and run a TensorFlow program.

When a stable Conda package of a framework is released, it's tested and pre-installed on the DLAMI. If you want to run the latest, untested nightly build, you can Install TensorFlow's Nightly Build (experimental) (p. 36) manually.

To run TensorFlow on the DLAMI with Conda

1. To activate TensorFlow, open an Amazon Elastic Compute Cloud (Amazon EC2) instance of the DLAMI with Conda.
   - For TensorFlow and Keras 2 on Python 3 with CUDA 9.0 and MKL-DNN, run this command:

     ```
     $ source activate tensorflow_p36
     ```

   - For TensorFlow and Keras 2 on Python 2 with CUDA 9.0 and MKL-DNN, run this command:

     ```
     $ source activate tensorflow_p27
     ```

2. Start the iPython terminal:

   ```
   (tensorflow_p36)$ ipython
   ```

3. Run a TensorFlow program to verify that it is working properly:

   ```


   import tensorflow as tf
   hello = tf.constant('Hello, TensorFlow!')
   sess = tf.Session()
   print(sess.run(hello))
   ```

   `Hello, TensorFlow!` should appear on your screen.

Install TensorFlow's Nightly Build (experimental)

You can install the latest TensorFlow build into either or both of the TensorFlow Conda environments on your Deep Learning AMI with Conda.

To install TensorFlow from a nightly build

1. - For the Python 3 TensorFlow environment, run the following command:

```
$ source activate tensorflow_p36
```

- For the Python 2 TensorFlow environment, run the following command:

```
$ source activate tensorflow_p27
```

2. Remove the currently installed TensorFlow.

   Note
   The remaining steps assume you are using the `tensorflow_p36` environment.

```
(tensorflow_p36)$ pip uninstall tensorflow
```

3. Install the latest nightly build of TensorFlow.

```
(tensorflow_p36)$ pip install tf-nightly
```

4. To verify you have successfully installed latest nightly build, start the IPython terminal and check the version of TensorFlow.

```
(tensorflow_p36)$ ipython
```

```
import tensorflow
print (tensorflow.__version__)
```

   The output should print something similar to `1.12.0-dev20181012`

More Tutorials

TensorFlow with Horovod (p. 55)

TensorBoard (p. 43)

TensorFlow Serving (p. 120)

For tutorials, see the folder called `Deep Learning AMI with Conda tutorials` in the home directory of the DLAMI.

For more tutorials and examples, see the TensorFlow documentation for the TensorFlow Python API or see the TensorFlow website.

TensorFlow 2

This tutorial shows how to activate TensorFlow 2 on an instance running the Deep Learning AMI with Conda (DLAMI on Conda) and run a TensorFlow 2 program.

When a stable Conda package of a framework is released, it's tested and pre-installed on the DLAMI. If you want to run the latest, untested nightly build, you can Install TensorFlow 2's Nightly Build (experimental) (p. 38) manually.

Activating TensorFlow 2

To run TensorFlow on the DLAMI with Conda

1. To activate TensorFlow 2, open an Amazon Elastic Compute Cloud (Amazon EC2) instance of the DLAMI with Conda.

2. For TensorFlow 2 and Keras 2 on Python 3 with CUDA 10.1 and MKL-DNN, run this command:

```
$ source activate tensorflow2_p36
```

3. Start the iPython terminal:

```
(tensorflow2_p36)$ ipython
```

4. Run a TensorFlow 2 program to verify that it is working properly:

```


import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
tf.print(hello)
```

`Hello, TensorFlow!` should appear on your screen.

Install TensorFlow 2's Nightly Build (experimental)

You can install the latest TensorFlow 2 build into either or both of the TensorFlow 2 Conda environments on your Deep Learning AMI with Conda.

To install TensorFlow from a nightly build

1. For the Python 3 TensorFlow 2 environment, run the following command:

```
$ source activate tensorflow2_p36
```

2. Remove the currently installed TensorFlow.

Note
The remaining steps assume you are using the `tensorflow2_p36` environment.

```
(tensorflow2_p36)$ pip uninstall tensorflow
```

3. Install the latest nightly build of TensorFlow.

```
(tensorflow2_p36)$ pip install tf-nightly
```

4. To verify you have successfully installed latest nightly build, start the IPython terminal and check the version of TensorFlow.

```
(tensorflow2_p36)$ ipython
```

```
import tensorflow
print (tensorflow.__version__)
```

The output should print something similar to `2.1.0-dev20191122`

More Tutorials

For tutorials, see the folder called `Deep Learning AMI with Conda tutorials` in the home directory of the DLAMI.

For more tutorials and examples, see the TensorFlow documentation for the TensorFlow Python API or see the TensorFlow website.

TensorFlow with Horovod

This tutorial shows how to activate TensorFlow with Horovod on an AWS Deep Learning AMI (DLAMI) with Conda. Horovod is pre-installed in the Conda environments for TensorFlow. The Python3 environment is recommended.

Note
Only P3.*, P2.*, and G3.* instance types are supported.

To activate TensorFlow and test Horovod on the DLAMI with Conda

1. Open an Amazon Elastic Compute Cloud (Amazon EC2) instance of the DLAMI with Conda. For help getting started with a DLAMI, see the section called “How to Get Started with the DLAMI” (p. 4).
2. (Recommended) For TensorFlow 1.15 with Horovod on Python 3 with CUDA 11, run the following command:

```
$ source activate tensorflow_p37
```

3. Start the iPython terminal:

```
(tensorflow_p37)$ ipython
```

4. Test importing TensorFlow with Horovod to verify that it's working properly:

```
import horovod.tensorflow as hvd
hvd.init()
```

The following may appear on your screen (you may ignore any warning messages).

```
--------------------------------------------------------------------------
[[55425,1],0]: A high-performance Open MPI point-to-point messaging module
was unable to find any relevant network interfaces:

Module: OpenFabrics (openib)
  Host: ip-172-31-72-4

Another transport will be used instead, although this may result in
lower performance.
--------------------------------------------------------------------------
```

More Info

- TensorFlow with Horovod (p. 55)
- For tutorials, see the `examples/horovod` folder in the home directory of the DLAMI.
- For even more tutorials and examples, see the Horovod GitHub project.

TensorFlow 2 with Horovod

This tutorial shows how to activate TensorFlow 2 with Horovod on an AWS Deep Learning AMI (DLAMI) with Conda. Horovod is pre-installed in the Conda environments for TensorFlow 2. The Python3 environment is recommended.

Note
Only P3.*, P2.*, and G3.* instance types are supported.

To activate TensorFlow 2 and test Horovod on the DLAMI with Conda

1. Open an Amazon Elastic Compute Cloud (Amazon EC2) instance of the DLAMI with Conda. For help getting started with a DLAMI, see the section called "How to Get Started with the DLAMI" (p. 4).
   - (Recommended) For TensorFlow 2 with Horovod on Python 3 with CUDA 10, run this command:

     ```
     $ source activate tensorflow2_p36
     ```
   - For TensorFlow 2 with Horovod on Python 2 with CUDA 10, run this command:

     ```
     $ source activate tensorflow2_p27
     ```
2. Start the iPython terminal:

   ```
   (tensorflow2_p36)$ ipython
   ```
3. Test importing TensorFlow 2 with Horovod to verify that it's working properly:

   ```
   import horovod.tensorflow as hvd
   hvd.init()
   ```

   If you don't receive any output, then Horovod is working properly. The following may appear on your screen (you may ignore any warning messages).

   ```
   --------------------------------------------------------------------------
   [[55425,1],0]: A high-performance Open MPI point-to-point messaging module
   was unable to find any relevant network interfaces:

   Module: OpenFabrics (openib)
     Host: ip-172-31-72-4

   Another transport will be used instead, although this may result in
   lower performance.
   --------------------------------------------------------------------------
   ```

More Info

- For tutorials, see the `examples/horovod` folder in the home directory of the DLAMI.
- For even more tutorials and examples, see the Horovod GitHub project.

Theano

Theano Tutorial

Note
We no longer include the CNTK, Caffe, Caffe2 and Theano Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

To activate the framework, follow these instructions on your Deep Learning AMI with Conda.

For Theano + Keras in Python 3 with CUDA 9 with cuDNN 7:

```
$ source activate theano_p36
```

For Theano + Keras in Python 2 with CUDA 9 with cuDNN 7:

```
$ source activate theano_p27
```

Start the iPython terminal.

```
(theano_p36)$ ipython
```

Run a quick Theano program.

```
import numpy
import theano
import theano.tensor as T
from theano import pp
x = T.dscalar('x')
y = x ** 2
gy = T.grad(y, x)
pp(gy)
```

You should see Theano computing a symbolic gradient.

More Tutorials

For further tutorials and examples refer to the framework's official docs, Theano Python API, and the Theano website.

Debugging and Visualization

Learn about the debugging and visualization options for the DLAMI. Click on one of the options to learn how to use it.

Topics

MXBoard

MXBoard lets you to visually inspect and interpret your MXNet runs and graphs using the TensorBoard software. It runs a web server that serves a webpage for viewing and interacting with the MXBoard visualizations.

MXNet, TensorBoard, and MXBoard are preinstalled with the Deep Learning AMI with Conda (DLAMI with Conda). In this tutorial, you use an MXBoard function to generate logs that are compatible with TensorBoard.

Topics

Using MXNet with MXBoard

Generate MXBoard Log Data Compatible with TensorBoard

1. Connect to your Amazon Elastic Compute Cloud (Amazon EC2) instance of the DLAMI with Conda.
2. Activate the Python 3 MXNet environment.

```
$ source activate mxnet_p36
```

3. Prepare a Python script for writing data generated by the normal operator to an event file. The data is generated ten times with decreasing standard deviation, then written to the event file each time. You will see data distribution gradually become more centered around the mean value. Note that you will specify the event file in the logs folder. You pass this folder path to the TensorBoard binary.

```
$ vi mxboard_normal.py
```

4. Paste the following in the file and save it:

```

import mxnet as mx
from mxboard import SummaryWriter


with SummaryWriter(logdir='./logs') as sw:
    for i in range(10):
        # create a normal distribution with fixed mean and decreasing std
        data = mx.nd.normal(loc=0, scale=10.0/(i+1), shape=(10, 3, 8, 8))
        sw.add_histogram(tag='norml_dist', values=data, bins=200, global_step=i)
```

5. Run the script. This will generate logs in a `logs` folder that you can use for visualizations.

```
$ python mxboard_normal.py
```

6. Now you must switch to the TensorFlow environment to use TensorBoard and MXBoard to visualize the logs. This is a required dependency for MXBoard and TensorBoard.

```
$ source activate tensorflow_p36
```

7. Pass the location of the logs to `tensorboard`:

   ```
   $ tensorboard --logdir=./logs --host=127.0.0.1 --port=8888
   ```

   TensorBoard launches the visualization web server on port 8888.
8. For easy access from your local browser, you can change the web server port to port 80 or another port. Whichever port you use, you will need to open this port in the EC2 security group for your DLAMI. You can also use port forwarding. For instructions on changing your security group settings and port forwarding, see Set up a Jupyter Notebook Server (p. 15). The default settings are described in the next step.

   Note
   If you need to run both Jupyter server and a MXBoard server, use a different port for each.
9. Open port 8888 (or the port you assigned to the visualization web server) on your EC2 instance.

   a. Open your EC2 instance in the Amazon EC2console at https://console.aws.amazon.com/ec2/.
   b. In the Amazon EC2 console, choose **Network & Security**, then choose **Security Groups**.
   c. For **Security Group**, , choose the one that was created most recently (see the timestamp in the description).
   d. Choose the **Inbound** tab, and choose **Edit**.
   e. Choose **Add Rule**.
   f. In the new row, type the following:

      Type : Custom **`TCP Rule`**

      Protocol: **`TCP`**

      Port Range: **`8888`** (or the port that you assigned to the visualization server)

      Source: **`Custom IP (specify address/range)`**
10. If you want to visualize the data from local browser, type the following command to forward the data that is rendering on the EC2 instance to your local machine.

    ```
    $ ssh -Y -L localhost:8888:localhost:8888 user_id@ec2_instance_ip
    ```
11. Open the web page for the MXBoard visualizations by using the public IP or DNS address of the EC2 instance that's running the DLAMI with Conda and the port that you opened for MXBoard:

    `http://127.0.0.1:8888`

More Info

To learn more about MXBoard, see the MXBoard website.

TensorBoard

TensorBoard lets you to visually inspect and interpret your TensorFlow runs and graphs. It runs a web server that serves a webpage for viewing and interacting with the TensorBoard visualizations.

TensorFlow and TensorBoard are preinstalled with the Deep Learning AMI with Conda (DLAMI with Conda). The DLAMI with Conda also includes an example script that uses TensorFlow to train an MNIST model with extra logging features enabled. MNIST is a database of handwritten numbers that is commonly used to train image recognition models. In this tutorial, you use the script to train an MNIST model, and TensorBoard and the logs to create visualizations.

Topics

Train an MNIST Model and Visualize the Training with TensorBoard

Visualize MNIST model training with TensorBoard

1. Connect to your Amazon Elastic Compute Cloud (Amazon EC2) instance of the DLAMI with Conda.
2. Activate the Python 2.7 TensorFlow environment and navigate to the directory that contains the folder with the TensorBoard example scripts:

```
$ source activate tensorflow_p27
$ cd ~/examples/tensorboard/
```

3. Run the script that trains an MNIST model with extended logging enabled:

```
$ python mnist_with_summaries.py
```

   The script writes the logs to `/tmp/tensorflow/mnist`.
4. Pass the location of the logs to `tensorboard`:

```
$ tensorboard --logdir=/tmp/tensorflow/mnist
```

   TensorBoard launches the visualization web server on port 6006.
5. For easy access from your local browser, you can change the web server port to port 80 or another port. Whichever port you use, you will need to open this port in the EC2 security group for your DLAMI. You can also use port forwarding. For instructions on changing your security group settings and port forwarding, see Set up a Jupyter Notebook Server (p. 15). The default settings are described in the next step.

   Note
   If you need to run both Jupyter server and a TensorBoard server, use a different port for each.
6. Open port 6006 (or the port you assigned to the visualization web server) on your EC2 instance.

   a. Open your EC2 instance in the Amazon EC2console at https://console.aws.amazon.com/ec2/.
   b. In the Amazon EC2 console, choose **Network & Security**, then choose**Security Groups**.
   c. For **Security Group**, , choose the one that was created most recently (see the time stamp in the description).
   d. Choose the **Inbound** tab, and choose **Edit**.
   e. Choose **Add Rule**.
   f. In the new row, type the followings:

      Type : Custom **`TCP Rule`**

      Protocol: **`TCP`**

      Port Range: **`6006`** (or the port that you assigned to the visualization server)

      Source: **`Custom IP (specify address/range)`**
7. Open the web page for the TensorBoard visualizations by using the public IP or DNS address of the EC2 instance that's running the DLAMI with Conda and the port that you opened for TensorBoard:

```
http:// YourInstancePublicDNS:6006
```

More Info

To learn more about TensorBoard, see the TensorBoard website.

Distributed Training

Learn about the options the DLAMI has for training with multiple GPUs. For increased performance, see Elastic Fabric Adapter (p. 62) Click on one of the options to learn how to use it.

Topics

Chainer

> **Note**
> We no longer include Chainer Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

Chainer is a flexible Python-based framework for easily and intuitively writing complex neural network architectures. Chainer makes it easy to use multi-GPU instances for training. Chainer also automatically logs results, graph loss and accuracy, and produces output for visualizing the neural network with a computational graph. It is included with the Deep Learning AMI with Conda (DLAMI with Conda).

The following topics show you how to train on multiple GPUs, a single GPU, and a CPU, create visualizations, and test your Chainer installation.

Topics

Training a Model with Chainer

This tutorial shows you how to use example Chainer scripts to train a model with the MNIST dataset. MNIST is a database of handwritten numbers that is commonly used to train image recognition models. The tutorial also shows the difference in training speed between training on a CPU and one or more GPUs.

Use Chainer to Train on Multiple GPUs

To train on multiple GPUs

1. Connect to the instance running Deep Learning AMI with Conda. Refer to the the section called "Instance Selection" (p. 7) or the Amazon EC2 documentation on how to select or connect to an instance. To run this tutorial, you will want to use an instance with at least two GPUs.
2. Activate the Python 3 Chainer environment:

```
$ source activate chainer_p36
```

3. To get the latest tutorials, clone the Chainer repository, and navigate to the examples folder:

```
(chainer_p36) :~$ cd ~/src
(chainer_p36) :~/src$ CHAINER_VERSION=v$(python -c "import chainer;
 print(chainer.__version__)")
(chainer_p36) :~/src$ git clone -b $CHAINER_VERSION https://github.com/chainer/
chainer.git
(chainer_p36) :~/src$ cd chainer/examples/mnist
```

4. Run the example in the `train_mnist_data_parallel.py` script. By default, the script uses the GPUs running on your instance of Deep Learning AMI with Conda. The script can be run on a maximum of two GPUs. It will ignore any GPUs past the first two. It detects one or both automatically. If you are running an instance without GPUs, skip to Use Chainer to Train with CPUs (p. 49), later in this tutorial.

```
(chainer_p36) :~/src/chainer/examples/mnist$ python train_mnist_data_parallel.py
```

Note
This example will return the following error due to the inclusion of a beta feature not included in the DLAMI.
`chainerx ModuleNotFoundError: No module named 'chainerx'`

While the Chainer script trains a model using the MNIST database, you see the results for each epoch.

Then you see example output as the script runs. The following example output was run on a p3.8xlarge instance. The script's output shows "GPU: 0, 1", which indicates that it is using the first two of the four available GPUs. The scripts typically use an index of GPUs starting with zero, instead of a total count.

```
GPU: 0, 1

# unit: 1000
# Minibatch-size: 400
# epoch: 20

epoch       main/loss   validation/main/loss  main/accuracy  validation/main/accuracy
 elapsed_time
1           0.277561    0.114709              0.919933       0.9654
 6.59261
2           0.0882352   0.0799204             0.973334       0.9752
 8.25162
3           0.0520674   0.0697055             0.983967       0.9786
 9.91661
4           0.0326329   0.0638036             0.989834       0.9805
 11.5767
5           0.0272191   0.0671859             0.9917         0.9796
 13.2341
```

```
6           0.0151008   0.0663898             0.9953         0.9813
 14.9068
7           0.0137765   0.0664415             0.995434       0.982
 16.5649
8           0.0116909   0.0737597             0.996          0.9801
 18.2176
9           0.00773858  0.0795216             0.997367       0.979
 19.8797
10          0.00705076  0.0825639             0.997634       0.9785
 21.5388
11          0.00773019  0.0858256             0.9978         0.9787
 23.2003
12          0.0120371   0.0940225             0.996034       0.9776
 24.8587
13          0.00906567  0.0753452             0.997033       0.9824
 26.5167
14          0.00852253  0.082996              0.996967       0.9812
 28.1777
15          0.00670928  0.102362              0.997867       0.9774
 29.8308
16          0.00873565  0.0691577             0.996867       0.9832
 31.498
17          0.00717177  0.094268              0.997767       0.9802
 33.152
18          0.00585393  0.0778739             0.998267       0.9827
 34.8268
19          0.00764773  0.107757              0.9975         0.9773
 36.4819
20          0.00620508  0.0834309             0.998167       0.9834
 38.1389
```

5. While your training is running it is useful to look at your GPU utilization. You can verify which GPUs are active and view their load. NVIDIA provides a tool for this, which can be run with the command `nvidia-smi`. However, it will only tell you a snapshot of the utilization, so it's more informative to combine this with the Linux command `watch`. The following command will use `watch` with `nvidia-smi` to refresh the current GPU utilization every tenth of a second. Open up another terminal session to your DLAMI, and run the following command:

```
(chainer_p36) :~$ watch -n0.1 nvidia-smi
```

You will see an output similar to the following result. Use `ctrl-c` to close the tool, or just keep it running while you try out other examples in your first terminal session.

```

Every 0.1s: nvidia-smi                                    Wed Feb 28 00:28:50 2018

Wed Feb 28 00:28:50 2018
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 384.111                Driver Version: 384.111                   |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000000:00:1B.0 Off |                    0 |
| N/A   46C    P0    56W / 300W |    728MiB / 16152MiB |     10%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  On   | 00000000:00:1C.0 Off |                    0 |
| N/A   44C    P0    53W / 300W |    696MiB / 16152MiB |      4%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  On   | 00000000:00:1D.0 Off |                    0 |
| N/A   42C    P0    38W / 300W |     10MiB / 16152MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  On   | 00000000:00:1E.0 Off |                    0 |
| N/A   46C    P0    40W / 300W |     10MiB / 16152MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     54418      C   python                                       718MiB |
|    1     54418      C   python                                       686MiB |
+-----------------------------------------------------------------------------+
```

In this example, GPU 0 and GPU 1 are active, and GPU 2 and 3 are not. You can also see memory utilization per GPU.

6. As training completes, note the elapsed time in your first terminal session. In the example, elapsed time is 38.1389 seconds.

Use Chainer to Train on a Single GPU

This example shows how to train on a single GPU. You might do this if you have only one GPU available or just to see how multi-GPU training might scale with Chainer.

To use Chainer to train on a single GPU

- For this example, you use another script, `train_mnist.py`, and tell it to use just GPU 0 with the `--gpu=0` argument. To see how a different GPUs activate in the `nvidia-smi` console, you can tell the script to use GPU number 1 by using `--gpu=1` .

```
(chainer_p36) :~/src/chainer/examples/mnist$ python train_mnist.py --gpu=0
```

```
GPU: 0
# unit: 1000
# Minibatch-size: 100
# epoch: 20

epoch       main/loss   validation/main/loss  main/accuracy  validation/main/accuracy
 elapsed_time
1           0.192348    0.0909235             0.940934       0.9719
 5.3861
2           0.0746767   0.069854              0.976566       0.9785
 8.97146
3           0.0477152   0.0780836             0.984982       0.976
 12.5596
4           0.0347092   0.0701098             0.988498       0.9783
 16.1577
5           0.0263807   0.08851               0.991515       0.9793
 19.7939
6           0.0253418   0.0945821             0.991599       0.9761
 23.4643
7           0.0209954   0.0683193             0.993398       0.981
 27.0317
8           0.0179036   0.080285              0.994149       0.9819
 30.6325
9           0.0183184   0.0690474             0.994198       0.9823
 34.2469
10          0.0127616   0.0776328             0.996165       0.9814
 37.8693
11          0.0145421   0.0970157             0.995365       0.9801
 41.4629
```

```
12          0.0129053   0.0922671             0.995899       0.981
 45.0233
13          0.0135988   0.0717195             0.995749       0.9857
 48.6271
14          0.00898215  0.0840777             0.997216       0.9839
 52.2269
15          0.0103909   0.123506              0.996832       0.9771
 55.8667
16          0.012099    0.0826434             0.996616       0.9847
 59.5001
17          0.0066183   0.101969              0.997999       0.9826
 63.1294
18          0.00989864  0.0877713             0.997116       0.9829
 66.7449
19          0.0101816   0.0972672             0.996966       0.9822
 70.3686
20          0.00833862  0.0899327             0.997649       0.9835
 74.0063
```

In this example, running on a single GPU took almost twice as long! Training larger models or larger datasets will yield different results from this example, so experiment to further evaluate GPU performance.

Use Chainer to Train with CPUs

Now try training on a CPU-only mode. Run the same script, `python train_mnist.py`, without arguments:

```
(chainer_p36) :~/src/chainer/examples/mnist$ python train_mnist.py
```

In the output, `GPU: -1` indicates that no GPU is used:

```
GPU: -1
# unit: 1000
# Minibatch-size: 100
# epoch: 20

epoch       main/loss   validation/main/loss  main/accuracy  validation/main/accuracy
 elapsed_time
1           0.192083    0.0918663             0.94195        0.9712
 11.2661
2           0.0732366   0.0790055             0.977267       0.9747
 23.9823
3           0.0485948   0.0723766             0.9844         0.9787
 37.5275
4           0.0352731   0.0817955             0.987967       0.9772
 51.6394
5           0.029566    0.0807774             0.990217       0.9764
 65.2657
6           0.025517    0.0678703             0.9915         0.9814
 79.1276
7           0.0194185   0.0716576             0.99355        0.9808
 93.8085
8           0.0174553   0.0786768             0.994217       0.9809
 108.648
9           0.0148924   0.0923396             0.994983       0.9791
 123.737
10          0.018051    0.099924              0.99445        0.9791
 139.483
11          0.014241    0.0860133             0.995783       0.9806
 156.132
```

```
12          0.0124222   0.0829303             0.995967       0.9822
 173.173
13          0.00846336  0.122346              0.997133       0.9769
 190.365
14          0.011392    0.0982324             0.996383       0.9803
 207.746
15          0.0113111   0.0985907             0.996533       0.9813
 225.764
16          0.0114328   0.0905778             0.996483       0.9811
 244.258
17          0.00900945  0.0907504             0.9974         0.9825
 263.379
18          0.0130028   0.0917099             0.996217       0.9831
 282.887
19          0.00950412  0.0850664             0.997133       0.9839
 303.113
20          0.00808573  0.112367              0.998067       0.9778
 323.852
```

In this example, MNIST was trained in 323 seconds, which is more than 11x longer than training with two GPUs. If you've ever doubted the power of GPUs, this example shows how much more efficient they are.

Graphing Results

Chainer also automatically logs results, graph loss and accuracy, and produces output for plotting the computational graph.

