Amazon Elastic Inference

Developer Guide

Amazon Elastic Inference: Developer Guide

Table of Contents

What Is Amazon Elastic Inference?

Amazon Elastic Inference (Elastic Inference) is a resource you can attach to your Amazon Elastic Compute Cloud CPU instances, Amazon Deep Learning Containers, and SageMaker instances. Elastic Inference helps you accelerate your deep learning (DL) inference workloads. Elastic Inference accelerators come in multiple sizes and help you build intelligent capabilities into your applications.

Elastic Inference distributes model operations defined by TensorFlow, Apache MXNet (MXNet), and PyTorch between low-cost, DL inference accelerators and the CPU of the instance. Elastic Inference also supports the open neural network exchange (ONNX) format through MXNet.

Prerequisites

You need an Amazon Web Services account and should be familiar with launching an Amazon EC2, Amazon Deep Learning Containers, or SageMaker instances to successfully run Amazon Elastic Inference. To launch an Amazon EC2 instance, complete the steps in Setting up with Amazon EC2. Amazon S3 resources are required for installing packages via pip. For more information about setting up Amazon S3 resources, see the Amazon Simple Storage Service Getting Started Guide.

Pricing for Amazon Elastic Inference

You are charged for each second that an Elastic Inference accelerator is attached to an instance in the `running` state. You are not charged for an accelerator attached to an instance that is in the `pending`, `stopping`, `stopped`, `shutting-down`, or `terminated` state. You are also not charged when an Elastic Inference accelerator is in the `unknown` or `impaired` state.

You do not incur AWS PrivateLink charges for VPC endpoints to the Elastic Inference service when you have accelerators provisioned in the subnet.

For more information about pricing by Region for Elastic Inference, see Elastic Inference Pricing.

Elastic Inference Uses

You can use Elastic Inference in the following use cases:

- For Elastic Inference-enabled TensorFlow and TensorFlow 2 with Python, see Using TensorFlow Models with Elastic Inference (p. 11)
- For Elastic Inference-enabled MXNet with Python, Java, and Scala, see Using MXNet Models with Elastic Inference (p. 30)
- For Elastic Inference-enabled PyTorch with Python, see Using PyTorch Models with Elastic Inference (p. 48)
- For Elastic Inference with SageMaker, see MXNet Elastic Inference with SageMaker (p. 58)
- For Amazon Deep Learning Containers with Elastic Inference on Amazon EC2, Amazon ECS, and SageMaker, see Using Amazon Deep Learning Containers With Elastic Inference (p. 60)

- For security information on Elastic Inference, see Security in Amazon Elastic Inference (p. 74)
- To troubleshoot your Elastic Inference workflow, see Troubleshooting (p. 84)

Next Up

Amazon Elastic Inference Basics (p. 2)

Amazon Elastic Inference Basics

When you configure an Amazon EC2 instance to launch with an Elastic Inference accelerator, AWS finds available accelerator capacity. It then establishes a network connection between your instance and the accelerator.

The following Elastic Inference accelerator types are available. You can attach any Elastic Inference accelerator type to any Amazon EC2 instance type.

Accelerator Type	FP32 Throughput (TFLOPS)	FP16 Throughput (TFLOPS)	Memory (GB)
eia2.medium	1	8	2
eia2.large	2	16	4
eia2.xlarge	4	32	8

