aws

Developer Guide

AWS Deep Learning Containers

AWS Deep Learning Containers: Developer Guide

Table of Contents

What are AWS Deep Learning Containers?

AWS Deep Learning Containers are pre-built Docker images that make it easier to run popular deep learning frameworks and tools on AWS. They provide a consistent, up-to-date, secure, and optimized runtime environment for your deep learning applications hosted on AWS infrastructure. To get started, see Getting Started with AWS Deep Learning Containers.

Key Features

Pre-Installed Deep Learning Frameworks

AWS Deep Learning Containers include pre-installed and configured versions of leading deep learning frameworks such as TensorFlow and PyTorch. This eliminates the need to build and maintain your own Docker images from scratch.

Hardware Acceleration

AWS Deep Learning Containers are optimized for CPU-based, GPU-accelerated, and AWS silicon-based deep learning. They support CUDA, cuDNN, and other necessary libraries for leveraging the power of GPU-based Amazon EC2 instances, as well as AWS-designed chips like Graviton CPUs and GPUs, AWS Trainium, and Intel's Habana-Gaudi processors.

AWS Service Integration

AWS Deep Learning Containers seamlessly integrate with a variety of AWS services, including SageMaker AI, Amazon Elastic Container Service (ECS), Amazon Elastic Kubernetes Service (EKS), Amazon EC2, and AWS ParallelCluster. This makes it easy to deploy and run your deep learning models and applications on AWS infrastructure.

Secure and Regularly Updated

AWS regularly maintains and updates the AWS Deep Learning Containers to ensure you have access to the latest versions of deep learning frameworks and dependencies. This helps keep your AWS-based deep learning environment secure and up-to-date, without the overhead of managing security patches and updates yourself. Keeping your deep learning containers updated with the latest security patches can be a resource-intensive task, but AWS Deep Learning Containers eliminate this burden by providing regular, automatic updates. This ensures your deep learning

environment remains secure and current, without requiring significant manual effort on your part. By automating the update process, AWS Deep Learning Containers allow you to focus on developing your deep learning models and applications, rather than worrying about the underlying infrastructure and security upkeep, which can improve your team's productivity and allow you to more efficiently leverage the latest deep learning capabilities in your AWS-hosted projects.

Use Cases

AWS Deep Learning Containers are particularly useful in the following AWS-based deep learning scenarios:

Model Training

Use AWS Deep Learning Containers to train your deep learning models on CPU-based, GPU-accelerated, or AWS silicon-powered Amazon EC2 instances, or leverage multi-node training on AWS ParallelCluster or SageMaker Hyperpod.

Model Deployment

Deploy your trained models using the AWS Deep Learning Containers for scalable, production-ready inference on AWS, such as through SageMaker AI.

Experimentation and Prototyping

Quickly spin up deep learning development environments on AWS using the pre-configured containers. AWS Deep Learning Containers are the default option for notebook in SageMaker AI Studio, making it easy to get started with experimentation and prototyping.

Continuous Integration and Delivery

Integrate the containers into your AWS-based CI/CD pipelines, such as those using Amazon ECS or Amazon EKS, for consistent, automated deep learning workloads.

Getting Started with AWS 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.

Use Cases

- For information on using Deep Learning Containers with SageMaker AI, see the Use Your Own Algorithms or Models with SageMaker AI Documentation.
- To learn about using Deep Learning Containers with SageMaker AI HyperPod on EKS, see Orchestrating SageMaker HyperPod clusters with Amazon EKS SageMaker AI .

Topics

- Customize Deep Learning Containers
- Amazon EC2 Tutorials
- Amazon ECS tutorials
- Amazon EKS Tutorials

Customize Deep Learning Containers

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

Example

In the following example Dockerfile, we added the AWS samples GitHub repo which contains many deep learning model examples into the PyTorch Inference deep learning container.

```
# Take base container
FROM 763104351884.dkr.ecr.us-east-1.amazonaws.com/pytorch-inference:2.4-cpu-py311-ec2

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

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
```

Amazon EC2 Tutorials

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

Before starting the following tutorials, complete the steps in Amazon EC2 setup.

Contents

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
```

Next steps

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

Training

This section shows how to run training on AWS Deep Learning Containers for Amazon EC2 using PyTorch and TensorFlow.

Contents

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 P4 Instances
- Amazon EC2 P5 Instances

- Amazon EC2 G5 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
```

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 PyTorch with Deep Learning Containers, see PyTorch Inference .

