AWS Deep Learning Containers

Developer Guide

AWS Deep Learning Containers: Developer Guide

Table of Contents

What are AWS Deep Learning Containers? ... 1
- About this guide ... 1
- Python 2 Support ... 1
- Prerequisites ... 1

Getting Started With Deep Learning Containers ... 2
- Amazon EC2 Tutorials ... 2
  - Amazon EC2 setup ... 2
  - Training ... 3
  - Inference ... 6
  - Custom Entrypoints ... 11
- Amazon ECS Tutorials ... 11
  - Amazon ECS setup ... 12
  - Training ... 14
  - Inference ... 20
  - Custom Entrypoints ... 32
- Amazon EKS Tutorials ... 32
  - Amazon EKS Setup ... 32
  - Training ... 38
  - Inference ... 56
  - Custom Entrypoints ... 72
  - Troubleshooting AWS Deep Learning Containers on EKS ... 73

Deep Learning Containers Images ... 76

Deep Learning Containers Resources ... 77
- Building Custom Images ... 77
  - How to Build Custom Images ... 77
- MKL Recommendations ... 78
  - MKL Recommendation for CPU containers ... 78

Security ... 83
- Data Protection ... 83
- Identity and Access Management ... 84
  - Authenticating With Identities ... 84
  - Managing Access Using Policies ... 86
  - IAM with Amazon EMR ... 87
- Logging and Monitoring ... 88
  - Usage Tracking ... 88
- Compliance Validation ... 88
- Resilience ... 89
- Infrastructure Security ... 89

Release Notes for Deep Learning Containers ... 90

Document History ... 91

AWS glossary ... 92

What are AWS Deep Learning Containers?

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

AWS Deep Learning Containers (Deep Learning Containers) are a set of Docker images for training and serving models in TensorFlow, TensorFlow 2, PyTorch, and Apache MXNet (Incubating). Deep Learning Containers provide optimized environments with TensorFlow and MXNet, Nvidia CUDA (for GPU instances), and Intel MKL (for CPU instances) libraries and are available in the Amazon Elastic Container Registry (Amazon ECR).

About this guide

This guide helps you set up and use AWS Deep Learning Containers. This guide also covers setting up Deep Learning Containers with Amazon EC2, Amazon ECS, Amazon EKS, and SageMaker. It covers several use cases that are common for deep learning, for both training and inference. This guide also provides several tutorials for each of the frameworks.

- To run training and inference on Deep Learning Containers for Amazon EC2 using MXNet, PyTorch, TensorFlow, and TensorFlow 2, see Amazon EC2 Tutorials (p. 2)
- To run training and inference on Deep Learning Containers for Amazon ECS using MXNet, PyTorch, and TensorFlow, see Amazon ECS tutorials (p. 11)
- Deep Learning Containers for Amazon EKS offer CPU, GPU, and distributed GPU-based training, as well as CPU and GPU-based inference. To run training and inference on Deep Learning Containers for Amazon EKS using MXNet, PyTorch, and TensorFlow, see Amazon EKS Tutorials (p. 32)
- For an explanantion of the Docker-based Deep Learning Containers images, the list of available images, and how to use them, see Deep Learning Containers Images (p. 76)
- For information on security in Deep Learning Containers, see Security in AWS Deep Learning Containers (p. 83)
- For a list of the latest Deep Learning Containers release notes, see Release Notes for Deep Learning Containers (p. 90)

Python 2 Support

The Python open source community has officially ended support for Python 2 on January 1, 2020. The TensorFlow and PyTorch community have announced that the TensorFlow 2.1 and PyTorch 1.4 releases will be the last ones supporting Python 2. Previous releases of the Deep Learning Containers that support Python 2 will continue to be available. However, we will provide updates to the Python 2 Deep Learning Containers only if there are security fixes published by the open source community for those versions. Deep Learning Containers releases with the next versions of the TensorFlow and PyTorch frameworks will not include the Python 2 environments.

Prerequisites

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

Getting Started With Deep Learning Containers

The following sections describe how to use Deep Learning Containers to run sample code from each of the frameworks on AWS infrastructure. For information on using Deep Learning Containers with SageMaker, see the Use Your Own Algorithms or Models with SageMaker Documentation.

Topics

- Amazon EC2 Tutorials (p. 2)
- Amazon ECS tutorials (p. 11)
- Amazon EKS Tutorials (p. 32)

Amazon EC2 Tutorials

This section shows how to run training and inference on Deep Learning Containers for EC2 using MXNet, PyTorch, TensorFlow, and TensorFlow 2.

Before starting the following tutorials, complete the steps in Amazon EC2 setup (p. 2).

Contents

- Amazon EC2 setup (p. 2)
- Training (p. 3)
- Inference (p. 6)
- Custom Entrypoints (p. 11)

Amazon EC2 setup

In this section, you learn how to set up AWS Deep Learning Containers with Amazon Elastic Compute Cloud.

Complete the following steps to configure your instance:

- Create an AWS Identity and Access Management user or modify an existing user with the following policies. You can search for them by name in the IAM console's policy tab.
  - AmazonECS_FullAccess Policy
  - AmazonEC2ContainerRegistryFullAccess

  For more information about creating or editing an IAM user, see Adding and Removing IAM Identity Permissions in the IAM user guide.
- Launch an Amazon Elastic Compute Cloud instance (CPU or GPU), preferably a Deep Learning Base AMI. Other AMIs work, but require relevant GPU drivers.
- Connect to your instance by using SSH. For more information about connections, see Troubleshooting Connecting to Your Instance in the *Amazon EC2 user guide.*.
- Ensure your AWS CLI is up to date using the steps in Installing the current AWS CLI Version.
- In your instance, run `aws configure` and provide the credentials of your created user.
- In your instance, run the following command to log in to the Amazon ECR repository where Deep Learning Containers images are hosted.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 763104351884.dkr.ecr.us-east-1.amazonaws.com
```

For a complete list of AWS Deep Learning Containers, refer to Deep Learning Containers Images (p. 76).

> **Note**
> MKL users: Read the AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78) to get the best training or inference performance.

Next steps

To learn about training and inference on Amazon EC2 with Deep Learning Containers, see Amazon EC2 Tutorials (p. 2).

Training

This section shows how to run training on AWS Deep Learning Containers for Amazon EC2 using Apache MXNet (Incubating), PyTorch, TensorFlow, and TensorFlow 2.

For a complete list of Deep Learning Containers, refer to Deep Learning Containers Images (p. 76).

> **Note**
> MKL users: Read the AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78) to get the best training or inference performance.

Contents

- TensorFlow training (p. 3)
- Apache MXNet (Incubating) training (p. 4)
- PyTorch training (p. 5)

TensorFlow training

After you log into your Amazon EC2 instance, you can run TensorFlow and TensorFlow 2 containers with the following commands. You must use `nvidia-docker` for GPU images.

- For CPU-based training, run the following.

```
$ docker run -it <CPU training container>
```

- For GPU-based training, run the following.

```
$ nvidia-docker run -it <GPU training container>
```

The previous command runs the container in interactive mode and provides a shell prompt inside the container. You can then run the following to import TensorFlow.

```
$ python
```

```
>> import tensorflow
```

Press Ctrl+D to return to the bash prompt. Run the following to begin training:

```
git clone https://github.com/fchollet/keras.git
```

```
$ cd keras
```

```
$ python examples/mnist_cnn.py
```

Next steps

To learn inference on Amazon EC2 using TensorFlow with Deep Learning Containers, see TensorFlow Inference (p. 6).

Apache MXNet (Incubating) training

To begin training with Apache MXNet (Incubating) from your Amazon EC2 instance, run the following command to run the container:

- For CPU

```
$ docker run -it <CPU training container>
```

- For GPU

```
$ nvidia-docker run -it <GPU training container>
```

In the terminal of the container, run the following to begin training.

- For CPU

```
$ git clone -b v1.4.x https://github.com/apache/incubator-mxnet.git
python incubator-mxnet/example/image-classification/train_mnist.py
```

- For GPU

```
$ git clone -b v1.4.x https://github.com/apache/incubator-mxnet.git
python incubator-mxnet/example/image-classification/train_mnist.py --gpus 0
```

MXNet training with GluonCV

In the terminal of the container, run the following to begin training using GluonCV. GluonCV v0.6.0 is included in the Deep Learning Containers.

- For CPU

```
$ git clone -b v0.6.0 https://github.com/dmlc/gluon-cv.git
python gluon-cv/scripts/classification/cifar/train_cifar10.py --model resnet18_v1b
```

- For GPU

```
$ git clone -b v0.6.0 https://github.com/dmlc/gluon-cv.git
python gluon-cv/scripts/classification/cifar/train_cifar10.py --num-gpus 1 --model
 resnet18_v1b
```

Next steps

To learn inference on Amazon EC2 using MXNet with Deep Learning Containers, see Apache MXNet (Incubating) Inference (p. 8).

PyTorch training

To begin training with PyTorch from your Amazon EC2 instance, use the following commands to run the container. You must use **nvidia-docker** for GPU images.

- For CPU

```
$ docker run -it <CPU training container>
```

- For GPU

```
$ nvidia-docker run -it <GPU training container>
```

- If you have docker-ce version 19.03 or later, you can use the --gpus flag with docker:

```
$ docker run -it --gpus <GPU training container>
```

Run the following to begin training.

- For CPU

```
$ git clone https://github.com/pytorch/examples.git
$ python examples/mnist/main.py --no-cuda
```

- For GPU

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

PyTorch distributed GPU training with NVIDIA Apex

NVIDIA Apex is a PyTorch extension with utilities for mixed precision and distributed training. For more information on the utilities offered with Apex, see the NVIDIA Apex website. Apex is currently supported by Amazon EC2 instances in the following families:

- Amazon EC2 P3 Instances
- Amazon EC2 P2 Instances
- Amazon EC2 G4 Instances
- Amazon EC2 G3 Instances

To begin distributed training using NVIDIA Apex, run the following in the terminal of the GPU training container. This example requires at least two GPUs on your Amazon EC2 instance to run parallel distributed training.

```
$ git clone https://github.com/NVIDIA/apex.git && cd apex
$ python -m torch.distributed.launch --nproc_per_node=2 examples/simple/distributed/
distributed_data_parallel.py
```

Next steps

To learn inference on Amazon EC2 using PyTorch with Deep Learning Containers, see PyTorch Inference (p. 10).

Inference

This section shows how to run inference on AWS Deep Learning Containers for Amazon Elastic Compute Cloud using Apache MXNet (Incubating), PyTorch, TensorFlow, and TensorFlow 2. You can also use Elastic Inference to run inference with AWS Deep Learning Containers. For tutorials and more information on Elastic Inference, see Using AWS Deep Learning Containers with Elastic Inference on Amazon EC2.

For a complete list of Deep Learning Containers, refer to Deep Learning Containers Images (p. 76).

> **Note**
> MKL users: read the AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78) to get the best training or inference performance.

Contents

- TensorFlow Inference (p. 6)
- TensorFlow 2 Inference (p. 7)
- Apache MXNet (Incubating) Inference (p. 8)
- PyTorch Inference (p. 10)

TensorFlow Inference

To demonstrate how to use Deep Learning Containers for inference, this example uses a simple *half plus two* model with TensorFlow Serving. We recommend using the Deep Learning Base AMI for TensorFlow. After you log into your instance, run the following:

```
$ git clone -b r1.15 https://github.com/tensorflow/serving.git
$ cd serving
$ git checkout r1.15
```

Use the commands here to start TensorFlow Serving with the Deep Learning Containers for this model. Unlike the Deep Learning Containers for training, model serving starts immediately upon running the container and runs as a background process.

- For CPU instances:

  ```
  $ docker run -p 8500:8500 -p 8501:8501 --name tensorflow-inference --mount
   type=bind,source=$(pwd)/tensorflow_serving/servables/tensorflow/testdata/
  saved_model_half_plus_two_cpu,target=/models/saved_model_half_plus_two -e
   MODEL_NAME=saved_model_half_plus_two -d <cpu inference container>
  ```

  For example:

  ```
  $ docker run -p 8500:8500 -p 8501:8501 --name tensorflow-inference --mount
   type=bind,source=$(pwd)/tensorflow_serving/servables/tensorflow/testdata/
  saved_model_half_plus_two_cpu,target=/models/saved_model_half_plus_two -e
   MODEL_NAME=saved_model_half_plus_two -d 763104351884.dkr.ecr.us-east-1.amazonaws.com/
  tensorflow-inference:1.15.0-cpu-py36-ubuntu18.04
  ```

- For GPU instances:

```
$ nvidia-docker run -p 8500:8500 -p 8501:8501 --name tensorflow-inference --
mount type=bind,source=$(pwd)/tensorflow_serving/servables/tensorflow/testdata/
saved_model_half_plus_two_gpu,target=/models/saved_model_half_plus_two -e
 MODEL_NAME=saved_model_half_plus_two -d <gpu inference container>
```

For example:

```
$ nvidia-docker run -p 8500:8500 -p 8501:8501 --name tensorflow-inference
 --mount type=bind,source=$(pwd)/tensorflow_serving/servables/tensorflow/
testdata/saved_model_half_plus_two_gpu,target=/models/sad_model_half_plus_two -e
 MODEL_NAME=saved_model_half_plus_two -d 763104351884.dkr.ecr.us-east-1.amazonaws.com/
tensorflow-inference:1.15.0-gpu-py36-cu100-ubuntu18.04
```

Next, run inference with the Deep Learning Containers.

```
$ curl -d '{"instances": [1.0, 2.0, 5.0]}' -X POST http://127.0.0.1:8501/v1/models/
saved_model_half_plus_two:predict
```

The output is similar to the following:

```
{
    "predictions": [2.5, 3.0, 4.5
    ]
}
```

Note
If you want to debug the container's output, you can attach to it using the container name, as in the following command:

```
$ docker attach <your docker container name>
```

In this example you used `tensorflow-inference`.

TensorFlow 2 Inference

To demonstrate how to use Deep Learning Containers for inference, this example uses a simple *half plus two* model with TensorFlow 2 Serving. We recommend using the Deep Learning Base AMI for TensorFlow 2. After you log into your instance run the following.

```
$ git clone -b r2.0 https://github.com/tensorflow/serving.git
$ cd serving
```

Use the commands here to start TensorFlow Serving with the Deep Learning Containers for this model. Unlike the Deep Learning Containers for training, model serving starts immediately upon running the container and runs as a background process.

- For CPU instances:

```
$ docker run -p 8500:8500 -p 8501:8501 --name tensorflow-inference --mount
 type=bind,source=$(pwd)/tensorflow_serving/servables/tensorflow/testdata/
saved_model_half_plus_two_cpu,target=/models/saved_model_half_plus_two -e
 MODEL_NAME=saved_model_half_plus_two -d <cpu inference container>
```

  For example:

```
$ docker run -p 8500:8500 -p 8501:8501 --name tensorflow-inference --mount
 type=bind,source=$(pwd)/tensorflow_serving/servables/tensorflow/testdata/
saved_model_half_plus_two_cpu,target=/models/saved_model_half_plus_two -e
 MODEL_NAME=saved_model_half_plus_two -d 763104351884.dkr.ecr.us-east-1.amazonaws.com/
tensorflow-inference:2.0.0-cpu-py36-ubuntu18.04
```

- For GPU instances:

```
$ nvidia-docker run -p 8500:8500 -p 8501:8501 --name tensorflow-inference --
mount type=bind,source=$(pwd)/tensorflow_serving/servables/tensorflow/testdata/
saved_model_half_plus_two_gpu,target=/models/saved_model_half_plus_two -e
 MODEL_NAME=saved_model_half_plus_two -d <gpu inference container>
```

For example:

```
$ nvidia-docker run -p 8500:8500 -p 8501:8501 --name tensorflow-inference
 --mount type=bind,source=$(pwd)/tensorflow_serving/servables/tensorflow/
testdata/saved_model_half_plus_two_gpu,target=/models/sad_model_half_plus_two -e
 MODEL_NAME=saved_model_half_plus_two -d 763104351884.dkr.ecr.us-east-1.amazonaws.com/
tensorflow-inference:2.0.0-gpu-py36-cu100-ubuntu18.04
```

Note
Loading the GPU model server may take some time.

Next, run inference with the Deep Learning Containers.

```
$ curl -d '{"instances": [1.0, 2.0, 5.0]}' -X POST http://127.0.0.1:8501/v1/models/
saved_model_half_plus_two:predict
```

The output is similar to the following.

```
{
    "predictions": [2.5, 3.0, 4.5
    ]
}
```

Note
To debug the container's output, you can use the name to attach to it as shown in the following command:

```
$ docker attach <your docker container name>
```

This example used `tensorflow-inference`.

Apache MXNet (Incubating) Inference

To begin inference with Apache MXNet (Incubating), this example uses a pretrained model from a public S3 bucket.

For CPU instances, run the following command.

```
$ docker run -it --name mms -p 80:8080  -p 8081:8081 <your container image id> \
mxnet-model-server --start --mms-config /home/model-server/config.properties \
--models squeezenet=https://s3.amazonaws.com/model-server/models/squeezenet_v1.1/
squeezenet_v1.1.model
```

For GPU instances, run the following command:

```
$ nvidia-docker run -it --name mms -p 80:8080  -p 8081:8081 <your container image id> \
mxnet-model-server --start --mms-config /home/model-server/config.properties \
--models squeezenet=https://s3.amazonaws.com/model-server/models/squeezenet_v1.1/
squeezenet_v1.1.model
```

The configuration file is included in the container.

With your server started, you can now run inference from a different window by using the following command.

```
$ curl -O https://s3.amazonaws.com/model-server/inputs/kitten.jpg
curl -X POST http://127.0.0.1/predictions/squeezenet -T kitten.jpg
```

After you are done using your container, you can remove it using the following command:

```
$ docker rm -f mms
```

MXNet Inference with GluonCV

To begin inference using GluonCV, this example uses a pretrained model from a public S3 bucket.

For CPU instances, run the following command.

```
$ docker run -it --name mms -p 80:8080  -p 8081:8081 <your container image id> \
mxnet-model-server --start --mms-config /home/model-server/config.properties \
--models gluoncv_yolo3=https://dlc-samples.s3.amazonaws.com/mxnet/gluon/gluoncv_yolo3.mar
```

For GPU instances, run the following command.

```
$ nvidia-docker run -it --name mms -p 80:8080  -p 8081:8081 <your container image id> \
mxnet-model-server --start --mms-config /home/model-server/config.properties \
--models gluoncv_yolo3=https://dlc-samples.s3.amazonaws.com/mxnet/gluon/gluoncv_yolo3.mar
```

The configuration file is included in the container.

With your server started, you can now run inference from a different window by using the following command.

```
$ curl -O https://dlc-samples.s3.amazonaws.com/mxnet/gluon/dog.jpg
curl -X POST http://127.0.0.1/predictions/gluoncv_yolo3/predict -T dog.jpg
```

Your output should look like the following:

```
{
  "bicycle": [
    "[ 79.674225  87.403786 409.43515  323.12167 ]",
    "[ 98.69891  107.480446 200.0086   155.13412 ]"
  ],
  "car": [
    "[336.61322   56.533463 499.30566  125.0233   ]"
  ],
  "dog": [
    "[100.50538 156.50375 223.014   384.60873]"
  ]
```

```
}
```

After you are done using your container, you can remove it using this command.

```
$ docker rm -f mms
```

PyTorch Inference

Deep Learning Containers with PyTorch version 1.6 and later use TorchServe for inference calls. Deep Learning Containers with PyTorch version 1.5 and earlier use `mxnet-model-server` for inference calls.

PyTorch 1.6 and later

To run inference with PyTorch, this example uses a model pretrained on Imagenet from a public S3 bucket. Inference is served using TorchServe. For more information, see this blog on Deploying PyTorch inference with TorchServe.

For CPU instances:

```
$ docker run -itd --name torchserve -p 80:8080  -p 8081:8081 <your container image id> \
torchserve --start --ts-config /home/model-server/config.properties \
--models pytorch-densenet=https://torchserve.s3.amazonaws.com/mar_files/densenet161.mar
```

For GPU instances

```
$ nvidia-docker run -itd --name torchserve -p 80:8080  -p 8081:8081 <your container image
 id> \
torchserve --start --ts-config /home/model-server/config.properties \
--models pytorch-densenet=https://torchserve.s3.amazonaws.com/mar_files/densenet161.mar
```

If you have docker-ce version 19.03 or later, you can use the `--gpus` flag when you start Docker.

The configuration file is included in the container.

With your server started, you can now run inference from a different window by using the following.

```
$ curl -O https://s3.amazonaws.com/model-server/inputs/flower.jpg
curl -X POST http://127.0.0.1:80/predictions/pytorch-densenet -T flower.jpg
```

After you are done using your container, you can remove it using the following.