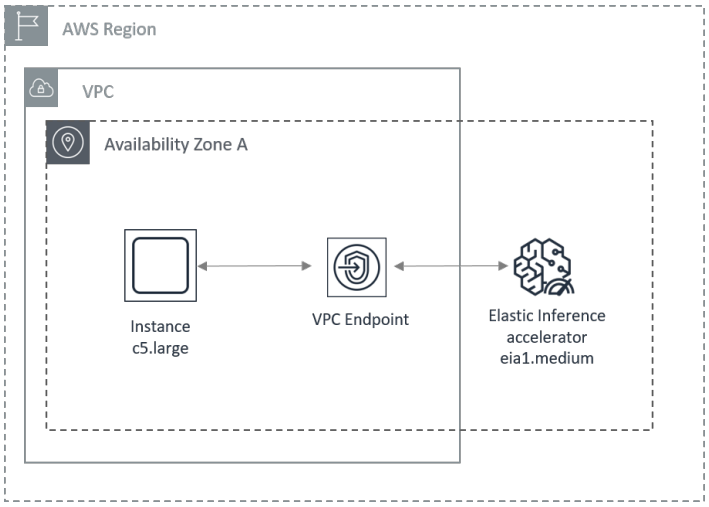
You can attach multiple Elastic Inference accelerators of various sizes to a single Amazon EC2 instance when launching the instance. With multiple accelerators, you can run inference for multiple models on a single fleet of Amazon EC2 instances. If your models require different amounts of GPU memory and compute capacity, you can choose the appropriate accelerator size to attach to your CPU. For faster response times, load your models to an Elastic Inference accelerator once and continue making inference calls on multiple accelerators without unloading any models for each call. By attaching multiple accelerators to a single instance, you avoid deploying multiple fleets of CPU or GPU instances and the associated cost. For more information on attaching multiple accelerators to a single instance, see Using TensorFlow Models with Elastic Inference, Using MXNet Models with Elastic Inference , and Using PyTorch Models with Elastic Inference.

> **Note**
> Attaching multiple Elastic Inference accelerators to a single Amazon EC2 instance requires that the instance has AWS Deep Learning AMI (DLAMI) version 25 or later. For more information on the AWS Deep Learning AMI, see What Is the AWS Deep Learning AMI?.

An Elastic Inference accelerator is not part of the hardware that makes up your instance. Instead, the accelerator is attached through the network using an AWS PrivateLink endpoint service. The endpoint service routes traffic from your instance to the Elastic Inference accelerator configured with your instance.

> **Note**
> An Elastic Inference accelerator cannot be modified through the management console of your instance.

Before you launch an instance with an Elastic Inference accelerator, you must create an AWS PrivateLink endpoint service. Only a single endpoint service is needed in every Availability Zone to connect instances with Elastic Inference accelerators. A single endpoint service can span multiple Availability Zones. For more information, see VPC Endpoint Services (AWS PrivateLink).



You can use Amazon Elastic Inference enabled TensorFlow, TensorFlow Serving, Apache MXNet, or PyTorch libraries to load models and make inference calls. The modified versions of these frameworks automatically detect the presence of Elastic Inference accelerators. They then optimally distribute the model operations between the Elastic Inference accelerator and the CPU of the instance. The AWS Deep Learning AMIs include the latest releases of Amazon Elastic Inference enabled TensorFlow, TensorFlow Serving, MXNet, and PyTorch. If you are using custom AMIs or container images, you can download and install the required TensorFlow, Apache MXNet, and PyTorch libraries from Amazon S3.

Elastic Inference Uses

You can use Elastic Inference in the following use cases:

- For Elastic Inference-enabled TensorFlow and TensorFlow 2 with Python, see Using TensorFlow Models with Elastic Inference (p. 11)
- For Elastic Inference-enabled MXNet with Python, Java, and Scala, see Using MXNet Models with Elastic Inference (p. 30)
- For Elastic Inference-enabled PyTorch with Python, see Using PyTorch Models with Elastic Inference (p. 48)
- For Elastic Inference with SageMaker, see MXNet Elastic Inference with SageMaker (p. 58)
- For Amazon Deep Learning Containers with Elastic Inference on Amazon EC2, Amazon ECS, and SageMaker, see Using Amazon Deep Learning Containers With Elastic Inference (p. 60)
- For security information on Elastic Inference, see Security in Amazon Elastic Inference (p. 74)
- To troubleshoot your Elastic Inference workflow, see Troubleshooting (p. 84)

Before you get started with Amazon Elastic Inference

Amazon Elastic Inference Service Limits

Before you start using Elastic Inference accelerators, be aware of the following limitations:

Limit	Description		
Elastic Inference accelerator instance limit	You can attach up to five Elastic Inference accelerators by default to each instance at a time, and only during instance launch. This is adjustable. We recommend testing the optimal setup before deploying to production.		
Elastic Inference Sharing	You cannot share an Elastic Inference accelerator between instances.		
Elastic Inference Transfer	You cannot detach an Elastic Inference accelerator from an instance or transfer it to another instance. If you no longer need an Elastic Inference accelerator, you must terminate your instance. You cannot change the Elastic Inference accelerator type. Terminate the instance and launch a new instance with a different Elastic Inference accelerator specification.		
Supported Libraries	Only the Amazon Elastic Inference enhanced MXNet, TensorFlow, and PyTorch libraries can make inference calls to Elastic Inference accelerators.		
Elastic Inference Attachment	Elastic Inference accelerators can only be attached to instances in a VPC.		
Reserving accelerator capacity	Pricing for Elastic Inference accelerators is available at On-Demand Instance rates only. You can attach an accelerator to a		