Inference

This section shows how to run inference on AWS Deep Learning Containers for Amazon Elastic Compute Cloud using PyTorch, and TensorFlow.

Contents

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 `multi-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. Inference is served using multi-model-server, which can support any framework as the backend. For more information, see multi-model-server.

For CPU instances:

```
$ docker run -itd --name mms -p 80:8080  -p 8081:8081 <your container image id> \
multi-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>
 \
multi-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
```

TensorFlow 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/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-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`.

Next steps

To learn about using custom entrypoints with Deep Learning Containers on Amazon ECS, see Custom Entrypoints.

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 PyTorch, and TensorFlow.

Before starting the following tutorials, complete the steps in Amazon ECS setup.

Contents

Amazon ECS setup

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

Contents

- Setting up Amazon ECS for Deep Learning Containers

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.
- Verify that you have the Amazon ECS Container Instance role. For more information, see 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 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*.
- Create a new security group or update an existing security group to have the ports open for your desired inference server.
  - 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 inform your instance type selection. If you select a GPU instance type, be sure to then choose the Amazon ECS GPU-optimized AMI. For CPU-based work, you can use the Amazon Linux or Amazon Linux 2 ECS-optimized AMIs. For more information about compatible instance types and Amazon ECS-optimized AMI IDs, see 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 console at https://console.aws.amazon.com/ecs/v2.
2. Select the cluster with your registered Amazon EC2 instances.
3. On the **Cluster** page, choose **Infrastructure**.
4. Under **Container instances**, verify that the the `instance-id` created in previous step is displayed. Also, note the values for the **CPU available** and **Memory available** 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.

Training

> **Note**
>
> This section shows how to run training on AWS Deep Learning Containers for Amazon Elastic Container Service using PyTorch and TensorFlow.

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

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 console at https://console.aws.amazon.com/ecs/v2.
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 **Logs**, 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.

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 classic 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 **Logs**, 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.

Inference

This section shows how to run inference on AWS Deep Learning Containers for Amazon Elastic Container Service (Amazon ECS) using PyTorch, and TensorFlow.

> **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

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.

Next steps

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

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 console at https://console.aws.amazon.com/ecs/v2.

   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 **Logs**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

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

   g. Under **Name** and then **Network Bindings**, under **External Link** note the IP address for port 8501 and use it in the next step.

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 console at https://console.aws.amazon.com/ecs/v2.

   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 **Logs**, choose **View logs in CloudWatch**. This takes you to the CloudWatch console to view the training progress logs.

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

g. Under **Name** and then **Network Bindings**, under **External Link** note the IP address for port 8501 and use it in the next step.

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.

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/multi-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

Amazon EKS tutorials provide training and inference examples and show how to set up and use AWS Deep Learning Containers on:

- Amazon Elastic Kubernetes Service (Amazon EKS)
- Kubeflow on AWS

Kubeflow on AWS is an optimized open source distribution of Kubeflow for Amazon Elastic Kubernetes Service (Amazon EKS). For more information, see AWS features for Kubeflow.

> **Note**
>
> All of the training and inference examples in this section run on a single node cluster.

The installation instructions for Kubeflow on AWS provide steps to create an Amazon EKS cluster before deploying the AWS distribution of Kubeflow.

Contents

Amazon EKS Setup

This section provides installation instructions to setup a deep learning environment running AWS Deep Learning Containers on Amazon Elastic Kubernetes Service (Amazon EKS).

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 software that is not AWS, so it 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 by following the steps in Creating an IAM user in your AWS account.
2. Get the credentials of this user.
   a. Open the IAM console at https://console.aws.amazon.com/iam/.
   b. Under `Users`, select your user.
   c. Select `Security Credentials`.
   d. Select `Create access key`.
   e. Download the key pair or copy the information for use later.
3. Add the following policies to your IAM user. These policies provide the required access for Amazon EKS, IAM, and Amazon Elastic Compute Cloud (Amazon EC2).
   a. Select `Permissions`.
   b. Select `Add permissions`.
   c. Select `Create policy`.
   d. From the `Create policy` window, select the `JSON` tab.
   e. Paste the following content.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
```