```
$ docker rm -f torchserve
```

PyTorch 1.5 and earlier

To run inference with PyTorch, this example uses a model pretrained on Imagenet from a public S3 bucket. Similar to MXNet containers, inference is served using mxnet-model-server, which can support any framework as the backend. For more information, see Model Server for Apache MXNet and this blog on Deploying PyTorch inference with MXNet Model Server.

For CPU instances:

```
$ docker run -itd --name mms -p 80:8080  -p 8081:8081 <your container image id> \
mxnet-model-server --start --mms-config /home/model-server/config.properties \
```

```
--models densenet=https://dlc-samples.s3.amazonaws.com/pytorch/multi-model-server/densenet/
densenet.mar
```

For GPU instances

```
$ nvidia-docker run -itd --name mms -p 80:8080  -p 8081:8081 <your container image id> \
mxnet-model-server --start --mms-config /home/model-server/config.properties \
--models densenet=https://dlc-samples.s3.amazonaws.com/pytorch/multi-model-server/densenet/
densenet.mar
```

If you have docker-ce version 19.03 or later, you can use the `--gpus` flag when you start Docker.

The configuration file is included in the container.

With your server started, you can now run inference from a different window by using the following.

```
$ curl -O https://s3.amazonaws.com/model-server/inputs/flower.jpg
curl -X POST http://127.0.0.1/predictions/densenet -T flower.jpg
```

After you are done using your container, you can remove it using the following.

```
$ docker rm -f mms
```

Next steps

To learn about using custom entrypoints with Deep Learning Containers on Amazon ECS, see Custom entrypoints (p. 32).

Custom Entrypoints

For some images, Deep Learning Containers uses a custom entrypoint script. If you want to use your own entrypoint, you can override the entrypoint as follows.

- To specify a custom entrypoint script to run, use this command.

  ```
  docker run --entrypoint=/path/to/custom_entrypoint_script -it <image> /bin/bash
  ```

- To set the entrypoint to be empty, use this command.

  ```
  docker run --entrypoint="" <image> /bin/bash
  ```

Amazon ECS tutorials

This section shows how to run training and inference on AWS Deep Learning Containers for Amazon ECS using MXNet, PyTorch, and TensorFlow.

Before starting the following tutorials, complete the steps in Amazon ECS setup (p. 12).

For a complete list of Deep Learning Containers, refer to Deep Learning Containers Images (p. 76).

> **Note**
> MKL users: Read the AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78) to get the best training or inference performance.

Contents

- Amazon ECS setup (p. 12)
- Training (p. 14)
- Inference (p. 20)
- Custom entrypoints (p. 32)

Amazon ECS setup

This section shows how to setup AWS Deep Learning Containers with Amazon Elastic Container Service.

Contents

- Prerequisites (p. 12)
- Setting up Amazon ECS for Deep Learning Containers (p. 12)

Prerequisites

This setup guide assumes that you have completed the following prerequisites:

- Install and configure the latest version of the AWS CLI. For more information about installing or upgrading the AWS CLI, see Installing the AWS Command Line Interface.
- Complete the steps in Setting Up with Amazon ECS.
- One of the following is true:
  - Your user has administrator access. For more information, see Setting Up with Amazon ECS.
  - Your user has the IAM permissions to create a service role. For more information, see Creating a Role to Delegate Permissions to an AWS Service.
  - A user with administrator access has manually created these IAM roles so that they're available on the account to be used. For more information, see Amazon ECS Service Scheduler IAM Role and Amazon ECS Container Instance IAM Role in the *Amazon Elastic Container Service Developer Guide*.
- The Amazon CloudWatch Logs IAM policy is added to the Amazon ECS Container Instance IAM role, which allows Amazon ECS to send logs to Amazon CloudWatch. For more information, see CloudWatch Logs IAM Policy in the *Amazon Elastic Container Service Developer Guide*.
- Generate a key pair. For more information see Amazon EC2 Key Pairs.
- Create a new security group or update an existing security group to have the ports open for your desired inference server.
  - For MXNet inference, ports 80 and 8081 open to TCP traffic.
  - For TensorFlow inference, ports 8501 and 8500 open to TCP traffic.

  For more information see Amazon EC2 Security Groups.

Setting up Amazon ECS for Deep Learning Containers

This section explains how to set up Amazon ECS to use Deep Learning Containers.

> **Important**
> If your account has already created the Amazon ECS service-linked role, then that role is used by default for your service unless you specify a role here. The service-linked role is required if your task definition uses the **awsvpc** network mode or if the service is configured to use any of the following: Service discovery, an external deployment controller, multiple target groups, or Elastic Inference accelerators. If this is the case, you should not specify a role here. For more information, see Using Service-Linked Roles for Amazon ECS in the *Amazon ECS Developer Guide*.

Run the following actions from your host.

1. Create an Amazon ECS cluster in the Region that contains the key pair and security group that you created previously.

```
aws ecs create-cluster --cluster-name ecs-ec2-training-inference --region us-east-1
```

2. Launch one or more Amazon EC2 instances into your cluster. For GPU-based work, refer to Working with GPUs on Amazon ECS in the *Amazon ECS Developer Guide* to guide your instance type selection. After you select your instance type, select an ECS-optimized AMI that fits your use case. For CPU-based work, you can use the Amazon Linux or Amazon Linux 2 ECS-optimized AMIs. For GPU-based work, you must use the ECS GPU-optimized AMI and a p2/p3 instance type. You can find the Amazon ECS-optimized AMI IDs at Amazon ECS-optimized AMIs. In this example, you launch one instance with a GPU-based AMI with 100 GB of disk size in us-east-1.

   a. Create a file named `my_script.txt` with the following contents. Reference the same cluster name that you created in the previous step.

   ```
   #!/bin/bash
   echo ECS_CLUSTER=ecs-ec2-training-inference >> /etc/ecs/ecs.config
   ```

   b. (Optional) Create a file named `my_mapping.txt` with the following content, which changes the size of the root volume after the instance is created.

   ```
   [
       {
           "DeviceName": "/dev/xvda",
           "Ebs": {
               "VolumeSize": 100
           }
       }
   ]
   ```

   c. Launch an Amazon EC2 instance with the Amazon ECS-optimized AMI and attach it to the cluster. Use the security group ID and key pair name that you created and replace them in the following command. To get the latest Amazon ECS-optimized AMI ID, see Amazon ECS-optimized AMIs in the *Amazon Elastic Container Service Developer Guide*.

   ```
   aws ec2 run-instances --image-id ami-0dfdeb4b6d47a87a2 \
                         --count 1 \
                         --instance-type p2.8xlarge \
                         --key-name key-pair-1234 \
                         --security-group-ids sg-abcd1234 \
                         --iam-instance-profile Name="ecsInstanceRole" \
                         --user-data file://my_script.txt \
                         --block-device-mapping file://my_mapping.txt \
                         --region us-east-1
   ```

   In the Amazon EC2 console, you can verify that this step was successful by the `instance-id` from the response.

You now have an Amazon ECS cluster with container instances running. Verify that the Amazon EC2 instances are registered with the cluster with the following steps.

To verify that the Amazon EC2 instance is registered with the cluster

1. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.
2. Select the cluster with your registered Amazon EC2 instances.
3. On the **Cluster** page, choose **ECS Instances**.

4. Verify that the **Agent Connected** value is **True** for the `instance-id` created in previous step. Also, note the CPU available and memory available from the console as these values can be useful in the following tutorials. It might take a few minutes to appear in the console.

Next steps

To learn about training and inference with Deep Learning Containers on Amazon ECS, see Amazon ECS tutorials (p. 11).

Training

This section shows how to run training on AWS Deep Learning Containers for Amazon Elastic Container Service using Apache MXNet (Incubating), PyTorch, TensorFlow, and TensorFlow 2.

For a complete list of Deep Learning Containers, refer to Deep Learning Containers Images (p. 76).

> **Note**
> MKL users: Read the AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78) to get the best training or inference performance.

> **Important**
> If your account has already created the Amazon ECS service-linked role, that role is used by default for your service unless you specify a role here. The service-linked role is required if your task definition uses the **awsvpc** network mode or if the service is configured to use service discovery. The role is also required if the service uses an external deployment controller, multiple target groups, or Elastic Inference accelerators in which case you should not specify a role here. For more information, see Using Service-Linked Roles for Amazon ECS in the *Amazon ECS Developer Guide*.

Contents

- TensorFlow training (p. 14)
- Apache MXNet (Incubating) training (p. 16)
- PyTorch training (p. 18)

TensorFlow training

Before you can run a task on your ECS cluster, you must register a task definition. Task definitions are lists of containers grouped together. The following example uses a sample Docker image that adds training scripts to Deep Learning Containers. You can use this script with either TensorFlow or TensorFlow 2. To use it with TensorFlow 2, change the Docker image to a TensorFlow 2 image.

1. Create a file named `ecs-deep-learning-container-training-taskdef.json` with the following contents.

   - For CPU

```
{
 "requiresCompatibilities": [
  "EC2"
 ],
 "containerDefinitions": [{
  "command": [
   "mkdir -p /test && cd /test && git clone https://github.com/fchollet/keras.git &&
 chmod +x -R /test/ && python keras/examples/mnist_cnn.py"
  ],
  "entryPoint": [
   "sh",
   "-c"
```

```
  ],
  "name": "tensorflow-training-container",
  "image": "763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-inference:1.15.2-
cpu-py36-ubuntu18.04",
  "memory": 4000,
  "cpu": 256,
  "essential": true,
  "portMappings": [{
   "containerPort": 80,
   "protocol": "tcp"
  }],
  "logConfiguration": {
   "logDriver": "awslogs",
   "options": {
    "awslogs-group": "awslogs-tf-ecs",
    "awslogs-region": "us-east-1",
    "awslogs-stream-prefix": "tf",
    "awslogs-create-group": "true"
   }
  }
 }],
 "volumes": [],
 "networkMode": "bridge",
 "placementConstraints": [],
 "family": "TensorFlow"
}
```

- For GPU

```
{
      "requiresCompatibilities": [
        "EC2"
      ],
      "containerDefinitions": [
        {
    "command": [
                "mkdir -p /test && cd /test && git clone https://github.com/fchollet/
keras.git && chmod +x -R /test/ && python keras/examples/mnist_cnn.py"
             ],
             "entryPoint": [
                "sh",
                "-c"
             ],
          "name": "tensorflow-training-container",
          "image": "763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-
training:1.15.2-gpu-py37-cu100-ubuntu18.04",
          "memory": 6111,
          "cpu": 256,
          "resourceRequirements" : [{
            "type" : "GPU",
            "value" : "1"
          }],
          "essential": true,
          "portMappings": [
            {
              "containerPort": 80,
              "protocol": "tcp"
            }
          ],
          "logConfiguration": {
              "logDriver": "awslogs",
              "options": {
                  "awslogs-group": "awslogs-tf-ecs",
                  "awslogs-region": "us-east-1",
                  "awslogs-stream-prefix": "tf",
```

```
            "awslogs-create-group": "true"
                }
            }
          }
        ],
        "volumes": [],
        "networkMode": "bridge",
        "placementConstraints": [],
        "family": "tensorflow-training"
    }
```

2. Register the task definition. Note the revision number in the output and use it in the next step.

```
aws ecs register-task-definition --cli-input-json file://ecs-deep-learning-container-training-taskdef.json
```

3. Create a task using the task definition. You need the revision number from the previous step and the name of the cluster you created during setup

```
aws ecs run-task --cluster ecs-ec2-training-inference --task-definition tf:1
```

4. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.
5. Select the `ecs-ec2-training-inference` cluster.
6. On the **Cluster** page, choose **Tasks**.
7. After your task is in a `RUNNING` state, choose the task identifier.
8. Under **Containers**, expand the container details.
9. Under **Log Configuration**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

Next steps

To learn inference on Amazon ECS using TensorFlow with Deep Learning Containers, see TensorFlow inference (p. 21).

Apache MXNet (Incubating) training

Before you can run a task on your Amazon Elastic Container Service cluster, you must register a task definition. Task definitions are lists of containers grouped together. The following example uses a sample Docker image that adds training scripts to Deep Learning Containers.

1. Create a file named `ecs-deep-learning-container-training-taskdef.json` with the following contents.

   - For CPU

```
{
   "requiresCompatibilities":[
      "EC2"
   ],
   "containerDefinitions":[
      {
         "command":[
            "git clone -b 1.4 https://github.com/apache/incubator-mxnet.git &&
 python /incubator-mxnet/example/image-classification/train_mnist.py"
         ],
         "entryPoint":[
            "sh",
            "-c"
```

```
         ],
         "name":"mxnet-training",
         "image":"763104351884.dkr.ecr.us-east-1.amazonaws.com/mxnet-training:1.6.0-
cpu-py36-ubuntu16.04",
         "memory":4000,
         "cpu":256,
         "essential":true,
         "portMappings":[
            {
               "containerPort":80,
               "protocol":"tcp"
            }
         ],
         "logConfiguration":{
            "logDriver":"awslogs",
            "options":{
               "awslogs-group":"/ecs/mxnet-training-cpu",
               "awslogs-region":"us-east-1",
               "awslogs-stream-prefix":"mnist",
               "awslogs-create-group":"true"
            }
         }
      }
   ],
   "volumes":[

   ],
   "networkMode":"bridge",
   "placementConstraints":[

   ],
   "family":"mxnet"
}
```

- For GPU

```
{
   "requiresCompatibilities":[
      "EC2"
   ],
   "containerDefinitions":[
      {
         "command":[
            "git clone -b 1.4 https://github.com/apache/incubator-mxnet.git &&
 python /incubator-mxnet/example/image-classification/train_mnist.py --gpus 0"
         ],
         "entryPoint":[
            "sh",
            "-c"
         ],
         "name":"mxnet-training",
         "image":"763104351884.dkr.ecr.us-east-1.amazonaws.com/mxnet-training:1.6.0-
gpu-py36-cu101-ubuntu16.04",
         "memory":4000,
         "cpu":256,
         "resourceRequirements":[
            {
               "type":"GPU",
               "value":"1"
            }
         ],
         "essential":true,
         "portMappings":[
            {
               "containerPort":80,
```

```
                "protocol":"tcp"
             }
          ],
          "logConfiguration":{
             "logDriver":"awslogs",
             "options":{
                "awslogs-group":"/ecs/mxnet-training-gpu",
                "awslogs-region":"us-east-1",
                "awslogs-stream-prefix":"mnist",
                "awslogs-create-group":"true"
             }
          }
       }
    ],
    "volumes":[

    ],
    "networkMode":"bridge",
    "placementConstraints":[

    ],
    "family":"mxnet-training"
}
```

2. Register the task definition. Note the revision number in the output and use it in the next step.

```
aws ecs register-task-definition --cli-input-json file://ecs-deep-learning-container-training-taskdef.json
```

3. Create a task using the task definition. You need the revision number from the previous step.

```
aws ecs run-task --cluster ecs-ec2-training-inference --task-definition mx:1
```

4. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.
5. Select the `ecs-ec2-training-inference` cluster.
6. On the **Cluster** page, choose **Tasks**.
7. After your task is in a `RUNNING` state, choose the task identifier.
8. Under **Containers**, expand the container details.
9. Under **Log Configuration**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

Next steps

To learn inference on Amazon ECS using MXNet with Deep Learning Containers, see Apache MXNet (Incubating) inference (p. 25).

PyTorch training

Before you can run a task on your Amazon ECS cluster, you must register a task definition. Task definitions are lists of containers grouped together. The following example uses a sample Docker image that adds training scripts to Deep Learning Containers.

1. Create a file named `ecs-deep-learning-container-training-taskdef.json` with the following contents.

   - For CPU

```
{
   "requiresCompatibilities":[
```

```
      "EC2"
   ],
   "containerDefinitions":[
      {
         "command":[
            "git clone https://github.com/pytorch/examples.git && python examples/
mnist/main.py --no-cuda"
         ],
         "entryPoint":[
            "sh",
            "-c"
         ],
         "name":"pytorch-training-container",
         "image":"763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-
training:1.5.1-cpu-py36-ubuntu16.04",
         "memory":4000,
         "cpu":256,
         "essential":true,
         "portMappings":[
            {
               "containerPort":80,
               "protocol":"tcp"
            }
         ],
         "logConfiguration":{
            "logDriver":"awslogs",
            "options":{
               "awslogs-group":"/ecs/pytorch-training-cpu",
               "awslogs-region":"us-east-1",
               "awslogs-stream-prefix":"mnist",
               "awslogs-create-group":"true"
            }
         }
      }
   ],
   "volumes":[

   ],
   "networkMode":"bridge",
   "placementConstraints":[

   ],
   "family":"pytorch"
}
```

- For GPU

```
{
      "requiresCompatibilities": [
        "EC2"
      ],
      "containerDefinitions": [
        {
    "command": [
                "git clone https://github.com/pytorch/examples.git && python
 examples/mnist/main.py"
             ],
             "entryPoint": [
                "sh",
                "-c"
             ],
          "name": "pytorch-training-container",
          "image": "763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-
training:1.5.1-gpu-py36-cu101-ubuntu16.04",
          "memory": 6111,
```

```
            "cpu": 256,
            "resourceRequirements" : [{
              "type" : "GPU",
              "value" : "1"
            }],
            "essential": true,
            "portMappings": [
              {
                "containerPort": 80,
                "protocol": "tcp"
              }
            ],
            "logConfiguration": {
                "logDriver": "awslogs",
                "options": {
                    "awslogs-group": "/ecs/pytorch-training-gpu",
                    "awslogs-region": "us-east-1",
                    "awslogs-stream-prefix": "mnist",
                        "awslogs-create-group": "true"
                }
            }
        }
    ],
    "volumes": [],
    "networkMode": "bridge",
    "placementConstraints": [],
    "family": "pytorch-training"
}
```

2. Register the task definition. Note the revision number in the output and use it in the next step.

```
aws ecs register-task-definition --cli-input-json file://ecs-deep-learning-container-training-taskdef.json
```

3. Create a task using the task definition. You need the revision identifier from the previous step.

```
aws ecs run-task --cluster ecs-ec2-training-inference --task-definition pytorch:1
```

4. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.
5. Select the `ecs-ec2-training-inference` cluster.
6. On the **Cluster** page, choose **Tasks**.
7. After your task is in a `RUNNING` state, choose the task identifier.
8. Under **Containers**, expand the container details.
9. Under **Log Configuration**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

Next steps

To learn inference on Amazon ECS using PyTorch with Deep Learning Containers, see PyTorch inference (p. 28).

Inference

This section shows how to run inference on AWS Deep Learning Containers for Amazon Elastic Container Service (Amazon ECS) using Apache MXNet (Incubating), PyTorch, TensorFlow, and TensorFlow 2. You can also use Elastic Inference to run inference with AWS Deep Learning Containers. For tutorials and more information on Elastic Inference, see Using AWS Deep Learning Containers with Elastic Inference on Amazon ECS.

For a complete list of Deep Learning Containers, see Deep Learning Containers Images (p. 76).

Note
MKL users: Read the AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78) to get the best training or inference performance.

Important
If your account has already created the Amazon ECS service-linked role, then that role is used by default for your service unless you specify a role here. The service-linked role is required if your task definition uses the **awsvpc** network mode. The role is also required if the service is configured to use service discovery, an external deployment controller, multiple target groups, or Elastic Inference accelerators in which case you should not specify a role here. For more information, see Using Service-Linked Roles for Amazon ECS in the *Amazon ECS Developer Guide*.

Contents

- TensorFlow inference (p. 21)
- Apache MXNet (Incubating) inference (p. 25)
- PyTorch inference (p. 28)

TensorFlow inference

The following examples use a sample Docker image that adds either CPU or GPU inference scripts to Deep Learning Containers from your host machine's command line.

CPU-based inference

Use the following example to run CPU-based inference.

1. Create a file named `ecs-dlc-cpu-inference-taskdef.json` with the following contents. You can use this with either TensorFlow or TensorFlow 2. To use it with TensorFlow 2, change the Docker image to a TensorFlow 2 image and clone the r2.0 serving repository branch instead of r1.15.

```
{
 "requiresCompatibilities": [
  "EC2"
 ],
 "containerDefinitions": [{
  "command": [
   "mkdir -p /test && cd /test && git clone -b r1.15 https://github.com/
tensorflow/serving.git && tensorflow_model_server --port=8500 --rest_api_port=8501
 --model_name=saved_model_half_plus_two --model_base_path=/test/serving/
tensorflow_serving/servables/tensorflow/testdata/saved_model_half_plus_two_cpu"
  ],
  "entryPoint": [
   "sh",
   "-c"
  ],
  "name": "tensorflow-inference-container",
  "image": "763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-inference:1.15.0-
cpu-py36-ubuntu18.04",
  "memory": 8111,
  "cpu": 256,
  "essential": true,
  "portMappings": [{
    "hostPort": 8500,
    "protocol": "tcp",
    "containerPort": 8500
   },
   {
```

```
    "hostPort": 8501,
    "protocol": "tcp",
    "containerPort": 8501
   },
   {
    "containerPort": 80,
    "protocol": "tcp"
   }
  ],
  "logConfiguration": {
   "logDriver": "awslogs",
   "options": {
    "awslogs-group": "/ecs/tensorflow-inference-gpu",
    "awslogs-region": "us-east-1",
    "awslogs-stream-prefix": "half-plus-two",
    "awslogs-create-group": "true"
   }
  }
 }],
 "volumes": [],
 "networkMode": "bridge",
 "placementConstraints": [],
 "family": "tensorflow-inference"
}
```

2. Register the task definition. Note the revision number in the output and use it in the next step.

