开发人员指南

深度学习 AMI

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

深度学习 AMI: 开发人员指南

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商标和商业外观不得用于任何非 Amazon 的商品或服务，也不得以任何可能引起客户混淆、贬低或诋毁 Amazon 的方式使用。所有非 Amazon 拥有的其他商标均为各自所有者的财产，这些所有者可能附属于 Amazon、与 Amazon 有关联或由 Amazon 赞助，也可能不是如此。

Table of Contents

AWS Deep Learning AMI 是什么？ ... 1
关于本指南 ... 1
先决条件 ... 1
用例 ... 1
特征 ... 2
预装框架 ... 2
预装 GPU 软件 ... 3
模型处理和可视化 ... 3
入门 ... 4
如何开始使用 DLAMI ... 4
DLAMI 选择 ... 4
CUDA 安装和框架绑定 ... 5
基础 ... 6
Conda ... 6
架构 ... 8
OS ... 8
实例选择 ... 8
定价 ... 10
区域可用性 ... 10
GPU ... 10
CPU ... 11
Inferentia ... 12
Trainium ... 12
Habana ... 13
框架支持策略 ... 14
支持的框架 ... 14
常见问题 ... 14
哪些框架版本会获得安全补丁？ ... 15
发布新框架版本时，AWS 会发布哪些映像？ ... 15
哪些图片有新增 SageMaker/AWS功能？ ... 15
“支持的框架”表中是如何定义当前版本的？ ... 15
如果我运行的版本不在“支持的框架”表中，该怎么办？ ... 15
DLAMIs 是否支持以前的版本？ TensorFlow ... 16
如何找到支持的框架版本的最新补丁映像？ ... 16

- 多长时间发布一次新映像？ 16
- 运行工作负载时，能在我的实例上以替代方式安装补丁吗？ 16
- 如果有新的补丁或更新的框架版本可用，会发生什么呢？ 16
- 是否可在不更改框架版本的情况下更新依赖项？ 16
- 对我的框架版本的主动支持何时结束？ 16
- 对于框架版本不再主动维护的映像，会为其安装补丁吗？ 18
- 如何使用旧框架版本？ 18
- 如何保持框架及其版本 up-to-date 的支持变更？ 18
- 是否需要商业许可证才能使用 Anaconda 存储库？ 18

启动 DLAMI 19
- 步骤 1：启动 DLAMI 19
  - 检索 DLAMI ID 20
  - 从 Amazon EC2 控制台启动 21
- 步骤 2：连接到 DLAMI 22
- 步骤 3：测试 DLAMI 22
- 步骤 4：管理 DLAMI 实例 22
- 清除 23
- Jupyter 设置 23
  - 保护 Jupyter 24
  - 启动服务器 25
  - 配置客户端 25
  - 登录 Jupyter Notebook 服务器 27

使用 DLAMI 29
- Conda DLAMI 29
  - 带 Conda 的深度学习 AMI 的简介 29
  - 登录到你的 DLAMI 30
  - 启动 TensorFlow 环境 31
  - 切换到 PyTorch Python 3 环境 32
  - 切换到 MXNet Python 3 环境 33
  - 删除环境 34
- 基础 DLAMI 34
  - 使用深度学习基础 AMI 34
  - 配置 CUDA 版本 34
- Jupyter 笔记本 34
  - 导航已安装的教程 35
  - 通过 Jupyter 切换环境 36

- 教程 36
  - 10 分钟教程 37
  - 激活框架 37
  - 调试和可视化 56
  - 分布式训练 60
  - Elastic Fabric Adapter 82
  - GPU 监控和优化 96
  - AWS 推论 105
  - Graviton DLAMI 126
  - Habana DLAMI 136
  - 推理 138
  - 将框架用于 ONNX 143
  - 模型处理 155
- 升级 DLAMI 164
  - DLAMI 升级 164
  - 软件更新 165
- 安全性 166
  - 数据保护 166
  - Identity and Access Management 167
    - 使用身份进行身份验证 168
    - 使用策略管理访问 170
    - IAM 与 Amazon EMR 结合使用 172
  - 日志记录和监控 172
    - 使用情况跟踪 172
  - 合规性验证 172
  - 韧性 173
  - 基础设施安全性 173
- DLAMI 的重要更改 174
  - 常见问题 174
    - 发生了什么变化？ 174
    - 为什么需要进行此更改？ 175
    - 哪些 DLAMI 受此更改的影响？ 175
    - 这对你意味着什么？ 176
    - 你应该什么时候开始使用新的 dLAMis？ 177
    - 新的 dLaMis 会不会在功能上有所损失？ 177
    - 那么 DLC 呢？ 177

相关信息 178
论坛 178
博客 178
常见问题 178
DLAMI 的发布说明 182
..... 182
基础 DLAMI 182
单框架 DLAMI 182
多框架 DLAMI 183
DLAMI 弃用通知 184
文档历史记录 186
AWS 术语表 190
..... cxci

AWS Deep Learning AMI 是什么？

欢迎阅读 AWS Deep Learning AMI 用户指南。

AWS Deep Learning AMI (DLAMI) 是在云中进行深度学习的一站式商店。此自定义计算机实例可用于大多数 Amazon EC2 区域中的各种实例类型，从仅包含 CPU 的小型实例到最新的高性能多 GPU 实例。它预配置了 NVIDIA CUDA 和 NVIDIA cuDNN，以及最流行的深度学习框架的最新版本。

关于本指南

本指南将帮助您启动并使用 DLAMI。它介绍了用于培训和推导的深度学习的几种常见使用案例。本指南还介绍了如何针对您的用途选择合适的 AMI 和您喜欢的实例类型。DLAMI 针对每个框架提供多个教程。它还提供有关分布式训练、调试、使用 AWS Inferentia 和其他关键概念的教程。您将找到关于如何配置 Jupyter 以在浏览器中运行教程的说明。

先决条件

您应该熟悉命令行工具和基本 Python 才能成功运行 DLAMI。框架自身提供有关如何使用每个框架的教程，但是本指南可以向您展示如何激活每个框架并找到适当的入门教程。

DLAMI 用例

了解深度学习：DLAMI 是学习或教授机器学习和深度学习框架的理想选择。它不需要对每个框架的安装进行故障排除，并在同一台计算机上运行框架。DLAMI 附带 Jupyter 笔记本，可以轻松地向不熟悉机器学习和深度学习的人运行框架提供的教程。

应用程序开发：如果您是应用程序开发人员，并且有兴趣使用深度学习来使您的应用程序利用 AI 方面的最新进步，则 DLAMI 是理想的测试平台。每个框架都附带了关于如何开始使用深度学习的教程，其中许多教程都有模型动物园，可以让您轻松试用深度学习，您不需要自己创建神经网络或进行模型训练。一些示例向您展示如何在几分钟内构建映像检测应用程序，或如何为您自己的聊天自动程序构建语音识别应用程序。

机器学习和数据分析：如果您是数据科学家或有兴趣通过深度学习处理数据，您会发现许多框架都支持 R 和 Spark。您将找到关于进行简单回归的教程，以及为个性化和预测系统构建可扩展的数据处理系统的教程。

研究：如果您是研究人员，并且希望试用新框架、测试新模型或训练新模型，DLAMI 和 AWS 的扩展功能可以减少安装和管理多个训练节点的繁琐工作。

Note

虽然您的最初选择可能是将实例类型升级到具有多个 GPU（最多 8 个）的更大实例，但您也可以通过创建 DLAMI 实例集群来进行水平扩展。有关集群构建的更多信息，请参阅 相关信息。

DLAMI 的功能

预装框架

目前有两种主要的 DLAMI 版本，均包含与操作系统 (OS) 和软件版本相关的其他变体：

- 带 Conda 的深度学习 AMI — 在单独的 Python 环境中使用 conda 程序包单独安装的框架
- 深度学习基础 AMI — 不安装任何框架；只有 NVIDIA CUDA 和其他依赖项

带 Conda 的深度学习 AMI 使用 conda 环境来隔离每个框架，以便您可以随意切换框架，而不用担心其依赖项发生冲突。

这是带 Conda 的深度学习 AMI 所支持框架的完整列表：

- Apache MXNet（孵化版）
- PyTorch
- TensorFlow 2

Note

从 v28 版本开始，我们将不再在 AWS Deep Learning AMI 中包含 CNTK、Caffe、Caffe2、Theano、Chainer 或 Keras Conda 环境。包含这些环境的先前版本的 AWS Deep Learning AMI 将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

预装 GPU 软件

即使您使用仅包含 CPU 的实例，DLAMI 也将具有 NVIDIA CUDA 和 NVIDIA cuDNN。无论实例类型是什么，安装的软件都是相同的。请记住，GPU 特定工具只适用于至少包含一个 GPU 的实例。选择 DLAMI 的实例类型中包含有关此内容的更多信息。

有关 CUDA 安装的更多信息，请参阅 CUDA 安装和框架绑定。

模型处理和可视化

带 Conda 的深度学习 AMI 预装有两种模型服务器，一种用于 MXNet，另一种用于 TensorFlow，还预装有 TensorBoard，用于模型可视化。

- 适用于 Apache MXNet 的模型服务器 (MMS)
- TensorFlow 上菜
- TensorBoard

入门

如何开始使用 DLAMI

本指南包括了关于选择适合您的 DLAMI、选择适合您的使用案例和预算的实例类型的技巧，以及介绍您可能感兴趣的自定义设置的 相关信息。

如果您刚开始使用 AWS 或 Amazon EC2，请先从 带 Conda 的深度学习 AMI 开始。如果您熟悉 Amazon EC2 和其他 AWS 服务（如 Amazon EMR、Amazon EFS 或 Amazon S3），并且有兴趣集成这些服务以用于需要分布式训练或推理的项目，则在 相关信息 中查看是否有符合您的使用案例的服务。

我们建议您查看选择 DLAMI，以了解最适合您的应用程序的实例类型。

另一个选项是这个快速教程：启动 AWS Deep Learning AMI（10 分钟内）。

下一个步骤

选择 DLAMI

选择 DLAMI

我们提供一系列 DLAMI 选项。为了帮助您为自己的使用案例选择正确的 DLAMI，我们按开发映像的硬件类型或功能对映像进行分组。我们的顶层分组是：

- DLAMI 类型：CUDA、基础、单框架、多框架 (Conda DLAMI)
- 计算架构：基于 x86 与基于 Arm 的 AWS Graviton
- 处理器类型：GPU、CPU 、Inferentia、Habana
- SDK：CUDA、AWS Neuron、SynapsesAI
- 操作系统：Amazon Linux、Ubuntu

本指南中的其他主题有助于进一步给您提供信息和进一步让您了解详情。

主题

- CUDA 安装和框架绑定
- 深度学习基础 AMI

- 带 Conda 的深度学习 AMI
- DLAMI CPU 架构选项
- DLAMI 操作系统选项

后续步骤

带 Conda 的深度学习 AMI

CUDA 安装和框架绑定

虽然深度学习都非常先进，但每个框架都提供“稳定”版本。这些稳定版本可能不适用于最新的 CUDA 或 cuDNN 实施和功能。您的用例和所需功能可以帮助您选择框架。如果您不确定，就请使用最新的带 Conda 的深度学习 AMI。它具有适用于所有框架的官方 `pip` 二进制文件（包含 CUDA 10），使用每个框架都支持的任何最新版本。如果您需要最新版本，并要自定义深度学习环境，就请使用深度学习基础 AMI。

如需进一步指导，请在 稳定版本与候选版本 上查看我们的指南。

选择带 CUDA 的 DLAMI

深度学习基础 AMI 具有所有可用的 CUDA 11 系列，包括 11.0、11.1 和 11.2。

带 Conda 的深度学习 AMI 具有所有可用的 CUDA 11 系列，包括 11.0、11.1 和 11.2。

> **Note**
>
> 从 v28 版本开始，我们将不再在 AWS Deep Learning AMI 中包含 CNTK、Caffe、Caffe2、Theano、Chainer 或 Keras Conda 环境。包含这些环境的先前版本的 AWS Deep Learning AMI 继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才为这些环境提供更新。

有关具体框架版本号，请参阅 DLAMI 的发布说明。

选择这个 DLAMI 类型，或通过后续步骤选项了解有关其他 DLAMI 的更多信息。

选择一个 CUDA 版本并在附录中查看具有该版本的完整 DLAMI 列表，或者使用后续步骤选项来了解不同 DLAMI 的更多信息。

后续步骤

深度学习基础 AMI

相关主题

- 有关在 CUDA 版本之间切换的说明，请参阅使用深度学习基础 AMI教程。

深度学习基础 AMI

深度学习基础 AMI 就像一张用于深度学习的空白画布。它包含您需要的一切，直到安装特定的框架，并具有您选择的 CUDA 版本。

为什么要选择基础 DLAMI

该 AMI 团队对于那些想要分享深度学习项目和建立最新版本的项目贡献者非常有用。适用于那些希望推出自己环境且有信心安装和使用最新 NVIDIA 软件的人员，从而他们可以专注于选择要安装的框架和版本。

选择这个 DLAMI 类型，或通过后续步骤选项了解有关其他 DLAMI 的更多信息。

后续步骤

使用 Conda 的 DLAMI

相关主题

- 使用深度学习基础 AMI

带 Conda 的深度学习 AMI

Conda DLAMI 使用 conda 虚拟环境。这些环境配置为单独安装不同的框架，并简化框架之间的切换。这对了解和体验 DLAMI 必须提供的所有框架很有好处。大多数用户都会发现新的带 Conda 的深度学习 AMI 非常适合他们。

这些 AMI 是主要的 DLAMI。它们将经常使用框架中的最新版本进行更新，并拥有最新的 GPU 驱动程序和软件。在大多数文档中，它们通常被称为 AWS Deep Learning AMI。

- Ubuntu 18.04 DLAMI 具有以下框架：Apache MXNet（孵化版）、PyTorch 和 TensorFlow 2。
- Amazon Linux 2 DLAMI 具有以下框架：Apache MXNet（孵化版）、PyTorch 和 TensorFlow 2。

> **Note**
>
> 从 v28 版本开始，我们将不再在 AWS Deep Learning AMI 中包含 CNTK、Caffe、Caffe2、Theano、Chainer 和 Keras Conda 环境。包含这些环境的先前版本的 AWS Deep Learning AMI 继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才为这些环境提供更新。

稳定版本与候选版本

Conda AMI 使用每个框架中最新正式版本的优化二进制文件。不会有候选版本和实验性功能。优化取决于框架对 Intel 的 MKL DNN 等加速技术的支持，这些技术可加快在 C5 和 C4 CPU 实例类型上的训练和推理速度。二进制文件也将被编译以支持高级 Intel 指令集，包括但不限于 AVX、AVX-2、SSE4.1 和 SSE4.2。这些指令将加快 Intel CPU 架构上向量和浮点运算的速度。此外，对于 GPU 实例类型，使用最新官方版本支持的任何版本来更新 CUDA 和 cuDNN。

带 Conda 的深度学习 AMI 会在框架首次激活时自动为您的 Amazon EC2 实例安装框架的最优化版本。有关更多信息，请参阅 使用带 Conda 的深度学习 AMI。

如果要使用自定义或优化的版本选项从源安装，深度学习基础 AMI 可能是您更好的选择。

Python 2 弃用

Python 开源社区已于 2020 年 1 月 1 日正式结束对 Python 2 的支持。TensorFlow 和 PyTorch 社区已经宣布：TensorFlow 2.1 和 PyTorch 1.4 版本是支持 Python 2 的最后版本。包含 Python 2 Conda 环境的 DLAMI 先前版本（v26、v25 等）继续可用。但是，只有在开源社区针对先前发布的 DLAMI 版本发布了安全修补程序时，我们才会在这些版本上提供关于 Python 2 Conda 环境的更新。包含最新版本 TensorFlow 和 PyTorch 框架的 DLAMI 版本不包含 Python 2 Conda 环境。

CUDA 支持

具体 CUDA 版本号可以在 GPU DLAMI 发布说明中找到。

后续步骤

DLAMI CPU 架构选项

相关主题

- 有关使用带 Conda 的深度学习 AMI 的教程，请参阅 使用带 Conda 的深度学习 AMI 教程。

DLAMI CPU 架构选项

AWS Deep Learning AMI 附带基于 x86 或基于 Arm 的 AWS Graviton2 CPU 架构。

选择一款 Graviton GPU DLAMI 来使用基于 Arm 的 CPU 架构。所有其他 GPU DLAMI 目前都基于 x86。

- AWS 深度学习 AMI Graviton GPU CUDA 11.4 (Ubuntu 20.04)
- AWS 深度学习 AMI Graviton GPU TensorFlow 2.6 (Ubuntu 20.04)
- AWS 深度学习 AMI Graviton GPU PyTorch 1.10 (Ubuntu 20.04)

有关 Gravon GPU DLAMI 入门的信息，请参阅 Graviton DLAMI。有关可用实例类型的更多详细信息，请参阅 选择 DLAMI 的实例类型。

后续步骤

DLAMI 操作系统选项

DLAMI 操作系统选项

DLAMI 在以下操作系统中提供。

- Amazon Linux 2
- Ubuntu 20.04
- Ubuntu 18.04

可以在已弃用的 DLAMI 上使用旧版本的操作系统。有关 DLAMI 弃用的更多信息，请参阅 DLAMI 的弃用

在选择 DLAMI 之前，请评估您需要的实例类型并确定您的 AWS 区域。

后续步骤

选择 DLAMI 的实例类型

选择 DLAMI 的实例类型

有关与特定 DLAMI 兼容的推荐 Amazon EC2 实例系列，请参阅 AWS Deep Learning AMI 目录。

更一般情况下，在为 DLAMI 选择实例类型时，请考虑以下几点。

- 如果您不熟悉深度学习，那么具有单个 GPU 的实例可能适合您的需求。
- 如果您注重预算，则可以使用仅含 CPU 的实例。
- 如果您希望优化深度学习模型推理的高性能和成本效益，则可以使用带有 AWS Inferentia 芯片的实例。
- 如果您想优化深度学习模型训练的高性能和成本效益，则可以使用带有 Habana 加速器的实例。
- 如果您正在寻找具有基于 Arm 的 CPU 架构的高性能 GPU 实例，则可以使用 G5g 实例类型。
- 如果您有兴趣运行预训练模型进行推理和预测，则可以将 Amazon Elastic Inference 附加到您的 Amazon EC2 实例。Amazon Elastic Inference 允许您使用只有一部分 GPU 的加速器。
- 对于高容量推理服务，具有大量内存的单个 CPU 实例或此类实例的集群可能是更好的解决方案。
- 如果您使用的是具有大量数据或较大批处理大小的大型模型，那么您需要具有更多内存的大型实例。您也可以将模型分发到 GPU 集群。您可能会发现，如果减小批处理大小，则使用内存较少的实例将是更好的选择。这可能会影响您的准确性和训练速度。
- 如果您有兴趣使用 NVIDIA 集体通信库 (NCCL) 来运行机器学习应用程序，且需要大规模的节点间通信，那么您可能需要使用 Elastic Fabric Adapter (EFA)。

有关实例的更多详细信息，请参阅 EC2 实例类型。

以下主题提供有关实例类型注意事项的信息。

> **Important**
>
> Deep Learning AMI 包含由 NVIDIA Corporation 开发、拥有或提供的驱动程序、软件或工具包。您同意仅在包含 NVIDIA 硬件的 Amazon EC2 实例上使用这些 NVIDIA 驱动程序、软件或工具包。

主题

- DLAMI 的定价
- DLAMI 区域可用性
- 推荐的 GPU 实例
- 推荐的 CPU 实例
- 推荐的 Inferentia 实例
- 推荐的 Trainium 实例

- 推荐的 Habana 实例

DLAMI 的定价

DLAMI 中包含的深度学习框架是免费的，每个都有自己的开源许可证。尽管 DLAMI 中包含的软件是免费的，但您仍然需要为底层 Amazon EC2 实例硬件付费。

一些 Amazon EC2 实例类型被标记为免费。可以在其中一个免费实例上运行 DLAMI。这意味着，当您只使用该实例的容量时，使用 DLAMI 是完全免费的。如果您需要一个功能更强大、具有更多 CPU 核心、更多磁盘空间、更多 RAM，或者一个或多个 GPU 的实例，那么您就需要一个不在免费套餐实例类中的实例。

有关实例选择和定价的更多信息，请参阅 Amazon EC2 定价。

DLAMI 区域可用性

每个区域支持不同范围的实例类型，而且不同区域的实例类型成本通常稍有不同。DLAMI 并非在每个区域都可用，但可以将 DLAMI 复制到您选择的区域。有关更多信息，请参阅复制 AMI。请注意区域选择列表，并确保您选择一个靠近您或您客户的区域。如果您打算使用不止一个 DLAMI 并且可能创建一个集群，请确保为集群中的所有节点使用相同的区域。

有关区域的更多信息，请访问 EC2 区域。

后续步骤

推荐的 GPU 实例

推荐的 GPU 实例

我们推荐的 GPU 实例适用于大多数深度学习目的。在 GPU 实例上训练新模型比在 CPU 实例上更快。您可以在有多个 GPU 实例的情况下以线性方式进行缩放，也可以在具有 GPU 的多个实例上使用分布式训练。要设置分布式训练，请参阅 分布式训练。

以下实例类型支持 DLAMI。有关 GPU 实例类型选项及其用途的信息，请参阅 EC2 实例类型并选择加速计算。

> **Note**
>
> 应将模型大小作为选择实例的一个考虑因素。如果模型超出了实例的可用 RAM，请为应用程序选择其他具有足够内存的实例类型。

- Amazon EC2 P3 实例最多有 8 个 NVIDIA Tesla V100 GPU。
- Amazon EC2 P4 实例最多有 8 个 NVIDIA Tesla A100 GPU。
- Amazon EC2 G3 实例最多有 4 个 NVIDIA Tesla M60 GPU。
- Amazon EC2 G4 实例最多有 4 个 NVIDIA T4 GPU。
- Amazon EC2 G5 实例最多有 8 个 NVIDIA A10G GPU。
- Amazon EC2 G5g 实例具有基于 Arm 的 AWS Graviton2 处理器。

DLAMI 实例提供了用于监控和优化 GPU 进程的工具。有关监控 GPU 进程的更多信息，请参阅 GPU 监控和优化。

有关使用 G5g 实例的特定教程，请参阅 Graviton DLAMI。

后续步骤

推荐的 CPU 实例

推荐的 CPU 实例

无论您是在关注预算，了解深度学习，还是只是想运行一项预测服务，您在 CPU 类别中都有许多经济实惠的选项。某些框架利用了 Intel 的 MKL DNN，从而加快了在 C5（并非在所有区域中都可用）、C4 和 C3 CPU 实例类型上的训练和推理速度。有关 CPU 实例类型的信息，请参阅 EC2 实例类型并选择优化计算。

> **Note**
>
> 应将模型大小作为选择实例的一个考虑因素。如果模型超出了实例的可用 RAM，请为应用程序选择其他具有足够内存的实例类型。

- Amazon EC2 C5 实例最多有 72 个 Intel vCPU。C5 实例在科学建模、批处理、分布式分析、高性能计算 (HPC) 以及机器和深度学习推理方面表现出色。
- Amazon EC2 C4 实例拥有多达 36 个 Intel vCPU。

后续步骤

推荐的 Inferentia 实例

推荐的 Inferentia 实例

AWS Inferentia 实例旨在为深度学习模型推理工作负载提供高性能和成本效益。具体来说，Inf2 实例类型使用 AWS Inferentia 芯片和 AWS Neuron SDK，后者与 TensorFlow 和 PyTorch 等流行的机器学习框架集成。

客户使用 Inf2 实例之后，能够以最低的云端成本来运行大规模的机器学习推理应用程序，例如搜索、推荐引擎、计算机视觉、语音识别、自然语言处理、个性化和欺诈检测。

> Note
>
> 应将模型大小作为选择实例的一个考虑因素。如果模型超出了实例的可用 RAM，请为应用程序选择其他具有足够内存的实例类型。

- Amazon EC2 Inf2 实例最多有 16 个 AWS Inferentia 芯片和 100 Gbps 的网络吞吐量。

有关 AWS Amazon DLAMI 入门的更多信息，请参阅 带有 DLAMI 的 AWS 推理芯片。

后续步骤

推荐的 Trainium 实例

推荐的 Trainium 实例

AWS Trainium 实例旨在为深度学习模型推理工作负载提供高性能和成本效益。具体来说，Trn1 实例类型使用 AWS Trainium 芯片和 AWS Neuron SDK，后者与 TensorFlow 和 PyTorch 等流行的机器学习框架集成。

客户使用 Trn1 实例之后，能够以最低的云端成本来运行大规模的机器学习推理应用程序，例如搜索、推荐引擎、计算机视觉、语音识别、自然语言处理、个性化和欺诈检测。

> Note
>
> 应将模型大小作为选择实例的一个考虑因素。如果模型超出了实例的可用 RAM，请为应用程序选择其他具有足够内存的实例类型。

- Amazon EC2 Trn1 实例最多有 16 个 AWS Trainium 芯片和 100 Gbps 的网络吞吐量。

后续步骤

推荐的 Habana 实例

推荐的 Habana 实例

带 Habana 加速器的实例旨在为深度学习模型训练工作负载提供高性能和成本效益。具体来说，DL1 实例类型使用来自英特尔旗下公司 Habana Labs 的 Habana Gaudi 加速器。DL1 实例非常适合训练各种应用程序中使用的机器学习模型，例如自然语言处理、对象检测和分类、推荐引擎和自动驾驶汽车感知。

带 Habana 加速器的实例配置有 Habana SynapseAI 软件，并预先集成了 TensorFlow 和 PyTorch 等流行的机器学习框架。如果您正在寻找具有最佳性价比的组合来训练深度学习模型，请考虑带 Habana 加速器的实例，以实现最低的训练成本。

> **Note**
>
> 应将模型大小作为选择实例的一个考虑因素。如果模型超出了实例的可用 RAM，请为应用程序选择其他具有足够内存的实例类型。

- Amazon EC2 DL1 实例最多有八个 Habana Gaudi 加速器、256 GB 加速器内存、4 TB 本地 NVMe 存储器和 400 Gbps 的网络吞吐量。

有关 Habana DLAMI 入门的更多信息，请参阅 Habana DLAMI。

框架支持策略

AWS Deep Learning AMI (DLAMI) 简化了深度学习工作负载的图像配置，并使用最新的框架、硬件、驱动程序、库和操作系统进行了优化。本页详细介绍了适用于 DLAMI 的框架支持策略。有关可用 DLAMI 的列表，请参阅 DLAMI 发布说明。

支持的框架

参考以下 AWS Deep Learning AMI 框架支持策略表，以查看哪些框架和版本受到主动支持。

请参阅补丁结束，以查看对于由原始框架维护团队主动支持的当前版本，AWS 可以支持多长时间。框架和版本可在单框架 DLAMI 或多框架 DLAMI 中使用。

> Note
>
> 在框架版本 x.y.z 中，x 表示主要版本，y 表示次要版本，z 表示补丁版本。例如，对于 TensorFlow 2.6.5，主版本为 2，次要版本为 6，补丁版本为 5。

有关特定映像的更多详细信息，请参阅发布说明：

- 单框架 DLAMI 发布说明
- 多框架 DLAMI 发布说明

常见问题

- 哪些框架版本会获得安全补丁？
- 发布新框架版本时，AWS 会发布哪些映像？
- 哪些图片有新增 SageMaker/AWS功能？
- “支持的框架”表中是如何定义当前版本的？
- 如果我运行的版本不在“支持的框架”表中，该怎么办？
- DLAMIs 是否支持以前的版本？ TensorFlow
- 如何找到支持的框架版本的最新补丁映像？
- 多长时间发布一次新映像？

- 运行工作负载时，能在我的实例上以替代方式安装补丁吗？
- 如果有新的补丁或更新的框架版本可用，会发生什么呢？
- 是否可在不更改框架版本的情况下更新依赖项？
- 对我的框架版本的主动支持何时结束？
- 对于框架版本不再主动维护的映像，会为其安装补丁吗？
- 如何使用旧框架版本？
- 如何保持框架及其版本 up-to-date 的支持变更？
- 是否需要商业许可证才能使用 Anaconda 存储库？

哪些框架版本会获得安全补丁？

如果框架版本在 AWS Deep Learning AMI 框架支持策略表中标记为已支持，它就会获得安全补丁。

发布新框架版本时，AWS 会发布哪些映像？

我们会在 TensorFlow 和的新版本发布后不久就会发布新 PyTorch的 DLAMI。这包括框架的主要版本、主要的次要版本和 major-minor-patch 版本。当新版本的驱动程序和库可用时，我们也会更新映像。有关映像维护的更多信息，请参阅 对我的框架版本的主动支持何时结束？。

哪些图片有新增 SageMaker/AWS功能？

新功能通常在最新版本的 dLaMis 中发布，适用于 PyTorch 和中。 TensorFlow有关新增 SageMaker 或AWS功能的详细信息，请参阅特定图像的发行说明。有关可用 DLAMI 的列表，请参阅 DLAMI 发布说明。有关映像维护的更多信息，请参阅 对我的框架版本的主动支持何时结束？。

“支持的框架”表中是如何定义当前版本的？

Frame AWS work S AWS Deep Learning AMIupport Policy 表中的当前版本是指在上提供的最新框架版本 GitHub。每个最新版本都包括对 DLAMI 中驱动程序、库和相关软件包的更新。有关映像维护的信息，请参阅 对我的框架版本的主动支持何时结束？

如果我运行的版本不在“支持的框架”表中，该怎么办？

如果您运行的版本不在 AWS Deep Learning AMI 框架支持策略表中，则可能没有最新的驱动程序、库和相关包。要获得更多 up-to-date 版本，我们建议您使用您选择的最新 DLAMI 升级到可用的支持框架之一。有关可用 DLAMI 的列表，请参阅 DLAMI 发布说明。

DLAMIs 是否支持以前的版本？ TensorFlow

不是。 如Framework Support P olicy表所述，我们支持每个框架最新主要版本的最新补丁版本，自其首次 GitHub 发布起365天后AWS Deep Learning AMI发布。有关更多信息，请参阅 如果我运行的版本不在“支持的框架”表中，该怎么办？。

如何找到支持的框架版本的最新补丁映像？

要使用具有最新框架版本的 DLAMI，请检索 DLAMI ID，然后使用 EC2 控制台来用该 ID 启动 DLAMI。有关检索 AWS Deep Learning AMI ID 的示例 AWS CLI 命令，请参阅 AWS Deep Learning AMI 目录中的深度学习框架部分。AWSCLI AMI ID 查询也包含在单框架 DLAMI 发布说明中。您选择的框架版本必须在 AWS Deep Learning AMI 框架支持策略表中标记为已支持。

多长时间发布一次新映像？

提供更新的补丁版本是我们的首要任务。我们通常会尽早创建安装了补丁的映像。我们会监控新修补的框架版本（例如 TensorFlow 2.9 到 TensorFlow 2.9.1）和新的次要发行版本（例如 TensorFlow 2.9 到 TensorFlow 2.10），并尽早提供它们。当现有版本与新版本 TensorFlow 的 CUDA 一起发布时，我们会为该版本发布支持新 CUDA 版本 TensorFlow 的新 DLAMI。