```
            "Effect": "Allow",
            "Action": "eks:*",
            "Resource": "*"
        }
    ]
}
```

f. Name the policy `EKSFullAccess` and create the policy.

g. Navigate back to the `Grant permissions` window.

h. Select `Attach existing policies directly`.

i. Search for `EKSFullAccess`, and select the check box.

j. Search for `AWSCloudFormationFullAccess`, and select the check box.

k. Search for `AmazonEC2FullAccess`, and select the check box.

l. Search for `IAMFullAccess`, and select the check box.

m. Search `AmazonEC2ContainerRegistryReadOnly`, and select the check box.

n. Search `AmazonEKS_CNI_Policy`, and select the check box.

o. Search `AmazonS3FullAccess`, and select the check box.

p. 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.

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 command corresponding to your operating system in *Amazon EKS User Guide*'s installation instructions. For more information about `eksctl`, see also eksctl documentation.
4. Install `kubectl` by following the steps in the Installing kubectl guide.

> **Note**
>
> You must use a `kubectl` version that is within one minor version difference of your Amazon EKS cluster control plane version. For example, a 1.18 `kubectl` client works with Kubernetes 1.17, 1.18 and 1.19 clusters.

5. Install `aws-iam-authenticator` by running the following commands. For more information on aws-iam-authenticator, see Installing `aws-iam-authenticator`.

```
$ curl -o aws-iam-authenticator https://amazon-eks.s3.us-
west-2.amazonaws.com/1.19.6/2021-01-05/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`.

GPU Clusters

1. Examine the following command to create a cluster using a p3.8xlarge instance type. You must make the following modifications before you run it.
   - `name` is what you 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.
   - `eks-version` is the Amazon EKS kubernetes version. For the supported Amazon EKS versions, see Available Amazon EKS Kubernetes versions.
   - `nodes` is the number of instances you want in your cluster. In this example, we're starting with three nodes.

- `node-type` refers to an instance class.
- `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 is 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.

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

```
$ eksctl create cluster <cluster-name> \
                      --version <eks-version> \
                      --nodes 3 \
                      --node-type=<p3.8xlarge> \
                      --timeout=40m \
                      --ssh-access \
                      --ssh-public-key <key_pair_name> \
                      --region <us-east-1> \
                      --zones=us-east-1a,us-east-1b,us-east-1d \
                      --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 [NVIDIA device plugin for Kubernetes] on your cluster with the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/
v1.12/nvidia-device-plugin.yml
$ kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/
v0.9.0/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.

Habana Clusters

Refer to the previous discussion on using the **eksctl** command to launch a GPU cluster, and modify `node-type` to use an instance with Habana Gaudi accelerators, such as the [DL1 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. In that case, the command will use the current actively controlled cluster (`current-context`).

```
$ 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 create -f
 https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v0.9.0/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 might switch frequently 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, visit Training or Inference.

Contents

Training

Once you've created a cluster using the steps in Amazon EKS Setup, you can use it to run training jobs. For training, you can use either a CPU, GPU, or distributed GPU example depending on the nodes in your cluster. The topics in the following sections show how to use PyTorch training example.

Contents

CPU Training

This section is for training on CPU-based containers.

Contents

PyTorch CPU training

This tutorial guides you through training a PyTorch model on your single node CPU pod.

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.

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.

Next steps

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

GPU Training

This section is for training on GPU-based clusters.

Contents

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.

Next steps

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

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.

Distributed GPU Training

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

Contents

Set up your cluster for distributed training

To run distributed training on EKS, you need the following components installed on your cluster.

- The default installation of Kubeflow with required components, such as PyTorch operators, and the NVIDIA plugin.
- MPI operators.

Download and run the script to install the required components in the cluster.

```
$ wget -O install_kubeflow.sh https://raw.githubusercontent.com/aws/deep-
learning-containers/master/test/dlc_tests/eks/eks_manifest_templates/kubeflow/
install_kubeflow.sh
$ chmod +x install_kubeflow.sh
$ ./install_kubeflow.sh <EKS_CLUSTER_NAME> <AWS_REGION>
```

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. Verify that the PyTorch custom resource is installed.