```
aws ecs register-task-definition --cli-input-json file://ecs-dlc-cpu-inference-taskdef.json
```

3. Create an Amazon ECS service. When you specify the task definition, replace `revision_id` with the revision number of the task definition from the output of the previous step.

```
aws ecs create-service --cluster ecs-ec2-training-inference \
                       --service-name cli-ec2-inference-cpu \
                       --task-definition Ec2TFInference:revision_id \
                       --desired-count 1 \
                       --launch-type EC2 \
                       --scheduling-strategy="REPLICA" \
                       --region us-east-1
```

4. Verify the service and get the network endpoint by completing the following steps.

   a. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.
   b. Select the `ecs-ec2-training-inference` cluster.
   c. On the **Cluster** page, choose **Services** and then **cli-ec2-inference-cpu**.
   d. After your task is in a `RUNNING` state, choose the task identifier.
   e. Under **Containers**, expand the container details.
   f. Under **Name** and then **Network Bindings**, under **External Link** note the IP address for port 8501 and use it in the next step.
   g. Under **Log Configuration**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

5. To run inference, use the following command. Replace the external IP address with the external link IP address from the previous step.

```
curl -d '{"instances": [1.0, 2.0, 5.0]}' -X POST http://<External ip>:8501/v1/models/saved_model_half_plus_two:predict
```

   The following is sample output.

```
{
    "predictions": [2.5, 3.0, 4.5
    ]
}
```

Important

If you are unable to connect to the external IP address, be sure that your corporate firewall is not blocking non-standards ports, like 8501. You can try switching to a guest network to verify.

GPU-based inference

Use the following example to run GPU-based inference.

1. Create a file named `ecs-dlc-gpu-inference-taskdef.json` with the following contents. You can use this with either TensorFlow or TensorFlow 2. To use it with TensorFlow 2, change the Docker image to a TensorFlow 2 image and clone the r2.0 serving repository branch instead of r1.15.

```
{
 "requiresCompatibilities": [
  "EC2"
 ],
 "containerDefinitions": [{
  "command": [
   "mkdir -p /test && cd /test && git clone -b r1.15 https://github.com/
tensorflow/serving.git && tensorflow_model_server --port=8500 --rest_api_port=8501
 --model_name=saved_model_half_plus_two --model_base_path=/test/serving/
tensorflow_serving/servables/tensorflow/testdata/saved_model_half_plus_two_gpu"
  ],
  "entryPoint": [
   "sh",
   "-c"
  ],
  "name": "tensorflow-inference-container",
  "image": "763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-inference:1.15.0-
gpu-py36-cu100-ubuntu18.04",
  "memory": 8111,
  "cpu": 256,
  "resourceRequirements": [{
   "type": "GPU",
   "value": "1"
  }],
  "essential": true,
  "portMappings": [{
    "hostPort": 8500,
    "protocol": "tcp",
    "containerPort": 8500
   },
   {
    "hostPort": 8501,
    "protocol": "tcp",
    "containerPort": 8501
   },
   {
    "containerPort": 80,
    "protocol": "tcp"
   }
  ],
  "logConfiguration": {
   "logDriver": "awslogs",
   "options": {
```

```
      "awslogs-group": "/ecs/TFInference",
      "awslogs-region": "us-east-1",
      "awslogs-stream-prefix": "ecs",
      "awslogs-create-group": "true"
     }
    }
  }],
  "volumes": [],
  "networkMode": "bridge",
  "placementConstraints": [],
  "family": "TensorFlowInference"
}
```

2. Register the task definition. Note the revision number in the output and use it in the next step.

```
aws ecs register-task-definition --cli-input-json file://ecs-dlc-gpu-inference-
taskdef.json
```

3. Create an Amazon ECS service. When you specify the task definition, replace `revision_id` with the revision number of the task definition from the output of the previous step.

```
aws ecs create-service --cluster ecs-ec2-training-inference \
                       --service-name cli-ec2-inference-gpu \
                       --task-definition Ec2TFInference:revision_id \
                       --desired-count 1 \
                       --launch-type EC2 \
                       --scheduling-strategy="REPLICA" \
                       --region us-east-1
```

4. Verify the service and get the network endpoint by completing the following steps.

   a. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.
   b. Select the `ecs-ec2-training-inference` cluster.
   c. On the **Cluster** page, choose **Services** and then **cli-ec2-inference-cpu**.
   d. After your task is in a `RUNNING` state, choose the task identifier.
   e. Under **Containers**, expand the container details.
   f. Under **Name** and then **Network Bindings**, under **External Link** note the IP address for port 8501 and use it in the next step.
   g. Under **Log Configuration**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

5. To run inference, use the following command. Replace the external IP address with the external link IP address from the previous step.

```
curl -d '{"instances": [1.0, 2.0, 5.0]}' -X POST http://<External ip>:8501/v1/models/
saved_model_half_plus_two:predict
```

The following is sample output.

```
{
    "predictions": [2.5, 3.0, 4.5
    ]
}
```

Important
If you are unable to connect to the external IP address, be sure that your corporate firewall is not blocking non-standards ports, like 8501. You can try switching to a guest network to verify.

Apache MXNet (Incubating) inference

Before you can run a task on your Amazon ECS cluster, you must register a task definition. Task definitions are lists of containers grouped together. The following examples use a sample Docker image that adds either CPU or GPU inference scripts to Deep Learning Containers from your host machine's command line.

CPU-based inference

Use the following task definition to run CPU-based inference.

1. Create a file named `ecs-dlc-cpu-inference-taskdef.json` with the following contents.

```
{
 "requiresCompatibilities": [
  "EC2"
 ],
 "containerDefinitions": [{
  "command": [
   "mxnet-model-server --start --mms-config /home/model-server/config.properties
 --models squeezenet=https://s3.amazonaws.com/model-server/models/squeezenet_v1.1/
squeezenet_v1.1.model"
  ],
  "name": "mxnet-inference-container",
  "image": "763104351884.dkr.ecr.us-east-1.amazonaws.com/mxnet-inference:1.6.0-cpu-
py36-ubuntu16.04",
  "memory": 8111,
  "cpu": 256,
  "essential": true,
  "portMappings": [{
    "hostPort": 8081,
    "protocol": "tcp",
    "containerPort": 8081
   },
   {
    "hostPort": 80,
    "protocol": "tcp",
    "containerPort": 8080
   }
  ],
  "logConfiguration": {
   "logDriver": "awslogs",
   "options": {
    "awslogs-group": "/ecs/mxnet-inference-cpu",
    "awslogs-region": "us-east-1",
    "awslogs-stream-prefix": "squeezenet",
    "awslogs-create-group": "true"
   }
  }
 }],
 "volumes": [],
 "networkMode": "bridge",
 "placementConstraints": [],
 "family": "mxnet-inference"
}
```

2. Register the task definition. Note the revision number in the output and use it in the next step.

```
aws ecs register-task-definition --cli-input-json file://ecs-dlc-cpu-inference-
taskdef.json
```

3. Create an Amazon ECS service. When you specify the task definition, replace `revision_id` with the revision number of the task definition from the output of the previous step.

```
aws ecs create-service --cluster ecs-ec2-training-inference \
                       --service-name cli-ec2-inference-cpu \
                       --task-definition Ec2TFInference:revision_id \
                       --desired-count 1 \
                       --launch-type EC2 \
                       --scheduling-strategy REPLICA \
                       --region us-east-1
```

4. Verify the service and get the endpoint.

   a. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.

   b. Select the `ecs-ec2-training-inference` cluster.

   c. On the **Cluster** page, choose **Services** and then **cli-ec2-inference-cpu**.

   d. After your task is in a `RUNNING` state, choose the task identifier.

   e. Under **Containers**, expand the container details.

   f. Under **Name** and then **Network Bindings**, under **External Link** note the IP address for port 8081 and use it in the next step.

   g. Under **Log Configuration**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

5. To run inference, use the following command. Replace the `external IP` address with the external link IP address from the previous step.

```
curl -O https://s3.amazonaws.com/model-server/inputs/kitten.jpg
curl -X POST http://<External ip>/predictions/squeezenet -T kitten.jpg
```

The following is sample output.

```
[
  {
    "probability": 0.8582226634025574,
    "class": "n02124075 Egyptian cat"
  },
  {
    "probability": 0.09160050004720688,
    "class": "n02123045 tabby, tabby cat"
  },
  {
    "probability": 0.037487514317035675,
    "class": "n02123159 tiger cat"
  },
  {
    "probability": 0.0061649843119084835,
    "class": "n02128385 leopard, Panthera pardus"
  },
  {
    "probability": 0.003171598305925727,
    "class": "n02127052 lynx, catamount"
  }
]
```

Important

If you are unable to connect to the external IP address, be sure that your corporate firewall is not blocking non-standards ports, like 8081. You can try switching to a guest network to verify.

GPU-based inference

Use the following task definition to run GPU-based inference.

```
{
 "requiresCompatibilities": [
  "EC2"
 ],
 "containerDefinitions": [{
  "command": [
   "mxnet-model-server --start --mms-config /home/model-server/config.properties
 --models  squeezenet=https://s3.amazonaws.com/model-server/models/squeezenet_v1.1/
squeezenet_v1.1.model"
  ],
  "name": "mxnet-inference-container",
  "image": "763104351884.dkr.ecr.us-east-1.amazonaws.com/mxnet-inference:1.6.0-gpu-py36-
cu101-ubuntu16.04",
  "memory": 8111,
  "cpu": 256,
  "resourceRequirements": [{
   "type": "GPU",
   "value": "1"
  }],
  "essential": true,
  "portMappings": [{
    "hostPort": 8081,
    "protocol": "tcp",
    "containerPort": 8081
   },
   {
    "hostPort": 80,
    "protocol": "tcp",
    "containerPort": 8080
   }
  ],
  "logConfiguration": {
   "logDriver": "awslogs",
   "options": {
    "awslogs-group": "/ecs/mxnet-inference-gpu",
    "awslogs-region": "us-east-1",
    "awslogs-stream-prefix": "squeezenet",
    "awslogs-create-group": "true"
   }
  }
 }],
 "volumes": [],
 "networkMode": "bridge",
 "placementConstraints": [],
 "family": "mxnet-inference"
}
```

1. Use the following command to register the task definition. Note the output of the revision number and use it in the next step.

   ```
   aws ecs register-task-definition --cli-input-json file://<Task definition file>
   ```

2. To create the service, replace the `revision_id` with the output from the previous step in the following command.

   ```
   aws ecs create-service --cluster ecs-ec2-training-inference \
                       --service-name cli-ec2-inference-gpu \
                       --task-definition Ec2TFInference:<revision_id> \
                       --desired-count 1 \
   ```

```
                      --launch-type "EC2" \
                      --scheduling-strategy REPLICA \
                      --region us-east-1
```

3. Verify the service and get the endpoint.

   a. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.

   b. Select the `ecs-ec2-training-inference` cluster.

   c. On the **Cluster** page, choose **Services** and then **cli-ec2-inference-cpu**.

   d. After your task is in a `RUNNING` state, choose the task identifier.

   e. Under **Containers**, expand the container details.

   f. Under **Name** and then **Network Bindings**, under **External Link** note the IP address for port 8081 and use it in the next step.

   g. Under **Log Configuration**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

4. To run inference, use the following command. Replace the `external IP` address with the external link IP address from the previous step.

```
curl -O https://s3.amazonaws.com/model-server/inputs/kitten.jpg
curl -X POST http://<External ip>/predictions/squeezenet -T kitten.jpg
```

   The following is sample output.

```
[
  {
    "probability": 0.8582226634025574,
    "class": "n02124075 Egyptian cat"
  },
  {
    "probability": 0.09160050004720688,
    "class": "n02123045 tabby, tabby cat"
  },
  {
    "probability": 0.037487514317035675,
    "class": "n02123159 tiger cat"
  },
  {
    "probability": 0.0061649843119084835,
    "class": "n02128385 leopard, Panthera pardus"
  },
  {
    "probability": 0.003171598305925727,
    "class": "n02127052 lynx, catamount"
  }
]
```

   Important
   If you are unable to connect to the external IP address, be sure that your corporate firewall is not blocking non-standards ports, like 8081. You can try switching to a guest network to verify.

PyTorch inference

Before you can run a task on your Amazon ECS cluster, you must register a task definition. Task definitions are lists of containers grouped together. The following examples use a sample Docker image that adds either CPU or GPU inference scripts to Deep Learning Containers.

CPU-based inference

Use the following task definition to run CPU-based inference.

1. Create a file named `ecs-dlc-cpu-inference-taskdef.json` with the following contents.

```
{
    "requiresCompatibilities": [
        "EC2"
    ],
    "containerDefinitions": [{
        "command": [
            "mxnet-model-server --start --mms-config /home/model-server/
config.properties --models densenet=https://dlc-samples.s3.amazonaws.com/pytorch/multi-
model-server/densenet/densenet.mar"
        ],
        "name": "pytorch-inference-container",
        "image": "763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-inference:1.3.1-
cpu-py36-ubuntu16.04",
        "memory": 8111,
        "cpu": 256,
        "essential": true,
        "portMappings": [{
                "hostPort": 8081,
                "protocol": "tcp",
                "containerPort": 8081
            },
            {
                "hostPort": 80,
                "protocol": "tcp",
                "containerPort": 8080
            }
        ],
        "logConfiguration": {
            "logDriver": "awslogs",
            "options": {
                "awslogs-group": "/ecs/densenet-inference-cpu",
                "awslogs-region": "us-east-1",
                "awslogs-stream-prefix": "densenet",
                "awslogs-create-group": "true"
            }
        }
    }],
    "volumes": [],
    "networkMode": "bridge",
    "placementConstraints": [],
    "family": "pytorch-inference"
}
```

2. Register the task definition. Note the revision number in the output and use it in the next step.

```
aws ecs register-task-definition --cli-input-json file://ecs-dlc-cpu-inference-
taskdef.json
```

3. Create an Amazon ECS service. When you specify the task definition, replace `revision_id` with the revision number of the task definition from the output of the previous step.

```
aws ecs create-service --cluster ecs-ec2-training-inference \
                        --service-name cli-ec2-inference-cpu \
                        --task-definition Ec2PTInference:revision_id \
                        --desired-count 1 \
                        --launch-type EC2 \
                        --scheduling-strategy REPLICA \
```

```
                        --region us-east-1
```

4. Verify the service and get the network endpoint by completing the following steps.

   a. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.
   b. Select the `ecs-ec2-training-inference` cluster.
   c. On the **Cluster** page, choose **Services** and then **cli-ec2-inference-cpu**.
   d. After your task is in a `RUNNING` state, choose the task identifiter.
   e. Under **Containers**, expand the container details.
   f. Under **Name** and then **Network Bindings**, under **External Link** note the IP address for port 8081 and use it in the next step.
   g. Under **Log Configuration**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

5. To run inference, use the following command. Replace the `external IP` address with the external link IP address from the previous step.

```
curl -O https://s3.amazonaws.com/model-server/inputs/flower.jpg
curl -X POST http://<External ip>/predictions/densenet -T flower.jpg
```

> **Important**
> If you are unable to connect to the external IP address, be sure that your corporate firewall is not blocking non-standards ports, like 8081. You can try switching to a guest network to verify.

GPU-based inference

Use the following task definition to run GPU-based inference.

```
{
    "requiresCompatibilities": [
        "EC2"
    ],
    "containerDefinitions": [{
        "command": [
            "mxnet-model-server --start --mms-config /home/model-server/config.properties
 --models densenet=https://dlc-samples.s3.amazonaws.com/pytorch/multi-model-server/
densenet/densenet.mar"
        ],
        "name": "pytorch-inference-container",
        "image": "763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-inference:1.3.1-gpu-
py36-cu101-ubuntu16.04",
        "memory": 8111,
        "cpu": 256,
        "essential": true,
        "portMappings": [{
                "hostPort": 8081,
                "protocol": "tcp",
                "containerPort": 8081
            },
            {
                "hostPort": 80,
                "protocol": "tcp",
                "containerPort": 8080
            }
        ],
        "logConfiguration": {
            "logDriver": "awslogs",
            "options": {
```

```
                "awslogs-group": "/ecs/densenet-inference-cpu",
                "awslogs-region": "us-east-1",
                "awslogs-stream-prefix": "densenet",
                "awslogs-create-group": "true"
            }
        }
    }],
    "volumes": [],
    "networkMode": "bridge",
    "placementConstraints": [],
    "family": "pytorch-inference"
}
```

1. Use the following command to register the task definition. Note the output of the revision number and use it in the next step.

```
aws ecs register-task-definition --cli-input-json file://<Task definition file>
```

2. To create the service, replace the `revision_id` with the output from the previous step in the following command.

```
aws ecs create-service --cluster ecs-ec2-training-inference \
                    --service-name cli-ec2-inference-gpu \
                    --task-definition Ec2PTInference:<revision_id> \
                    --desired-count 1 \
                    --launch-type "EC2" \
                    --scheduling-strategy REPLICA \
                    --region us-east-1
```

3. Verify the service and get the network endpoint by completing the following steps.

   a. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.
   b. Select the `ecs-ec2-training-inference` cluster.
   c. On the **Cluster** page, choose **Services** and then **cli-ec2-inference-cpu**.
   d. Once your task is in a `RUNNING` state, choose the task identifier.
   e. Under **Containers**, expand the container details.
   f. Under **Name** and then **Network Bindings**, under **External Link** note the IP address for port 8081 and use it in the next step.
   g. Under **Log Configuration**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

4. To run inference, use the following command. Replace the `external IP` address with the external link IP address from the previous step.

```
curl -O https://s3.amazonaws.com/model-server/inputs/flower.jpg
curl -X POST http://<External ip>/predictions/densenet -T flower.jpg
```

Important
If you are unable to connect to the external IP address, be sure that your corporate firewall is not blocking non-standards ports, like 8081. You can try switching to a guest network to verify.

Next steps

To learn about using Custom Entrypoints with Deep Learning Containers on Amazon ECS, see Custom entrypoints (p. 32).

Custom entrypoints

For some images, Deep Learning Containers use a custom entrypoint script. If you want to use your own entrypoint, you can override the entrypoint as follows.

Modify the `entryPoint` parameter in the JSON file that includes your task definition. Include the file path to your custom entry point script. An example is shown here.

```
"entryPoint":[
           "sh",
           "-c",
           "/usr/local/bin/mxnet-model-server --start --foreground --mms-config /home/
model-server/config.properties --models densenet=https://dlc-samples.s3.amazonaws.com/
pytorch/multi-model-server/densenet/densenet.mar"],
```

Amazon EKS Tutorials

This section shows how to run training and inference on AWS Deep Learning Containers for EKS using MXNet, PyTorch, and TensorFlow. Some examples cover single node or multi-node training. Inference uses only single node configurations.

Before starting the following tutorials, complete the steps in Amazon EKS Setup (p. 32).

Contents

- Amazon EKS Setup (p. 32)
- Training (p. 38)
- Inference (p. 56)
- Custom Entrypoints (p. 72)
- Troubleshooting AWS Deep Learning Containers on EKS (p. 73)

Amazon EKS Setup

This guide explains how to setup a deep learning environment using Amazon Elastic Kubernetes Service (Amazon EKS) and AWS Deep Learning Containers. Using Amazon EKS you can scale a production-ready environment for multiple-node training and inference with Kubernetes containers.

If you're not familiar with Kubernetes or Amazon EKS yet, that's okay. This guide and the related Amazon EKS documentation shows how to use the family of Kubernetes tools. This guide assumes that you're already familiar with your deep learning framework's multiple-node implementations and how to set up an inference server outside of containers.

A deep learning container setup on Amazon EKS consists of one or more containers, forming a cluster. You can have dedicated cluster types, such as a cluster for training and a cluster for inference. You might also want different instance types for your clusters depending on the demands of your deep learning neural networks and models.

Contents

- Custom Images (p. 33)
- Licensing (p. 33)
- Configure Security Settings (p. 33)
- Gateway Node (p. 33)

- GPU Clusters (p. 35)
- CPU Clusters (p. 36)
- Test Your Clusters (p. 36)
- Manage Your Clusters (p. 36)
- Cleanup (p. 37)
- Next steps (p. 38)

Custom Images

Custom images are helpful if you want to load your own code or datasets and have them available on each node in your cluster. Examples are provided that use custom images. You can try them out to get started without creating your own.

- Building AWS Deep Learning Containers Custom Images (p. 77)

Licensing

To use GPU hardware, use an Amazon Machine Image that has the necessary GPU drivers. We recommend using the Amazon EKS-optimized AMI with GPU support, which is used in subsequent steps of this guide. This AMI includes non-AWS software that requires an end user license agreement (EULA). You must subscribe to the EKS-optimized AMI in the AWS Marketplace and accept the EULA before you can use the AMI in your worker node groups.

> **Important**
> To subscribe to the AMI, visit the AWS Marketplace.

Configure Security Settings

To use Amazon EKS you must have a user account that has access to several security permissions. These are set with the AWS Identity and Access Management (IAM) tool.

1. Create an IAM user or update an existing IAM user. Refer to the IAM documentation to learn about creating or editing an IAM user.
2. Get the credentials of this user. Under Users, select your user, select Security Credentials, select Create access key pair, and download the key pair or copy the information for use later.
3. Add policies to this IAM user. These provide the required access for Amazon EKS, IAM, and Amazon Elastic Compute Cloud (Amazon EC2).
4. Search for AmazonEKSAdminPolicy, and select the check box.
5. Search for AmazonCloudFormationPolicy, and select the check box.
6. Search for AmazonEC2FullAccess, and select the check box.
7. Search for IAMFullAccess , and select the check box.
8. Search AmazonEC2ContainerRegistryReadOnly, and select the check box.
9. Search AmazonEKS_CNI_Policy, and select the check box.
10. Search AmazonS3FullAccess, and select the check box.
11. Accept the changes.

Gateway Node

To setup an Amazon EKS cluster, use the open source tool, `eksctl`. We recommend that you use an Amazon EC2 instance with the Deep Learning Base AMI (Ubuntu) to allocate and control your cluster. You can run these tools locally on your computer or an Amazon EC2 instance that you already have running. However, to simplify this guide we assume you're using a Deep Learning Base AMI (DLAMI) with Ubuntu 16.04. We refer to this as your gateway node.

Note
Your `eksctl` version must be 1.12 or later.

Before you start, consider the location of your training data or where you want to run your cluster for responding to inference requests. Typically your data and cluster for training or inference should be in the same Region. Also, you spin up your gateway node in this same Region. You can follow this quick 10 minute tutorial that guides you to launch a DLAMI to use as your gateway node.

1. Login to your gateway node.
2. Install or upgrade AWS CLI. To access the required new Kubernetes features, you must have the latest version.