运行工作负载时，能在我的实例上以替代方式安装补丁吗？

不能。DLAMI 的补丁更新不是“替代”更新。

您必须打开新的 EC2 实例，迁移您的工作负载和脚本，然后关闭之前的实例。

如果有新的补丁或更新的框架版本可用，会发生什么呢？

请定期查看发布说明页面以获取您的映像。我们鼓励您在新的补丁或更新的框架可用时将框架升级。有关可用 DLAMI 的列表，请参阅 DLAMI 发布说明。

是否可在不更改框架版本的情况下更新依赖项？

我们在不更改框架版本的情况下更新依赖项。但是，如果依赖项更新导致不兼容，我们就会创建不同版本的映像。请务必查看 DLAMI 发布说明，了解更新的依赖项信息。

对我的框架版本的主动支持何时结束？

DLAMI 映像是不可变的。一旦创建，就不会改变。结束对框架版本的主动支持涉及四个主要原因：

- 框架版本（补丁）升级
- AWS 安全补丁
- 补丁结束日期（已过期）
- 依赖关系 end-of-support

> **Note**
>
> 由于版本补丁升级和安全补丁的频率很高，我们建议您经常查看 DLAMI 发布说明页面，并在发生更改时进行升级。

框架版本（补丁）升级

如果你有一个基于 2.7.0 的 DLAMI 工作负载 TensorFlow并在版本为 2.7.1 之后发布，那么 TensorFlow AWS就要发布一个 2.7.1 GitHub 版本的新 DLAMI。 TensorFlow 2.7.1 版本的新镜像发布后，将不再主动维护之前版本为 TensorFlow 2.7.0 的图像。2.7.0 版本的 DLAMI TensorFlow 没有收到更多补丁。然后，2.7版的DLAMI发行说明页面将更新 TensorFlow 为最新信息。没有为每个次要补丁提供单独的发布说明页面。

由于补丁升级而创建的新 DLAMI 将使用新的 AMI ID 进行指定。

AWS 安全补丁

如果您的工作负载基于 TensorFlow 2.7.0 版本的映像并AWS制作了安全补丁，则会为 2.7.0 发布新版本的 DLAMI。 TensorFlow TensorFlow 2.7.0 版本的图像的先前版本已不再活跃维护。有关更多信息，请参阅 运行工作负载时，能在我的实例上以替代方式安装补丁吗？。有关查找最新 DLAMI 的步骤，请参阅 如何找到支持的框架版本的最新补丁映像？

由于补丁升级而创建的新 DLAMI 将使用新的 AMI ID 进行指定。

补丁结束日期（已过期）

DLAMIs 在 GitHub 发布日期 365 天后就到了补丁的终止日期。

对于多框架 DLAMI，当其中一个框架版本更新时，需要具有更新版本的新 DLAMI。不再主动维护使用旧框架版本的 DLAMI。

> **Important**
>
> 当有重大框架更新时，我们会例外处理。例如。如果 TensorFlow 1.15 更新到 TensorFlow 2.0，那么我们将在自 GitHub 发布之日起两年内继续支持最新版本的 TensorFlow 1.15，或者在 Origin 框架维护团队取消支持后的六个月内（以较早的日期为准）。

依赖关系 end-of-support

如果你正在使用 Python 3.6 在 TensorFlow 2.7.0 的 DLAMI 映像上运行工作负载，并且该版本的 Python 已标记为 end-of-support，那么所有基于 Python 3.6 的 DLAMI 图像都将不再被主动维护。同样，如果标记了像Ubuntu 16.04这样的操作系统版本 end-of-support，则所有依赖于Ubuntu 16.04的 DLAMI镜像都将不再被主动维护。

对于框架版本不再主动维护的映像，会为其安装补丁吗？

不会。不再主动维护的图像就不会有新版本。

如何使用旧框架版本？

要将 DLAMI 与旧框架版本结合使用，请检索 DLAMI ID，然后使用 EC2 控制台来用该 ID 启动 DLAMI。有关检索 AMI ID 的 AWS CLI 命令，请参阅 AWS 深度学习 AMI 目录中的深度学习框架部分。AWSCLI AMI ID 查询也包含在单框架 DLAMI 发布说明中。

如何保持框架及其版本 up-to-date 的支持变更？

up-to-date 使用 DLAMI 发行说明中的 Framework Su pport Policy 表，继续使用 D LAMI AWS Deep Learning AMI 框架和版本。

是否需要商业许可证才能使用 Anaconda 存储库？

Anaconda 转向了针对某些用户的商业许可模式。主动维护的 DLAMI 已从 Anaconda 通道迁移到公开可用的开源 Conda 版本 (conda-forge)。

启动和配置 DLAMI

如果您在这里，您应该已经有了想要启动哪个 AMI 的想法。否则，请在 AWS Deep Learning AMI 目录中找到 DLAMI 及其相关硬件、框架和 ID 检索，或者在 DLAMI 的发布说明 中查看当前和历史 DLAMI 发布说明。

您还应该知道您要选择哪个实例类型和区域。否则，请浏览选择 DLAMI 的实例类型。

> Note
>
> 我们将在示例中使用 p3.16xlarge 作为默认实例类型。只需将此替换为您心中的任何实例类型。

> Important
>
> 如果您计划使用 Elastic Inference，则必须先完成 Elastic Inference 设置，然后才能启动您的 DLAMI。

主题

- 步骤 1：启动 DLAMI
- 步骤 2：连接到 DLAMI
- 步骤 3：测试 DLAMI
- 步骤 4：管理 DLAMI 实例
- 清除
- 设置 Jupyter Notebook 服务器

步骤 1：启动 DLAMI

> Note
>
> 对于此演练，我们可能会针对深度学习 AMI (Ubuntu 18.04) 进行引用。即使选择其他 DLAMI，您也应能按照本指南进行操作。

1. 找到您的 DLAMI 的 ID
2. 从您的 DLAMI 中启动 Amazon EC2 实例

   您将使用 Amazon EC2 控制台。请按照 从 Amazon EC2 控制台启动 中详述的说明进行操作

   > **Tip**
   >
   > CLI 选项：如果选择使用 AWS CLI 来启动 DLAMI，您将需要 AMI 的 ID、区域和实例类型，以及您的安全令牌信息。请确保您已拥有 AMI 和实例 ID。如果您尚未设置 AWS CLI，请首先使用安装 AWS 命令行界面指南进行设置。

3. 如果您已经完成了上述某个选项的步骤，请等待实例准备好。这通常仅需要几分钟时间。您可以在 EC2 控制台中验证实例的状态。

检索 DLAMI ID

每个 AMI 都有唯一标识符 (ID)。您可以使用 AWS 命令行界面 (AWS CLI) 来查询您选择的 DLAMI 的 ID。如果您尚未安装 AWS CLI，请参阅 AWS CLI 入门。

> **Note**
>
> 提示：您可以在 AWS Deep Learning AMI 目录中找到所有 DLAMI 及其相关的处理器/加速器、操作系统、计算架构、推荐 Amazon EC2 实例系列、支持状态和 ID 检索查询。有关其他信息（驱动程序、Python 版本、Amazon EBS 类型），另请参阅 DLAMI 的发布说明 中的 DLAMI 发布说明。

1. 请确保已配置您的 AWS 凭证。

```
aws configure
```

2. 使用以下命令来检索您的 DLAMI 的 ID，或查找 AWS Deep Learning AMI 目录中提供的查询。

```
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

您可以为给定框架指定发布版本，也可以通过将版本号替换为问号来获取最新版本。

3. 该输出值应该类似于以下内容：

```
ami-094c089c38ed069f2
```

复制此 DLAMI ID，然后按 q 退出提示。

下一个步骤

从 Amazon EC2 控制台启动

从 Amazon EC2 控制台启动

Note

要使用 Elastic Fabric Adapter (EFA) 来启动实例，请参阅这些步骤。

1. 打开 EC2 控制台。
2. 请注意最顶层的导航中的当前区域。如果这不是您所需的 AWS 区域，请更改此选项，然后再继续。有关更多信息，请参阅 EC2 区域。
3. 选择启动实例。
4. 为您的实例输入名称，然后选择适合您的 DLAMI。

   a. 在我的 AMI 中查找现有 DLAMI，或者选择快速入门。
   b. 按 DLAMI ID 进行搜索。浏览这些选项，然后选中您的选择。
5. 选择一个实例类型。您可以在 AWS Deep Learning AMI 目录中找到适用于您的 DLAMI 的推荐实例系列。有关 DLAMI 实例类型的一般建议，请参阅实例选择。

Note

如果您要使用 Elastic Inference (EI)，请单击配置实例详细信息，选择添加 Amazon EI 加速器，然后选择 Amazon EI 加速器的大小。

6. 选择启动实例。

Tip

如需带屏幕截图的演练，请查看使用 AWS 深度学习 AMI 的深度学习入门。

下一个步骤

步骤 2：连接到 DLAMI

步骤 2：连接到 DLAMI

连接到您从客户端（Windows、MacOS 或 Linux）启动的 DLAMI。有关更多信息，请参阅适用于 Linux 实例的 Amazon EC2 用户指南中的连接到您的 Linux 实例。

如果在登录后您希望执行 Jupyter 设置，请在手边保留一份 SSH 登录命令。您将使用这个命令的各种变体连接到 Jupyter 网页。

下一个步骤

步骤 3：测试 DLAMI

步骤 3：测试 DLAMI

根据您的 DLAMI 版本不同，您的测试选项也会不同：

- 带 Conda 的深度学习 AMI — 转到 使用带 Conda 的深度学习 AMI。
- 深度学习基础 AMI — 引用您所需框架的安装文档。

您还可以创建 Jupyter 笔记本电脑，试用教程，或开始使用 Python 编码。有关更多信息，请参阅设置 Jupyter Notebook 服务器。

步骤 4：管理 DLAMI 实例

一旦有修补程序和更新推出便立即应用，从而始终保持操作系统和其他已安装软件为最新。

如果您使用的是 Amazon Linux 或 Ubuntu，则当您登录到您的 DLAMI 时，便会收到更新通知（如果有）并且会看到更新说明。有关 Amazon Linux 维护的更多信息，请参阅更新实例软件。对于 Ubuntu 实例，请参阅官方 Ubuntu 文档。

在 Windows 上，定期检查 Windows Update 有无软件和安全更新。如果您愿意，可以自动应用更新。

Important

有关 Meltdown 和 Spectre 漏洞以及如何修补操作系统解决以这些问题的信息，请参阅安全公告 AWS-2018-013。

清除

当不再需要 DLAMI 时，您可以将其停止或终止，以免持续产生费用。停止的实例会保留，以便您可以稍后再继续。您的配置、文件和其他非易失性信息都存储在 Amazon S3 上的卷中。在实例停止的情况下，将收取您小额 S3 费用来保留该卷，但在计算资源处于停止状态时将不再对其收取费用。当您重新启动实例时，它将挂载该卷，且您的数据会重新恢复。如果您终止实例，它将消失，且您将无法重新启动它。您的数据实际仍驻留在 S3 中，因此，为了防止任何进一步的费用，您还需要删除该卷。有关更多说明，请参阅适用于 Linux 实例的 Amazon EC2 用户指南中的终止实例。

设置 Jupyter Notebook 服务器

Jupyter 笔记本服务器允许您从 DLAMI 实例创建和运行 Jupyter 笔记本。使用 Jupyter 笔记本，您可以在使用 AWS 基础架构和访问 DLAMI 中内置的软件包的同时，进行机器学习 (ML) 实验的训练和推理。有关 Jupyter 笔记本的更多信息，请参阅 Jupyter 笔记本文档。

要设置 Jupyter 笔记本服务器，您需要：

- 在您的 Amazon EC2 DLAMI 实例上配置 Jupyter 笔记本服务器。
- 配置您的客户端，以便您可以连接到 Jupyter 笔记本服务器。我们提供适用于 Windows、macOS 和 Linux 客户端的配置说明。
- 登录 Jupyter 笔记本服务器，测试设置。

要完成设置 Jupyter 的步骤，请按照以下主题中的说明进行操作。设置 Jupyter 笔记本服务器后，请参阅 运行 Jupyter 笔记本电脑教程，了解有关运行 DLAMI 中附带的示例笔记本的信息。

主题

- 保护您的 Jupyter 服务器
- 启动 Jupyter Notebook 服务器
- 配置客户端以连接到 Jupyter 服务器
- 登录 Jupyter Notebook 服务器进行测试

保护您的 Jupyter 服务器

下面我们使用 SSL 和自定义密码来设置 Jupyter。

连接到 Amazon EC2 实例，然后完成以下步骤。

配置 Jupyter 服务器

1. Jupyter 提供了一个密码实用工具。运行以下命令，在命令提示符处输入您的首选密码。

```
$ jupyter notebook password
```

   输出类似如下：

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

2. 创建自签名 SSL 证书。按照提示填写您认为适当的区域。如果要将提示留空，则必须输入 `.`。您的答案将不会影响证书的功能性。

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

> **Note**
>
> 您可能希望创建一个常规的 SSL 证书，该证书为第三方签名，且不会导致浏览器向您发出安全警告。此过程涉及内容较多。访问 Jupyter 的文档以了解更多信息。

下一个步骤

[启动 Jupyter Notebook 服务器]

启动 Jupyter Notebook 服务器

现在，您可以通过登录实例，然后运行以下命令来启动 Jupyter 服务器，该命令将使用您在上一步中创建的 SSL 证书。

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

启动服务器后，即可通过 SSH 隧道从您的客户端计算机连接到服务器。当服务器运行时，您将看到来自 Jupyter 的一些输出，这些输出证实服务器正在运行。此时，请忽略称您可以通过本地主机 URL 访问服务器的标注，因为这其实不可以，直到您创建了隧道为止。

> **Note**
>
> 当您使用 Jupyter Web 界面切换框架时，Jupyter 将处理切换环境。更多有关信息见于通过 Jupyter 切换环境。

下一个步骤

配置客户端以连接到 Jupyter 服务器

配置客户端以连接到 Jupyter 服务器

配置您的客户端连接到 Jupyter Notebook 服务器后，您可以在您的工作区中的服务器上创建和访问笔记本，并在服务器上运行深度学习代码。

有关配置信息，请选择以下链接之一。

主题

- 配置 Windows 客户端
- 配置 Linux 或 macOS 客户端

配置 Windows 客户端

准备

请确保您拥有设置 SSH 隧道所需的以下信息：

- Amazon EC2 实例的公有 DNS 名称。您可以在 EC2 控制台中找到公有 DNS 名称。
- 私有密钥文件的密钥对。有关访问密钥对的更多信息，请参阅 Amazon EC2 用户指南（适用于 Linux 实例）中的 Amazon EC2 密钥对。

从 Windows 客户端使用 Jupyter 笔记本

请参阅这些关于从 Windows 客户端连接到您的 Amazon EC2 实例的指南。

1. 排查实例的连接问题
2. 使用 PuTTY 从 Windows 连接到 Linux 实例

要创建连接 Jupyter 服务器的隧道，建议先在您的 Windows 客户端上安装 Git Bash，然后按照 Linux/macOS 客户端说明操作。不过，您可以使用任何方法打开包含端口映射的 SSH 隧道。请参阅 Jupyter 的文档了解更多信息。

下一个步骤

配置 Linux 或 macOS 客户端

配置 Linux 或 macOS 客户端

1. 打开 终端。
2. 运行以下命令，将本地端口 8888 上的所有请求转发到远程 Amazon EC2 实例上的端口 8888。通过替换密钥位置来更新命令，以访问 Amazon EC2 实例和您的 Amazon EC2 实例的公有 DNS 名称。注意，对于 Amazon Linux AMI，用户名是 `ec2-user` 而非 ubuntu。

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

   此命令将在您的客户端与运行 Jupyter 笔记本服务器的远程 Amazon EC2 实例之间打开一个隧道。

下一个步骤

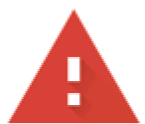
登录 Jupyter Notebook 服务器进行测试

登录 Jupyter Notebook 服务器进行测试

现在，您已准备就绪，可登录到 Jupyter Notebook 服务器。

接下来，通过浏览器测试与服务器的连接。

1. 在浏览器的地址栏中，键入以下 URL，或单击此链接：https://localhost:8888
2. 借助自签名 SSL 证书，您的浏览器会警告和提示您避免继续访问网站。



Your connection is not private

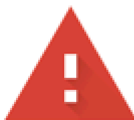
Attackers might be trying to steal your information from **localhost** (for example, passwords, messages, or credit cards). Learn more

NET::ERR_CERT_AUTHORITY_INVALID

☐ Help improve Safe Browsing by sending some system information and page content to Google. Privacy policy

Back to safety

由于这是您自己设置的，所以可以安全地继续。根据您的浏览器情况，有可能出现“高级”、“显示详细信息”等类似按钮。



Your connection is not private

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

NET::ERR_CERT_AUTHORITY_INVALID

☐ Help improve Safe Browsing by sending some system information and page content to Google. Privacy policy

Hide advanced　　Back to safety

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

Proceed to localhost (unsafe)

单击此消息，然后单击“前进到本地主机”链接。如果连接成功，则会看到 Jupyter Notebook 服务器网页。此时，系统将要求您提供先前设置的密码。

现在，您可以访问 DLAMI 上运行的 Jupyter 笔记本服务器了。您可以创建新的笔记本或运行提供的教程。

使用 DLAMI

主题

- 使用带 Conda 的深度学习 AMI
- 使用深度学习基础 AMI
- 运行 Jupyter 笔记本电脑教程
- 教程

以下部分介绍如何使用带 Conda 的深度学习 AMI来切换环境、从每个框架运行示例代码以及运行 Jupyter，以便您可以尝试不同的 Notebook 教程。

使用带 Conda 的深度学习 AMI

主题

- 带 Conda 的深度学习 AMI 的简介
- 登录到你的 DLAMI
- 启动 TensorFlow 环境
- 切换到 PyTorch Python 3 环境
- 切换到 MXNet Python 3 环境
- 删除环境

带 Conda 的深度学习 AMI 的简介

Conda 是一个开源程序包管理系统和环境管理系统，在 Windows、macOS 和 Linux 上运行。Conda 快速安装、运行和更新程序包及其依赖项。Conda 可轻松创建、保存、加载和切换本地计算机上的环境。

带 Conda 的深度学习 AMI 已配置完成，以便让您轻松切换深度学习环境。以下说明为您介绍与 conda 相关的一些基本命令。它们还可以帮助您验证框架的基本导入正常运行，并且您可以使用框架运行一些简单操作。然后，您可以继续查看随 DLAMI 提供的更全面的教程，或者每个框架的项目站点上提供的框架示例。

登录到你的 DLAMI

登录服务器后，您会看到服务器的“每日消息”（MOTD），它介绍了可以用来切换不同深度学习框架的各种 Conda 命令。以下是示例 MOTD。由于新版本 DLAMI 的发布，您的特定 MOTD 可能不同。

> **Note**
>
> 从 v28 版本开始，我们就不再包含 CNTK、Caffe、Caffe2、Theano、Chainer 和 Keras Conda 环境了。 AWS Deep Learning AMI 包含这些环境 AWS Deep Learning AMI 的先前版本将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

```
=============================================================================
       __|  __|_  )
       _|  (     /   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
```

每个 Conda 命令具有以下模式：

`source activate framework_python-version`

例如，您可能会看到 `for MXNet(+Keras1) with Python3 (CUDA 10.1) _________________________ source activate mxnet_p36`，这意味着该环境具有 MXNet、Keras 1、Python 3 和 CUDA 10.1。要激活此环境，您可以使用以下命令：

```
$ source activate mxnet_p36
```

启动 TensorFlow 环境

> **Note**
>
> 在启动您的第一个 Conda 环境时，请在其加载期间耐心等待。带 Conda 的深度学习 AMI 会在框架首次激活时自动为您的 EC2 实例安装框架的最优化版本。您不应期望后续的延迟。

1. 激活 Python 3 的 TensorFlow 虚拟环境。

   ```
   $ source activate tensorflow_p37
   ```

2. 启动 iPython 终端。

   ```
   (tensorflow_37)$ ipython
   ```

3. 运行一个快速 TensorFlow 程序。

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

```
print(sess.run(hello))
```

您应该会看到“Hello, Tensorflow!”

后续步骤

运行 Jupyter 笔记本电脑教程

切换到 PyTorch Python 3 环境

如果您仍然处于 iPython 控制台中，则使用 `quit()`，然后准备切换环境。

- 激活 Python 3 的 PyTorch 虚拟环境。

```
$ source activate pytorch_p36
```

测试一些 PyTorch 代码

要测试您的安装，请使用 Python 编写用于创建和打印数组的 PyTorch 代码。

1. 启动 iPython 终端。

```
(pytorch_p36)$ ipython
```

2. 导入 PyTorch。

```
import torch
```

   您可能会看到一条关于第三方软件包的警告消息。您可以忽略它。

3. 创建一个 5x3 矩阵，将元素随机初始化。打印数组。

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

   验证结果。

```
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]])
```

切换到 MXNet Python 3 环境

如果您仍然处于 iPython 控制台中，则使用 `quit()`，然后准备切换环境。

- 激活适用于 Python 3 的 MXNet 虚拟环境。

```
$ source activate mxnet_p36
```

测试一些 MXNet 代码

要测试您的安装，请使用 Python 编写 MXNet 代码，以使用 `NDArray` API 创建并打印数组。有关更多信息，请参阅 NDArray API。

1. 启动 iPython 终端。

```
(mxnet_p36)$ ipython
```

2. 导入 MXNet。

```
import mxnet as mx
```

您可能会看到一条关于第三方软件包的警告消息。您可以忽略它。

3. 创建一个 5x5 矩阵、一个 `NDArray` 实例，将元素初始化为 0。打印数组。

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

验证结果。

```
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)
```

您可以在 MXNet 教程部分找到 MXNet 的更多示例。

删除环境

如果您用尽了 DLAMI 上的空间，则可以选择卸载不用的 Conda 软件包：

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

使用深度学习基础 AMI

使用深度学习基础 AMI

Base AMI 附带 GPU 驱动程序基础平台，以及可用来部署您自己的自定义深度学习环境的加速库。默认情况下，该 AMI 配置为采用 NVIDIA CUDA 11.0 环境。您也可以在不同版本的 CUDA 之间切换。有关如何执行此操作，请参阅以下说明。

配置 CUDA 版本

您可以通过运行 NVIDIA 的 nvcc 程序来验证 CUDA 版本。

```
nvcc --version
```

您可以使用以下 bash 命令来选择并验证特定 CUDA 版本。

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

有关更多信息，请参阅基础 DLAMI 发布说明。

运行 Jupyter 笔记本电脑教程

每个深度学习项目的源代码都附带了教程和示例，大多数情况下，它们可以在任何 DLAMI 上运行。如果您选择了 带 Conda 的深度学习 AMI，那么您将获得一些已经建立并准备好尝试的精选教程的额外好处。

> **Important**
> 要运行安装在 DLAMI 上的 Jupyter 笔记本教程，您需要 设置 Jupyter Notebook 服务器。

Jupyter 服务器运行后，您可以通过 Web 浏览器运行这些教程。如果您正在运行带 Conda 的深度学习 AMI，或者如果您已经建立了 Python 环境，则可以从 Jupyter 笔记本界面切换 Python 内核。在尝试运行特定于框架的教程之前，请选择合适的内核。我们为带 Conda 的深度学习 AMI 的用户提供了更多这方面的示例。

> **Note**
> 许多教程都需要额外 Python 模块，您的 DLAMI 上可能尚未设置这些模块。如果您收到类似 `"xyz module not found"` 的错误，请登录到 DLAMI，激活环境（如上所述），然后安装必要的模块。

> **Tip**
> 深度学习教程和示例通常依赖于一个或多个 GPU。如果您的实例类型没有 GPU，您可能需要更改一些示例代码才能使其运行。

导航已安装的教程

一旦登录到 Jupyter 服务器且可以看到教程目录（仅限带 Conda 的深度学习 AMI 上）时，就会看到按每个框架名称排列的教程文件夹。如果您没有看到某个框架，则表明在您当前 DLAMI 上该框架的教程不可用。单击框架名称以查看列出的教程，然后单击一个教程，将其启动。

在带 Conda 的深度学习 AMI 上第一次运行 Notebook 时，它会想知道您要使用哪个环境。它会提示您从列表中进行选择。每个环境都根据以下模式命名：

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

例如，您可能会看到 `Environment (conda_mxnet_p36)`，这意味着该环境具有 MXNet 和 Python 3。您也可能会看到 `Environment (conda_mxnet_p27)`，这意味着该环境具有 MXNet 和 Python 2。

> **Tip**
>
> 如果您想知道哪个版本的 CUDA 处于活动状态，一种查看方法是在首次登录到 DLAMI 时在 MOTD 中查看。

通过 Jupyter 切换环境

如果您决定尝试一个不同框架的教程，一定要验证当前正在运行的内核。此信息可以在 Jupyter 界面的右上方看到，就在注销按钮的下方。您可以在任何打开的笔记本电脑上更改内核，方法是依次单击 Kernel、Change Kernel 菜单项，然后单击正运行的笔记本电脑适合的环境。

此时您需要重新运行任何单元，因为内核中的更改将会擦除之前运行的任何内容的状态。

> **Tip**
>
> 在框架之间进行切换可能很有趣而且很有教育意义，但是可能导致耗尽内存。如果您开始遇到错误，请查看运行 Jupyter 服务器的终端窗口。这里有一些有用的消息和错误记录，你可能会看到一个 out-of-memory错误。要解决这一问题，您可以转到 Jupyter 服务器的主页中，单击 Running 选项卡，然后为每个教程单击 Shutdown，因为这些教程可能仍然在后台运行，导致耗尽了所有内存。

后续步骤

有关每个框架的更多示例和示例代码，请单击 Next 或继续 Apache MXNet（孵化版）。

教程

以下是有关如何使用带 Conda 的深度学习 AMI 的软件的教程。

主题

- 10 分钟教程
- 激活框架
- 调试和可视化
- 分布式训练

- Elastic Fabric Adapter
- GPU 监控和优化
- 带有 DLAMI 的 AWS 推理芯片
- Graviton DLAMI
- Habana DLAMI
- 推理
- 将框架用于 ONNX
- 模型处理

10 分钟教程

- 启动 AWS Deep Learning AMI （10 分钟后）
- 在 Amazon EC2 上使用 DLC 来训练深度学习模型（10 分钟内）

激活框架

以下是在带 Conda 的深度学习 AMI 上安装的深度学习框架。单击某个框架即可了解如何将其激活。

主题

- Apache MXNet（孵化版）
- Caffe2
- Chainer
- CNTK
- Keras
- PyTorch
- TensorFlow
- TensorFlow 2
- TensorFlow 和 Horovod
- TensorFlow 2 和 Horovod
- Theano

Apache MXNet（孵化版）

激活 Apache MXNet（孵化版）

本教程介绍如何在运行带 Conda 的深度学习 AMI（Conda 上的 DLAMI）的实例上激活 MXNet 并运行 MXNet 程序。

当框架的稳定 Conda 程序包发布时，它会在 DLAMI 上进行测试并预安装。如果您希望运行最新的、未经测试的每日构建版本，您可以手动安装 MXNet 的每日构建版本（试验）。

在带 Conda 的 DLAMI 上运行 MXNet

1. 要激活该框架，请打开带 Conda 的 DLAMI 的 Amazon Elastic Compute Cloud (Amazon EC2) 实例。

   - 对于使用 CUDA 9.0 和 MKL-DNN 的 Python 3 上的 MXNet 和 Keras 2，运行以下命令：

     ```
     $ source activate mxnet_p36
     ```

   - 对于使用 CUDA 9.0 和 MKL-DNN 的 Python 2 上的 MXNet 和 Keras 2，运行以下命令：

     ```
     $ source activate mxnet_p27
     ```

2. 启动 iPython 终端。

   ```
   (mxnet_p36)$ ipython
   ```

3. 运行快速 MXNet 程序。创建一个 5x5 矩阵、一个 `NDArray` 实例，将元素初始化为 0。打印数组。

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

4. 验证结果。

   ```
   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)
   ```

安装 MXNet 的每日构建版本（试验）

您可以将最新的 MXNet 构建版本安装到带 Conda 的深度学习 AMI 上的任一或两个 MXNet Conda 环境中。

从每日构建版本安装 MXNet

1. • 对于 Python 3 MXNet 环境，请运行以下命令：

   ```
   $ source activate mxnet_p36
   ```

   • 对于 Python 2 MXNet 环境，请运行以下命令：

   ```
   $ source activate mxnet_p27
   ```

2. 删除当前安装的 MXNet。

   > **Note**
   >
   > 其余步骤假定您使用的是 `mxnet_p36` 环境。

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

3. 安装 MXNet 的最新的每日构建版本。

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

4. 要验证您是否已成功安装最新的每日构建版本，请启动 IPython 终端并检查 MXNet 版本。

   ```
   (mxnet_p36)$ ipython
   ```

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

   输出应打印 MXNet 的最新稳定版本。

更多教程

您可以在 DLAMI 主目录的带 Conda 的深度学习 AMI 教程文件夹中找到更多教程。

1. 使用 Apache mxNet（孵化中）对 50 模型进行推理 ResNet
2. 将适用于推理的 Apache MXNet（孵化版）与 ONNX 模型结合使用
3. 适用于 Apache MXNet 的模型服务器 (MMS)

有关更多教程和示例，请参阅该框架的官方 Python 文档、适用于 MXNet 的 Python API 或 Apache MXNet 网站。

Caffe2

Note

从 v28 版本开始， AWS Deep Learning AMI 中将不再包含 CNTK、Caffe、Caffe2 和 Theano Conda 环境。包含这些环境 AWS Deep Learning AMI 的先前版本将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

Caffe2 教程

要激活框架，请按照这些有关带 Conda 的深度学习 AMI 的说明进行操作。

只有使用 CUDA 9 和 cuDNN 7 的 Python 2 选项：

```
$ source activate caffe2_p27
```

启动 iPython 终端。

```
(caffe2_p27)$ ipython
```

运行快速 Caffe2 程序。

```
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)
```

您应该会看到系统输出初始 numpy 随机数组，然后这些数组加载到了 Caffe2 blob 中。请注意，加载之后，它们是相同的。

更多教程

如需查看更多教程和示例，请参阅该框架的官方 Python 文档、适用于 Caffe2 的 Python API 和 Caffe2 网站。

Chainer

> **Note**
>
> 从 v28 版本开始，我们将不再在 AWS Deep Learning AMI 中包含 Chainer Conda 环境。包含这些环境 AWS Deep Learning AMI 的先前版本将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

Chainer 是一种基于 Python 的灵活框架，用于轻松直观地编写复杂的神经网络架构。利用 Chainer，您可以轻松使用多 GPU 实例进行训练。Chainer 还会自动记录结果、图表损失和精度并生成用于使用计算图来可视化神经网络的输出。它包含在带 Conda 的深度学习 AMI（带 Conda 的 DLAMI）中。

激活 Chainer

1. 连接到运行带 Conda 的深度学习 AMI 的实例。有关如何选择或连接到实例，请参阅 the section called “实例选择” 或 Amazon EC2 文档。
2. • 激活 Python 3 Chainer 环境：

   ```
   $ source activate chainer_p36
   ```

   • 激活 Python 2 Chainer 环境：

   ```
   $ source activate chainer_p27
   ```

3. 启动 iPython 终端：

```
(chainer_p36)$ ipython
```

4. 测试导入 Chainer 以验证其是否运行正常：

```
import chainer
```

   您可能会看到几条警告消息，但没有错误。

更多信息

- 请尝试有关Chainer 的教程。
- 您之前下载的位于源内的 `Chainer` 示例文件夹包含更多示例。请试用这些示例以了解其性能。
- 要了解有关 Chainer 的更多信息，请参阅 Chainer 文档网站。

CNTK

> **Note**
>
> 从 v28 版本开始， AWS Deep Learning AMI 中将不再包含 CNTK、Caffe、Caffe2 和 Theano Conda 环境。包含这些环境 AWS Deep Learning AMI 的先前版本将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

激活 CNTK

本教程介绍如何在运行带 Conda 的深度学习 AMI（Conda 上的 DLAMI）的实例上激活 CNTK 并运行 CNTK 程序。

当框架的稳定 Conda 程序包发布时，它会在 DLAMI 上进行测试并预安装。如果您希望运行最新的、未经测试的每日构建版本，您可以手动安装 CNTK 的每日构建版本（试验）。

在带 Conda 的 DLAMI 上运行 CNTK

1. 要激活 CNTK，请打开带 Conda 的 DLAMI 的 Amazon Elastic Compute Cloud (Amazon EC2) 实例。
   - 对于使用 CUDA 9 和 cuDNN 7 的 Python 3：