   ```
   $ kubectl get crd
   ```

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

2. Ensure that the NVIDIA plugin `daemonset` is running.

```
$ kubectl get daemonset -n kubeflow
```

The output should should look similar to the following.

```
NAME                           DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE
 SELECTOR AGE
nvidia-device-plugin-daemonset   3         3      3        3        3    <none>
 35h
```

3. Use the following text to create a gloo-based distributed data parallel job. Save it in a file named `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/aws-samples-
pytorch-training:1.7.1-gpu-py36-cu110-ubuntu18.04-example"
            args:
              - "--backend"
              - "gloo"
              - "--epochs"
              - "5"
    Worker:
      replicas: 2
      restartPolicy: OnFailure
      template:
        spec:
          containers:
          - name: "pytorch"
```

```
            image: "763104351884.dkr.ecr.us-east-1.amazonaws.com/aws-samples-
pytorch-training:1.7.1-gpu-py36-cu110-ubuntu18.04-example"
            args:
              - "--backend"
              - "gloo"
              - "--epochs"
              - "5"
            resources:
              limits:
                nvidia.com/gpu: 1
```

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

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

5. 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>
```

Inference

Once you've created a cluster using the steps in Amazon EKS Setup, you can use it to run inference jobs. For inference, you can use either a CPU or GPU example depending on the nodes in your cluster. Inference supports only single node configurations. The following topics show how to run inference with AWS Deep Learning Containers on EKS using PyTorch, and TensorFlow.

Contents

CPU Inference

This section guides you on running inference on Deep Learning Containers for EKS CPU clusters using PyTorch, and TensorFlow.

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

Contents

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.

   ```
   $ 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. 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:
        - multi-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.

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.

```
$ 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
]
}
```

Next steps

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

GPU Inference

This section shows how to run inference on Deep Learning Containers for EKS GPU clusters PyTorch, and TensorFlow.

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

Contents

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:
        - multi-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.

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.

   ```
   $ 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
]
}
```

Next steps

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

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/multi-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

The following are common errors that might be returned in the command line when using AWS Deep Learning Containers on an Amazon EKS cluster. Each error is followed by a solution to the error.

Troubleshooting

Topics

Setup Errors

The following errors might be returned when setting up Deep Learning Containers on your Amazon EKS cluster.

- **Error: registry `kubeflow` does not exist**

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

  To solve this error, run the following command.

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

- **Error: context deadline exceeded**

  ```
  $ 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
  ```

To solve this error, verify that you have not exceeded capacity for your account. You can also try to create your cluster in a different region.

- **Error: The connection to the server localhost:8080 was refused**

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

To solve this error, copy the cluster to the Kubernetes configuration by running the following.

```
cp ~/.kube/eksctl/clusters/<cluster-name> ~/.kube/config
```

- **Error: handle object: patching object from cluster: merging object with existing state: Unauthorized**

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

This error is due to a concurrency issue that can occur when multiple users with different authorization or credentials credentials try to start jobs on the same cluster. Verify that you are starting a job on the correct cluster.

- **Error: Could not create app; directory '/home/ubuntu/kubeflow-tf-hvd' already exists**

```
$ 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
```

You can safely ignore this warning. However, you may have additional cleanup to do inside that folder. To simplify cleanup, delete the folder.

Usage Errors

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

If you see this error message while using the Amazon EKS cluster, run the NVIDIA device plugin installation step again. Verify that you have targeted the right cluster by either passing in the specific config file or switching your active cluster to the targeted cluster.

Cleanup Errors

The following errors might be returned when cleaning up the resources of your Amazon EKS cluster.

- **Error: the server doesn't have a resource type "*namspace*"**

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

  Verify the spelling of your namespace is correct.

- **Error: the server has asked for the client to provide credentials**

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

  To solve this error, verify 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.

- **Error: finding app root from starting path: : unable to find ksonnet project**

```
$ ks delete default
  ERROR finding app root from starting path: : unable to find ksonnet project
```

  To solve this error, verify that you are in the directory created by the ksonnet app. This is the folder where `ks init` was run.

- **Error: Error from server (NotFound): pods "openmpi-master" not found**