```
$ sudo pip install --upgrade awscli
```

3. Install eksctl by running the following commands. For more information about eksctl, see the eksctl blog post on the AWS Open Source Blog.

```
$ curl --silent \
--location "https://github.com/weaveworks/eksctl/releases/download/latest_release/
eksctl_$(uname -s)_amd64.tar.gz" \
| tar xz -C /tmp
$ sudo mv /tmp/eksctl /usr/local/bin
```

4. Install `kubectl` by running the following commands. For more information about kubectl, see Installing kubectl.

```
$ curl -o kubectl https://amazon-eks.s3-us-west-2.amazonaws.com/1.11.5/2018-12-06/bin/
linux/amd64/kubectl
$ chmod +x ./kubectl
$ mkdir -p $HOME/bin && cp ./kubectl $HOME/bin/kubectl && export PATH=$HOME/bin:$PATH
```

5. Install `aws-iam-authenticator` by running the following commands. For more information on aws-iam-authenticator, see Getting Started with Amazon EKS.

```
$ curl -o aws-iam-authenticator https://amazon-eks.s3-us-
west-2.amazonaws.com/1.11.5/2018-12-06/bin/linux/amd64/aws-iam-authenticator
$ chmod +x aws-iam-authenticator
$ cp ./aws-iam-authenticator $HOME/bin/aws-iam-authenticator && export PATH=$HOME/bin:
$PATH
```

6. Run `aws configure` for the IAM user from the Security Configuration section. You are copying the IAM user's AWS Access Key, then the AWS Secret Access Key that you accessed in the IAM console and pasting these into the prompts from `aws configure`.
7. Install ksonnet:

```
$ export KS_VER=0.13.1
$ export KS_PKG=ks_${KS_VER}_linux_amd64
$ wget -O /tmp/${KS_PKG}.tar.gz https://github.com/ksonnet/ksonnet/releases/download/v
${KS_VER}/${KS_PKG}.tar.gz
$ mkdir -p ${HOME}/bin
$ tar -xvf /tmp/$KS_PKG.tar.gz -C ${HOME}/bin
$ sudo mv ${HOME}/bin/$KS_PKG/ks /usr/local/bin
```

GPU Clusters

1. Examine the following command to create a cluster using a p3.8xlarge instance type. You will need to make modifications to it before you run it.

   - `name` is what you will use to manage your cluster. You can change `cluster-name` to be whatever name you like as long as there are no spaces or special characters.
   - `nodes` is the number of instances you want in your cluster. In this example, we're starting with three nodes.
   - `node-type` refers to instance class. You can choose a different instance class if you already know what kind will work best for your situation.
   - `timeout` and `*ssh-access *` can be left alone.
   - `ssh-public-key` is the name of the key that you want to use to login your worker nodes. Either use a security key you already use or create a new one but be sure to swap out the ssh-public-key with a key that was allocated for the Region you used. Note: You only need to provide the key name as seen in the 'key pairs' section of the Amazon EC2 Console.
   - `region` is the Amazon EC2 Region where the cluster will be launched. If you plan to use training data that resides in a specific Region (other than *`<us-east-1>`*) we recommend that you use the same Region. The ssh-public-key must have access to launch instances in this Region.

     Note
     The rest of this guide assumes *`<us-east-1>`* as the Region.
   - auto-kubeconfig can be left alone.

2. After you have made changes to the command, run it, and wait. It can take several minutes for a single node cluster, and will take even longer if you chose to create a large cluster.

```
$ eksctl create cluster <cluster-name> \
                      --version 1.11 \
                      --nodes 3 \
                      --node-type=<p3.8xlarge> \
                      --timeout=40m \
                      --ssh-access \
                      --ssh-public-key <key_pair_name> \
                      --region <us-east-1> \
                      --auto-kubeconfig
```

   You should see something similar to the following output:

```
EKS cluster "training-1" in "us-east-1" region is ready
```

3. Ideally the auto-kubeconfig should have configured your cluster. However, if you run into issues you can run the command below to set your kubeconfig. This command can also be used if you want to change your gateway node and manage your cluster from elsewhere.

```
$ aws eks --region <region> update-kubeconfig --name <cluster-name>
```

   You should see something similar to the following output:

```
Added new context arn:aws:eks:us-east-1:999999999999:cluster/training-1 to /home/
ubuntu/.kube/config
```

4. If you plan to use GPU instance types, make sure to run the following step to install the NVIDIA device plugin for Kubernetes:

```
$ kubectl apply -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v1.12/
nvidia-device-plugin.yml
```

5. Verify the GPUs available on each node in your cluster

```
$ kubectl get nodes "-o=custom-
columns=NAME:.metadata.name,GPU:.status.allocatable.nvidia\.com/gpu"
```

CPU Clusters

Refer to the previous section's discussion on using the **eksctl** command to launch a GPU cluster, and modify `node-type` to use a CPU instance type.

Test Your Clusters

1. You can run a **kubectl** command on the cluster to check its status. Try the command to make sure it is picking up the current cluster you want to manage.

```
$ kubectl get nodes -o wide
```

2. Take a look in ~/.kube. This directory has the kubeconfig files for the various clusters configured from your gateway node. If you browse further into the folder you can find ~/.kube/eksctl/clusters - This holds the kubeconfig file for clusters created using eksctl. This file has some details which you ideally shouldn't have to modify, since the tools are generating and updating the configurations for you, but it is good to reference when troubleshooting.
3. Verify that the cluster is active.

```
$ aws eks --region <region> describe-cluster --name <cluster-name> --query
 cluster.status
```

You should see the following output:

```
"ACTIVE"
```

4. Verify the kubectl context if you have multiple clusters set up from the same host instance. Sometimes it helps to make sure that the default context found by **kubectl** is set properly. Check this using the following command:

```
$ kubectl config get-contexts
```

5. If the context is not set as expected, fix this using the following command:

```
$ aws eks --region <region> update-kubeconfig --name <cluster-name>
```

Manage Your Clusters

When you want to control or query a cluster you can address it by the configuration file using the kubeconfig parameter. This is useful when you have more than one cluster. For example, if you have a separate cluster called “training-gpu-1” you can call the **get pods** command on it by passing the configuration file as a parameter as follows:

```
$ kubectl --kubeconfig=/home/ubuntu/.kube/eksctl/clusters/training-gpu-1 get pods
```

It is useful to note that you can run this same command without the kubeconfig parameter and it will report status on your current actively controlled cluster.

```
$ kubectl get pods
```

If you setup multiple clusters and they have yet to have the NVIDIA plugin installed, you can install it this way:

```
$ kubectl --kubeconfig=/home/ubuntu/.kube/eksctl/clusters/training-gpu-1 apply -f https://
raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v1.12/nvidia-device-plugin.yml
```

You also change the active cluster by updating the kubeconfig, passing the name of the cluster you want to manage. The following command updates kubeconfig and removes the need to use the kubeconfig parameter.

```
$ aws eks —region us-east-1 update-kubeconfig —name training-gpu-1
```

If you follow all of the examples in this guide, you will frequently switch between active clusters. This is so you can orchestrate training or inference or use different frameworks running on different clusters.

Cleanup

When you're done using the cluster, delete it to avoid incurring additional costs.

```
$ eksctl delete cluster --name=<cluster-name>
```

To delete only a pod, run the following:

```
$ kubectl delete pods <name>
```

To reset the secret for access to the cluster, run the following:

```
$ kubectl delete secret ${SECRET} -n ${NAMESPACE} || true
```

To delete a `nodegroup` attached to a cluster, run the following:

```
$ eksctl delete nodegroup --name <cluster_name>
```

To attach a `nodegroup` to a cluster, run the following:

```
$ eksctl create nodegroup
                      --cluster <cluster-name> \
                      --node-ami <ami_id> \
                      --nodes <num_nodes> \
                      --node-type=<instance_type> \
                      --timeout=40m \
                      --ssh-access \
                      --ssh-public-key <key_pair_name> \
                      --region <us-east-1> \
```

```
                       --auto-kubeconfig
```

Next steps

To learn about training and inference with Deep Learning Containers on Amazon EKS, see Amazon EKS Tutorials (p. 32).

Training

After you have a cluster running, you can try a training job. This section will guide you through Apache MXNet (Incubating), PyTorch, TensorFlow, and TensorFlow 2 training examples.

Contents

- CPU Training (p. 38)
- GPU Training (p. 42)
- Distributed GPU Training (p. 47)

CPU Training

This section is for training on CPU-based containers.

For a complete list of Deep Learning Containers, see Deep Learning Containers Images (p. 76). For tips about the best configuration settings if you're using the Intel Math Kernel Library (MKL), see AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78).

Contents

- Apache MXNet (Incubating) CPU training (p. 38)
- TensorFlow CPU training (p. 39)
- PyTorch CPU training (p. 41)

Apache MXNet (Incubating) CPU training

This tutorial guides you on training with Apache MXNet (Incubating) on your single node CPU cluster.

1. Create a pod file for your cluster. A pod file will provide the instructions about what the cluster should run. This pod file will download the MXNet repository and run an MNIST example. Open **vi** or **vim** and copy and past the following content. Save this file as `mxnet.yaml`.

```
apiVersion: v1
kind: Pod
metadata:
  name: mxnet-training
spec:
  restartPolicy: OnFailure
  containers:
  - name: mxnet-training
    image:  763104351884.dkr.ecr.us-east-1.amazonaws.com/mxnet-inference:1.6.0-cpu-
py36-ubuntu16.04
    command: ["/bin/sh","-c"]
    args: ["git clone -b v1.4.x https://github.com/apache/incubator-mxnet.git &&
 python ./incubator-mxnet/example/image-classification/train_mnist.py"]
```

2. Assign the pod file to the cluster using **kubectl**.

```
$ kubectl create -f mxnet.yaml
```

3. You should see the following output:

```
pod/mxnet-training created
```

4. Check the status. The name of the job "mxnet-training" was in the mxnet.yaml file. It will now appear in the status. If you're running any other tests or have previously run something, it appears in this list. Run this several times until you see the status change to "Running".

```
$ kubectl get pods
```

You should see the following output:

```
NAME READY STATUS RESTARTS AGE
mxnet-training 0/1 Running 8 19m
```

5. Check the logs to see the training output.

```
$ kubectl logs mxnet-training
```

You should see something similar to the following output:

```
Cloning into 'incubator-mxnet'...
INFO:root:Epoch[0] Batch [0-100]    Speed: 18437.78 samples/sec    accuracy=0.777228
INFO:root:Epoch[0] Batch [100-200]    Speed: 16814.68 samples/sec    accuracy=0.907188
INFO:root:Epoch[0] Batch [200-300]    Speed: 18855.48 samples/sec    accuracy=0.926719
INFO:root:Epoch[0] Batch [300-400]    Speed: 20260.84 samples/sec    accuracy=0.938438
INFO:root:Epoch[0] Batch [400-500]    Speed: 9062.62 samples/sec    accuracy=0.938594
INFO:root:Epoch[0] Batch [500-600]    Speed: 10467.17 samples/sec    accuracy=0.945000
INFO:root:Epoch[0] Batch [600-700]    Speed: 11082.03 samples/sec    accuracy=0.954219
INFO:root:Epoch[0] Batch [700-800]    Speed: 11505.02 samples/sec    accuracy=0.956875
INFO:root:Epoch[0] Batch [800-900]    Speed: 9072.26 samples/sec    accuracy=0.955781
INFO:root:Epoch[0] Train-accuracy=0.923424
...
```

6. Check the logs to watch the training progress. You can also continue to check "**get pods**" to refresh the status. When the status changes to "`Completed`", the training job is done.

Next steps

To learn CPU-based inference on Amazon EKS using MXNet with Deep Learning Containers, see Apache MXNet (Incubating) CPU inference (p. 57).

TensorFlow CPU training

This tutorial guides you on training TensorFlow models on your single node CPU cluster.

1. Create a pod file for your cluster. A pod file will provide the instructions about what the cluster should run. This pod file will download Keras and run a Keras example. This example uses the TensorFlow framework. Open **vi** or **vim** and copy and paste the following content. Save this file as `tf.yaml`. You can use this with either TensorFlow or TensorFlow 2. To use it with TensorFlow 2, change the Docker image to a TensorFlow 2 image.

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: tensorflow-training
spec:
  restartPolicy: OnFailure
  containers:
  - name: tensorflow-training
    image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-inference:1.15.2-
cpu-py36-ubuntu18.04
    command: ["/bin/sh","-c"]
    args: ["git clone https://github.com/fchollet/keras.git && python /keras/examples/
mnist_cnn.py"]
```

2. Assign the pod file to the cluster using **kubectl**.

```
$ kubectl create -f tf.yaml
```

3. You should see the following output:

```
pod/tensorflow-training created
```

4. Check the status. The name of the job “tensorflow-training” was in the tf.yaml file. It will now appear in the status. If you're running any other tests or have previously run something, it appears in this list. Run this several times until you see the status change to “Running”.

```
$ kubectl get pods
```

   You should see the following output:

```
NAME READY STATUS RESTARTS AGE
tensorflow-training 0/1 Running 8 19m
```

5. Check the logs to see the training output.

```
$ kubectl logs tensorflow-training
```

   You should see something similar to the following output:

```
Cloning into 'keras'...
Using TensorFlow backend.
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz

    8192/11490434 [..............................] - ETA: 0s
 6479872/11490434 [===============>..............] - ETA: 0s
 8740864/11490434 [=====================>........] - ETA: 0s
11493376/11490434 [==============================] - 0s 0us/step
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
2019-03-19 01:52:33.863598: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your
 CPU supports instructions that this TensorFlow binary was not compiled to use: AVX512F
2019-03-19 01:52:33.867616: I tensorflow/core/common_runtime/process_util.cc:69]
 Creating new thread pool with default inter op setting: 2. Tune using
 inter_op_parallelism_threads for best performance.

  128/60000 [..............................] - ETA: 10:43 - loss: 2.3076 - acc: 0.0625
  256/60000 [..............................] - ETA: 5:59 - loss: 2.2528 - acc: 0.1445
  384/60000 [..............................] - ETA: 4:24 - loss: 2.2183 - acc: 0.1875
  512/60000 [..............................] - ETA: 3:35 - loss: 2.1652 - acc: 0.1953
```

```
640/60000 [.............................] - ETA: 3:05 - loss: 2.1078 - acc: 0.2422
...
```

6. You can check the logs to watch the training progress. You can also continue to check "**get pods**" to refresh the status. When the status changes to "`Completed`" you will know that the training job is done.

Next steps

To learn CPU-based inference on Amazon EKS using TensorFlow with Deep Learning Containers, see TensorFlow CPU inference (p. 58).

PyTorch CPU training

This tutorial guides you on training with PyTorch on your single node CPU cluster.

1. Create a pod file for your cluster. A pod file will provide the instructions about what the cluster should run. This pod file will download the PyTorch repository and run an MNIST example. Open **vi** or **vim**, then copy and paste the following content. Save this file as `pytorch.yaml`.

```
apiVersion: v1
kind: Pod
metadata:
  name: pytorch-training
spec:
  restartPolicy: OnFailure
  containers:
  - name: pytorch-training
    image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-training:1.5.1-cpu-
py36-ubuntu16.04
    command:
      - "/bin/sh"
      - "-c"
    args:
    - "git clone https://github.com/pytorch/examples.git && python examples/mnist/
main.py --no-cuda"
    env:
    - name: OMP_NUM_THREADS
      value: "36"
    - name: KMP_AFFINITY
      value: "granularity=fine,verbose,compact,1,0"
    - name: KMP_BLOCKTIME
      value: "1"
```

2. Assign the pod file to the cluster using **kubectl**.

```
$ kubectl create -f pytorch.yaml
```

3. You should see the following output:

```
pod/pytorch-training created
```

4. Check the status. The name of the job "pytorch-training" was in the pytorch.yaml file. It will now appear in the status. If you're running any other tests or have previously run something, it appears in this list. Run this several times until you see the status change to "Running".

```
$ kubectl get pods
```

You should see the following output:

```
NAME READY STATUS RESTARTS AGE
pytorch-training 0/1 Running 8 19m
```

5. Check the logs to see the training output.

```
$ kubectl logs pytorch-training
```

You should see something similar to the following output:

```
Cloning into 'examples'...
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ../data/
MNIST/raw/train-images-idx3-ubyte.gz
9920512it [00:00, 40133996.38it/s]
Extracting ../data/MNIST/raw/train-images-idx3-ubyte.gz to ../data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ../data/
MNIST/raw/train-labels-idx1-ubyte.gz
Extracting ../data/MNIST/raw/train-labels-idx1-ubyte.gz to ../data/MNIST/raw
32768it [00:00, 831315.84it/s]
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ../data/
MNIST/raw/t10k-images-idx3-ubyte.gz
1654784it [00:00, 13019129.43it/s]
Extracting ../data/MNIST/raw/t10k-images-idx3-ubyte.gz to ../data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ../data/
MNIST/raw/t10k-labels-idx1-ubyte.gz
8192it [00:00, 337197.38it/s]
Extracting ../data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/MNIST/raw
Processing...
Done!
Train Epoch: 1 [0/60000 (0%)]    Loss: 2.300039
Train Epoch: 1 [640/60000 (1%)]    Loss: 2.213470
Train Epoch: 1 [1280/60000 (2%)]    Loss: 2.170460
Train Epoch: 1 [1920/60000 (3%)]    Loss: 2.076699
Train Epoch: 1 [2560/60000 (4%)]    Loss: 1.868078
Train Epoch: 1 [3200/60000 (5%)]    Loss: 1.414199
Train Epoch: 1 [3840/60000 (6%)]    Loss: 1.000870
```

6. Check the logs to watch the training progress. You can also continue to check "**get pods**" to refresh the status. When the status changes to "`Completed`" you will know that the training job is done.

See EKS Cleanup for information on cleaning up a cluster after you're done using it.

Next steps

To learn CPU-based inference on Amazon EKS using PyTorch with Deep Learning Containers, see PyTorch CPU inference (p. 62).

GPU Training

This section is for training on GPU-based clusters.

For a complete list of Deep Learning Containers, refer to Deep Learning Containers Images (p. 76). For tips about the best configuration settings if you're using the Intel Math Kernel Library (MKL), see AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78).

Contents

- Apache MXNet (Incubating) GPU training (p. 43)
- TensorFlow GPU training (p. 44)

- PyTorch GPU training (p. 45)

Apache MXNet (Incubating) GPU training

This tutorial guides you on training with Apache MXNet (Incubating) on your single node GPU cluster.