```
$ source activate cntk_p36
```

- 对于使用 CUDA 9 和 cuDNN 7 的 Python 2：

```
$ source activate cntk_p27
```

2. 启动 iPython 终端。

```
(cntk_p36)$ ipython
```

3. - 如果您具有 CPU 实例，请运行此快速 CNTK 程序。

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

   您应该看到 CNTK 版本，然后输出一个 2x3 的 3 数组。

   - 如果您有 GPU 实例，可以使用以下代码示例对其进行测试。如果 CNTK 可以访问 GPU，则 `True` 的结果就是您所期望的。

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

安装 CNTK 的每日构建版本（试验）

您可以将最新的 CNTK 构建版本安装到带 Conda 的深度学习 AMI 上的任一或两个 CNTK Conda 环境中。

从每日构建安装 CNTK

1. - 对于使用 CUDA 9.0 和 MKL-DNN 的 Python 3 上的 CNTK 和 Keras 2，运行以下命令：

```
$ source activate cntk_p36
```

   - 对于使用 CUDA 9.0 和 MKL-DNN 的 Python 2 上的 CNTK 和 Keras 2，运行以下命令：

```
$ source activate cntk_p27
```

2. 其余步骤假定您使用的是 `cntk_p36` 环境。删除当前安装的 CNTK。

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

3. 要安装 CNTK 每日构建，您首先需要从 CNTK 每日构建网站查找您要安装的版本。

4. • （适用于 GPU 实例的选项）- 要安装每日构建版本，您将使用以下内容，从而在所需的构建版本中进行替换：

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

将上一条命令中的 URL 替换为您的当前 Python 环境的 GPU 版本。

• （适用于 CPU 实例的选项）- 要安装每日构建版本，您将使用以下内容，从而在所需的构建版本中进行替换：

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

将上一条命令中的 URL 替换为您的当前 Python 环境的 CPU 版本。

5. 要验证您已成功安装最新的每日构建版本，请启动 IPython 终端并检查 CNTK 的版本。

```
(cntk_p36)$ ipython
```

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

输出应类似于以下内容：`2.6-rc0.dev20181015`

更多教程

有关更多教程和示例，请参阅该框架的官方 Python 文档、适用于 CNTK 的 Python API 或 CNTK 网站。

Keras

Keras 教程

1. 要激活此框架，请在您的the section called “Conda DLAMI” CLI 上使用这些命令。

- 对于使用 CUDA 9 和 cuDNN 7 的 Python 3 上的具有 MXNet 后端的 Keras 2：

  ```
  $ source activate mxnet_p36
  ```

- 对于使用 CUDA 9 和 cuDNN 7 的 Python 2 上的具有 MXNet 后端的 Keras 2：

  ```
  $ source activate mxnet_p27
  ```

- 对于在 Python 3 上有 TensorFlow 后端的 Keras 2 和 CUDA 9 和 cudnn 7：

  ```
  $ source activate tensorflow_p36
  ```

- 对于在 Python 2 上有 TensorFlow 后端的 Keras 2 和 CUDA 9 和 cudnn 7：

  ```
  $ source activate tensorflow_p27
  ```

2. 要测试导入 Keras 以验证激活哪些后端，请使用这些命令：

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

   以下内容应显示您的屏幕上：

   ```
   Using MXNet backend
   ```

   如果 Keras 正在使用 TensorFlow，则会显示以下内容：

   ```
   Using TensorFlow backend
   ```

   如果您收到错误，或如果仍在使用错误的后端，则可以手动更新您的 Keras 配置。编辑 `~/.keras/keras.json` 文件并将后端设置更改为 `mxnet` 或 `tensorflow`。

更多教程

- 有关使用具有 MXNet 后端的 Keras 的多 GPU 教程，请尝试Keras-MXNet Multi-GPU 训练教程。

- 您可以在带 Conda 的深度学习 AMI `~/examples/keras-mxnet` 目录中查找有关具有 MXNet 后端的 Keras 的示例。
- 你可以在带有 Conda 的深度学习 AMI 目录中找到带有 TensorFlow 后端的 Kera `~/examples/keras` s 的示例。
- 有关其他教程和示例，请参阅 Keras 网站。

PyTorch

正在激活 PyTorch

当框架的稳定 Conda 程序包发布时，它会在 DLAMI 上进行测试并预安装。如果您希望运行最新的、未经测试的每日构建版本，您可以手动Install PyTorch 的夜间构建（实验版）。

要激活当前安装的框架，请按照这些有关带 Conda 的深度学习 AMI 的说明进行操作。

对于使用 PyTorch CUDA 10 和 MKL-DNN 的 Python 3，请运行以下命令：

```
$ source activate pytorch_p36
```

对于使用 PyTorch CUDA 10 和 MKL-DNN 的 Python 2，请运行以下命令：

```
$ source activate pytorch_p27
```

启动 iPython 终端。

```
(pytorch_p36)$ ipython
```

运行一个快速 PyTorch 程序。

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

您应该会看到系统输出初始随机数组，然后输出大小，然后添加另一个随机数组。

Install PyTorch 的夜间构建（实验版）

如何 PyTorch 从夜间版本中安装

您可以使用 Conda 将最新 PyTorch 版本安装到深度学习 AMI 上的任一或两个 PyTorch Conda 环境中。

1. • （Python 3 的选项）-激活 Python 3 PyTorch 环境：

   ```
   $ source activate pytorch_p36
   ```

   • （Python 2 的选项）-激活 Python 2 PyTorch 环境：

   ```
   $ source activate pytorch_p27
   ```

2. 其余步骤假定您使用的是 `pytorch_p36` 环境。移除当前安装的 PyTorch：

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

3. • （GPU 实例的选项）-使用 CUDA 10.0 安装最新的夜间版本： PyTorch

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

   • （CPU 实例的选项）-为没有 GPU PyTorch 的实例安装最新的夜间版本：

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

4. 要验证您是否已成功安装最新的夜间版本，请启动 IPython 终端并检查的版本。 PyTorch

   ```
   (pytorch_p36)$ ipython
   ```

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

   输出应类似于以下内容：`1.0.0.dev20180922`

5. 要验证 PyTorch 夜间版本是否与 MNIST 示例配合使用，您可以从 PyTorch的示例存储库中运行测试脚本：

```
(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
```

更多教程

您可以在 DLAMI 主目录中的带 Conda 的深度学习 AMI 教程文件夹中找到更多教程。有关更多教程和示例，请参阅该框架的官方PyTorch 文档、文档和PyTorch网站。

- PyTorch 到 ONNX 到 MXnet 教程
- PyTorch 到 ONNX 到 CNTK 教程

TensorFlow

正在激活 TensorFlow

本教程介绍如何在运行带有 Conda 的深度学习 AMI（Conda TensorFlow 上的 DLAMI）的实例上激活并运行程序。 TensorFlow

当框架的稳定 Conda 程序包发布时，它会在 DLAMI 上进行测试并预安装。如果您希望运行最新的、未经测试的每日构建版本，您可以手动Install TensorFlow 的夜间构建（实验版）。

和 Cond TensorFlow a 一起在 DLAMI 上跑步

1. 要激活 TensorFlow，请使用 Conda 打开 DLAMI 的亚马逊弹性计算云 (Amazon EC2) 实例。
   - 对于带有 CUDA 9.0 TensorFlow 和 MKL-DNN 的 Python 3 上的 Keras 2，请运行以下命令：

     ```
     $ source activate tensorflow_p36
     ```

   - 对于带有 CUDA 9.0 TensorFlow 和 MKL-DNN 的 Python 2 上的 Keras 2，请运行以下命令：

     ```
     $ source activate tensorflow_p27
     ```

2. 启动 iPython 终端：

   ```
   (tensorflow_p36)$ ipython
   ```

3. 运行一个 TensorFlow 程序以验证它是否正常运行：

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

`Hello, TensorFlow!` 应显示在您的屏幕上。

Install TensorFlow 的夜间构建（实验版）

您可以使用 Conda 将最新 TensorFlow 版本安装到深度学习 AMI 上的任一或两个 TensorFlow Conda 环境中。

TensorFlow 从夜间版本中安装

1. - 对于 Python 3 TensorFlow 环境，请运行以下命令：

     ```
     $ source activate tensorflow_p36
     ```

   - 对于 Python 2 TensorFlow 环境，请运行以下命令：

     ```
     $ source activate tensorflow_p27
     ```

2. 移除当前安装的 TensorFlow。

   > **Note**
   >
   > 其余步骤假定您使用的是 `tensorflow_p36` 环境。

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

3. 安装最新的夜间版本。 TensorFlow

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

4. 要验证您是否已成功安装最新的夜间版本，请启动 IPython 终端并检查的版本。 TensorFlow

```
(tensorflow_p36)$ ipython
```

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

输出应类似于以下内容：1.12.0-dev20181012

更多教程

TensorFlow 和 Horovod

TensorBoard

TensorFlow 上菜

有关教程，请参阅 DLAMI 的主目录中名为 Deep Learning AMI with Conda tutorials 的文件夹。

有关更多教程和示例，请参阅 TensorFlow Python API 的 TensorFlow 文档或TensorFlow访问网站。

TensorFlow 2

本教程介绍如何在运行带有 Conda 的深度学习 AMI（Conda 上的 DLAMI）的实例上激活 TensorFlow 2 并运行 2 程序。 TensorFlow

当框架的稳定 Conda 程序包发布时，它会在 DLAMI 上进行测试并预安装。如果您希望运行最新的、未经测试的每日构建版本，您可以手动安装 TensorFlow 2 的 Nightly Build（实验版）。

正在激活 TensorFlow 2

和 Cond TensorFlow a 一起在 DLAMI 上跑步

1. 要激活 TensorFlow 2，请使用 Conda 打开 DLAMI 的亚马逊弹性计算云 (Amazon EC2) 实例。
2. 对于使用 TensorFlow CUDA 10.1 和 MKL-DNN 的 Python 3 上的 2 和 Keras 2，请运行以下命令：

```
$ source activate tensorflow2_p36
```

3. 启动 iPython 终端：

```
(tensorflow2_p36)$ ipython
```

4. 运行 TensorFlow 2 程序以验证它是否正常运行：

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

   `Hello, TensorFlow!` 应显示在您的屏幕上。

安装 TensorFlow 2 的 Nightly Build（实验版）

你可以使用 Conda 将最新的 TensorFlow 2 个版本安装到 TensorFlow 2 个 Conda 环境中的一个或两个环境中。

TensorFlow 从夜间版本中安装

1. 对于 Python 3 TensorFlow 2 环境，请运行以下命令：

```
$ source activate tensorflow2_p36
```

2. 移除当前安装的 TensorFlow。

> **Note**
>
> 其余步骤假定您使用的是 `tensorflow2_p36` 环境。

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

3. 安装最新的夜间版本。 TensorFlow

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

4. 要验证您是否已成功安装最新的夜间版本，请启动 IPython 终端并检查的版本。 TensorFlow

```
(tensorflow2_p36)$ ipython
```

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

输出应类似于以下内容：2.1.0-dev20191122

更多教程

有关教程，请参阅 DLAMI 的主目录中名为 Deep Learning AMI with Conda tutorials 的文件夹。

有关更多教程和示例，请参阅 TensorFlow Python API 的 TensorFlow 文档或TensorFlow访问网站。

TensorFlow 和 Horovod

本教程展示了如何在（ AWS Deep Learning AMI DLAMI）上 TensorFlow 使用 Horovod 激活 Conda。Horovod 已预先安装在 Conda 环境中，用于。 TensorFlow推荐使用 Python3 环境。

> **Note**
>
> 仅支持 P3.*、P2.* 和 G3.* 实例类型。

使用 Conda 在 DLAMI 上激活 TensorFlow 和测试 Horovod

1. 打开带 Conda 的 DLAMI 的 Amazon Elastic Compute Cloud (Amazon EC2) 实例。有关 DLAMI 入门帮助，请参阅 the section called "如何开始使用 DLAMI"。
2. （推荐）对于在带有 CUDA 11 的 Python 3 上使用 Horovod 的 TensorFlow 1.15，请运行以下命令：

```
$ source activate tensorflow_p37
```

3. 启动 iPython 终端：

```
(tensorflow_p37)$ ipython
```

4. TensorFlow 使用 Horovod 测试导入以验证其是否正常运行：

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

以下内容可能显示在您的屏幕上（您可能会忽略任何警告消息）。

```

--------------------------------------------------------------------------
[[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.
--------------------------------------------------------------------------

```

更多信息

- TensorFlow 和 Horovod
- 有关教程，请参阅 DLAMI 的主目录中的 `examples/horovod` 文件夹。
- 有关更多教程和示例，请参阅 Horovod 项目 GitHub 。

TensorFlow 2 和 Horovod

本教程展示了如何使用 Horovod 在（ AWS Deep Learning AMI DLAMI ）上使用 Conda 激活 TensorFlow 2。Horovod 已预装在 Conda 环境中，适用于 2. TensorFlow 推荐使用 Python3 环境。

> **Note**
>
> 仅支持 P3.*、P2.* 和 G3.* 实例类型。

激活 TensorFlow 2 然后用 Conda 在 DLAMI 上测试 Horovod

1. 打开带 Conda 的 DLAMI 的 Amazon Elastic Compute Cloud (Amazon EC2) 实例。有关 DLAMI 入门帮助，请参阅 the section called “如何开始使用 DLAMI”。

- （推荐）对于在使用 CUDA 10 的 Python 3 上使用 Horovod 的 TensorFlow 2，请运行以下命令：

```
$ source activate tensorflow2_p36
```

- 对于在 Python TensorFlow 2 上使用 Horovod 的 2，使用 CUDA 10，请运行以下命令：

```
$ source activate tensorflow2_p27
```

2. 启动 iPython 终端：

```
(tensorflow2_p36)$ ipython
```

3. 使用 Horovod 测试导入 TensorFlow 2 以验证其是否正常运行：

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

如果没有收到任何输出，表示 Horovod 工作正常。以下内容可能显示在您的屏幕上（您可能会忽略任何警告消息）。

```
--------------------------------------------------------------------------
[[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.
--------------------------------------------------------------------------
```

更多信息

- 有关教程，请参阅 DLAMI 的主目录中的 `examples/horovod` 文件夹。

- 有关更多教程和示例，请参阅 Horovod 项目 GitHub 。

Theano

Theano 教程

> Note
>
> 从 v28 版本开始， AWS Deep Learning AMI 中将不再包含 CNTK、Caffe、Caffe2 和 Theano Conda 环境。包含这些环境 AWS Deep Learning AMI 的先前版本将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

要激活框架，请按照这些有关带 Conda 的深度学习 AMI 的说明进行操作。

对于使用 CUDA 9 和 cuDNN 7 的 Python 3 中的 Theano + Keras：

```
$ source activate theano_p36
```

对于使用 CUDA 9 和 cuDNN 7 的 Python 2 中的 Theano + Keras：

```
$ source activate theano_p27
```

启动 iPython 终端。

```
(theano_p36)$ ipython
```

运行快速 Theano 程序。

```
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)
```

您应该会看到 Theano 计算符号梯度。

更多教程

如需查看更多教程和示例，请参阅该框架的官方文档、Theano Python API 和 Theano 网站。

调试和可视化

了解适用于 DLAMI 的调试和可视化选项。单击其中一个选项可了解如何使用该选项。

主题

- MXBoard
- TensorBoard

MXBoard

MxBoard 允许您使用 TensorBoard 该软件直观地检查和解释您的 MXnet 运行和图表。它运行了一个 Web 服务器，该服务器提供了一个用于查看 MXBoard 可视化并与之交互的网页。

MxNet 和 MxBoard 预装了带有 Conda 的深度学习 AMI（带有 Conda 的 DLAMI）。 TensorBoard在本教程中，您将使用 MxBoard 函数生成与 TensorBoard兼容的日志。

主题

- 将 MXNet 与 MXBoard 结合使用
- 更多信息

将 MXNet 与 MXBoard 结合使用

生成兼容 MxBoard 日志数据 TensorBoard

1. 连接到带 Conda 的 DLAMI 的 Amazon Elastic Compute Cloud (Amazon EC2) 实例。
2. 激活 Python 3 MXNet 环境。

```
$ source activate mxnet_p36
```

3. 准备 Python 脚本以将一般运算符生成的数据写入事件文件中。数据生成 10 次，标准偏差减小，然后每次将写入到事件文件中。您将看到数据分布逐渐以平均值为中心。请注意，您将在日志文件夹中指定事件文件。您将此文件夹路径传递给 TensorBoard 二进制文件。

```
$ vi mxboard_normal.py
```

4. 将以下内容粘贴到文件中并保存它：

```
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. 运行 脚本。这将在 `logs` 文件夹中生成可用于可视化的日志。

```
$ python mxboard_normal.py
```

6. 现在，您必须切换到要使用的 TensorFlow 环境 TensorBoard 和 MxBoard 才能可视化日志。这是 MxBoard 和 TensorBoard的必需依赖项。

```
$ source activate tensorflow_p36
```

7. 将日志的位置传递到 `tensorboard`：

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

   TensorBoard 在端口 8888 上启动可视化 Web 服务器。

8. 为了方便从您的本地浏览器进行访问，您可以将 Web 服务器端口更改为端口 80 或其他端口。无论您使用哪个端口，都需要在 EC2 安全组中为您的 DLAMI 打开此端口。您还可以使用端口转发。有关更改安全组设置和端口转发的说明，请参阅设置 Jupyter Notebook 服务器。默认设置如下一步中所述。

> Note
>
> 如果您需要同时运行 Jupyter 服务器和 MXBoard 服务器，请对每个服务器使用不同的端口。

9. 在您的 EC2 实例上打开端口 8888 (或您分配给可视化 Web 服务器的端口)。

   a. 在 Amazon EC2 控制台中打开您的 EC2 实例，网址为：https://console.aws.amazon.com/ec2/。

   b. 在 Amazon EC2 控制台中，选择网络与安全，然后选择安全组。

   c. 对于安全组，选择最近创建的一个安全组（请参阅描述中的时间戳）。

   d. 选择入站选项卡，然后选择编辑。

   e. 选择添加规则。

   f. 在新行中，键入以下内容：

      类型：自定义 **TCP Rule**

      协议：**TCP**

      端口范围：**8888**（或您分配给可视化服务器的端口）

      源：**Custom IP (specify address/range)**

10. 如果您需要从本地浏览器可视化数据，请键入以下命令以将 EC2 实例上渲染的数据转发到本地计算机。

```
$ ssh -Y -L localhost:8888:localhost:8888 user_id@ec2_instance_ip
```

11. 使用运行带 Conda 的 DLAMI 的 EC2 实例的公有 IP 或 DNS 地址以及您为 MXBoard 打开的端口，以便打开用于 MXBoard 可视化的网页：

    http://127.0.0.1:8888

更多信息

要了解有关 MXBoard 的更多信息，请参阅 MXBoard 网站。

TensorBoard

TensorBoard允许您直观地检查和解释您的 TensorFlow 运行和图表。它运行一个 Web 服务器，该服务器提供一个用于查看 TensorBoard 可视化效果并与之交互的网页。

TensorFlow 并且预装了带 TensorBoard 有 Conda 的深度学习 AMI（带有 Conda 的 DLAMI）。带有 Conda 的 DLAMI 还包括一个示例脚本，该脚本 TensorFlow 用于训练启用了额外日志功能的 MNIST

模型。MNIST 是通常用于训练图像识别模型的手写编号的数据库。在本教程中，您将使用脚本来训练 MNIST 模型， TensorBoard 并使用日志来创建可视化效果。

主题

- 训练 MNIST 模型并使用可视化训练 TensorBoard
- 更多信息

训练 MNIST 模型并使用可视化训练 TensorBoard

使用可视化 MNIST 模型训练 TensorBoard

1. 连接到带 Conda 的 DLAMI 的 Amazon Elastic Compute Cloud (Amazon EC2) 实例。
2. 激活 Python 2.7 TensorFlow 环境并导航到包含 TensorBoard 示例脚本的文件夹的目录：

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

3. 运行训练启用了延长日志记录的 MNIST 模型的脚本：

```
$ python mnist_with_summaries.py
```

该脚本将日志写入 `/tmp/tensorflow/mnist`。

4. 将日志的位置传递到 `tensorboard`：

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

TensorBoard 在端口 6006 上启动可视化 Web 服务器。

5. 为了方便从您的本地浏览器进行访问，您可以将 Web 服务器端口更改为端口 80 或其他端口。无论您使用哪个端口，都需要在 EC2 安全组中为您的 DLAMI 打开此端口。您还可以使用端口转发。有关更改安全组设置和端口转发的说明，请参阅设置 Jupyter Notebook 服务器。默认设置如下一步中所述。

> **Note**
>
> 如果您需要同时运行 Jupyter 服务器和服务器，请为 TensorBoard 每台服务器使用不同的端口。

6. 在您的 EC2 实例上打开端口 6006 (或您分配给可视化 Web 服务器的端口)。

   a. 在 Amazon EC2 控制台中打开您的 EC2 实例，网址为：https://console.aws.amazon.com/ec2/。

   b. 在 Amazon EC2 控制台中，选择网络与安全，然后选择安全组。

   c. 对于安全组，选择最近创建的一个安全组（请参阅描述中的时间戳）。

   d. 选择入站选项卡，然后选择编辑。

   e. 选择添加规则。

   f. 在新行中，键入以下内容：

      类型：自定义 **TCP Rule**

      协议：**TCP**

      端口范围：**6006**（或您分配给可视化服务器的端口）

      源：**Custom IP (specify address/range)**

7. 使用使用 Conda 运行 DLAMI 的 EC2 实例的公有 IP 或 DNS 地址以及您打开的端口，打开 TensorBoard 可视化效果的网页： TensorBoard

   http:// ***YourInstancePublicDNS*****:6006**

更多信息

要了解更多信息 TensorBoard，请TensorBoard访问网站。

分布式训练

了解 DLAMI 中用于训练多个 GPU 的选项。要提高性能，请参阅Elastic Fabric Adapter单击其中一个选项可了解如何使用该选项。

主题

- Chainer
- 带 MXNet 的 Keras
- TensorFlow 和 Horovod

Chainer

从 v28 版本开始，我们将不再在 AWS Deep Learning AMI 中包含 Chainer Conda 环境。包含这些环境 AWS Deep Learning AMI 的先前版本将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

Chainer 是一种基于 Python 的灵活框架，用于轻松直观地编写复杂的神经网络架构。利用 Chainer，您可以轻松使用多 GPU 实例进行训练。Chainer 还会自动记录结果、图表损失和精度并生成用于使用计算图来可视化神经网络的输出。它包含在带 Conda 的深度学习 AMI（带 Conda 的 DLAMI）中。

以下主题介绍如何在多个 GPU、单个 GPU 和一个 CPU 上进行训练，如何创建可视化以及如何测试您的 Chainer 安装。

主题

- 使用 Chainer 训练模型
- 使用 Chainer 在多个 GPU 上训练
- 使用 Chainer 在单个 GPU 上训练
- 使用 Chainer 在 CPU 上训练
- 绘制结果
- 测试 Chainer
- 更多信息

使用 Chainer 训练模型

本教程介绍如何使用示例 Chainer 脚本来通过 MNIST 数据集训练模型。MNIST 是通常用于训练图像识别模型的手写编号的数据库。本教程还将介绍在一个 CPU 上训练与在一个或多个 GPU 上训练之间的训练速度差异。

使用 Chainer 在多个 GPU 上训练

在多个 GPU 上训练

1. 连接到运行带 Conda 的深度学习 AMI 的实例。有关如何选择或连接到实例，请参阅 the section called “实例选择” 或 Amazon EC2 文档。要运行此教程，您将需要使用带至少两个 GPU 的实例。

2. 激活 Python 3 Chainer 环境：

```
$ source activate chainer_p36
```

3. 要获取最新教程，请克隆 Chainer 存储库并导航到示例文件夹：

```
(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. 在 `train_mnist_data_parallel.py` 脚本中运行示例。默认情况下，该脚本使用在带 Conda 的深度学习 AMI 的实例上运行的 GPU。该脚本最多可在两个 GPU 上运行。它将忽略前两个 GPU 之后的所有 GPU。它会自动检测其中一个 GPU 或检测到这两个 GPU。如果您运行的是不带 GPU 的实例，请跳到本教程后面的使用 Chainer 在 CPU 上训练。

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

> **Note**
>
> 由于包含未在 DLAMI 中包含 beta 功能，此示例将返回以下错误。
>
> `chainerx ModuleNotFoundError: No module named 'chainerx'`

当 Chainer 脚本使用 MNIST 数据库训练模型时，您会看到每个纪元的结果。

然后，您会在脚本运行时看到示例输出。以下示例输出是在 p3.8xlarge 实例上运行的。该脚本的输出显示“GPU: 0, 1”，这表示它正在使用 4 个可用 GPU 中的前两个。这些脚本通常使用的是以零开头而不是以总计数开头的 GPU 索引。

```
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. 当您的训练正在运行时，查看您的 GPU 利用率会很有用。您可以验证哪些 GPU 处于活动状态并查看它们的负载。NVIDIA 为此提供了一种工具，可使用命令 `nvidia-smi` 运行该工具。但是，

它只会为您提供利用率的快照，因此，将该工具与 Linux 命令 watch 结合使用更具参考性。以下命令将 watch 与 nvidia-smi 结合使用，以便以十分之一秒的时间间隔刷新当前 GPU 利用率。打开指向您的 DLAMI 的另一个终端会话，然后运行以下命令：

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

您将看到一个类似于以下结果的输出。使用 ctrl-c 关闭该工具，或者在您在第一个终端会话中试用其他示例的同时保持其运行。

```
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 |
+-----------------------------------------------------------------------------+
```

在本示例中，GPU 0 和 GPU 1 处于活动状态，而 GPU 2 和 3 不处于活动状态。您还可以查看每个 GPU 的内存利用率。

6. 在训练完成后，记下您的第一个终端会话的已用时间。在本示例中，已用时间为 38.1389 秒。

使用 Chainer 在单个 GPU 上训练

本示例介绍如何在单个 GPU 上训练。如果您只有一个 GPU 可用或者只是想了解多 GPU 训练如何利用 Chainer 进行扩展，则可以执行此操作。

使用 Chainer 在单个 GPU 上训练

- 在本示例中，您使用另一个脚本 `train_mnist.py`，并指示它仅使用带 `--gpu=0` 参数的 GPU 0。要查看不同的 GPU 如何在 `nvidia-smi` 控制台中激活，您可以通过使用 `--gpu=1` 来指示脚本使用 GPU 编号 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
```

在本示例中，在单个 GPU 上运行花费了将近两倍的时间！训练较大的模型或较大的数据集将产生不同于本示例的结果，因此，请通过试验来进一步评估 GPU 性能。

使用 Chainer 在 CPU 上训练

现在尝试在仅 CPU 模式下训练。运行相同的脚本 `python train_mnist.py` (不带参数)：

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

在输出中，`GPU: -1` 表示未使用任何 GPU：

```
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
```

在本示例中，MNIST 在 323 秒内完成了训练，这比使用两个 GPU 时的训练的时间多 11 倍以上。如果您曾怀疑过 GPU 的能力，本示例将展示它们的效率高多少。

绘制结果

Chainer 还会自动记录结果、图表损失和精度并生成用于绘制计算图的输出。

生成计算图

1. 在任何训练运行完成之后，您可导航到 `result` 目录并查看运行的精度和损失（以两个自动生成的图像形式显示）。现在，导航到该处，然后列出内容：

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

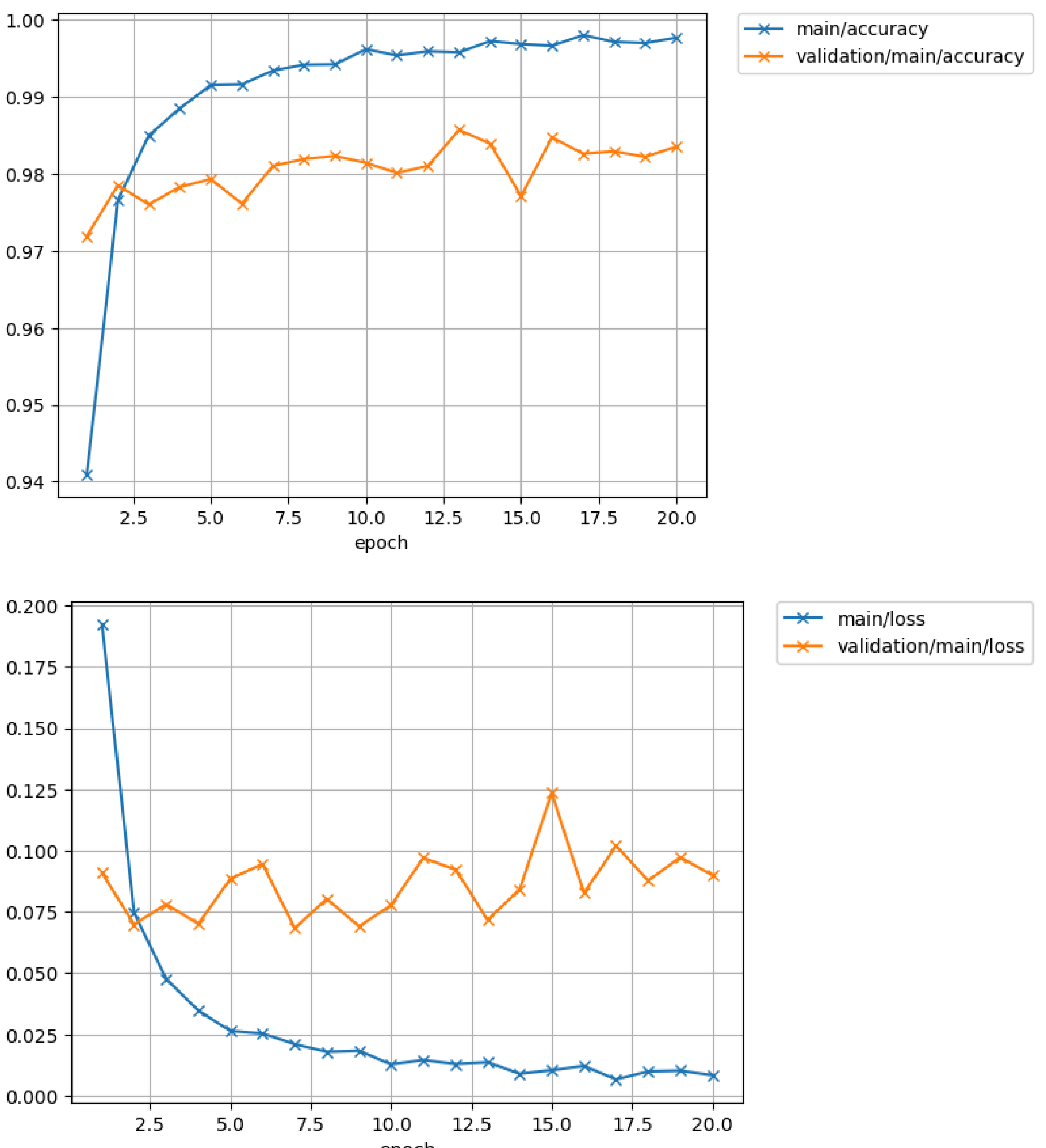
   `result` 目录包含两个 .png 格式的文件：`accuracy.png` 和 `loss.png`。

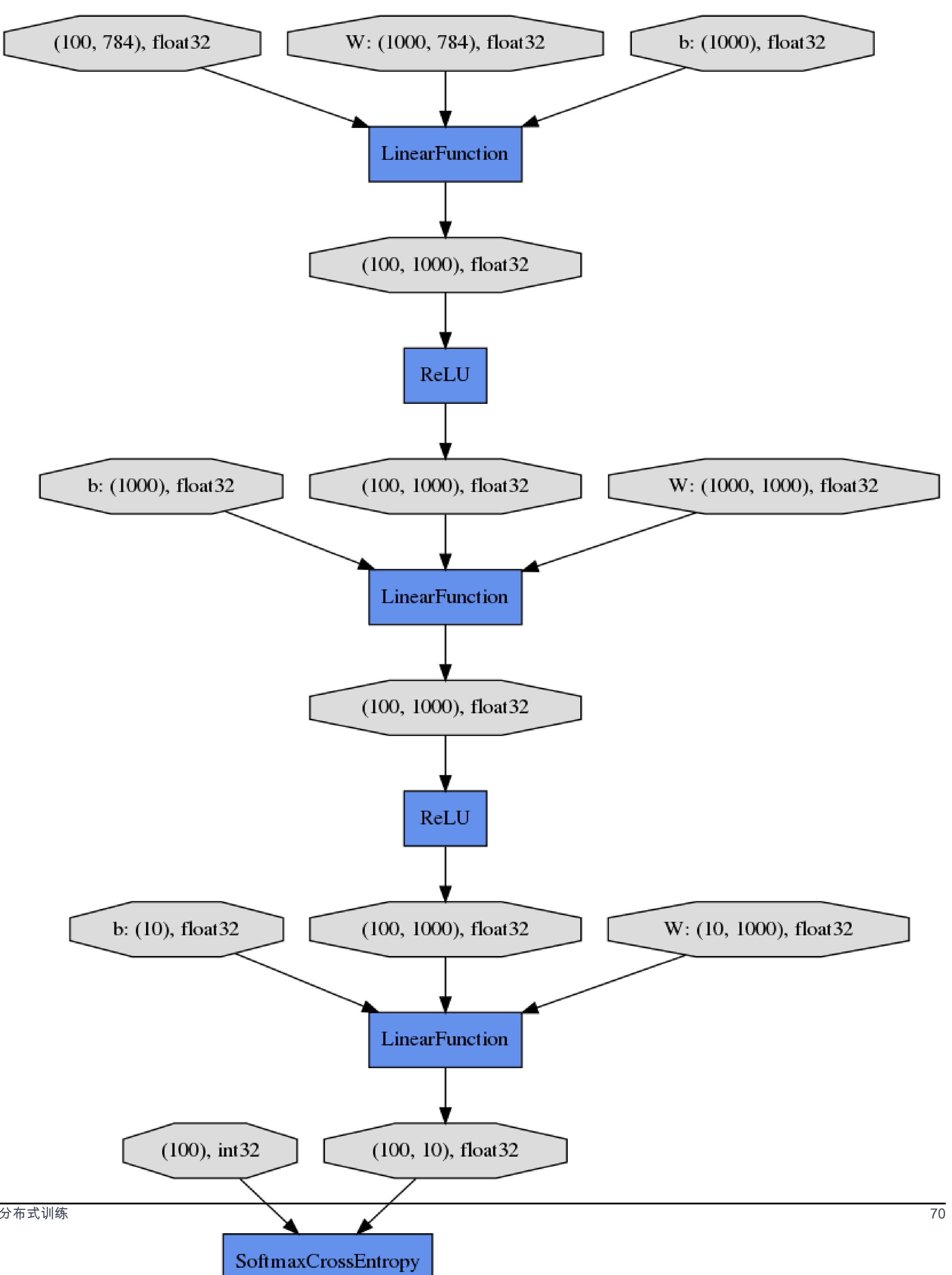
2. 要查看这些图表，请使用 `scp` 命令将它们复制到您的本地计算机。

   在 macOS 终端中，运行以下 `scp` 命令会将这三个文件全部下载到您的 `Downloads` 文件夹。将密钥文件的位置和服务器地址的占位符替换为您的信息。对于其他操作系统，请使用合适的 `scp` 命令格式。注意，对于 Amazon Linux AMI，用户名是 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
   ```