```
$ kubectl logs -n ${NAMESPACE} -f ${COMPONENT}-master > results/benchmark_1.out
  Error from server (NotFound): pods "openmpi-master" not found
```

  This error might be caused by trying to access resources after the context is deleted. Deleting the default context causes the corresponding resources to be deleted as well.

Release Notes

Check the latest release notes for AWS Deep Learning Containers built for specific machine learning frameworks, infrastructures, and AWS services.

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

PyTorch Deep Learning Containers

Version	Type	Service	Architecture	Release Note
2.5	Training	EC2 ECS EKS	X86	AWS Deep Learning Containers for PyTorch 2.5 (Training on EC2, ECS, and EKS): November 25, 2024
2.5	Training	SageMaker	X86	AWS Deep Learning Containers for PyTorch 2.5 (Training on SageMaker AI): November 25, 2024
2.5	Inference	EC2 ECS EKS	X86	AWS Deep Learning Containers for PyTorch 2.5

Version	Type	Service	Architecture	Release Note
				(Inference on EC2, ECS, and EKS): November 13, 2024
2.5	Inference	SageMaker	X86	AWS Deep Learning Containers for PyTorch 2.5 (Inference on SageMaker AI): November 13, 2024
2.5	Inference	EC2 ECS EKS	Arm64	AWS Deep Learning Containers for PyTorch 2.5 ARM64 (Inference on EC2, ECS, and EKS): December 11, 2024
2.5	Inference	SageMaker	Arm64	AWS Deep Learning Containers for PyTorch 2.5 ARM64 (Inference on SageMaker AI): December 11, 2024

Version	Type	Service	Architecture	Release Note
2.4	Training	EC2 ECS EKS	X86	AWS Deep Learning Containers for PyTorch 2.4 (Training on EC2, ECS, and EKS): September 26, 2024
2.4	Training	SageMaker	X86	AWS Deep Learning Containers for PyTorch 2.4 (Training on SageMaker AI): September 26, 2024
2.4	Inference	EC2 ECS EKS	X86	AWS Deep Learning Containers for PyTorch 2.4 (Inference on EC2, ECS, and EKS): September 26, 2024
2.4	Inference	SageMaker	X86	AWS Deep Learning Containers for PyTorch 2.4 (Inference on SageMaker AI): October 3, 2024

Version	Type	Service	Architecture	Release Note
2.4	Inference	EC2 ECS EKS	Arm64	AWS Deep Learning Containers for PyTorch 2.4 Graviton (Inference on EC2, ECS, and EKS): July 11, 2024
2.4	Inference	SageMaker	Arm64	AWS Deep Learning Containers for PyTorch 2.4 Graviton (Inference on SageMaker AI): July 11, 2024
2.3	Training	EC2 ECS EKS	X86	AWS Deep Learning Containers for PyTorch 2.3 (Training on EC2, ECS, and EKS): May 30, 2024
2.3	Training	SageMaker	X86	AWS Deep Learning Containers for PyTorch 2.3 (Training on SageMaker AI): May 30, 2024

Version	Type	Service	Architecture	Release Note
2.3	Inference	EC2 ECS EKS	X86	AWS Deep Learning Containers for PyTorch 2.3 (Inference on EC2, ECS, and EKS): July 11, 2024
2.3	Inference	SageMaker	X86	AWS Deep Learning Containers for PyTorch 2.3 (Inference on SageMaker AI): July 11, 2024
2.3	Inference	EC2 ECS EKS	Arm64	AWS Deep Learning Containers for PyTorch 2.3 Graviton (Inference on EC2, ECS, and EKS): July 11, 2024
2.3	Inference	SageMaker	Arm64	AWS Deep Learning Containers for PyTorch 2.3 Graviton (Inference on SageMaker AI): July 11, 2024

Version	Type	Service	Architecture	Release Note
2.2	Training	EC2 ECS EKS	X86	AWS Deep Learning Containers for PyTorch 2.2 (Training on EC2, ECS, and EKS): March 20, 2024
2.2	Training	SageMaker	X86	AWS Deep Learning Containers for PyTorch 2.2 (Training on SageMaker AI): March 20, 2024
2.2	Inference	EC2 ECS EKS	X86	AWS Deep Learning Containers for PyTorch 2.2 (Inference on EC2, ECS, and EKS): March 20, 2024
2.2	Inference	SageMaker	X86	AWS Deep Learning Containers for PyTorch 2.2 (Inference on SageMaker AI): March 20, 2024

Version	Type	Service	Architecture	Release Note
2.2	Inference	EC2 ECS EKS	Arm64	AWS Deep Learning Containers for PyTorch 2.2 Graviton (Inference on EC2, ECS, and EKS): April 25, 2024
2.2	Inference	SageMaker	Arm64	AWS Deep Learning Containers for PyTorch 2.2 Graviton (Inference on SageMaker AI): April 25, 2024

TensorFlow Deep Learning Containers

Version	Type	Service	Architecture	Release Note
2.16	Training	EC2 ECS EKS	X86	AWS Deep Learning Containers for TensorFlow 2.16 (Training on EC2, ECS, and EKS): August 20, 2024
2.16	Training	SageMaker	X86	AWS Deep Learning

Version	Type	Service	Architecture	Release Note
				Containers for TensorFlow 2.16 (Training on SageMaker AI): August 20, 2024
2.16	Inference	EC2 ECS EKS	X86	AWS Deep Learning Containers for TensorFlow 2.16 (Inference on EC2, ECS, and EKS): July 19, 2024
2.16	Inference	SageMaker	X86	AWS Deep Learning Containers for TensorFlow 2.16 (Inference on SageMaker AI): July 19, 2024
2.16	Inference	EC2 ECS EKS	Arm64	AWS Deep Learning Containers for TensorFlow 2.16 Graviton (Inference on EC2, ECS, and EKS): July 19, 2024

Version	Type	Service	Architecture	Release Note
2.16	Inference	SageMaker	Arm64	AWS Deep Learning Containers for TensorFlow 2.16 Graviton (Inference on SageMaker AI): July 19, 2024

Receive Notifications on New Updates

You can receive notifications whenever a new DLC is released. Notifications are published with Amazon SNS using the following topic.