1. Create a pod file for your cluster. A pod file will provide the instructions about what the cluster should run. This pod file will download the MXNet repository and run an MNIST example. Open **vi** or **vim** and copy and past the following content. Save this file as `mxnet.yaml`.

```
apiVersion: v1
kind: Pod
metadata:
  name: mxnet-training
spec:
  restartPolicy: OnFailure
  containers:
  - name: mxnet-training
    image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/mxnet-training:1.6.0-gpu-py36-cu101-ubuntu16.04
    command: ["/bin/sh","-c"]
    args: ["git clone -b v1.4.x https://github.com/apache/incubator-mxnet.git && python ./incubator-mxnet/example/image-classification/train_mnist.py"]
```

2. Assign the pod file to the cluster using **kubectl**.

```
$ kubectl create -f mxnet.yaml
```

3. You should see the following output:

```
pod/mxnet-training created
```

4. Check the status. The name of the job “tensorflow-training” was in the tf.yaml file. It will now appear in the status. If you're running any other tests or have previously run something, it will appear in this list. Run this several times until you see the status change to “`Running`”.

```
$ kubectl get pods
```

   You should see the following output:

```
NAME READY STATUS RESTARTS AGE
mxnet-training 0/1 Running 8 19m
```

5. Check the logs to see the training output.

```
$ kubectl logs mxnet-training
```

   You should see something similar to the following output:

```
Cloning into 'incubator-mxnet'...
INFO:root:Epoch[0] Batch [0-100]    Speed: 18437.78 samples/sec    accuracy=0.777228
INFO:root:Epoch[0] Batch [100-200]    Speed: 16814.68 samples/sec    accuracy=0.907188
INFO:root:Epoch[0] Batch [200-300]    Speed: 18855.48 samples/sec    accuracy=0.926719
INFO:root:Epoch[0] Batch [300-400]    Speed: 20260.84 samples/sec    accuracy=0.938438
INFO:root:Epoch[0] Batch [400-500]    Speed: 9062.62 samples/sec    accuracy=0.938594
INFO:root:Epoch[0] Batch [500-600]    Speed: 10467.17 samples/sec    accuracy=0.945000
INFO:root:Epoch[0] Batch [600-700]    Speed: 11082.03 samples/sec    accuracy=0.954219
INFO:root:Epoch[0] Batch [700-800]    Speed: 11505.02 samples/sec    accuracy=0.956875
```

```
INFO:root:Epoch[0] Batch [800-900]    Speed: 9072.26 samples/sec    accuracy=0.955781
INFO:root:Epoch[0] Train-accuracy=0.923424
...
```

6. Check the logs to watch the training progress. You can also continue to check "**get pods**" to refresh the status. When the status changes to "`Completed`", the training job is done.

Next steps

To learn GPU-based inference on Amazon EKS using MXNet with Deep Learning Containers, see Apache MXNet (Incubating) GPU inference (p. 64).

TensorFlow GPU training

This tutorial guides you on training TensorFlow models on your single node GPU cluster.

1. Create a pod file for your cluster. A pod file will provide the instructions about what the cluster should run. This pod file will download Keras and run a Keras example. This example uses the TensorFlow framework. Open **vi** or **vim** and copy and past the following content. Save this file as `tf.yaml`. You can use this with either TensorFlow or TensorFlow 2. To use it with TensorFlow 2, change the Docker image to a TensorFlow 2 image.

```
apiVersion: v1
kind: Pod
metadata:
  name: tensorflow-training
spec:
  restartPolicy: OnFailure
  containers:
  - name: tensorflow-training
    image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-training:1.15.2-gpu-py37-cu100-ubuntu18.04
    command: ["/bin/sh","-c"]
    args: ["git clone https://github.com/fchollet/keras.git && python /keras/examples/mnist_cnn.py"]
    resources:
      limits:
        nvidia.com/gpu: 1
```

2. Assign the pod file to the cluster using **kubectl**.

```
$ kubectl create -f tf.yaml
```

3. You should see the following output:

```
pod/tensorflow-training created
```

4. Check the status. The name of the job "tensorflow-training" was in the tf.yaml file. It will now appear in the status. If you're running any other tests or have previously run something, it appears in this list. Run this several times until you see the status change to "`Running`".

```
$ kubectl get pods
```

You should see the following output:

```
NAME READY STATUS RESTARTS AGE
tensorflow-training 0/1 Running 8 19m
```

5. Check the logs to see the training output.

```
$ kubectl logs tensorflow-training
```

You should see something similar to the following output:

```
Cloning into 'keras'...
Using TensorFlow backend.
Downloading data from https://s3.amazonaws.com/img-datasets/mnist.npz

    8192/11490434 [..............................] - ETA: 0s
 6479872/11490434 [===============>..............] - ETA: 0s
 8740864/11490434 [=====================>........] - ETA: 0s
11493376/11490434 [==============================] - 0s 0us/step
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
2019-03-19 01:52:33.863598: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your
 CPU supports instructions that this TensorFlow binary was not compiled to use: AVX512F
2019-03-19 01:52:33.867616: I tensorflow/core/common_runtime/process_util.cc:69]
 Creating new thread pool with default inter op setting: 2. Tune using
 inter_op_parallelism_threads for best performance.

  128/60000 [..............................] - ETA: 10:43 - loss: 2.3076 - acc: 0.0625
  256/60000 [..............................] - ETA: 5:59 - loss: 2.2528 - acc: 0.1445
  384/60000 [..............................] - ETA: 4:24 - loss: 2.2183 - acc: 0.1875
  512/60000 [..............................] - ETA: 3:35 - loss: 2.1652 - acc: 0.1953
  640/60000 [..............................] - ETA: 3:05 - loss: 2.1078 - acc: 0.2422
  ...
```

6. Check the logs to watch the training progress. You can also continue to check "**get pods**" to refresh the status. When the status changes to "`Completed`", the training job is done.

Next steps

To learn GPU-based inference on Amazon EKS using TensorFlow with Deep Learning Containers, see TensorFlow GPU inference (p. 66).

PyTorch GPU training

This tutorial guides you on training with PyTorch on your single node GPU cluster.

1. Create a pod file for your cluster. A pod file will provide the instructions about what the cluster should run. This pod file will download the PyTorch repository and run an MNIST example. Open **vi** or **vim**, then copy and paste the following content. Save this file as `pytorch.yaml`.

```
apiVersion: v1
kind: Pod
metadata:
  name: pytorch-training
spec:
  restartPolicy: OnFailure
  containers:
  - name: pytorch-training
    image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-training:1.5.1-gpu-
py36-cu101-ubuntu16.04
    command:
      - "/bin/sh"
      - "-c"
    args:
```

```
    - "git clone https://github.com/pytorch/examples.git && python examples/mnist/
main.py --no-cuda"
    env:
    - name: OMP_NUM_THREADS
      value: "36"
    - name: KMP_AFFINITY
      value: "granularity=fine,verbose,compact,1,0"
    - name: KMP_BLOCKTIME
      value: "1"
```

2. Assign the pod file to the cluster using **kubectl**.

```
$ kubectl create -f pytorch.yaml
```

3. You should see the following output:

```
pod/pytorch-training created
```

4. Check the status. The name of the job "pytorch-training" was in the pytorch.yaml file. It will now appear in the status. If you're running any other tests or have previously run something, it appears in this list. Run this several times until you see the status change to "`Running`".

```
$ kubectl get pods
```

   You should see the following output:

```
NAME READY STATUS RESTARTS AGE
pytorch-training 0/1 Running 8 19m
```

5. Check the logs to see the training output.

```
$ kubectl logs pytorch-training
```

   You should see something similar to the following output:

```
Cloning into 'examples'...
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ../data/
MNIST/raw/train-images-idx3-ubyte.gz
9920512it [00:00, 40133996.38it/s]
Extracting ../data/MNIST/raw/train-images-idx3-ubyte.gz to ../data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ../data/
MNIST/raw/train-labels-idx1-ubyte.gz
Extracting ../data/MNIST/raw/train-labels-idx1-ubyte.gz to ../data/MNIST/raw
32768it [00:00, 831315.84it/s]
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ../data/
MNIST/raw/t10k-images-idx3-ubyte.gz
1654784it [00:00, 13019129.43it/s]
Extracting ../data/MNIST/raw/t10k-images-idx3-ubyte.gz to ../data/MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ../data/
MNIST/raw/t10k-labels-idx1-ubyte.gz
8192it [00:00, 337197.38it/s]
Extracting ../data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ../data/MNIST/raw
Processing...
Done!
Train Epoch: 1 [0/60000 (0%)]    Loss: 2.300039
Train Epoch: 1 [640/60000 (1%)]    Loss: 2.213470
Train Epoch: 1 [1280/60000 (2%)]    Loss: 2.170460
Train Epoch: 1 [1920/60000 (3%)]    Loss: 2.076699
Train Epoch: 1 [2560/60000 (4%)]    Loss: 1.868078
Train Epoch: 1 [3200/60000 (5%)]    Loss: 1.414199
```

```
Train Epoch: 1 [3840/60000 (6%)]    Loss: 1.000870
```

6. Check the logs to watch the training progress. You can also continue to check "**get pods**" to refresh the status. When the status changes to "`Completed`", the training job is done.

See EKS Cleanup for information on cleaning up a cluster after you're done using it.

Next steps

To learn GPU-based inference on Amazon EKS using PyTorch with Deep Learning Containers, see PyTorch GPU inference (p. 70).

Distributed GPU Training

This section is for running distributed training on multi-node GPU clusters.

For a complete list of Deep Learning Containers, refer to Deep Learning Containers Images (p. 76).

Contents

- Apache MXNet (Incubating) distributed GPU training (p. 47)
- Apache MXNet (Incubating) with Horovod distributed GPU training (p. 47)
- TensorFlow with Horovod distributed GPU training (p. 53)
- PyTorch distributed GPU training (p. 54)

To run distributed training on EKS, you will be using Kubeflow. The Kubeflow project is dedicated to making deployments of machine learning (ML) workflows on Kubernetes simple, portable and scalable.

Setup your cluster for distributed training by installing Kubeflow

1. Install Kubeflow.

```
$ export KUBEFLOW_VERSION=0.4.1
$ curl https://raw.githubusercontent.com/kubeflow/kubeflow/v${KUBEFLOW_VERSION}/
scripts/download.sh | bash
```

2. When using Kubeflow packages, you will soon run into Github API limits. You need to create a Github token and export it as follows. You do not need to select any scopes.

```
$ export GITHUB_TOKEN=<token>
```

Apache MXNet (Incubating) distributed GPU training

This tutorial will guide you on distributed training with Apache MXNet (Incubating) on your multi-node GPU cluster. It uses Parameter Server. To run MXNet distributed training on EKS, we will use the Kubernetes MXNet-operator called `MXJob`. It provides a custom resource that makes it easy to run distributed or non-distributed MXNet jobs (training and tuning) on Kubernetes.

1. Setup a namespace.

```
$ NAMESPACE=kubeflow-dist-train-mx; kubectl --kubeconfig=/home/ubuntu/.kube/eksctl/
clusters/training-gpu-1 create namespace ${NAMESPACE}
```

2. Set the app name and initialize it.

```
$ APP_NAME=kubeflow-mx-ps; ks init ${APP_NAME}; cd ${APP_NAME}
```

3. Change the namespace used by the default environment to `${NAMESPACE}`.

```
$ ks env set default --namespace ${NAMESPACE}
```

4. Install MXNet-operator for kubeflow. This is needed to run MXNet distributed training with parameter server.

```
$ ks registry add kubeflow github.com/kubeflow/kubeflow/tree/${KUBEFLOW_VERSION}/
kubeflow
$ ks pkg install kubeflow/mxnet-job@v${KUBEFLOW_VERSION}
```

5. Generate Kubernetes-compatible, jsonnet component manifest file.

```
$ ks generate mxnet-operator mxnet-operator
```

6. Apply the configuration settings.

```
$ ks apply default -c mxnet-operator
```

7. Using a Custom Resource Definition (CRD) gives users the ability to create and manage MX Jobs just like builtin K8s resources. Verify that the MXNet custom resource is installed.

```
$ kubectl get crd
```

The output should include `mxjobs.kubeflow.org`.

Running MNIST distributed training with parameter server example

Your first task is to create a pod file(mx_job_dist.yaml) for your job according to the available cluster configuration and job to run. There are 3 jobModes you need to specify: Scheduler, Server and Worker. You can specify how many pods you want to spawn with the field replicas. The instance type of the Scheduler, Server, and Worker will be of the type specified at cluster creation.

- Scheduler: There is only one scheduler. The role of the scheduler is to set up the cluster. This includes waiting for messages that each node has come up and which port the node is listening on. The scheduler then lets all processes know about every other node in the cluster, so that they can communicate with each other.
- Server: There can be multiple servers which store the model's parameters, and communicate with workers. A server may or may not be co-located with the worker processes.
- Worker: A worker node actually performs training on a batch of training samples. Before processing each batch, the workers pull weights from servers. The workers also send gradients to the servers after each batch. Depending on the workload for training a model, it might not be a good idea to run multiple worker processes on the same machine.
- Provide container image you want to use with the field image.
- You can provide `restartPolicy` from one of the Always, OnFailure and Never. It determines whether pods will be restarted when they exit or not.
- Provide container image you want to use with the field image.

1. Based on the previous discussion, you would modify the following code block according to your requirements and save it in a file called `mx_job_dist.yaml`.

```
apiVersion: "kubeflow.org/v1alpha1"
kind: "MXJob"
metadata:
  name: "gpu-dist-job"
spec:
  jobMode: "dist"
  replicaSpecs:
    - replicas: 1
      mxReplicaType: SCHEDULER
      PsRootPort: 9000
      template:
        spec:
          containers:
            - image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/aws-samples-mxnet-
training:1.4.1-gpu-py36-cu100-ubuntu16.04-example
              name: mxnet
          restartPolicy: OnFailure
    - replicas: 2
      mxReplicaType: SERVER
      template:
        spec:
          containers:
            - image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/aws-samples-mxnet-
training:1.4.1-gpu-py36-cu100-ubuntu16.04-example
              name: mxnet
          restartPolicy: OnFailure
    - replicas: 2
      mxReplicaType: WORKER
      template:
        spec:
          containers:
            - image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/aws-samples-mxnet-
training:1.4.1-gpu-py36-cu100-ubuntu16.04-example
              name: mxnet
              command: ["python"]
              args: ["/incubator-mxnet/example/image-classification/train_mnist.py","--
num-epochs","1","--num-layers","2","--kv-store","dist_device_sync","--gpus","0,1"]
              resources:
                limits:
                  nvidia.com/gpu: 2
          restartPolicy: OnFailure
```

2. Run distributed training job with the pod file you just created.

```
$ # Create a job by defining MXJob
kubectl create -f mx_job_dist.yaml
```

3. List the running jobs.

```
$ kubectl get mxjobs
```

4. To get status of a running job, run the following. Replace the JOB variable with whatever the job's name is.

```
$ JOB=gpu-dist-job
kubectl get -o yaml mxjobs $JOB
```

The output should be similar to the following:

```
apiVersion: kubeflow.org/v1alpha1
kind: MXJob
```

```
metadata:
  creationTimestamp: 2019-03-21T22:00:38Z
  generation: 1
  name: gpu-dist-job
  namespace: default
  resourceVersion: "2523104"
  selfLink: /apis/kubeflow.org/v1alpha1/namespaces/default/mxjobs/gpu-dist-job
  uid: c2e67f05-4c24-11e9-a6d4-125f5bb10ada
spec:
  RuntimeId: j1ht
  jobMode: dist
  mxImage: jzp1025/mxnet:test
  replicaSpecs:
  - PsRootPort: 9000
    mxReplicaType: SCHEDULER
    replicas: 1
    template:
      metadata:
        creationTimestamp: null
      spec:
        containers:
        - image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/aws-samples-mxnet-
training:1.4.1-gpu-py36-cu100-ubuntu16.04-example
          name: mxnet
          resources: {}
        restartPolicy: OnFailure
  - PsRootPort: 9091
    mxReplicaType: SERVER
    replicas: 2
    template:
      metadata:
        creationTimestamp: null
      spec:
        containers:
        - image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/aws-samples-mxnet-
training:1.4.1-gpu-py36-cu100-ubuntu16.04-example
          name: mxnet
          resources: {}
  - PsRootPort: 9091
    mxReplicaType: WORKER
    replicas: 2
    template:
      metadata:
        creationTimestamp: null
      spec:
        containers:
        - args:
          - /incubator-mxnet/example/image-classification/train_mnist.py
          - --num-epochs
          - "15"
          - --num-layers
          - "2"
          - --kv-store
          - dist_device_sync
          - --gpus
          - "0"
          command:
          - python
          image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/aws-samples-mxnet-
training:1.4.1-gpu-py36-cu100-ubuntu16.04-example
          name: mxnet
          resources:
            limits:
              nvidia.com/gpu: 1
        restartPolicy: OnFailure
  terminationPolicy:
```

```
    chief:
      replicaIndex: 0
      replicaName: WORKER
status:
  phase: Running
  reason: ""
  replicaStatuses:
  - ReplicasStates:
      Running: 1
    mx_replica_type: SCHEDULER
    state: Running
  - ReplicasStates:
      Running: 2
    mx_replica_type: SERVER
    state: Running
  - ReplicasStates:
      Running: 2
    mx_replica_type: WORKER
    state: Running
  state: Running
```

> **Note**
> Status provides information about the state of the resources.
> Phase - Indicates the phase of a job and will be one of Creating, Running, CleanUp, Failed, Done.
> State - Provides the overall status of the job and will be one of Running, Succeeded, Failed.

5. To cleanup and rerun a job:

```
$ eksctl delete cluster --name=<cluster-name>
```

If you want to delete a job, change directories to where you launched the job and run the following:

```
$ ks delete default
$ kubectl delete -f mx_job_dist.yaml
```

Apache MXNet (Incubating) with Horovod distributed GPU training

This tutorial shows how to setup distributed training of Apache MXNet (Incubating) models on your multi-node GPU cluster that uses Horovod. It uses an example image that already has a training script included, and it uses a 3-node cluster with `node-type=p3.8xlarge`. This tutorial runs the Horovod example script for MXNet on an MNIST model.

1. Set the app name and initialize it.

```
$ APP_NAME=kubeflow-mxnet-hvd; ks init ${APP_NAME}; cd ${APP_NAME}
```

2. Install mpi-operator from kubeflow in this app's folder.

```
$ KUBEFLOW_VERSION=v0.5.1
$ ks registry add kubeflow github.com/kubeflow/kubeflow/tree/${KUBEFLOW_VERSION}/
kubeflow
$ ks pkg install kubeflow/common@${KUBEFLOW_VERSION}
$ ks pkg install kubeflow/mpi-job@${KUBEFLOW_VERSION}
$ ks generate mpi-operator mpi-operator
$ ks param set mpi-operator image mpioperator/mpi-operator:0.2.0
$ ks apply default -c mpi-operator
```

3. Create a MPI Job template and define the number of nodes (replicas) and number of GPUs each node has (gpusPerReplica). You can also use your image and customize command.

```
IMAGE="763104351884.dkr.ecr.us-east-1.amazonaws.com/aws-samples-mxnet-training:1.6.0-gpu-py36-cu101-ubuntu16.04-example"
GPUS_PER_WORKER=4
NUMBER_OF_WORKERS=3
JOB_NAME=mx-mnist-horovod-job
ks generate mpi-job-custom ${JOB_NAME}

ks param set ${JOB_NAME} replicas ${NUMBER_OF_WORKERS}
ks param set ${JOB_NAME} gpusPerReplica ${GPUS_PER_WORKER}
ks param set ${JOB_NAME} image ${IMAGE}


ks param set ${JOB_NAME} command "mpirun,-mca,btl_tcp_if_exclude,lo,-mca,pml,ob1,-mca,btl,^openib,--bind-to,none,-map-by,slot,-x,LD_LIBRARY_PATH,-x,PATH,-x,NCCL_SOCKET_IFNAME=eth0,-x,NCCL_DEBUG=INFO,python,/horovod/examples/mxnet_mnist.py"
ks param set ${JOB_NAME} args -- --epochs=10,--lr=0.001
```

4. Check the job manifest that was created to verify everything looks okay.

```
$ ks show default -c ${JOB_NAME}
```

5. Apply the manifest to the default environment. The MPI Job will create a launch pod and logs will be aggregated in this pod.

```
$ ks apply default -c ${JOB_NAME}
```

6. Check the status. The name of the job “mxnet-training” was in the mxnet.yaml file. It will now appear in the status. If you're running any other tests or have previously run something, it will appear in this list. Run this several times until you see the status change to “Running”.

```
$ kubectl get pods -o wide
```

You should see something similar to the following output:

```
NAME                                     READY     STATUS    RESTARTS   AGE       IP
            NODE                              NOMINATED NODE
mpi-operator-5fff9d76f5-wvf56            1/1       Running   0          23m
 192.168.10.117   ip-192-168-22-21.ec2.internal    <none>
mx-mnist-horovod-job-launcher-d7w6t      1/1       Running   0          21m
 192.168.13.210   ip-192-168-4-253.ec2.internal    <none>
mx-mnist-horovod-job-worker-0            1/1       Running   0          22m
 192.168.17.216   ip-192-168-4-253.ec2.internal    <none>
mx-mnist-horovod-job-worker-1            1/1       Running   0          22m
 192.168.20.228   ip-192-168-27-148.ec2.internal   <none>
mx-mnist-horovod-job-worker-2            1/1       Running   0          22m
 192.168.11.70    ip-192-168-22-21.ec2.internal    <none>
```

7. Based on the name of the launcher pod above, check the logs to see the training output.

```
$ kubectl logs -f --tail 10 mx-mnist-horovod-job-launcher-d7w6t
```

8. You can check the logs to watch the training progress. You can also continue to check “get pods” to refresh the status. When the status changes to “Completed” you will know that the training job is done.