Limit	Description		
	Reserved Instance, Scheduled Reserved Instance, or Spot Instance. However, the On-Demand Instance price for the Elastic Inference accelerator applies. You cannot reserve or schedule Elastic Inference accelerator capacity.		

Choosing an Instance and Accelerator Type for Your Model

Demands on CPU compute resources, CPU memory, GPU-based acceleration, and GPU memory vary significantly between different types of deep learning models. The latency and throughput requirements of the application also determine the amount of instance compute and Elastic Inference acceleration you need. Consider the following when you choose an instance and accelerator type combination for your model:

- Before you evaluate the right combination of resources for your model or application stack, you should determine the target latency, throughput needs, and constraints. For example, let's assume your application must respond within 300 milliseconds (ms). If data retrieval (including any authentication) and preprocessing takes 200ms, you have a 100-ms window to work with for the inference request. Using this analysis, you can determine the lowest cost infrastructure combination that meets these targets.
- Start with a reasonably small combination of resources. For example, a budget-friendly `c5.xlarge` CPU instance type along with an `eia2.medium` accelerator type. This combination has been tested to work well for various computer vision workloads (including a large version of ResNet: ResNet-200). The combination gives comparable or better performance than a more costly `p2.xlarge` GPU instance. You can then resize the instance or accelerator type depending on your latency targets.
- I/O data transfer between instance and accelerator adds to inference latency because Elastic Inference accelerators are attached over the network.
- If you use multiple models with your accelerator, you might need a larger accelerator size to better support both compute and memory needs. This also applies if you use the same model from multiple application processes on the instance.
- You can convert your model to mixed precision, which uses the higher FP16 TFLOPS of the accelerator, to provide lower latency and higher performance.

Using Amazon Elastic Inference with EC2 Auto Scaling

When you create an Auto Scaling group, you can specify the information required to configure the Amazon EC2 instances. This includes Elastic Inference accelerators. To do this, specify a launch template with your instance configuration and the Elastic Inference accelerator type.

Working with Amazon Elastic Inference

To work with Amazon Elastic Inference, set up and launch your Amazon Elastic Compute Cloud instance with Elastic Inference. After that, use Elastic Inference accelerators that are powered by the Elastic Inference enabled versions of TensorFlow, TensorFlow Serving, Apache MXNet (MXNet), and PyTorch. You can do this with few changes to your code.

Topics

Setting Up to Launch Amazon EC2 with Elastic Inference

The most convenient way to set up Amazon EC2 with Elastic Inference uses the Elastic Inference setup script described in https://aws.amazon.com/blogs/machine-learning/launch-ei-accelerators-in-minutes-with-the-amazon-elastic-inference-setup-tool-for-ec2/. To manually launch an instance and associate it with an Elastic Inference accelerator, first configure your security groups and AWS PrivateLink endpoint services. Then, configure an instance role with the Elastic Inference policy.

Topics

Configuring Your Security Groups for Elastic Inference

You need two security groups. One for inbound and outbound traffic for the new Elastic Inference VPC endpoint. A second one for outbound traffic for the associated Amazon EC2 instances that you launch.

Configure Your Security Groups for Elastic Inference

To configure a security group for an Elastic Inference accelerator (console)

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the left navigation pane, choose **Security**, **Security Groups**.

3. Choose **Create Security Group**
4. Under **Create Security Group**, specify a name and description for the security group and choose the ID of the VPC. Choose **Create** and then choose **Close**.
5. Select the check box next to your security group and choose **Actions, Edit inbound rules**. Add a rule to allow HTTPS traffic on port 443 as follows:

   a. Choose **Add Rule**.
   b. For **Type**, select **HTTPS**.
   c. For **Source**, specify a CIDR block (for example, 0.0.0.0/0) or the security group for your instance.
   d. To allow traffic for port 22 to the EC2 instance, repeat the procedure. For **Type**, select **SSH**.
   e. Choose **Save rules** and then choose **Close**.
6. Choose **Edit outbound rules**. Choose **Add rule**. To allow traffic for all ports, for **Type**, select **All Traffic**.
7. Choose **Save rules**.