下图分别是精度图、损失图和计算图的示例。


main/accuracy
validation/main/accuracy
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/loss
validation/main/loss
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


(100, 784), float32
W: (1000, 784), float32
b: (1000), float32
LinearFunction
(100, 1000), float32
ReLU
b: (1000), float32
(100, 1000), float32
W: (1000, 1000), float32
LinearFunction
(100, 1000), float32
ReLU
b: (10), float32
(100, 1000), float32
W: (10, 1000), float32
LinearFunction
(100), int32
(100, 10), float32
SoftmaxCrossEntropy

测试 Chainer

要使用预安装的测试脚本测试 Chainer 和验证 GPU 支持，请运行以下命令：

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

这将下载 Chainer 源代码并运行 Chainer 多 GPU MNIST 示例。

更多信息

要了解有关 Chainer 的更多信息，请参阅 Chainer 文档网站。Chainer 示例文件夹包含多个示例。请试用这些示例以了解其性能。

带 MXNet 的 Keras

本教程介绍如何在带 Conda 的深度学习 AMI 上激活和使用带 MXNet 后端的 Keras 2。

在带 Conda 的深度学习 AMI 上激活并测试带 MXNet 后端的 Keras

1. 要激活带 MXNet 后端的 Keras，请打开带 Conda 的 DLAMI 的 Amazon Elastic Compute Cloud (Amazon EC2) 实例。
   - 对于 Python 3，运行以下命令：

     ```
     $ source activate mxnet_p36
     ```
   - 对于 Python 2，运行以下命令：

     ```
     $ source activate mxnet_p27
     ```
2. 启动 iPython 终端：

   ```
   (mxnet_p36)$ ipython
   ```
3. 测试导入带 MXNet 的 Keras 以验证其是否运行正常：

   ```
   import keras as k
   ```

   以下内容应显示在您的屏幕上（可能出现在一些警告消息之后）。

```
Using MXNet backend
```

Note

如果您遇到错误，或者 TensorFlow 后端仍在使用中，则需要手动更新 Keras 配置。编辑 `~/.keras/keras.json` 文件并将后端设置更改为 `mxnet`。

Keras-MXNet Multi-GPU 训练教程

训练卷积神经网络 (CNN)

1. 在 DLAMI 中打开终端和 SSH。
2. 导航到 `~/examples/keras-mxnet/`文件夹。
3. 在终端窗口中运行 `nvidia-smi` 以确定 DLAMI 上的可用 GPU 数。在下一步骤中，您将按原样运行脚本（如果您有四个 GPU）。
4. （可选）运行以下命令可打开脚本以进行编辑。

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

5. （可选）脚本具有定义 GPU 数的以下行。如果需要，请更新它。

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

6. 现在，运行训练。

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

Note

利用 `channels_first` image_data_format 设置，Keras-MXNet 的运行速度可以加快 2 倍。要更改为 `channels_first`，请编辑 Keras 配置文件 (`~/.keras/keras.json`) 并设置以下内容：`"image_data_format": "channels_first"`。
有关更多的性能优化方法，请参阅 Keras-MXNet 性能优化指南。

更多信息

- 您可以在带 Conda 的深度学习 AMI `~/examples/keras-mxnet` 目录中查找有关具有 MXNet 后端的 Keras 的示例。
- 有关更多教程和示例，请参阅 Keras-MX GitHub net 项目。

TensorFlow 和 Horovod

本教程介绍如何在带有 Conda 的深度学习 AMI 上 TensorFlow 与 Horovod 配合使用。Horovod 已预装在 Conda 环境中。 TensorFlow推荐使用 Python 3 环境。此处的说明假定您具有一个正在运行的 DLAMI 实例，此实例包含一个或多个 GPU。有关更多信息，请参阅 如何开始使用 DLAMI。

> **Note**
>
> 仅支持 P3.*、P2.* 和 G3.* 实例类型。

> **Note**
>
> 在两个位置有 mpirun（通过 OpenMPI）可用。它在 `/usr/bin` 和 `/home/ubuntu/anaconda3/envs/<env>/bin` 中可用。env 是与框架相对应的环境，例如 Tensorflow 和 Apache MXNet。在 Conda 环境中可以使用较新的 OpenMPI 版本。我们建议使用 mpirun 二进制文件的绝对路径或 -- 前缀标志来运行 mpi 工作负载。例如，对于 Tensorflow python36 环境，请使用以下任一方式：
>
> ```
> /home/ubuntu/anaconda3/envs/tensorflow_p36/bin/mpirun <args>
>
> or
>
> mpirun --prefix /home/ubuntu/anaconda3/envs/tensorflow_p36/bin <args>
> ```

使用 Horovod 激活并 TensorFlow 进行测试

1. 验证您的实例是否具有活动的 GPU。NVIDIA 为此提供了一个工具：

```
$ nvidia-smi
```

2. 激活 Python 3 TensorFlow 环境：

```
$ source activate tensorflow_p36
```

3. 启动 iPython 终端：

```
(tensorflow_p36)$ ipython
```

4. TensorFlow 使用 Horovod 测试导入以验证其是否正常运行：

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

以下内容可能显示在您的屏幕上（可能出现在一些警告消息之后）。

```


--------------------------------------------------------------------------
[[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.
--------------------------------------------------------------------------
```

配置您的 Horovod 主机文件

您可以将 Horovod 用于单节点多 GPU 训练或多节点多 GPU 训练。如果您打算使用多个节点进行分布式训练，则必须向主机文件添加每个 DLAMI 的私有 IP 地址。您当前登录的 DLAMI 称为领导。作为集群的一部分的其他 DLAMI 实例称为成员。

在开始本部分之前，请启动一个或多个 DLAMI，然后等待它们进入准备就绪状态。示例脚本需要一个主机文件，即使您计划只使用一个 DLAMI，也请创建仅具有一个条目的主机文件。如果您在训练开始后编辑主机文件，则必须重新启动训练以使已添加或删除的主机生效。

针对训练配置 Horovod

1. 将目录更改为训练脚本所在的目录。

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

2. 使用 vim 编辑领导的主目录中的文件。

```
vim hosts
```

3. 在 Amazon Elastic Compute Cloud 控制台中选择其中一个成员，控制台的说明窗格将出现。查找私有 IP 字段，并将 IP 复制并粘贴到一个文本文件中。在新行上复制每个成员的私有 IP。然后，在每个 IP 的旁边添加一个空格，然后添加文本 `slots=8`，如下所示。这表示每个实例具有的 GPU 的数目。p3.16xlarge 实例具有 8 个 GPU，因此，如果您选择其他实例类型，请提供每个实例的实际 GPU 数。对于领导，您可使用 `localhost`。对于包含 4 个节点的集群，它应类似于以下内容：

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

保存文件并退回到领导的终端。

4. 将成员实例使用的 SSH 密钥添加到 ssh 代理中。

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

5. 现在，您的领导知道如何联系每个成员。这一切都将在专用网络接口上发生。接下来，使用简短的 bash 函数来帮助将命令发送到每个成员。

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

6. 告诉其他成员不要做 “StrickHostKeyChecking”，因为这可能会导致训练停止响应。

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

使用合成数据训练

DLAMI 附带一个可用于使用合成数据训练模型的示例脚本。这将测试您的领导是否能与集群的成员通信。需要主机文件。有关说明，请参阅 配置您的 Horovod 主机文件。

使用示例数据测试 Horovod 训练

1. `~/examples/horovod/tensorflow/train_synthetic.sh` 默认为 8 个 GPU，但您可以为它提供要运行的 GPU 的数量。以下示例运行此脚本，并将 4 作为 4 个 GPU 的参数传递。

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

    在显示一些警告消息后，您将看到以下输出，其验证 Horovod 正在使用 4 个 GPU。

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

    然后，在显示一些其他警告后，您将看到表的开头和一些数据点。如果您不想查看 1000 次批处理，请退出训练。

    ```
        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 先使用所有本地 GPU，然后再尝试使用集群成员的 GPU。因此，要确保跨集群的分布式训练正常运行，请试用您计划使用的全部 GPU。例如，如果您有 4 个属于 p3.16xlarge 实例类型的成员，则集群中可包含 32 个 GPU。这是您希望试用全部 32 个 GPU 的位置。

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

    您的输出与之前的测试类似。Horovod 大小为 32，速度约为四倍。随着此实验完成，您已测试您的领导及其与成员通信的能力。如果您遇到任何问题，请查看 故障排除 部分。

准备数据 ImageNet 集

在本节中，您将下载 ImageNet 数据集，然后从原始数据集生成 TFRecord 格式的数据集。DLAMI 上为该数据集提供了一组预处理脚本，您可以将其用于 ImageNet 其中一个数据集， ImageNet 也可以用作另一个数据集的模板。还提供了为 ImageNet 配置的主要训练脚本。以下部分假定您已通过带 8 个 GPU 的 EC2 实例启动了 DLAMI。建议使用 p3.16xlarge 实例类型。

在 DLAMI 上的 `~/examples/horovod/tensorflow/utils` 目录中，您可以找到以下脚本：

- `utils/preprocess_imagenet.py`-使用它可以将原始 ImageNet数据集转换为TFRecord格式。
- `utils/tensorflow_image_resizer.py`-使用它可以按照建议调整TFRecord数据集的大小，以便进行 ImageNet 训练。

准备数据 ImageNet 集

1. 访问 image-net.org，创建账户，获取访问密钥，然后下载数据集。image-net.org 托管原始数据集。要下载它，你需要有一个 ImageNet 账户和一个访问密钥。该帐户是免费的，要获得免费访问密钥，您必须同意 ImageNet许可证。
2. 使用图像预处理脚本从原始 ImageNet 数据集生成 TFRecord 格式的数据集。从 `~/examples/horovod/tensorflow/utils` 目录中：

```
python preprocess_imagenet.py \
         --local_scratch_dir=[YOUR DIRECTORY] \
         --imagenet_username=[imagenet account] \
         --imagenet_access_key=[imagenet access key]
```

3. 使用图像调整大小脚本。如果您调整图像的大小，则训练的运行速度会更快，并且可以更好地与ResNet 参考论文保持一致。从 `~/examples/horovod/utils/preprocess` 目录中：

```
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
```

在单个 DLAMI 上训练 ResNet -50 ImageNet 模型

> **Note**
>
> - 本教程中的脚本预计经过预处理的训练数据位于 `~/data/tf-imagenet/` 文件夹中。有关说明，请参阅 准备数据 ImageNet 集。
> - 需要主机文件。有关说明，请参阅 配置您的 Horovod 主机文件。

使用 Horovod 在数据集上训练 ResNet 50 CNN ImageNet

1. 导航到 `~/examples/horovod/tensorflow`文件夹。

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

2. 验证您的配置并设置要在训练中使用的 GPU 数量。首先，查看与脚本位于相同文件夹中的 `hosts`。如果您使用的是具有少于 8 个 GPU 的实例，则必须更新此文件。默认为 `localhost slots=8`。将数量 8 更新为要使用的 GPU 的数量。
3. 提供了一个 Shell 脚本，此脚本采用您计划使用的 GPU 数作为其唯一参数。运行此脚本以开始训练。以下示例对 4 个 GPU 使用 4。

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

4. 完成此操作需要几个小时。它使用 `mpirun` 来跨您的 GPU 分布训练。

在 DLAMI 集群上训练 ResNet -50 ImageNet 模型

> **Note**
>
> - 本教程中的脚本预计经过预处理的训练数据位于 `~/data/tf-imagenet/` 文件夹中。有关说明，请参阅 准备数据 ImageNet 集。
> - 需要主机文件。有关说明，请参阅 配置您的 Horovod 主机文件。

此示例将引导您在 DLAMI 集群中的多个节点上使用准备好的数据集训练 ResNet -50 模型。

- 要获得更快的性能，建议您将数据集本地置于集群的每个成员中。

使用此 `copyclust` 函数将数据复制到其他成员。

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

或者，如果您的文件位于 S3 存储桶中，请使用 `runclust` 函数直接将文件下载到每个成员。

```
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/\""
```

使用允许您一次性管理多个节点的工具将节省大量时间。您可以等待每个步骤并单独管理每个实例，也可以使用 tmux 或 screen 等工具断开连接并恢复会话。

在复制操作完成后，您已准备好开始训练。运行脚本，并将 32 作为我们用于此次运行的 32 个 GPU 的参数传递。如果您担心断开连接并终止会话（这将结束训练运行），请使用 tmux 或类似工具。

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

以下输出是你在 32 个 GPU 上运行训练时看到 ImageNet的结果。32 个 GPU 需要 90 – 110 分钟。

```
    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
```

在训练运行完成后，脚本将跟进评估运行。它将在领导上运行，因为它运行得足够快，而不必将作业分配给其他成员。以下是评估运行的输出。

```
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
```

以下是使用 256 个 GPU 运行此脚本时的示例输出，其中运行时间为 14 到 15 分钟。

```


  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
```

故障排除

以下命令可能有助于解决您在使用 Horovod 进行实验时出现的错误。

- 如果训练因某种原因发生崩溃，则 mpirun 可能无法清理每台计算机上的所有 python 进程。在此情况下，在您开始下一个作业之前，请停止所有计算机上的 Python 进程，如下所示：

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

- 如果进程突然完成且没有错误，请尝试删除您的日志文件夹。

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

- 如果弹出其他无法解释的问题，请检查您的磁盘空间。如果您已退出，请尝试删除日志文件夹，因为其中包含所有检查点和数据。您也可以为每个成员增加卷的大小。

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

- 作为最后的手段，您也可以尝试重新启动。

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

如果您尝试在不支持的实例类型上使用 TensorFlow Horovod，则可能会收到以下错误代码：

```
---------------------------------------------------------------------------
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_9KernelDefEN4absl11strin
```

更多信息

有关实用工具和示例，请参阅 DLAMI 的主目录中的 ~/examples/horovod 文件夹。

有关更多教程和示例，请参阅 Horovod 项目 GitHub 。

Elastic Fabric Adapter

Elastic Fabric Adapter (EFA) 是一种网络设备，可以将其附加到您的 DLAMI 实例以加快高性能计算 (HPC) 应用程序的速度。借助 AWS 云提供的可扩展性、灵活性和弹性，EFA 使您能够实现本地 HPC 集群的应用程序性能。

以下主题将向您展示如何开始结合使用 EFA 与 DLAMI。

Note

从这个基础 GPU DLAMI 列表中选择您的 DLAMI

主题

- 使用 EFA 启动 AWS Deep Learning AMI 实例
- 在 DLAMI 上使用 EFA

使用 EFA 启动 AWS Deep Learning AMI 实例

最新基础 DLAMI 可随时与 EFA 结合使用，并随附所需的驱动程序、内核模块、libfabric、openmpi 和适用于 GPU 实例的 NCCL OFI 插件。

您可以在发布说明中找到基础 DLAMI 的支持 CUDA 版本。

注意：

- 在 EFA 上使用 `mpirun` 运行 NCCL 应用程序时，必须将 EFA 支持的安装的完整路径指定为：

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

- 要使您的应用程序能够使用 EFA，请将 `FI_PROVIDER="efa"` 添加到 `mpirun` 命令，如在 DLAMI 上使用 EFA中所示。

主题

- 准备 EFA 启用的安全组
- 启动实例
- 验证 EFA 附件

准备 EFA 启用的安全组

EFA 需要一个允许所有进出安全组本身的入站和出站流量的安全组。有关更多信息，请参阅 EFA 文档。

1. 通过以下网址打开 Amazon EC2 控制台：https://console.aws.amazon.com/ec2/。
2. 在导航窗格中，选择安全组，然后选择创建安全组。
3. 在创建安全组窗口中，执行以下操作：
   - 对于安全组名称，请输入一个描述性的安全组名称，例如 `EFA-enabled security group`。
   - （可选）对于描述，请输入安全组的简要描述。
   - 对于 VPC，请选择要在其中启动启用了 EFA 的实例的 VPC。
   - 选择创建。
4. 选择您创建的安全组，然后在描述 选项卡上复制组 ID。
5. 在入站和出站选项卡上，执行以下操作：

  - 选择编辑。
  - 对于类型，请选择所有流量。
  - 对于 Source，选择 Custom。
  - 将您复制的安全组 ID 粘贴到该字段中。
  - 选择保存。

6. 启用入站流量，请参考授权您 Linux 实例的入站流量。如果跳过此步骤，您将无法与您的 DLAMI 实例进行通信。

启动实例

目前 AWS Deep Learning AMI ，以下实例类型和操作系统支持上的 EFA：

- p3dn.24xLarge：亚马逊 Linux 2、Ubuntu 20.04
- p4d.24xLarge：亚马逊 Linux 2、Ubuntu 20.04
- p5.48xLarge：亚马逊 Linux 2、Ubuntu 20.04

以下部分介绍了如何启动 EFA 启用的 DLAMI 实例。有关启动 EFA 启用的 DLAMI 实例的更多信息，请参阅在集群置放群组中启动 EFA 启用的实例。

1. 通过以下网址打开 Amazon EC2 控制台：https://console.aws.amazon.com/ec2/。
2. 选择 Launch Instance（启动实例）。
3. 在选择 AMI 页面上，选择在 DLAMI 发行说明页面上找到的支持的 D LAMI
4. 在选择实例类型页面上，选择以下支持的实例类型之一，然后选择下一步：配置实例详细信息。有关支持的实例列表，请参阅此链接：EFA 和 MPI 入门
5. 在配置实例详细信息页面中，执行以下操作：

  - 对于实例的数量，请输入要启动的启用了 EFA 的实例数量。
  - 对于网络 和子网，请选择要在其中启动实例的 VPC 和子网。
  - [可选] 对于置放群组，选择向置放群组添加实例。为获得最佳性能，请在置放群组中启动实例。
  - [可选] 对于置放群组名称，选择添加到新的置放群组，输入置放群组的描述性名称，然后在置放群组策略中，选择群集。
  - 请务必在此页面上启用“Elastic Fabric Adapter”。如果禁用此选项，请将子网更改为支持所选实例类型的子网。

- 在网络接口部分中，为设备 eth0 选择新网络接口。您可以选择指定主 IPv4 地址以及一个或多个辅助 IPv4 地址。如果在具有关联的 IPv6 CIDR 块的子网中启动实例，您可以选择指定主 IPv6 地址以及一个或多个辅助 IPv6 地址。
- 选择下一步：添加存储。

6. 在添加存储页面上，除了 AMI 指定的卷（如根设备卷）以外，还要指定要附加到实例的卷，然后选择下一步：添加标签。
7. 在添加标签页面上，为实例指定标签（例如，便于用户识别的名称），然后选择下一步：配置安全组。
8. 在 “配置安全组” 页面上，在 “分配安全组” 中，选择 “选择现有安全组”，然后选择您之前创建的安全组。
9. 选择审核并启动。
10. 在核查实例启动页面上，检查这些设置，然后选择启动以选择一个密钥对并启动您的实例。

验证 EFA 附件

通过控制台

启动实例后，请在 AWS 控制台中查看实例详细信息。为此，请在 EC2 控制台中选择实例，然后查看页面下部窗格中的 Description (描述) 选项卡。找到参数“Network Interfaces: eth0”，然后单击 eth0，这将弹出一个弹出窗口。确保已启用“Elastic Fabric Adapter”。

如果未启用 EFA，您可以通过以下任一方式解决此问题：

- 终止 EC2 实例并使用相同步骤启动新实例。确保已附加 EFA。
- 将 EFA 附加到现有实例。

    1. 在 EC2 控制台中，转到“网络接口”。
    2. 单击“创建虚拟网络接口”。
    3. 选择您的实例所在的相同子网。
    4. 确保启用“Elastic Fabric Adapter”并点击“创建”。
    5. 返回“EC2 实例”选项卡并选择您的实例。
    6. 转到“操作：实例状态”并在附加 EFA 之前停止实例。
    7. 从 “操作”中，选择“网络连接：连接网络接口”。
    8. 选择您刚刚创建的界面，然后点击“附加”。
    9. 重新启动您的实例。

通过实例

以下测试脚本已存在于 DLAMI 中。运行它以确保内核模块正确加载。

```
$ fi_info -p efa
```

您的输出应类似于以下内容。

```
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
```

验证安全组配置

以下测试脚本已存在于 DLAMI 中。运行它以确保您创建的安全组配置正确。

```
$ cd /opt/amazon/efa/test/
$ ./efa_test.sh
```

您的输出应类似于以下内容。

```
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
```

如果它停止响应或未完成，请确保您的安全组具有正确的入站/出站规则。

在 DLAMI 上使用 EFA

以下部分描述如何在 AWS Deep Learning AMI上使用 EFA 来运行多节点应用程序。

使用 EFA 来运行多节点应用程序

要在节点群集中运行应用程序，需要进行以下配置

主题

- 启用无密码 SSH
- 创建主机文件
- NCCL 测试

启用无密码 SSH

选择集群中的一个节点作为领导节点。其余节点称为成员节点。

1. 在领导节点上，生成 RSA 密钥对。

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

2. 更改领导节点上私有密钥的权限。

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

3. 将公钥复制`~/.ssh/id_rsa.pub`到集群中的成员节点，并将其附加到`~/.ssh/authorized_keys`集群中的成员节点。
4. 现在，您应该能够使用私有 ip 从领导节点直接登录到成员节点。

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

5. 通过将以下内容添加到领导节点上的 ~/.ssh/config 文件中，禁用 strictHostKey检查并在领导节点上启用代理转发：

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

6. 在 Amazon Linux 2 实例上，在领导节点上运行以下命令，为配置文件提供正确的权限：

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

创建主机文件

在领导节点上，创建主机文件以标识集群中的节点。主机文件必须针对集群中的每个节点都有一个条目。创建文件 ~/hosts 并使用私有 IP 添加每个节点，如下所示：

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

NCCL 测试

> **Note**
>
> 这些测试是使用 EFA 版本 1.30.0 和 OFI NCCL Plugin 1.7.4 运行的。

下面列出了由 Nvidia 提供的 NCCL 测试子集，用于测试多个计算节点的功能和性能

支持的实例：p3dn、P4、P5

功能测试

NCCL 消息传输多节点测试

nccl_message_transfer 是一项简单的测试，可确保 NCCL OFI 插件按预期工作。该测试验证 NCCL 的连接建立和数据传输 API 的功能。当使用 EFA 来运行 NCCL 应用程序时，请确保按照示例所示使用到 mpirun 的完整路径。根据集群中实例和 GPU 的数量更改参数 np 和 N。有关更多信息，请参阅 AWS OFI NCCL 文档。

以下 nccl_message_transfer 测试适用于通用 CUDA xx.x 版本。通过替换脚本中的 CUDA 版本，您可以在您的 Amazon EC2 实例中运行适用于任何可用 CUDA 版本的命令。

```
$/opt/amazon/openmpi/bin/mpirun -n 2 -N 1 --hostfile hosts \
-x LD_LIBRARY_PATH=/usr/local/cuda-xx.x/efa/lib:/usr/local/cuda-xx.x/lib:/usr/
local/cuda-xx.x/lib64:/usr/local/cuda-xx.x:$LD_LIBRARY_PATH \
--mca btl tcp,self --mca btl_tcp_if_exclude lo,docker0 --bind-to none \
opt/aws-ofi-nccl/tests/nccl_message_transfer
```

您的输出应与以下内容类似。您可以检查输出，以查看 EFA 是否正在用作 OFI 提供程序。

```
INFO: Function: nccl_net_ofi_init Line: 1069: NET/OFI Selected Provider is efa (found 4
 nics)
INFO: Function: nccl_net_ofi_init Line: 1160: NET/OFI Using transport protocol SENDRECV
INFO: Function: configure_ep_inorder Line: 261: NET/OFI Setting
 FI_OPT_EFA_SENDRECV_IN_ORDER_ALIGNED_128_BYTES not supported.
INFO: Function: configure_nccl_proto Line: 227: NET/OFI Setting NCCL_PROTO to "simple"
INFO: Function: main Line: 86: NET/OFI Process rank 1 started. NCCLNet device used on
 ip-172-31-13-179 is AWS Libfabric.
INFO: Function: main Line: 91: NET/OFI Received 4 network devices
INFO: Function: main Line: 111: NET/OFI Network supports communication using CUDA
 buffers. Dev: 3
INFO: Function: main Line: 118: NET/OFI Server: Listening on dev 3
INFO: Function: main Line: 131: NET/OFI Send connection request to rank 1
INFO: Function: main Line: 173: NET/OFI Send connection request to rank 0
INFO: Function: main Line: 137: NET/OFI Server: Start accepting requests
INFO: Function: main Line: 141: NET/OFI Successfully accepted connection from rank 1
INFO: Function: main Line: 145: NET/OFI Send 8 requests to rank 1
INFO: Function: main Line: 179: NET/OFI Server: Start accepting requests
INFO: Function: main Line: 183: NET/OFI Successfully accepted connection from rank 0
INFO: Function: main Line: 187: NET/OFI Rank 1 posting 8 receive buffers
INFO: Function: main Line: 161: NET/OFI Successfully sent 8 requests to rank 1
INFO: Function: main Line: 251: NET/OFI Got completions for 8 requests for rank 0
INFO: Function: main Line: 251: NET/OFI Got completions for 8 requests for rank 1
```

性能测试

P4d.24xlarge 上的多节点 NCCL 性能测试

要使用 EFA 来检查 NCCL 性能，请运行官方 NCCL-Tests 存储库中提供的标准 NCCL 性能测试。DLAMI 附带了这个已经为 CUDA XX.X 构建的测试。同样，你可以使用 EFA 运行自己的脚本。

构建您自己的脚本时，请参阅以下指南：

- 当使用 EFA 来运行 NCCL 应用程序时，按照示例所示使用到 mpirun 的完整路径。

- 根据集群中实例和 GPU 的数量更改参数 np 和 N。
- 添加 NCCL_DEBUG=INFO 标志，并确保日志将 EFA 用法指示为“所选提供程序是 EFA”。
- 设置要解析的训练日志位置以进行验证

```
TRAINING_LOG="testEFA_$(date +"%N").log"
```

在任何成员节点上使用 `watch nvidia-smi` 命令来监视 GPU 使用情况。以下 `watch nvidia-smi` 命令适用于通用 CUDA xx.x 版本，并且依赖于您的实例的操作系统。通过替换脚本中的 CUDA 版本，您可以在您的 Amazon EC2 实例中运行适用于任何可用 CUDA 版本的命令。

- Amazon Linux 2：

```
 $ /opt/amazon/openmpi/bin/mpirun -n 16 -N 8 \
-x NCCL_DEBUG=INFO -x --mca pml ^cm \
-x LD_LIBRARY_PATH=/usr/local/cuda-xx.x/efa/lib:/usr/local/cuda-xx.x/lib:/usr/
local/cuda-xx.x/lib64:/usr/local/cuda-xx.x:/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 \
/usr/local/cuda-xx.x/efa/test-cuda-xx.x/all_reduce_perf -x NCCL_PROTO=simple -b 8 -e
 1G -f 2 -g 1 -c 1 -n 100 | tee ${TRAINING_LOG}
```

- Ubuntu 20.04：

```
$ /opt/amazon/openmpi/bin/mpirun -n 16 -N 8 \
-x NCCL_DEBUG=INFO -x --mca pml ^cm \
-x LD_LIBRARY_PATH=/usr/local/cuda-xx.x/efa/lib:/usr/local/cuda-xx.x/lib:/usr/
local/cuda-xx.x/lib64:/usr/local/cuda-xx.x:/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 \
/usr/local/cuda-xx.x/efa/test-cuda-xx.x/all_reduce_perf -x NCCL_PROTO=simple-b 8 -e
 1G -f 2 -g 1 -c 1 -n 100 | tee ${TRAINING_LOG}
```