To generate the computational graph

1. After any training run finishes, you may navigate to the `result` directory and view the run's accuracy and loss in the form of two automatically generated images. Navigate there now, and list the contents:

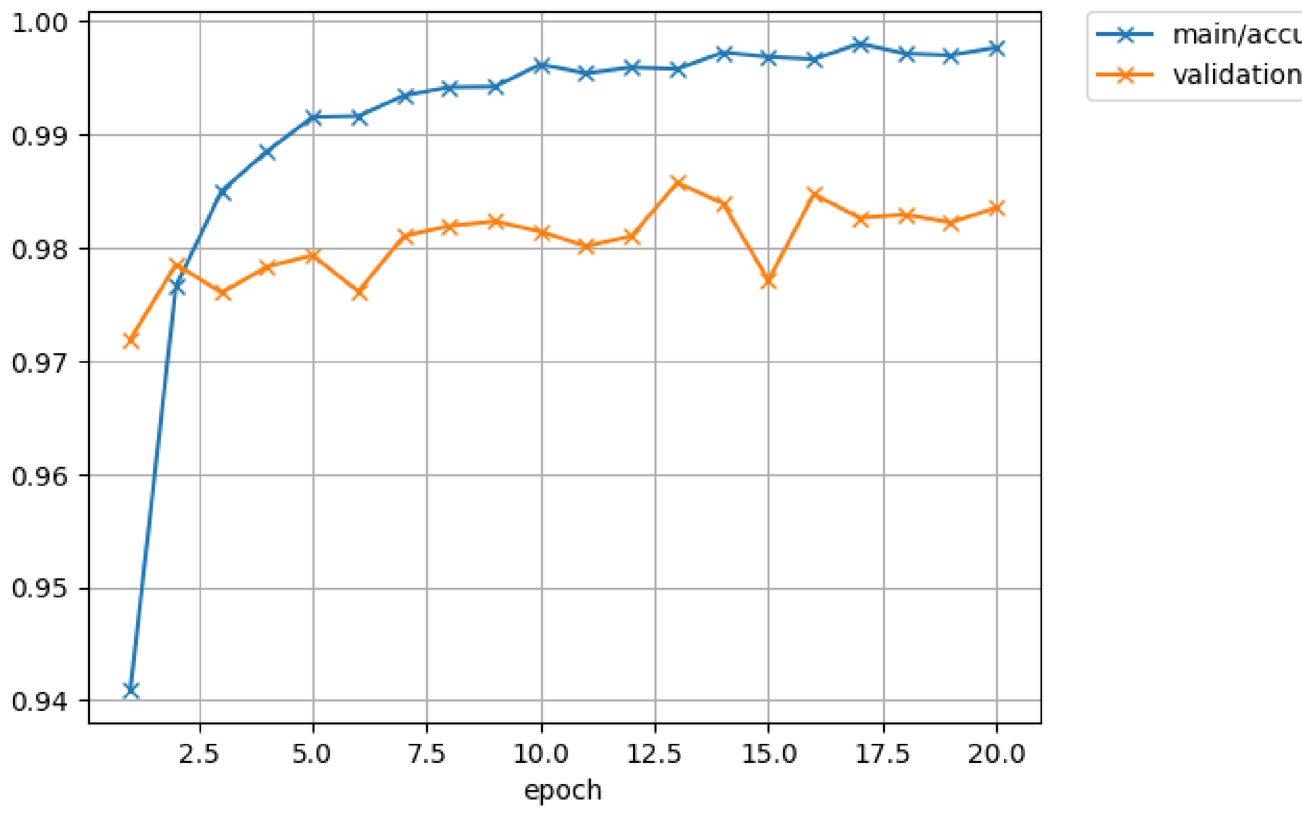
   ```
   (chainer_p36) :~/src/chainer/examples/mnist$ cd result
   (chainer_p36) :~/src/chainer/examples/mnist/result$ ls
   ```

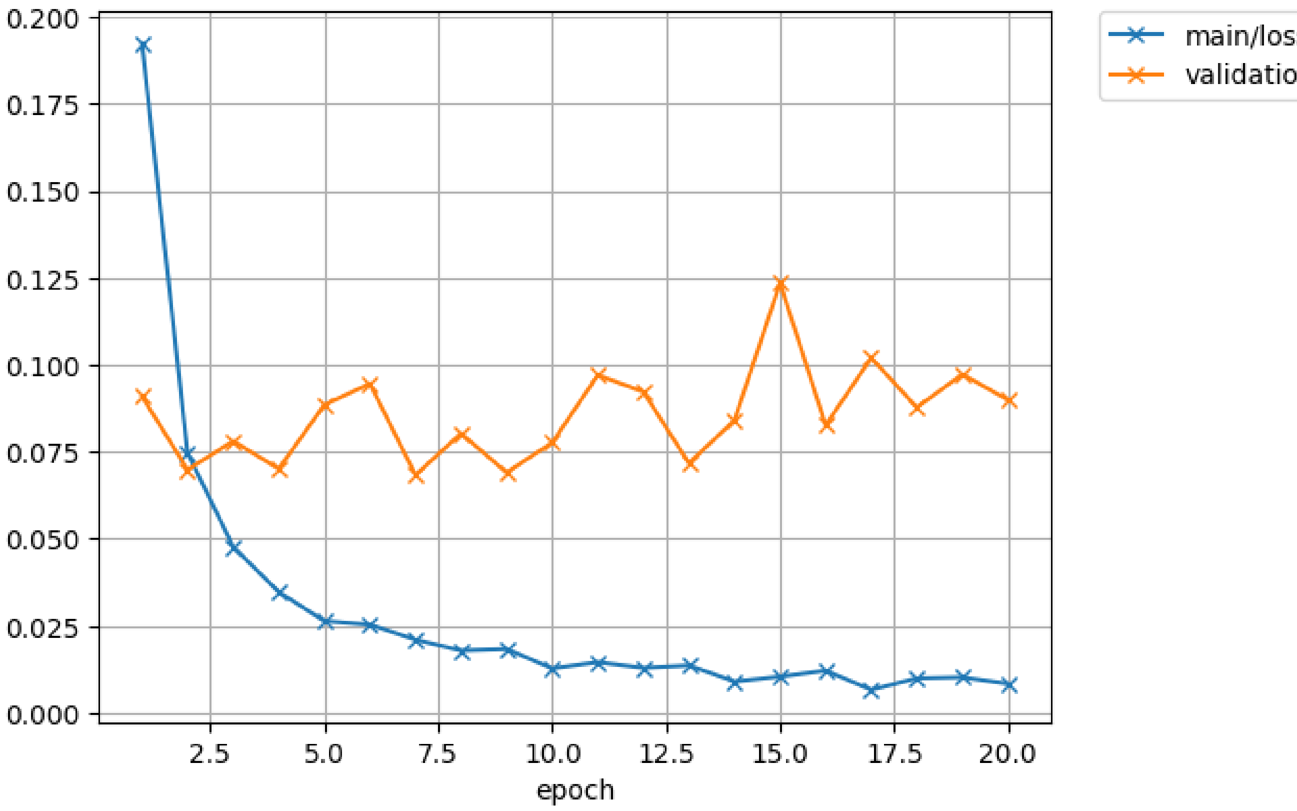
   The `result` directory contains two files in .png format: `accuracy.png` and `loss.png`.
2. To view the graphs, use the `scp` command to copy them to your local computer.

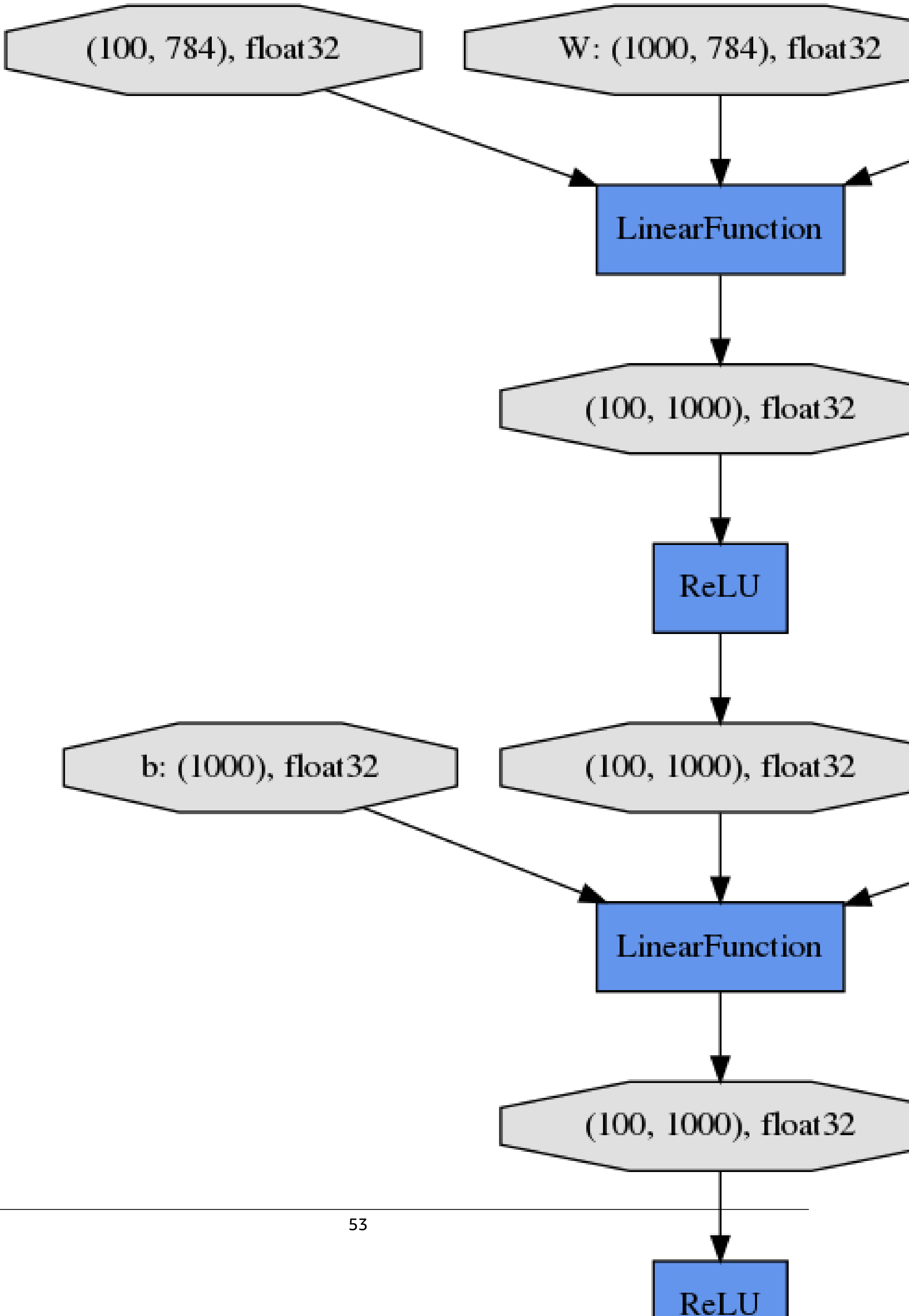
   In a macOS terminal, running the following `scp` command downloads all three files to your `Downloads` folder. Replace the placeholders for the location of the key file and server address with your information. For other operating systems, use the appropriate `scp` command format. Note, for an Amazon Linux AMI, the user name is ec2-user.

   ```
   (chainer_p36) :~/src/chainer/examples/mnist/result$ scp -i "your-key-file.pem"
    ubuntu@your-dlami-address.compute-1.amazonaws.com:~/src/chainer/examples/mnist/result/
   *.png ~/Downloads
   ```

The following images are examples of accuracy, loss, and computational graphs, respectively.


1.00
0.99
0.98
0.97
0.96
0.95
0.94
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
epoch
main/accu
validation


0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
epoch
main/los
validatio


(100, 784), float32
W: (1000, 784), float32
LinearFunction
(100, 1000), float32
ReLU
b: (1000), float32
(100, 1000), float32
LinearFunction
(100, 1000), float32
ReLU

Testing Chainer

To test Chainer and verify GPU support with a preinstalled test script, run the following command:

```
(chainer_p36) :~/src/chainer/examples/mnist/result$ cd ~/src/bin
(chainer_p36) :~/src/bin$ ./testChainer
```

This downloads Chainer source code and runs the Chainer multi-GPU MNIST example.

More Info

To learn more about Chainer, see the Chainer documentation website. The `Chainer` examples folder contains more examples. Try them to see how they perform.

Keras with MXNet

This tutorial shows how to activate and use Keras 2 with the MXNet backend on a Deep Learning AMI with Conda.

Activate Keras with the MXNet backend and test it on the DLAMI with Conda

1. To activate Keras with the MXNet backend, open an Amazon Elastic Compute Cloud (Amazon EC2) instance of the DLAMI with Conda.
   - For Python 3, run this command:

     ```
     $ source activate mxnet_p36
     ```
   - For Python 2, run this command:

     ```
     $ source activate mxnet_p27
     ```
2. Start the iPython terminal:

   ```
   (mxnet_p36)$ ipython
   ```
3. Test importing Keras with MXNet to verify that it is working properly:

   ```
   import keras as k
   ```

   The following should appear on your screen (possibly after a few warning messages).

   ```
   Using MXNet backend
   ```

   Note
   If you get an error, or if the TensorFlow backend is still being used, you need to update your Keras config manually. Edit the `~/.keras/keras.json` file and change the backend setting to `mxnet`.

Keras-MXNet Multi-GPU Training Tutorial

Train a convolutional neural network (CNN)

1. Open a terminal and SSH into your DLAMI.

2. Navigate to the `~/examples/keras-mxnet/` folder.
3. Run `nvidia-smi` in your terminal window to determine the number of available GPUs on your DLAMI. In the next step, you will run the script as-is if you have four GPUs.
4. (Optional) Run the following command to open the script for editing.

```
(mxnet_p36)$ vi cifar10_resnet_multi_gpu.py
```

5. (Optional) The script has the following line that defines the number of GPUs. Update it if necessary.

```
model = multi_gpu_model(model, gpus=4)
```

6. Now, run the training.

```
(mxnet_p36)$ python cifar10_resnet_multi_gpu.py
```

Note
Keras-MXNet runs up to two times faster with the `channels_first` image_data_format set. To change to `channels_first`, edit your Keras config file (`~/.keras/keras.json`) and set the following: `"image_data_format": "channels_first"`.
For more performance tuning techniques, see Keras-MXNet performance tuning guide.

More Info

- You can find examples for Keras with a MXNet backend in the Deep Learning AMI with Conda `~/examples/keras-mxnet` directory.
- For even more tutorials and examples, see the Keras-MXNet GitHub project.

TensorFlow with Horovod

This tutorial shows how to use TensorFlow with Horovod on a Deep Learning AMI with Conda. Horovod is preinstalled in the Conda environments for TensorFlow. The Python 3 environment is recommended. The instructions here assume you have a working DLAMI instance with one or more GPUs. For more information, see How to Get Started with the DLAMI (p. 4).

Note
Only P3.*, P2.*, and G3.* instance types are supported.

Note
There are two locations where mpirun (via OpenMPI) is available. It is available in `/usr/bin` and `/home/ubuntu/anaconda3/envs/<env>/bin`. `env` is an environment corresponding to the framework, such as Tensorflow and Apache MXNet. The newer OpenMPI versions are available in the conda environments. We recommend using the absolute path of the mpirun binary or the --prefix flag to run mpi workloads. For example, with the Tensorflow python36 environment, use either:

```
/home/ubuntu/anaconda3/envs/tensorflow_p36/bin/mpirun <args>

or

mpirun --prefix /home/ubuntu/anaconda3/envs/tensorflow_p36/bin <args>
```

Activate and Test TensorFlow with Horovod

1. Verify that your instance has active GPUs. NVIDIA provides a tool for this:

```
$ nvidia-smi
```

2. Activate the Python 3 TensorFlow environment:

```
$ source activate tensorflow_p36
```

3. Start the iPython terminal:

```
(tensorflow_p36)$ ipython
```

4. Test importing TensorFlow with Horovod to verify that it is working properly:

```
import horovod.tensorflow as hvd
hvd.init()
```

   The following may appear on your screen (possibly after a few warning messages).

```

--------------------------------------------------------------------------
[[55425,1],0]: A high-performance Open MPI point-to-point messaging module
was unable to find any relevant network interfaces:

Module: OpenFabrics (openib)
  Host: ip-172-31-72-4

Another transport will be used instead, although this may result in
lower performance.
--------------------------------------------------------------------------
```

Configure Your Horovod Hosts File

You can use Horovod for single-node, multi-GPU training, or for multiple-node, multi-GPU training. If you plan to use multiple nodes for distributed training, you must add each DLAMI private IP address to a hosts file. The DLAMI you are currently logged into is referred to as the leader. Other DLAMI instances that are part of the cluster are referred to as members.

Before you start this section, launch one or more DLAMI, and wait for them to be in the **Ready** state. The example scripts expect a hosts file, so even if you plan to use only one DLAMI, create a hosts file with only one entry. If you edit the hosts file after training commences, you must restart training for added or removed hosts to take effect.

To configure Horovod for training

1. Change directories to where the training scripts reside.

```
cd ~/examples/horovod/tensorflow
```

2. Use vim to edit a file in the leader's home directory.

```
vim hosts
```

3. Select one of the members in the Amazon Elastic Compute Cloud console, and the description pane of the console appears. Find the **Private IPs** field and copy the IP and paste it in a text file. Copy each member's private IP on a new line. Then, next to each IP, add a space and then the text `slots=8` as shown below. This represents how many GPUs each instance has. The p3.16xlarge

instances have 8 GPUs, so if you chose a different instance type, you would provide the actual number of GPUs for each instance. For the leader you can use `localhost`. With a cluster of 4 nodes, it should look similar to the following:

```
172.100.1.200 slots=8
172.200.8.99 slots=8
172.48.3.124 slots=8
localhost slots=8
```

Save the file and exit back to the leader's terminal.

4. Add the SSH key used by the member instances to the ssh-agent.

```
eval `ssh-agent -s`
ssh-add <key_name>.pem
```

5. Now your leader knows how to reach each member. This is all going to happen on the private network interfaces. Next, use a short bash function to help send commands to each member.

```
function runclust(){ while read -u 10 host; do host=${host%% slots*}; ssh -o
 "StrictHostKeyChecking no" $host ""$2""; done 10<$1; };
```

6. Tell the other members not to do “StrickHostKeyChecking” as this may cause training to stop responding.

```
runclust hosts "echo \"StrictHostKeyChecking no\" >> ~/.ssh/config"
```

Train with Synthetic Data

Your DLAMI ships with an example script to train a model with synthetic data. This tests whether your leader can communicate with the members of the cluster. A hosts file is required. Refer to Configure Your Horovod Hosts File (p. 56) for instructions.

To test Horovod training with example data

1. `~/examples/horovod/tensorflow/train_synthetic.sh` defaults to 8 GPUs, but you can provide it the number of GPUs you want to run. The following example runs the script, passing 4 as a parameter for 4 GPUs.

```
$ ./train_synthetic.sh 4
```

After some warning messages, you see the following output that verifies Horovod is using 4 GPUs.

```
PY3.6.5 |Anaconda custom (64-bit)| (default, Apr 29 2018, 16:14:56) [GCC
 7.2.0]TF1.11.0Horovod size: 4
```

Then, after some other warnings, you see the start of a table and some data points. If you don't want to watch for 1,000 batches, break out of the training.

```

    Step Epoch  Speed  Loss   FinLoss LR
    0    0.0    105.6  6.794   7.708 6.40000
    1    0.0    311.7  0.000   4.315 6.38721
    100    0.1   3010.2  0.000 34.446 5.18400
    200    0.2   3013.6  0.000 13.077 4.09600
    300    0.2   3012.8  0.000  6.196 3.13600
    400    0.3   3012.5   0.000   3.551 2.30401
```

2. Horovod uses all local GPUs first before attempting to use the GPUs of the members of the cluster. So, to make sure distributed training across the cluster is working, try out the full number of GPUs you intend to use. If, for example, you have 4 members that are p3.16xlarge instance type, you have 32 GPUs across your cluster. This is where you would want to try out the full 32GPUs.

   ```
   ./train_synthetic.sh 32
   ```

   Your output is similar to the previous test. The Horovod size is 32, and roughly four-times the speed. With this experimentation completed, you have tested your leader and its ability to communicate with the members. If you run into any issues, check the Troubleshooting (p. 61) section.

Prepare the ImageNet Dataset

In this section, you download the ImageNet dataset, then generate a TFRecord-format dataset from the raw dataset. A set of preprocessing scripts is provided on the DLAMI for the ImageNet dataset that you can use for either ImageNet or as a template for another dataset. The main training scripts that are configured for ImageNet are also provided. The following section assumes that you have launched a DLAMI with an EC2 instance with 8 GPUs. We recommend the p3.16xlarge instance type.

In the `~/examples/horovod/tensorflow/utils` directory on your DLAMI you find the following scripts:

- `utils/preprocess_imagenet.py` - Use this to convert the raw ImageNet dataset to the `TFRecord` format.
- `utils/tensorflow_image_resizer.py` - Use this to resize the `TFRecord` dataset as recommended for ImageNet training.

Prepare the ImageNet Dataset

1. Visit image-net.org, create an account, acquire an access key, and download the dataset. image-net.org hosts the raw dataset. To download it, you are required to have an ImageNet account and an access key. The account is free, and to get the free access key you must agree to the ImageNet license.
2. Use the image preprocessing script to generate a TFRecord format dataset from the raw ImageNet dataset. From the `~/examples/horovod/tensorflow/utils` directory:

   ```
   python preprocess_imagenet.py \
            --local_scratch_dir=[YOUR DIRECTORY] \
            --imagenet_username=[imagenet account] \
            --imagenet_access_key=[imagenet access key]
   ```
3. Use the image resizing script. If you resize the images, training runs more quickly and better aligns with the ResNet reference paper. From the `~/examples/horovod/utils/preprocess` directory:

   ```
   python tensorflow_image_resizer.py \
             -d imagenet \
             -i [PATH TO TFRECORD TRAINING DATASET] \
             -o  [PATH TO RESIZED TFRECORD TRAINING DATASET] \
             --subset_name train \
             --num_preprocess_threads 60 \
             --num_intra_threads 2 \
             --num_inter_threads 2
   ```

Train a ResNet-50 ImageNet Model on a Single DLAMI

Note

- The script in this tutorial expects the preprocessed training data to be in the `~/data/tf-imagenet/` folder. Refer to Prepare the ImageNet Dataset (p. 58) for instructions.
- A hosts file is required. Refer to Configure Your Horovod Hosts File (p. 56) for instructions.

Use Horovod to Train a ResNet50 CNN on the ImageNet Dataset

1. Navigate to the `~/examples/horovod/tensorflow` folder.

```
cd ~/examples/horovod/tensorflow
```

2. Verify your configuration and set the number of GPUs to use in training. First, review the `hosts` that is in the same folder as the scripts. This file must be updated if you use an instance with fewer than 8 GPUs. By default it says `localhost slots=8`. Update the number 8 to be the number of GPUs you want to use.
3. A shell script is provided that takes the number of GPUs you plan to use as its only parameter. Run this script to start training. The example below uses 4 for four GPUs.

```
./train.sh 4
```

4. It takes several hours to finish. It uses `mpirun` to distribute the training across your GPUs.

Train a ResNet-50 ImageNet Model on a Cluster of DLAMIs

Note

- The script in this tutorial expects the preprocessed training data to be in the `~/data/tf-imagenet/` folder. Refer to Prepare the ImageNet Dataset (p. 58) for instructions.
- A hosts file is required. Refer to Configure Your Horovod Hosts File (p. 56) for instructions.

This example walks you through training a ResNet-50 model on a prepared dataset across multiple nodes in a cluster of DLAMIs.

- For faster performance, we recommend that you have the dataset locally on each member of the cluster.

  Use this `copyclust` function to copy data to other members.

```
function copyclust(){ while read -u 10 host; do host=${host%% slots*}; rsync -azv "$2"
 $host:"$3"; done 10<$1; };
```

Or, if you have the files sitting in an S3 bucket, use the `runclust` function to download the files to each member directly.

```
runclust hosts "tmux new-session -d \"export AWS_ACCESS_KEY_ID=YOUR_ACCESS_KEY && export
 AWS_SECRET_ACCESS_KEY=YOUR_SECRET && aws s3 sync s3://your-imagenet-bucket ~/data/tf-
imagenet/ && aws s3 sync s3://your-imagenet-validation-bucket ~/data/tf-imagenet/\""
```

Using tools that let you manage multiple nodes at once is a great time-saver. You can either wait for each step and manage each instance separately, or use tools such as tmux or screen to let you disconnect and resume sessions.

After the copying is completed, you're ready to start training. Run the script, passing 32 as a parameter for the 32 GPUs we're using for this run. Use tmux or similar tool if you're concerned about disconnecting and terminating your session, which would end the training run.

```
./train.sh 32
```

The following output is what you see when running the training on ImageNet with 32 GPUs. Thirty-two GPUs take 90–110 minutes.

```

    Step Epoch  Speed  Loss   FinLoss LR
    0   0.0   440.6  6.935  7.850 0.00100
    1   0.0  2215.4  6.923  7.837 0.00305
    50   0.3 19347.5  6.515  7.425 0.10353
    100   0.6 18631.7  6.275  7.173 0.20606
    150   1.0 19742.0  6.043  6.922 0.30860
    200   1.3 19790.7  5.730  6.586 0.41113
    250   1.6 20309.4  5.631  6.458 0.51366
    300   1.9 19943.9  5.233  6.027 0.61619
    350   2.2 19329.8  5.101  5.864 0.71872
    400   2.6 19605.4  4.787  5.519 0.82126
    ...
    13750  87.9 19398.8  0.676  1.082 0.00217
    13800  88.2 19827.5  0.662  1.067 0.00156
    13850  88.6 19986.7  0.591  0.997 0.00104
    13900  88.9 19595.1  0.598  1.003 0.00064
    13950  89.2 19721.8  0.633  1.039 0.00033
    14000  89.5 19567.8  0.567  0.973 0.00012
    14050  89.8 20902.4  0.803  1.209 0.00002
    Finished in 6004.354426383972
```

After a training run is completed, the script follows up with an evaluation run. It runs on the leader because it runs quickly enough without having to distribute the job to the other members. The following is the output of the evaluation run.

```
Horovod size: 32
Evaluating
Validation dataset size: 50000
[ip-172-31-36-75:54959] 7 more processes have sent help message help-btl-vader.txt / cma-
permission-denied
[ip-172-31-36-75:54959] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help /
 error messages
  step  epoch  top1    top5     loss   checkpoint_time(UTC)
 14075   90.0  75.716   92.91    0.97  2018-11-14 08:38:28
```

The following is an example output when this script is run with 256 GPUs where the runtime was between 14 and 15 minutes.

```

  Step Epoch  Speed  Loss   FinLoss LR
  1400  71.6 143451.0  1.189  1.720 0.14850
  1450  74.2 142679.2  0.897  1.402 0.10283
  1500  76.7 143268.6  1.326  1.809 0.06719
  1550  79.3 142660.9  1.002  1.470 0.04059
  1600  81.8 143302.2  0.981  1.439 0.02190
  1650  84.4 144808.2  0.740  1.192 0.00987
  1700  87.0 144790.6  0.909  1.359 0.00313
  1750  89.5 143499.8  0.844  1.293 0.00026
Finished in 860.5105031204224

Finished evaluation
1759   90.0  75.086   92.47    0.99  2018-11-20 07:18:18
```

Troubleshooting

The following command may help get past errors that come up when you experiment with Horovod.

- If the training crashes for some reason, mpirun may fail to clean up all the python processes on each machine. In that case, before you start the next job, stop the python processes on all machines as follows:

```
runclust hosts "pkill -9 python"
```

- If the process finishes abruptly without error, try deleting your log folder.

```
runclust hosts "rm -rf ~/imagenet_resnet/"
```

- If other unexplained issues pop up, check your disk space. If you're out, try removing the logs folder since that is full of checkpoints and data. You can also increase the size of the volumes for each member.

```
runclust hosts "df /"
```

- As a last resort you can also try rebooting.

```
runclust hosts "sudo reboot"
```

You may receive the following error code if you try to use TensorFlow with Horovod on an unsupported instance type:

```
---------------------------------------------------------------------------
NotFoundError Traceback (most recent call last)
<ipython-input-3-e90ed6cabab4> in <module>()
----> 1 import horovod.tensorflow as hvd

~/anaconda3/envs/tensorflow_p36/lib/python3.6/site-packages/horovod/tensorflow/__init__.py
 in <module>()
** *34* check_extension('horovod.tensorflow', 'HOROVOD_WITH_TENSORFLOW', __file__,
 'mpi_lib')
** *35*
---> 36 from horovod.tensorflow.mpi_ops import allgather, broadcast, _allreduce
** *37* from horovod.tensorflow.mpi_ops import init, shutdown
** *38* from horovod.tensorflow.mpi_ops import size, local_size, rank, local_rank

~/anaconda3/envs/tensorflow_p36/lib/python3.6/site-packages/horovod/tensorflow/mpi_ops.py
 in <module>()
** *56*
** *57* MPI_LIB = _load_library('mpi_lib' + get_ext_suffix(),
---> 58 ['HorovodAllgather', 'HorovodAllreduce'])
** *59*
** *60* _basics = _HorovodBasics(__file__, 'mpi_lib')

~/anaconda3/envs/tensorflow_p36/lib/python3.6/site-packages/horovod/tensorflow/mpi_ops.py
 in _load_library(name, op_list)
** *43* """
** *44* filename = resource_loader.get_path_to_datafile(name)
---> 45 library = load_library.load_op_library(filename)
** *46* for expected_op in (op_list or []):
** *47* for lib_op in library.OP_LIST.op:

~/anaconda3/envs/tensorflow_p36/lib/python3.6/site-packages/tensorflow/python/framework/
load_library.py in load_op_library(library_filename)
** *59* RuntimeError: when unable to load the library or get the python wrappers.
** *60* """
---> 61 lib_handle = py_tf.TF_LoadLibrary(library_filename)
** *62*
** *63* op_list_str = py_tf.TF_GetOpList(lib_handle)

NotFoundError: /home/ubuntu/anaconda3/envs/tensorflow_p36/lib/python3.6/site-packages/
horovod/tensorflow/mpi_lib.cpython-36m-x86_64-linux-gnu.so: undefined symbol:
 _ZN10tensorflow14kernel_factory17OpKernelRegistrar12InitInternalEPKNS_9KernelDefEN4absl11string_viewEP
```

More Info

For utilities and examples, see the `~/examples/horovod` folder in the home directory of the DLAMI.