```
arn:aws:sns:us-west-2:767397762724:dlc-updates
```

Messages are posted here when a new DLC is published. The version, metadata, and regional image URI's of the container will be included in the message.

These messages can be received using several different methods. We recommend that you use the following method.

1. Open the Amazon SNS Console.
2. In the navigation bar, change the AWS Region to **US West (Oregon)**, if necessary. You must select the region where the SNS notification that you're subscribing to was created.
3. In the navigation pane, choose **Subscriptions, Create subscription**.
4. For the **Create subscription** dialog box, do the following:
    a. For **Topic ARN**, copy and paste the following Amazon Resource Name (ARN): **`arn:aws:sns:us-west-2:767397762724:dlc-updates`**
    b. For **Protocol**, choose one from **[Amazon SQS, AWS Lambda, Email, Email-JSON]**
    c. For **Endpoint**, enter the email address or **Amazon Resource Name (ARN)** of resource that you will use to receive the notifications.
    d. Choose **Create subscription**.
5. You receive a confirmation email with the subject line *AWS Notification - Subscription Confirmation*. Open the email and choose **Confirm subscription** to complete your subscription.

Support Policy

AWS Deep Learning Containers (DLCs) simplify image configuration for deep learning workloads and are optimized with the latest frameworks, hardware, drivers, libraries, and operating systems. This page details the framework support policy for DLCs.

Supported Frameworks

Reference the following AWS Deep Learning Containers Framework Support Policy table to **check which frameworks and versions are actively supported**.

Refer to **End of patch** to check how long AWS supports current versions that are actively supported by the origin framework's maintenance team. Frameworks and versions are available in single-framework DLCs.

> **Note**
>
> In the framework version *x.y.z*, *x* refers to the major version, *y* refers to the minor version, and *z* refers to the patch version. For example, for TensorFlow 2.6.5, the major version is 2, the minor version is 6, and the patch version is 5.

Frequently Asked Questions

- What framework versions get security patches?
- What images does AWS publish when new framework versions are released?
- What images get new SageMaker AI/AWS features?
- How is current version defined in the Supported Frameworks table?
- What if I am running a version that is not in the Supported Frameworks table?
- Do DLCs support previous versions of TensorFlow?
- How can I find the latest patched image for a supported framework version?
- How frequently are new images released?
- Will my instance be patched in place while my workload is running?

What framework versions get security patches?

If the framework version is labeled **Supported** in the AWS Deep Learning Containers Framework Support Policy table, it gets security patches.

What images does AWS publish when new framework versions are released?

We publish new DLCs soon after new versions of TensorFlow and PyTorch are released. This includes major versions, major-minor versions, and major-minor-patch versions of frameworks. We also update images when new versions of drivers and libraries become available. For more information on image maintenance, see When does active support for my framework version end?

What images get new SageMaker AI/AWS features?

New features typically release in the latest version of DLCs for PyTorch and TensorFlow. Refer to the release notes for a specific image for details on new SageMaker AI or AWS features. For a list of available DLCs, see Release Notes for AWS Deep Learning Containers. For more information on image maintenance, see When does active support for my framework version end?

How is current version defined in the Supported Frameworks table?

The current version in the AWS Deep Learning Containers Framework Support Policy table refers to the newest framework version that AWS makes available on GitHub. Each latest release includes updates to the drivers, libraries, and relevant packages in the DLC. For information on image maintenance, see When does active support for my framework version end?

What if I am running a version that is not in the Supported Frameworks table?

If you are running a version that is not in the AWS Deep Learning Containers Framework Support Policy table, you may not have the most updated drivers, libraries, and relevant packages. For a more up-to-date version, we recommend that you upgrade to one of the supported frameworks available using the latest DLC of your choice. For a list of available DLCs, see Release Notes for AWS Deep Learning Containers.

Do DLCs support previous versions of TensorFlow?

No. We support the latest patch version of each framework's latest major version released 365 days from its initial GitHub release as stated in the AWS Deep Learning Containers Framework Support Policy table. For more information, see What if I am running a version that is not in the Supported Frameworks table?

How can I find the latest patched image for a supported framework version?

To use a DLC with the latest framework version, browse the DLC GitHub release tags to find the sample image URI of your choice and use it to pull the latest available Docker image. The framework version that you choose must be labeled **Supported** in the AWS Deep Learning Containers Framework Support Policy table.

How frequently are new images released?

Providing updated patch versions is our highest priority. We routinely create patched images at the earliest opportunity. We monitor for newly patched framework versions (ex. TensorFlow 2.9 to TensorFlow 2.9.1) and new minor release versions (ex. TensorFlow 2.9 to TensorFlow 2.10) and make them available at the earliest opportunity. When an existing version of TensorFlow is released with a new version of CUDA, we release a new DLC for that version of TensorFlow with support for the new CUDA version.

Will my instance be patched in place while my workload is running?

No. Patch updates for DLC are not "in-place" updates.

You must delete the existing image on your instance and pull the latest container image without terminating you instance.

What happens when a new patched or updated framework version is available?

Regularly check the release notes page for your image. We encourage you to upgrade to new patched or updated frameworks when they are available. For a list of available DLCs, see Release Notes for AWS Deep Learning Containers.

Are dependencies updated without changing the framework version?

We update dependencies without changing the framework version. However, if a dependency update causes an incompatibility, we create an image with a different version. Be sure to check the Release Notes for AWS Deep Learning Containers for updated dependency information.

When does active support for my framework version end?

DLC images are immutable. Once they are created they do not change. There are four main reasons why active support for a framework version ends:

- Framework version (patch) upgrades
- AWS security patches
- End of patch date (Aging out)
- Dependency end-of-support

> **Note**
>