To configure a security group for an Elastic Inference accelerator (AWS CLI)

1. Create a security group using the create-security-group command:

```
aws ec2 create-security-group
--description insert a description for the security group
--group-name assign a name for the security group
[--vpc-id enter the VPC ID]
```

2. Create inbound rules using the authorize-security-group-ingress command:

```
aws ec2 authorize-security-group-ingress --group-id insert the security group ID --
protocol tcp --port 443 --cidr 0.0.0.0/0
```

```
aws ec2 authorize-security-group-ingress --group-id insert the security group ID --
protocol tcp --port 22 --cidr 0.0.0.0/0
```

3. The default setting for outbound rules allows all traffic from all ports for this instance.

Configuring AWS PrivateLink Endpoint Services

Elastic Inference uses VPC endpoints to privately connect the instance in your VPC with their associated Elastic Inference accelerator. Create a VPC endpoint for Elastic Inference before you launch instances with accelerators. This needs to be done just one time per VPC. For more information, see Interface VPC Endpoints (AWS PrivateLink).

To configure an AWS PrivateLink endpoint service (console)

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. In the left navigation pane, choose **Endpoints**, **Create Endpoint**.
3. For **Service category**, choose **Find service by name**.
4. For **Service Name**, select **com.amazonaws.*<your-region>*.elastic-inference.runtime**.

   For example, for the us-west-2 Region, select **com.amazonaws.us-west-2.elastic-inference.runtime**.
5. For **Subnets**, select one or more Availability Zones where the endpoint should be created. Where you plan to launch instances with accelerators, you must select subnets for the Availability Zone.

6. Enable the private DNS name and enter the security group for your endpoint. Choose **Create endpoint**. Note the VPC endpoint ID for later.
7. The security group that we configured for the endpoint in previous steps must allow inbound traffic to port 443.

To configure an AWS PrivateLink endpoint service (AWS CLI)

- Use the https://docs.aws.amazon.com/cli/latest/reference/ec2/create-vpc-endpoint.html command and specify the following: VPC ID, type of VPC endpoint (interface), service name, subnets to use the endpoint, and security groups to associate with the endpoint network interfaces. For information about how to set up a security group for your VPC endpoint, see the section called "Configuring Your Security Groups for Elastic Inference" (p. 6).

```
aws ec2 create-vpc-endpoint --vpc-id vpc-insert VPC ID --vpc-endpoint-type Interface
 --service-name com.amazonaws.us-west-2.elastic-inference.runtime --subnet-id
 subnet-insert subnet --security-group-id sg-insert security group ID
```

Configuring an Instance Role with an Elastic Inference Policy

To launch an instance with an Elastic Inference accelerator, you must provide an IAM role that allows actions on Elastic Inference accelerators.

To configure an instance role with an Elastic Inference policy (console)

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. In the left navigation pane, choose **Policies**, **Create Policy**.
3. Choose **JSON** and paste the following policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elastic-inference:Connect",
                "iam:List*",
                "iam:Get*",
                "ec2:Describe*",
                "ec2:Get*"
            ],
            "Resource": "*"
        }
    ]
}
```

   Note
   You may get a warning message about the Elastic Inference service not being recognizable. This is a known issue and does not block creation of the policy.

4. Choose **Review policy** and enter a name for the policy, such as `ec2-role-trust-policy.json`, and a description.
5. Choose **Create policy**.
6. In the left navigation pane, choose **Roles**, **Create role**.
7. Choose **AWS service**, **EC2**, **Next: Permissions**.

8. Select the name of the policy that you just created (`ec2-role-trust-policy.json`). Choose **Next: Tags**.
9. Provide a role name and choose **Create Role**.

When you create your instance, select the role under **Configure Instance Details** in the launch wizard.

To configure an instance role with an Elastic Inference policy (AWS CLI)

- To configure an instance role with an Elastic Inference policy, follow the steps in Creating an IAM Role. Add the following policy to your instance:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elastic-inference:Connect",
                "iam:List*",
                "iam:Get*",
                "ec2:Describe*",
                "ec2:Get*"
            ],
            "Resource": "*"
        }
    ]
}
```

Note
You may get a warning message about the Elastic Inference service not being recognizable. This is a known issue and does not block creation of the policy.