For even more tutorials and examples, see the Horovod GitHub project.

Elastic Fabric Adapter

An Elastic Fabric Adapter (EFA) is a network device that you can attach to your DLAMI instance to accelerate High Performance Computing (HPC) applications. EFA enables you to achieve the application performance of an on-premises HPC cluster, with the scalability, flexibility, and elasticity provided by the AWS Cloud.

The following topics show you how to get started using EFA with the DLAMI.

Topics

Launching a AWS Deep Learning AMI Instance With EFA

The latest DLAMI is ready to use with EFA and comes with the required drivers, kernel modules, libfabric, openmpi and the NCCL OFI plugin for GPU instances.

Supported CUDA Versions: NCCL Applications with EFA are only supported on CUDA-10.0, CUDA-10.1, CUDA-10.2, and CUDA-11.0 because the NCCL OFI plugin requires a NCCL version > 2.4.2.

Note:

- When running a NCCL Application using `mpirun` on EFA, you will have to specify the full path to the EFA supported installation as:

```
/opt/amazon/openmpi/bin/mpirun <command>
```

- To enable your application to use EFA, add `FI_PROVIDER="efa"` to the `mpirun` command as shown in Using EFA on the DLAMI (p. 65).

Topics

Prepare an EFA Enabled Security Group

EFA requires a security group that allows all inbound and outbound traffic to and from the security group itself. For more information, see the EFA Documentation.

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. In the navigation pane, choose **Security Groups** and then choose **Create Security Group**.
3. In the **Create Security Group** window, do the following:
   - For **Security group name**, enter a descriptive name for the security group, such as `EFA-enabled security group`.
   - (Optional) For **Description**, enter a brief description of the security group.
   - For **VPC**, select the VPC into which you intend to launch your EFA-enabled instances.
   - Choose **Create**.
4. Select the security group that you created, and on the **Description** tab, copy the **Group ID**.
5. On the **Inbound** and **Outbound** tabs, do the following:
   - Choose **Edit**.
   - For **Type**, choose **All traffic**.
   - For **Source**, choose **Custom**.
   - Paste the security group ID that you copied into the field.
   - Choose **Save**.
6. Enable inbound traffic referring to Authorizing Inbound Traffic for Your Linux Instances. If you skip this step, you won't be able to communicate with your DLAMI instance.

Launch Your Instance

EFA on the AWS Deep Learning AMI is currently supported with the following instance types and operating systems:

- P3dn.24xlarge: Amazon Linux, Amazon Linux 2, Ubuntu 16.04, and Ubuntu 18.04
- P4d.24xlarge: Amazon Linux 2, Ubuntu 16.04, and Ubuntu 18.04

The following section shows how to launch an EFA enabled DLAMI instance. For more information on launching an EFA enabled instance, see Launch EFA-Enabled Instances into a Cluster Placement Group.

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
2. Choose **Launch Instance**.
3. On the **Choose an AMI** page, Select the ‘Deep Learning AMI (Ubuntu 18.04) Version 25.0
4. On the **Choose an Instance Type** page, select one of the following supported instance types and then choose **Next: Configure Instance Details.** Refer to this link for the list of supported instances: Get started with EFA and MPI
5. On the **Configure Instance Details** page, do the following:
   - For **Number of instances**, enter the number of EFA-enabled instances that you want to launch.
   - For **Network** and **Subnet**, select the VPC and subnet into which to launch the instances.
   - [Optional] For **Placement group**, select **Add instance to placement group**. For best performance, launch the instances within a placement group.

- [Optional] For **Placement group name**, select **Add to a new placement group**, enter a descriptive name for the placement group, and then for **Placement group strategy**, select **cluster**.
- Make sure to enable the **"Elastic Fabric Adapter"** on this page. If this option is disabled, change the subnet to one that supports your selected instance type.
- In the **Network Interfaces** section, for device **eth0**, choose **New network interface**. You can optionally specify a primary IPv4 address and one or more secondary IPv4 addresses. If you're launching the instance into a subnet that has an associated IPv6 CIDR block, you can optionally specify a primary IPv6 address and one or more secondary IPv6 addresses.
- Choose **Next: Add Storage**.

6. On the **Add Storage** page, specify the volumes to attach to the instances in addition to the volumes specified by the AMI (such as the root device volume), and then choose **Next: Add Tags**.
7. On the **Add Tags** page, specify tags for the instances, such as a user-friendly name, and then choose **Next: Configure Security Group**.
8. On the **Configure Security Group** page, for **Assign a security group**, select **Select an existing security group**, and then select the security group that you created previously**.**
9. Choose **Review and Launch**.
10. On the **Review Instance Launch** page, review the settings, and then choose **Launch** to choose a key pair and to launch your instances.

Verify EFA Attachment

From the Console

After launching the instance, check the instance details in the AWS Console. To do this, select the instance in the EC2 console and look at the Description Tab in the lower pane on the page. Find the parameter 'Network Interfaces: eth0' and click on eth0 which opens a pop-up. Make sure that 'Elastic Fabric Adapter' is enabled.

If EFA is not enabled, you can fix this by either:

- Terminating the EC2 instance and launching a new one with the same steps. Make sure the EFA is attached.
- Attach EFA to an existing instance.

  1. In the EC2 Console, go to Network Interfaces.
  2. Click on Create a Network Interface.
  3. Select the same subnet that your instance is in.
  4. Make sure to enable the 'Elastic Fabric Adapter' and click on Create.
  5. Go back to the EC2 Instances Tab and select your instance.
  6. Go to Actions: Instance State and stop the instance before you attach EFA.
  7. From Actions, select Networking: Attach Network Interface.
  8. Select the interface you just created and click on attach.
  9. Restart your instance.

From the Instance

The following test script is already present on the DLAMI. Run it to ensure that the kernel modules are loaded correctly.

```
$ fi_info -p efa
```

Your output should look similar to the following.

```
provider: efa
    fabric: EFA-fe80::e5:56ff:fe34:56a8
    domain: efa_0-rdm
    version: 2.0
    type: FI_EP_RDM
    protocol: FI_PROTO_EFA
provider: efa
    fabric: EFA-fe80::e5:56ff:fe34:56a8
    domain: efa_0-dgrm
    version: 2.0
    type: FI_EP_DGRAM
    protocol: FI_PROTO_EFA
provider: efa;ofi_rxd
    fabric: EFA-fe80::e5:56ff:fe34:56a8
    domain: efa_0-dgrm
    version: 1.0
    type: FI_EP_RDM
    protocol: FI_PROTO_RXD
```

Verify Security Group Configuration

The following test script is already present on the DLAMI. Run it to ensure that the security group you created is configured correctly.

```
$ cd ~/src/bin/efa-tests/
$ ./efa_test.sh
```

Your output should look similar to the following.

```
Starting server...
Starting client...
bytes   #sent   #ack     total       time     MB/sec    usec/xfer   Mxfers/sec
64      10      =10      1.2k        0.02s      0.06    1123.55       0.00
256     10      =10      5k          0.00s     17.66      14.50       0.07
1k      10      =10      20k         0.00s     67.81      15.10       0.07
4k      10      =10      80k         0.00s    237.45      17.25       0.06
64k     10      =10      1.2m        0.00s    921.10      71.15       0.01
1m      10      =10      20m         0.01s   2122.41     494.05       0.00
```

If it stops responding or does not complete, ensure that your security group has the correct inbound/outbound rules.

Using EFA on the DLAMI

The following section describes how to use EFA to run multi-node applications on the AWS Deep Learning AMI.

Topics

Running Multi-Node Applications with EFA

To run an application across a cluster of nodes some configuration is needed.

Enable Passwordless SSH

Select one node in your cluster as the leader node. The remaining nodes are referred to as the member nodes.

1. On the leader node, generate the RSA keypair.

   ```
   ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa
   ```

2. Change the permissions of the private key on the leader node.

   ```
   chmod 600 ~/.ssh/id_rsa
   ```

3. Copy the public key `~/.ssh/id_rsa.pub` to and append it to `~/.ssh/authorized_keys` of the member nodes in the cluster.
4. You should now be able to directly login to the member nodes from the leader node using the private ip.

   ```
   ssh <member private ip>
   ```

5. Disable strictHostKeyChecking and enable agent forwarding on the leader node by adding the following to the ~/.ssh/config file on the leader node:

   ```
   Host *
       ForwardAgent yes
   Host *
       StrictHostKeyChecking no
   ```

6. On Amazon Linux and Amazon Linux 2 instances, run the following command on the leader node to provide correct permissions to the config file:

   ```
   chmod 600 ~/.ssh/config
   ```

Create Hosts File

On the leader node, create a hosts file to identify the nodes in the cluster. The hosts file must have an entry for each node in the cluster. Create a file ~/hosts and add each node using the private ip as follows:

```
localhost slots=8
<private ip of node 1> slots=8
<private ip of node 2> slots=8
```

2-Node NCCL Plugin Check

The nccl_message_transfer is a simple test to ensure that the NCCL OFI Plugin is working as expected. The test validates functionality of NCCL's connection establishment and data transfer APIs. Make sure you use the complete path to mpirun as shown in the example while running NCCL applications with EFA. Change the params `np` and `N` based on the number of instances and GPUs in your cluster. For more information, see the AWS OFI NCCL documentation.

P3dn.24xlarge check

The following nccl_message_transfer test is for CUDA 10.0. You can run the commands for CUDA 10.1 and 10.2 by replacing the CUDA version.

```
$/opt/amazon/openmpi/bin/mpirun \
         -n 2 -N 1 --hostfile hosts \
         -x LD_LIBRARY_PATH=/usr/local/cuda-10.0/efa/lib:/usr/local/cuda-10.0/lib:/usr/
local/cuda-10.0/lib64:/usr/local/cuda-10.0:$LD_LIBRARY_PATH \
```

```
        -x FI_PROVIDER="efa" --mca btl tcp,self --mca btl_tcp_if_exclude lo,docker0 --
bind-to none \
        ~/src/bin/efa-tests/efa-cuda-10.0/nccl_message_transfer
```

Your output should look like the following. You can check the output to see that EFA is being used as the OFI provider.

```
INFO: Function: ofi_init Line: 702: NET/OFI Forcing AWS OFI ndev 4
INFO: Function: ofi_init Line: 714: NET/OFI Selected Provider is efa
INFO: Function: main Line: 49: NET/OFI Process rank 1 started. NCCLNet device used on
 ip-172-31-15-30 is AWS Libfabric.
INFO: Function: main Line: 53: NET/OFI Received 4 network devices
INFO: Function: main Line: 57: NET/OFI Server: Listening on dev 0
INFO: Function: ofi_init Line: 702: NET/OFI Forcing AWS OFI ndev 4
INFO: Function: ofi_init Line: 714: NET/OFI Selected Provider is efa
INFO: Function: main Line: 49: NET/OFI Process rank 0 started. NCCLNet device used on
 ip-172-31-15-30 is AWS Libfabric.
INFO: Function: main Line: 53: NET/OFI Received 4 network devices
INFO: Function: main Line: 57: NET/OFI Server: Listening on dev 0
INFO: Function: main Line: 96: NET/OFI Send connection request to rank 0
INFO: Function: main Line: 69: NET/OFI Send connection request to rank 1
INFO: Function: main Line: 100: NET/OFI Server: Start accepting requests
INFO: Function: main Line: 73: NET/OFI Server: Start accepting requests
INFO: Function: main Line: 103: NET/OFI Successfully accepted connection from rank 0
INFO: Function: main Line: 107: NET/OFI Rank 1 posting 255 receive buffers
INFO: Function: main Line: 76: NET/OFI Successfully accepted connection from rank 1
INFO: Function: main Line: 80: NET/OFI Sent 255 requests to rank 1
INFO: Function: main Line: 131: NET/OFI Got completions for 255 requests for rank 0
INFO: Function: main Line: 131: NET/OFI Got completions for 255 requests for rank 1
```

Multi-node NCCL Performance Test on P3dn.24xlarge

To check NCCL Performance with EFA, run the standard NCCL Performance test that is available on the official NCCL-Tests Repo. The DLAMI comes with this test already built for both CUDA 10.0, 10.1, and 10.2. You can similarly run your own script with EFA.

When constructing your own script, refer to the following guidance:

- Provide the FI_PROVIDER="efa" flag to enable EFA use.
- Use the complete path to mpirun as shown in the example while running NCCL applications with EFA.
- Change the params np and N based on the number of instances and GPUs in your cluster.
- Add the NCCL_DEBUG=INFO flag and make sure that the logs indicate EFA usage as "Selected Provider is EFA".

Use the command `watch nvidia-smi` on any of the member nodes to monitor GPU usage. The following `watch nvidia-smi` commands are for CUDA 10.0 and depend on the Operating System of your instance. You can run the commands for CUDA 10.1 and 10.2 by replacing the CUDA version.

- Amazon Linux and Amazon Linux 2:

```
$ /opt/amazon/openmpi/bin/mpirun \
         -x FI_PROVIDER="efa" -n 16 -N 8 \
         -x NCCL_DEBUG=INFO \
         -x NCCL_TREE_THRESHOLD=0 \
         -x LD_LIBRARY_PATH=/usr/local/cuda-10.0/efa/lib:/usr/local/cuda-10.0/lib:/usr/
local/cuda-10.0/lib64:/usr/local/cuda-10.0:/opt/amazon/efa/lib64:/opt/amazon/openmpi/
lib64:$LD_LIBRARY_PATH \
         --hostfile hosts --mca btl tcp,self --mca btl_tcp_if_exclude lo,docker0 --bind-
to none \
         $HOME/src/bin/efa-tests/efa-cuda-10.0/all_reduce_perf -b 8 -e 1G -f 2 -g 1 -c 1
 -n 100
```

- Ubuntu 16.04 and Ubuntu 18.04:

```
$ /opt/amazon/openmpi/bin/mpirun \
         -x FI_PROVIDER="efa" -n 16 -N 8 \
         -x NCCL_DEBUG=INFO \
         -x NCCL_TREE_THRESHOLD=0 \
         -x LD_LIBRARY_PATH=/usr/local/cuda-10.0/efa/lib:/usr/local/cuda-10.0/lib:/usr/
local/cuda-10.0/lib64:/usr/local/cuda-10.0:/opt/amazon/efa/lib:/opt/amazon/openmpi/lib:
$LD_LIBRARY_PATH \
         --hostfile hosts --mca btl tcp,self --mca btl_tcp_if_exclude lo,docker0 --bind-
to none \
         $HOME/src/bin/efa-tests/efa-cuda-10.0/all_reduce_perf -b 8 -e 1G -f 2 -g 1 -c 1
 -n 100
```

Your output should look like the following.

```
# nThread 1 nGpus 1 minBytes 8 maxBytes 1073741824 step: 2(factor) warmup iters: 5 iters:
 100 validation: 1
#
# Using devices
#   Rank  0 Pid   3801 on ip-172-31-41-105 device  0 [0x00] Tesla V100-SXM2-32GB
#   Rank  1 Pid   3802 on ip-172-31-41-105 device  1 [0x00] Tesla V100-SXM2-32GB
#   Rank  2 Pid   3803 on ip-172-31-41-105 device  2 [0x00] Tesla V100-SXM2-32GB
#   Rank  3 Pid   3804 on ip-172-31-41-105 device  3 [0x00] Tesla V100-SXM2-32GB
#   Rank  4 Pid   3805 on ip-172-31-41-105 device  4 [0x00] Tesla V100-SXM2-32GB
#   Rank  5 Pid   3807 on ip-172-31-41-105 device  5 [0x00] Tesla V100-SXM2-32GB
#   Rank  6 Pid   3810 on ip-172-31-41-105 device  6 [0x00] Tesla V100-SXM2-32GB
#   Rank  7 Pid   3813 on ip-172-31-41-105 device  7 [0x00] Tesla V100-SXM2-32GB
#   Rank  8 Pid   4124 on ip-172-31-41-36 device  0 [0x00] Tesla V100-SXM2-32GB
#   Rank  9 Pid   4125 on ip-172-31-41-36 device  1 [0x00] Tesla V100-SXM2-32GB
#   Rank 10 Pid   4126 on ip-172-31-41-36 device  2 [0x00] Tesla V100-SXM2-32GB
#   Rank 11 Pid   4127 on ip-172-31-41-36 device  3 [0x00] Tesla V100-SXM2-32GB
#   Rank 12 Pid   4128 on ip-172-31-41-36 device  4 [0x00] Tesla V100-SXM2-32GB
#   Rank 13 Pid   4130 on ip-172-31-41-36 device  5 [0x00] Tesla V100-SXM2-32GB
#   Rank 14 Pid   4132 on ip-172-31-41-36 device  6 [0x00] Tesla V100-SXM2-32GB
#   Rank 15 Pid   4134 on ip-172-31-41-36 device  7 [0x00] Tesla V100-SXM2-32GB
ip-172-31-41-105:3801:3801 [0] NCCL INFO Bootstrap : Using [0]ens5:172.31.41.105<0>
ip-172-31-41-105:3801:3801 [0] NCCL INFO NET/OFI Forcing AWS OFI ndev 4
ip-172-31-41-105:3801:3801 [0] NCCL INFO NET/OFI Selected Provider is efa
NCCL version 2.4.8+cuda10.0
ip-172-31-41-105:3810:3810 [6] NCCL INFO Bootstrap : Using [0]ens5:172.31.41.105<0>
ip-172-31-41-105:3810:3810 [6] NCCL INFO NET/OFI Forcing AWS OFI ndev 4
ip-172-31-41-105:3810:3810 [6] NCCL INFO NET/OFI Selected Provider is efa
ip-172-31-41-105:3805:3805 [4] NCCL INFO Bootstrap : Using [0]ens5:172.31.41.105<0>
ip-172-31-41-105:3805:3805 [4] NCCL INFO NET/OFI Forcing AWS OFI ndev 4
ip-172-31-41-105:3805:3805 [4] NCCL INFO NET/OFI Selected Provider is efa
ip-172-31-41-105:3807:3807 [5] NCCL INFO Bootstrap : Using [0]ens5:172.31.41.105<0>
ip-172-31-41-105:3807:3807 [5] NCCL INFO NET/OFI Forcing AWS OFI ndev 4
ip-172-31-41-105:3807:3807 [5] NCCL INFO NET/OFI Selected Provider is efa
ip-172-31-41-105:3803:3803 [2] NCCL INFO Bootstrap : Using [0]ens5:172.31.41.105<0>
ip-172-31-41-105:3803:3803 [2] NCCL INFO NET/OFI Forcing AWS OFI ndev 4
ip-172-31-41-105:3803:3803 [2] NCCL INFO NET/OFI Selected Provider is efa
ip-172-31-41-105:3813:3813 [7] NCCL INFO Bootstrap : Using [0]ens5:172.31.41.105<0>
ip-172-31-41-105:3802:3802 [1] NCCL INFO Bootstrap : Using [0]ens5:172.31.41.105<0>
ip-172-31-41-105:3813:3813 [7] NCCL INFO NET/OFI Forcing AWS OFI ndev 4
ip-172-31-41-105:3813:3813 [7] NCCL INFO NET/OFI Selected Provider is efa
ip-172-31-41-105:3802:3802 [1] NCCL INFO NET/OFI Forcing AWS OFI ndev 4
ip-172-31-41-105:3802:3802 [1] NCCL INFO NET/OFI Selected Provider is efa
ip-172-31-41-105:3804:3804 [3] NCCL INFO Bootstrap : Using [0]ens5:172.31.41.105<0>
ip-172-31-41-105:3804:3804 [3] NCCL INFO NET/OFI Forcing AWS OFI ndev 4
```

```
ip-172-31-41-105:3804:3804 [3] NCCL INFO NET/OFI Selected Provider is efa
ip-172-31-41-105:3801:3862 [0] NCCL INFO Setting affinity for GPU 0 to
 ffffffff,ffffffff,ffffffff
ip-172-31-41-105:3801:3862 [0] NCCL INFO NCCL_TREE_THRESHOLD set by environment to 0.
ip-172-31-41-36:4128:4128 [4] NCCL INFO Bootstrap : Using [0]ens5:172.31.41.36<0>
ip-172-31-41-36:4128:4128 [4] NCCL INFO NET/OFI Forcing AWS OFI ndev 4
ip-172-31-41-36:4128:4128 [4] NCCL INFO NET/OFI Selected Provider is efa

-----------------------------some output truncated-----------------------------------

ip-172-31-41-105:3804:3869 [3] NCCL INFO comm 0x7f8c5c0025b0 rank 3 nranks 16 cudaDev 3
 nvmlDev 3 - Init COMPLETE
#
#                                                     out-of-place
 in-place
#       size         count    type   redop     time   algbw   busbw  error     time   algbw
   busbw  error
#        (B)    (elements)                     (us)  (GB/s)  (GB/s)            (us)  (GB/s)
  (GB/s)
ip-172-31-41-105:3801:3801 [0] NCCL INFO Launch mode Parallel
ip-172-31-41-36:4124:4191 [0] NCCL INFO comm 0x7f28400025b0 rank 8 nranks 16 cudaDev 0
 nvmlDev 0 - Init COMPLETE
ip-172-31-41-36:4126:4192 [2] NCCL INFO comm 0x7f62240025b0 rank 10 nranks 16 cudaDev 2
 nvmlDev 2 - Init COMPLETE
ip-172-31-41-105:3802:3867 [1] NCCL INFO comm 0x7f5ff00025b0 rank 1 nranks 16 cudaDev 1
 nvmlDev 1 - Init COMPLETE
ip-172-31-41-36:4132:4193 [6] NCCL INFO comm 0x7ffa0c0025b0 rank 14 nranks 16 cudaDev 6
 nvmlDev 6 - Init COMPLETE
ip-172-31-41-105:3803:3866 [2] NCCL INFO comm 0x7fe9600025b0 rank 2 nranks 16 cudaDev 2
 nvmlDev 2 - Init COMPLETE
ip-172-31-41-36:4127:4188 [3] NCCL INFO comm 0x7f6ad00025b0 rank 11 nranks 16 cudaDev 3
 nvmlDev 3 - Init COMPLETE
ip-172-31-41-105:3813:3868 [7] NCCL INFO comm 0x7f341c0025b0 rank 7 nranks 16 cudaDev 7
 nvmlDev 7 - Init COMPLETE
ip-172-31-41-105:3810:3864 [6] NCCL INFO comm 0x7f5f980025b0 rank 6 nranks 16 cudaDev 6
 nvmlDev 6 - Init COMPLETE
ip-172-31-41-36:4128:4187 [4] NCCL INFO comm 0x7f234c0025b0 rank 12 nranks 16 cudaDev 4
 nvmlDev 4 - Init COMPLETE
ip-172-31-41-36:4125:4194 [1] NCCL INFO comm 0x7f2ca00025b0 rank 9 nranks 16 cudaDev 1
 nvmlDev 1 - Init COMPLETE
ip-172-31-41-36:4134:4190 [7] NCCL INFO comm 0x7f0ca40025b0 rank 15 nranks 16 cudaDev 7
 nvmlDev 7 - Init COMPLETE
ip-172-31-41-105:3807:3865 [5] NCCL INFO comm 0x7f3b280025b0 rank 5 nranks 16 cudaDev 5
 nvmlDev 5 - Init COMPLETE
ip-172-31-41-36:4130:4189 [5] NCCL INFO comm 0x7f62080025b0 rank 13 nranks 16 cudaDev 5
 nvmlDev 5 - Init COMPLETE
ip-172-31-41-105:3805:3863 [4] NCCL INFO comm 0x7fec100025b0 rank 4 nranks 16 cudaDev 4
 nvmlDev 4 - Init COMPLETE
           8             2   float     sum    145.4    0.00    0.00  2e-07    152.8    0.00
    0.00  1e-07
          16             4   float     sum    137.3    0.00    0.00  1e-07    137.2    0.00
    0.00  1e-07
          32             8   float     sum    137.0    0.00    0.00  1e-07    137.4    0.00
    0.00  1e-07
          64            16   float     sum    137.7    0.00    0.00  1e-07    137.5    0.00
    0.00  1e-07
         128            32   float     sum    136.2    0.00    0.00  1e-07    135.3    0.00
    0.00  1e-07
         256            64   float     sum    136.4    0.00    0.00  1e-07    137.4    0.00
    0.00  1e-07
         512           128   float     sum    135.5    0.00    0.01  1e-07    151.0    0.00
    0.01  1e-07
        1024           256   float     sum    151.0    0.01    0.01  2e-07    137.7    0.01
    0.01  2e-07
        2048           512   float     sum    138.1    0.01    0.03  5e-07    138.1    0.01
    0.03  5e-07
```

```
        4096          1024     float     sum    140.5    0.03    0.05  5e-07    140.3    0.03
    0.05  5e-07
        8192          2048     float     sum    144.6    0.06    0.11  5e-07    144.7    0.06
    0.11  5e-07
       16384          4096     float     sum    149.4    0.11    0.21  5e-07    149.3    0.11
    0.21  5e-07
       32768          8192     float     sum    156.7    0.21    0.39  5e-07    183.9    0.18
    0.33  5e-07
       65536         16384     float     sum    167.7    0.39    0.73  5e-07    183.6    0.36
    0.67  5e-07
      131072         32768     float     sum    193.8    0.68    1.27  5e-07    193.0    0.68
    1.27  5e-07
      262144         65536     float     sum    243.9    1.07    2.02  5e-07    258.3    1.02
    1.90  5e-07
      524288        131072     float     sum    309.0    1.70    3.18  5e-07    309.0    1.70
    3.18  5e-07
     1048576        262144     float     sum    709.3    1.48    2.77  5e-07    693.2    1.51
    2.84  5e-07
     2097152        524288     float     sum   1116.4    1.88    3.52  5e-07   1105.7    1.90
    3.56  5e-07
     4194304       1048576     float     sum   2088.9    2.01    3.76  5e-07   2157.3    1.94
    3.65  5e-07
     8388608       2097152     float     sum   2869.7    2.92    5.48  5e-07   2847.2    2.95
    5.52  5e-07
    16777216       4194304     float     sum   4631.7    3.62    6.79  5e-07   4643.9    3.61
    6.77  5e-07
    33554432       8388608     float     sum   8769.2    3.83    7.17  5e-07   8743.5    3.84
    7.20  5e-07
    67108864      16777216     float     sum    16964    3.96    7.42  5e-07    16846    3.98
    7.47  5e-07
   134217728      33554432     float     sum    33403    4.02    7.53  5e-07    33058    4.06
    7.61  5e-07
   268435456      67108864     float     sum    59045    4.55    8.52  5e-07    58625    4.58
    8.59  5e-07
   536870912     134217728     float     sum   115842    4.63    8.69  5e-07   115590    4.64
    8.71  5e-07
  1073741824     268435456     float     sum   228178    4.71    8.82  5e-07   224997    4.77
    8.95  5e-07
# Out of bounds values : 0 OK
# Avg bus bandwidth    : 2.80613
#
```

P4d.24xlarge check

The following nccl_message_transfer test is for CUDA 11.0.

```
$/opt/amazon/openmpi/bin/mpirun \
         -n 2 -N 1 --hostfile hosts \
         -x LD_LIBRARY_PATH=/usr/local/cuda-11.0/efa/lib:/usr/local/cuda-11.0/lib:/usr/
local/cuda-11.0/lib64:/usr/local/cuda-11.0:$LD_LIBRARY_PATH \
         -x FI_EFA_USE_DEVICE_RDMA=1 -x NCCL_ALGO=ring -x --mca pml ^cm \
         -x FI_PROVIDER="efa" --mca btl tcp,self --mca btl_tcp_if_exclude lo,docker0 --
bind-to none \
         ~/src/bin/efa-tests/efa-cuda-11.0/nccl_message_transfer
```

Your output should look like the following. You can check the output to see that EFA is being used as the OFI provider.

```
INFO: Function: ofi_init Line: 1078: NET/OFI Running on P4d platform, Setting
 NCCL_TOPO_FILE environment variable to /usr/local/cuda-11.0/efa/share/aws-ofi-nccl/xml/
p4d-24xl-topo.xml
```

```
INFO: Function: ofi_init Line: 1078: NET/OFI Running on P4d platform, Setting
 NCCL_TOPO_FILE environment variable to /usr/local/cuda-11.0/efa/share/aws-ofi-nccl/xml/
p4d-24xl-topo.xml
INFO: Function: ofi_init Line: 1152: NET/OFI Selected Provider is efa
INFO: Function: main Line: 68: NET/OFI Process rank 1 started. NCCLNet device used on
 ip-172-31-79-191 is AWS Libfabric.
INFO: Function: main Line: 72: NET/OFI Received 4 network devices
INFO: Function: main Line: 107: NET/OFI Network supports communication using CUDA buffers.
 Dev: 3
INFO: Function: main Line: 113: NET/OFI Server: Listening on dev 3
INFO: Function: ofi_init Line: 1152: NET/OFI Selected Provider is efa
INFO: Function: main Line: 68: NET/OFI Process rank 0 started. NCCLNet device used on
 ip-172-31-70-99 is AWS Libfabric.
INFO: Function: main Line: 72: NET/OFI Received 4 network devices
INFO: Function: main Line: 107: NET/OFI Network supports communication using CUDA buffers.
 Dev: 3
INFO: Function: main Line: 113: NET/OFI Server: Listening on dev 3
INFO: Function: main Line: 126: NET/OFI Send connection request to rank 1
INFO: Function: main Line: 160: NET/OFI Send connection request to rank 0
INFO: Function: main Line: 164: NET/OFI Server: Start accepting requests
INFO: Function: main Line: 167: NET/OFI Successfully accepted connection from rank 0
INFO: Function: main Line: 171: NET/OFI Rank 1 posting 255 receive buffers
INFO: Function: main Line: 130: NET/OFI Server: Start accepting requests
INFO: Function: main Line: 133: NET/OFI Successfully accepted connection from rank 1
INFO: Function: main Line: 137: NET/OFI Sent 255 requests to rank 1
INFO: Function: main Line: 223: NET/OFI Got completions for 255 requests for rank 1
INFO: Function: main Line: 223: NET/OFI Got completions for 255 requests for rank 0
```

Multi-node NCCL Performance Test on P4d.24xlarge

To check NCCL Performance with EFA, run the standard NCCL Performance test that is available on the official NCCL-Tests Repo. The DLAMI comes with this test already built for CUDA 11.0. You can similarly run your own script with EFA.