您的输出应与以下内容类似：

```
# nThread 1 nGpus 1 minBytes 8 maxBytes 1073741824 step: 2(factor) warmup iters: 5
 iters: 100 agg iters: 1 validation: 1 graph: 0
#
```

```
# Using devices
#  Rank  0 Group  0 Pid   9591 on ip-172-31-4-37 device  0 [0x10] NVIDIA A100-SXM4-40GB
#  Rank  1 Group  0 Pid   9592 on ip-172-31-4-37 device  1 [0x10] NVIDIA A100-SXM4-40GB
#  Rank  2 Group  0 Pid   9593 on ip-172-31-4-37 device  2 [0x20] NVIDIA A100-SXM4-40GB
#  Rank  3 Group  0 Pid   9594 on ip-172-31-4-37 device  3 [0x20] NVIDIA A100-SXM4-40GB
#  Rank  4 Group  0 Pid   9595 on ip-172-31-4-37 device  4 [0x90] NVIDIA A100-SXM4-40GB
#  Rank  5 Group  0 Pid   9596 on ip-172-31-4-37 device  5 [0x90] NVIDIA A100-SXM4-40GB
#  Rank  6 Group  0 Pid   9597 on ip-172-31-4-37 device  6 [0xa0] NVIDIA A100-SXM4-40GB
#  Rank  7 Group  0 Pid   9598 on ip-172-31-4-37 device  7 [0xa0] NVIDIA A100-SXM4-40GB
#  Rank  8 Group  0 Pid  10216 on ip-172-31-13-179 device  0 [0x10] NVIDIA A100-
SXM4-40GB
#  Rank  9 Group  0 Pid  10217 on ip-172-31-13-179 device  1 [0x10] NVIDIA A100-
SXM4-40GB
#  Rank 10 Group  0 Pid  10218 on ip-172-31-13-179 device  2 [0x20] NVIDIA A100-
SXM4-40GB
#  Rank 11 Group  0 Pid  10219 on ip-172-31-13-179 device  3 [0x20] NVIDIA A100-
SXM4-40GB
#  Rank 12 Group  0 Pid  10220 on ip-172-31-13-179 device  4 [0x90] NVIDIA A100-
SXM4-40GB
#  Rank 13 Group  0 Pid  10221 on ip-172-31-13-179 device  5 [0x90] NVIDIA A100-
SXM4-40GB
#  Rank 14 Group  0 Pid  10222 on ip-172-31-13-179 device  6 [0xa0] NVIDIA A100-
SXM4-40GB
#  Rank 15 Group  0 Pid  10223 on ip-172-31-13-179 device  7 [0xa0] NVIDIA A100-
SXM4-40GB
ip-172-31-4-37:9591:9591 [0] NCCL INFO Bootstrap : Using ens32:172.31.4.37
ip-172-31-4-37:9591:9591 [0] NCCL INFO NET/Plugin: Failed to find ncclCollNetPlugin_v6
 symbol.
ip-172-31-4-37:9591:9591 [0] NCCL INFO NET/Plugin: Failed to find ncclCollNetPlugin
 symbol (v4 or v5).
ip-172-31-4-37:9591:9591 [0] NCCL INFO cudaDriverVersion 12020
NCCL version 2.18.5+cuda12.2
...
ip-172-31-4-37:9024:9062 [6] NCCL INFO NET/OFI Initializing aws-ofi-nccl 1.7.4-aws
ip-172-31-4-37:9020:9063 [2] NCCL INFO NET/OFI Using CUDA runtime version 11070
ip-172-31-4-37:9020:9063 [2] NCCL INFO NET/OFI Configuring AWS-specific options
ip-172-31-4-37:9024:9062 [6] NCCL INFO NET/OFI Using CUDA runtime version 11070
ip-172-31-4-37:9024:9062 [6] NCCL INFO NET/OFI Configuring AWS-specific options
ip-172-31-4-37:9024:9062 [6] NCCL INFO NET/OFI Setting provider_filter to efa
ip-172-31-4-37:9024:9062 [6] NCCL INFO NET/OFI Setting FI_EFA_FORK_SAFE environment
 variable to 1
ip-172-31-4-37:9024:9062 [6] NCCL INFO NET/OFI Disabling NVLS support due to NCCL
 version 21602
ip-172-31-4-37:9020:9063 [2] NCCL INFO NET/OFI Setting provider_filter to efa
```

```
ip-172-31-4-37:9020:9063 [2] NCCL INFO NET/OFI Setting FI_EFA_FORK_SAFE environment
 variable to 1
ip-172-31-4-37:9020:9063 [2] NCCL INFO NET/OFI Disabling NVLS support due to NCCL
 version 21602
ip-172-31-4-37:9020:9063 [2] NCCL INFO NET/OFI Running on p4d.24xlarge platform,
 Setting NCCL_TOPO_FILE environment variable to /opt/aws-ofi-nccl/share/aws-ofi-nccl/
xml/p4d-24xl-topo.xml
...
-----------------------------some output truncated-----------------------------------
#                                                                 out-of-place
           in-place
#       size         count      type   redop    root     time   algbw   busbw #wrong
  time   algbw   busbw #wrong
#        (B)    (elements)                               (us)  (GB/s)  (GB/s)
  (us)  (GB/s)  (GB/s)
           0             0     float     sum      -1    11.02    0.00    0.00      0
 11.04    0.00    0.00      0
           0             0     float     sum      -1    11.01    0.00    0.00      0
 11.00    0.00    0.00      0
           0             0     float     sum      -1    11.02    0.00    0.00      0
 11.02    0.00    0.00      0
           0             0     float     sum      -1    11.01    0.00    0.00      0
 11.00    0.00    0.00      0
           0             0     float     sum      -1    11.02    0.00    0.00      0
 11.02    0.00    0.00      0
         256             4     float     sum      -1    632.7    0.00    0.00      0
 628.2    0.00    0.00      0
         512             8     float     sum      -1    627.4    0.00    0.00      0
 629.6    0.00    0.00      0
        1024            16     float     sum      -1    632.2    0.00    0.00      0
 631.7    0.00    0.00      0
        2048            32     float     sum      -1    631.0    0.00    0.00      0
 634.2    0.00    0.00      0
        4096            64     float     sum      -1    623.3    0.01    0.01      0
 633.6    0.01    0.01      0
        8192           128     float     sum      -1    635.1    0.01    0.01      0
 633.5    0.01    0.01      0
       16384           256     float     sum      -1    634.8    0.03    0.02      0
 637.0    0.03    0.02      0
       32768           512     float     sum      -1    647.9    0.05    0.05      0
 636.8    0.05    0.05      0
       65536          1024     float     sum      -1    658.9    0.10    0.09      0
 667.0    0.10    0.09      0
```

```
         131072          2048     float     sum      -1    671.9    0.20    0.18       0
  662.9    0.20    0.19       0
         262144          4096     float     sum      -1    692.1    0.38    0.36       0
  685.1    0.38    0.36       0
         524288          8192     float     sum      -1    715.3    0.73    0.69       0
  696.6    0.75    0.71       0
        1048576         16384     float     sum      -1    734.6    1.43    1.34       0
  729.2    1.44    1.35       0
        2097152         32768     float     sum      -1    785.9    2.67    2.50       0
  794.5    2.64    2.47       0
        4194304         65536     float     sum      -1    837.2    5.01    4.70       0
  837.6    5.01    4.69       0
        8388608        131072     float     sum      -1    929.2    9.03    8.46       0
  931.4    9.01    8.44       0
       16777216        262144     float     sum      -1   1773.6    9.46    8.87       0
  1772.8    9.46    8.87       0
       33554432        524288     float     sum      -1   2110.2   15.90   14.91       0
  2116.1   15.86   14.87       0
       67108864       1048576     float     sum      -1   2650.9   25.32   23.73       0
  2658.1   25.25   23.67       0
      134217728       2097152     float     sum      -1   3943.1   34.04   31.91       0
  3945.9   34.01   31.89       0
      268435456       4194304     float     sum      -1   7216.5   37.20   34.87       0
  7178.6   37.39   35.06       0
      536870912       8388608     float     sum      -1    13680   39.24   36.79       0
  13676   39.26   36.80       0
[   1073741824      16777216     float     sum      -1    25645   41.87   39.25       0
  25497   42.11   39.48       0 ] <- Used For Benchmark
...
# Out of bounds values : 0 OK
# Avg bus bandwidth    : 7.46044
```

验证测试

要验证 EFA 测试返回的结果是否有效，请使用以下测试进行确认：

- 使用 EC2 实例元数据获取实例类型：

```
TOKEN=$(curl -X PUT "http://169.254.169.254/latest/api/token" -H "X-aws-ec2-metadata-token-ttl-seconds: 21600")
INSTANCE_TYPE=$(curl -H "X-aws-ec2-metadata-token: $TOKEN" -v http://169.254.169.254/latest/meta-data/instance-type)
```

- 运行性能测试
- 设置以下参数

```
CUDA_VERSION
CUDA_RUNTIME_VERSION
NCCL_VERSION
```

- 验证结果，如下所示：

```
RETURN_VAL=`echo $?`
if [ ${RETURN_VAL} -eq 0 ]; then

    # Information on how the version come from logs
    #
    # ip-172-31-27-205:6427:6427 [0] NCCL INFO cudaDriverVersion 12020
    # NCCL version 2.16.2+cuda11.8
    # ip-172-31-27-205:6427:6820 [0] NCCL INFO NET/OFI Initializing aws-ofi-nccl
 1.7.1-aws
    # ip-172-31-27-205:6427:6820 [0] NCCL INFO NET/OFI Using CUDA runtime version
 11060

    # cudaDriverVersion 12020  --> This is max supported cuda version by nvidia
 driver
    # NCCL version 2.16.2+cuda11.8 --> This is NCCL version compiled with cuda
 version
    # Using CUDA runtime version 11060 --> This is selected cuda version

    # Validation of logs
    grep "NET/OFI Using CUDA runtime version ${CUDA_RUNTIME_VERSION}" ${TRAINING_LOG}
 || { echo "Runtime cuda text not found"; exit 1; }
    grep "NET/OFI Initializing aws-ofi-nccl" ${TRAINING_LOG} || { echo "aws-ofi-nccl
 is not working, please check if it is installed correctly"; exit 1; }
    grep "NET/OFI Configuring AWS-specific options" ${TRAINING_LOG} || { echo "AWS-
specific options text not found"; exit 1; }
    grep "Using network AWS Libfabric" ${TRAINING_LOG} || { echo "AWS Libfabric text
 not found"; exit 1; }
    grep "busbw" ${TRAINING_LOG} || { echo "busbw text not found"; exit 1; }
    grep "Avg bus bandwidth " ${TRAINING_LOG} || { echo "Avg bus bandwidth text not
 found"; exit 1; }
    grep "NCCL version $NCCL_VERSION" ${TRAINING_LOG} || { echo "Text not found: NCCL
 version $NCCL_VERSION"; exit 1; }

    if [[ ${INSTANCE_TYPE} == "p4d.24xlarge" ]]; then
```

```
        grep "NET/AWS Libfabric/0/GDRDMA" ${TRAINING_LOG} || { echo "Text not found:
 NET/AWS Libfabric/0/GDRDMA"; exit 1; }
        grep "NET/OFI Selected Provider is efa (found 4 nics)" ${TRAINING_LOG} ||
 { echo "Selected Provider is efa text not found"; exit 1; }
        grep "aws-ofi-nccl/xml/p4d-24xl-topo.xml" ${TRAINING_LOG} || { echo "Topology
 file not found"; exit 1; }
    elif [[ ${INSTANCE_TYPE} == "p4de.24xlarge" ]]; then
        grep "NET/AWS Libfabric/0/GDRDMA" ${TRAINING_LOG} || { echo "Avg bus
 bandwidth text not found"; exit 1; }
        grep "NET/OFI Selected Provider is efa (found 4 nics)" ${TRAINING_LOG} ||
 { echo "Avg bus bandwidth text not found"; exit 1; }
        grep "aws-ofi-nccl/xml/p4de-24xl-topo.xml" ${TRAINING_LOG} || { echo
 "Topology file not found"; exit 1; }
    elif [[ ${INSTANCE_TYPE} == "p5.48xlarge" ]]; then
        grep "NET/AWS Libfabric/0/GDRDMA" ${TRAINING_LOG} || { echo "Avg bus
 bandwidth text not found"; exit 1; }
        grep "NET/OFI Selected Provider is efa (found 32 nics)" ${TRAINING_LOG} ||
 { echo "Avg bus bandwidth text not found"; exit 1; }
        grep "aws-ofi-nccl/xml/p5.48xl-topo.xml" ${TRAINING_LOG} || { echo "Topology
 file not found"; exit 1; }
    elif [[ ${INSTANCE_TYPE} == "p3dn.24xlarge" ]]; then
        grep "NET/OFI Selected Provider is efa (found 4 nics)" ${TRAINING_LOG} ||
 { echo "Selected Provider is efa text not found"; exit 1; }
    fi
    echo "****************************** check_efa_nccl_all_reduce passed for cuda
 version ${CUDA_VERSION} ******************************"
else
    echo "****************************** check_efa_nccl_all_reduce failed for cuda
 version ${CUDA_VERSION} ******************************"
fi
```

- 要访问基准数据，我们可以解析多节点 all_reduce 测试的最后一行表输出：

```
benchmark=$(sudo cat ${TRAINING_LOG} | grep '1073741824' | tail -n1 | awk -F " "
 '{{print $12}}' | sed 's/ //' | sed  's/  5e-07//')
if [[ -z "${benchmark}" ]]; then
  echo "benchmark variable is empty"
  exit 1
fi

echo "Benchmark throughput: ${benchmark}"
```

GPU 监控和优化

以下部分将指导您完成 GPU 优化和监控选项。本部分的组织方式如同一个典型工作流程一样，其中包含监控监督预处理和训练。

- 监控
  - 使用监控 GPU CloudWatch
- 优化
  - 预处理
  - 训练

监控

您的 DLAMI 已预安装多个 GPU 监控工具。本指南还将介绍可用于下载和安装的工具。

- 使用监控 GPU CloudWatch-预装的实用程序，可向 Amazon CloudWatch 报告 GPU 使用情况统计信息。
- nvidia-smi CLI — 一个监控总体 GPU 计算和内存利用率的实用工具。它已预先安装在你的 AWS Deep Learning AMI (DLAMI) 上。
- NVML C 库 - 一个基于 C 的 API，可直接访问 GPU 监控和管理功能。此项已在后台由 nvidia-smi CLI 所使用且已预安装在您的 DLAMI 上。它还具有 Python 和 Perl 绑定以方便采用这些请求进行开发。你的 DLAMI 上预装的 gpumon.py 实用程序使用的是来自的 pynvml 包。nvidia-ml-py
- NVIDIA DCGM - 一个集群管理工具。请访问开发人员页面，了解如何安装和配置此工具。

> **Tip**
>
> 请查看 NVIDIA 的开发人员博客，了解有关使用已安装在您的 DLAMI 上的 CUDA 工具的最新信息。
>
> - 使用 Nsight IDE 和 nvprof 监控 TensorCore 利用率。

使用监控 GPU CloudWatch

当您将 DLAMI 与 GPU 结合使用时，您可能会发现，您要寻找在训练或推理期间跟踪其使用率的方式。这对于优化您的数据管道以及调整深度学习网络非常有用。

有两种方法可以配置 GPU 指标 CloudWatch：

- 使用 AWS CloudWatch 代理配置指标（推荐）
- 使用预安装 gpumon.py 脚本来配置指标

使用 AWS CloudWatch 代理配置指标（推荐）

将您的 DLAMI 与 CloudWatch 统一代理集成，以配置 GPU 指标并监控 Amazon EC2 加速实例中 GPU 协进程的利用率。

使用 DLAMI 来配置 GPU 指标的方式有四种：

- 配置最低 GPU 指标
- 配置部分 GPU 指标
- 配置所有可用 GPU 指标
- 配置自定义 GPU 指标

有关更新和安全补丁的信息，请参阅 代理的安全补丁 AWS CloudWatch

先决条件

首先，您必须配置 Amazon EC2 实例 IAM 权限，以允许您的实例将指标推送到 CloudWatch。有关详细步骤，请参阅创建用于 CloudWatch 代理的 IAM 角色和用户。

配置最低 GPU 指标

使用 `dlami-cloudwatch-agent@minimal` `systemd` 服务来配置最低 GPU 指标。此服务配置以下指标：

- `utilization_gpu`
- `utilization_memory`

您可以在以下位置找到适用于最低预先配置 GPU 指标的 `systemd` 服务：

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

使用以下命令启用并启动 `systemd` 服务：

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

配置部分 GPU 指标

使用 `dlami-cloudwatch-agent@partial` systemd 服务来配置部分 GPU 指标。此服务配置以下指标：

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

您可以在以下位置找到适用于部分预先配置 GPU 指标的 systemd 服务：

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

使用以下命令启用并启动 systemd 服务：

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

配置所有可用 GPU 指标

使用 `dlami-cloudwatch-agent@all` systemd 服务来配置所有可用 GPU 指标。此服务配置以下指标：

- `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

您可以在以下位置找到适用于所有可用预先配置 GPU 指标的 systemd 服务：

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

使用以下命令启用并启动 systemd 服务：

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

配置自定义 GPU 指标

如果预配置的指标不符合您的要求，则可以创建自定义 CloudWatch 代理配置文件。

创建自定义配置文件

要创建自定义配置文件，请参阅手动创建或编辑 CloudWatch 代理配置文件中的详细步骤。

在此示例中，假设架构定义位于 /opt/aws/amazon-cloudwatch-agent/etc/amazon-cloudwatch-agent.json。

使用您的自定义文件来配置指标

运行以下命令根据您的自定义文件配置 CloudWatch 代理：

```
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
```

代理的安全补丁 AWS CloudWatch

新发布的 DLAMI 配置了最新的可用 AWS CloudWatch 代理安全补丁。请参阅以下部分，根据您选择的操作系统，使用最新安全补丁来更新您当前的DLAMI。

Amazon Linux 2

yum用于获取亚马逊 Linux 2 DLAMI 的最新 AWS CloudWatch 代理安全补丁。

```
sudo yum update
```

Ubuntu

要使用Ubuntu获取DLAMI的最新 AWS CloudWatch 安全补丁，必须 AWS CloudWatch 使用Amazon S3下载链接重新安装代理。

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

有关使用 Amazon S3 下载链接安装 AWS CloudWatch 代理的更多信息，请参阅在服务器上安装和运行 CloudWatch 代理。

使用预安装 **gpumon.py** 脚本来配置指标

一个名为 gpumon.py 的实用工具已预安装在您的 DLAMI 上。它集成 CloudWatch 并支持监控每个 GPU 的使用情况：GPU 内存、GPU 温度和 GPU 功率。该脚本会定期将监控的数据发送到 CloudWatch。您可以通过更改脚本中的一些设置来配置要发送到 CloudWatch 的数据的粒度级别。但是，在启动脚本之前，您需要进行设置 CloudWatch 才能接收指标。

如何使用设置和运行 GPU 监控 CloudWatch

1. 创建 IAM 用户，或修改现有用户以制定向其发布指标的策略 CloudWatch。如果创建新用户，请记下凭证，因为您将在下一步中需要这些凭证。

   要搜索的 IAM 策略是 “cloudwatch:PutMetricData”。要添加的策略如下所示：

   ```
   {
      "Version": "2012-10-17",
      "Statement": [
           {
   ```

```
                "Action": [
                    "cloudwatch:PutMetricData"
                 ],
                 "Effect": "Allow",
                 "Resource": "*"
            }
    ]
}
```

> **Tip**
>
> 有关创建 IAM 用户和为添加策略的更多信息 CloudWatch，请参阅 CloudWatch 文档。

2. 在您的 DLAMI 上，运行 AWS 配置并指定 IAM 用户凭证。

```
$ aws configure
```

3. 您可能需要先对 gpumon 实用工具进行一些修改，然后再运行该工具。您可以在以下代码块中定义的位置中找到 gpumon 实用工具和 README。有关 `gpumon.py` 脚本的更多信息，请参阅脚本的 Amazon S3 位置。

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

选项：

- 如果您的实例不在 us-east-1 中，请在 gpumon.py 中更改区域。
- 使用更改其他参数，例如 CloudWatch`namespace`或报告周期`store_reso`。

4. 目前，该脚本仅支持 Python 3。激活您的首选框架的 Python 3 环境或激活 DLAMI 一般 Python 3 环境。

```
$ source activate python3
```

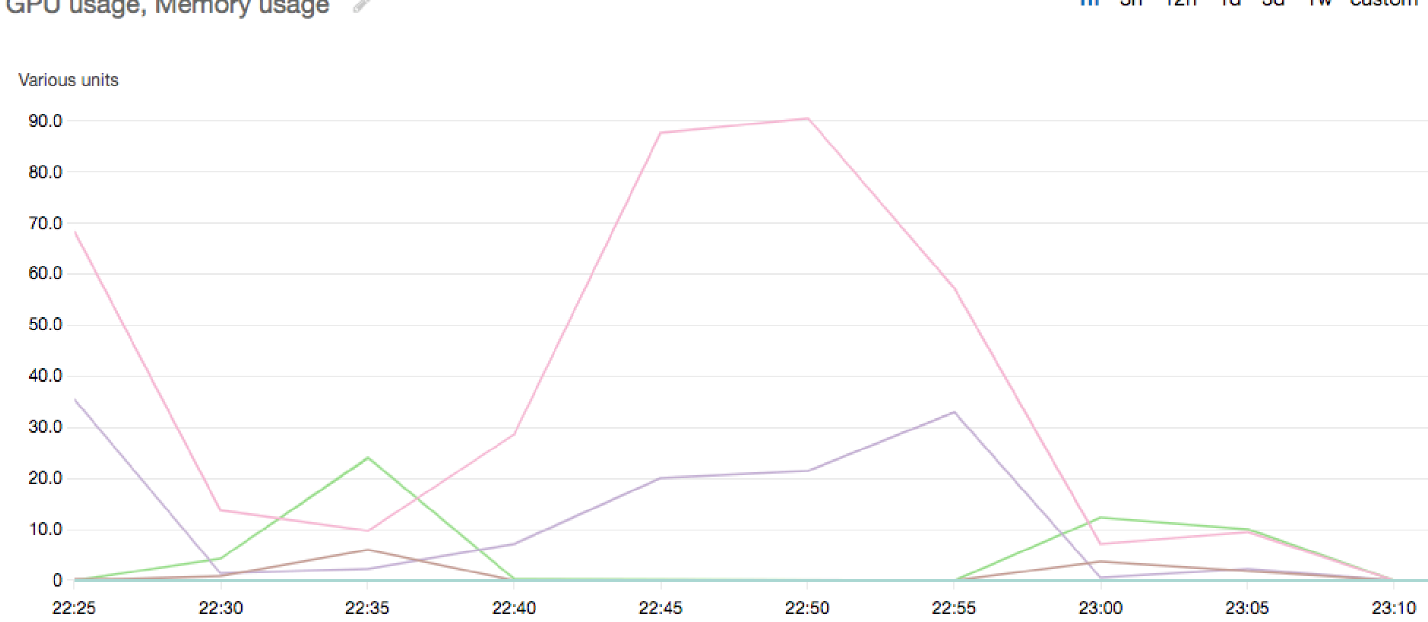
5. 在后台中运行 gpumon 实用工具。

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

6. 打开浏览器前往 https://console.aws.amazon.com/cloudwatch/，然后选择指标。它将有一个命名空间 “DeepLearningTrain”。

> **Tip**
>
> 您可以修改 gpumon.py 来更改该命名空间。您也可以通过调整 `store_reso` 来修改报告间隔。

以下是一个示例 CloudWatch 图表，报告了 gpumon.py 在监控 p2.8xlarge 实例上的训练作业的情况。



您可能对有关 GPU 监控和优化的以下其他主题感兴趣：

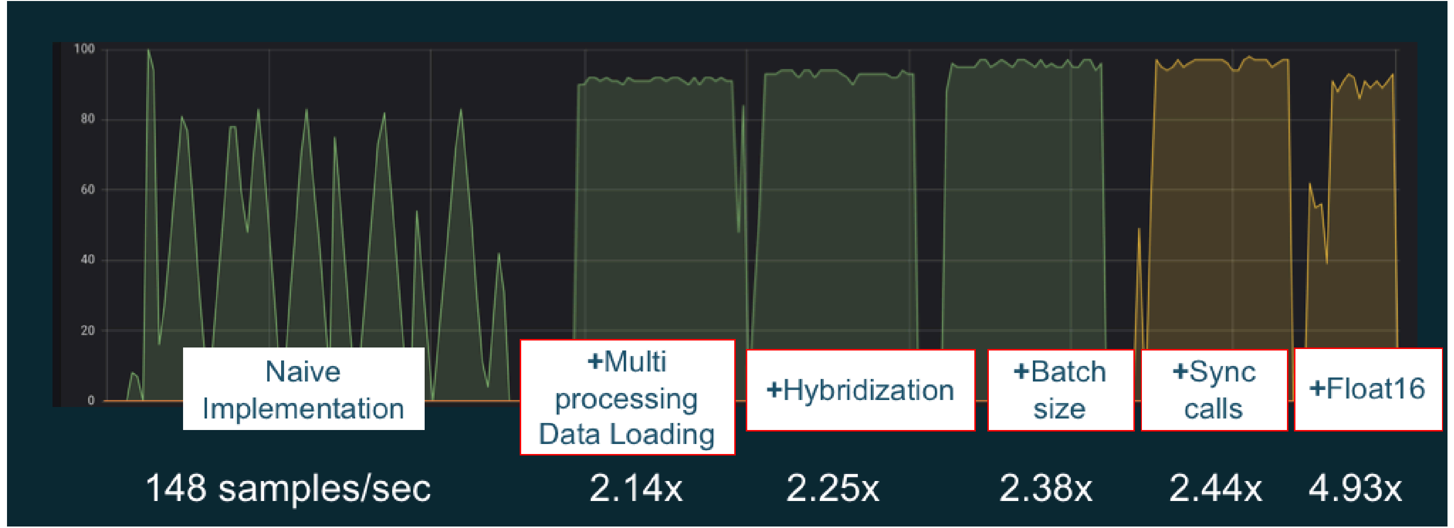
- 监控
  - 使用监控 GPU CloudWatch
- 优化
  - 预处理
  - 训练

优化

要充分利用您的 GPU，您可以优化数据管道并调整深度学习网络。如以下图表所述，神经网络的简单或基本实施对 GPU 的使用可能会不一致且不充分。当您优化预处理和数据加载时，您可以从您的 CPU 到 GPU 减少瓶颈。您可以通过使用混合化（该框架支持时）、调整批大小并同步调用来调整神

经网络本身。您也可以在大多数框架中使用多精度（float16 或 int8）训练，从而可以显著提高吞吐量。

以下图表显示应用不同优化时的累积性能提升。您的结果将取决于要处理的数据和要优化的网络。



示例 GPU 性能优化。图表来源：MXNet Gluon 的性能技巧

以下指南介绍将使用您的 DLAMI 并帮助您提升 GPU 性能的选项。

主题

- 预处理
- 训练

预处理

通过转换或扩增的数据预处理通常可以是一个绑定 CPU 的流程，而且这可以是您的整体管道中的瓶颈。框架具有用于图像处理的内置运算符，但 DALI（数据扩增库）通过框架的内置选项展示了改进的性能。

- NVIDIA 数据扩增库 (DALI)：DALI 将数据扩增卸载到 GPU。该项未预安装在 DLAMI 上，但您可以通过安装它或在您的 DLAMI 或其他 Amazon Elastic Compute Cloud 实例上加载支持的框架容器来访问它。有关详细信息，请参阅 NVIDIA 网站上的 DALI 项目页面。有关示例用例和下载代码示例，请参阅 SageMaker 预处理训练性能示例。
- nvJPEG：一个面向 C 编程人员的 GPU 加速型 JPEG 解码器库。它支持解码单个图像或批处理以及深度学习中常见的后续转换操作。nvJPEG 具有内置 DALI，或者您可以从 NVIDIA 网站的 nvjpeg 页面下载并单独使用它。

您可能对有关 GPU 监控和优化的以下其他主题感兴趣：

- 监控
  - 使用监控 GPU CloudWatch
- 优化
  - 预处理
  - 训练

训练

利用混合精度训练，您可以使用相同的内存量部署更大的网络，或者减少内存使用量（与您的单精度或双精度网络相比），并且您将看到计算性能增加。您还将受益于更小且更快的数据传输，这在多节点分布式训练中是一个重要因素。要利用混合精度训练，您需要调整数据转换和损失比例。以下是介绍如何针对支持混合精度的框架执行此操作的指南。

- NVIDIA 深度学习 SDK ——NVIDIA 网站上描述了 MXnet 的混合精度实现的文档，和。 PyTorch TensorFlow

> **Tip**
>
> 请务必针对您选择的框架检查网站，并且搜索“混合精度”或“fp16”，了解最新的优化方法。下面是可能对您有帮助的一些混合精度指南：
>
> - 混合精度训练 TensorFlow（视频）-在 NVIDIA 博客网站上。
> - 结合使用 float16 与 MXNet 进行混合精度训练 - MXNet 网站上的常见问题解答文章。
> - NVIDIA Apex：一款用于轻松进行混合精度训练的工具 PyTorch ——NVIDIA 网站上的一篇博客文章。

您可能对有关 GPU 监控和优化的以下其他主题感兴趣：

- 监控
  - 使用监控 GPU CloudWatch
- 优化
  - 预处理
  - 训练

带有 DLAMI 的 AWS 推理芯片

AWS Inferentia 是一款由其设计的自定义机器学习芯片 AWS ，可用于高性能的推理预测。 要使用该芯片，请设置亚马逊弹性计算云实例，然后使用 Ne AWS uron 软件开发套件 (SDK) 调用 Inferentia 芯片。为了向客户提供最佳 Inferentia 体验，Neuron 已内置在 AWS Deep Learning AMI (DLAMI) 中。

以下主题将向您展示如何开始将 Inferentia 与 DLAMI 结合使用。

内容

- 启动带有神经元的 DLAMI 实例 AWS
- 将 DLAMI 与神经元一起使用 AWS

启动带有神经元的 DLAMI 实例 AWS

最新的 DLAMI 已准备好 AWS 与 Inferentia 一起使用，并附带 Neuron API 包。 AWS 要启动 DLAMI 实例，请参阅启动和配置 DLAMI。获得 DLAMI 后，请使用此处的步骤确保 AWS 您的推理芯片 AWS 和 Neuron 资源处于活动状态。

内容

- 验证您的实例
- 识别 AWS 推理设备
- 查看资源使用量
- 使用 Neuron Monitor（Neuron 监视器）
- 升级 Neuron 软件

验证您的实例

在使用您的实例之前，验证该实例是否已针对 Neuron 进行正确的设置和配置。

识别 AWS 推理设备

要确定实例上的 Inferentia 设备数量，请使用以下命令：

```
neuron-ls
```

如果您的实例已附加了 Inferentia 设备，则输出将如下所示：

```
+--------+--------+--------+-----------+--------------+
| 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 |
+--------+--------+--------+-----------+--------------+
```

提供的输出取自 Inf1.6xlarge 实例，包括以下各列：

- 神经元设备：分配给的逻辑 ID。 NeuronDevice在将多个运行时配置为使用不同的 NeuronDevices运行时时，会使用此 ID。
- NEURON CORES： NeuronCores 存在于 NEURON CORES 中的 NeuronDevice数量。
- 神经元内存：中的 DRAM 内存量。 NeuronDevice
- 连接的设备：其他 NeuronDevices 连接到 NeuronDevice。
- PCI BDF：的 PCI 总线设备功能 (BDF) ID。 NeuronDevice

查看资源使用量

使用命令查看有关 NeuronCore vCPU 利用率、内存使用率、已加载模型和 Neuron 应用程序的有用信息。`neuron-top`不`neuron-top`带参数启动将显示所有使用的机器学习应用程序的数据 NeuronCores。

```
neuron-top
```

当应用程序使用四时 NeuronCores，输出应类似于下图：

有关用于监控和优化基于 Neuron 的推理应用程序的资源的更多信息，请参阅 Neuron 工具。

使用 Neuron Monitor（Neuron 监视器）

Neuron Monitor 从系统上运行的 Neuron 运行时系统收集指标，并将收集的数据以 JSON 格式流式传输到 stdout。这些指标按指标组进行组织，您可以通过提供配置文件进行配置。有关 Neuron Monitor 的更多信息，请参阅 Neuron Monitor 用户指南。