9. To cleanup and rerun a job:

```
# make sure ${JOB_NAME} and ${APP_NAME} are still set
$ ks delete default -c ${JOB_NAME}
$ ks delete default
$ cd .. && rm -rf ${APP_NAME}
```

Next steps

To learn GPU-based inference on Amazon EKS using MXNet with Deep Learning Containers, see Apache MXNet (Incubating) GPU inference (p. 64).

TensorFlow with Horovod distributed GPU training

This tutorial shows how to setup distributed training of TensorFlow models on your multi-node GPU cluster that uses Horovod. It uses an example image that already has a training script included, and it uses a 3-node cluster with `node-type=p3.16xlarge`. You can use this tutorial with either TensorFlow or TensorFlow 2. To use it with TensorFlow 2, change the Docker image to a TensorFlow 2 image.

1. Set the app name and initialize it.

```
$ APP_NAME=kubeflow-tf-hvd; ks init ${APP_NAME}; cd ${APP_NAME}
```

2. Install mpi-operator from kubeflow in this app's folder.

```
$ KUBEFLOW_VERSION=v0.5.1
$ ks registry add kubeflow github.com/kubeflow/kubeflow/tree/${KUBEFLOW_VERSION}/
kubeflow
$ ks pkg install kubeflow/common@${KUBEFLOW_VERSION}
$ ks pkg install kubeflow/mpi-job@${KUBEFLOW_VERSION}
$ ks generate mpi-operator mpi-operator
$ ks param set mpi-operator image mpioperator/mpi-operator:0.2.0
$ ks apply default -c mpi-operator
```

3. Create a MPI Job template and define the number of nodes (replicas) and number of GPUs each node has (gpusPerReplica), you can also bring your image and customize command.

```
IMAGE="763104351884.dkr.ecr.us-east-1.amazonaws.com/aws-samples-tensorflow-
training:1.14.0-gpu-py36-cu100-ubuntu16.04-example"
GPUS_PER_WORKER=2
NUMBER_OF_WORKERS=3
JOB_NAME=tf-resnet50-horovod-job
ks generate mpi-job-custom ${JOB_NAME}

ks param set ${JOB_NAME} replicas ${NUMBER_OF_WORKERS}
ks param set ${JOB_NAME} gpusPerReplica ${GPUS_PER_WORKER}
ks param set ${JOB_NAME} image ${IMAGE}

ks param set ${JOB_NAME} command "mpirun,-mca,btl_tcp_if_exclude,lo,-mca,pml,ob1,-
mca,btl,^openib,--bind-to,none,-map-by,slot,-x,LD_LIBRARY_PATH,-x,PATH,-
x,NCCL_SOCKET_IFNAME=eth0,-x,NCCL_DEBUG=INFO,python,/deep-learning-models/models/
resnet/tensorflow/train_imagenet_resnet_hvd.py"
ks param set ${JOB_NAME} args -- --num_epochs=10,--synthetic
```

4. Check the job manifest that was created to verify everything looks okay.

```
$ ks show default -c ${JOB_NAME}
```

5. Now apply the manifest to the default environment. The MPI Job will create a launch pod and logs will be aggregated in this pod.

```
$ ks apply default -c ${JOB_NAME}
```

6. Check the status. The name of the job “tensorflow-training” was in the tf.yaml file. It will now appear in the status. If you're running any other tests or have previously run something, it will appear in this list. Run this several times until you see the status change to “Running”.

```
$ kubectl get pods -o wide
```

You should see something similar to the following output:

```
NAME                                        READY     STATUS    RESTARTS   AGE       IP
            NODE                                NOMINATED NODE
mpi-operator-5fff9d76f5-wvf56               1/1       Running   0          23m
 192.168.10.117   ip-192-168-22-21.ec2.internal    <none>
tf-resnet50-horovod-job-launcher-d7w6t      1/1       Running   0          21m
 192.168.13.210   ip-192-168-4-253.ec2.internal    <none>
tf-resnet50-horovod-job-worker-0            1/1       Running   0          22m
 192.168.17.216   ip-192-168-4-253.ec2.internal    <none>
tf-resnet50-horovod-job-worker-1            1/1       Running   0          22m
 192.168.20.228   ip-192-168-27-148.ec2.internal   <none>
tf-resnet50-horovod-job-worker-2            1/1       Running   0          22m
 192.168.11.70    ip-192-168-22-21.ec2.internal    <none>
```

7. Based on the name of the launcher pod above, check the logs to see the training output.

```
$ kubectl logs -f --tail 10 tf-resnet50-horovod-job-launcher-d7w6t
```

8. You can check the logs to watch the training progress. You can also continue to check “get pods” to refresh the status. When the status changes to “Completed” you will know that the training job is done.
9. To cleanup and rerun a job:

```
# make sure ${JOB_NAME} and ${APP_NAME} are still set
$ ks delete default -c ${JOB_NAME}
$ ks delete default
$ cd .. && rm -rf ${APP_NAME}
```

Next steps

To learn GPU-based inference on Amazon EKS using TensorFlow with Deep Learning Containers, see TensorFlow GPU inference (p. 66).

PyTorch distributed GPU training

This tutorial will guide you on distributed training with PyTorch on your multi-node GPU cluster. It uses Gloo as the backend.

1. Setup a namespace.

```
$ NAMESPACE=pytorch-multi-node-training; kubectl create namespace ${NAMESPACE}
```

2. Set the app name and initialize it.

```
$ APP_NAME=eks-pytorch-mnist-app; ks init ${APP_NAME}; cd ${APP_NAME}
```

3. Change the namespace used by the default environment to `${NAMESPACE}`.

```
$ ks env set default --namespace ${NAMESPACE}
```

4. Install pytorch-operator for kubeflow. This is needed to run PyTorch distributed training with parameter server.

```
$ export KUBEFLOW_VERSION=0.6.1
$ ks registry add kubeflow github.com/kubeflow/kubeflow/tree/v${KUBEFLOW_VERSION}/
kubeflow
$ ks pkg install kubeflow/pytorch-job@v${KUBEFLOW_VERSION}
```

5. Generate Kubernetes-compatible, jsonnet component manifest file.

```
$ ks generate pytorch-operator pytorch-operator
```

6. Apply the configuration settings.

```
$ ks apply default -c pytorch-operator
```

7. Using a Custom Resource Definition (CRD) gives users the ability to create and manage PyTorch Jobs just like built-in K8s resources. Verify that the PyTorch custom resource is installed.

```
$ kubectl get crd
```

8. Apply the nvidia plugin.

```
$ kubectl apply -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v1.12/
nvidia-device-plugin.yml
```

9. Use the following text to create a gloo-based distributed data parallel job. Save it in a file called `distributed.yaml`.

```
apiVersion: kubeflow.org/v1
kind: PyTorchJob
metadata:
  name: "kubeflow-pytorch-gpu-dist-job"
spec:
  pytorchReplicaSpecs:
    Master:
      replicas: 1
      restartPolicy: OnFailure
      template:
        spec:
          containers:
          - name: "pytorch"
            image: "763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-
training:1.3.1-gpu-py36-cu101-ubuntu16.04"
            args:
              - "--backend"
              - "gloo"
              - "--epochs"
              - "5"
    Worker:
      replicas: 2
      restartPolicy: OnFailure
      template:
        spec:
          containers:
          - name: "pytorch"
            image: "763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-
training:1.3.1-gpu-py36-cu101-ubuntu16.04 "
```

```
          args:
            - "--backend"
            - "gloo"
            - "--epochs"
            - "5"
          resources:
            limits:
              nvidia.com/gpu: 1
```

10. Run distributed training job with the pod file you just created.

```
$ # kubectl create -f distributed.yaml
```

11. You can check the status of the job using the following:

```
$ kubectl logs kubeflow-pytorch-gpu-dist-job
```

    To view logs continuously, use:

```
$kubectl logs -f <pod>
```

See EKS Cleanup for information on cleaning up a cluster after you're done using it.

Next steps

To learn GPU-based inference on Amazon EKS using PyTorch with Deep Learning Containers, see PyTorch GPU inference (p. 70).

Inference

The following topics show how to run inference on AWS Deep Learning Containers for EKS using Apache MXNet (Incubating), PyTorch, TensorFlow, and TensorFlow 2. Inference uses either CPU or GPU clusters, but only with single node configurations.

Contents

- CPU Inference (p. 56)
- GPU Inference (p. 64)

CPU Inference

This section guides you on running inference on Deep Learning Containers for EKS CPU clusters using Apache MXNet (Incubating), PyTorch, TensorFlow, and TensorFlow 2.

For a complete list of Deep Learning Containers, see Available Deep Learning Containers Images.

> **Note**
> If you're using MKL, see AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78) to get the best training or inference performance.

Contents

- Apache MXNet (Incubating) CPU inference (p. 57)
- TensorFlow CPU inference (p. 58)
- PyTorch CPU inference (p. 62)

Apache MXNet (Incubating) CPU inference

In this tutorial, you create a Kubernetes Service and a Deployment to run CPU inference with MXNet. The Kubernetes Service exposes a process and its ports. When you create a Kubernetes Service, you can specify the kind of Service you want using `ServiceTypes`. The default `ServiceType` is `ClusterIP`. The Deployment is responsible for ensuring that a certain number of pods is always up and running.

1. Create the namespace. You may need to change the kubeconfig to point to the right cluster. Verify that you have setup a "training-cpu-1" or change this to your CPU cluster's config. For more information on setting up your cluster, see Amazon EKS Setup (p. 32).

```
$ NAMESPACE=mx-inference; kubectl —kubeconfig=/home/ubuntu/.kube/eksctl/clusters/
training-cpu-1 create namespace ${NAMESPACE}
```

2. (Optional step when using public models.) Set up your model at a network location that is mountable, like in Amazon S3. For information on how to upload a trained model to S3, see TensorFlow CPU inference (p. 58). Apply the secret to your namespace. For more information on secrets, see the Kubernetes Secrets documentation.

```
$ kubectl -n ${NAMESPACE} apply -f secret.yaml
```

3. Create a file named `mx_inference.yaml`with the following content. This example file specifies the model, MXNet inference image used, and the location of the model. This example uses a public model, so you don't need to modify it.

```
---
kind: Service
apiVersion: v1
metadata:
  name: squeezenet-service
  labels:
    app: squeezenet-service
spec:
  ports:
  - port: 8080
    targetPort: mms
  selector:
    app: squeezenet-service
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: squeezenet-service
  labels:
    app: squeezenet-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: squeezenet-service
  template:
    metadata:
      labels:
        app: squeezenet-service
    spec:
      containers:
      - name: squeezenet-service
        image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/mxnet-inference:1.6.0-cpu-
py36-ubuntu16.04
        args:
        - mxnet-model-server
        - --start
```

```
        - --mms-config /home/model-server/config.properties
        - --models squeezenet=https://s3.amazonaws.com/model-server/model_archive_1.0/
squeezenet_v1.1.mar
        ports:
        - name: mms
          containerPort: 8080
        - name: mms-management
          containerPort: 8081
        imagePullPolicy: IfNotPresent
```

4. Apply the configuration to a new pod in the previously defined namespace.

```
$ kubectl -n ${NAMESPACE} apply -f mx_inference.yaml
```

   Your output should be similar to the following:

```
service/squeezenet-service created
deployment.apps/squeezenet-service created
```

5. Check the status of the pod.

```
$ kubectl get pods -n ${NAMESPACE}
```

   Repeat the status check until you see the following "RUNNING" state:

```
NAME                      READY     STATUS    RESTARTS    AGE
squeezenet-service-xvw1   1/1       Running   0           3m
```

6. To further describe the pod, run the following:

```
$ kubectl describe pod <pod_name> -n ${NAMESPACE}
```

7. Because the serviceType here is ClusterIP, you can forward the port from your container to your host machine using the following command:

```
$ kubectl port-forward -n ${NAMESPACE} `kubectl get pods -n ${NAMESPACE} --
selector=app=squeezenet-service -o jsonpath='{.items[0].metadata.name}'` 8080:8080 &
```

8. Download an image of a kitten.

```
$ curl -O https://s3.amazonaws.com/model-server/inputs/kitten.jpg
```

9. Run inference on the model using the image of the kitten:

```
$ curl -X POST http://127.0.0.1:8080/predictions/squeezenet -T kitten.jpg
```

TensorFlow CPU inference

In this tutorial, you create a Kubernetes Service and a Deployment to run CPU inference with TensorFlow. The Kubernetes Service exposes a process and its ports. When you create a Kubernetes Service, you can specify the kind of Service you want using `ServiceTypes`. The default `ServiceType` is `ClusterIP`. The Deployment is responsible for ensuring that a certain number of pods is always up and running.

1. Create the namespace. You may need to change the kubeconfig to point to the right cluster. Verify that you have setup a "training-cpu-1" or change this to your CPU cluster's config. For more information on setting up your cluster, see Amazon EKS Setup (p. 32).

```
$ NAMESPACE=tf-inference; kubectl —kubeconfig=/home/ubuntu/.kube/eksctl/clusters/
training-cpu-1 create namespace ${NAMESPACE}
```

2. Models served for inference can be retrieved in different ways, such as using shared volumes and Amazon S3. Because the Kubernetes Service requires access to Amazon S3 and Amazon ECR, you must store your AWS credentials as a Kubernetes secret. For the purpose of this example, use S3 to store and fetch trained models.

   Verify your AWS credentials. They must have S3 write access.

```
$ cat ~/.aws/credentials
```

3. The output will be similar to the following:

```
$ [default]
aws_access_key_id = YOURACCESSKEYID
aws_secret_access_key = YOURSECRETACCESSKEY
```

4. Encode the credentials using base64.

   Encode the access key first.

```
$ echo -n 'YOURACCESSKEYID' | base64
```

   Encode the secret access key next.

```
$ echo -n 'YOURSECRETACCESSKEY' | base64
```

   Your output should look similar to the following:

```
$ echo -n 'YOURACCESSKEYID' | base64
RkFLRUFXU0FDQ0VTU0tFWUlE
$ echo -n 'YOURSECRETACCESSKEY' | base64
RkFLRUFXU1NFQ1JFVEFDQ0VTU0tFWQ==
```

5. Create a file named `secret.yaml` with the following content in your home directory. This file is used to store the secret.

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-s3-secret
type: Opaque
data:
  AWS_ACCESS_KEY_ID: YOURACCESSKEYID
  AWS_SECRET_ACCESS_KEY: YOURSECRETACCESSKEY
```

6. Apply the secret to your namespace.

```
$ kubectl -n ${NAMESPACE} apply -f secret.yaml
```

7. Clone the tensorflow-serving repository.

```
$ git clone https://github.com/tensorflow/serving/
$ cd serving/tensorflow_serving/servables/tensorflow/testdata/
```

8. Sync the pretrained `saved_model_half_plus_two_cpu` model to your S3 bucket.

```
$ aws s3 sync saved_model_half_plus_two_cpu s3://<your_s3_bucket>/
saved_model_half_plus_two
```

9. Create a file named `tf_inference.yaml` with the following content. Update `--model_base_path` to use your S3 bucket. You can use this with either TensorFlow or TensorFlow 2. To use it with TensorFlow 2, change the Docker image to a TensorFlow 2 image.

```
---
  kind: Service
  apiVersion: v1
  metadata:
    name: half-plus-two
    labels:
      app: half-plus-two
  spec:
    ports:
    - name: http-tf-serving
      port: 8500
      targetPort: 8500
    - name: grpc-tf-serving
      port: 9000
      targetPort: 9000
    selector:
      app: half-plus-two
      role: master
    type: ClusterIP
  ---
  kind: Deployment
  apiVersion: apps/v1
  metadata:
    name: half-plus-two
    labels:
      app: half-plus-two
      role: master
  spec:
    replicas: 1
    selector:
      matchLabels:
        app: half-plus-two
        role: master
    template:
      metadata:
        labels:
          app: half-plus-two
          role: master
      spec:
        containers:
        - name: half-plus-two
          image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-
inference:1.15.0-cpu-py36-ubuntu18.04
          command:
          - /usr/bin/tensorflow_model_server
          args:
          - --port=9000
          - --rest_api_port=8500
          - --model_name=saved_model_half_plus_two
          - --model_base_path=s3://tensorflow-trained-models/saved_model_half_plus_two
          ports:
          - containerPort: 8500
          - containerPort: 9000
          imagePullPolicy: IfNotPresent
          env:
          - name: AWS_ACCESS_KEY_ID
```

```
              valueFrom:
                secretKeyRef:
                  key: AWS_ACCESS_KEY_ID
                  name: aws-s3-secret
            - name: AWS_SECRET_ACCESS_KEY
              valueFrom:
                secretKeyRef:
                  key: AWS_SECRET_ACCESS_KEY
                  name: aws-s3-secret
            - name: AWS_REGION
              value: us-east-1
            - name: S3_USE_HTTPS
              value: "true"
            - name: S3_VERIFY_SSL
              value: "true"
            - name: S3_ENDPOINT
              value: s3.us-east-1.amazonaws.com
```

10. Apply the configuration to a new pod in the previously defined namespace.

```
$ kubectl -n ${NAMESPACE} apply -f tf_inference.yaml
```

Your output should be similar to the following:

```
service/half-plus-two created
deployment.apps/half-plus-two created
```

11. Check the status of the pod.

```
$ kubectl get pods -n ${NAMESPACE}
```

Repeat the status check until you see the following "RUNNING" state:

```
NAME                      READY     STATUS     RESTARTS    AGE
half-plus-two-vmwp9   1/1       Running    0           3m
```

12. To further describe the pod, you can run:

```
$ kubectl describe pod <pod_name> -n ${NAMESPACE}
```

13. Because the serviceType is ClusterIP, you can forward the port from your container to your host machine.

```
$ kubectl port-forward -n ${NAMESPACE} `kubectl get pods -n ${NAMESPACE} --
selector=app=half-plus-two -o jsonpath='{.items[0].metadata.name}'` 8500:8500 &
```

14. Place the following json string in a file named `half_plus_two_input.json`

```
{"instances": [1.0, 2.0, 5.0]}
```

15. Run inference on the model.

```
$ curl -d @half_plus_two_input.json -X POST http://localhost:8500/v1/models/
saved_model_half_plus_two_cpu:predict
```

Your output should look like the following:

```
{
```

```
    "predictions": [2.5, 3.0, 4.5
    ]
}
```

PyTorch CPU inference

In this approach, you create a Kubernetes Service and a Deployment to run CPU inference with PyTorch. The Kubernetes Service exposes a process and its ports. When you create a Kubernetes Service, you can specify the kind of Service you want using `ServiceTypes`. The default `ServiceType` is `ClusterIP`. The Deployment is responsible for ensuring that a certain number of pods is always up and running.

1. Create the namespace. You may need to change the kubeconfig to point to the right cluster. Verify that you have setup a "training-cpu-1" or change this to your CPU cluster's config. For more information on setting up your cluster, see Amazon EKS Setup (p. 32).

```
$ NAMESPACE=pt-inference; kubectl create namespace ${NAMESPACE}
```

2. (Optional step when using public models.) Setup your model at a network location that is mountable, like in Amazon S3. For information on how to upload a trained model to S3, see TensorFlow CPU inference (p. 58). Apply the secret to your namespace. For more information on secrets, see the Kubernetes Secrets documentation.

```
$ kubectl -n ${NAMESPACE} apply -f secret.yaml
```

3. Create a file named `pt_inference.yaml` with the following content. This example file specifies the model, PyTorch inference image used, and the location of the model. This example uses a public model, so you don't need to modify it.

```
---
kind: Service
apiVersion: v1
metadata:
  name: densenet-service
  labels:
    app: densenet-service
spec:
  ports:
  - port: 8080
    targetPort: mms
  selector:
    app: densenet-service
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: densenet-service
  labels:
    app: densenet-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: densenet-service
  template:
    metadata:
      labels:
        app: densenet-service
    spec:
      containers:
      - name: densenet-service
```

```
        image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-inference:1.3.1-
cpu-py36-ubuntu16.04
        args:
        - mxnet-model-server
        - --start
        - --mms-config /home/model-server/config.properties
        - --models densenet=https://dlc-samples.s3.amazonaws.com/pytorch/multi-model-
server/densenet/densenet.mar
        ports:
        - name: mms
          containerPort: 8080
        - name: mms-management
          containerPort: 8081
        imagePullPolicy: IfNotPresent
```

4. Apply the configuration to a new pod in the previously defined namespace.

```
$ kubectl -n ${NAMESPACE} apply -f pt_inference.yaml
```

Your output should be similar to the following:

```
service/densenet-service created
deployment.apps/densenet-service created
```

5. Check the status of the pod and wait for the pod to be in “RUNNING” state:

```
$ kubectl get pods -n ${NAMESPACE} -w
```

Your output should be similar to the following:

```
NAME                     READY     STATUS     RESTARTS    AGE
densenet-service-xvw1    1/1       Running    0           3m
```

6. To further describe the pod, run the following:

```
$ kubectl describe pod <pod_name> -n ${NAMESPACE}
```

7. Because the serviceType here is ClusterIP, you can forward the port from your container to your host machine.