When constructing your own script, refer to the following guidance:

- Provide the FI_PROVIDER="efa" flag to enable EFA use.
- Use the complete path to mpirun as shown in the example while running NCCL applications with EFA.
- Change the params np and N based on the number of instances and GPUs in your cluster.
- Add the NCCL_DEBUG=INFO flag and make sure that the logs indicate EFA usage as "Selected Provider is EFA".
- Add the `FI_EFA_USE_DEVICE_RDMA=1` flag to use EFA's RDMA functionality for one-sided and two-sided transfer.

Use the command `watch nvidia-smi` on any of the member nodes to monitor GPU usage. The following `watch nvidia-smi` commands are for CUDA 11.0 and depend on the Operating System of your instance.

- Amazon Linux 2:

```
$ /opt/amazon/openmpi/bin/mpirun \
         -x FI_PROVIDER="efa" -n 16 -N 8 \
         -x NCCL_DEBUG=INFO \
         -x FI_EFA_USE_DEVICE_RDMA=1 -x NCCL_ALGO=ring -x --mca pml ^cm \
         -x LD_LIBRARY_PATH=/usr/local/cuda-11.0/efa/lib:/usr/local/cuda-11.0/lib:/usr/
local/cuda-11.0/lib64:/usr/local/cuda-11.0:/opt/amazon/efa/lib64:/opt/amazon/openmpi/
lib64:$LD_LIBRARY_PATH \
         --hostfile hosts --mca btl tcp,self --mca btl_tcp_if_exclude lo,docker0 --bind-
to none \
         $HOME/src/bin/efa-tests/efa-cuda-11.0/all_reduce_perf -b 8 -e 1G -f 2 -g 1 -c 1
 -n 100
```

- Ubuntu 16.04 and Ubuntu 18.04:

```
$ /opt/amazon/openmpi/bin/mpirun \
         -x FI_PROVIDER="efa" -n 16 -N 8 \
         -x NCCL_DEBUG=INFO \
         -x FI_EFA_USE_DEVICE_RDMA=1 -x NCCL_ALGO=ring -x --mca pml ^cm \
         -x LD_LIBRARY_PATH=/usr/local/cuda-11.0/efa/lib:/usr/local/cuda-11.0/lib:/usr/
local/cuda-11.0/lib64:/usr/local/cuda-11.0:/opt/amazon/efa/lib:/opt/amazon/openmpi/lib:
$LD_LIBRARY_PATH \
         --hostfile hosts --mca btl tcp,self --mca btl_tcp_if_exclude lo,docker0 --bind-
to none \
         $HOME/src/bin/efa-tests/efa-cuda-11.0/all_reduce_perf -b 8 -e 1G -f 2 -g 1 -c 1
 -n 100
```

Your output should look like the following.

```
# nThread 1 nGpus 1 minBytes 8 maxBytes 1073741824 step: 2(factor) warmup iters: 5 iters:
 100 validation: 1
    #
    # Using devices
    #   Rank  0 Pid  18546 on ip-172-31-70-88 device  0 [0x10] A100-SXM4-40GB
    #   Rank  1 Pid  18547 on ip-172-31-70-88 device  1 [0x10] A100-SXM4-40GB
    #   Rank  2 Pid  18548 on ip-172-31-70-88 device  2 [0x20] A100-SXM4-40GB
    #   Rank  3 Pid  18549 on ip-172-31-70-88 device  3 [0x20] A100-SXM4-40GB
    #   Rank  4 Pid  18550 on ip-172-31-70-88 device  4 [0x90] A100-SXM4-40GB
    #   Rank  5 Pid  18551 on ip-172-31-70-88 device  5 [0x90] A100-SXM4-40GB
    #   Rank  6 Pid  18552 on ip-172-31-70-88 device  6 [0xa0] A100-SXM4-40GB
    #   Rank  7 Pid  18556 on ip-172-31-70-88 device  7 [0xa0] A100-SXM4-40GB
    #   Rank  8 Pid  19502 on ip-172-31-78-249 device  0 [0x10] A100-SXM4-40GB
    #   Rank  9 Pid  19503 on ip-172-31-78-249 device  1 [0x10] A100-SXM4-40GB
    #   Rank 10 Pid  19504 on ip-172-31-78-249 device  2 [0x20] A100-SXM4-40GB
    #   Rank 11 Pid  19505 on ip-172-31-78-249 device  3 [0x20] A100-SXM4-40GB
    #   Rank 12 Pid  19506 on ip-172-31-78-249 device  4 [0x90] A100-SXM4-40GB
    #   Rank 13 Pid  19507 on ip-172-31-78-249 device  5 [0x90] A100-SXM4-40GB
    #   Rank 14 Pid  19508 on ip-172-31-78-249 device  6 [0xa0] A100-SXM4-40GB
    #   Rank 15 Pid  19509 on ip-172-31-78-249 device  7 [0xa0] A100-SXM4-40GB
    ip-172-31-70-88:18546:18546 [0] NCCL INFO Bootstrap : Using [0]eth0:172.31.71.137<0>
 [1]eth1:172.31.70.88<0> [2]eth2:172.31.78.243<0> [3]eth3:172.31.77.226<0>
    ip-172-31-70-88:18546:18546 [0] NCCL INFO NET/OFI Running on P4d platform, Setting
 NCCL_TOPO_FILE environment variable to /usr/local/cuda-11.0/efa/share/aws-ofi-nccl/xml/
p4d-24xl-topo.xml
    ip-172-31-70-88:18546:18546 [0] NCCL INFO NET/OFI Selected Provider is efa
    ip-172-31-70-88:18546:18546 [0] NCCL INFO NET/Plugin: Failed to find
 ncclCollNetPlugin_v3 symbol.
    ip-172-31-70-88:18546:18546 [0] NCCL INFO Using network AWS Libfabric
    NCCL version 2.7.8+cuda11.0
    ip-172-31-70-88:18552:18552 [6] NCCL INFO Bootstrap : Using [0]eth0:172.31.71.137<0>
 [1]eth1:172.31.70.88<0> [2]eth2:172.31.78.243<0> [3]eth3:172.31.77.226<0>
    ip-172-31-70-88:18552:18552 [6] NCCL INFO NET/OFI Running on P4d platform, Setting
 NCCL_TOPO_FILE environment variable to /usr/local/cuda-11.0/efa/share/aws-ofi-nccl/xml/
p4d-24xl-topo.xml
    ip-172-31-70-88:18551:18551 [5] NCCL INFO Bootstrap : Using [0]eth0:172.31.71.137<0>
 [1]eth1:172.31.70.88<0> [2]eth2:172.31.78.243<0> [3]eth3:172.31.77.226<0>
    ip-172-31-70-88:18556:18556 [7] NCCL INFO Bootstrap : Using [0]eth0:172.31.71.137<0>
 [1]eth1:172.31.70.88<0> [2]eth2:172.31.78.243<0> [3]eth3:172.31.77.226<0>
    ip-172-31-70-88:18548:18548 [2] NCCL INFO Bootstrap : Using [0]eth0:172.31.71.137<0>
 [1]eth1:172.31.70.88<0> [2]eth2:172.31.78.243<0> [3]eth3:172.31.77.226<0>
    ip-172-31-70-88:18550:18550 [4] NCCL INFO Bootstrap : Using [0]eth0:172.31.71.137<0>
 [1]eth1:172.31.70.88<0> [2]eth2:172.31.78.243<0> [3]eth3:172.31.77.226<0>
    ip-172-31-70-88:18552:18552 [6] NCCL INFO NET/OFI Selected Provider is efa
    ip-172-31-70-88:18552:18552 [6] NCCL INFO NET/Plugin: Failed to find
 ncclCollNetPlugin_v3 symbol.
    ip-172-31-70-88:18552:18552 [6] NCCL INFO Using network AWS Libfabric
    ip-172-31-70-88:18556:18556 [7] NCCL INFO NET/OFI Running on P4d platform, Setting
 NCCL_TOPO_FILE environment variable to /usr/local/cuda-11.0/efa/share/aws-ofi-nccl/xml/
p4d-24xl-topo.xml
    ip-172-31-70-88:18548:18548 [2] NCCL INFO NET/OFI Running on P4d platform, Setting
 NCCL_TOPO_FILE environment variable to /usr/local/cuda-11.0/efa/share/aws-ofi-nccl/xml/
p4d-24xl-topo.xml
    ip-172-31-70-88:18550:18550 [4] NCCL INFO NET/OFI Running on P4d platform, Setting
 NCCL_TOPO_FILE environment variable to /usr/local/cuda-11.0/efa/share/aws-ofi-nccl/xml/
p4d-24xl-topo.xml
    ip-172-31-70-88:18551:18551 [5] NCCL INFO NET/OFI Running on P4d platform, Setting
 NCCL_TOPO_FILE environment variable to /usr/local/cuda-11.0/efa/share/aws-ofi-nccl/xml/
p4d-24xl-topo.xml
    ip-172-31-70-88:18547:18547 [1] NCCL INFO Bootstrap : Using [0]eth0:172.31.71.137<0>
 [1]eth1:172.31.70.88<0> [2]eth2:172.31.78.243<0> [3]eth3:172.31.77.226<0>
    ip-172-31-70-88:18549:18549 [3] NCCL INFO Bootstrap : Using [0]eth0:172.31.71.137<0>
 [1]eth1:172.31.70.88<0> [2]eth2:172.31.78.243<0> [3]eth3:172.31.77.226<0>
    ip-172-31-70-88:18547:18547 [1] NCCL INFO NET/OFI Running on P4d platform, Setting
 NCCL_TOPO_FILE environment variable to /usr/local/cuda-11.0/efa/share/aws-ofi-nccl/xml/
p4d-24xl-topo.xml
    ip-172-31-70-88:18549:18549 [3] NCCL INFO NET/OFI Running on P4d platform, Setting
 NCCL_TOPO_FILE environment variable to /usr/local/cuda-11.0/efa/share/aws-ofi-nccl/xml/
p4d-24xl-topo.xml
    ip-172-31-70-88:18547:18547 [1] NCCL INFO NET/OFI Selected Provider is efa
    ip-172-31-70-88:18547:18547 [1] NCCL INFO NET/Plugin: Failed to find
 ncclCollNetPlugin_v3 symbol.
    ip-172-31-70-88:18547:18547 [1] NCCL INFO Using network AWS Libfabric
    ip-172-31-70-88:18549:18549 [3] NCCL INFO NET/OFI Selected Provider is efa
    ip-172-31-70-88:18549:18549 [3] NCCL INFO NET/Plugin: Failed to find
 ncclCollNetPlugin_v3 symbol.
    ip-172-31-70-88:18549:18549 [3] NCCL INFO Using network AWS Libfabric
    ip-172-31-70-88:18551:18551 [5] NCCL INFO NET/OFI Selected Provider is efa
    ip-172-31-70-88:18551:18551 [5] NCCL INFO NET/Plugin: Failed to find
 ncclCollNetPlugin_v3 symbol.
    ip-172-31-70-88:18551:18551 [5] NCCL INFO Using network AWS Libfabric
    ip-172-31-70-88:18556:18556 [7] NCCL INFO NET/OFI Selected Provider is efa
    ip-172-31-70-88:18556:18556 [7] NCCL INFO NET/Plugin: Failed to find
 ncclCollNetPlugin_v3 symbol.
    ip-172-31-70-88:18556:18556 [7] NCCL INFO Using network AWS Libfabric
    ip-172-31-70-88:18548:18548 [2] NCCL INFO NET/OFI Selected Provider is efa
    ip-172-31-70-88:18548:18548 [2] NCCL INFO NET/Plugin: Failed to find
 ncclCollNetPlugin_v3 symbol.
    ip-172-31-70-88:18548:18548 [2] NCCL INFO Using network AWS Libfabric
    ip-172-31-70-88:18550:18550 [4] NCCL INFO NET/OFI Selected Provider is efa
    ip-172-31-70-88:18550:18550 [4] NCCL INFO NET/Plugin: Failed to find
 ncclCollNetPlugin_v3 symbol.
    ip-172-31-70-88:18550:18550 [4] NCCL INFO Using network AWS Libfabric
-----------------------------some output truncated-----------------------------------
    #
    #                                                  out-of-place
  in-place
    #       size         count    type   redop     time   algbw   busbw  error     time
 algbw   busbw  error
    #        (B)    (elements)                     (us)  (GB/s)  (GB/s)           (us)
 (GB/s)  (GB/s)
    ip-172-31-70-88:18546:18546 [0] NCCL INFO Launch mode Parallel
             8             2   float     sum    158.9    0.00    0.00  2e-07    158.1
 0.00    0.00  1e-07
            16             4   float     sum    158.3    0.00    0.00  1e-07    159.3
 0.00    0.00  1e-07
            32             8   float     sum    157.8    0.00    0.00  1e-07    158.1
 0.00    0.00  1e-07
            64            16   float     sum    158.7    0.00    0.00  1e-07    158.4
 0.00    0.00  6e-08
```

```
            128            32    float     sum    160.2    0.00    0.00  6e-08    158.8
0.00    0.00  6e-08
            256            64    float     sum    159.8    0.00    0.00  6e-08    159.8
0.00    0.00  6e-08
            512           128    float     sum    161.7    0.00    0.01  6e-08    161.7
0.00    0.01  6e-08
           1024           256    float     sum    177.8    0.01    0.01  5e-07    177.4
0.01    0.01  5e-07
           2048           512    float     sum    198.1    0.01    0.02  5e-07    198.1
0.01    0.02  5e-07
           4096          1024    float     sum    226.2    0.02    0.03  5e-07    225.8
0.02    0.03  5e-07
           8192          2048    float     sum    249.3    0.03    0.06  5e-07    249.4
0.03    0.06  5e-07
          16384          4096    float     sum    250.4    0.07    0.12  5e-07    251.0
0.07    0.12  5e-07
          32768          8192    float     sum    256.7    0.13    0.24  5e-07    257.2
0.13    0.24  5e-07
          65536         16384    float     sum    269.8    0.24    0.46  5e-07    271.2
0.24    0.45  5e-07
         131072         32768    float     sum    288.3    0.45    0.85  5e-07    286.8
0.46    0.86  5e-07
         262144         65536    float     sum    296.1    0.89    1.66  5e-07    295.6
0.89    1.66  5e-07
         524288        131072    float     sum    376.7    1.39    2.61  5e-07    382.0
1.37    2.57  5e-07
        1048576        262144    float     sum    448.6    2.34    4.38  5e-07    451.1
2.32    4.36  5e-07
        2097152        524288    float     sum    620.2    3.38    6.34  5e-07    615.9
3.41    6.38  5e-07
        4194304       1048576    float     sum    768.2    5.46   10.24  5e-07    759.8
5.52   10.35  5e-07
        8388608       2097152    float     sum   1228.5    6.83   12.80  5e-07   1223.3
6.86   12.86  5e-07
       16777216       4194304    float     sum   2002.7    8.38   15.71  5e-07   2004.5
8.37   15.69  5e-07
       33554432       8388608    float     sum   2988.8   11.23   21.05  5e-07   3012.0
11.14   20.89  5e-07
       67108864      16777216    float     sum   8072.1    8.31   15.59  5e-07   8102.4
8.28   15.53  5e-07
      134217728      33554432    float     sum    11431   11.74   22.01  5e-07    11474
11.70   21.93  5e-07
      268435456      67108864    float     sum    17603   15.25   28.59  5e-07    17641
15.22   28.53  5e-07
      536870912     134217728    float     sum    35110   15.29   28.67  5e-07    35102
15.29   28.68  5e-07
     1073741824     268435456    float     sum    70231   15.29   28.67  5e-07    70110
15.32   28.72  5e-07
   # Out of bounds values : 0 OK
   # Avg bus bandwidth    : 7.14456
```

GPU Monitoring and Optimization

The following section will guide you through GPU optimization and monitoring options. This section is organized like a typical workflow with monitoring overseeing preprocessing and training.

Monitoring

Your DLAMI comes preinstalled with several GPU monitoring tools. This guide also mentions tools that are available to download and install.

- Monitor GPUs with CloudWatch (p. 75) - a preinstalled utility that reports GPU usage statistics to Amazon CloudWatch.
- nvidia-smi CLI - a utility to monitor overall GPU compute and memory utilization. This is preinstalled on your AWS Deep Learning AMI (DLAMI).
- NVML C library - a C-based API to directly access GPU monitoring and management functions. This used by the nvidia-smi CLI under the hood and is preinstalled on your DLAMI. It also has Python and Perl bindings to facilitate development in those languages. The gpumon.py utility preinstalled on your DLAMI uses the pynvml package from nvidia-ml-py.
- NVIDIA DCGM - A cluster management tool. Visit the developer page to learn how to install and configure this tool.

Tip
Check out NVIDIA's developer blog for the latest info on using the CUDA tools installed your DLAMI:

- Monitoring TensorCore utilization using Nsight IDE and nvprof.

Monitor GPUs with CloudWatch

When you use your DLAMI with a GPU you might find that you are looking for ways to track its usage during training or inference. This can be useful for optimizing your data pipeline, and tuning your deep learning network.

There are two ways to configure GPU metrics with CloudWatch:

- Configure metrics with the AWS CloudWatch agent (Recommended) (p. 75)
- Configure metrics with the preinstalled `gpumon.py` script (p. 78)

Configure metrics with the AWS CloudWatch agent (Recommended)

Integrate your DLAMI with the unified CloudWatch agent to configure GPU metrics and monitor the utilization of GPU coprocesses in Amazon EC2 accelerated instances.

There are four ways to configure GPU metrics with your DLAMI:

- Configure minimal GPU metrics (p. 76)
- Configure partial GPU metrics (p. 76)
- Configure all available GPU metrics (p. 76)
- Configure custom GPU metrics (p. 77)

For information on updates and security patches, see Security patching for the AWS CloudWatch agent (p. 77)

Prerequisites

To get started, you must configure Amazon EC2 instance IAM permissions that allow your instance to push metrics to CloudWatch. For detailed steps, see Create IAM roles and users for use with the CloudWatch agent.

Configure minimal GPU metrics

Configure minimal GPU metrics using the `dlami-cloudwatch-agent@minimal` `systemd` service. This service configures the following metrics:

- `utilization_gpu`
- `utilization_memory`

You can find the `systemd` service for minimal preconfigured GPU metrics in the following location:

```
/opt/aws/amazon-cloudwatch-agent/etc/dlami-amazon-cloudwatch-agent-minimal.json
```

Enable and start the `systemd` service with the following commands:

```
sudo systemctl enable dlami-cloudwatch-agent@minimal
sudo systemctl start dlami-cloudwatch-agent@minimal
```

Configure partial GPU metrics

Configure partial GPU metrics using the `dlami-cloudwatch-agent@partial` `systemd` service. This service configures the following metrics:

- `utilization_gpu`
- `utilization_memory`
- `memory_total`
- `memory_used`
- `memory_free`

You can find the `systemd` service for partial preconfigured GPU metrics in the following location:

```
/opt/aws/amazon-cloudwatch-agent/etc/dlami-amazon-cloudwatch-agent-partial.json
```

Enable and start the `systemd` service with the following commands:

```
sudo systemctl enable dlami-cloudwatch-agent@partial
sudo systemctl start dlami-cloudwatch-agent@partial
```

Configure all available GPU metrics

Configure all available GPU metrics using the `dlami-cloudwatch-agent@all` `systemd` service. This service configures the following metrics:

- `utilization_gpu`
- `utilization_memory`
- `memory_total`
- `memory_used`
- `memory_free`
- `temperature_gpu`
- `power_draw`
- `fan_speed`
- `pcie_link_gen_current`

- `pcie_link_width_current`
- `encoder_stats_session_count`
- `encoder_stats_average_fps`
- `encoder_stats_average_latency`
- `clocks_current_graphics`
- `clocks_current_sm`
- `clocks_current_memory`
- `clocks_current_video`

You can find the `systemd` service for all available preconfigured GPU metrics in the following location:

```
/opt/aws/amazon-cloudwatch-agent/etc/dlami-amazon-cloudwatch-agent-all.json
```

Enable and start the `systemd` service with the following commands:

```
sudo systemctl enable dlami-cloudwatch-agent@all
sudo systemctl start dlami-cloudwatch-agent@all
```

Configure custom GPU metrics

If the preconfigured metrics do not meet your requirements, you can create a custom CloudWatch agent configuration file.

Create a custom configuration file

To create a custom configuration file, refer to the detailed steps in Manually create or edit the CloudWatch agent configuration file.

For this example, assume that the schema definition is located at `/opt/aws/amazon-cloudwatch-agent/etc/amazon-cloudwatch-agent.json`.

Configure metrics with your custom file

Run the following command to configure the CloudWatch agent according to your custom file:

```
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl \
-a fetch-config -m ec2 -s -c \
file:/opt/aws/amazon-cloudwatch-agent/etc/amazon-cloudwatch-agent.json
```

Security patching for the AWS CloudWatch agent

Newly released DLAMIs are configured with the latest available AWS CloudWatch agent security patches. Refer to the following sections to update your current DLAMI with the latest security patches depending on your operating system of choice.

Amazon Linux 2

Use `yum` to get the latest AWS CloudWatch agent security patches for an Amazon Linux 2 DLAMI.

```
 sudo yum update
```

Ubuntu

To get the latest AWS CloudWatch security patches for a DLAMI with Ubuntu, it is necessary to reinstall the AWS CloudWatch agent using an Amazon S3 download link.

```
wget https://s3.region.amazonaws.com/amazoncloudwatch-agent-region/ubuntu/arm64/latest/
amazon-cloudwatch-agent.deb
```

For more information on installing the AWS CloudWatch agent using Amazon S3 download links, see Installing and running the CloudWatch agent on your servers.

Configure metrics with the preinstalled `gpumon.py` script

A utility called gpumon.py is preinstalled on your DLAMI. It integrates with CloudWatch and supports monitoring of per-GPU usage: GPU memory, GPU temperature, and GPU Power. The script periodically sends the monitored data to CloudWatch. You can configure the level of granularity for data being sent to CloudWatch by changing a few settings in the script. Before starting the script, however, you will need to setup CloudWatch to receive the metrics.

How to setup and run GPU monitoring with CloudWatch

1. Create an IAM user, or modify an existing one to have a policy for publishing the metric to CloudWatch. If you create a new user please take note of the credentials as you will need these in the next step.

   The IAM policy to search for is “cloudwatch:PutMetricData”. The policy that is added is as follows:

   ```
   {
      "Version": "2012-10-17",
      "Statement": [
           {
               "Action": [
                   "cloudwatch:PutMetricData"
                ],
                "Effect": "Allow",
                "Resource": "*"
           }
      ]
   }
   ```

   Tip
   For more information on creating an IAM user and adding policies for CloudWatch, refer to the CloudWatch documentation.

2. On your DLAMI, run AWS configure and specify the IAM user credentials.

   ```
   $ aws configure
   ```

3. You might need to make some modifications to the gpumon utility before you run it. You can find the gpumon utility and README in the location defined in the following code block. For more information on the `gpumon.py` script, see the Amazon S3 location of the script.

   ```
   Folder: ~/tools/GPUCloudWatchMonitor
   Files:  ~/tools/GPUCloudWatchMonitor/gpumon.py
          ~/tools/GPUCloudWatchMonitor/README
   ```

   Options:

   - Change the region in gpumon.py if your instance is NOT in us-east-1.
   - Change other parameters such as the CloudWatch `namespace` or the reporting period with `store_reso`.