> Due to the frequency of version patch upgrades and security patches, we recommend checking the release notes page for your DLC often, and upgrading when changes are made.

Framework version (patch) upgrades

If you have a DLC workload based on TensorFlow 2.7.0 and TensorFlow releases version 2.7.1 on GitHub, then AWS releases a new DLC with TensorFlow 2.7.1. The previous images with 2.7.0 are longer actively maintained once the new image with TensorFlow 2.7.1 is released. The DLC with TensorFlow 2.7.0 does not receive further patches. The DLC release notes page for TensorFlow 2.7

is then updated with the latest information. There is no individual release note page for each minor patch.

New DLCs created due to patch upgrades are designated with updated release tags. If changes are not backwards compatible, the tag will change major versions rather than minor versions (ex. v1.0 will change to v2.0 rather than v 1.2).

AWS security patches

If you have a workload based on an image with TensorFlow 2.7.0 and AWS makes a security patch, then a new version of the DLC is released for TensorFlow 2.7.0. The previous version of the images with TensorFlow 2.7.0 is no longer actively maintained. For more information, see Will my instance be patched in place while my workload is running? For steps on finding the latest DLC, see How can I find the latest patched image for a supported framework version?

New DLCs created due to patch upgrades are designated with updated release tags. If changes are not backwards compatible, the tag will change major versions rather than minor versions (ex. v1.0 will change to v2.0 rather than v 1.2).

End of patch date (Aging out)

DLCs hit their end of patch date 365 days after the GitHub release date.

> **Important**
>
> We make an exception when there is a major framework update. For example. if TensorFlow 1.15 updates to TensorFlow 2.0, then we continue to support the most recent version of TensorFlow 1.15 for a period of two years from the date of the GitHub release or six months after the origin framework maintenance team drops support, whichever date is earlier.

Dependency end-of-support

If you are running a workload on a TensorFlow 2.7.0 DLC image with Python 3.6 and that version of Python is marked for end-of-support, then all DLC images based on Python 3.6 will no longer be actively maintained. Similarly, if an OS version like Ubuntu 16.04 is marked for end-of-support, then all DLC images that are dependent on Ubuntu 16.04 will no longer be actively maintained.

Will images with framework versions that are no longer actively maintained be patched?

No. Images that are no longer actively maintained will not have new releases.

How do I use an older framework version?

To use a DLC with an older framework version, browse the DLC GitHub release tags to find the image URI of your choice and use it to pull the docker image.

How do I stay up-to-date with support changes in frameworks and their versions?

Stay up-to-date with DLC frameworks and versions using the DLC release notes, and the Available Deep Learning Containers Images page.

Do I need a commercial license to use the Anaconda Repository?

Anaconda shifted to a commercial licensing model for certain users. Actively maintained DLCs have been migrated to the publicly available open-source version of Conda (conda-forge) from the Anaconda channel.

> **Warning**
>
> If you are actively using Anaconda to install and manage your packages and their dependencies in a DLC that is no longer actively maintained, you are responsible for complying with the governing license from the Anaconda Repository, if you determine that the terms apply to you. Alternatively, you can migrate to one of the currently-supported DLCs listed in the AWS Deep Learning Containers Framework Support Policy table or you can install packages using conda-forge as a source.

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

The AWS shared responsibility model applies to data protection in AWS Deep Learning Containers. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- 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 sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text 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 tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Identity and Access Management in AWS Deep Learning 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

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the

recommended method to sign requests yourself, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might 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 Multi-factor authentication in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one 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 don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

IAM Users and Groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

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 Use cases for IAM users 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. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). 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 Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **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 Cross account resource access in IAM in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

- **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.
- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

Managing Access Using Policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) 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*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-Based Policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed 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 Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-Based Policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access Control Lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other Policy Types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an 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 Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) 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.

Monitoring and Usage Tracking in AWS Deep Learning Containers

Your AWS Deep Learning Containers do 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 Studio.

Usage Tracking

AWS uses customer feedback and usage information to improve the quality of the services and software we offer to customers. We have added usage data collection to the supported AWS Deep Learning Containers in order to better understand customer usage and guide future improvements. Usage tracking for Deep Learning Containers is activated by default. Customers can change their settings at any point of time to activate or deactivate usage tracking.

Usage tracking for AWS Deep Learning Containers collects the *instance ID*, *frameworks*, *framework versions*, *container types*, and *Python versions* used for the containers. AWS also logs the event time in which it receives this metadata.

No information on the commands used within the containers is collected or retained. No other information about the containers is collected or retained.

To opt out of usage tracking, set the `OPT_OUT_TRACKING` environment variable to true.