```
$ kubectl port-forward -n ${NAMESPACE} `kubectl get pods -n ${NAMESPACE} --
selector=app=densenet-service -o jsonpath='{.items[0].metadata.name}'` 8080:8080 &
```

8. With your server started, you can now run inference from a different window using the following:

```
$ curl -O https://s3.amazonaws.com/model-server/inputs/flower.jpg
curl -X POST http://127.0.0.1:8080/predictions/densenet -T flower.jpg
```

See EKS Cleanup for information on cleaning up a cluster after you're done using it.

Next steps

To learn about using Custom Entrypoints with Deep Learning Containers on Amazon EKS, see Custom Entrypoints (p. 72).

GPU Inference

This section shows how to run inference on Deep Learning Containers for EKS GPU clusters using Apache MXNet (Incubating), PyTorch, TensorFlow, and TensorFlow 2.

For a complete list of Deep Learning Containers, see Available Deep Learning Containers Images.

> **Note**
> MKL users: read the AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78) to get the best training or inference performance.

Contents

- Apache MXNet (Incubating) GPU inference (p. 64)
- TensorFlow GPU inference (p. 66)
- PyTorch GPU inference (p. 70)

Apache MXNet (Incubating) GPU inference

In this approach, you create a Kubernetes Service and a Deployment. The Kubernetes Service exposes a process and its ports. When you create a Kubernetes Service, you can specify the kind of Service you want using `ServiceTypes`. The default `ServiceType` is `ClusterIP`. The Deployment is responsible for ensuring that a certain number of pods is always up and running.

1. For GPU-base inference, install the NVIDIA device plugin for Kubernetes:

```
$ kubectl apply -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v1.12/
nvidia-device-plugin.yml
```

2. Verify that the nvidia-device-plugin-daemonset is running correctly.

```
$ kubectl get daemonset -n kube-system
```

   The output will be similar to the following:

```
NAME                               DESIRED   CURRENT   READY     UP-TO-DATE   AVAILABLE
 NODE SELECTOR    AGE
aws-node                           3         3         3         3            3
 <none>           6d
kube-proxy                         3         3         3         3            3
 <none>           6d
nvidia-device-plugin-daemonset     3         3         3         3            3
 <none>           57s
```

3. Create the namespace. You might need to change the kubeconfig to point to the right cluster. Verify that you have setup a "training-gpu-1" or change this to your GPU cluster's config. For more information on setting up your cluster, see Amazon EKS Setup (p. 32).

```
$ NAMESPACE=mx-inference; kubectl —kubeconfig=/home/ubuntu/.kube/eksctl/clusters/
training-gpu-1 create namespace ${NAMESPACE}
```

4. (Optional step when using public models.) Setup your model at a network location that is mountable e.g., in S3. Refer to the steps to upload a trained model to S3 mentioned in the section Inference with TensorFlow. Apply the secret to your namespace. For more information on secrets, see the Kubernetes Secrets documentation.

```
$ kubectl -n ${NAMESPACE} apply -f secret.yaml
```

5. Create the file `mx_inference.yaml`. Use the contents of the next code block as its content.

```
---
kind: Service
apiVersion: v1
metadata:
  name: squeezenet-service
  labels:
    app: squeezenet-service
spec:
  ports:
  - port: 8080
    targetPort: mms
  selector:
    app: squeezenet-service
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: squeezenet-service
  labels:
    app: squeezenet-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: squeezenet-service
  template:
    metadata:
      labels:
        app: squeezenet-service
    spec:
      containers:
      - name: squeezenet-service
        image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/mxnet-inference:1.6.0-gpu-
py36-cu101-ubuntu16.04
        args:
        - mxnet-model-server
        - --start
        - --mms-config /home/model-server/config.properties
        - --models squeezenet=https://s3.amazonaws.com/model-server/model_archive_1.0/
squeezenet_v1.1.mar
        ports:
        - name: mms
          containerPort: 8080
        - name: mms-management
          containerPort: 8081
        imagePullPolicy: IfNotPresent
        resources:
          limits:
            cpu: 4
            memory: 4Gi
            nvidia.com/gpu: 1
          requests:
            cpu: "1"
            memory: 1Gi
```

6. Apply the configuration to a new pod in the previously defined namespace:

```
$ kubectl -n ${NAMESPACE} apply -f mx_inference.yaml
```

Your output should be similar to the following:

```
service/squeezenet-service created
deployment.apps/squeezenet-service created
```

7. Check status of the pod and wait for the pod to be in “RUNNING” state:

```
$ kubectl get pods -n ${NAMESPACE}
```

8. Repeat the check status step until you see the following "RUNNING" state:

```
NAME                        READY     STATUS    RESTARTS    AGE
squeezenet-service-xvw1     1/1       Running   0           3m
```

9. To further describe the pod, you can run:

```
$ kubectl describe pod <pod_name> -n ${NAMESPACE}
```

10. Since the serviceType here is ClusterIP, you can forward the port from your container to your host machine (the ampersand runs this in the background):

```
$ kubectl port-forward -n ${NAMESPACE} `kubectl get pods -n ${NAMESPACE} --
selector=app=squeezenet-service -o jsonpath='{.items[0].metadata.name}'` 8080:8080 &
```

11. Download an image of a kitten:

```
$ curl -O https://s3.amazonaws.com/model-server/inputs/kitten.jpg
```

12. Run inference on the model:

```
$ curl -X POST http://127.0.0.1:8080/predictions/squeezenet -T kitten.jpg
```

TensorFlow GPU inference

In this approach, you create a Kubernetes Service and a Deployment. The Kubernetes Service exposes a process and its ports. When you create a Kubernetes Service, you can specify the kind of Service you want using `ServiceTypes`. The default `ServiceType` is `ClusterIP`. The Deployment is responsible for ensuring that a certain number of pods is always up and running.

1. For GPU-base inference, install the NVIDIA device plugin for Kubernetes:

```
$ kubectl apply -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v1.12/
nvidia-device-plugin.yml
```

2. Verify that the nvidia-device-plugin-daemonset is running correctly.

```
$ kubectl get daemonset -n kube-system
```

The output will be similar to the following:

```
NAME                                 DESIRED   CURRENT   READY     UP-TO-DATE   AVAILABLE
 NODE SELECTOR      AGE
aws-node                             3         3         3         3            3
 <none>             6d
kube-proxy                           3         3         3         3            3
 <none>             6d
```

```
nvidia-device-plugin-daemonset   3         3         3         3             3
 <none>          57s
```

3. Create the namespace. You might need to change the kubeconfig to point to the right cluster. Verify that you have setup a "training-gpu-1" or change this to your GPU cluster's config. For more information on setting up your cluster, see Amazon EKS Setup (p. 32).

```
$ NAMESPACE=tf-inference; kubectl —kubeconfig=/home/ubuntu/.kube/eksctl/clusters/
training-gpu-1 create namespace ${NAMESPACE}
```

4. Models served for inference can be retrieved in different ways e.g., using shared volumes, S3 etc. Since the service will require access to S3 and ECR, you must store your AWS credentials as a Kubernetes secret. For the purpose of this example, you will use S3 to store and fetch trained models.

   Check your AWS credentials. These must have S3 write access.

```
$ cat ~/.aws/credentials
```

5. The output will be something similar to the following:

```
$ [default]
aws_access_key_id = FAKEAWSACCESSKEYID
aws_secret_access_key = FAKEAWSSECRETACCESSKEY
```

6. Encode the credentials using base64. Encode the access key first.

```
$ echo -n 'FAKEAWSACCESSKEYID' | base64
```

   Encode the secret access key next.

```
$ echo -n 'FAKEAWSSECRETACCESSKEYID' | base64
```

   Your output should look similar to the following:

```
$ echo -n 'FAKEAWSACCESSKEYID' | base64
RkFLRUFXU0FDQ0VTU0tFWUlE
$ echo -n 'FAKEAWSSECRETACCESSKEY' | base64
RkFLRUFXU1NFQ1JFVEFDQ0VTU0tFWQ==
```

7. Create a yaml file to store the secret. Save it as secret.yaml in your home directory.

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-s3-secret
type: Opaque
data:
  AWS_ACCESS_KEY_ID: RkFLRUFXU0FDQ0VTU0tFWUlE
  AWS_SECRET_ACCESS_KEY: RkFLRUFXU1NFQ1JFVEFDQ0VTU0tFWQ==
```

8. Apply the secret to your namespace:

```
$ kubectl -n ${NAMESPACE} apply -f secret.yaml
```

9. In this example, you will clone the tensorflow-serving repository and sync a pretrained model to an S3 bucket. The following sample names the bucket `tensorflow-serving-models`. It also syncs a saved model to an S3 bucket called `saved_model_half_plus_two_gpu`.

```
$ git clone https://github.com/tensorflow/serving/
$ cd serving/tensorflow_serving/servables/tensorflow/testdata/
```

10. Sync the CPU model.

```
$ aws s3 sync saved_model_half_plus_two_gpu s3://<your_s3_bucket>/
saved_model_half_plus_two_gpu
```

11. Create the file `tf_inference.yaml`. Use the contents of the next code block as its content, and update `--model_base_path` to use your S3 bucket. You can use this with either TensorFlow or TensorFlow 2. To use it with TensorFlow 2, change the Docker image to a TensorFlow 2 image.

```
---
  kind: Service
  apiVersion: v1
  metadata:
    name: half-plus-two
    labels:
      app: half-plus-two
  spec:
    ports:
    - name: http-tf-serving
      port: 8500
      targetPort: 8500
    - name: grpc-tf-serving
      port: 9000
      targetPort: 9000
    selector:
      app: half-plus-two
      role: master
    type: ClusterIP
---
  kind: Deployment
  apiVersion: apps/v1
  metadata:
    name: half-plus-two
    labels:
      app: half-plus-two
      role: master
  spec:
    replicas: 1
    selector:
      matchLabels:
        app: half-plus-two
        role: master
    template:
      metadata:
        labels:
          app: half-plus-two
          role: master
      spec:
        containers:
        - name: half-plus-two
          image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-
inference:1.15.0-gpu-py36-cu100-ubuntu18.04
          command:
          - /usr/bin/tensorflow_model_server
          args:
          - --port=9000
          - --rest_api_port=8500
          - --model_name=saved_model_half_plus_two_gpu
```

```
          - --model_base_path=s3://tensorflow-trained-models/
saved_model_half_plus_two_gpu
          ports:
          - containerPort: 8500
          - containerPort: 9000
          imagePullPolicy: IfNotPresent
          env:
          - name: AWS_ACCESS_KEY_ID
            valueFrom:
              secretKeyRef:
                key: AWS_ACCESS_KEY_ID
                name: aws-s3-secret
          - name: AWS_SECRET_ACCESS_KEY
            valueFrom:
              secretKeyRef:
                key: AWS_SECRET_ACCESS_KEY
                name: aws-s3-secret
          - name: AWS_REGION
            value: us-east-1
          - name: S3_USE_HTTPS
            value: "true"
          - name: S3_VERIFY_SSL
            value: "true"
          - name: S3_ENDPOINT
            value: s3.us-east-1.amazonaws.com
            resources:
              limits:
                cpu: 4
                memory: 4Gi
                nvidia.com/gpu: 1
              requests:
                cpu: "1"
                memory: 1Gi
```

12. Apply the configuration to a new pod in the previously defined namespace:

```
$ kubectl -n ${NAMESPACE} apply -f tf_inference.yaml
```

Your output should be similar to the following:

```
service/half-plus-two created
deployment.apps/half-plus-two created
```

13. Check status of the pod and wait for the pod to be in “RUNNING” state:

```
$ kubectl get pods -n ${NAMESPACE}
```

14. Repeat the check status step until you see the following "RUNNING" state:

```
NAME                     READY     STATUS    RESTARTS   AGE
half-plus-two-vmwp9   1/1       Running    0          3m
```

15. To further describe the pod, you can run:

```
$ kubectl describe pod <pod_name> -n ${NAMESPACE}
```

16. Since the serviceType here is ClusterIP, you can forward the port from your container to your host machine (the ampersand runs this in the background):

```
$ kubectl port-forward -n ${NAMESPACE} `kubectl get pods -n ${NAMESPACE} --
selector=app=half-plus-two -o jsonpath='{.items[0].metadata.name}'` 8500:8500 &
```

17. Place the following json string in a file called `half_plus_two_input.json`

```
{"instances": [1.0, 2.0, 5.0]}
```

18. Run inference on the model:

```
$ curl -d @half_plus_two_input.json -X POST http://localhost:8500/v1/models/
saved_model_half_plus_two_cpu:predict
```

The expected output is as follows:

```
{
    "predictions": [2.5, 3.0, 4.5
    ]
}
```

PyTorch GPU inference

In this approach, you create a Kubernetes Service and a Deployment. The Kubernetes Service exposes a process and its ports. When you create a Kubernetes Service, you can specify the kind of Service you want using `ServiceTypes`. The default `ServiceType` is `ClusterIP`. The Deployment is responsible for ensuring that a certain number of pods is always up and running.

1. For GPU-base inference, install the NVIDIA device plugin for Kubernetes.

```
$ kubectl apply -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v1.12/
nvidia-device-plugin.yml
```

2. Verify that the nvidia-device-plugin-daemonset is running correctly.

```
$ kubectl get daemonset -n kube-system
```

The output will be similar to the following.

```
NAME                              DESIRED   CURRENT   READY     UP-TO-DATE   AVAILABLE
 NODE SELECTOR   AGE
aws-node                          3         3         3         3            3
 <none>          6d
kube-proxy                        3         3         3         3            3
 <none>          6d
nvidia-device-plugin-daemonset    3         3         3         3            3
 <none>          57s
```

3. Create the namespace.

```
$ NAMESPACE=pt-inference; kubectl create namespace ${NAMESPACE}
```

4. (Optional step when using public models.) Setup your model at a network location that is mountable e.g., in S3. Refer to the steps to upload a trained model to S3 mentioned in the section Inference with TensorFlow. Apply the secret to your namespace. For more information on secrets, see the Kubernetes Secrets documentation.

```
$ kubectl -n ${NAMESPACE} apply -f secret.yaml
```

5. Create the file `pt_inference.yaml`. Use the contents of the next code block as its content.

```
---
kind: Service
apiVersion: v1
metadata:
  name: densenet-service
  labels:
    app: densenet-service
spec:
  ports:
  - port: 8080
    targetPort: mms
  selector:
    app: densenet-service
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: densenet-service
  labels:
    app: densenet-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: densenet-service
  template:
    metadata:
      labels:
        app: densenet-service
    spec:
      containers:
      - name: densenet-service
        image: "763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-inference:1.3.1-gpu-py36-cu101-ubuntu16.04"
        args:
        - mxnet-model-server
        - --start
        - --mms-config /home/model-server/config.properties
        - --models densenet=https://dlc-samples.s3.amazonaws.com/pytorch/multi-model-server/densenet/densenet.mar
        ports:
        - name: mms
          containerPort: 8080
        - name: mms-management
          containerPort: 8081
        imagePullPolicy: IfNotPresent
        resources:
          limits:
            cpu: 4
            memory: 4Gi
            nvidia.com/gpu: 1
          requests:
            cpu: "1"
            memory: 1Gi
```

6. Apply the configuration to a new pod in the previously defined namespace.

```
$ kubectl -n ${NAMESPACE} apply -f pt_inference.yaml
```

Your output should be similar to the following:

```
service/densenet-service created
deployment.apps/densenet-service created
```

7. Check status of the pod and wait for the pod to be in “RUNNING” state.

```
$ kubectl get pods -n ${NAMESPACE}
```

Your output should be similar to the following:

```
NAME                     READY     STATUS    RESTARTS   AGE
densenet-service-xvw1    1/1       Running   0          3m
```

8. To further describe the pod, you can run:

```
$ kubectl describe pod <pod_name> -n ${NAMESPACE}
```

9. Since the serviceType here is ClusterIP, you can forward the port from your container to your host machine (the ampersand runs this in the background).

```
$ kubectl port-forward -n ${NAMESPACE} `kubectl get pods -n ${NAMESPACE} --
selector=app=densenet-service -o jsonpath='{.items[0].metadata.name}'` 8080:8080 &
```

10. With your server started, you can now run inference from a different window.

```
$ curl -O https://s3.amazonaws.com/model-server/inputs/flower.jpg
curl -X POST http://127.0.0.1:8080/predictions/densenet -T flower.jpg
```

See EKS Cleanup for information on cleaning up a cluster after you're done using it.

Next steps

To learn about using Custom Entrypoints with Deep Learning Containers on Amazon EKS, see Custom Entrypoints (p. 72).

Custom Entrypoints

For some images, AWS containers use a custom entrypoint script. If you want to use your own entrypoint, you can override the entrypoint as follows.

Update the `command` parameter in your pod file. Replace the `args` parameters with your custom entrypoint script.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: pytorch-multi-model-server-densenet
spec:
  restartPolicy: OnFailure
  containers:
  - name: pytorch-multi-model-server-densenet
    image: 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-inference:1.2.0-cpu-py36-
ubuntu16.04
    command:
```

```
      - "/bin/sh"
      - "-c"
    args:
      - /usr/local/bin/mxnet-model-server
      - --start
      - --mms-config /home/model-server/config.properties
      - --models densenet="https://dlc-samples.s3.amazonaws.com/pytorch/multi-model-server/
densenet/densenet.mar"
```

`command` is the Kubernetes field name for `entrypoint`. Refer to this table of Kubernetes field names for more information.

If the EKS cluster has expired IAM permissions to access the ECR repository holding the image, or you are using kubectl from a different user than the one that created the cluster, you will receive the following error.

```
error: unable to recognize "job.yaml": Unauthorized
```

To address this issue, you need to refresh the IAM tokens. Run the following script.

```
set -ex

AWS_ACCOUNT=${AWS_ACCOUNT}
AWS_REGION=us-east-1
DOCKER_REGISTRY_SERVER=https://${AWS_ACCOUNT}.dkr.ecr.${AWS_REGION}.amazonaws.com
DOCKER_USER=AWS
DOCKER_PASSWORD=`aws ecr get-login --region ${AWS_REGION} --registry-ids ${AWS_ACCOUNT} |
 cut -d' ' -f6`
kubectl delete secret aws-registry || true
kubectl create secret docker-registry aws-registry \
--docker-server=$DOCKER_REGISTRY_SERVER \
--docker-username=$DOCKER_USER \
--docker-password=$DOCKER_PASSWORD
kubectl patch serviceaccount default -p '{"imagePullSecrets":[{"name":"aws-registry"}]}'
```

Append the following under `spec` in your pod file.

```
imagePullSecrets:
    - name: aws-registry
```

Troubleshooting AWS Deep Learning Containers on EKS

Troubleshooting

Topics

- Setup Errors (p. 73)
- Usage Errors (p. 74)
- Cleanup Errors (p. 75)

Setup Errors

Error: ERROR registry `kubeflow` does not exist

Example:

```
$ ks pkg install kubeflow/tf-serving
ERROR registry 'kubeflow' does not exist
```

Solution: Run the following command.

```
ks registry add kubeflow github.com/google/kubefl ow/tree/master/kubeflow
```

Error: context deadline exceeded

Example:

```
$ eksctl create cluster <args>
[#] waiting for CloudFormation stack "eksctl-training-cluster-1-nodegroup-ng-8c4c94bc" to
 reach "CREATE_COMPLETE" status: RequestCanceled: waiter context canceled
caused by: context deadline exceeded
```

Solution: Verify that you have not exceeded capacity for your account. Retry the command in a different region.

```
$ kubectl get nodes
The connection to the server localhost:8080 was refused - did you specify the right host or
 port?
```

Solution: Try running `cp ~/.kube/eksctl/clusters/<name-of-cluster> ~/.kube/config`

```
$ ks apply default
ERROR handle object: patching object from cluster: merging object with existing state:
 Unauthorized
```

Solution: This is a concurrency issue that can occur when multiple users with different authorization/ credentials try to start jobs on the same cluster.

```
$ APP_NAME=kubeflow-tf-hvd; ks init ${APP_NAME}; cd ${APP_NAME}
INFO Using context "arn:aws:eks:eu-west-1:999999999999:cluster/training-gpu-1" from
 kubeconfig file "/home/ubuntu/.kube/config"
ERROR Could not create app; directory '/home/ubuntu/kubeflow-tf-hvd' already exists
```

Solution: ignore the warning. However you may have additional cleanup to do inside that folder. You may want to delete the folder to simplify cleanup.

Usage Errors

```
ssh: Could not resolve hostname openmpi-worker-1.openmpi.kubeflow-dist-train-tf: Name or
 service not known
```

Solution: At any point while using the EKS cluster, if you see this error message, run the NVIDIA device plugin for Kubernetes installation step again. Make sure that you have targeted the right cluster by either passing in the specific config file or by switching your active cluster to the targeted cluster.

Cleanup Errors

```
$ kubectl delete namespace ${NAMESPACE}
error: the server doesn't have a resource type "namspace"
```

Solution: check the spelling of your namespace. It might be a typo.