4. Currently the script only supports Python 3. Activate your preferred framework’s Python 3 environment or activate the DLAMI general Python 3 environment.

```
$ source activate python3
```

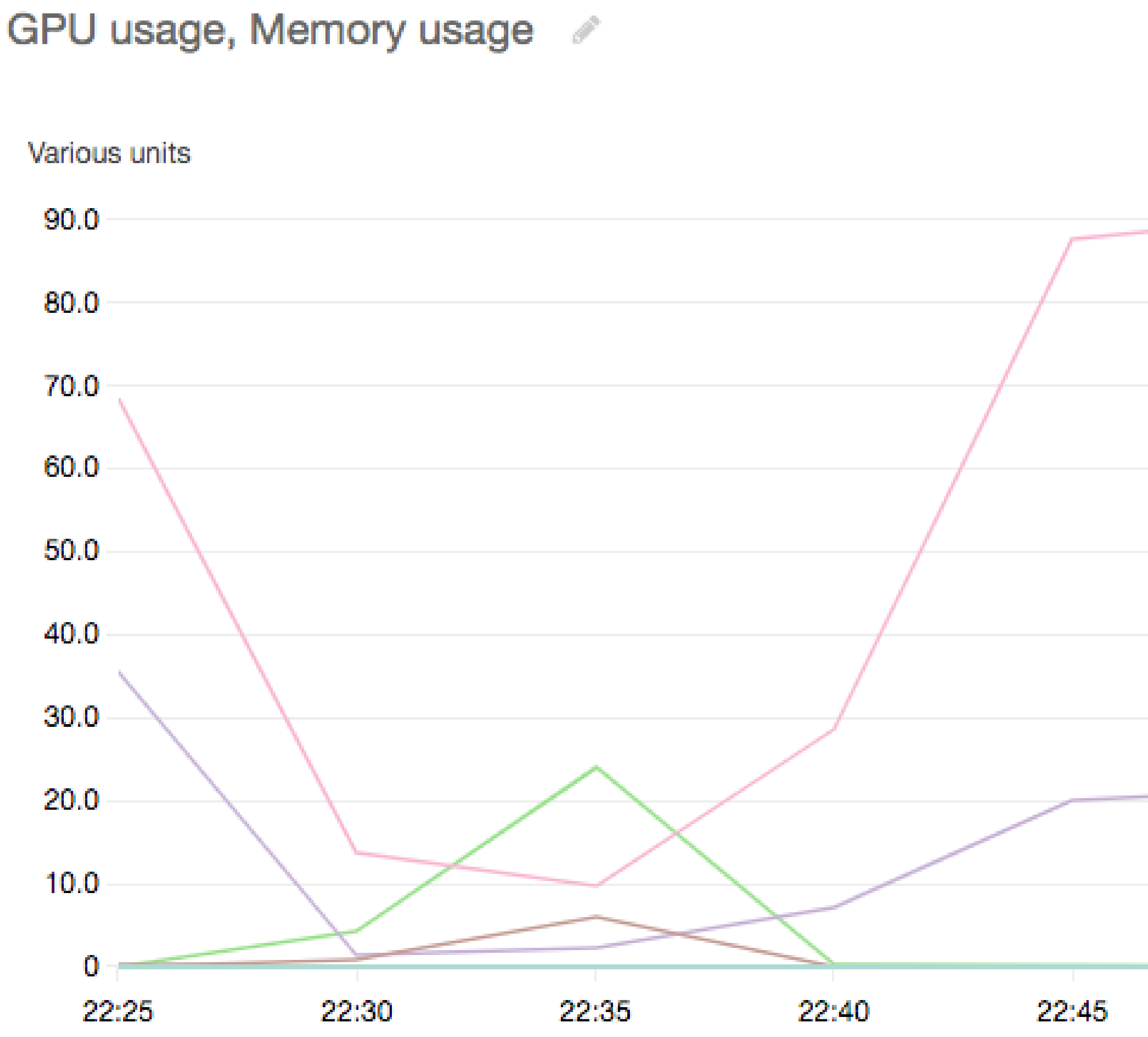
5. Run the gpumon utility in background.

```
(python3)$ python gpumon.py &
```

6. Open your browser to the https://console.aws.amazon.com/cloudwatch/ then select metric. It will have a namespace 'DeepLearningTrain'.

   Tip
   You can change the namespace by modifying gpumon.py. You can also modify the reporting interval by adjusting `store_reso`.

The following is an example CloudWatch chart reporting on a run of gpumon.py monitoring a training job on p2.8xlarge instance.



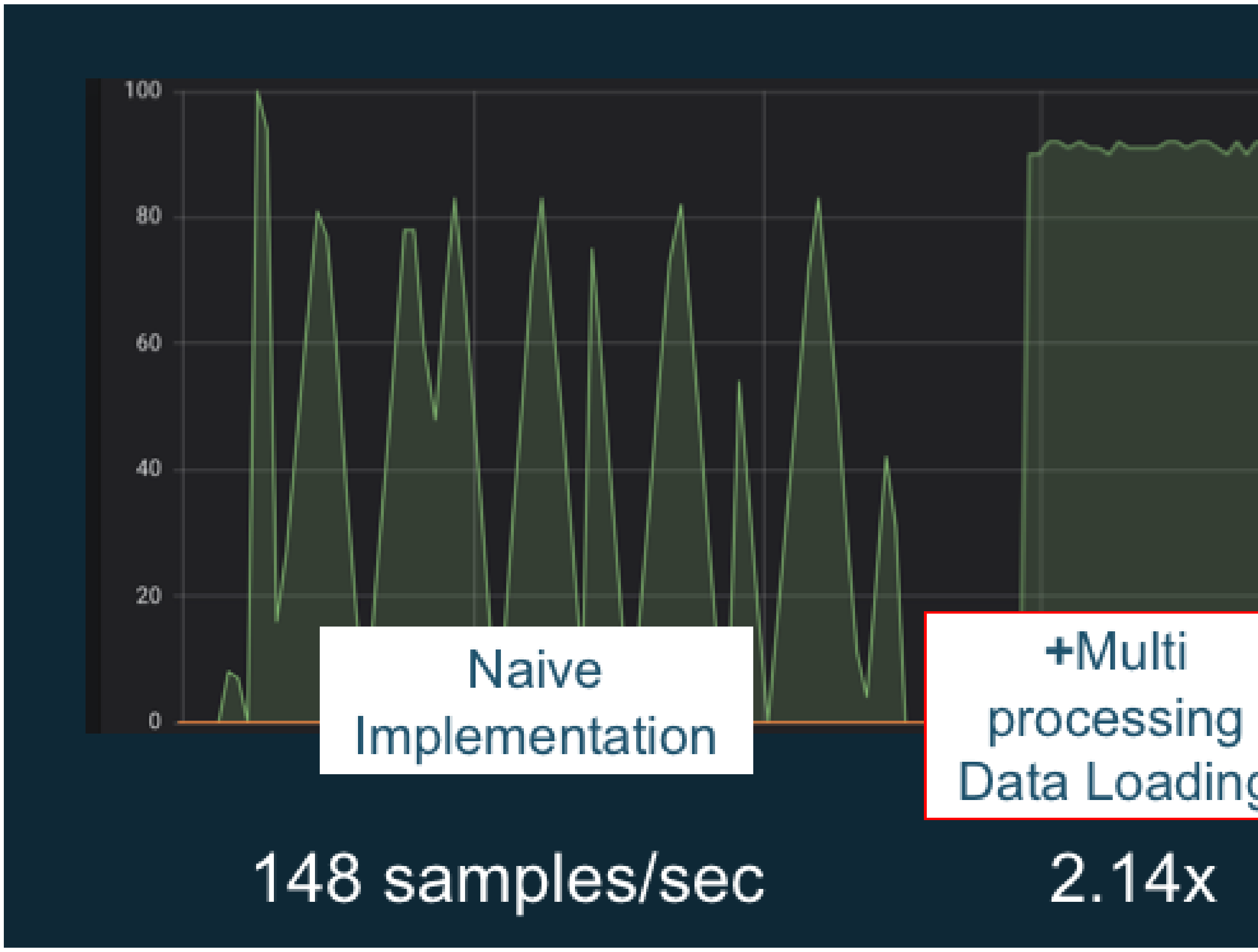
You might be interested in these other topics on GPU monitoring and optimization:

- Monitoring (p. 75)
  - Monitor GPUs with CloudWatch (p. 75)
- Optimization (p. 80)
  - Preprocessing (p. 81)
  - Training (p. 81)

Optimization

To make the most of your GPUs, you can optimize your data pipeline and tune your deep learning network. As the following chart describes, a naive or basic implementation of a neural network might use the GPU inconsistently and not to its fullest potential. When you optimize your preprocessing and data loading, you can reduce the bottleneck from your CPU to your GPU. You can adjust the neural network itself, by using hybridization (when supported by the framework), adjusting batch size, and synchronizing calls. You can also use multiple-precision (float16 or int8) training in most frameworks, which can have a dramatic effect on improving throughput.

The following chart shows the cumulative performance gains when applying different optimizations. Your results will depend on the data you are processing and the network you are optimizing.



Example GPU performance optimizations. Chart source: Performance Tricks with MXNet Gluon

The following guides introduce options that will work with your DLAMI and help you boost GPU performance.

Topics

Preprocessing

Data preprocessing through transformations or augmentations can often be a CPU-bound process, and this can be the bottleneck in your overall pipeline. Frameworks have built-in operators for image processing, but DALI (Data Augmentation Library) demonstrates improved performance over frameworks' built-in options.

- NVIDIA Data Augmentation Library (DALI): DALI offloads data augmentation to the GPU. It is not preinstalled on the DLAMI, but you can access it by installing it or loading a supported framework container on your DLAMI or other Amazon Elastic Compute Cloud instance. Refer to the DALI project page on the NVIDIA website for details. For an example use-case and to download code samples, see the SageMaker Preprocessing Training Performance sample.
- nvJPEG: a GPU-accelerated JPEG decoder library for C programmers. It supports decoding single images or batches as well as subsequent transformation operations that are common in deep learning. nvJPEG comes built-in with DALI, or you can download from the NVIDIA website's nvjpeg page and use it separately.

You might be interested in these other topics on GPU monitoring and optimization:

- Monitoring (p. 75)
  - Monitor GPUs with CloudWatch (p. 75)
- Optimization (p. 80)
  - Preprocessing (p. 81)
  - Training (p. 81)

Training

With mixed-precision training you can deploy larger networks with the same amount of memory, or reduce memory usage compared to your single or double precision network, and you will see compute performance increases. You also get the benefit of smaller and faster data transfers, an important factor in multiple node distributed training. To take advantage of mixed-precision training you need to adjust data casting and loss scaling. The following are guides describing how to do this for the frameworks that support mixed-precision.

- NVIDIA Deep Learning SDK - docs on the NVIDIA website describing mixed-precision implementation for MXNet, PyTorch, and TensorFlow.

Tip
Be sure to check the website for your framework of choice, and search for "mixed precision" or "fp16" for the latest optimization techniques. Here are some mixed-precision guides you might find helpful:

- Mixed-precision training with TensorFlow (video) - on the NVIDIA blog site.
- Mixed-precision training using float16 with MXNet - an FAQ article on the MXNet website.
- NVIDIA Apex: a tool for easy mixed-precision training with PyTorch - a blog article on the NVIDIA website.

You might be interested in these other topics on GPU monitoring and optimization:

- Monitoring (p. 75)
  - Monitor GPUs with CloudWatch (p. 75)
- Optimization (p. 80)
  - Preprocessing (p. 81)
  - Training (p. 81)

The AWS Inferentia Chip With DLAMI

AWS Inferentia is a custom machine learning chip designed by AWS that you can use for high-performance inference predictions. In order to use the chip, set up an Amazon Elastic Compute Cloud instance and use the AWS Neuron software development kit (SDK) to invoke the Inferentia chip. To provide customers with the best Inferentia experience, Neuron has been built into the AWS Deep Learning AMI (DLAMI).

The following topics show you how to get started using Inferentia with the DLAMI.

Contents

Launching a DLAMI Instance with AWS Neuron

The latest DLAMI is ready to use with AWS Inferentia and comes with the AWS Neuron API package. To launch a DLAMI instance, see Launching and Configuring a DLAMI. After you have a DLAMI, use the steps here to ensure that your AWS Inferentia chip and AWS Neuron resources are active.

Contents

Verify Your Instance

Before using your instance, verify that it's properly setup and configured with Neuron.

Identifying AWS Inferentia Devices

To identify the number of Inferentia devices on your instance, use the following command:

```
neuron-ls
```

If your instance has Inferentia devices attached to it, your output will look similar to the following:

```
+--------+--------+--------+-----------+--------------+
| NEURON | NEURON | NEURON | CONNECTED |     PCI      |
| DEVICE | CORES  | MEMORY |  DEVICES  |     BDF      |
+--------+--------+--------+-----------+--------------+
| 0      | 4      | 8 GB   | 1         | 0000:00:1c.0 |
| 1      | 4      | 8 GB   | 2, 0      | 0000:00:1d.0 |
| 2       | 4       | 8 GB    | 3, 1       | 0000:00:1e.0 |
| 3       | 4       | 8 GB    | 2          | 0000:00:1f.0 |
+---------+---------+---------+------------+--------------+
```

The supplied output is taken from an Inf1.6xlarge instance and includes the following columns:

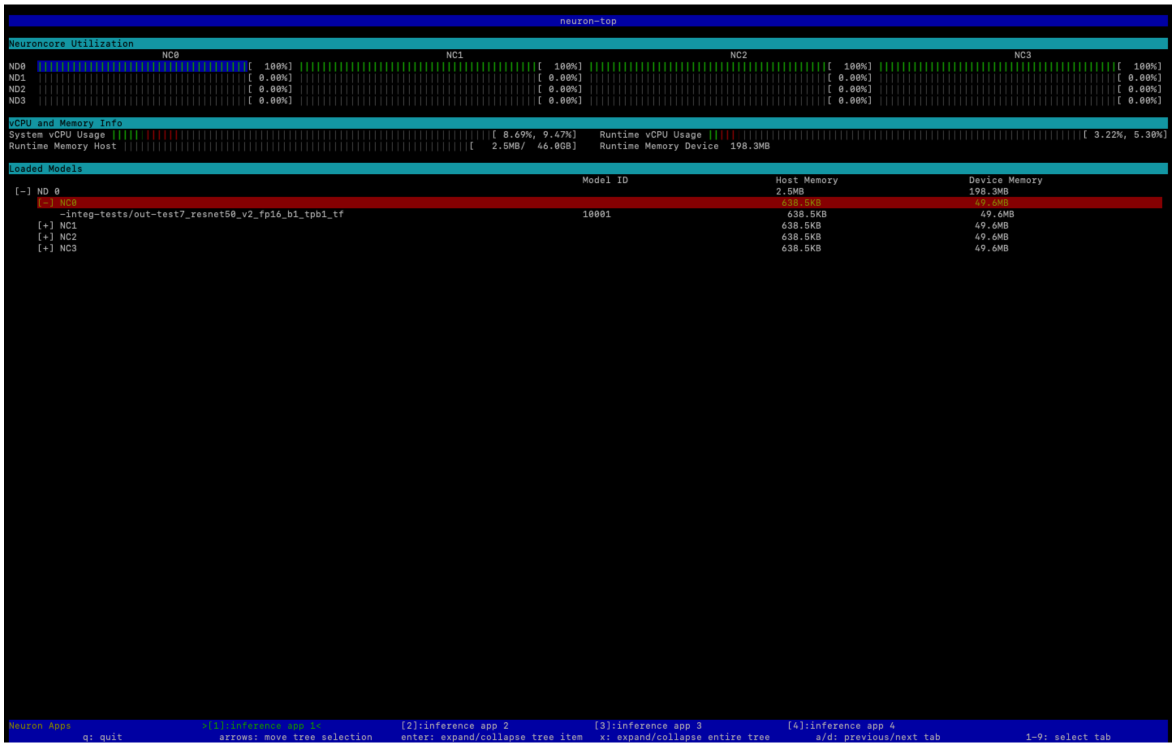
- NEURON DEVICE: The logical ID assigned to the NeuronDevice. This ID is used when configuring multiple runtimes to use different NeuronDevices.
- NEURON CORES: The number of NeuronCores present in the NeuronDevice.
- NEURON MEMORY: The amount of DRAM memory in the NeuronDevice.
- CONNECTED DEVICES: Other NeuronDevices connected to the NeuronDevice.
- PCI BDF: The PCI Bus Device Function (BDF) ID of the NeuronDevice.

View Resource Usage

View useful information about NeuronCore and vCPU utilization, memory usage, loaded models, and Neuron applications with the `neuron-top` command. Launching `neuron-top` with no arguments will show data for all machine learning applications that utilize NeuronCores.

```
neuron-top
```

When an application is using four NeuronCores, the output should look similar to the following image:



```
                                              neuron-top

Neuroncore Utilization
                  NC0                     NC1                     NC2                     NC3
ND0  [  100%]                [  100%]                [  100%]                [  100%]
ND1  [ 0.00%]                [ 0.00%]                [ 0.00%]                [ 0.00%]
ND2  [ 0.00%]                [ 0.00%]                [ 0.00%]                [ 0.00%]
ND3  [ 0.00%]                [ 0.00%]                [ 0.00%]                [ 0.00%]

vCPU and Memory Info
System vCPU Usage    [ 8.69%, 9.47%]    Runtime vCPU Usage    [ 3.22%, 5.30%]
Runtime Memory Host  [   2.5MB/  46.0GB]    Runtime Memory Device  198.3MB

Loaded Models
                                                    Model ID        Host Memory     Device Memory
 [-] ND 0                                                           2.5MB           198.3MB
     [-] NC0                                                         638.5KB         49.6MB
         -integ-tests/out-test7_resnet50_v2_fp16_b1_tpb1_tf  10001    638.5KB         49.6MB
     [+] NC1                                                         638.5KB         49.6MB
     [+] NC2                                                         638.5KB         49.6MB
     [+] NC3                                                         638.5KB         49.6MB

Neuron Apps          >[1]:inference app 1<      [2]:inference app 2            [3]:inference app 3            [4]:inference app 4
        q: quit       arrows: move tree selection  enter: expand/collapse tree item  x: expand/collapse entire tree  a/d: previous/next tab      1-9: select tab
```

For more information on resources to monitor and optimize Neuron-based inference applications, see Neuron Tools.

Using Neuron Monitor (neuron-monitor)

Neuron Monitor collects metrics from the Neuron runtimes running on the system and streams the collected data to stdout in JSON format. These metrics are organized into metric groups that you configure by providing a configuration file. For more information on Neuron Monitor, see the User Guide for Neuron Monitor.

Upgrading Neuron Software

For information on how to update Neuron SDK software within DLAMI, see the AWS Neuron Setup Guide.

Next Step

Using the DLAMI with AWS Neuron (p. 84)

Using the DLAMI with AWS Neuron

A typical workflow with the AWS Neuron SDK is to compile a previously trained machine learning model on a compilation server. After this, distribute the artifacts to the Inf1 instances for execution. AWS Deep Learning AMI (DLAMI) comes pre-installed with everything you need to compile and run inference in an Inf1 instance that uses Inferentia.

The following sections describe how to use the DLAMI with Inferentia.

Contents

Using TensorFlow-Neuron and the AWS Neuron Compiler

This tutorial shows how to use the AWS Neuron compiler to compile the Keras ResNet-50 model and export it as a saved model in SavedModel format. This format is a typical TensorFlow model interchangeable format. You also learn how to run inference on an Inf1 instance with example input.

For more information about the Neuron SDK, see the AWS Neuron SDK documentation.

Contents

Prerequisites

Before using this tutorial, you should have completed the set up steps in Launching a DLAMI Instance with AWS Neuron (p. 82). You should also have a familiarity with deep learning and using the DLAMI.

Activate the Conda environment

Activate the TensorFlow-Neuron conda environment using the following command:

```
source activate aws_neuron_tensorflow_p36
```

To exit the current conda environment, run the following command:

```
source deactivate
```

Resnet50 Compilation

Create a Python script called **`tensorflow_compile_resnet50.py`** that has the following content. This Python script compiles the Keras ResNet50 model and exports it as a saved model.

```
import os
import time
import shutil
import tensorflow as tf
import tensorflow.neuron as tfn
import tensorflow.compat.v1.keras as keras
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input

# Create a workspace
WORKSPACE = './ws_resnet50'
os.makedirs(WORKSPACE, exist_ok=True)

# Prepare export directory (old one removed)
model_dir = os.path.join(WORKSPACE, 'resnet50')
compiled_model_dir = os.path.join(WORKSPACE, 'resnet50_neuron')
shutil.rmtree(model_dir, ignore_errors=True)
shutil.rmtree(compiled_model_dir, ignore_errors=True)

# Instantiate Keras ResNet50 model
keras.backend.set_learning_phase(0)
model = ResNet50(weights='imagenet')

# Export SavedModel
tf.saved_model.simple_save(
 session            = keras.backend.get_session(),
 export_dir         = model_dir,
 inputs             = {'input': model.inputs[0]},
 outputs            = {'output': model.outputs[0]})

# Compile using Neuron
tfn.saved_model.compile(model_dir, compiled_model_dir)

# Prepare SavedModel for uploading to Inf1 instance
shutil.make_archive(compiled_model_dir, 'zip', WORKSPACE, 'resnet50_neuron')
```

Compile the model using the following command:

```
python tensorflow_compile_resnet50.py
```

The compilation process will take a few minutes. When it completes, your output should look like the following:

```
...
INFO:tensorflow:fusing subgraph neuron_op_d6f098c01c780733 with neuron-cc
INFO:tensorflow:Number of operations in TensorFlow session: 4638
INFO:tensorflow:Number of operations after tf.neuron optimizations: 556
INFO:tensorflow:Number of operations placed on Neuron runtime: 554
INFO:tensorflow:Successfully converted ./ws_resnet50/resnet50 to ./ws_resnet50/
resnet50_neuron
```

```
...
```

After compilation, the saved model is zipped at **`ws_resnet50/resnet50_neuron.zip`**. Unzip the model and download the sample image for inference using the following commands:

```
unzip ws_resnet50/resnet50_neuron.zip -d .
curl -O https://raw.githubusercontent.com/awslabs/mxnet-model-server/master/docs/images/
kitten_small.jpg
```

ResNet50 Inference

Create a Python script called **`tensorflow_infer_resnet50.py`** that has the following content. This script runs inference on the downloaded model using a previously compiled inference model.

```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications import resnet50

# Create input from image
img_sgl = image.load_img('kitten_small.jpg', target_size=(224, 224))
img_arr = image.img_to_array(img_sgl)
img_arr2 = np.expand_dims(img_arr, axis=0)
img_arr3 = resnet50.preprocess_input(img_arr2)
# Load model
COMPILED_MODEL_DIR = './ws_resnet50/resnet50_neuron/'
predictor_inferentia = tf.contrib.predictor.from_saved_model(COMPILED_MODEL_DIR)
# Run inference
model_feed_dict={'input': img_arr3}
infa_rslts = predictor_inferentia(model_feed_dict);
# Display results
print(resnet50.decode_predictions(infa_rslts["output"], top=5)[0])
```

Run inference on the model using the following command:

```
python tensorflow_infer_resnet50.py
```

Your output should look like the following:

```
...
[('n02123045', 'tabby', 0.6918919), ('n02127052', 'lynx', 0.12770271), ('n02123159',
 'tiger_cat', 0.08277027), ('n02124075', 'Egyptian_cat', 0.06418919), ('n02128757',
 'snow_leopard', 0.009290541)]
```

Next Step

Using AWS Neuron TensorFlow Serving (p. 86)

Using AWS Neuron TensorFlow Serving

This tutorial shows how to construct a graph and add an AWS Neuron compilation step before exporting the saved model to use with TensorFlow Serving. TensorFlow Serving is a serving system that allows you to scale-up inference across a network. Neuron TensorFlow Serving uses the same API as normal TensorFlow Serving. The only difference is that a saved model must be compiled for AWS Inferentia and the entry point is a different binary named `tensorflow_model_server_neuron`. The binary is found at `/usr/local/bin/tensorflow_model_server_neuron` and is pre-installed in the DLAMI.

For more information about the Neuron SDK, see the AWS Neuron SDK documentation.

Contents

Prerequisites

Before using this tutorial, you should have completed the set up steps in Launching a DLAMI Instance with AWS Neuron (p. 82). You should also have a familiarity with deep learning and using the DLAMI.

Activate the Conda environment

Activate the TensorFlow-Neuron conda environment using the following command:

```
source activate aws_neuron_tensorflow_p36
```

If you need to exit the current conda environment, run:

```
source deactivate
```

Compile and Export the Saved Model

Create a Python script called `tensorflow-model-server-compile.py` with the following content. This script constructs a graph and compiles it using Neuron. It then exports the compiled graph as a saved model.

```
import tensorflow as tf
import tensorflow.neuron
import os

tf.keras.backend.set_learning_phase(0)
model = tf.keras.applications.ResNet50(weights='imagenet')
sess = tf.keras.backend.get_session()
inputs = {'input': model.inputs[0]}
outputs = {'output': model.outputs[0]}

# save the model using tf.saved_model.simple_save
modeldir = "./resnet50/1"
tf.saved_model.simple_save(sess, modeldir, inputs, outputs)

# compile the model for Inferentia
neuron_modeldir = os.path.join(os.path.expanduser('~'), 'resnet50_inf1', '1')
tf.neuron.saved_model.compile(modeldir, neuron_modeldir, batch_size=1)
```

Compile the model using the following command:

```
python tensorflow-model-server-compile.py
```

Your output should look like the following:

```
...
INFO:tensorflow:fusing subgraph neuron_op_d6f098c01c780733 with neuron-cc
INFO:tensorflow:Number of operations in TensorFlow session: 4638
INFO:tensorflow:Number of operations after tf.neuron optimizations: 556
INFO:tensorflow:Number of operations placed on Neuron runtime: 554
INFO:tensorflow:Successfully converted ./resnet50/1 to /home/ubuntu/resnet50_inf1/1
```

Serving the Saved Model

Once the model has been compiled, you can use the following command to serve the saved model with the tensorflow_model_server_neuron binary:

```
tensorflow_model_server_neuron --model_name=resnet50_inf1 \
    --model_base_path=$HOME/resnet50_inf1/ --port=8500 &
```

Your output should look like the following. The compiled model is staged in the Inferentia device’s DRAM by the server to prepare for inference.

```
...
2019-11-22 01:20:32.075856: I external/org_tensorflow/tensorflow/cc/saved_model/
loader.cc:311] SavedModel load for tags { serve }; Status: success. Took 40764
 microseconds.
2019-11-22 01:20:32.075888: I tensorflow_serving/servables/tensorflow/
saved_model_warmup.cc:105] No warmup data file found at /home/ubuntu/resnet50_inf1/1/
assets.extra/tf_serving_warmup_requests
2019-11-22 01:20:32.075950: I tensorflow_serving/core/loader_harness.cc:87] Successfully
 loaded servable version {name: resnet50_inf1 version: 1}
2019-11-22 01:20:32.077859: I tensorflow_serving/model_servers/
server.cc:353] Running gRPC ModelServer at 0.0.0.0:8500 ...
```

Generate inference requests to the model server

Create a Python script called `tensorflow-model-server-infer.py` with the following content. This script runs inference via gRPC, which is service framework.

```
import numpy as np
import grpc
import tensorflow as tf
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc
from tensorflow.keras.applications.resnet50 import decode_predictions

if __name__ == '__main__':
    channel = grpc.insecure_channel('localhost:8500')
    stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
    img_file = tf.keras.utils.get_file(
        "./kitten_small.jpg",
        "https://raw.githubusercontent.com/awslabs/mxnet-model-server/master/docs/images/
kitten_small.jpg")
    img = image.load_img(img_file, target_size=(224, 224))
    img_array = preprocess_input(image.img_to_array(img)[None, ...])
    request = predict_pb2.PredictRequest()
    request.model_spec.name = 'resnet50_inf1'
    request.inputs['input'].CopyFrom(
        tf.contrib.util.make_tensor_proto(img_array, shape=img_array.shape))
    result = stub.Predict(request)
    prediction = tf.make_ndarray(result.outputs['output'])
    print(decode_predictions(prediction))
```

Run inference on the model by using gRPC with the following command:

```
python tensorflow-model-server-infer.py
```

Your output should look like the following:

```
[[('n02123045', 'tabby', 0.6918919), ('n02127052', 'lynx', 0.12770271), ('n02123159',
 'tiger_cat', 0.08277027), ('n02124075', 'Egyptian_cat', 0.06418919), ('n02128757',
 'snow_leopard', 0.009290541)]]
```

Using MXNet-Neuron and the AWS Neuron Compiler

The MXNet-Neuron compilation API provides a method to compile a model graph that you can run on an AWS Inferentia device.

In this example, you use the API to compile a ResNet-50 model and use it to run inference.

For more information about the Neuron SDK, see the AWS Neuron SDK documentation.

Contents

Prerequisites

Before using this tutorial, you should have completed the set up steps in Launching a DLAMI Instance with AWS Neuron (p. 82). You should also have a familiarity with deep learning and using the DLAMI.