```
OPT_OUT_TRACKING=true
```

Failure Rate Tracking

When using a first-party Amazon SageMaker AI AWS Deep Learning Containers container, the SageMaker AI team will collect failure rate metadata to improve the quality of AWS Deep Learning Containers. Failure rate tracking for AWS Deep Learning Containers is active by default. Customers can change their settings to activate or deactivate failure rate tracking when creating an Amazon SageMaker AI endpoint.

Failure rate tracking for AWS Deep Learning Containers collects the *Instance ID*, *ModelServer name*, *ModelServer version*, *ErrorType*, and *ErrorCode*. AWS also logs the event time in which it receives this metadata.

No information on the commands used within the containers is collected or retained. No other information about the containers is collected or retained.

To opt out of failure rate tracking, set the `OPT_OUT_TRACKING` environment variable to `true`.

```
OPT_OUT_TRACKING=true
```

Usage Tracking in the following Framework Versions

While we recommend updating to supported Deep Learning Containers, to opt-out of usage tracking for Deep Learning Containers that use these frameworks, set the `OPT_OUT_TRACKING` environment variable to true **and** use a custom entry point to disable the call for the following services:

- Amazon EC2 Custom Entrypoints
- Amazon ECS Custom Entrypoints
- Amazon EKS Custom Entrypoints

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 AI. 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.

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

Change	Description	Date
Deep Learning Containers Developer Guide Launch	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 Glossary Reference*.