```
$ ks delete default
ERROR the server has asked for the client to provide credentials
```

Solution: Make sure that `~/.kube/config` points to the correct cluster and that AWS credentials have been correctly configured using `aws configure` or by exporting AWS environment variables

```
$ ks delete default
ERROR finding app root from starting path: : unable to find ksonnet project
$ kubectl logs -n ${NAMESPACE} -f ${COMPONENT}-master > results/benchmark_1.out
Error from server (NotFound): pods "openmpi-master" not found
```

Solution: Make sure you are cd'ed correctly into the ksonnet app created (the folder where `ks init` was run), and note that deleting the default context will cause the corresponding resources to be deleted.

```
$ ks component rm openmpi
ERROR finding app root from starting path: : unable to find ksonnet project
```

Solution: Make sure you are cd'ed correctly into the ksonnet app created (the folder where `ks init` was run).

Deep Learning Containers Images

AWS Deep Learning Containers are available as Docker images in Amazon ECR. Each Docker image is built for training or inference on a specific Deep Learning framework version, python version, with CPU or GPU support.

For the full list of available Deep Learning Containers and information on pulling them, see Available Deep Learning Containers Images.

Once you've selected your desired Deep Learning Containers image, continue with the one of the following:

- To run training and inference on Deep Learning Containers for Amazon EC2 using MXNet, PyTorch, TensorFlow, and TensorFlow 2, see Amazon EC2 Tutorials (p. 2)
- To run training and inference on Deep Learning Containers for Amazon ECS using MXNet, PyTorch, and TensorFlow, see Amazon ECS tutorials (p. 11)
- Deep Learning Containers for Amazon EKS offer CPU, GPU, and distributed GPU-based training, as well as CPU and GPU-based inference. To run training and inference on Deep Learning Containers for Amazon EKS using MXNet, PyTorch, and TensorFlow, see Amazon EKS Tutorials (p. 32)
- For information on security in Deep Learning Containers, see Security in AWS Deep Learning Containers (p. 83)
- For a list of the latest Deep Learning Containers release notes, see Release Notes for Deep Learning Containers (p. 90)

Deep Learning Containers Resources

The following topics describe additional AWS Deep Learning Containers resources.

Contents

- Building AWS Deep Learning Containers Custom Images (p. 77)
- AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations (p. 78)

Building AWS Deep Learning Containers Custom Images

How to Build Custom Images

We can easily customize both training and inference with Deep Learning Containers to add custom frameworks, libraries, and packages using Docker files.

Training with TensorFlow

In the following example Dockerfile, the resulting Docker image will have TensorFlow v1.15.2 optimized for GPUs and built to support Horovod and Python 3 for multi-node distributed training. It will also have the AWS samples GitHub repo which contains many deep learning model examples.

```
#Take base container
FROM 763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-training:1.15.2-gpu-py36-
cu100-ubuntu18.04

# Add custom stack of code
RUN git clone https://github.com/aws-samples/deep-learning-models
```

Training with Apache MXNet (Incubating)

In the following example Dockerfile, the resulting Docker image will have Apache MXNet (Incubating) v1.6.0 optimized for GPU inference built to support Horovod and Python 3. It will also have the MXNet GitHub repo which contains many deep learning model examples.

```
# Take the base MXNet Container
FROM 763104351884.dkr.ecr.us-east-1.amazonaws.com/mxnet-training:1.6.0-gpu-py36-cu101-
ubuntu16.04

# Add Custom stack of code
RUN git clone -b 1.6.0 https://github.com/apache/incubator-mxnet.git

ENTRYPOINT ["python", "/incubator-mxnet/example/image-classification/train_mnist.py"]
```

Build the Docker image, pointing to your personal Docker registry (usually your username), with the image's custom name and custom tag.

```
docker build -f Dockerfile -t <registry>/<any name>:<any tag>
```

Push to your personal Docker Registry:

```
docker push <registry>/<any name>:<any tag>
```

You can use the following command to run the container:

```
docker run -it < name or tag>
```

> **Important**
> You may need to login to access to the Deep Learning Containers image repository. Specify your region in the following command:
>
> ```
> aws ecr get-login-password --region us-east-1 | docker login --username AWS --
> password-stdin 763104351884.dkr.ecr.us-east-1.amazonaws.com
> ```

AWS Deep Learning Containers Intel Math Kernel Library (MKL) Recommendations

MKL Recommendation for CPU containers

Contents

- EC2 guide to set environment variables (p. 79)
- ECS guide to set environment variables (p. 80)
- EKS guide to set environment variables (p. 81)

The performance for training and inference workloads for a Deep Learning framework on CPU instances can vary and depend on a variety of configuration settings. As an example, on AWS EC2 c5.18xlarge instances the number of physical cores is 36 while the number of logical cores is 72. MKL's configuration settings for training and inference are influenced by these factors. By updating MKL's configuration to match your instance's capabilities, you may achieve performance improvements.

Consider the following examples using an Intel-MKL-optimized TensorFlow binary:

- A ResNet50v2 model, trained with TensorFlow and served for inference with TensorFlow Serving was observed to achieve 2x inference performance when the MKL settings were adjusted to match the instance's number cores. The following settings were used on a c5.18xlarge instance.

```
export TENSORFLOW_INTER_OP_PARALLELISM=2
# For an EC2 c5.18xlarge instance, number of logical cores = 72
export TENSORFLOW_INTRA_OP_PARALLELISM=72
# For an EC2 c5.18xlarge instance, number of physical cores = 36
export OMP_NUM_THREADS=36
export KMP_AFFINITY='granularity=fine,verbose,compact,1,0'
# For an EC2 c5.18xlarge instance, number of physical cores / 4 = 36 /4 = 9
```

```
export TENSORFLOW_SESSION_PARALLELISM=9
export KMP_BLOCKTIME=1
export KMP_SETTINGS=0
```

- A ResNet50_v1.5 model, trained with TensorFlow on the ImageNet dataset and using a NHWC image shape, the training throughput performance was observed to be around 9x faster. This is compared to the binary without MKL optimizations and measured in terms of samples/second. The following environment variables were used:

```
export TENSORFLOW_INTER_OP_PARALLELISM=0
# For an EC2 c5.18xlarge instance, number of logical cores = 72
export TENSORFLOW_INTRA_OP_PARALLELISM=72
# For an EC2 c5.18xlarge instance, number of physical cores = 36
export OMP_NUM_THREADS=36
export KMP_AFFINITY='granularity=fine,verbose,compact,1,0'
# For an EC2 c5.18xlarge instance, number of physical cores / 4 = 36 /4 = 9
export KMP_BLOCKTIME=1
export KMP_SETTINGS=0
```

The following links will help you learn how to use to tune Intel MKL and your Deep Learning framework's settings to optimize your deep learning workload:

- General best practices for Intel-optimized TensorFlow Serving
- TensorFlow performance
- Some Tips for improving Apache MXNet performance
- MXNet with Intel MKL-DNN - Performance Benchmarking

EC2 guide to set environment variables

Refer to `docker run` documentation on how to set environment variables when creating a container: https://docs.docker.com/engine/reference/run/#env-environment-variables

The following is an example on setting en environment variable called `OMP_NUM_THREADS` for docker run.

```
ubuntu@ip-172-31-95-248:~$ docker run -e OMP_NUM_THREADS=36 -it --entrypoint ""
 999999999999.dkr.ecr.us-east-1.amazonaws.com/beta-tensorflow-inference:1.13-py2-cpu-build
 bash
root@d437faf9b684:/# echo $OMP_NUM_THREADS
36
```

In rare cases Intel MKL can have adverse effects. To disable MKL with TensorFlow, set the following environment variables:

```
export TF_DISABLE_MKL=1
export TF_DISABLE_POOL_ALLOCATOR=1
```

ECS guide to set environment variables

To specify the environment variables for a container at runtime in ECS, you must edit the **ECS task definition**. Add the environment variables in the form of 'name' and 'value' key-pairs in `containerDefinitions` part of the task definition. The following is an example of setting `OMP_NUM_THREADS` and `KMP_BLOCKTIME` variables.

```
{
    "requiresCompatibilities": [
        "EC2"
    ],
    "containerDefinitions": [{
        "command": [
            "mkdir -p /test && cd /test && git clone -b r1.13 https://github.com/
tensorflow/serving.git && tensorflow_model_server --port=8500 --rest_api_port=8501
 --model_name=saved_model_half_plus_two_cpu --model_base_path=/test/serving/
tensorflow_serving/servables/tensorflow/testdata/saved_model_half_plus_two_cpu"
        ],
        "entryPoint": [
            "sh",
            "-c"
        ],
        "name": "EC2TFInference",
        "image": "999999999999.dkr.ecr.us-east-1.amazonaws.com/tf-inference:1.12-cpu-py3-
ubuntu16.04",
        "memory": 8111,
        "cpu": 256,
        "essential": true,
        "environment": [{
              "name": "OMP_NUM_THREADS",
              "value": "36"
            },
            {
              "name": "KMP_BLOCKTIME",
              "value": 1
            }
        ],
        "portMappings": [{
                "hostPort": 8500,
                "protocol": "tcp",
                "containerPort": 8500
            },
            {
                "hostPort": 8501,
                "protocol": "tcp",
                "containerPort": 8501
            },
            {
                "containerPort": 80,
                "protocol": "tcp"
            }
        ],
        "logConfiguration": {
            "logDriver": "awslogs",
            "options": {
                "awslogs-group": "/ecs/TFInference",
                "awslogs-region": "us-west-2",
                "awslogs-stream-prefix": "ecs",
                "awslogs-create-group": "true"
            }
        }
    }],
    "volumes": [],
```

```
    "networkMode": "bridge",
    "placementConstraints": [],
    "family": "Ec2TFInference"
}
```

In rare cases Intel MKL can have adverse effects. To disable MKL with TensorFlow, set the following environment variables:

```
            {
              "name": "TF_DISABLE_MKL",
              "value": 1
            },
            {
              "name": "TF_DISABLE_POOL_ALLOCATOR",
              "value": 1
            }
```

EKS guide to set environment variables

To specify the environment variables for the container at runtime, edit the raw manifests of your EKS job (.yaml, .json) . The following snippet of a manifest shows the definition of a container, with name `squeezenet-service`. Along with other attributes such as args and ports, the environment variables are listed in the form of 'name' and 'value' key-pairs.

```
      containers:
      - name: squeezenet-service
        image: 999999999999.dkr.ecr.us-east-1.amazonaws.com/beta-mxnet-inference:1.4.0-py3-
gpu-build
        command:
        - mxnet-model-server
        args:
        - --start
        - --mms-config /home/model-server/config.properties
        - --models squeezenet=https://s3.amazonaws.com/model-server/models/squeezenet_v1.1/
squeezenet_v1.1.model
        ports:
        - name: mms
          containerPort: 8080
        - name: mms-management
          containerPort: 8081
        imagePullPolicy: IfNotPresent
        env:
        - name: AWS_REGION
          value: us-east-1
        - name: OMP_NUM_THREADS
          value: 36
        - name: TENSORFLOW_INTER_OP_PARALLELISM
          value: 0
        - name: KMP_AFFINITY
          value: 'granularity=fine,verbose,compact,1,0'
        - name: KMP_BLOCKTIME
          value: 1
```

In rare cases Intel MKL can have adverse effects. To disable MKL with TensorFlow, set the following environment variables:

```
- name: TF_DISABLE_MKL
          value: 1
- name: TF_DISABLE_POOL_ALLOCATOR
```

```
        value: 1
```

Security in AWS Deep Learning Containers

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Deep Learning Containers, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Deep Learning Containers. The following topics show you how to configure Deep Learning Containers to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Deep Learning Containers resources.

For more information, see Security in Amazon EC2, Security in Amazon ECS, Security in Amazon EKS, and Security in Amazon SageMaker.

Topics

- Data Protection in AWS Deep Learning Containers (p. 83)
- Identity and Access Management in AWS Deep Learning Containers (p. 84)
- Logging and Monitoring in AWS Deep Learning Containers (p. 88)
- Compliance Validation for AWS Deep Learning Containers (p. 88)
- Resilience in AWS Deep Learning Containers (p. 89)
- Infrastructure Security in AWS Deep Learning Containers (p. 89)

Data Protection in AWS Deep Learning Containers

AWS Deep Learning Containers conforms to the AWS shared responsibility model, which includes regulations and guidelines for data protection. AWS is responsible for protecting the global infrastructure that runs all the AWS services. AWS maintains control over data hosted on this infrastructure, including the security configuration controls for handling customer content and personal data. AWS customers and APN partners, acting either as data controllers or data processors, are responsible for any personal data that they put in the AWS Cloud.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual user accounts with AWS Identity and Access Management (IAM), so that each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing personal data that is stored in Amazon S3.

We strongly recommend that you never put sensitive identifying information, such as your customers' account numbers, into free-form fields such as a **Name** field. This includes when you work with Deep Learning Containers or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into Deep Learning Containers or other services might get picked up for inclusion in diagnostic logs. When you provide a URL to an external server, don't include credentials information in the URL to validate your request to that server.

For more information about data protection, see Data Protection in Amazon EC2, Data Protection in Amazon SageMaker, and the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

Identity and Access Management in AWS Deep Learning Containers

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Deep Learning Containers resources. IAM is an AWS service that you can use with no additional charge.

For more information on Identity and Access Management, see Identity and Access Management for Amazon EC2, Identity and Access Management for Amazon ECS, Identity and Access Management for Amazon EKS, and Identity and Access Management for Amazon SageMaker.

Topics

- Authenticating With Identities (p. 84)
- Managing Access Using Policies (p. 86)
- IAM with Amazon EMR (p. 87)

Authenticating With Identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see The IAM Console and Sign-in Page in the *IAM User Guide*.

You must be *authenticated* (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication, or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email or your IAM user name. You can access AWS programmatically using your root user or IAM user access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 Signing Process in the *AWS General Reference*.

Regardless of the authentication method that you use, you might also be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Using Multi-Factor Authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM Users and Groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing Access Keys for IAM Users in the *IAM User Guide*. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to Create an IAM User (Instead of a Role) in the *IAM User Guide*.

IAM Roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM Roles in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated Users and Roles in the *IAM User Guide*.

- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM Roles Differ from Resource-based Policies in the *IAM User Guide*.
- **AWS service access** – A service role is an IAM role that a service assumes to perform actions in your account on your behalf. When you set up some AWS service environments, you must define a role for the service to assume. This service role must include all the permissions that are required for the service to access the AWS resources that it needs. Service roles vary from service to service, but many allow you to choose your permissions as long as you meet the documented requirements for that service. Service roles provide access only within your account and cannot be used to grant access to services in other accounts. You can create, modify, and delete a service role from within IAM. For example, you can create a role that allows Amazon Redshift to access an Amazon S3 bucket on your behalf and then load data from that bucket into an Amazon Redshift cluster. For more information, see Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide*.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM Role to Grant Permissions to Applications Running on Amazon EC2 Instances in the *IAM User Guide*.

To learn whether to use IAM roles, see When to Create an IAM Role (Instead of a User) in the *IAM User Guide*.

Managing Access Using Policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when an entity (root user, IAM user, or IAM role) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON Policies in the *IAM User Guide*.

An IAM administrator can use policies to specify who has access to AWS resources, and what actions they can perform on those resources. Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-Based Policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, role, or group. These policies control what actions that identity can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM Policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing Between Managed Policies and Inline Policies in the *IAM User Guide*.

Resource-Based Policies

Resource-based policies are JSON policy documents that you attach to a resource such as an Amazon S3 bucket. Service administrators can use these policies to define what actions a specified principal (account member, user, or role) can perform on that resource and under what conditions. Resource-based policies are inline policies. There are no managed resource-based policies.

Access Control Lists (ACLs)

Access control lists (ACLs) are a type of policy that controls which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format. Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access Control List (ACL) Overview in the *Amazon Simple Storage Service Developer Guide*.

Other Policy Types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions Boundaries for IAM Entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see How SCPs Work in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session Policies in the *IAM User Guide*.

Multiple Policy Types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy Evaluation Logic in the *IAM User Guide*.

IAM with Amazon EMR

You can use AWS Identity and Access Management with Amazon EMR to define users, AWS resources, groups, roles, and policies. You can also control which AWS services these users and roles can access.

For more information on using IAM with Amazon EMR, see AWS Identity and Access Management for Amazon EMR.

Logging and Monitoring in AWS Deep Learning Containers

Your AWS Deep Learning Containers does not come with monitoring utilities. For information on monitoring, see GPU Monitoring and Optimization, Monitoring Amazon EC2, Monitoring Amazon ECS, Monitoring Amazon EKS, and Monitoring Amazon SageMaker.

Usage Tracking

The following Deep Learning Containers include code that allows AWS to collect the instance types, frameworks, framework versions, container types, and Python versions used for the containers. No information on the commands used within the containers is collected or retained. No other information about the containers is collected or retained.

- TensorFlow 1.15
- TensorFlow 2.0
- TensorFlow 2.1
- PyTorch 1.2
- PyTorch 1.3.1
- MXNet 1.6

To opt-out of usage tracking, use a custom entrypoint to disable the call for the following services:

- Amazon EC2 Custom Entrypoints
- Amazon ECS Custom Entrypoints
- Amazon EKS Custom Entrypoints

To opt out of usage tracking for TensorFlow (version >=1.15), PyTorch (version >= 1.5), and MXNet (version >= 1.6) containers, you also need to set the `OPT_OUT_TRACKING` environment variable.

```
OPT_OUT_TRACKING=true
```

Compliance Validation for AWS Deep Learning Containers

Third-party auditors assess the security and compliance of services as part of multiple AWS compliance programs. For information on the supported compliance programs, see Compliance Validation for Amazon EC2, Compliance Validation for Amazon ECS, Compliance Validation for Amazon EKS, and Compliance Validation for Amazon SageMaker.

For a list of AWS services in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using Deep Learning Containers is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Resilience in AWS Deep Learning Containers

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

For information on features to help support your data resiliency and backup needs, see Resilience in Amazon EC2, Resilience in Amazon EKS, and Resilience in Amazon SageMaker.

Infrastructure Security in AWS Deep Learning Containers

The infrastructure security of AWS Deep Learning Containers is backed by Amazon EC2, Amazon ECS, Amazon EKS, or SageMaker. For more information, see Infrastructure Security in Amazon EC2, Infrastructure Security in Amazon ECS, Infrastructure Security in Amazon EKS, and Infrastructure Security in Amazon SageMaker.

Release Notes for Deep Learning Containers

For current AWS Deep Learning Containers release notes, see:

TensorFlow

- AWS Deep Learning Containers with Elastic Inference for TensorFlow 2.0.0: August 31, 2020
- AWS Deep Learning Containers with Elastic Inference for TensorFlow 1.15.0: August 31, 2020
- AWS Deep Learning Containers for TensorFlow 2.3.0: August 7, 2020
- AWS Deep Learning Containers for TensorFlow 2.2.0 with CUDA 10.2: July 24, 2020
- AWS Deep Learning Containers for TensorFlow 2.2.0 with CUDA 10.1: July 20, 2020
- AWS Deep Learning Containers for Tensorflow 1.15.3 July 29, 2020
- AWS Deep Learning Containers for TensorFlow 2.1.1: June 25, 2020
- AWS Deep Learning Containers for TensorFlow 2.0.2: June 19, 2020
- AWS Deep Learning Containers for TensorFlow 2.2.0 with CUDA 10.1: May 20, 2020
- AWS Deep Learning Containers for Tensorflow 2.1.0: March 19, 2020
- AWS Deep Learning Containers for TensorFlow 2.0: February 26, 2020
- AWS Deep Learning Containers for TensorFlow 1.15 with python-3.7 support: May 6, 2020
- AWS Deep Learning Containers for Tensorflow 1.15.2 March 19, 2020

PyTorch

- AWS Deep Learning Containers with Elastic Inference for PyTorch 1.3.1: August 31, 2020
- AWS Deep Learning Containers for PyTorch 1.6.0: August 3, 2020
- AWS Deep Learning Containers for PyTorch 1.5.1: June 19, 2020
- AWS Deep Learning Containers for PyTorch 1.4: June 06, 2020
- AWS Deep Learning Containers for PyTorch 1.5.0: May 05, 2020
- AWS Deep Learning Containers for PyTorch 1.4: April 03, 2020

MXNet

- AWS Deep Learning Containers MXNet 1.7.0: September 17, 2020
- AWS Deep Learning Containers with Elastic Inference for MXNet 1.5.1: August 31, 2020
- AWS Deep Learning Containers MXNet 1.6.0: July 20, 2020
- AWS Deep Learning Containers MXNet 1.6.0: April 03, 2020

Document History for Deep Learning Containers Developer Guide

The following table describes the documentation for this release of Deep Learning Containers.

- **API version: latest**
- **Latest documentation update:** February 26, 2020

update-history-change	update-history-description	update-history-date
Apache MXNet with Horovod (p. 1)	Apache MXNet tutorial was added to the developer guide.	February 26, 2020
Deep Learning Containers Developer Guide Launch (p. 1)	Deep Learning Containers setup and tutorials were added to the developer guide.	February 17, 2020

AWS glossary

For the latest AWS terminology, see the AWS glossary in the *AWS General Reference*.