Activate the Conda Environment

Activate the MXNet-Neuron conda environment using the following command:

```
source activate aws_neuron_mxnet_p36
```

To exit the current conda environment, run:

```
source deactivate
```

Resnet50 Compilation

Create a Python script called **`mxnet_compile_resnet50.py`** with the following content. This script uses the MXNet-Neuron compilation Python API to compile a ResNet-50 model.

```
import mxnet as mx
import numpy as np

print("downloading...")
path='http://data.mxnet.io/models/imagenet/'
mx.test_utils.download(path+'resnet/50-layers/resnet-50-0000.params')
mx.test_utils.download(path+'resnet/50-layers/resnet-50-symbol.json')
print("download finished.")

sym, args, aux = mx.model.load_checkpoint('resnet-50', 0)

print("compile for inferentia using neuron... this will take a few minutes...")
inputs = { "data" : mx.nd.ones([1,3,224,224], name='data', dtype='float32') }

sym, args, aux = mx.contrib.neuron.compile(sym, args, aux, inputs)

print("save compiled model...")
mx.model.save_checkpoint("compiled_resnet50", 0, sym, args, aux)
```

Compile the model using the following command:

```
python mxnet_compile_resnet50.py
```

Compilation will take a few minutes. When compilation has finished, the following files will be in your current directory:

```
resnet-50-0000.params
resnet-50-symbol.json
compiled_resnet50-0000.params
compiled_resnet50-symbol.json
```

ResNet50 Inference

Create a Python script called **`mxnet_infer_resnet50.py`** with the following content. This script downloads a sample image and uses it to run inference with the compiled model.

```
import mxnet as mx
import numpy as np

path='http://data.mxnet.io/models/imagenet/'
mx.test_utils.download(path+'synset.txt')

fname = mx.test_utils.download('https://raw.githubusercontent.com/awslabs/mxnet-model-
server/master/docs/images/kitten_small.jpg')
img = mx.image.imread(fname)

# convert into format (batch, RGB, width, height)
img = mx.image.imresize(img, 224, 224)
# resize
```

```
img = img.transpose((2, 0, 1))
# Channel first
img = img.expand_dims(axis=0)
# batchify
img = img.astype(dtype='float32')

sym, args, aux = mx.model.load_checkpoint('compiled_resnet50', 0)
softmax = mx.nd.random_normal(shape=(1,))
args['softmax_label'] = softmax
args['data'] = img
# Inferentia context
ctx = mx.neuron()

exe = sym.bind(ctx=ctx, args=args, aux_states=aux, grad_req='null')
with open('synset.txt', 'r') as f:
    labels = [l.rstrip() for l in f]

exe.forward(data=img)
prob = exe.outputs[0].asnumpy()
# print the top-5
prob = np.squeeze(prob)
a = np.argsort(prob)[::-1]
for i in a[0:5]:
    print('probability=%f, class=%s' %(prob[i], labels[i]))
```

Run inference with the compiled model using the following command:

```
python mxnet_infer_resnet50.py
```

Your output should look like the following:

```
probability=0.642454, class=n02123045 tabby, tabby cat
probability=0.189407, class=n02123159 tiger cat
probability=0.100798, class=n02124075 Egyptian cat
probability=0.030649, class=n02127052 lynx, catamount
probability=0.016278, class=n02129604 tiger, Panthera tigris
```

Next Step

Using MXNet-Neuron Model Serving (p. 91)

Using MXNet-Neuron Model Serving

In this tutorial, you learn to use a pre-trained MXNet model to perform real-time image classification with Multi Model Server (MMS). MMS is a flexible and easy-to-use tool for serving deep learning models that are trained using any machine learning or deep learning framework. This tutorial includes a compilation step using AWS Neuron and an implementation of MMS using MXNet.

For more information about the Neuron SDK, see the AWS Neuron SDK documentation.

Contents

Prerequisites

Before using this tutorial, you should have completed the set up steps in Launching a DLAMI Instance with AWS Neuron (p. 82). You should also have a familiarity with deep learning and using the DLAMI.

Activate the Conda Environment

Activate the MXNet-Neuron conda environment by using the following command:

```
source activate aws_neuron_mxnet_p36
```

To exit the current conda environment, run:

```
source deactivate
```

Download the Example Code

To run this example, download the example code using the following commands:

```
git clone https://github.com/awslabs/multi-model-server
cd multi-model-server/examples/mxnet_vision
```

Compile the Model

Create a Python script called `multi-model-server-compile.py` with the following content. This script compiles the ResNet50 model to the Inferentia device target.

```
import mxnet as mx
from mxnet.contrib import neuron
import numpy as np

path='http://data.mxnet.io/models/imagenet/'
mx.test_utils.download(path+'resnet/50-layers/resnet-50-0000.params')
mx.test_utils.download(path+'resnet/50-layers/resnet-50-symbol.json')
mx.test_utils.download(path+'synset.txt')

nn_name = "resnet-50"

#Load a model
sym, args, auxs = mx.model.load_checkpoint(nn_name, 0)

#Define compilation parameters#  - input shape and dtype
inputs = {'data' : mx.nd.zeros([1,3,224,224], dtype='float32') }

# compile graph to inferentia target
csym, cargs, cauxs = neuron.compile(sym, args, auxs, inputs)

# save compiled model
mx.model.save_checkpoint(nn_name + "_compiled", 0, csym, cargs, cauxs)
```

To compile the model, use the following command:

```
python multi-model-server-compile.py
```

Your output should look like the following:

```
...
[21:18:40] src/nnvm/legacy_json_util.cc:209: Loading symbol saved by previous version
 v0.8.0. Attempting to upgrade...
```

```
[21:18:40] src/nnvm/legacy_json_util.cc:217: Symbol successfully upgraded!
[21:19:00] src/operator/subgraph/build_subgraph.cc:698: start to execute partition graph.
[21:19:00] src/nnvm/legacy_json_util.cc:209: Loading symbol saved by previous version
 v0.8.0. Attempting to upgrade...
[21:19:00] src/nnvm/legacy_json_util.cc:217: Symbol successfully upgraded!
```

Create a file named `signature.json` with the following content to configure the input name and shape:

```
{
  "inputs": [
    {
      "data_name": "data",
      "data_shape": [
        1,
        3,
        224,
        224
      ]
    }
  ]
}
```

Download the `synset.txt` file by using the following command. This file is a list of names for ImageNet prediction classes.

```
curl -O https://s3.amazonaws.com/model-server/model_archive_1.0/examples/squeezenet_v1.1/
synset.txt
```

Create a custom service class following the template in the `model_server_template` folder. Copy the template into your current working directory by using the following command:

```
cp -r ../model_service_template/* .
```

Edit the `mxnet_model_service.py` module to replace the `mx.cpu()` context with the `mx.neuron()` context as follows. You also need to comment out the unnecessary data copy for `model_input` because MXNet-Neuron does not support the NDArray and Gluon APIs.

```
...
self.mxnet_ctx = mx.neuron() if gpu_id is None else mx.gpu(gpu_id)
...
#model_input = [item.as_in_context(self.mxnet_ctx) for item in model_input]
```

Package the model with model-archiver using the following commands:

```
cd ~/multi-model-server/examples
model-archiver --force --model-name resnet-50_compiled --model-path mxnet_vision --handler
 mxnet_vision_service:handle
```

Run Inference

Start the Multi Model Server and load the model that uses the RESTful API by using the following commands. Ensure that **neuron-rtd** is running with the default settings.

```
cd ~/multi-model-server/
multi-model-server --start --model-store examples > /dev/null # Pipe to log file if you
 want to keep a log of MMS
```

```
curl -v -X POST "http://localhost:8081/models?
initial_workers=1&max_workers=4&synchronous=true&url=resnet-50_compiled.mar"
sleep 10 # allow sufficient time to load model
```

Run inference using an example image with the following commands:

```
curl -O https://raw.githubusercontent.com/awslabs/multi-model-server/master/docs/images/
kitten_small.jpg
curl -X POST http://127.0.0.1:8080/predictions/resnet-50_compiled -T kitten_small.jpg
```

Your output should look like the following:

```
[
  {
    "probability": 0.6388034820556641,
    "class": "n02123045 tabby, tabby cat"
  },
  {
    "probability": 0.16900072991847992,
    "class": "n02123159 tiger cat"
  },
  {
    "probability": 0.12221276015043259,
    "class": "n02124075 Egyptian cat"
  },
  {
    "probability": 0.028706775978207588,
    "class": "n02127052 lynx, catamount"
  },
  {
    "probability": 0.01915954425930977,
    "class": "n02129604 tiger, Panthera tigris"
  }
]
```

To cleanup after the test, issue a delete command via the RESTful API and stop the model server using the following commands:

```
curl -X DELETE http://127.0.0.1:8081/models/resnet-50_compiled

multi-model-server --stop
```

You should see the following output:

```
{
  "status": "Model \"resnet-50_compiled\" unregistered"
}
Model server stopped.
Found 1 models and 1 NCGs.
Unloading 10001 (MODEL_STATUS_STARTED) :: success
Destroying NCG 1 :: success
```

Using PyTorch-Neuron and the AWS Neuron Compiler

The PyTorch-Neuron compilation API provides a method to compile a model graph that you can run on an AWS Inferentia device.

A trained model must be compiled to an Inferentia target before it can be deployed on Inf1 instances. The following tutorial compiles the torchvision ResNet50 model and exports it as a saved TorchScript module. This model is then used to run inference.

For convenience, this tutorial uses an Inf1 instance for both compilation and inference. In practice, you may compile your model using another instance type, such as the c5 instance family. You must then deploy your compiled model to the Inf1 inference server. For more information, see the AWS Neuron PyTorch SDK Documentation.

Contents

Prerequisites

Before using this tutorial, you should have completed the set up steps in Launching a DLAMI Instance with AWS Neuron (p. 82). You should also have a familiarity with deep learning and using the DLAMI.

Activate the Conda Environment

Activate the PyTorch-Neuron conda environment using the following command:

```
source activate aws_neuron_pytorch_p36
```

To exit the current conda environment, run:

```

source deactivate
```

Resnet50 Compilation

Create a Python script called **`pytorch_trace_resnet50.py`** with the following content. This script uses the PyTorch-Neuron compilation Python API to compile a ResNet-50 model.

> **Note**
> There is a dependency between versions of torchvision and the torch package that you should be aware of when compiling torchvision models. These dependency rules can be managed through pip. Torchvision==0.6.1 matches the torch==1.5.1 release, while torchvision==0.8.2 matches the torch==1.7.1 release.

```
import torch
import numpy as np
import os
import torch_neuron
from torchvision import models

image = torch.zeros([1, 3, 224, 224], dtype=torch.float32)

## Load a pretrained ResNet50 model
model = models.resnet50(pretrained=True)

## Tell the model we are using it for evaluation (not training)
model.eval()
model_neuron = torch.neuron.trace(model, example_inputs=[image])

## Export to saved model
model_neuron.save("resnet50_neuron.pt")
```

Run the compilation script.

```
python pytorch_trace_resnet50.py
```

Compilation will take a few minutes. When compilation has finished, the compiled model is saved as `resnet50_neuron.pt` in the local directory.

ResNet50 Inference

Create a Python script called **`pytorch_infer_resnet50.py`** with the following content. This script downloads a sample image and uses it to run inference with the compiled model.

```
import os
import time
import torch
import torch_neuron
import json
import numpy as np

from urllib import request

from torchvision import models, transforms, datasets

## Create an image directory containing a small kitten
os.makedirs("./torch_neuron_test/images", exist_ok=True)
request.urlretrieve("https://raw.githubusercontent.com/awslabs/mxnet-model-server/master/
docs/images/kitten_small.jpg",
                    "./torch_neuron_test/images/kitten_small.jpg")


## Fetch labels to output the top classifications
request.urlretrieve("https://s3.amazonaws.com/deep-learning-models/image-models/
imagenet_class_index.json","imagenet_class_index.json")
idx2label = []

with open("imagenet_class_index.json", "r") as read_file:
    class_idx = json.load(read_file)
    idx2label = [class_idx[str(k)][1] for k in range(len(class_idx))]

## Import a sample image and normalize it into a tensor
normalize = transforms.Normalize(
    mean=[0.485, 0.456, 0.406],
    std=[0.229, 0.224, 0.225])

eval_dataset = datasets.ImageFolder(
    os.path.dirname("./torch_neuron_test/"),
    transforms.Compose([
    transforms.Resize([224, 224]),
    transforms.ToTensor(),
    normalize,
    ])
)

image, _ = eval_dataset[0]
image = torch.tensor(image.numpy()[np.newaxis, ...])

## Load model
model_neuron = torch.jit.load( 'resnet50_neuron.pt' )

## Predict
results = model_neuron( image )

# Get the top 5 results
top5_idx = results[0].sort()[1][-5:]
```

```
# Lookup and print the top 5 labels
top5_labels = [idx2label[idx] for idx in top5_idx]

print("Top 5 labels:\n {}".format(top5_labels) )
```

Run inference with the compiled model using the following command:

```
python pytorch_infer_resnet50.py
```

Your output should look like the following:

```
Top 5 labels:
 ['tiger', 'lynx', 'tiger_cat', 'Egyptian_cat', 'tabby']
```

The Graviton DLAMI

AWS Graviton GPU DLAMIs are designed to provide high performance and cost efficiency for deep learning workloads. Specifically, the G5g instance type features the Arm-based AWS Graviton2 processor, which was built from the ground up by AWS and optimized for how customers run their workloads in the cloud. AWS Graviton GPU DLAMIs are pre-configured with Docker, NVIDIA Docker, NVIDIA Driver, CUDA, CuDNN, NCCL, and TensorRT, as well as popular machine learning frameworks such as TensorFlow and PyTorch.

With the G5g instance type, you can take advantage of the price and performance benefits of Graviton2 to deploy GPU-accelerated deep learning models at a significantly lower cost when compared with x86-based instances with GPU acceleration.

Select a Graviton DLAMI

Launch a G5g instance with the Graviton DLAMI of your choice.

For step-by-step instructions on launching a DLAMI, see Launching and Configuring a DLAMI.

For a list of the most recent Graviton DLAMIs, see the Release Notes for DLAMI.

Get Started

The following topics show you how to get started using the Graviton DLAMI.

Contents

Using the Graviton GPU DLAMI

The AWS Deep Learning AMI is ready to use with Arm processor-based Graviton GPUs. The Graviton GPU DLAMI comes with a foundational platform of GPU drivers and acceleration libraries to deploy your own customized deep learning environment. Docker and NVIDIA Docker are preconfigured on the Graviton GPU DLAMI to let you deploy containerzied applications. Check the release notes for additional details on the Graviton GPU DLAMI.

Contents

Check GPU Status

Use the NVIDIA System Management Interface to check the status of your Graviton GPU.

```
nvidia-smi
```

The output of the `nvidia-smi` command should be similar to the following:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.82.01    Driver Version: 470.82.01    CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA T4G          On   | 00000000:00:1F.0 Off |                    0 |
| N/A   32C    P8     8W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Check CUDA Version

Run the following command to check your CUDA version:

```
/usr/local/cuda/bin/nvcc --version | grep Cuda
```

Your output should look similar to the following:

```
nvcc: NVIDIA (R) Cuda compiler driver
Cuda compilation tools, release 11.4, V11.4.120
```

Verify Docker

Run a CUDA container from DockerHub to verify Docker functionality on your Graviton GPU:

```
sudo docker run --platform=linux/arm64 --rm \
    --gpus all nvidia/cuda:11.4.2-base-ubuntu20.04 nvidia-smi
```

Your output should look similar to the following:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.82.01    Driver Version: 470.82.01    CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA T4G          On   | 00000000:00:1F.0 Off |                    0 |
| N/A   33C    P8     9W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

TensorRT

Use the following command to access the TensorRT command line tool:

```
trtexec
```

Your output should look similar to the following:

```
&&&& RUNNING TensorRT.trtexec [TensorRT v8200] # trtexec
...
&&&& PASSED TensorRT.trtexec [TensorRT v8200] # trtexec
```

There are TensorRT Python wheels available for installation on demand. You can find these wheels in the following file locations:

```
/usr/local/tensorrt/graphsurgeon/
### graphsurgeon-0.4.5-py2.py3-none-any.whl

/usr/local/tensorrt/onnx_graphsurgeon/
### onnx_graphsurgeon-0.3.12-py2.py3-none-any.whl

/usr/local/tensorrt/python/
### tensorrt-8.2.0.6-cp36-none-linux_aarch64.whl
### tensorrt-8.2.0.6-cp37-none-linux_aarch64.whl
### tensorrt-8.2.0.6-cp38-none-linux_aarch64.whl
### tensorrt-8.2.0.6-cp39-none-linux_aarch64.whl

/usr/local/tensorrt/uff/
### uff-0.6.9-py2.py3-none-any.whl
```

For additional details, see the NVIDIA TensorRT documentation.

Run CUDA Samples

The Graviton GPU DLAMI provides pre-compiled CUDA samples to help you verify different CUDA functionalities.

```
ls /usr/local/cuda/compiled_samples
```

For example, run the `vectorAdd` sample with the following command:

```
/usr/local/cuda/compiled_samples/vectorAdd
```

Your output should look similar to the following:

```
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

Run the `transpose` sample:

```
/usr/local/cuda/compiled_samples/transpose
```

Your output should look similar to the following:

```
Transpose Starting...

GPU Device 0: "Turing" with compute capability 7.5

> Device 0: "NVIDIA T4G"
> SM Capability 7.5 detected:
> [NVIDIA T4G] has 40 MP(s) x 64 (Cores/MP) = 2560 (Cores)
> Compute performance scaling factor = 1.00

Matrix size: 1024x1024 (64x64 tiles), tile size: 16x16, block size: 16x16

transpose simple copy       , Throughput = 185.1781 GB/s, Time = 0.04219 ms, Size = 1048576
 fp32 elements, NumDevsUsed = 1, Workgroup = 256
transpose shared memory copy, Throughput = 163.8616 GB/s, Time = 0.04768 ms, Size = 1048576
 fp32 elements, NumDevsUsed = 1, Workgroup = 256
transpose naive             , Throughput = 98.2805 GB/s, Time = 0.07949 ms, Size = 1048576
 fp32 elements, NumDevsUsed = 1, Workgroup = 256
transpose coalesced         , Throughput = 127.6759 GB/s, Time = 0.06119 ms, Size = 1048576
 fp32 elements, NumDevsUsed = 1, Workgroup = 256
transpose optimized         , Throughput = 156.2960 GB/s, Time = 0.04999 ms, Size = 1048576
 fp32 elements, NumDevsUsed = 1, Workgroup = 256
transpose coarse-grained    , Throughput = 155.9157 GB/s, Time = 0.05011 ms, Size = 1048576
 fp32 elements, NumDevsUsed = 1, Workgroup = 256
transpose fine-grained      , Throughput = 158.4177 GB/s, Time = 0.04932 ms, Size = 1048576
 fp32 elements, NumDevsUsed = 1, Workgroup = 256
transpose diagonal          , Throughput = 133.4277 GB/s, Time = 0.05855 ms, Size = 1048576
 fp32 elements, NumDevsUsed = 1, Workgroup = 256
Test passed
```

Next Up

Using the Graviton GPU TensorFlow DLAMI (p. 100)

Using the Graviton GPU TensorFlow DLAMI

The AWS Deep Learning AMI is ready to use with Arm processor-based Graviton GPUs, and comes optimized for TensorFlow. The Graviton GPU TensorFlow DLAMI includes a Python environment pre-configured with TensorFlow Serving for deep learning inference use cases. Check the release notes for additional details on the Graviton GPU TensorFlow DLAMI.

Contents

Verify TensorFlow Serving Availability

Run the following command to verify the availability and version of TensorFlow Serving:

```
tensorflow_model_server --version
```

Your output should look similar to the following:

```
TensorFlow ModelServer: 0.0.0+dev.sha.3e05381e
TensorFlow Library: 2.8.0
```

Verify TensorFlow and TensorFlow Serving API Availability

Run the following command to verify the availability of TensorFlow and the TensorFlow Serving API:

```
python3 -c "import tensorflow, tensorflow_serving"
```

If the command is successful, there is no output.

Run Example Inference with TensorFlow Serving

Use the following commands to download a pre-trained ResNet50 model and run inference using TensorFlow Serving:

```
# Clone the TensorFlow Serving repository
git clone https://github.com/tensorflow/serving

# Download pre-trained ResNet50 model
mkdir -p ${HOME}/resnet/1 && cd ${HOME}/resnet/1
wget https://tfhub.dev/tensorflow/resnet_50/classification/1?tf-hub-format=compressed -O
 resnet_50_classification_1.tar.gz
tar -xzvf resnet_50_classification_1.tar.gz && rm resnet_50_classification_1.tar.gz

# Start TensorFlow Serving
cd $HOME
tensorflow_model_server \
  --rest_api_port=8501 \
  --model_name="resnet" \
  --model_base_path="${HOME}/resnet" &
```

Your output should look similar to the following:

```
2021-11-10 06:18:51.028341: I tensorflow_serving/model_servers/server_core.cc:486] Finished
 adding/updating models
2021-11-10 06:18:51.028420: I tensorflow_serving/model_servers/server.cc:133] Using
 InsecureServerCredentials
2021-11-10 06:18:51.028460: I tensorflow_serving/model_servers/server.cc:383] Profiler
 service is enabled
2021-11-10 06:18:51.028889: I tensorflow_serving/model_servers/server.cc:409] Running gRPC
 ModelServer at 0.0.0.0:8500 ...
[evhttp_server.cc : 245] NET_LOG: Entering the event loop ...
2021-11-10 06:18:51.030985: I tensorflow_serving/model_servers/server.cc:430] Exporting
 HTTP/REST API at:localhost:8501 ...
```

Use the TensorFlow Serving `resnet_client` example to run inference:

```
python3 serving/tensorflow_serving/example/resnet_client.py
```

Your output should look similar to the following:

```
2021-11-10 06:18:59.335327: I external/org_tensorflow/tensorflow/stream_executor/cuda/
cuda_dnn.cc:368] Loaded cuDNN version 8204
2021-11-10 06:18:59.956156: I external/org_tensorflow/tensorflow/core/platform/default/
subprocess.cc:304] Start cannot spawn child process
Prediction class: 285, avg latency: 111.4673 ms
```

Stop TensorFlow Serving with the following command:

```
kill $(pidof tensorflow_model_server)
```

Next Up

Using the Graviton GPU PyTorch DLAMI (p. 102)

Using the Graviton GPU PyTorch DLAMI

The AWS Deep Learning AMI is ready to use with Arm processor-based Graviton GPUs, and comes optimized for PyTorch. The Graviton GPU PyTorch DLAMI includes a Python environment pre-configured with PyTorch, TorchVision, and TorchServe for deep learning training and inference use cases. Check the release notes for additional details on the Graviton GPU PyTorch DLAMI.

Contents

Verify PyTorch Python Environment

Connect to your G5g instance and activate the base Conda environment with the following command:

```
source activate base
```

Your command prompt should indicate that you are working in the base Conda environment, which contains PyTorch, TorchVision, and other libraries.

```
(base) $
```

Verify the default tool paths of the PyTorch environment:

```
(base) $ which python
/opt/conda/bin/python

(base) $ which pip
/opt/conda/bin/pip

(base) $ which conda
/opt/conda/bin/conda

(base) $ which mamba
```

```
/opt/conda/bin/mamba
```

Verify that Torch and TorchVersion are available, check their versions, and test for basic functionality:

```
(base) $ python
Python 3.8.12 | packaged by conda-forge | (default, Oct 12 2021, 23:06:28)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch, torchvision
>>> torch.__version__
'1.10.0'
>>> torchvision.__version__
'0.11.1'
>>> v = torch.autograd.Variable(torch.randn(10, 3, 224, 224))
>>> v = torch.autograd.Variable(torch.randn(10, 3, 224, 224)).cuda()
>>> assert isinstance(v, torch.Tensor)
```

Run Training Sample with PyTorch

Run a sample MNIST training job:

```
git clone https://github.com/pytorch/examples.git
cd examples/mnist
python main.py
```

Your output should look similar to the following:

```
...
Train Epoch: 14 [56320/60000 (94%)]    Loss: 0.021424
Train Epoch: 14 [56960/60000 (95%)]    Loss: 0.023695
Train Epoch: 14 [57600/60000 (96%)]    Loss: 0.001973
Train Epoch: 14 [58240/60000 (97%)]    Loss: 0.007121
Train Epoch: 14 [58880/60000 (98%)]    Loss: 0.003717
Train Epoch: 14 [59520/60000 (99%)]    Loss: 0.001729
Test set: Average loss: 0.0275, Accuracy: 9916/10000 (99%)
```

Run Inference Sample with PyTorch

Use the following commands to download a pre-trained densenet161 model and run inference using TorchServe:

```
# Set up TorchServe
cd $HOME
git clone https://github.com/pytorch/serve.git
mkdir -p serve/model_store
cd serve

# Download a pre-trained densenet161 model
wget https://download.pytorch.org/models/densenet161-8d451a50.pth >/dev/null

# Save the model using torch-model-archiver
torch-model-archiver --model-name densenet161 \
    --version 1.0 \
    --model-file examples/image_classifier/densenet_161/model.py \
    --serialized-file densenet161-8d451a50.pth \
    --handler image_classifier \
    --extra-files examples/image_classifier/index_to_name.json  \
    --export-path model_store

# Start the model server
```

```
torchserve --start --no-config-snapshots \
    --model-store model_store \
    --models densenet161=densenet161.mar &> torchserve.log

# Wait for the model server to start
sleep 30

# Run a prediction request
curl http://127.0.0.1:8080/predictions/densenet161 -T examples/image_classifier/kitten.jpg
```

Your output should look similar to the following:

```
{
  "tiger_cat": 0.4693363308906555,
  "tabby": 0.4633873701095581,
  "Egyptian_cat": 0.06456123292446136,
  "lynx": 0.0012828150065615773,
  "plastic_bag": 0.00023322898778133094
}
```

Use the following commands to unregister the densenet161 model and stop the server:

```
curl -X DELETE http://localhost:8081/models/densenet161/1.0
torchserve --stop
```

Your output should look similar to the following:

```
{
  "status": "Model \"densenet161\" unregistered"
}
TorchServe has stopped.
```

The Habana DLAMI

Instances with Habana accelerators are designed to provide high performance and cost efficiency for deep learning model training workloads. Specifically, DL1 instance types use Habana Gaudi accelerators from Habana Labs, an Intel company. Instances with Habana accelerators are configured with Habana SynapseAI software and pre-integrated with popular machine learning frameworks such as TensorFlow and PyTorch.

The following topics show you how to get started using Habana Gaudi hardware with the DLAMI.

Contents

Launching a Habana DLAMI

The latest DLAMI is ready to use with Habana Gaudi accelerators. Use the following steps to launch your Habana DLAMI and ensure that your Python and framework-specific resources are active. For additional setup resources, see the Habana Gaudi Setup and Installation respository.

Contents

Select a Habana DLAMI

Launch a DL1 instance with the Habana DLAMI of your choice.

For step-by-step instructions on launching a DLAMI, see Launching and Configuring a DLAMI.

For a list of the most recent Habana DLAMIs, see the Release Notes for DLAMI.

Activate Python Environment

Connect to your DL1 instance and activate the recommended Python environment for your Habana DLAMI. To check your recommended Python environment, select your DLAMI in the Release Notes.

Import Machine Learning Framework

Instances with Habana accelerators are pre-integrated with popular machine learning frameworks such as TensorFlow and PyTorch. Import the machine learning framework of your choice.

Import TensorFlow

To use TensorFlow on your Habana DLAMI, navigate to the folder of the Python environment that you activated and import TensorFlow.

```
/usr/bin/$PYTHON_VERSION
import tensorflow
tensorflow.__version__
```

To check the TensorFlow version compatible with your Habana DLAMI, select your DLAMI in the Release Notes.

Import PyTorch

To use PyTorch on your Habana DLAMI, navigate to the folder of the Python environment that you activated and import the appropriate PyTorch version.