升级 Neuron 软件

有关如何在 DLAMI 中更新 Neuron SDK 软件的信息，请参阅 AWS 《神经元设置指南》。

下一个步骤

将 DLAMI 与神经元一起使用 AWS

将 DLAMI 与神经元一起使用 AWS

Ne AWS uron SDK 的典型工作流程是在编译服务器上编译之前训练过的机器学习模型。之后，将构件分发给 Inf1 实例以供执行。 AWS Deep Learning AMI (DLAMI) 预装了在使用 Inferentia 的 Inf1 实例中编译和运行推理所需的一切。

以下部分说明如何将 DLAMI 与 Inferentia 结合使用。

内容

- 使用 TensorFlow-Neuron 和 Neuron 编译器 AWS
- 使用 AWS 神经元服务 TensorFlow
- 使用 MXnet-Neuron 和 Neuron 编译器 AWS
- 使用 MXNet-Neuron 模型处理
- 使用 PyTorch-Neuron 和 Neuron 编译器 AWS

使用 TensorFlow-Neuron 和 Neuron 编译器 AWS

本教程演示如何使用 Ne AWS uron 编译器编译 Keras ResNet -50 模型并将其以格式导出为已保存的模型。 SavedModel 这种格式是典型的 TensorFlow 模型可互换格式。您还可以学习如何使用示例输入，在 Inf1 实例上运行推理过程。

有关 Neuron SDK 的更多信息，请参阅 AWS Neuron SDK 文档。

内容

- 先决条件
- 激活 Conda 环境
- Resnet50 编译
- ResNet50 推论

先决条件

使用本教程之前，您应已完成 启动带有神经元的 DLAMI 实例 AWS 中的设置步骤。您还应该熟悉深度学习知识以及如何使用 DLAMI。

激活 Conda 环境

使用以下命令激活 TensorFlow-Neuron conda 环境：

```
source activate aws_neuron_tensorflow_p36
```

要退出当前 Conda 环境，请运行以下命令：

```
source deactivate
```

Resnet50 编译

创建一个名为 **tensorflow_compile_resnet50.py** 的 Python 脚本，其中包含以下内容。此 Python 脚本编译 Keras ResNet 50 模型并将其导出为已保存的模型。

```
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')
```

使用以下命令编译该模型：

```
python tensorflow_compile_resnet50.py
```

编译过程将需要几分钟时间。完成后，您的输出应与以下内容类似：

```
...
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
...
```

编译后，保存的模型将进行压缩，放置在 **ws_resnet50/resnet50_neuron.zip** 中。使用以下命令对模型解压缩，并下载用于推理的示例图像：

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

ResNet50 推论

创建一个名为 **tensorflow_infer_resnet50.py** 的 Python 脚本，其中包含以下内容。此脚本使用先前编译的推理模型，对下载的模型运行推理过程。

```
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])
```

使用以下命令对模型运行推理过程：

```
python tensorflow_infer_resnet50.py
```

您的输出应与以下内容类似：

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

下一个步骤

[使用 AWS 神经元服务 TensorFlow](#)

使用 AWS 神经元服务 TensorFlow

本教程展示了在导出保存的模型以用 AWS 于 Serving 之前，如何构造图形并添加 Neuron 编译步骤 TensorFlow 。 TensorFlow Serving 是一种服务系统，允许您在网络上扩大推理规模。神经元 TensorFlow 服务使用与普通 TensorFlow 服务相同的 API。唯一的区别是，必须为 AWS Inferentia 编译保存的模型，并且入口点是名为的不同二进制文件。`tensorflow_model_server_neuron`二进

制文件位于 `/usr/local/bin/tensorflow_model_server_neuron` 中，并已预安装在 DLAMI 中。

有关 Neuron SDK 的更多信息，请参阅 AWS Neuron SDK 文档。

内容

- 先决条件
- 激活 Conda 环境
- 编译和导出保存的模型
- 处理保存的模型
- 生成发送给模型服务器的推理请求

先决条件

使用本教程之前，您应已完成 启动带有神经元的 DLAMI 实例 AWS 中的设置步骤。您还应该熟悉深度学习知识以及如何使用 DLAMI。

激活 Conda 环境

使用以下命令激活 TensorFlow-Neuron conda 环境：

```
source activate aws_neuron_tensorflow_p36
```

如果需要退出当前 Conda 环境，请运行：

```
source deactivate
```

编译和导出保存的模型

创建一个名 `tensorflow-model-server-compile.py` 为的 Python 脚本，其中包含以下内容。该脚本构造一个图形并使用 Neuron 对其进行编译。然后，它将编译的图形导出为保存的模型。

```
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)
```

使用以下命令编译该模型：

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

您的输出应与以下内容类似：

```
...
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
```

处理保存的模型

当模型编译完成后，您可以使用以下命令，通过 tensorflow_model_server_neuron 二进制文件处理保存的模型：

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

您的输出应与以下内容类似。编译的模型由服务器暂存在 Inferentia 设备的 DRAM 中，以准备好执行推理过程。

```
...
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 ...
```

生成发送给模型服务器的推理请求

创建一个名为 `tensorflow-model-server-infer.py` 的 Python 脚本，其中包含以下内容。该脚本通过 GRPC（这是一个服务框架）运行推理过程。

```
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))
```

通过使用 GRPC 以及以下命令，在模型上运行推理过程：

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

您的输出应与以下内容类似：

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

使用 MXnet-Neuron 和 Neuron 编译器 AWS

MXnet-Neuron 编译 API 提供了一种编译模型图的方法，您可以在 Inferentia 设备上运行该模型。AWS

在此示例中，您使用 API 编译 ResNet -50 模型并使用它来运行推理。

有关 Neuron SDK 的更多信息，请参阅 AWS Neuron SDK 文档。

内容

- 先决条件
- 激活 Conda 环境
- Resnet50 编译
- ResNet50 推论

先决条件

使用本教程之前，您应已完成 启动带有神经元的 DLAMI 实例 AWS 中的设置步骤。您还应该熟悉深度学习知识以及如何使用 DLAMI。

激活 Conda 环境

使用以下命令可激活 MXNet-Neuron Conda 环境：

```
source activate aws_neuron_mxnet_p36
```

要退出当前 Conda 环境，请运行：

```
source deactivate
```

Resnet50 编译

创建一个名为 **mxnet_compile_resnet50.py** 的 Python 脚本，其中包含以下内容。此脚本使用 mxNet-Neuron 编译 Python API 来编译 ResNet -50 模型。

```
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)
```

使用以下命令编译该模型：

```
python mxnet_compile_resnet50.py
```

编译需要几分钟。当编译完成后，以下文件将出现在您的当前目录中：

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

```
compiled_resnet50-symbol.json
```

ResNet50 推论

创建一个名为 **mxnet_infer_resnet50.py** 的 Python 脚本，其中包含以下内容。此脚本会下载一个示例映像，然后使用该映像对已编译的模型运行推理过程。

```
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]))
```

使用以下命令对已编译模型运行推理过程：

```
python mxnet_infer_resnet50.py
```

您的输出应与以下内容类似：

```
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
```

下一个步骤

使用 MXNet-Neuron 模型处理

使用 MXNet-Neuron 模型处理

在本教程中，您将学习如何使用预训练的 MXNet 模型，通过多模型服务器 (MMS) 执行实时图像分类。MMS 是一种灵活的 easy-to-use 工具，用于提供使用任何机器学习或深度学习框架训练的深度学习模型。本教程包括使用 Ne AWS uron 的编译步骤和使用 MXNet 实现彩信。

有关 Neuron SDK 的更多信息，请参阅 AWS Neuron SDK 文档。

内容

- 先决条件
- 激活 Conda 环境
- 下载示例代码
- 编译模型
- 运行推理

先决条件

使用本教程之前，您应已完成 启动带有神经元的 DLAMI 实例 AWS 中的设置步骤。您还应该熟悉深度学习知识以及如何使用 DLAMI。

激活 Conda 环境

使用以下命令可激活 MXNet-Neuron Conda 环境：

```
source activate aws_neuron_mxnet_p36
```

要退出当前 Conda 环境，请运行：

```
source deactivate
```

下载示例代码

要运行本示例，请使用以下命令下载示例代码：

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

编译模型

创建一个名为 `multi-model-server-compile.py` 的 Python 脚本，其中包含以下内容。此脚本将 ResNet 50 模型编译为 Inferentia 设备目标。

```
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)
```

要编译模型，请使用以下命令：

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

您的输出应与以下内容类似：

```
...
[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!
```

创建一个名为 `signature.json` 的文件，其中包含以下内容，以便配置输入名称和形状：

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

使用以下命令下载 `synset.txt` 文件。此文件是 ImageNet 预测类的名称列表。

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

基于 `model_server_template` 文件夹中的模板，创建自定义服务类。使用以下命令，将模板复制到您的当前工作目录中：

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

编辑 `mxnet_model_service.py` 模块，将 `mx.cpu()` 上下文替换为 `mx.neuron()` 上下文，如下所示。您还需要注释掉不必要的 `model_input` 数据副本，因为 MXNet-Neuron 不支持 NDarray 和 Gluon API。

```
...
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]
```

使用以下命令，通过模型归档程序对模型进行打包：

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

运行推理

使用以下命令启动多模型服务器并加载使用 RESTful API 的模型。确保 neuron-rtd 正在使用默认设置运行。

```
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
```

通过以下命令，使用示例图像运行推理：

```
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
```

您的输出应与以下内容类似：

```
[
```

```
  {
    "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"
  }
]
```

要执行测试后清理，请通过 RESTful API 发出删除命令，并使用以下命令停止模型服务器：

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

multi-model-server --stop
```

您应看到以下输出：

```
{
  "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
```

使用 PyTorch-Neuron 和 Neuron 编译器 AWS

PyTorch-Neuron 编译 API 提供了一种编译模型图的方法，您可以在 AWS Inferentia 设备上运行该模型。

经过训练的模型必须先编译为 Inferentia 目标，才能部署在 Inf1 实例上。以下教程编译 torchvision ResNet 50 模型并将其导出为已保存的模块。 TorchScript 此模型随后将用于运行推理。

为方便起见，本教程使用 Inf1 实例进行编译和推理。在实际操作中，您也可以使用其他实例类型来编译模型，例如 c5 实例系列。然后，您必须将已编译的模型部署到 Inf1 推理服务器中。有关更多信息，请参阅 Ne AWS uron PyTorch SDK 文档。

内容

- 先决条件
- 激活 Conda 环境
- Resnet50 编译
- ResNet50 推论

先决条件

使用本教程之前，您应已完成 启动带有神经元的 DLAMI 实例 AWS 中的设置步骤。您还应该熟悉深度学习知识以及如何使用 DLAMI。

激活 Conda 环境

使用以下命令激活 PyTorch-Neuron conda 环境：

```
source activate aws_neuron_pytorch_p36
```

要退出当前 Conda 环境，请运行：

```

source deactivate
```

Resnet50 编译

创建一个名为 **pytorch_trace_resnet50.py** 的 Python 脚本，其中包含以下内容。此脚本使用 PyTorch-Neuron 编译 Python API 来编译 ResNet -50 模型。

> **Note**
>
> 在编译 torchvision 模型时，您需要注意：torchvision 与 torch 软件包的版本之间存在依赖关系。这些依赖关系规则可以通过 pip 进行管理。Torchvision==0.6.1 与 torch==1.5.1 版本匹配，而 torchvision==0.8.2 与 torch==1.7.1 版本匹配。

```
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")
```

运行编译脚本。

```
python pytorch_trace_resnet50.py
```

编译需要几分钟。编译完成后，已编译的模型将以 `resnet50_neuron.pt` 的形式保存在本地目录中。

ResNet50 推论

创建一个名为 **`pytorch_infer_resnet50.py`** 的 Python 脚本，其中包含以下内容。此脚本会下载一个示例映像，然后使用该映像对已编译的模型运行推理过程。

```
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) )
```

使用以下命令对已编译模型运行推理过程：

```
python pytorch_infer_resnet50.py
```

您的输出应与以下内容类似：

```
Top 5 labels:
 ['tiger', 'lynx', 'tiger_cat', 'Egyptian_cat', 'tabby']
```

Graviton DLAMI

AWS Graviton GPU DLAMI 旨在为深度学习工作负载提供高性能和成本效益。具体而言，g5G 实例类型采用基于 ARM 的 G AWS raviton2 处理器，该处理器是从头开始构建的， AWS 并针对客户在云中运行工作负载的方式进行了优化。 AWS Graviton GPU dLAMIs 预先配置了 Docker、NVIDIA Docker、NVIDIA Driver、CUDA、cudnn、NCCL 和 Tensorrt，以及流行的机器学习框架，例如和。TensorFlow PyTorch

借助 G5g 实例类型，您可以利用 Graviton2 的价格和性能优势来部署基于 GPU 加速的深度学习模型，与基于 x86 的带有 GPU 加速的实例相比，成本大幅降低。

选择 Graviton DLAMI

使用您选择的 Graviton DLAMI 来启动 G5g 实例。

有关启动 DLAMI 的 step-by-step 说明，请参阅启动和配置 DLAMI。

有关最新 Graviton DLAMI 的列表，请参阅 DLAMI 发布说明。

开始使用

以下主题将向您展示如何开始使用 Graviton DLAMI。

内容

- 使用 Graviton GPU DLAMI
- 使用 Graviton GPU DLAMI TensorFlow

- 使用 Graviton GPU DLAMI PyTorch

使用 Graviton GPU DLAMI

AWS Deep Learning AMI 它已准备好与基于 Arm 处理器的 Graviton GPU 配合使用。Graviton GPU DLAMI 附带 GPU 驱动程序基础平台，以及可用来部署您自己的自定义深度学习环境的加速库。Docker 和 NVIDIA Docker 在 Graviton GPU DLAMI 上进行了预配置，允许您部署容器化应用程序。有关 Graviton GPU DLAMI 的更多详细信息，请查看发布说明。

内容

- 检查 GPU 状态
- 检查 CUDA 版本
- 验证 Docker
- TensorRT
- 运行 CUDA 示例

检查 GPU 状态

使用 NVIDIA 系统管理界面来检查 Graviton GPU 的状态。

```
nvidia-smi
```

`nvidia-smi` 命令的输出应与以下内容类似：

```
+-----------------------------------------------------------------------------+
| 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                                                 |
+-----------------------------------------------------------------------------+
```

检查 CUDA 版本

要检查 CUDA 版本，请运行以下命令：

```
/usr/local/cuda/bin/nvcc --version | grep Cuda
```

您的输出应类似于以下内容：

```
nvcc: NVIDIA (R) Cuda compiler driver
Cuda compilation tools, release 11.4, V11.4.120
```

验证 Docker

从中运行 CUDA 容器DockerHub来验证 Graviton GPU 上的 Docker 功能：

```
sudo docker run --platform=linux/arm64 --rm \
    --gpus all nvidia/cuda:11.4.2-base-ubuntu20.04 nvidia-smi
```

您的输出应类似于以下内容：

```
+-----------------------------------------------------------------------------+
| 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

使用以下命令来访问 TensorRT 命令行工具：

```
trtexec
```

您的输出应类似于以下内容：

```
&&&& RUNNING TensorRT.trtexec [TensorRT v8200] # trtexec
...
&&&& PASSED TensorRT.trtexec [TensorRT v8200] # trtexec
```

TensorRT Python 滚轮可供按需安装。您可以在以下文件位置找到这些滚轮：

```
/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
```

有关其他详细信息，请参阅 NVIDIA TensorRT 文档。

运行 CUDA 示例

Graviton GPU DLAMI 提供了预编译的 CUDA 示例，可以帮助您验证不同的 CUDA 功能。

```
ls /usr/local/cuda/compiled_samples
```

例如，使用以下命令来运行 vectorAdd 示例：

```
/usr/local/cuda/compiled_samples/vectorAdd
```

您的输出应类似于以下内容：

```
[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
```

运行 transpose 示例：

```
/usr/local/cuda/compiled_samples/transpose
```

您的输出应类似于以下内容：

```
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
```

后续步骤

使用 Graviton GPU DLAMI TensorFlow

使用 Graviton GPU DLAMI TensorFlow

AWS Deep Learning AMI 它已准备好与基于 Arm 处理器的 Graviton GPU 配合使用，并且已针对此进行了优化。 TensorFlowGraviton GPU DL TensorFlow AMI 包括一个 Python 环境，该环境预先配置了用于深度学习推理TensorFlow 用例的服务。有关Graviton GPU D TensorFlow LAMI的更多详细信息，请查看发行说明。

内容

- 验证 TensorFlow 服务可用性
- 验证 TensorFlow 和提供 TensorFlow API 可用性
- 使用服务运行示例推理 TensorFlow

验证 TensorFlow 服务可用性

运行以下命令以验证 Serving 的可用性和 TensorFlow 版本：

```
tensorflow_model_server --version
```

您的输出应类似于以下内容：

```
TensorFlow ModelServer: 0.0.0+dev.sha.3e05381e
TensorFlow Library: 2.8.0
```

验证 TensorFlow 和提供 TensorFlow API 可用性

运行以下命令来验证 TensorFlow 和 TensorFlow 服务 API 的可用性：

```
python3 -c "import tensorflow, tensorflow_serving"
```

如果命令成功，则无任何输出。

使用服务运行示例推理 TensorFlow

使用以下命令下载预训练的 ResNet 50 模型并使用 TensorFlow Serving 运行推理：

```
# 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" &
```

您的输出应类似于以下内容：

```
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 ...
```

使用 Ser TensorFlow vin `resnet_client` g 示例来运行推理：

```
python3 serving/tensorflow_serving/example/resnet_client.py
```

您的输出应类似于以下内容：

```
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
```

使用以下命令停止 TensorFlow 服务：

```
kill $(pidof tensorflow_model_server)
```

后续步骤

使用 Graviton GPU DLAMI PyTorch

使用 Graviton GPU DLAMI PyTorch

AWS Deep Learning AMI 它已准备好与基于 Arm 处理器的 Graviton GPU 配合使用，并且已针对此进行了优化。 PyTorchGraviton GPU DL PyTorch AMI 包括一个预先配置了、和的 Python 环境 TorchVision，用于深度学习训练TorchServe和推理PyTorch用例。有关Graviton GPU D PyTorch LAMI 的更多详细信息，请查看发行说明。

内容

- 验证 PyTorch Python 环境
- 使用运行训练示例 PyTorch
- 使用运行推理示例 PyTorch

验证 PyTorch Python 环境

使用以下命令来连接您的 G5g 实例并激活基础 Conda 环境：

```
source activate base
```

您的命令提示符应表明您正在基本 Conda 环境中工作，该环境包含 PyTorch TorchVision、和其他库。

```
(base) $
```

验证 PyTorch 环境的默认刀具路径：

```
(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
```

验证 Torch 和 TorchVersion 是否可用，检查其版本，并测试其基本功能：

```
(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)
```

使用运行训练示例 PyTorch

运行示例 MNIST 训练作业：

```
git clone https://github.com/pytorch/examples.git
cd examples/mnist
python main.py
```

您的输出应类似于以下内容：

```
...
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%)
```

使用运行推理示例 PyTorch

使用以下命令下载预训练的 densenet161 模型并使用以下命令运行推理： 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
```

您的输出应类似于以下内容：

```
{
  "tiger_cat": 0.4693363308906555,
```

```
  "tabby": 0.4633873701095581,
  "Egyptian_cat": 0.06456123292446136,
  "lynx": 0.0012828150065615773,
  "plastic_bag": 0.00023322898778133094
}
```

使用以下命令来注销 densenet161 模型并停止服务器：

```
curl -X DELETE http://localhost:8081/models/densenet161/1.0
torchserve --stop
```

您的输出应类似于以下内容：

```
{
  "status": "Model \"densenet161\" unregistered"
}
TorchServe has stopped.
```

Habana DLAMI

带 Habana 加速器的实例旨在为深度学习模型训练工作负载提供高性能和成本效益。具体来说，DL1 实例类型使用来自英特尔旗下公司 Habana Labs 的 Habana Gaudi 加速器。带有 Habana 加速器的实例使用 Habana SynapSeai 软件进行配置，并预先集成了流行的机器学习框架，例如和。 TensorFlow PyTorch

以下主题将向您展示如何开始将 Habana Gaudi 硬件与 DLAMI 结合使用。

内容

- 启动 Habana DLAMI

启动 Habana DLAMI

最新的 DLAMI 已准备好与 Habana Gaudi 加速器结合使用。使用以下步骤来启动 Habana DLAMI，并确保您的 Python 和框架特定资源都处于活动状态。有关更多设置资源，请参阅 Habana Gaudi 设置和安装存储库。

内容

- 选择 Habana DLAMI

- 激活 Python 环境
- 导入机器学习框架

选择 Habana DLAMI

使用您选择的 Habana DLAMI 来启动 DL1 实例。

有关启动 DLAMI 的 step-by-step 说明，请参阅启动和配置 DLAMI。

有关最新 Habana DLAMI 的列表，请参阅 DLAMI 发布说明。

激活 Python 环境

连接到您的 DL1 实例，为您的 Habana DLAMI 激活推荐的 Python 环境。要查看您的推荐 Python 环境，请在发布说明中选择您的 DLAMI。

导入机器学习框架

带有 Habana 加速器的实例已预先集成了流行的机器学习框架，例如和。 TensorFlow PyTorch导入您选择的机器学习框架。

导入 TensorFlow

要在 Habana DLAMI TensorFlow 上使用，请导航到您激活并导入的 Python 环境文件夹。 TensorFlow

```
/usr/bin/$PYTHON_VERSION
import tensorflow
tensorflow.__version__
```

要查看与你的 Habana DLAMI 兼容的 TensorFlow 版本，请在发行说明中选择你的 DLAMI。

导入 PyTorch

要在 Habana DLAMI PyTorch 上使用，请导航到您激活的 Python 环境文件夹，然后导入相应的版本。 PyTorch

```
/usr/bin/$PYTHON_VERSION
import torch
```

```
torch.__version__
```

要查看与你的 Habana DLAMI 兼容的 PyTorch 版本，请在发行说明中选择你的 DLAMI。

有关如何在 Habana DLAMI 中运行 TensorFlow 和 PyTorch 训练机器学习模型的更多信息，请参阅 Habana 模型参考资料库。 GitHub 有关使用 Habana DLAMI 的更多资源，请访问 Habana Gaudi 文档。

推理

此部分提供了有关如何使用 DLAMI 的框架和工具来运行推理的教程。

有关使用 Elastic Inference 的教程，请参阅使用 Amazon Elastic Inference

带框架的推理

- 将适用于推理的 Apache MXNet（孵化版）与 ONNX 模型结合使用
- 使用 Apache mxNet（孵化中）对 50 模型进行推理 ResNet
- 在推理中将 CNTK 与 ONNX 模型配合使用

推理工具

- 适用于 Apache MXNet 的模型服务器 (MMS)
- TensorFlow 上菜

将适用于推理的 Apache MXNet（孵化版）与 ONNX 模型结合使用

如何将适用于图像推理的 ONNX 模型与 Apache MXNet（孵化版）结合使用

1. - （适用于 Python 3 的选项）— 激活 Python 3 Apache MXNet（孵化版）环境：

     ```
     $ source activate mxnet_p36
     ```

   - （适用于 Python 2 的选项）— 激活 Python 2 Apache MXNet（孵化版）环境：

     ```
     $ source activate mxnet_p27
     ```

2. 其余步骤假定您使用的是 `mxnet_p36` 环境。
3. 下载一张哈士奇的照片。

```
$ curl -O https://upload.wikimedia.org/wikipedia/commons/b/b5/Siberian_Husky_bi-
eyed_Flickr.jpg
```

4. 下载使用此模型的类的列表。

```
$ curl -O https://gist.githubusercontent.com/yrevar/6135f1bd8dcf2e0cc683/raw/
d133d61a09d7e5a3b36b8c111a8dd5c4b5d560ee/imagenet1000_clsid_to_human.pkl
```

5. 下载 ONNX 格式的预训练的 VGG 16 模型。

```
$ wget -O vgg16.onnx https://github.com/onnx/models/raw/master/vision/
classification/vgg/model/vgg16-7.onnx
```

6. 使用您的首选文本编辑器来创建具有以下内容的脚本。此脚本将使用哈士奇的图片，从预训练模型中获得预测结果，然后在类文件中查找，并返回一个图片分类结果。

```
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. 然后运行脚本，您应看到一个如下所示的结果：

```
248
Eskimo dog, husky
```

使用 Apache mxNet（孵化中）对 50 模型进行推理 ResNet

如何将预训练的 Apache MXNet（孵化版）模型与适用于图像推理的 Symbol API 和 MXNet 结合使用

1. • （适用于 Python 3 的选项）— 激活 Python 3 Apache MXNet（孵化版）环境：

   ```
   $ source activate mxnet_p36
   ```

   • （适用于 Python 2 的选项）— 激活 Python 2 Apache MXNet（孵化版）环境：

   ```
   $ source activate mxnet_p27
   ```

2. 其余步骤假定您使用的是 mxnet_p36 环境。
3. 使用您的首选文本编辑器来创建具有以下内容的脚本。该脚本将下载 ResNet -50模型文件（resnet-50-0000.params和resnet-50-symbol.json）和标签列表（synset.txt），下载猫咪图像以从预训练模型中获取预测结果，然后在标签结果列表中查找，返回预测结果。

```
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. 然后运行脚本，您应看到一个如下所示的结果：

```
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
```

在推理中将 CNTK 与 ONNX 模型配合使用

> **Note**
>
> 从 v28 版本开始， AWS Deep Learning AMI 中将不再包含 CNTK、Caffe、Caffe2 和 Theano Conda 环境。包含这些环境 AWS Deep Learning AMI 的先前版本将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

> **Note**
>
> 本教程中使用的 VGG-16 模型会占用大量内存。选择您的 AWS Deep Learning AMI 实例时，您可能需要一个 RAM 超过 30 GB 的实例。

如何在推理中将 ONNX 模型与 CNTK 配合使用

1. - （适用于 Python 3 的选项）- 激活 Python 3 CNTK 环境：

     ```
     $ source activate cntk_p36
     ```

   - （适用于 Python 2 的选项）- 激活 Python 2 CNTK 环境：

     ```
     $ source activate cntk_p27
     ```

2. 其余步骤假定您使用的是 cntk_p36 环境。
3. 使用文本编辑器创建一个新文件，并在脚本中使用以下程序以在 CNTK 中打开 ONNX 格式文件。

   ```
   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!")
   ```

   在运行此脚本后，CNTK 将加载模型。
4. 您还可以尝试使用 CNTK 运行推理。首先，下载一张哈士奇的照片。

   ```
   $ curl -O https://upload.wikimedia.org/wikipedia/commons/b/b5/Siberian_Husky_bi-eyed_Flickr.jpg
   ```

5. 接下来，下载将使用此模型的类的列表。

```
$ curl -O https://gist.githubusercontent.com/yrevar/6135f1bd8dcf2e0cc683/raw/
d133d61a09d7e5a3b36b8c111a8dd5c4b5d560ee/imagenet1000_clsid_to_human.pkl
```

6. 编辑先前创建的脚本以包含以下内容。此新版本将使用哈士奇的图片，得到一个预测结果，然后在类文件中查找，返回一个预测结果。

```
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. 然后运行脚本，您应看到一个如下所示的结果：

```
248
Eskimo dog, husky
```

将框架用于 ONNX

对于部分框架，带 Conda 的深度学习 AMI 现在支持开放神经网络交换 (ONNX) 模型。选择下方列出的主题之一，以了解如何在带 Conda 的深度学习 AMI 上使用 ONNX。

如果要在 DLAMI 上使用现有 ONNX 模型，请参阅 将适用于推理的 Apache MXNet（孵化版）与 ONNX 模型结合使用。

关于 ONNX

开放神经网络交换 (ONNX) 是一种用于表示深度学习模型的开放格式。ONNX 受到 Amazon Web Services、Microsoft、Facebook 和其他多个合作伙伴的支持。您可以使用任何选定的框架来设计、训练和部署深度学习模型。ONNX 模型的好处是，它们可以在框架之间轻松移动。

带 Conda 的深度学习 AMI 当前的特色是支持以下教程集中的一些 ONNX 功能。

- 有关将 Apache MXNet 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 MXNet 的教程
- PyTorch 到 ONNX 到 CNTK 教程
- PyTorch 到 ONNX 到 MXnet 教程

您可能还需要参考 ONNX 项目文档和教程：

- ONNX Project 开启 GitHub
- ONNX 教程

有关将 Apache MXNet 转换为 ONNX，然后加载到 CNTK 的教程

Note

从 v28 版本开始， AWS Deep Learning AMI 中将不再包含 CNTK、Caffe、Caffe2 和 Theano Conda 环境。包含这些环境 AWS Deep Learning AMI 的先前版本将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

ONNX 概述

开放神经网络交换 (ONNX) 是一种用于表示深度学习模型的开放格式。ONNX 受到 Amazon Web Services、Microsoft、Facebook 和其他多个合作伙伴的支持。您可以使用任何选定的框架来设计、训练和部署深度学习模型。ONNX 模型的好处是，它们可以在框架之间轻松移动。

本教程介绍如何将带 Conda 的深度学习 AMI 与 ONNX 结合使用。通过执行以下步骤，您可以训练模型或从一个框架中加载预先训练的模型，将此模型导出为 ONNX，然后将此模型导入到另一个框架中。

ONNX 先决条件

要使用此 ONNX 教程，您必须有权访问带 Conda 的深度学习 AMI 版本 12 或更高版本。有关如何开始使用带 Conda 的深度学习 AMI 的更多信息，请参阅 带 Conda 的深度学习 AMI。

> **Important**
>
> 这些示例使用可能需要多达 8 GB 内存（或更多）的函数。请务必选择具有足量内存的实例类型。

使用带 Conda 的深度学习 AMI 来启动终端会话以开始以下教程。

将 Apache MXNet（孵化）模型转换为 ONNX，然后将模型加载到 CNTK 中

如何从 Apache MXNet（孵化）中导出模型

您可以将最新的 MXNet 构建版本安装到带 Conda 的深度学习 AMI 上的任一或两个 MXNet Conda 环境中。

1. - （适用于 Python 3 的选项）- 激活 Python 3 MXNet 环境：

     ```
     $ source activate mxnet_p36
     ```

   - （适用于 Python 2 的选项）- 激活 Python 2 MXNet 环境：

     ```
     $ source activate mxnet_p27
     ```

2. 其余步骤假定您使用的是 mxnet_p36 环境。
3. 下载模型文件。

   ```
   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. 要将模型文件从 MXNet 导出为 ONNX 格式，使用文本编辑器创建一个新文件，并在脚本中使用以下程序。

   ```
   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)
```

您可能会看到一些警告消息，不过您目前可以安全地忽略它们。在您运行此脚本后，您将在同一目录中看到新创建的 .onnx 文件。

5. 现在您有一个 ONNX 文件，可以将其与以下示例结合使用来尝试运行推理：

   - 在推理中将 CNTK 与 ONNX 模型配合使用

ONNX 教程

- 有关将 Apache MXNet 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 MXNet 的教程
- PyTorch 到 ONNX 到 MXnet 教程
- PyTorch 到 ONNX 到 CNTK 教程

有关将 Chainer 转换为 ONNX，然后加载到 CNTK 的教程

Note

从 v28 版本开始， AWS Deep Learning AMI 中将不再包含 CNTK、Caffe、Caffe2 和 Theano Conda 环境。包含这些环境 AWS Deep Learning AMI 的先前版本将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

ONNX 概述

开放神经网络交换 (ONNX) 是一种用于表示深度学习模型的开放格式。ONNX 受到 Amazon Web Services、Microsoft、Facebook 和其他多个合作伙伴的支持。您可以使用任何选定的框架来设计、训练和部署深度学习模型。ONNX 模型的好处是，它们可以在框架之间轻松移动。

本教程介绍如何将带 Conda 的深度学习 AMI 与 ONNX 结合使用。通过执行以下步骤，您可以训练模型或从一个框架中加载预先训练的模型，将此模型导出为 ONNX，然后将此模型导入到另一个框架中。

ONNX 先决条件

要使用此 ONNX 教程，您必须有权访问带 Conda 的深度学习 AMI 版本 12 或更高版本。有关如何开始使用带 Conda 的深度学习 AMI 的更多信息，请参阅 带 Conda 的深度学习 AMI。

> **Important**
>
> 这些示例使用可能需要多达 8 GB 内存（或更多）的函数。请务必选择具有足量内存的实例类型。

使用带 Conda 的深度学习 AMI 来启动终端会话以开始以下教程。

将 Chainer 模型转换为 ONNX，然后将模型加载到 CNTK 中

首先，激活 Chainer 环境：