Launching an Instance with Elastic Inference

You can now configure Amazon EC2 instances with accelerators to launch within your subnet. Choose any supported Amazon EC2 instance type and Elastic Inference accelerator size. Elastic Inference accelerators are available to all current generation instance types. There are two accelerator types.

EIA2 is the second generation of Elastic Inference accelerators. It offers improved performance and increased memory. With up to 8 GB of GPU memory, EIA2 is a cost-effective resource for deploying machine learning (ML) models. Use it for applications such as image classification, object detection, automated speech recognition, and language translation. Your accelerator memory choices depend on the size of your input and models. You can choose from the following Elastic Inference accelerators:

- `eia2.medium` with 2 GB of accelerator memory
- `eia2.large` with 4 GB of accelerator memory
- `eia2.xlarge` with 8 GB of accelerator memory

Note: We continue to support EIA1 in three sizes: eia1.medium, eia1.large, and eia1.xlarge

You can launch an instance with Elastic Inference automatically by using the Amazon Elastic Inference setup tool for EC2, or manually using the console or AWS Command Line Interface.

To launch an instance with Elastic Inference (console)

1. Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.

2. Choose **Launch Instance**.
3. Under **Choose an Amazon Machine Image**, select an Amazon Linux or Ubuntu AMI. We recommend one of the Deep Learning AMIs.

   Note
   Attaching multiple Elastic Inference accelerators to a single Amazon EC2 instance requires that the instance has AWS Deep Learning AMI (DLAMI) Version 25 or later.
4. Under **Choose an Instance Type**, select the hardware configuration of your instance.
5. Choose **Next: Configure Instance Details**.
6. Under **Configure Instance Details**, check the configuration settings. Ensure that you are using the VPC with the security groups for the instance and the Elastic Inference accelerator that you set up earlier. For more information, see Configuring Your Security Groups for Elastic Inference (p. 6).
7. For **IAM role**, select the role that you created in the Configuring an Instance Role with an Elastic Inference Policy (p. 8) procedure.
8. Select **Add an Elastic Inference accelerator**.
9. Select the size and amount of Elastic Inference accelerators. Your options are: `eia2.medium`, `eia2.large`, and `eia2.xlarge`.
10. To add another Elastic Inference accelerator, choose **Add**. Then select the size and amount of accelerators to add.
11. (Optional) You can choose to add storage and tags by choosing **Next** at the bottom of the page. Or, you can let the instance wizard complete the remaining configuration steps for you.
12. In the Add Security Group step, choose the security group created previously.
13. Review the configuration of your instance and choose **Launch**.
14. You are prompted to choose an existing key pair for your instance or to create a new key pair. For more information, see Amazon EC2 Key Pairs..

    Warning
    Don't select the **Proceed without a key pair** option. If you launch your instance without a key pair, then you can't connect to it.
15. After making your key pair selection, choose **Launch Instances**.
16. A confirmation page lets you know that your instance is launching. To close the confirmation page and return to the console, choose **View Instances**.
17. Under **Instances**, you can view the status of the launch. It takes a short time for an instance to launch. When you launch an instance, its initial state is `pending`. After the instance starts, its state changes to `running`.
18. It can take a few minutes for the instance to be ready so that you can connect to it. Check that your instance has passed its status checks. You can view this information in the **Status Checks** column.

To launch an instance with Elastic Inference (AWS CLI)

To launch an instance with Elastic Inference at the command line, you need your key pair name, subnet ID, security group ID, AMI ID, and the name of the instance profile that you created in the section Configuring an Instance Role with an Elastic Inference Policy (p. 8). For the security group ID, use the one you created for your instance that contains the AWS PrivateLink endpoint. For more information, see Configuring Your Security Groups for Elastic Inference (p. 6)). For more information about the AMI ID, see Finding a Linux AMI.