```
/usr/bin/$PYTHON_VERSION
import torch
torch.__version__
```

To check the PyTorch version compatible with your Habana DLAMI, select your DLAMI in the Release Notes.

For more information on how to run and train machine learning models in TensorFlow and PyTorch using your Habana DLAMI, see the Habana Model References GitHub repository. For additional resources on working with your Habana DLAMI, visit the Habana Gaudi documentation.

Inference

This section provides tutorials on how to run inference using the DLAMI's frameworks and tools.

For tutorials using Elastic Inference, see Working with Amazon Elastic Inference

Inference with Frameworks

Inference Tools

- Model Server for Apache MXNet (MMS) (p. 118)
- TensorFlow Serving (p. 120)

Use Apache MXNet (Incubating) for Inference with an ONNX Model

How to Use an ONNX Model for Image Inference with Apache MXNet (Incubating)

1. - (Option for Python 3) - Activate the Python 3 Apache MXNet (Incubating) environment:

     ```
     $ source activate mxnet_p36
     ```

   - (Option for Python 2) - Activate the Python 2 Apache MXNet (Incubating) environment:

     ```
     $ source activate mxnet_p27
     ```

2. The remaining steps assume you are using the `mxnet_p36` environment.
3. Download a picture of a husky.

   ```
   $ curl -O https://upload.wikimedia.org/wikipedia/commons/b/b5/Siberian_Husky_bi-
   eyed_Flickr.jpg
   ```

4. Download a list of classes that work with this model.

   ```
   $ curl -O https://gist.githubusercontent.com/yrevar/6135f1bd8dcf2e0cc683/raw/
   d133d61a09d7e5a3b36b8c111a8dd5c4b5d560ee/imagenet1000_clsid_to_human.pkl
   ```

5. Download the pre-trained VGG 16 model in ONNX format.

   ```
   $ wget -O vgg16.onnx https://github.com/onnx/models/raw/master/vision/classification/
   vgg/model/vgg16-7.onnx
   ```

6. Use a your preferred text editor to create a script that has the following content. This script will use the image of the husky, get a prediction result from the pre-trained model, then look this up in the file of classes, returning an image classification result.

   ```
   import mxnet as mx
   import mxnet.contrib.onnx as onnx_mxnet
   import numpy as np
   from collections import namedtuple
   from PIL import Image
   import pickle

   # Preprocess the image
   img = Image.open("Siberian_Husky_bi-eyed_Flickr.jpg")
   img = img.resize((224,224))
   rgb_img = np.asarray(img, dtype=np.float32) - 128
   bgr_img = rgb_img[..., [2,1,0]]
   img_data = np.ascontiguousarray(np.rollaxis(bgr_img,2))
   img_data = img_data[np.newaxis, :, :, :].astype(np.float32)

   # Define the model's input
   data_names = ['data']
   Batch = namedtuple('Batch', data_names)
   ```

```
# Set the context to cpu or gpu
ctx = mx.cpu()

# Load the model
sym, arg, aux = onnx_mxnet.import_model("vgg16.onnx")
mod = mx.mod.Module(symbol=sym, data_names=data_names, context=ctx, label_names=None)
mod.bind(for_training=False, data_shapes=[(data_names[0],img_data.shape)],
 label_shapes=None)
mod.set_params(arg_params=arg, aux_params=aux, allow_missing=True, allow_extra=True)

# Run inference on the image
mod.forward(Batch([mx.nd.array(img_data)]))
predictions = mod.get_outputs()[0].asnumpy()
top_class = np.argmax(predictions)
print(top_class)
labels_dict = pickle.load(open("imagenet1000_clsid_to_human.pkl", "rb"))
print(labels_dict[top_class])
```

7. Then run the script, and you should see a result as follows:

```
248
Eskimo dog, husky
```

Use Apache MXNet (Incubating) for Inference with a ResNet 50 Model

How to Use a Pre-Trained Apache MXNet (Incubating) Model with the Symbol API for Image Inference with MXNet

1. • (Option for Python 3) - Activate the Python 3 Apache MXNet (Incubating) environment:

   ```
   $ source activate mxnet_p36
   ```

   • (Option for Python 2) - Activate the Python 2 Apache MXNet (Incubating) environment:

   ```
   $ source activate mxnet_p27
   ```

2. The remaining steps assume you are using the `mxnet_p36` environment.
3. Use a your preferred text editor to create a script that has the following content. This script will download the ResNet-50 model files (resnet-50-0000.params and resnet-50-symbol.json) and labels list (synset.txt), download a cat image to get a prediction result from the pre-trained model, then look this up in the result in labels list, returning a prediction result.

```
import mxnet as mx
import numpy as np

path='http://data.mxnet.io/models/imagenet/'
[mx.test_utils.download(path+'resnet/50-layers/resnet-50-0000.params'),
 mx.test_utils.download(path+'resnet/50-layers/resnet-50-symbol.json'),
 mx.test_utils.download(path+'synset.txt')]

ctx = mx.cpu()

with open('synset.txt', 'r') as f:
    labels = [l.rstrip() for l in f]

sym, args, aux = mx.model.load_checkpoint('resnet-50', 0)
```

```
fname = mx.test_utils.download('https://github.com/dmlc/web-data/blob/master/mxnet/doc/
tutorials/python/predict_image/cat.jpg?raw=true')
img = mx.image.imread(fname)
# convert into format (batch, RGB, width, height)
img = mx.image.imresize(img, 224, 224) # resize
img = img.transpose((2, 0, 1)) # Channel first
img = img.expand_dims(axis=0) # batchify
img = img.astype(dtype='float32')
args['data'] = img

softmax = mx.nd.random_normal(shape=(1,))
args['softmax_label'] = softmax

exe = sym.bind(ctx=ctx, args=args, aux_states=aux, grad_req='null')

exe.forward()
prob = exe.outputs[0].asnumpy()
# print the top-5
prob = np.squeeze(prob)
a = np.argsort(prob)[::-1]
for i in a[0:5]:
    print('probability=%f, class=%s' %(prob[i], labels[i]))
```

4. Then run the script, and you should see a result as follows:

```
probability=0.418679, class=n02119789 kit fox, Vulpes macrotis
probability=0.293495, class=n02119022 red fox, Vulpes vulpes
probability=0.029321, class=n02120505 grey fox, gray fox, Urocyon cinereoargenteus
probability=0.026230, class=n02124075 Egyptian cat
probability=0.022557, class=n02085620 Chihuahua
```

Use CNTK for Inference with an ONNX Model

Note
We no longer include the CNTK, Caffe, Caffe2 and Theano Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

Note
The VGG-16 model used in this tutorial consumes a large amount of memory. When selecting your AWS Deep Learning AMI instance, you may need an instance with more than 30 GB of RAM.

How to Use an ONNX Model for Inference with CNTK

1. - (Option for Python 3) - Activate the Python 3 CNTK environment:

     ```
     $ source activate cntk_p36
     ```

   - (Option for Python 2) - Activate the Python 2 CNTK environment:

     ```
     $ source activate cntk_p27
     ```

2. The remaining steps assume you are using the `cntk_p36` environment.
3. Create a new file with your text editor, and use the following program in a script to open ONNX format file in CNTK.

```
import cntk as C
```

```
# Import the Chainer model into CNTK via the CNTK import API
z = C.Function.load("vgg16.onnx", device=C.device.cpu(), format=C.ModelFormat.ONNX)
print("Loaded vgg16.onnx!")
```

After you run this script, CNTK will have loaded the model.

4. You may also try running inference with CNTK. First, download a picture of a husky.

```
$ curl -O https://upload.wikimedia.org/wikipedia/commons/b/b5/Siberian_Husky_bi-
eyed_Flickr.jpg
```

5. Next, download a list of classes will work with this model.

```
$ curl -O https://gist.githubusercontent.com/yrevar/6135f1bd8dcf2e0cc683/raw/
d133d61a09d7e5a3b36b8c111a8dd5c4b5d560ee/imagenet1000_clsid_to_human.pkl
```

6. Edit the previously created script to have the following content. This new version will use the image of the husky, get a prediction result, then look this up in the file of classes, returning a prediction result.

```
import cntk as C
import numpy as np
from PIL import Image
from IPython.core.display import display
import pickle

# Import the model into CNTK via the CNTK import API
z = C.Function.load("vgg16.onnx", device=C.device.cpu(), format=C.ModelFormat.ONNX)
print("Loaded vgg16.onnx!")
img = Image.open("Siberian_Husky_bi-eyed_Flickr.jpg")
img = img.resize((224,224))
rgb_img = np.asarray(img, dtype=np.float32) - 128
bgr_img = rgb_img[..., [2,1,0]]
img_data = np.ascontiguousarray(np.rollaxis(bgr_img,2))
predictions = np.squeeze(z.eval({z.arguments[0]:[img_data]}))
top_class = np.argmax(predictions)
print(top_class)
labels_dict = pickle.load(open("imagenet1000_clsid_to_human.pkl", "rb"))
print(labels_dict[top_class])
```

7. Then run the script, and you should see a result as follows:

```
248
Eskimo dog, husky
```

Using Frameworks with ONNX

The Deep Learning AMI with Conda now supports Open Neural Network Exchange (ONNX) models for some frameworks. Choose one of the topics listed below to learn how to use ONNX on your Deep Learning AMI with Conda.

If you want to use an existing ONNX model on a DLAMI, see Use Apache MXNet (Incubating) for Inference with an ONNX Model (p. 106).

About ONNX

The Open Neural Network Exchange (ONNX) is an open format used to represent deep learning models. ONNX is supported by Amazon Web Services, Microsoft, Facebook, and several other partners. You can design, train, and deploy deep learning models with any framework you choose. The benefit of ONNX models is that they can be moved between frameworks with ease.

The Deep Learning AMI with Conda currently highlights some of the ONNX features in the following collection of tutorials.

- Apache MXNet to ONNX to CNTK Tutorial (p. 110)
- Chainer to ONNX to CNTK Tutorial (p. 111)
- Chainer to ONNX to MXNet Tutorial (p. 113)
- PyTorch to ONNX to CNTK Tutorial (p. 114)
- PyTorch to ONNX to MXNet Tutorial (p. 116)

You might also want to refer to the ONNX project documentation and tutorials:

- ONNX Project on GitHub
- ONNX Tutorials

Apache MXNet to ONNX to CNTK Tutorial

Note
We no longer include the CNTK, Caffe, Caffe2 and Theano Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

ONNX Overview

The Open Neural Network Exchange (ONNX) is an open format used to represent deep learning models. ONNX is supported by Amazon Web Services, Microsoft, Facebook, and several other partners. You can design, train, and deploy deep learning models with any framework you choose. The benefit of ONNX models is that they can be moved between frameworks with ease.

This tutorial shows you how to use the Deep Learning AMI with Conda with ONNX. By following these steps, you can train a model or load a pre-trained model from one framework, export this model to ONNX, and then import the model in another framework.

ONNX Prerequisites

To use this ONNX tutorial, you must have access to a Deep Learning AMI with Conda version 12 or later. For more information about how to get started with a Deep Learning AMI with Conda, see Deep Learning AMI with Conda (p. 6).

Important
These examples use functions that might require up to 8 GB of memory (or more). Be sure to choose an instance type with enough memory.

Launch a terminal session with your Deep Learning AMI with Conda to begin the following tutorial.

Convert an Apache MXNet (incubating) Model to ONNX, then Load the Model into CNTK

How to Export a Model from Apache MXNet (incubating)

You can install the latest MXNet build into either or both of the MXNet Conda environments on your Deep Learning AMI with Conda.

1. - (Option for Python 3) - Activate the Python 3 MXNet environment:

     ```
     $ source activate mxnet_p36
     ```

   - (Option for Python 2) - Activate the Python 2 MXNet environment:

     ```
     $ source activate mxnet_p27
     ```

2. The remaining steps assume that you are using the `mxnet_p36` environment.
3. Download the model files.

   ```
   curl -O https://s3.amazonaws.com/onnx-mxnet/model-zoo/vgg16/vgg16-symbol.json
   curl -O https://s3.amazonaws.com/onnx-mxnet/model-zoo/vgg16/vgg16-0000.params
   ```

4. To export the model files from MXNet to the ONNX format, create a new file with your text editor and use the following program in a script.

   ```
   import numpy as np
   import mxnet as mx
   from mxnet.contrib import onnx as onnx_mxnet
   converted_onnx_filename='vgg16.onnx'

   # Export MXNet model to ONNX format via MXNet's export_model API
   converted_onnx_filename=onnx_mxnet.export_model('vgg16-symbol.json',
    'vgg16-0000.params', [(1,3,224,224)], np.float32, converted_onnx_filename)

   # Check that the newly created model is valid and meets ONNX specification.
   import onnx
   model_proto = onnx.load(converted_onnx_filename)
   onnx.checker.check_model(model_proto)
   ```

   You may see some warning messages, but you can safely ignore those for now. After you run this script, you will see the newly created .onnx file in the same directory.
5. Now that you have an ONNX file you can try running inference with it with the following example:
   - Use CNTK for Inference with an ONNX Model (p. 108)

ONNX Tutorials

- Apache MXNet to ONNX to CNTK Tutorial (p. 110)
- Chainer to ONNX to CNTK Tutorial (p. 111)
- Chainer to ONNX to MXNet Tutorial (p. 113)
- PyTorch to ONNX to MXNet Tutorial (p. 116)
- PyTorch to ONNX to CNTK Tutorial (p. 114)

Chainer to ONNX to CNTK Tutorial

Note
We no longer include the CNTK, Caffe, Caffe2 and Theano Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

ONNX Overview

The Open Neural Network Exchange (ONNX) is an open format used to represent deep learning models. ONNX is supported by Amazon Web Services, Microsoft, Facebook, and several other partners. You can design, train, and deploy deep learning models with any framework you choose. The benefit of ONNX models is that they can be moved between frameworks with ease.

This tutorial shows you how to use the Deep Learning AMI with Conda with ONNX. By following these steps, you can train a model or load a pre-trained model from one framework, export this model to ONNX, and then import the model in another framework.

ONNX Prerequisites

To use this ONNX tutorial, you must have access to a Deep Learning AMI with Conda version 12 or later. For more information about how to get started with a Deep Learning AMI with Conda, see Deep Learning AMI with Conda (p. 6).

Important
These examples use functions that might require up to 8 GB of memory (or more). Be sure to choose an instance type with enough memory.

Launch a terminal session with your Deep Learning AMI with Conda to begin the following tutorial.

Convert a Chainer Model to ONNX, then Load the Model into CNTK

First, activate the Chainer environment:

```
$ source activate chainer_p36
```

Create a new file with your text editor, and use the following program in a script to fetch a model from Chainer's model zoo, then export it to the ONNX format.

```
import numpy as np
import chainer
import chainercv.links as L
import onnx_chainer

# Fetch a vgg16 model
model = L.VGG16(pretrained_model='imagenet')

# Prepare an input tensor
x = np.random.rand(1, 3, 224, 224).astype(np.float32) * 255

# Run the model on the data
with chainer.using_config('train', False):
  chainer_out = model(x).array

# Export the model to a .onnx file
out = onnx_chainer.export(model, x, filename='vgg16.onnx')

# Check that the newly created model is valid and meets ONNX specification.
import onnx
model_proto = onnx.load("vgg16.onnx")
onnx.checker.check_model(model_proto)
```

After you run this script, you will see the newly created .onnx file in the same directory.

Now that you have an ONNX file you can try running inference with it with the following example:

- Use CNTK for Inference with an ONNX Model (p. 108)

ONNX Tutorials

- Apache MXNet to ONNX to CNTK Tutorial (p. 110)
- Chainer to ONNX to CNTK Tutorial (p. 111)
- Chainer to ONNX to MXNet Tutorial (p. 113)
- PyTorch to ONNX to MXNet Tutorial (p. 116)
- PyTorch to ONNX to CNTK Tutorial (p. 114)

Chainer to ONNX to MXNet Tutorial

ONNX Overview

The Open Neural Network Exchange (ONNX) is an open format used to represent deep learning models. ONNX is supported by Amazon Web Services, Microsoft, Facebook, and several other partners. You can design, train, and deploy deep learning models with any framework you choose. The benefit of ONNX models is that they can be moved between frameworks with ease.

This tutorial shows you how to use the Deep Learning AMI with Conda with ONNX. By following these steps, you can train a model or load a pre-trained model from one framework, export this model to ONNX, and then import the model in another framework.

ONNX Prerequisites

To use this ONNX tutorial, you must have access to a Deep Learning AMI with Conda version 12 or later. For more information about how to get started with a Deep Learning AMI with Conda, see Deep Learning AMI with Conda (p. 6).

> **Important**
> These examples use functions that might require up to 8 GB of memory (or more). Be sure to choose an instance type with enough memory.

Launch a terminal session with your Deep Learning AMI with Conda to begin the following tutorial.

Convert a Chainer Model to ONNX, then Load the Model into MXNet

First, activate the Chainer environment:

```
$ source activate chainer_p36
```

Create a new file with your text editor, and use the following program in a script to fetch a model from Chainer's model zoo, then export it to the ONNX format.

```
import numpy as np
import chainer
import chainercv.links as L
import onnx_chainer

# Fetch a vgg16 model
model = L.VGG16(pretrained_model='imagenet')

# Prepare an input tensor
x = np.random.rand(1, 3, 224, 224).astype(np.float32) * 255

# Run the model on the data
with chainer.using_config('train', False):
  chainer_out = model(x).array
```

```
# Export the model to a .onnx file
out = onnx_chainer.export(model, x, filename='vgg16.onnx')

# Check that the newly created model is valid and meets ONNX specification.
import onnx
model_proto = onnx.load("vgg16.onnx")
onnx.checker.check_model(model_proto)
```

After you run this script, you will see the newly created .onnx file in the same directory.

Now that you have an ONNX file you can try running inference with it with the following example:

- Use Apache MXNet (Incubating) for Inference with an ONNX Model (p. 106)

ONNX Tutorials

- Apache MXNet to ONNX to CNTK Tutorial (p. 110)
- Chainer to ONNX to CNTK Tutorial (p. 111)
- Chainer to ONNX to MXNet Tutorial (p. 113)
- PyTorch to ONNX to MXNet Tutorial (p. 116)
- PyTorch to ONNX to CNTK Tutorial (p. 114)

PyTorch to ONNX to CNTK Tutorial

Note
We no longer include the CNTK, Caffe, Caffe2 and Theano Conda environments in the AWS Deep Learning AMI starting with the v28 release. Previous releases of the AWS Deep Learning AMI that contain these environments will continue to be available. However, we will only provide updates to these environments if there are security fixes published by the open source community for these frameworks.

ONNX Overview

The Open Neural Network Exchange (ONNX) is an open format used to represent deep learning models. ONNX is supported by Amazon Web Services, Microsoft, Facebook, and several other partners. You can design, train, and deploy deep learning models with any framework you choose. The benefit of ONNX models is that they can be moved between frameworks with ease.

This tutorial shows you how to use the Deep Learning AMI with Conda with ONNX. By following these steps, you can train a model or load a pre-trained model from one framework, export this model to ONNX, and then import the model in another framework.

ONNX Prerequisites

To use this ONNX tutorial, you must have access to a Deep Learning AMI with Conda version 12 or later. For more information about how to get started with a Deep Learning AMI with Conda, see Deep Learning AMI with Conda (p. 6).

Important
These examples use functions that might require up to 8 GB of memory (or more). Be sure to choose an instance type with enough memory.

Launch a terminal session with your Deep Learning AMI with Conda to begin the following tutorial.

Convert a PyTorch Model to ONNX, then Load the Model into CNTK

First, activate the PyTorch environment:

```
$ source activate pytorch_p36
```

Create a new file with your text editor, and use the following program in a script to train a mock model in PyTorch, then export it to the ONNX format.

```
# Build a Mock Model in Pytorch with a convolution and a reduceMean layer\
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
import torch.onnx as torch_onnx

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=(3,3), stride=1,
 padding=0, bias=False)

    def forward(self, inputs):
        x = self.conv(inputs)
        #x = x.view(x.size()[0], x.size()[1], -1)
        return torch.mean(x, dim=2)

# Use this an input trace to serialize the model
input_shape = (3, 100, 100)
model_onnx_path = "torch_model.onnx"
model = Model()
model.train(False)

# Export the model to an ONNX file
dummy_input = Variable(torch.randn(1, *input_shape))
output = torch_onnx.export(model,
                          dummy_input,
                          model_onnx_path,
                          verbose=False)
```

After you run this script, you will see the newly created .onnx file in the same directory. Now, switch to the CNTK Conda environment to load the model with CNTK.

Next, activate the CNTK environment:

```
$ source deactivate
$ source activate cntk_p36
```

Create a new file with your text editor, and use the following program in a script to open ONNX format file in CNTK.

```
import cntk as C
# Import the PyTorch model into CNTK via the CNTK import API
z = C.Function.load("torch_model.onnx", device=C.device.cpu(), format=C.ModelFormat.ONNX)
```

After you run this script, CNTK will have loaded the model.

You may also export to ONNX using CNTK by appending the following to your previous script then running it.

```
# Export the model to ONNX via the CNTK export API
```

```
z.save("cntk_model.onnx", format=C.ModelFormat.ONNX)
```

ONNX Tutorials

- Apache MXNet to ONNX to CNTK Tutorial (p. 110)
- Chainer to ONNX to CNTK Tutorial (p. 111)
- Chainer to ONNX to MXNet Tutorial (p. 113)
- PyTorch to ONNX to MXNet Tutorial (p. 116)
- PyTorch to ONNX to CNTK Tutorial (p. 114)

PyTorch to ONNX to MXNet Tutorial

ONNX Overview

The Open Neural Network Exchange (ONNX) is an open format used to represent deep learning models. ONNX is supported by Amazon Web Services, Microsoft, Facebook, and several other partners. You can design, train, and deploy deep learning models with any framework you choose. The benefit of ONNX models is that they can be moved between frameworks with ease.

This tutorial shows you how to use the Deep Learning AMI with Conda with ONNX. By following these steps, you can train a model or load a pre-trained model from one framework, export this model to ONNX, and then import the model in another framework.

ONNX Prerequisites

To use this ONNX tutorial, you must have access to a Deep Learning AMI with Conda version 12 or later. For more information about how to get started with a Deep Learning AMI with Conda, see Deep Learning AMI with Conda (p. 6).

Important
These examples use functions that might require up to 8 GB of memory (or more). Be sure to choose an instance type with enough memory.

Launch a terminal session with your Deep Learning AMI with Conda to begin the following tutorial.

Convert a PyTorch Model to ONNX, then Load the Model into MXNet

First, activate the PyTorch environment:

```
$ source activate pytorch_p36
```

Create a new file with your text editor, and use the following program in a script to train a mock model in PyTorch, then export it to the ONNX format.

```
# Build a Mock Model in PyTorch with a convolution and a reduceMean layer
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.autograd import Variable
import torch.onnx as torch_onnx

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=(3,3), stride=1,
 padding=0, bias=False)

    def forward(self, inputs):
        x = self.conv(inputs)
        #x = x.view(x.size()[0], x.size()[1], -1)
        return torch.mean(x, dim=2)

# Use this an input trace to serialize the model
input_shape = (3, 100, 100)
model_onnx_path = "torch_model.onnx"
model = Model()
model.train(False)

# Export the model to an ONNX file
dummy_input = Variable(torch.randn(1, *input_shape))
output = torch_onnx.export(model,
                          dummy_input,
                          model_onnx_path,
                          verbose=False)
print("Export of torch_model.onnx complete!")
```

After you run this script, you will see the newly created .onnx file in the same directory. Now, switch to the MXNet Conda environment to load the model with MXNet.

Next, activate the MXNet environment:

```
$ source deactivate
$ source activate mxnet_p36
```

Create a new file with your text editor, and use the following program in a script to open ONNX format file in MXNet.

```
import mxnet as mx
from mxnet.contrib import onnx as onnx_mxnet
import numpy as np

# Import the ONNX model into MXNet's symbolic interface
sym, arg, aux = onnx_mxnet.import_model("torch_model.onnx")
print("Loaded torch_model.onnx!")
print(sym.get_internals())
```

After you run this script, MXNet will have loaded the model, and will print some basic model information.

ONNX Tutorials

- Apache MXNet to ONNX to CNTK Tutorial (p. 110)
- Chainer to ONNX to CNTK Tutorial (p. 111)
- Chainer to ONNX to MXNet Tutorial (p. 113)
- PyTorch to ONNX to MXNet Tutorial (p. 116)
- PyTorch to ONNX to CNTK Tutorial (p. 114)

Model Serving

The following are model serving options installed on the Deep Learning AMI with Conda. Click on one of the options to learn how to use it.

Topics

Model Server for Apache MXNet (MMS)

Model Server for Apache MXNet (MMS) is a flexible tool for serving deep learning models that have been exported from Apache MXNet (incubating) or exported to an Open Neural Network Exchange (ONNX) model format. MMS comes preinstalled with the DLAMI with Conda. This tutorial for MMS will demonstrate how to serve an image classification model.

Topics

Serve an Image Classification Model on MMS

This tutorial shows how to serve an image classification model with MMS. The model is provided via the MMS Model Zoo, and is automatically downloaded when you start MMS. Once the server is running, it listens for prediction requests. When you upload an image, in this case, an image of a kitten, the server returns a prediction of the top 5 matching classes out of the 1,000 classes that the model was trained on. More information on the models, how they were trained, and how to test them can be found in the MMS Model Zoo.

To serve an example image classification model on MMS

1. Connect to an Amazon Elastic Compute Cloud (Amazon EC2) instance of the Deep Learning AMI with Conda.
2. Activate an MXNet environment:
   - For MXNet and Keras 2 on Python 3 with CUDA 9.0 and MKL-DNN, run this command:

     ```
     $ source activate mxnet_p36
     ```
   - For MXNet and Keras 2 on Python 2 with CUDA 9.0 and MKL-DNN, run this command:

     ```
     $ source activate mxnet_p27
     ```
3. Run MMS with the following command. Adding `> /dev/null` will quiet the log output while you run further tests.

   ```
   $ mxnet-model-server --start > /dev/null
   ```

   MMS is now running on your host, and is listening for inference requests.
4. Next, use a curl command to administer MMS's management endpoints, and tell it what model you want it to serve.

   ```
   $ curl -X POST "http://localhost:8081/models?url=https%3A%2F%2Fs3.amazonaws.com
   %2Fmodel-server%2Fmodels%2Fsqueezenet_v1.1%2Fsqueezenet_v1.1.model"
   ```
5. MMS needs to know the number of workers you would like to use. For this test you can try 3.

```
$ curl -v -X PUT "http://localhost:8081/models/squeezenet_v1.1?min_worker=3"
```

6. Download an image of a kitten and send it to the MMS predict endpoint:

```
$ curl -O https://s3.amazonaws.com/model-server/inputs/kitten.jpg
$ curl -X POST http://127.0.0.1:8080/predictions/squeezenet_v1.1 -T kitten.jpg
```

   The predict endpoint returns a prediction in JSON similar to the following top five predictions, where the image has a 94% probability of containing an Egyptian cat, followed by a 5.5% chance it has a lynx or catamount:

```
{
 "prediction": [
  [{
    "class": "n02124075 Egyptian cat",
    "probability": 0.940
   },
   {
    "class": "n02127052 lynx, catamount",
    "probability": 0.055
   },
   {
    "class": "n02123045 tabby, tabby cat",
    "probability": 0.002
   },
   {
    "class": "n02123159 tiger cat",
    "probability": 0.0003
   },
   {
    "class": "n02123394 Persian cat",
    "probability": 0.0002
   }
  ]
 ]
}
```

7. Test out some more images, or if you have finished testing, stop the server:

```
$ mxnet-model-server --stop
```

This tutorial focuses on basic model serving. MMS also supports using Elastic Inference with model serving. For more information, see Model Serving with Amazon Elastic Inference

When you're ready to learn more about other MMS features, see the MMS documentation on GitHub.

Other Examples

MMS has a variety of examples that you can run on your DLAMI. You can view them on the MMS project repository.

More Info

For more MMS documentation, including how to set up MMS with Docker—or to take advantage of the latest MMS features, star the MMS project page on GitHub.

TensorFlow Serving

TensorFlow Serving is a flexible, high-performance serving system for machine learning models.

The `tensorflow-serving-api` is pre-installed with Deep Learning AMI with Conda! You will find an example scripts to train, export, and serve an MNIST model in `~/examples/tensorflow-serving/`.

To run any of these examples, first connect to your Deep Learning AMI with Conda and activate the TensorFlow environment.

```
$ source activate tensorflow_p37
```

Now change directories to the serving example scripts folder.