```
$ source activate chainer_p36
```

使用文本编辑器创建一个新文件，并在脚本中使用以下程序来从 Chainer Model Zoo 中提取模型，然后将它导出为 ONNX 格式。

```
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)
```

在您运行此脚本后，您将在同一目录中看到新创建的 .onnx 文件。

现在您有一个 ONNX 文件，可以将其与以下示例结合使用来尝试运行推理：

- 在推理中将 CNTK 与 ONNX 模型配合使用

ONNX 教程

- 有关将 Apache MXNet 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 MXNet 的教程
- PyTorch 到 ONNX 到 MXnet 教程
- PyTorch 到 ONNX 到 CNTK 教程

有关将 Chainer 转换为 ONNX，然后加载到 MXNet 的教程

ONNX 概述

开放神经网络交换 (ONNX) 是一种用于表示深度学习模型的开放格式。ONNX 受到 Amazon Web Services、Microsoft、Facebook 和其他多个合作伙伴的支持。您可以使用任何选定的框架来设计、训练和部署深度学习模型。ONNX 模型的好处是，它们可以在框架之间轻松移动。

本教程介绍如何将带 Conda 的深度学习 AMI 与 ONNX 结合使用。通过执行以下步骤，您可以训练模型或从一个框架中加载预先训练的模型，将此模型导出为 ONNX，然后将此模型导入到另一个框架中。

ONNX 先决条件

要使用此 ONNX 教程，您必须有权访问带 Conda 的深度学习 AMI 版本 12 或更高版本。有关如何开始使用带 Conda 的深度学习 AMI 的更多信息，请参阅 带 Conda 的深度学习 AMI。

Important

这些示例使用可能需要多达 8 GB 内存（或更多）的函数。请务必选择具有足量内存的实例类型。

使用带 Conda 的深度学习 AMI 来启动终端会话以开始以下教程。

将 Chainer 模型转换为 ONNX，然后将模型加载到 MXNet 中

首先，激活 Chainer 环境：

```
$ source activate chainer_p36
```

使用文本编辑器创建一个新文件，并在脚本中使用以下程序来从 Chainer Model Zoo 中提取模型，然后将它导出为 ONNX 格式。

```
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)
```

在您运行此脚本后，您将在同一目录中看到新创建的 .onnx 文件。

现在您有一个 ONNX 文件，可以将其与以下示例结合使用来尝试运行推理：

- 将适用于推理的 Apache MXNet（孵化版）与 ONNX 模型结合使用

ONNX 教程

- 有关将 Apache MXNet 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 MXNet 的教程
- PyTorch 到 ONNX 到 MXnet 教程
- PyTorch 到 ONNX 到 CNTK 教程

PyTorch 到 ONNX 到 CNTK 教程

> **Note**
>
> 从 v28 版本开始， AWS Deep Learning AMI 中将不再包含 CNTK、Caffe、Caffe2 和 Theano Conda 环境。包含这些环境 AWS Deep Learning AMI 的先前版本将继续可用。但是，只有在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。

ONNX 概述

开放神经网络交换 (ONNX) 是一种用于表示深度学习模型的开放格式。ONNX 受到 Amazon Web Services、Microsoft、Facebook 和其他多个合作伙伴的支持。您可以使用任何选定的框架来设计、训练和部署深度学习模型。ONNX 模型的好处是，它们可以在框架之间轻松移动。

本教程介绍如何将带 Conda 的深度学习 AMI 与 ONNX 结合使用。通过执行以下步骤，您可以训练模型或从一个框架中加载预先训练的模型，将此模型导出为 ONNX，然后将此模型导入到另一个框架中。

ONNX 先决条件

要使用此 ONNX 教程，您必须有权访问带 Conda 的深度学习 AMI 版本 12 或更高版本。有关如何开始使用带 Conda 的深度学习 AMI 的更多信息，请参阅 带 Conda 的深度学习 AMI。

Important

这些示例使用可能需要多达 8 GB 内存（或更多）的函数。请务必选择具有足量内存的实例类型。

使用带 Conda 的深度学习 AMI 来启动终端会话以开始以下教程。

将 PyTorch 模型转换为 ONNX，然后将模型加载到 CNTK

首先，激活 PyTorch 环境：

```
$ source activate pytorch_p36
```

使用文本编辑器创建新文件，在脚本中使用以下程序训练模拟模型 PyTorch，然后将其导出为 ONNX 格式。

```
# 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)
```

在您运行此脚本后，您将在同一目录中看到新创建的 .onnx 文件。现在，切换到 CNTK Conda 环境以使用 CNTK 加载模型。

接下来，激活 CNTK 环境：

```
$ source deactivate
$ source activate cntk_p36
```

使用文本编辑器创建一个新文件，并在脚本中使用以下程序以在 CNTK 中打开 ONNX 格式文件。

```
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)
```

在运行此脚本后，CNTK 将加载模型。

您也可以通过以下方式使用 CNTK 导出为 ONNX：将以下内容追加到上一脚本，然后运行它。

```
# Export the model to ONNX via the CNTK export API
z.save("cntk_model.onnx", format=C.ModelFormat.ONNX)
```

ONNX 教程

- 有关将 Apache MXNet 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 MXNet 的教程
- PyTorch 到 ONNX 到 MXnet 教程
- PyTorch 到 ONNX 到 CNTK 教程

PyTorch 到 ONNX 到 MXnet 教程

ONNX 概述

开放神经网络交换 (ONNX) 是一种用于表示深度学习模型的开放格式。ONNX 受到 Amazon Web Services、Microsoft、Facebook 和其他多个合作伙伴的支持。您可以使用任何选定的框架来设计、训练和部署深度学习模型。ONNX 模型的好处是，它们可以在框架之间轻松移动。

本教程介绍如何将带 Conda 的深度学习 AMI 与 ONNX 结合使用。通过执行以下步骤，您可以训练模型或从一个框架中加载预先训练的模型，将此模型导出为 ONNX，然后将此模型导入到另一个框架中。

ONNX 先决条件

要使用此 ONNX 教程，您必须有权访问带 Conda 的深度学习 AMI 版本 12 或更高版本。有关如何开始使用带 Conda 的深度学习 AMI 的更多信息，请参阅 带 Conda 的深度学习 AMI。

> **Important**
>
> 这些示例使用可能需要多达 8 GB 内存（或更多）的函数。请务必选择具有足量内存的实例类型。

使用带 Conda 的深度学习 AMI 来启动终端会话以开始以下教程。

将 PyTorch 模型转换为 ONNX，然后将模型加载到 MXnet

首先，激活 PyTorch 环境：

```
$ source activate pytorch_p36
```

使用文本编辑器创建新文件，在脚本中使用以下程序训练模拟模型 PyTorch，然后将其导出为 ONNX 格式。

```
# 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!")
```

在您运行此脚本后，您将在同一目录中看到新创建的 .onnx 文件。现在，切换到 MXNet Conda 环境以使用 MXNet 加载模型。

接下来，激活 MXNet 环境：

```
$ source deactivate
$ source activate mxnet_p36
```

使用文本编辑器创建一个新文件，并在脚本中使用以下程序以在 MXNet 中打开 ONNX 格式文件。

```
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())
```

运行此脚本后，MXNet 将拥有加载的模型，并打印一些基本模型信息。

ONNX 教程

- 有关将 Apache MXNet 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 CNTK 的教程
- 有关将 Chainer 转换为 ONNX，然后加载到 MXNet 的教程
- PyTorch 到 ONNX 到 MXnet 教程
- PyTorch 到 ONNX 到 CNTK 教程

模型处理

以下是已在带 Conda 的深度学习 AMI 上安装的模型处理选项。单击其中一个选项可了解如何使用该选项。

主题

- 适用于 Apache MXNet 的模型服务器 (MMS)
- TensorFlow 上菜
- TorchServe

适用于 Apache MXNet 的模型服务器 (MMS)

适用于 Apache MXNet 的模型服务器 (MMS) 是一个灵活的工具，用于处理从 Apache MXNet（孵化）导出或导出为开放神经网络交换 (ONNX) 模型格式的深度学习模型。MMS 随 DLAMI with Conda 一起预安装。本 MMS 教程将演示如何处理图像分类模型。

主题

- 在 MMS 上处理图像分类模型
- 其他示例

- 更多信息

在 MMS 上处理图像分类模型

本教程介绍如何利用 MMS 处理图像分类模型。该模型通过 MMS Model Zoo 提供，在您启动 MMS 时自动进行下载。在服务器开始运行后，它会立即侦听预测请求。在这种情况下，如果您上传图像 (一个小猫的图像)，服务器会返回在其上训练该模型的 1,000 类中匹配的前 5 个类的预测。有关模型、模型的训练方式以及模型的测试方式的更多信息可在 MMS Model Zoo 中找到。

在 MMS 上处理示例图像分类模型

1. 连接到带 Conda 的深度学习 AMI 的 Amazon Elastic Compute Cloud (Amazon EC2) 实例。
2. 激活 MXNet 环境：
   - 对于使用 CUDA 9.0 和 MKL-DNN 的 Python 3 上的 MXNet 和 Keras 2，运行以下命令：

     ```
     $ source activate mxnet_p36
     ```

   - 对于使用 CUDA 9.0 和 MKL-DNN 的 Python 2 上的 MXNet 和 Keras 2，运行以下命令：

     ```
     $ source activate mxnet_p27
     ```

3. 使用以下命令运行 MMS。添加 `> /dev/null` 将在您运行其他测试时生成无提示日志输出。

   ```
   $ mxnet-model-server --start > /dev/null
   ```

   MMS 现在正在您的主机上运行并且正在侦听推理请求。
4. 接下来，使用 curl 命令管理 MMS 的管理终端节点，并告知它您希望它处理的模型。

   ```
   $ 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 需要知道您要使用的工作线程的数量。对于此测试，您可以尝试 3。

   ```
   $ curl -v -X PUT "http://localhost:8081/models/squeezenet_v1.1?min_worker=3"
   ```

6. 下载一个小猫图像并将其发送到 MMS 预测终端节点：

   ```
   $ 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
   ```

该预测终端节点将返回一个 JSON 格式的预测 (类似于下面的前 5 个预测)，其中，图像具有 94% 的可能性为埃及猫，然后有 5.5% 的可能性为猞猁或山猫：

```
{
 "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. 测试更多映像，或者如果您已完成测试，请停止服务器：

```
$ mxnet-model-server --stop
```

本教程重点介绍基本模型处理。MMS 还支持将 Elastic Inference 与模型服务一起使用。有关更多信息，请参阅模型服务与 Amazon Elastic Inference

当你准备好进一步了解其他彩信功能时，请参阅上的彩信文档。 GitHub

其他示例

MMS 具有可在您的 DLAMI 上运行的各种示例。您可以在 MMS 项目存储库上查看它们。

更多信息

要了解更多彩信文档，包括如何使用 Docker 设置彩信，或者如何利用最新的彩信功能，请将彩信项目页面加上星标。 GitHub

TensorFlow 上菜

TensorFlow S@@ er ving 是一款适用于机器学习模型的灵活、高性能的服务系统。

`tensorflow-serving-api` 预装了带 Conda 的深度学习 AMI！您将在 `~/examples/tensorflow-serving/` 中找到一个用于训练、导出和处理 MNIST 模型的示例脚本。

要运行这些示例中的任何一个，请先使用 Conda 连接到您的深度学习 AMI 并激活 TensorFlow 环境。

```
$ source activate tensorflow_p37
```

现在，将目录更改至服务示例脚本文件夹。

```
$ cd ~/examples/tensorflow-serving/
```

处理预训练的 Inception 模型

以下是您可尝试为不同的模型（如 Inception）提供服务的示例。作为一般规则，您需要将可维护模型和客户端脚本下载到您的 DLAMI。

使用 Inception 模型处理和测试推理

1. 下载该模型。

    ```
    $ curl -O https://s3-us-west-2.amazonaws.com/tf-test-models/INCEPTION.zip
    ```

2. 解压缩模型。

    ```
    $ unzip INCEPTION.zip
    ```

3. 下载一张哈士奇的照片。

```
$ curl -O https://upload.wikimedia.org/wikipedia/commons/b/b5/Siberian_Husky_bi-eyed_Flickr.jpg
```

4. 启动服务器。请注意，对于 Amazon Linux，您必须将用于 `model_base_path` 的目录从 `/home/ubuntu` 更改为 `/home/ec2-user`。

```
$ tensorflow_model_server --model_name=INCEPTION --model_base_path=/home/ubuntu/examples/tensorflow-serving/INCEPTION/INCEPTION --port=9000
```

5. 对于在前台运行的服务器，您需要启动另一个终端会话才能继续。打开一个新的终端并 TensorFlow 使用激活`source activate tensorflow_p37`。然后，使用您的首选文本编辑器创建具有以下内容的脚本。将它命名为 `inception_client.py`。此脚本将映像文件名用作参数，并从预训练模型中获得预测结果。

```
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. 现在，运行将服务器位置和端口以及哈士奇照片的文件名作为参数传递的脚本。

```
$ python3 inception_client.py --server=localhost:9000 --image Siberian_Husky_bi-eyed_Flickr.jpg
```

训练和处理 MNIST 模型

对于本教程，我们将导出模型并随后通过 `tensorflow_model_server` 应用程序处理它。最后，您可以使用示例客户端脚本测试模型服务器。

运行将训练和导出 MNIST 模型的脚本。作为脚本的唯一参数，您需要为其提供一个文件夹位置以保存该模型。现在，我们可以把它放入 `mnist_model` 中。该脚本将会为您创建此文件夹。

```
$ python mnist_saved_model.py /tmp/mnist_model
```

请耐心等待，因为此脚本可能需要一段时间才能提供输出。当训练完成并最终导出模型后，您应该看到以下内容：

```
Done training!
Exporting trained model to mnist_model/1
Done exporting!
```

下一步是运行 `tensorflow_model_server` 以处理导出的模型。

```
$ tensorflow_model_server --port=9000 --model_name=mnist --model_base_path=/tmp/
mnist_model
```

为您提供了一个客户端脚本来测试服务器。

要对其进行测试，您将需要打开一个新的终端窗口。

```
$ python mnist_client.py --num_tests=1000 --server=localhost:9000
```

更多功能和示例

如果您有兴趣了解有关 TensorFlow 服务的更多信息，请TensorFlow 访问该网站。

您也可以使用 Amazon Elastic In ference TensorFlow 提供服务。如需更多信息，请查看有关如何将 Elastic Inference 与 TensorFlow 服务结合使用的指南。

TorchServe

TorchServe 是一款灵活的工具，用于提供已从中导出的深度学习模型 PyTorch。 TorchServe 预装了带有 Conda 的深度学习 AMI，从 v34 开始。

有关使用的更多信息 TorchServe，请参阅 PyTorch文档模型服务器。

主题

在上提供图像分类模型 TorchServe

本教程介绍如何使用提供图像分类模型 TorchServe。它使用提供的 DenseNet -161 模型。 PyTorch 服务器运行后，它会监听预测请求。在这种情况下，如果您上传图像（一张小猫的图像），服务器会返回在其上训练该模型的类中匹配的前 5 个类的预测。

在上提供图像分类模型示例 TorchServe

1. 使用带 Conda 的深度学习 AMI（v34 或更高版本）来连接到 Amazon Elastic Compute Cloud (Amazon EC2) 实例。
2. 激活 `pytorch_latest_p36` 环境。

   ```
   source activate pytorch_latest_p36
   ```

3. 克隆 TorchServe 存储库，然后创建一个目录来存储您的模型。

   ```
   git clone https://github.com/pytorch/serve.git
   ```

```
mkdir model_store
```

4. 使用模型存档程序来存档模型。该`extra-files`参数使用`TorchServe`存储库中的文件，因此如有必要，请更新路径。 有关模型存档器的更多信息，请参阅 Torch 模型存档器。 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. 运行 TorchServe 以启动终端节点。添加 > `/dev/null` 会使日志输出静音。

```
torchserve --start --ncs --model-store model_store --models densenet161.mar > /dev/
null
```

6. 下载小猫的图像并将其发送到 TorchServe预测端点：

```
curl -O https://s3.amazonaws.com/model-server/inputs/kitten.jpg
curl http://127.0.0.1:8080/predictions/densenet161 -T kitten.jpg
```

该预测终端节点将返回一个 JSON 格式的预测（类似于下面的前 5 个预测），其中，图像具有 47% 的可能性为埃及猫，然后有 46% 的可能性为虎斑猫。

```
{
 "tiger_cat": 0.46933576464653015,
 "tabby": 0.463387668132782,
 "Egyptian_cat": 0.0645613968372345,
 "lynx": 0.0012828196631744504,
 "plastic_bag": 0.00023323058849200606
}
```

7. 当您完成测试时，停止服务器：

```
torchserve --stop
```

其他示例

TorchServe 提供了各种各样的示例，您可以在 DLAMI 实例上运行这些示例。您可以在 TorchServe项目存储库示例页面上查看它们。

更多信息

有关更多 TorchServe 文档，包括如何 TorchServe使用 Docker 进行设置和最新 TorchServe 功能，请参阅上的 TorchServe GitHub项目页面。

升级 DLAMI

在此处，您将找到有关升级 DLAMI 的信息以及有关在 DLAMI 上更新软件的提示。

主题

- 升级到新版 DLAMI
- 有关软件更新的提示

升级到新版 DLAMI

DLAMI 的系统映像定期更新，以利用新的深度学习框架版本、CUDA、其他软件更新和性能优化。如果您已使用 DLAMI 一段时间并想要利用更新，则需要启动新实例。您还必须手动传输任何数据集、检查点或其他宝贵的数据。或者，您可以使用 Amazon EBS 来保留数据，并将其附加到新的 DLAMI。通过这种方法，您可以经常升级，同时最大限度地减少转换数据所需的时间。

> **Note**
>
> 在 DLAMI 之间附加和移动 Amazon EBS 卷时，必须在同一个可用区同时具有 DLAMI 和新卷。

1. 使用 Amazon EC2 控制台来创建新的 Amazon EBS 卷。有关详细说明，请参阅创建 Amazon EBS 卷。
2. 将新创建的 Amazon EBS 卷附加到现有 DLAMI。有关详细说明，请参阅附加 Amazon EBS 卷。
3. 传输您的数据，如数据集、检查点和配置文件。
4. 启动 DLAMI。有关详细指导，请参阅启动和配置 DLAMI。
5. 从旧 DLAMI 中分离 Amazon EBS 卷。有关详细说明，请参阅分离 Amazon EBS 卷。
6. 将该 Amazon EBS 卷附加到新的 DLAMI。请按照步骤 2 中的相关说明附加卷。
7. 在确认数据可用于新的 DLAMI 上时，停止并终止旧的 DLAMI。有关更详细的清理说明，请参阅清除。

有关软件更新的提示

有时，您可能想要在 DLAMI 上手动更新软件。通常建议您使用 `pip` 来更新 Python 软件包。您还应在带 Conda 的深度学习 AMI 上的 Conda 环境内使用 `pip` 以更新软件包。有关升级和安装说明，请参阅特定框架或软件的网站。

> **Note**
>
> 我们不能保证软件包更新一定成功。尝试在依赖项不兼容的环境中升级软件包可能会导致安装失败。在这种情况下，您应该联系库维护人员，看看是否可以更新软件包依赖项。或者，您可以尝试以允许更新的方式来修改环境。但是，这种修改可能意味着删除或更新现有软件包，这意味着我们无法再保证此环境的稳定性。

如果您有兴趣运行某个特定软件包的最新主分支，请激活相应的环境，然后将 `--pre` 添加到 `pip install --upgrade` 命令的结尾：例如：

```
source activate mxnet_p36
pip install --upgrade mxnet --pre
```

AWS Deep Learning AMI 预装了许多 Conda 环境和许多软件包。由于预装软件包数量众多，要找到一组保证兼容的软件包非常困难。您可能会看到“环境不一致，请仔细检查软件包计划”的警告。DLAMI 确保 DLAMI 提供的所有环境都是正确的，但不能保证用户安装的任何软件包都能正常运行。

安全性 AWS Deep Learning AMI

云安全 AWS 是重中之重。作为 AWS 客户，您可以受益于专为满足大多数安全敏感型组织的要求而构建的数据中心和网络架构。

安全是双方 AWS 的共同责任。责任共担模式将其描述为云的安全性和云中的安全性：

- 云安全 — AWS 负责保护在 AWS 云中运行 AWS 服务的基础架构。 AWS 还为您提供可以安全使用的服务。作为AWS 合规计划合规计划合规计划合的一部分，第三方审计师定期测试和验证我们安全的有效性。要了解适用于DLAMI的合规计划，AWS 请参阅按合规计划提供的范围内的服务按合规 AWS。
- 云端安全-您的责任由您使用的 AWS 服务决定。您还需要对其他因素负责，包括您的数据的敏感性、您公司的要求以及适用的法律法规。

此文档将帮助您了解如何在使用 DLAMI 时应用责任共担模式。以下主题说明如何配置 DLAMI 以实现您的安全性和合规性目标。您还将学习如何使用其他 AWS 服务来帮助您监控和保护您的DLAMI资源。

有关更多信息，请参阅 Amazon EC2 中的安全性。

主题

- 中的数据保护 AWS Deep Learning AMI
- Identity and Access Management AWS Deep Learning AMI
- 登录和监控 AWS Deep Learning AMI
- 的合规性验证 AWS Deep Learning AMI
- 韧性在 AWS Deep Learning AMI
- 中的基础设施安全 AWS Deep Learning AMI

中的数据保护 AWS Deep Learning AMI

分 AWS 担责任模型适用于 AWS 深度学习 AMI 中的数据保护。如本模型所述 AWS ，负责保护运行所有内容的全球基础架构 AWS Cloud。您负责维护对托管在此基础设施上的内容的控制。您还负责您所使用的 AWS 服务 的安全配置和管理任务。有关数据隐私的更多信息，请参阅数据隐私常见问题。有关欧洲数据保护的信息，请参阅 AWS 安全性博客 上的 AWS 责任共担模式和 GDPR 博客文章。

出于数据保护目的，我们建议您保护 AWS 账户 凭证并使用 AWS IAM Identity Center 或 AWS Identity and Access Management (IAM) 设置个人用户。这样，每个用户只获得履行其工作职责所需的权限。我们还建议您通过以下方式保护数据：

- 对每个账户使用 multi-factor authentication（MFA）。
- 使用 SSL/TLS 与资源通信。 AWS 我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 使用设置 API 和用户活动日志 AWS CloudTrail。
- 使用 AWS 加密解决方案以及其中的所有默认安全控件 AWS 服务。
- 使用高级托管安全服务（例如 Amazon Macie），它有助于发现和保护存储在 Amazon S3 中的敏感数据。
- 如果您在 AWS 通过命令行界面或 API 进行访问时需要经过 FIPS 140-2 验证的加密模块，请使用 FIPS 端点。有关可用的 FIPS 端点的更多信息，请参阅《美国联邦信息处理标准（FIPS）第 140-2 版》。

我们强烈建议您切勿将机密信息或敏感信息（如您客户的电子邮件地址）放入标签或自由格式文本字段（如名称字段）。这包括您使用控制台、AP AWS CLI I 或 SDK 使用 DLAMI AWS 服务 或其他工具包的情况。 AWS 在用于名称的标签或自由格式文本字段中输入的任何数据都可能会用于计费或诊断日志。如果您向外部服务器提供网址，强烈建议您不要在网址中包含凭证信息来验证对该服务器的请求。

Identity and Access Management AWS Deep Learning AMI

AWS Identity and Access Management (IAM) AWS 服务 可帮助管理员安全地控制对 AWS 资源的访问权限。IAM 管理员控制可以通过身份验证（登录）和授权（具有权限）使用 DLAMI 资源的人员。您可以使用 IAM AWS 服务 ，无需支付额外费用。

有关 Identity and Access Management 的更多信息，请参阅 Amazon EC2 的 Identity and Access Management。

主题

- 使用身份进行身份验证
- 使用策略管理访问
- IAM 与 Amazon EMR 结合使用

使用身份进行身份验证

身份验证是您 AWS 使用身份凭证登录的方式。您必须以 IAM 用户身份或通过担 AWS 账户根用户任 IAM 角色进行身份验证（登录 AWS）。

您可以使用通过身份源提供的凭据以 AWS 联合身份登录。 AWS IAM Identity Center （IAM Identity Center）用户、贵公司的单点登录身份验证以及您的 Google 或 Facebook 凭据就是联合身份的示例。当您以联合身份登录时，您的管理员以前使用 IAM 角色设置了身份联合验证。当你使用联合访问 AWS 时，你就是在间接扮演一个角色。

根据您的用户类型，您可以登录 AWS Management Console 或 AWS 访问门户。有关登录的更多信息 AWS，请参阅《AWS 登录 用户指南》中的如何登录到您 AWS 账户的。

如果您 AWS 以编程方式访问，则会 AWS 提供软件开发套件 (SDK) 和命令行接口 (CLI)，以便使用您的凭据对请求进行加密签名。如果您不使用 AWS 工具，则必须自己签署请求。有关使用推荐的方法自行签署请求的更多信息，请参阅 IAM 用户指南中的签署 AWS API 请求。

无论使用何种身份验证方法，您可能需要提供其他安全信息。例如， AWS 建议您使用多重身份验证 (MFA) 来提高账户的安全性。要了解更多信息，请参阅《AWS IAM Identity Center 用户指南》中的 多重身份验证 和《IAM 用户指南》中的 在 AWS中使用多重身份验证（MFA）。

AWS 账户 root 用户

创建时 AWS 账户，首先要有一个登录身份，该身份可以完全访问账户中的所有资源 AWS 服务 和资源。此身份被称为 AWS 账户 root 用户，使用您创建账户时使用的电子邮件地址和密码登录即可访问该身份。强烈建议您不要使用根用户执行日常任务。保护好根用户凭证，并使用这些凭证来执行仅根用户可以执行的任务。有关需要您以根用户身份登录的任务的完整列表，请参阅 IAM 用户指南 中的 需要根用户凭证的任务。

IAM 用户和群组

I AM 用户是您 AWS 账户 内部对个人或应用程序具有特定权限的身份。在可能的情况下，我们建议使用临时凭证，而不是创建具有长期凭证（如密码和访问密钥）的 IAM 用户。但是，如果您有一些特定的使用场景需要长期凭证以及 IAM 用户，我们建议您轮换访问密钥。有关更多信息，请参阅《IAM 用户指南》中的对于需要长期凭证的使用场景定期轮换访问密钥。

IAM 组是一个指定一组 IAM 用户的身份。您不能使用组的身份登录。您可以使用组来一次性为多个用户指定权限。如果有大量用户，使用组可以更轻松地管理用户权限。例如，您可能具有一个名为 IAMAdmins 的组，并为该组授予权限以管理 IAM 资源。

用户与角色不同。用户唯一地与某个人员或应用程序关联，而角色旨在让需要它的任何人代入。用户具有永久的长期凭证，而角色提供临时凭证。要了解更多信息，请参阅 IAM 用户指南中的何时创建 IAM 用户（而不是角色）。

IAM 角色

I AM 角色是您内部具有特定权限 AWS 账户 的身份。它类似于 IAM 用户，但与特定人员不关联。您可以 AWS Management Console 通过切换角色在中临时担任 IAM 角色。您可以通过调用 AWS CLI 或 AWS API 操作或使用自定义 URL 来代入角色。有关使用角色的方法的更多信息，请参阅《IAM 用户指南》中的 使用 IAM 角色。

具有临时凭证的 IAM 角色在以下情况下很有用：

- Federated user access（联合用户访问）– 要向联合身份分配权限，请创建角色并为角色定义权限。当联合身份进行身份验证时，该身份将与角色相关联并被授予由此角色定义的权限。有关联合身份验证的角色的信息，请参阅《IAM 用户指南》中的 为第三方身份提供商创建角色。如果您使用 IAM Identity Center，则需要配置权限集。为控制您的身份在进行身份验证后可以访问的内容，IAM Identity Center 将权限集与 IAM 中的角色相关联。有关权限集的信息，请参阅《AWS IAM Identity Center 用户指南》中的权限集。
- 临时 IAM 用户权限 – IAM 用户可代入 IAM 用户或角色，以暂时获得针对特定任务的不同权限。
- 跨账户存取 – 您可以使用 IAM 角色以允许不同账户中的某个人（可信主体）访问您的账户中的资源。角色是授予跨账户访问权限的主要方式。但是，对于某些资源 AWS 服务，您可以将策略直接附加到资源（而不是使用角色作为代理）。要了解用于跨账户访问的角色和基于资源的策略之间的差别，请参阅《IAM 用户指南》中的 IAM 角色与基于资源的策略有何不同。
- 跨服务访问 — 有些 AWS 服务 使用其他 AWS 服务服务中的功能。例如，当您在某个服务中进行调用时，该服务通常会在 Amazon QLDB 中运行应用程序或在 Simple Storage Service（Amazon S3）中存储对象。服务可能会使用发出调用的主体的权限、使用服务角色或使用服务相关角色来执行此操作。
  - 转发访问会话 (FAS) — 当您使用 IAM 用户或角色在中执行操作时 AWS，您被视为委托人。使用某些服务时，您可能会执行一个操作，此操作然后在不同服务中启动另一个操作。FAS 使用调用委托人的权限以及 AWS 服务 向下游服务发出请求的请求。 AWS 服务只有当服务收到需要与其他 AWS 服务 或资源交互才能完成的请求时，才会发出 FAS 请求。在这种情况下，您必须具有执行这两个操作的权限。有关发出 FAS 请求时的策略详情，请参阅转发访问会话。
  - 服务角色 - 服务角色是服务代表您在您的账户中执行操作而分派的 IAM 角色。IAM 管理员可以在 IAM 中创建、修改和删除服务角色。有关更多信息，请参阅《IAM 用户指南》中的创建向 AWS 服务委派权限的角色。

- 服务相关角色-服务相关角色是一种与服务相关联的服务角色。 AWS 服务服务可以代入代表您执行操作的角色。服务相关角色出现在您的中 AWS 账户 ，并且归服务所有。IAM 管理员可以查看但不能编辑服务相关角色的权限。
- 在 A@@ mazon EC2 上运行的应用程序 — 您可以使用 IAM 角色管理在 EC2 实例上运行并发出 AWS CLI 或 AWS API 请求的应用程序的临时证书。这优先于在 EC2 实例中存储访问密钥。要向 EC2 实例分配 AWS 角色并使其可供其所有应用程序使用，您需要创建附加到该实例的实例配置文件。实例配置文件包含角色，并使 EC2 实例上运行的程序能够获得临时凭证。有关更多信息，请参阅《IAM 用户指南》中的 使用 IAM 角色为 Amazon EC2 实例上运行的应用程序授予权限。

要了解是使用 IAM 角色还是 IAM 用户，请参阅 IAM 用户指南中的何时创建 IAM 角色（而不是用户）。

使用策略管理访问

您可以 AWS 通过创建策略并将其附加到 AWS 身份或资源来控制中的访问权限。策略是其中的一个对象 AWS ，当与身份或资源关联时，它会定义其权限。 AWS 在委托人（用户、root 用户或角色会话）发出请求时评估这些策略。策略中的权限确定是允许还是拒绝请求。大多数策略都以 JSON 文档的 AWS 形式存储在中。有关 JSON 策略文档的结构和内容的更多信息，请参阅《IAM 用户指南》中的 JSON 策略概览。