1. Use the run-instances command to launch your instance and accelerator:

```
aws ec2 run-instances --image-id ami-image ID --instance-type m5.large --subnet-id
 subnet-subnet ID --elastic-inference-accelerator Type=eia2.large --key-name key
 pair name --security-group-ids sg-security group ID --iam-instance-profile
 Name="accelerator profile name"
```

To launch an instance with multiple accelerators, you can add multiple `Type` parameters to `--elastic-inference-accelerator`.

```
aws ec2 run-instances --image-id ami-image ID --instance-type m5.large --subnet-
id subnet-subnet ID --elastic-inference-accelerator Type=eia2.large,Count=2
 Type=eia2.xlarge  --key-name key pair name --region region name --security-group-ids
 sg-security group ID
```

2. When the `run-instances` operation succeeds, your output is similar to the following. The `ElasticInferenceAcceleratorArn` identifies the Elastic Inference accelerator.

```
"ElasticInferenceAcceleratorAssociations": [
 {
 "ElasticInferenceAcceleratorArn": "arn:aws:elastic-
inference:us-west-2:204044812891:elastic-inference-accelerator/
eia-3e1de7c2f64a4de8b970c205e838af6b",
 "ElasticInferenceAcceleratorAssociationId": "eia-assoc-031f6f53ddcd5f260",
 "ElasticInferenceAcceleratorAssociationState": "associating",
 "ElasticInferenceAcceleratorAssociationTime": "2018-10-05T17:22:20.000Z"
 }
 ],
```

You are now ready to run your models using TensorFlow, MXNet, or PyTorch on the provided AMI.

Once your Elastic Inference accelerator is running, you can use the `describe-accelerators` AWS CLI. This command returns information about the accelerator, such as the region it is in and the name of the accelerator. For more information about the usage of this command, see the Elastic Inference AWS CLI Command Reference.

Using TensorFlow Models with Elastic Inference

Amazon Elastic Inference (Elastic Inference) is available only on instances that were launched with an Elastic Inference accelerator.

The Elastic Inference enabled version of TensorFlow allows you to use Elastic Inference accelerators with minimal changes to your TensorFlow code.

Topics

Elastic Inference Enabled TensorFlow

Preinstalled EI Enabled TensorFlow

The Elastic Inference enabled packages are available in the AWS Deep Learning AMI. AWS Deep Learning AMIs come with supported TensorFlow version and `ei_for_tf` pre-installed. Elastic Inference enabled TensorFlow 2 requires AWS Deep Learning AMI v28 or higher. You also have Docker container options with *Using Amazon Deep Learning Containers With Elastic Inference* (p. 60).

Installing EI Enabled TensorFlow

If you're not using a AWS Deep Learning AMI instance, you can download the packages from the Amazon S3 bucket to build it in to your own Amazon Linux or Ubuntu AMIs.

Install `ei_for_tf`.

```
pip install -U ei_for_tf*.whl
```

If the TensorFlow version is lower than the required version, pip upgrades TensorFlow to the appropriate version. If the TensorFlow version is higher than the required version, there will be a warning about the incompatibility. Your program fails at run-time if the TensorFlow version incompatibility isn't fixed.

Additional Requirements and Considerations

TensorFlow 2.0 Differences

Starting with TensorFlow 2.0, the Elastic Inference package is a separate pip wheel, instead of an enhanced TensorFlow pip wheel. The prefix for import statements for the Elastic Inference specific API have changed from tensorflow.contrib.ei to ei_for_tf.

To see the compatible TensorFlow version for a specific ei_for_tf version, see the `ei_for_tf_compatibility.txt` file in the Amazon S3 bucket.

Model Formats Supported

Elastic Inference supports the TensorFlow saved_model format via TensorFlow Serving.

Warmup

Elastic Inference TensorFlow Serving provides a warmup feature to preload models and reduce the delay that is typical of the first inference request. Amazon Elastic Inference TensorFlow Serving only supports warming up the "fault-finders" signature definition.

TensorFlow Elastic Inference with Python

With Elastic Inference TensorFlow Serving, the standard TensorFlow Serving interface remains unchanged. The only difference is that the entry point is a different binary named `amazonei_tensorflow_model_server`.

TensorFlow Serving and Predictor are the only inference modes that Elastic Inference supports. If you haven't tried TensorFlow Serving before, we recommend that you try the TensorFlow Serving tutorial first.