```
$ cd ~/examples/tensorflow-serving/
```

Serve a Pretrained Inception Model

The following is an example you can try for serving different models like Inception. As a general rule, you need a servable model and client scripts to be already downloaded to your DLAMI.

Serve and Test Inference with an Inception Model

1. Download the model.

   ```
   $ curl -O https://s3-us-west-2.amazonaws.com/tf-test-models/INCEPTION.zip
   ```

2. Untar the model.

   ```
   $ unzip INCEPTION.zip
   ```

3. Download a picture of a husky.

   ```
   $ curl -O https://upload.wikimedia.org/wikipedia/commons/b/b5/Siberian_Husky_bi-
   eyed_Flickr.jpg
   ```

4. Launch the server. Note, that for Amazon Linux, you must change the directory used for `model_base_path`, from `/home/ubuntu` to `/home/ec2-user`.

   ```
   $ tensorflow_model_server --model_name=INCEPTION --model_base_path=/home/ubuntu/
   examples/tensorflow-serving/INCEPTION/INCEPTION --port=9000
   ```

5. With the server running in the foreground, you need to launch another terminal session to continue. Open a new terminal and activate TensorFlow with `source activate tensorflow_p37`. Then use your preferred text editor to create a script that has the following content. Name it `inception_client.py`. This script will take an image filename as a parameter, and get a prediction result from the pre-trained model.

   ```
   from __future__ import print_function

   import grpc
   import tensorflow as tf
   import argparse

   from tensorflow_serving.apis import predict_pb2
   from tensorflow_serving.apis import prediction_service_pb2_grpc
   ```

```
parser = argparse.ArgumentParser(
    description='TF Serving Test',
    formatter_class=argparse.ArgumentDefaultsHelpFormatter
)
parser.add_argument('--server_address', default='localhost:9000',
                    help='Tenforflow Model Server Address')
parser.add_argument('--image', default='Siberian_Husky_bi-eyed_Flickr.jpg',
                    help='Path to the image')
args = parser.parse_args()


def main():
  channel = grpc.insecure_channel(args.server_address)
  stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)
  # Send request
  with open(args.image, 'rb') as f:
    # See prediction_service.proto for gRPC request/response details.
    request = predict_pb2.PredictRequest()
    request.model_spec.name = 'INCEPTION'
    request.model_spec.signature_name = 'predict_images'

    input_name = 'images'
    input_shape = [1]
    input_data = f.read()
    request.inputs[input_name].CopyFrom(
      tf.make_tensor_proto(input_data, shape=input_shape))

    result = stub.Predict(request, 10.0)  # 10 secs timeout
    print(result)

  print("Inception Client Passed")


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

6. Now run the script passing the server location and port and the husky photo's filename as the parameters.

```
$ python3 inception_client.py --server=localhost:9000 --image Siberian_Husky_bi-
eyed_Flickr.jpg
```

Train and Serve an MNIST Model

For this tutorial we will export a model then serve it with the `tensorflow_model_server` application. Finally, you can test the model server with an example client script.

Run the script that will train and export an MNIST model. As the script's only argument, you need to provide a folder location for it to save the model. For now we can just put it in `mnist_model`. The script will create the folder for you.

```
$ python mnist_saved_model.py /tmp/mnist_model
```

Be patient, as this script may take a while before providing any output. When the training is complete and the model is finally exported you should see the following:

```
Done training!
Exporting trained model to mnist_model/1
Done exporting!
```

Your next step is to run `tensorflow_model_server` to serve the exported model.

```
$ tensorflow_model_server --port=9000 --model_name=mnist --model_base_path=/tmp/mnist_model
```

A client script is provided for you to test the server.

To test it out, you will need to open a new terminal window.

```
$ python mnist_client.py --num_tests=1000 --server=localhost:9000
```

More Features and Examples

If you are interested in learning more about TensorFlow Serving, check out the TensorFlow website.

You can also use TensorFlow Serving with Amazon Elastic Inference. Check out the guide on how to Use Elastic Inference with TensorFlow Serving for more info.

TorchServe

TorchServe is a flexible tool for serving deep learning models that have been exported from PyTorch. TorchServe comes preinstalled with the Deep Learning AMI with Conda starting with v34.

For more information on using TorchServe, see Model Server for PyTorch Documentation.

Topics

Serve an Image Classification Model on TorchServe

This tutorial shows how to serve an image classification model with TorchServe. It uses a DenseNet-161 model provided by PyTorch. Once the server is running, it listens for prediction requests. When you upload an image, in this case, an image of a kitten, the server returns a prediction of the top 5 matching classes out of the classes that the model was trained on.

To serve an example image classification model on TorchServe

1. Connect to an Amazon Elastic Compute Cloud (Amazon EC2) instance with Deep Learning AMI with Conda v34 or later.
2. Activate the `pytorch_latest_p36` environment.

   ```
   source activate pytorch_latest_p36
   ```

3. Clone the TorchServe repository, then create a directory to store your models.

   ```
   git clone https://github.com/pytorch/serve.git
   mkdir model_store
   ```

4. Archive the model using the model archiver. The `extra-files` param uses a file from the `TorchServe` repo, so update the path if necessary. For more information about the model archiver, see Torch Model archiver for TorchServe.

   ```
   wget https://download.pytorch.org/models/densenet161-8d451a50.pth
   torch-model-archiver --model-name densenet161 --version 1.0 --model-file ./
   serve/examples/image_classifier/densenet_161/model.py --serialized-file
    densenet161-8d451a50.pth --export-path model_store --extra-files ./serve/examples/
   image_classifier/index_to_name.json --handler image_classifier
   ```

5. Run TorchServe to start an endpoint. Adding `> /dev/null` quiets the log output.

```
torchserve --start --ncs --model-store model_store --models densenet161.mar > /dev/null
```

6. Download an image of a kitten and send it to the TorchServe predict endpoint:

   ```
   curl -O https://s3.amazonaws.com/model-server/inputs/kitten.jpg
   curl http://127.0.0.1:8080/predictions/densenet161 -T kitten.jpg
   ```

   The predict endpoint returns a prediction in JSON similar to the following top five predictions, where the image has a 47% probability of containing an Egyptian cat, followed by a 46% chance it has a tabby cat.

   ```
   {
    "tiger_cat": 0.46933576464653015,
    "tabby": 0.463387668132782,
    "Egyptian_cat": 0.0645613968372345,
    "lynx": 0.0012828196631744504,
    "plastic_bag": 0.00023323058849200606
   }
   ```

7. When you finish testing, stop the server:

   ```
   torchserve --stop
   ```

Other Examples

TorchServe has a variety of examples that you can run on your DLAMI instance. You can view them on the TorchServe project repository examples page.

More Info

For more TorchServe documentation, including how to set up TorchServe with Docker and the latest TorchServe features, see the TorchServe project pageon GitHub.

Upgrading Your DLAMI

Here you will find information on upgrading your DLAMI and tips on updating software on your DLAMI.

Topics

Upgrading to a New DLAMI Version

DLAMI's system images are updated on a regular basis to take advantage of new deep learning framework releases, CUDA and other software updates, and performance tuning. If you have been using a DLAMI for some time and want to take advantage of an update, you would need to launch a new instance. You would also have to manually transfer any datasets, checkpoints, or other valuable data. Instead, you may use Amazon EBS to retain your data and attach it to a new DLAMI. In this way, you can upgrade often, while minimizing the time it takes to transition your data.

Note
When attaching and moving Amazon EBS volumes between DLAMIs, you must have both the DLAMIs and the new volume in the same Availability Zone.

1. Use the Amazon EC2console to create a new Amazon EBS volume. For detailed directions, see Creating an Amazon EBS Volume.
2. Attach your newly created Amazon EBS volume to your existing DLAMI. For detailed directions, see Attaching an Amazon EBS Volume.
3. Transfer your data, such as datasets, checkpoints, and configuration files.
4. Launch a DLAMI. For detailed directions, see Launching and Configuring a DLAMI (p. 12).
5. Detach the Amazon EBS volume from your old DLAMI. For detailed directions, see Detaching an Amazon EBS Volume.
6. Attach the Amazon EBS volume to your new DLAMI. Follow the instructions from the Step 2 to attach the volume.
7. After you verify that your data is available on your new DLAMI, stop and terminate your old DLAMI. For detailed clean-up instructions, see Clean Up (p. 15).

Tips for Software Updates

From time to time, you may want to manually update software on your DLAMI. It is generally recommended that you use `pip` to update Python packages. You should also use `pip` to update packages within a Conda environment on the Deep Learning AMI with Conda. Refer to the particular framework's or software's website for upgrading and installation instructions.

Note
We cannot guarantee that a package update will be successful. Attempting to update a package in an environment with incompatible dependencies can result in a failure. In such a case, you should contact the library maintainer to see if it is possible to update the package dependencies. Alternatively, you can attempt to modify the environment in such a way that allows the update. However, this modification will likely mean removing or updating existing packages, which means that we can no longer guarantee stability of this environment.

If you are interested in running the latest main branch of a particular package, activate the appropriate environment, then add `--pre` to the end of the `pip install --upgrade` command. For example:

```
source activate mxnet_p36
pip install --upgrade mxnet --pre
```

The AWS Deep Learning AMI comes with many Conda environments and many packages preinstalled. Due to the number of packages preinstalled, finding a set of packages that are guaranteed to be compatible is difficult. You may see a warning "The environment is inconsistent, please check the package plan carefully". DLAMI ensures that all the DLAMI-provided environments are correct, but cannot guarantee that any user installed packages will function correctly.

Security in AWS Deep Learning AMI

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to DLAMI, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using DLAMI. The following topics show you how to configure DLAMI to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your DLAMI resources.

For more information, see Security in Amazon EC2.

Topics

Data Protection in AWS Deep Learning AMI

The AWS shared responsibility model applies to data protection in AWS Deep Learning AMI. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM). That way each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We recommend TLS 1.2 or later.
- Set up API and user activity logging with AWS CloudTrail.

- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form fields such as a **Name** field. This includes when you work with DLAMI or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Identity and Access Management in AWS Deep Learning AMI

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use DLAMI resources. IAM is an AWS service that you can use with no additional charge.

For more information on Identity and Access Management, see Identity and Access Management for Amazon EC2.

Topics

Authenticating With Identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see Signing in to the AWS Management Console as an IAM user or root user in the *IAM User Guide*.

You must be *authenticated* (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email address or your IAM user name. You can access AWS programmatically using your root user or IAM users access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 signing process in the *AWS General Reference*.

Regardless of the authentication method that you use, you might also be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM Users and Groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing access keys for IAM users in the *IAM User Guide*. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to create an IAM user (instead of a role) in the *IAM User Guide*.

IAM Roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM roles in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated users and roles in the *IAM User Guide*.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.

  - **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see in the *Service Authorization Reference*.
  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) in the *IAM User Guide*.

Managing Access Using Policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. You can sign in as the root user or an IAM user, or you can assume an IAM role. When you then make a request, AWS evaluates the related identity-based or resource-based policies. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-Based Policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing between managed policies and inline policies in the *IAM User Guide*.

Resource-Based Policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access Control Lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other Policy Types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see How SCPs work in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple Policy Types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

IAM with Amazon EMR

You can use AWS Identity and Access Management with Amazon EMR to define users, AWS resources, groups, roles, and policies. You can also control which AWS services these users and roles can access.

For more information on using IAM with Amazon EMR, see AWS Identity and Access Management for Amazon EMR.

Logging and Monitoring in AWS Deep Learning AMI

Your AWS Deep Learning AMI instance comes with several GPU monitoring tools including a utility that reports GPU usage statistics to Amazon CloudWatch. For more information, see GPU Monitoring and Optimization and Monitoring Amazon EC2.

Usage Tracking

The following AWS Deep Learning AMI operating system distributions include code that allows AWS to collect instance type, instance ID, DLAMI type, and OS information. No information on the commands used within the DLAMI is collected or retained. No other information about the DLAMI is collected or retained.

- Ubuntu 16.04
- Ubuntu 18.04
- Ubuntu 20.04
- Amazon Linux 2

To opt out of usage tracking for your DLAMI, add a tag to your Amazon EC2 instance during launch. The tag should use the key `OPT_OUT_TRACKING` with the associated value set to `true`. For more information, see Tag your Amazon EC2 resources.

Compliance Validation for AWS Deep Learning AMI

Third-party auditors assess the security and compliance of AWS Deep Learning AMI as part of multiple AWS compliance programs. For information on the supported compliance programs, see Compliance Validation for Amazon EC2.

For a list of AWS services in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using DLAMI is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.

- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience in AWS Deep Learning AMI

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

For information on features to help support your data resiliency and backup needs, see Resilience in Amazon EC2.

Infrastructure Security in AWS Deep Learning AMI

The infrastructure security of AWS Deep Learning AMI is backed by Amazon EC2. For more information, see Infrastructure Security in Amazon EC2.

Related Information

Topics

Forums

- Forum: AWS Deep Learning AMIs

Related Blog Posts

- Updated List of Articles Related to Deep Learning AMIs
- Launch a AWS Deep Learning AMI (in 10 minutes)
- Faster Training with Optimized TensorFlow 1.6 on Amazon EC2 C5 and P3 Instances
- New AWS Deep Learning AMIs for Machine Learning Practitioners
- New Training Courses Available: Introduction to Machine Learning & Deep Learning on AWS
- Journey into Deep Learning with AWS

FAQ

- **Q.** How do I keep track of product announcements related to DLAMI?

  Here are two suggestions for this:
  - Bookmark this blog category, "AWS Deep Learning AMIs" found here: Updated List of Articles Related to Deep Learning AMIs.
  - "Watch" the Forum: AWS Deep Learning AMIs
- **Q.** Are the NVIDIA drivers and CUDA installed?

  Yes. Some DLAMIs have different versions. The Deep Learning AMI with Conda (p. 6) has the most recent versions of any DLAMI. This is covered in more detail in CUDA Installations and Framework Bindings (p. 5). You can also refer to the specific AMI's detail page on the marketplace to confirm what is installed.
- **Q.** Is cuDNN installed?

  Yes.
- **Q.** How do I see that the GPUs are detected and their current status?

  Run `nvidia-smi`. This will show one or more GPUs, depending on the instance type, along with their current memory consumption.

- **Q.** Are virtual environments set up for me?

  Yes, but only on the Deep Learning AMI with Conda (p. 6).
- **Q.** What version of Python is installed?

  Each DLAMI has both Python 2 and 3. The Deep Learning AMI with Conda (p. 6) have environments for both versions for each framework.
- **Q.** Is Keras installed?

  This depends on the AMI. The Deep Learning AMI with Conda (p. 6) has Keras available as a front end for each framework. The version of Keras depends on the framework's support for it.
- **Q.** Is it free?

  All of the DLAMIs are free. However, depending on the instance type you choose, the instance may not be free. See Pricing for the DLAMI (p. 8) for more info.
- **Q.** I'm getting CUDA errors or GPU-related messages from my framework. What's wrong?

  Check what instance type you used. It needs to have a GPU for many examples and tutorials to work. If running `nvidia-smi` shows no GPU, then you need to spin up another DLAMI using an instance with one or more GPUs. See Selecting the Instance Type for DLAMI (p. 7) for more info.
- **Q.** Can I use Docker?

  Docker has been pre-installed since version 14 of the Deep Learning AMI with Conda. Note that you will want to use nvidia-docker on GPU instances to make use of the GPU.
- **Q.** What regions are Linux DLAMIs available in?

Region	Code
US East (Ohio)	us-east-2
US East (N. Virginia)	us-east-1
GovCloud	us-gov-west-1
US West (N. California)	us-west-1
US West (Oregon)	us-west-2
Beijing (China)	cn-north-1
Ningxia (China)	cn-northwest-1
Asia Pacific (Mumbai)	ap-south-1
Asia Pacific (Seoul)	ap-northeast-2
Asia Pacific (Singapore)	ap-southeast-1
Asia Pacific (Sydney)	ap-southeast-2
Asia Pacific (Tokyo)	ap-northeast-1
Canada (Central)	ca-central-1
EU (Frankfurt)	eu-central-1
EU (Ireland)	eu-west-1
EU (London)	eu-west-2

Region	Code
EU (Paris)	eu-west-3
SA (Sao Paulo)	sa-east-1

- **Q.** What regions are Windows DLAMIs available in?

Region	Code
US East (Ohio)	us-east-2
US East (N. Virginia)	us-east-1
GovCloud	us-gov-west-1
US West (N. California)	us-west-1
US West (Oregon)	us-west-2
Beijing (China)	cn-north-1
Asia Pacific (Mumbai)	ap-south-1
Asia Pacific (Seoul)	ap-northeast-2
Asia Pacific (Singapore)	ap-southeast-1
Asia Pacific (Sydney)	ap-southeast-2
Asia Pacific (Tokyo)	ap-northeast-1
Canada (Central)	ca-central-1
EU (Frankfurt)	eu-central-1
EU (Ireland)	eu-west-1
EU (London)	eu-west-2
EU (Paris)	eu-west-3
SA (Sao Paulo)	sa-east-1

Release Notes for DLAMI

Note
AWS Deep Learning AMIs have a nightly release cadence for security patches. These incremental security patches are not included in official release notes.

For information on related hardware, frameworks, and ID retrieval, see the AWS Deep Learning AMI Catalog. For the current and historic DLAMI release notes, see:

Single-framework DLAMI

TensorFlow-specific AMI

- AWS Deep Learning AMI GPU TensorFlow 2.9 (Amazon Linux 2)
- AWS Deep Learning AMI GPU TensorFlow 2.9 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU TensorFlow 2.8 (Amazon Linux 2)
- AWS Deep Learning AMI GPU TensorFlow 2.8 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU TensorFlow 2.7 (Amazon Linux 2)
- AWS Deep Learning AMI GPU TensorFlow 2.7 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU TensorFlow 2.6 (Amazon Linux 2)
- AWS Deep Learning AMI GPU TensorFlow 2.6 (Ubuntu 20.04)
- AWS Deep Learning AMI Graviton GPU TensorFlow 2.6 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU TensorFlow 2.6 (Ubuntu 18.04)
- AWS Deep Learning AMI GPU TensorFlow 2.5 (Amazon Linux 2)
- AWS Deep Learning AMI GPU TensorFlow 2.5 (Ubuntu 20.04)

PyTorch-specific AMI

- AWS Deep Learning AMI GPU PyTorch 1.11 (Amazon Linux 2)
- AWS Deep Learning AMI GPU PyTorch 1.11 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU PyTorch 1.10 (Amazon Linux 2)
- AWS Deep Learning AMI GPU PyTorch 1.10 (Ubuntu 20.04)
- AWS Deep Learning AMI Graviton GPU PyTorch 1.10 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU PyTorch 1.10 (Ubuntu 18.04)
- AWS Deep Learning AMI GPU PyTorch 1.9 (Amazon Linux 2)
- AWS Deep Learning AMI GPU PyTorch 1.9 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU PyTorch 1.9 (Ubuntu 18.04)

MXNet-specific AMI

- AWS Deep Learning AMI GPU MXNet 1.9 (Amazon Linux 2)
- AWS Deep Learning AMI GPU MXNet 1.9 (Ubuntu 20.04)

Multi-framework DLAMI

- AWS Deep Learning AMI (Amazon Linux 2)
- AWS Deep Learning AMI (Amazon Linux)
- AWS Deep Learning AMI (Ubuntu 18.04)
- AWS Deep Learning AMI (Ubuntu 16.04)

GPU DLAMI

CUDA 11.1

- AWS Deep Learning AMI GPU CUDA 11.1 (Amazon Linux 2)
- AWS Deep Learning AMI GPU CUDA 11.1 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU CUDA 11.1 (Ubuntu 18.04)

CUDA 11.2

- AWS Deep Learning AMI GPU CUDA 11.2 (Amazon Linux 2)
- AWS Deep Learning AMI GPU CUDA 11.2 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU CUDA 11.2 (Ubuntu 18.04)

CUDA 11.3

- AWS Deep Learning AMI GPU CUDA 11.3 (Amazon Linux 2)
- AWS Deep Learning AMI GPU CUDA 11.3 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU CUDA 11.3 (Ubuntu 18.04)

CUDA 11.4

- AWS Deep Learning AMI GPU CUDA 11.4 (Amazon Linux 2)
- AWS Deep Learning AMI GPU CUDA 11.4 (Ubuntu 20.04)
- AWS Deep Learning AMI Graviton GPU CUDA 11.4 (Ubuntu 20.04)
- AWS Deep Learning AMI GPU CUDA 11.4 (Ubuntu 18.04)

CUDA 11.5

- AWS Deep Learning AMI GPU CUDA 11.5 (Amazon Linux 2)
- AWS Deep Learning AMI GPU CUDA 11.5 (Ubuntu 20.04)

Habana DLAMI

TensorFlow-specific AMI

- AWS Deep Learning AMI Habana TensorFlow SynapseAI (Amazon Linux 2)
- AWS Deep Learning AMI Habana TensorFlow SynapseAI (Ubuntu 20.04)
- AWS Deep Learning AMI Habana TensorFlow SynapseAI (Ubuntu 18.04)

PyTorch-specific AMI

- AWS Deep Learning AMI Habana PyTorch SynapseAI (Amazon Linux 2)
- AWS Deep Learning AMI Habana PyTorch SynapseAI (Ubuntu 20.04)
- AWS Deep Learning AMI Habana PyTorch SynapseAI (Ubuntu 18.04)

Base DLAMI

- AWS Deep Learning Base AMI (Amazon Linux 2)
- AWS Deep Learning Base AMI (Amazon Linux)
- AWS Deep Learning Base AMI GPU CUDA 11 (Ubuntu 20.04)
- AWS Deep Learning Base AMI (Ubuntu 18.04)
- AWS Deep Learning Base AMI (Ubuntu 16.04)

DLAMI deprecation notices

The following table lists information on deprecated features in the AWS Deep Learning AMI.

Deprecated Feature	Deprecation Date	Deprecation Notice
Ubuntu 16.04	10/07/2021	Ubuntu Linux 16.04 LTS reached the end of its five-year LTS window on April 30, 2021 and is no longer supported by its vendor. There are no longer updates to the Deep Learning Base AMI (Ubuntu 16.04) in new releases as of October 2021. Previous releases will continue to be available.
Amazon Linux	10/07/2021	Amazon Linux is end-of-life as of December 2020. There are no longer updates to the Deep Learning AMI (Amazon Linux) in new releases as of October 2021. Previous releases of the Deep Learning AMI (Amazon Linux) will continue to be available.
Chainer	07/01/2020	Chainer has announced the end of major releases as of December, 2019. Consequently, we will no longer include Chainer Conda environments on the DLAMI starting July 2020. Previous releases of the DLAMI that contain these environments will continue to be available. We will provide updates to these environments only if there are security fixes published by the open source community for these frameworks.
Python 3.6	06/15/2020	Due to customer requests, we are moving to Python 3.7 for new TF/MX/PT releases.
Python 2	01/01/2020	The Python open source community has officially ended support for Python 2. The TensorFlow, PyTorch, and MXNet communities

Deprecated Feature	Deprecation Date	Deprecation Notice
		have also announced that TensorFlow 1.15, TensorFlow 2.1, PyTorch 1.4, and MXNet 1.6.0 releases will be the last ones supporting Python 2.

Document History for AWS Deep Learning AMI Developer Guide

update-history-change	update-history-description	update-history-date
Graviton DLAMI (p. 97)	The AWS Deep Learning AMI now supports images on Arm processor-based Graviton GPUs.	November 29, 2021
Habana DLAMI (p. 104)	The AWS Deep Learning AMI now supports Habana Gaudi hardware and the Habana SynapseAI SDK.	October 25, 2021
TensorFlow 2 (p. 37)	The Deep Learning AMI with Conda now comes with TensorFlow 2 with CUDA 10.	December 3, 2019
AWS Inferentia (p. 82)	The Deep Learning AMI now supports AWS Inferentia hardware and the AWS Neuron SDK.	December 3, 2019
Using TensorFlow Serving with an Inception Model (p. 120)	An example for using inference with an Inception model was added for TensorFlow Serving, for both with and without Elastic Inference.	November 28, 2018
Training with 256 GPUs with TensorFlow and Horovod (p. 55)	The TensorFlow with Horovod tutorial was updated to add an example of multiple-node training.	November 28, 2018
Elastic Inference (p. 12)	Elastic inference prerequisites and related info was added to the setup guide.	November 28, 2018
MMS v1.0 released on the DLAMI. (p. 118)	The MMS tutorial was updated to use the new model archive format (.mar) and demonstrates the new start and stop features.	November 15, 2018
Installing TensorFlow from a Nightly Build (p. 36)	A tutorial was added that covers how you can uninstall TensorFlow, then install a nightly build of TensorFlow on your Deep Learning AMI with Conda.	October 16, 2018
Installing CNTK from a Nightly Build (p. 31)	A tutorial was added that covers how you can uninstall CNTK,	October 16, 2018

	then install a nightly build of CNTK on your Deep Learning AMI with Conda.	
Installing Apache MXNet (Incubating) from a Nightly Build (p. 28)	A tutorial was added that covers how you can uninstall MXNet, then install a nightly build of MXNet on your Deep Learning AMI with Conda.	October 16, 2018
Installing PyTorch from a Nightly Build (p. 34)	A tutorial was added that covers how you can uninstall PyTorch, then install a nightly build of PyTorch on your Deep Learning AMI with Conda.	September 25, 2018
Docker is now pre-installed on your DLAMI (p. 133)	Since v14 of the Deep Learning AMI with Conda, Docker and NVIDIA's version of Docker for GPUs has been pre-installed.	September 25, 2018
TensorBoard Tutorial (p. 44)	Example was moved to ~/examples/tensorboard. Tutorial paths updated.	July 23, 2018
MXBoard Tutorial (p. 41)	A tutorial on how to use MXBoard for visualization of MXNet models was added.	July 23, 2018
Distributed Training Tutorials (p. 45)	A tutorial on how to use Keras-MXNet for multi-GPU training was added. Chainer's tutorial was updated to for v4.2.0.	July 23, 2018
Conda Tutorial (p. 22)	The example MOTD was updated to reflect a more recent release.	July 23, 2018
Chainer Tutorial (p. 46)	The tutorial was updated to use the latest examples from Chainer's source.	July 23, 2018

Earlier Updates:

The following table describes important changes in each release of the AWS Deep Learning AMI before July, 2018.

Change	Description	Date
TensorFlow with Horovod	Added a tutorial for training ImageNet with TensorFlow and Horovod.	June 6, 2018
Upgrading guide	Added the upgrading guide.	May 15, 2018
New regions and new 10 minute tutorial	New regions added: US West (N. California), South America, Canada (Central), EU (London), and EU (Paris). Also, the first release of a 10-minute tutorial	April 26, 2018

Change	Description	Date
	titled: "Getting Started with Deep Learning AMI".	
Chainer tutorial	A tutorial for using Chainer in multi-GPU, single GPU, and CPU modes was added. CUDA integration was upgraded from CUDA 8 to CUDA 9 for several frameworks.	February 28, 2018
Linux AMIs v3.0, plus introduction of MXNet Model Server, TensorFlow Serving, and TensorBoard	Added tutorials for Conda AMIs with new model and visualization serving capabilities using MXNet Model Server v0.1.5, TensorFlow Serving v1.4.0, and TensorBoard v0.4.0. AMI and framework CUDA capabilities described in Conda and CUDA overviews. Latest release notes moved to https://aws.amazon.com/releasenotes/	January 25, 2018
Linux AMIs v2.0	Base, Source, and Conda AMIs updated with NCCL 2.1. Source and Conda AMIs updated with MXNet v1.0, PyTorch 0.3.0, and Keras 2.0.9.	December 11, 2017
Two Windows AMI options added	Windows 2012 R2 and 2016 AMIs released: added to AMI selection guide and added to release notes.	November 30, 2017
Initial documentation release	Detailed description of change with link to topic/section that was changed.	November 15, 2017

AWS glossary

For the latest AWS terminology, see the AWS glossary in the *AWS General Reference*.