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体 可以对什么资源执行操作，以及在什么条件下执行。

默认情况下，用户和角色没有权限。要授予用户对所需资源执行操作的权限，IAM 管理员可以创建 IAM policy。管理员随后可以向角色添加 IAM policy，用户可以代入角色。

IAM policy 定义操作的权限，无关乎您使用哪种方法执行操作。例如，假设您有一个允许 `iam:GetRole` 操作的策略。拥有该策略的用户可以从 AWS Management Console AWS CLI、或 AWS API 获取角色信息。

基于身份的策略

基于身份的策略是可附加到身份（如 IAM 用户、用户组或角色）的 JSON 权限策略文档。这些策略控制用户和角色可在何种条件下对哪些资源执行哪些操作。要了解如何创建基于身份的策略，请参阅 IAM 用户指南中的创建 IAM policy。

基于身份的策略可以进一步归类为内联策略或托管式策略。内联策略直接嵌入单个用户、组或角色中。托管策略是独立的策略，您可以将其附加到中的多个用户、群组和角色 AWS 账户。托管策略包括

AWS 托管策略和客户托管策略。要了解如何在托管式策略和内联策略之间进行选择，请参阅《IAM 用户指南》中的在托管式策略与内联策略之间进行选择。

基于资源的策略

基于资源的策略是附加到资源的 JSON 策略文档。基于资源的策略的示例包括 IAM 角色信任策略和 Simple Storage Service（Amazon S3）存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。对于在其中附加策略的资源，策略定义指定主体可以对该资源执行哪些操作以及在什么条件下执行。您必须在基于资源的策略中指定主体。委托人可以包括账户、用户、角色、联合用户或 AWS 服务。

基于资源的策略是位于该服务中的内联策略。您不能在基于资源的策略中使用 IAM 中的 AWS 托管策略。

访问控制列表（ACL）

访问控制列表（ACL）控制哪些主体（账户成员、用户或角色）有权访问资源。ACL 与基于资源的策略类似，尽管它们不使用 JSON 策略文档格式。

Amazon S3 和 Amazon VPC 就是支持 ACL 的服务示例。 AWS WAF要了解有关 ACL 的更多信息，请参阅 Amazon Simple Storage Service 开发人员指南 中的访问控制列表（ACL）概览。

其他策略类型

AWS 支持其他不太常见的策略类型。这些策略类型可以设置更常用的策略类型向您授予的最大权限。

- 权限边界 – 权限边界是一个高级特征，用于设置基于身份的策略可以为 IAM 实体（IAM 用户或角色）授予的最大权限。您可为实体设置权限边界。这些结果权限是实体基于身份的策略及其权限边界的交集。在 `Principal` 中指定用户或角色的基于资源的策略不受权限边界限制。任一项策略中的显式拒绝将覆盖允许。有关权限边界的更多信息，请参阅 IAM 用户指南中的 IAM 实体的权限边界。
- 服务控制策略 (SCP)-SCP 是 JSON 策略，用于指定组织或组织单位 (OU) 的最大权限。 AWS Organizations AWS Organizations 是一项用于对您的企业拥有的多 AWS 账户 项进行分组和集中管理的服务。如果在组织内启用了所有特征，则可对任意或全部账户应用服务控制策略（SCP）。SCP 限制成员账户中的实体（包括每个 AWS 账户根用户实体）的权限。有关 Organizations 和 SCP 的更多信息，请参阅 AWS Organizations 用户指南中的 SCP 的工作原理。
- 会话策略 – 会话策略是当您以编程方式为角色或联合身份用户创建临时会话时作为参数传递的高级策略。结果会话的权限是用户或角色的基于身份的策略和会话策略的交集。权限也可以来自基于资源的策略。任一项策略中的显式拒绝将覆盖允许。有关更多信息，请参阅 IAM 用户指南中的会话策略。

多个策略类型

当多个类型的策略应用于一个请求时，生成的权限更加复杂和难以理解。要了解在涉及多种策略类型时如何 AWS 确定是否允许请求，请参阅 IAM 用户指南中的策略评估逻辑。

IAM 与 Amazon EMR 结合使用

您可以 AWS Identity and Access Management 与 Amazon EMR 一起使用来定义用户、 AWS 资源、群组、角色和策略。您还可以控制这些用户和角色可以访问哪些 AWS 服务。

有关将 IAM 与 Amazon EMR 结合使用的更多信息，请参阅 Amazon EMR 的AWS Identity and Access Management。

登录和监控 AWS Deep Learning AMI

您的 AWS Deep Learning AMI 实例附带了多个 GPU 监控工具，包括一个向 Amazon 报告 GPU 使用情况统计数据的实用工具 CloudWatch。有关更多信息，请参阅 GPU 监控和优化以及监控 Amazon EC2。

使用情况跟踪

以下 AWS Deep Learning AMI 操作系统发行版包括 AWS 允许收集实例类型、实例 ID、DLAMI 类型和操作系统信息的代码。不会收集或保留有关在 DLAMI 中使用的命令的信息。不会收集或保留有关 DLAMI 的其他信息。

- Ubuntu 16.04
- Ubuntu 18.04
- Ubuntu 20.04
- Amazon Linux 2

要选择退出对您的 DLAMI 进行使用情况跟踪，请在启动期间为您的 Amazon EC2 实例添加标签。标签应使用密钥 `OPT_OUT_TRACKING`，并将关联值设置为 `true`。有关更多信息，请参阅标记 Amazon EC2 资源。

的合规性验证 AWS Deep Learning AMI

AWS Deep Learning AMI 作为多个合规计划的一部分，第三方审计师对安全性和 AWS 合规性进行评估。有关支持的合规性计划的信息，请参阅 Amazon EC2 的合规性验证。

有关特定合规计划范围内的 AWS 服务列表，请参阅合规计划范围内的AWS 服务按合规计划。有关一般信息，请参阅AWS 合规计划AWS。

您可以使用下载第三方审计报告 AWS Artifact。有关更多信息，请参阅在 Artifact 中下载报告在 AWS Ar 。

您在使用DLAMI时的合规责任取决于您的数据的敏感度、贵公司的合规目标以及适用的法律和法规。AWS 提供了以下资源来帮助实现合规性：

- 安全性与合规性快速入门指南 - 这些部署指南讨论了架构注意事项，并提供了在 AWS上部署基于安全性和合规性的基准环境的步骤。
- AWS 合AWS 规资源 — 此工作簿和指南集可能适用于您的行业和所在地区。
- 使用AWS Config 开发人员指南中的规则评估资源 — 该 AWS Config 服务评估您的资源配置在多大程度上符合内部实践、行业准则和法规。
- AWS Security Hub— 此 AWS 服务可全面了解您的安全状态 AWS ，帮助您检查是否符合安全行业标准和最佳实践。

韧性在 AWS Deep Learning AMI

AWS 全球基础设施是围绕 AWS 区域和可用区构建的。 AWS 区域提供多个物理隔离和隔离的可用区，这些可用区通过低延迟、高吞吐量和高度冗余的网络相连。利用可用区，您可以设计和操作在可用区之间无中断地自动实现失效转移的应用程序和数据库。与传统的单个或多个数据中心基础设施相比，可用区具有更高的可用性、容错性和可扩展性。

有关 AWS 区域和可用区的更多信息，请参阅AWS 全球基础设施。

有关帮助支持数据弹性和备份需求的功能的信息，请参阅 Amazon EC2 中的弹性。

中的基础设施安全 AWS Deep Learning AMI

的基础设施安全由 Amazon EC2 提供支持。 AWS Deep Learning AMI 有关更多信息，请参阅 Amazon EC2 中的基础设施安全性。

DLAMI 的重要更改

常见问题

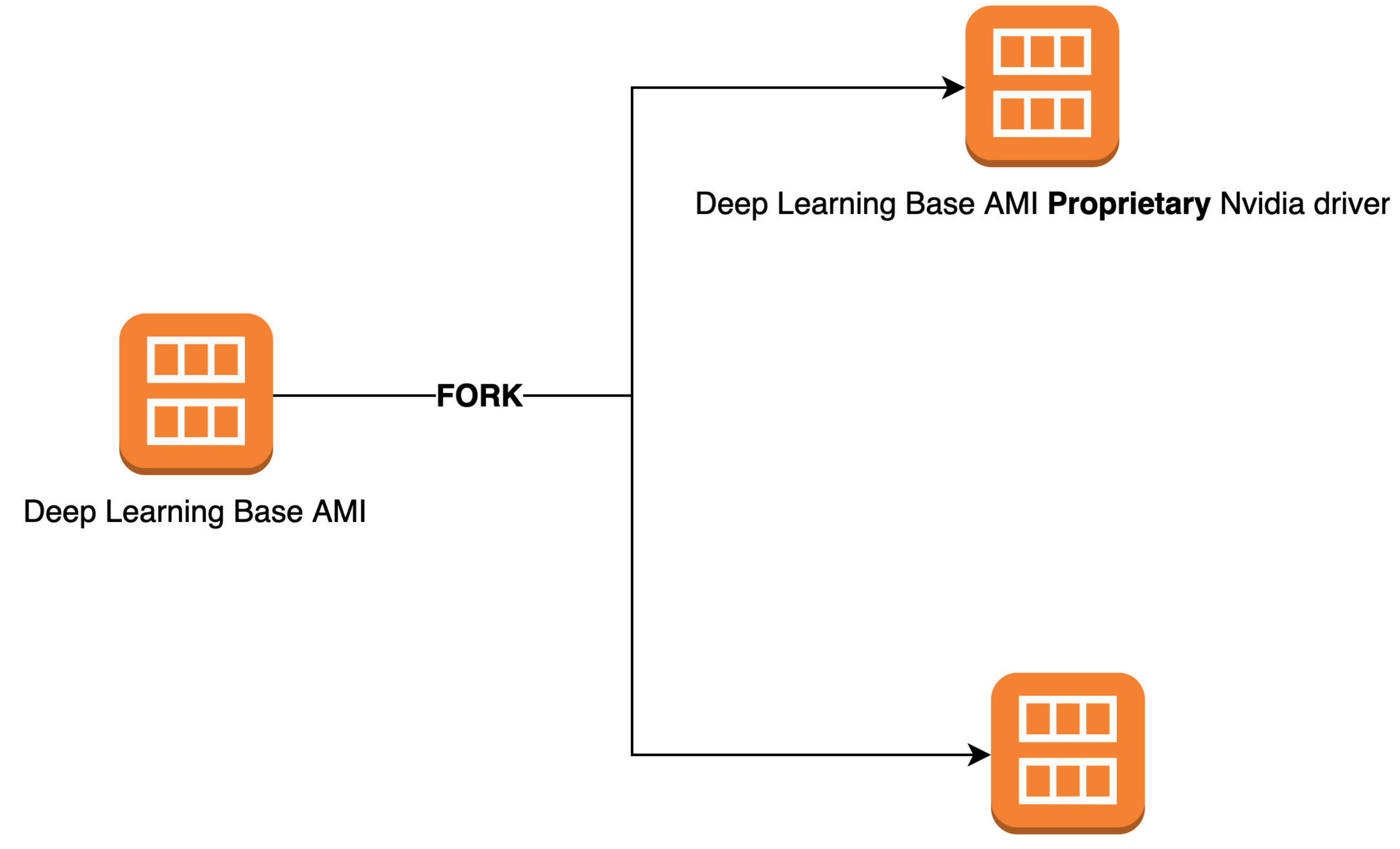
- 发生了什么变化？
- 为什么需要进行此更改？
- 哪些 DLAMI 受此更改的影响？
- 这对你意味着什么？
- 你应该什么时候开始使用新的 dLAMis？
- 新的 dLaMis 会不会在功能上有所损失？
- 那么 DLC 呢？

发生了什么变化？

2023 年 11 月 15 日 AWS Deep Learning AMI （dLAMis）将分为两个独立的小组：

- 使用 Nvidia 专有驱动程序（支持 P3、p3dn、G3）的 DLAMI。
- 使用 Nvidia OSS 驱动程序（支持 g4dn、G5、P4、P5）的 DLAMI。

因此，将使用新的名称和新的 AMI ID 为这两个类别分别创建新的 DLAMI。这些 DLAMI 不可互换，也就是说，来自一个组的 dLAMI 将不支持另一个组支持的实例，例如，支持 p5 的 DLAMI 将不支持 g3，反之亦然。



为什么需要进行此更改？

目前，适用于 NVIDIA GPU 的 DLAMI 包括来自 NVIDIA 的专有内核驱动程序。但是，最近，上游 Linux 内核社区接受了一项变更，该变更将专有内核驱动程序（例如 NVIDIA GPU 驱动程序）与其他内核驱动程序的通信隔离开来。此更改禁用了 P4/P5 系列实例上的 GPUDirect RDMA，这种机制允许 GPU 高效地使用 EFA 进行分布式训练。因此，dLAMIs 将使用 OpenRM 驱动程序（NVIDIA 开源驱动程序），该驱动程序与开源 EFA 驱动程序相关联，以支持 g4dn、G5、P4 和 P5。但是，此 OpenRM 驱动程序不支持较旧的实例（P3、G3 等） 因此，为了确保我们继续提供支持这两种类型实例的最新、高性能和安全的DLAMI，我们将DLAMI分为两组——一组使用OpenRM驱动程序（支持 G4dn、G5、P4和P5），另一组使用较旧的专有驱动程序（支持较旧的实例 P3、p3dn、G3）。

哪些 DLAMI 受此更改的影响？

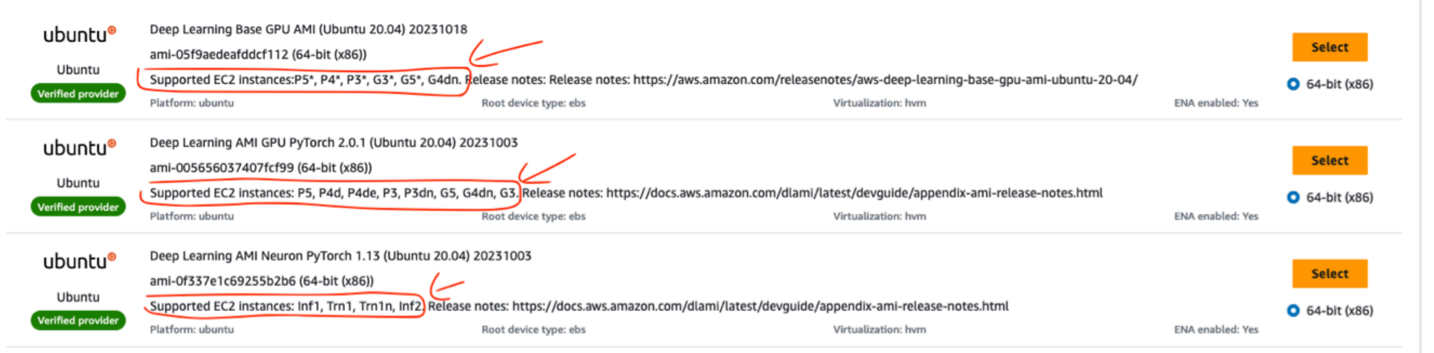
所有 DLAMI 都受此更改的影响。

这对你意味着什么？

只要在兼容的实例类型上运行，新 DLAMI 将继续提供当前 DLAMI 的功能、性能和安全性。如果您使用的是 DLAMI，则需要确保在每个 DLAMI 的发行说明中提到的兼容实例上启动 DLAMI（见此处）。例如：您需要将此更改适应于：

- 使用正确的 CLI 查询调用 DLAMI（见下文）
- 在兼容的实例类型上通过控制台和 CLI 启动 DLAMI

如果您要从 EC2 控制台启动 DLAMI 快速入门：每个 DLAMI 描述都列出了 EC2 控制台支持的实例类型。您应该在兼容的实例上启动 DLAMI。



如果您使用 CLI 启动 DLAMIs，则必须修改查询。例如：

目前，以下 CLI 查询用于支持所有实例 [P3、p3dn、G3、g4dn、G5、P4、P5] 的基本 DLAMI：

```
aws ec2 describe-images --region us-east-1 --owners amazon \
--filters 'Name=name,Values=Deep Learning Base AMI (Amazon Linux 2) ????????'
 'Name=state,Values=available' \
--query 'reverse(sort_by(Images, &CreationDate))[:1].ImageId' --output text
```

新的 CLI 查询将是：

对于支持 P3、p3dn 和 G3 的基础 DLAMI，请执行以下操作：

```
aws ec2 describe-images --region us-east-1 --owners amazon \
--filters 'Name=name,Values=Deep Learning Base Proprietary Nvidia Driver AMI (Amazon
 Linux 2) Version ??.?' 'Name=state,Values=available' \
--query 'reverse(sort_by(Images, &CreationDate))[:1].ImageId' --output text
```

对于支持 g4dN、G5、P4 和 P5 的基础 DLAMI：

```
aws ec2 describe-images --region us-east-1 --owners amazon \
--filters 'Name=name,Values=Deep Learning Base OSS Nvidia Driver AMI (Amazon Linux 2)
 Version ??.?' 'Name=state,Values=available' \
--query 'reverse(sort_by(Images, &CreationDate))[:1].ImageId' --output text
```

请在此处参阅新 AMI 的更新版本说明。有关如何在 EC2 实例上启动 AMI，请参阅此处的说明。

你应该什么时候开始使用新的 dLAMis？

您应该尽快开始使用新的 DLAMis 来获取最新的框架、依赖关系、补丁和功能。或者，如果你使用的是在 2023 年 8 月 11 日之前发布的 Amazon Linux 2 DLAMI，那么你可以选择在 2023 年 11 月 30 日之前继续实时修补他们的 DLAMI（参见此处的说明）。

新的 dLaMis 会不会在功能上有所损失？

不，新的 DLAMis 不会丢失任何功能。拆分后的新 DLAMI 将继续提供拆分前旧 DLAMI 的所有功能、性能和安全性，前提是它们是在兼容的实例上运行的。我们将 DLAMI 分为两组，以便我们继续提供最新、高性能且安全的 DLAMI，供您在各种实例上使用。

那么 DLC 呢？

DLC 不包含 NVIDIA 驱动程序，因此它们不受此更改的影响。但是您应确保 DLC 在与底层实例兼容的 AMI 上运行。

相关信息

主题

- 论坛
- 相关博客文章
- 常见问题

论坛

- 论坛：AWS 深度学习 AMI

相关博客文章

- 与深度学习 AMI 相关的文章更新列表
- 启动 AWS Deep Learning AMI（10 分钟内）
- 在 Amazon EC2 C5 和 P3 实例上使用经过优化的 TensorFlow 1.6 来加快训练速度
- 适用于机器学习从业人员的全新 AWS 深度学习 AMI
- 提供全新培训课程：介绍 AWS 上的机器学习和深度学习
- 使用 AWS 的深度学习之旅

常见问题

- 问：如何跟踪与 DLAMI 相关的产品发布信息？

  以下是两个建议：

  - 将此博客类别“AWS 深度学习 AMI”加入书签，可在此处找到：与深度学习 AMI 相关的文章更新列表。
  - “观看”论坛：AWS 深度学习 AMI
- 问：是否安装了 NVIDIA 驱动程序和 CUDA？

  是。某些 DLAMI 具有不同的版本。带 Conda 的深度学习 AMI 包含任何 DLAMI 的最新版本。CUDA 安装和框架绑定 中进行了详细介绍。您也可以参阅特定 AMI 的发布说明，确认安装了什么。

- 问：是否安装了 cuDNN？

  是。

- 问：如何查看检测到的 GPU 及其当前状态？

  运行 `nvidia-smi`。根据实例类型，将会显示一个或多个 GPU，及其当前的内存消耗。

- 问：是否为我设置了虚拟环境？

  是的，但只在 带 Conda 的深度学习 AMI 上。

- 问：安装了 Python 的哪个版本？

  每个 DLAMI 都具有 Python 2 和 3。带 Conda 的深度学习 AMI 具有适用于每个框架的两个版本的环境。

- 问：是否安装了 Keras？

  视 AMI 而定。带 Conda 的深度学习 AMI 在每个框架的前端提供 Keras。Keras 的版本取决于框架的支持。

- 问：是否免费？

  所有 DLAMI 都是免费的。但是，具体取决于您所选择的实例类型，实例可能不免费。有关更多信息，请参阅 DLAMI 的定价。

- 问：我收到了关于我的框架的 CUDA 错误或 GPU 相关消息。这是怎么回事？

  检查您使用的实例类型。很多示例和教程需要有 GPU 才能正常运行。如果运行 `nvidia-smi` 后没有显示任何 GPU，则您需要使用带有一个或多个 GPU 的实例来加速另一个 DLAMI。有关更多信息，请参阅 选择 DLAMI 的实例类型。

- 问：是否可以使用 Docker？

  从带 Conda 的深度学习 AMI 版本 14 开始，都已预先安装 Docker。请注意，您需要在 GPU 实例上使用 nvidia-docker，才能使用 GPU。

- 问：哪些区域提供 Linux DLAMI？

区域	代码
美国东部（俄亥俄州）	us-east-2

区域	代码
美国东部（弗吉尼亚州北部）	us-east-1
GovCloud	us-gov-west-1
美国西部（北加利福尼亚）	us-west-1
美国西部（俄勒冈州）	us-west-2
北京（中国）	cn-north-1
宁夏（中国）	cn-northwest-1
亚太地区（孟买）	ap-south-1
亚太地区（首尔）	ap-northeast-2
亚太地区（新加坡）	ap-southeast-1
亚太地区（悉尼）	ap-southeast-2
亚太地区（东京)	ap-northeast-1
加拿大（中部）	ca-central-1
欧洲（法兰克福）	eu-central-1
欧洲（爱尔兰）	eu-west-1
欧洲（伦敦）	eu-west-2
欧洲（巴黎）	eu-west-3
南美洲（圣保罗）	sa-east-1

- 问：哪些区域提供 Windows DLAMI？

区域	代码
美国东部（俄亥俄州）	us-east-2
美国东部（弗吉尼亚州北部）	us-east-1
GovCloud	us-gov-west-1
美国西部（北加利福尼亚）	us-west-1
美国西部（俄勒冈州）	us-west-2
北京（中国）	cn-north-1
亚太地区（孟买）	ap-south-1
亚太地区（首尔）	ap-northeast-2
亚太地区（新加坡）	ap-southeast-1
亚太地区（悉尼）	ap-southeast-2
亚太地区（东京)	ap-northeast-1
加拿大（中部）	ca-central-1
欧洲（法兰克福）	eu-central-1
欧洲（爱尔兰）	eu-west-1
欧洲（伦敦）	eu-west-2
欧洲（巴黎）	eu-west-3
南美洲（圣保罗）	sa-east-1

DLAMI 的发布说明

> **Note**
>
> AWS Deep Learning AMI每晚都会发布安全补丁。这些增量安全补丁未包含在官方发布说明中。

有关任何不支持的框架发行说明，请参阅 DLAMI S upport Policy 页面。

基础 DLAMI

GPU

- AWS 深度学习基础 AMI（亚马逊 Linux 2）
- AWS 深度学习基础 AMI (Ubuntu 20.04)

AWS Neuron

- AWS 深度学习基础 AMI Neuron（亚马逊 Linux 2）
- AWS 深度学习基础 AMI 神经元 (Ubuntu 20.04)

高通

- AWS 深度学习基础高通 AMI（亚马逊 Linux 2）

单框架 DLAMI

PyTorch-特定的 AMI

- GPU
  - AWS 深度学习 AMI GPU PyTorch 2.1 (Ubuntu 20.04)
  - AWS 深度学习 AMI GPU PyTorch 1.13 (亚马逊 Linux 2)
  - AWS 深度学习 AMI GPU PyTorch 1.13 (Ubuntu 20.04)

- AWS Neuron
  - AWS 深度学习 AMI Neuron PyTorch 1.13（亚马逊 Linux 2）
  - AWS 深度学习 AMI Neuron PyTorch 1.13 (Ubuntu 20.04)

TensorFlow-特定的 AMI

- GPU
  - AWS 深度学习 AMI GPU TensorFlow 2.15 (亚马逊 Linux 2)
  - AWS 深度学习 AMI GPU TensorFlow 2.15 (Ubuntu 20.04)
  - AWS 深度学习 AMI GPU TensorFlow 2.13 (亚马逊 Linux 2)
  - AWS 深度学习 AMI GPU TensorFlow 2.13 (Ubuntu 20.04)
- AWS Neuron
  - AWS 深度学习 AMI Neuron TensorFlow 2.10 (亚马逊 Linux 2)
  - AWS 深度学习 AMI Neuron TensorFlow 2.10 (Ubuntu 20.04)

多框架 DLAMI

GPU

> **Note**
>
> 如果您只使用一个机器学习框架，我们建议您使用 单框架 DLAMI

- AWS 深度学习 AMI（亚马逊 Linux 2）

AWS Neuron

- AWS 深度学习 AMI 神经元 (Ubuntu 22.04)

DLAMI 弃用通知

下表列出了 AWS Deep Learning AMI 中的已弃用功能的相关信息。

已弃用的功能	弃用日期	弃用通知
Ubuntu 16.04	2021/10/07	Ubuntu Linux 16.04 LTS 于 2021 年 4 月 30 日结束了为期五年的 LTS 窗口，其供应商不再提供支持。自 2021 年 10 月起，不再在新版本中更新深度学习基础 AMI (Ubuntu 16.04)。先前的版本将继续可用。
Amazon Linux	2021/10/07	Amazon Linux 自 2020 年 12 月起停用。自 2021 年 10 月起，不再在新版本中更新深度学习 AMI (Amazon Linux)。先前的深度学习 AMI (Amazon Linux) 版本将继续可用。
Chainer	2020 年 7 月 1 日	Chainer 宣布自 2019 年 12 月起停用主要版本。因此，从 2020 年 7 月起，我们将不再在 DLAMI 中包含 Chainer Conda 环境。包含这些环境的先前 DLAMI 版本将继续可用。仅在开源社区针对这些框架发布安全修补程序时，我们才会为这些环境提供更新。
Python 3.6	2020 年 6 月 15 日	根据客户的要求，我们将为新版本的 TF/MX/PT 迁移到 Python 3.7。

已弃用的功能	弃用日期	弃用通知
Python 2	2020 年 1 月 1 日	Python 开源社区已正式结束对 Python 2 的支持。 TensorFlow、PyTorch 和 MXNet 社区还宣布 TensorFlow 1.15、TensorFlow 2.1、PyTorch 1.4 和 MXNet 1.6.0 版本将是最后支持 Python 2 的版本。

《AWS Deep Learning AMI 开发人员指南》的文档历史记录

变更	说明	日期
Graviton DLAMI	AWS Deep Learning AMI 现在支持基于 Arm 处理器的 Graviton GPU 上的图像。	2021 年 11 月 29 日
Habana DLAMI	AWS Deep Learning AMI 现在支持 Habana Gaudi 硬件和 Habana SynapseAI SDK。	2021 年 10 月 25 日
TensorFlow 2	在带有 Conda 的深度学习 AMI 中，现在提供了附带 CUDA 10 的 TensorFlow 2。	2019 年 12 月 3 日
AWS Inferentia	深度学习 AMI 现在支持 AWS Inferentia 硬件和 AWS Neuron SDK。	2019 年 12 月 3 日
结合使用 TensorFlow Serving 与 Inception 模型	为 TensorFlow Serving 添加了使用 Inception 模型进行推理的示例（对于有和没有 Elastic Inference 的情况）。	2018 年 11 月 28 日
使用具有 TensorFlow 和 Horovod 的 256 GPU 进行训练	TensorFlow with Horovod 教程已更新，增加了多节点训练的示例。	2018 年 11 月 28 日
Elastic Inference	弹性推理先决条件和相关信息已添加到设置指南中。	2018 年 11 月 28 日
在 DLAMI 上发布的 MMS v1.0。	此 MMS 教程已更新为使用新的模型存档格式 (.mar) 并演示新的启动和停止功能。	2018 年 11 月 15 日
从每日构建版本安装 TensorFlow	增加了一个教程，其中讲述如何在带 Conda 的深度学习 AMI	2018 年 10 月 16 日

	上卸载 TensorFlow，然后安装 TensorFlow 的每日构建版本。	
从每日构建版本安装 CNTK	增加了一个教程，其中讲述如何在带 Conda 的深度学习 AMI 上卸载 CNTK，然后安装 CNTK 的每日构建版本。	2018 年 10 月 16 日
从每日构建版本安装 Apache MXNet（孵化版）	增加了一个教程，其中讲述如何在带 Conda 的深度学习 AMI 上卸载 MXNet，然后安装 MXNet 的每日构建版本。	2018 年 10 月 16 日
从每日构建版本安装 PyTorch	增加了一个教程，其中讲述如何在带 Conda 的深度学习 AMI 上卸载 PyTorch，然后安装 PyTorch 的每日构建版本。	2018 年 9 月 25 日
现在，Docker 预先安装在您的 DLAMI 上	从带 Conda 的深度学习 AMI 版本 14 开始，都已预先安装 Docker 和 NVIDIA 的 Docker for GPU 版本。	2018 年 9 月 25 日
TensorBoard 教程	示例已移至 ~/examples/tensorboard。更新了教程路径。	2018 年 7 月 23 日
MXBoard 教程	增加了有关如何将 MXBoard 用于 MXNet 模型可视化的教程。	2018 年 7 月 23 日
分布式训练教程	增加了有关如何将 Keras-MXNet 用于多 GPU 训练的教程。针对 v4.2.0 更新了 Chainer 的教程。	2018 年 7 月 23 日
Conda 教程	已更新示例 MOTD 以反映更新的版本。	2018 年 7 月 23 日

Chainer 教程	此教程已更新为使用来自 Chainer 源的最新示例。	2018 年 7 月 23 日

早期更新:

下表描述了 2018 年 7 月之前每个 AWS Deep Learning AMI 发行版中的重要更改。

更改	说明	日期
TensorFlow with Horovod	增加了有关利用 TensorFlow 和 Horovod 训练 ImageNet 的教程。	2018 年 6 月 6 日
升级指南	添加了升级指南。	2018 年 5 月 15 日
新区域和新的 10 分钟教程	添加的新区域：美国西部 (加利福利亚北部)、南美洲、加拿大 (中部)、欧洲 (伦敦) 和欧洲 (巴黎)。此外，首次发布的 10 分钟教程的标题为：“深度学习 AMI 入门”。	2018 年 4 月 26 日
Chainer 教程	添加了有关在多 GPU、单个 GPU 和 CPU 中使用 Chainer 的教程。CUDA 集成已针对多个框架从 CUDA 8 升级到 CUDA 9。	2018 年 2 月 28 日
Linux AMIs v3.0，以及 MXNet Model Server、TensorFlow Serving 和 TensorBoard 的引入	添加了适用于 Conda AMI 的教程（使用新模型）以及可视化服务功能（使用 MXNet Model Server v0.1.5、TensorFlow Serving v1.4.0 和 TensorBoard v0.4.0。AMI 和框架 CUDA 功能（在 Conda 和 CUDA 概述中描述）。最新发行说明迁	2018 年 1 月 25 日

更改	说明	日期
	移到 https://aws.amazon.com/releasenotes/	
Linux AMI v2.0	基础、源和 Conda AMI 均已更新，涵盖了 NCCL 2.1。源和 Conda AMI 的更新还涵盖了 MXNet v1.0、PyTorch 0.3.0 和 Keras 2.0.9。	2017 年 12 月 11 日
添加了两个 Windows AMI 选项	发布了 Windows 2012 R2 和 2016 AMI：添加到 AMI 选择指南中并添加到发行说明中。	2017 年 11 月 30 日
初始文档版本	更改的详细说明，带有指向所更改主题/章节的链接。	2017 年 11 月 15 日

AWS 术语表

有关最新的 AWS 术语，请参阅《AWS 词汇表参考》中的 AWS 词汇表。

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。