This release of Elastic Inference TensorFlow Serving has been tested to perform well and provide cost-saving benefits with the following deep learning use cases and network architectures (and similar variants):

Use Case	Example Network Topology
Image Recognition	Inception, ResNet, MVCNN
Object Detection	SSD, RCNN

Use Case	Example Network Topology
Neural Machine Translation	GNMT

Note
These tutorials assume usage of a DLAMI with v26 or later, and Elastic Inference enabled Tensorflow.

Topics

Activate the Tensorflow Elastic Inference Environment

1.
   - (Option for Python 3) - Activate the Python 3 TensorFlow Elastic Inference environment:

     ```
     $ source activate amazonei_tensorflow_p36
     ```

   - (Option for Python 2) - Activate the Python 2.7 TensorFlow Elastic Inference environment:

     ```
     $ source activate amazonei_tensorflow_p27
     ```

2. The remaining parts of this guide assume you are using the `amazonei_tensorflow_p27` environment.

If you are switching between Elastic Inference enabled MXNet, TensorFlow, or PyTorch environments, you must stop and then start your instance in order to reattach the Elastic Inference accelerator. Rebooting is not sufficient since the process requires a complete shut down.

Use Elastic Inference with TensorFlow Serving

The following is an example of serving a Single Shot Detector (SSD) with a ResNet backbone.

Serve and Test Inference with an Inception Model

1. Download the model.

   ```
   curl -O https://s3-us-west-2.amazonaws.com/aws-tf-serving-ei-example/ssd_resnet.zip
   ```

2. Unzip the model.

   ```
   unzip ssd_resnet.zip -d /tmp
   ```

3. Download a picture of three dogs to your home directory.

   ```
   curl -O https://raw.githubusercontent.com/awslabs/mxnet-model-server/master/docs/
   images/3dogs.jpg
   ```

4. Use the built-in `EI Tool` to get the device ordinal number of all attached Elastic Inference accelerators. For more information on `EI Tool`, see Monitoring Elastic Inference Accelerators.

```
/opt/amazon/ei/ei_tools/bin/ei describe-accelerators --json
```

Your output should look like the following:

```
{
  "ei_client_version": "1.5.0",
  "time": "Fri Nov  1 03:09:38 2019",
  "attached_accelerators": 2,
  "devices": [
    {
      "ordinal": 0,
      "type": "eia1.xlarge",
      "id": "eia-679e4c622d584803aed5b42ab6a97706",
      "status": "healthy"
    },
    {
      "ordinal": 1,
      "type": "eia1.xlarge",
      "id": "eia-6c414c6ee37a4d93874afc00825c2f28",
      "status": "healthy"
    }
  ]
}
```

5. Navigate to the folder where `AmazonEI_TensorFlow_Serving` is installed and run the following command to launch the server. Set `EI_VISIBLE_DEVICES` to the device ordinal or device ID of the attached Elastic Inference accelerator that you want to use. This device will then be accessible using `id 0`. For more information on `EI_VISIBLE_DEVICES`, see Monitoring Elastic Inference Accelerators. Note, `model_base_path` must be an absolute path.

```
EI_VISIBLE_DEVICES=<ordinal number> amazonei_tensorflow_model_server --
model_name=ssdresnet --model_base_path=/tmp/ssd_resnet50_v1_coco --port=9000
```

6. While the server is running in the foreground, launch another terminal session. Open a new terminal and activate the TensorFlow environment.

```
source activate amazonei_tensorflow_p27
```

7. Use your preferred text editor to create a script that has the following content. Name it `ssd_resnet_client.py`. This script will take an image filename as a parameter and get a prediction result from the pretrained model.

```
from __future__ import print_function

import grpc
import tensorflow as tf
from PIL import Image
import numpy as np
import time
import os
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc

tf.app.flags.DEFINE_string('server', 'localhost:9000',
                           'PredictionService host:port')
tf.app.flags.DEFINE_string('image', '', 'path to image in JPEG format')
FLAGS = tf.app.flags.FLAGS

coco_classes_txt = "https://raw.githubusercontent.com/amikelive/coco-labels/master/
coco-labels-paper.txt"
```

```
local_coco_classes_txt = "/tmp/coco-labels-paper.txt"
# it's a file like object and works just like a file
os.system("curl -o %s -O %s"%(local_coco_classes_txt, coco_classes_txt))
NUM_PREDICTIONS = 5
with open(local_coco_classes_txt) as f:
  classes = ["No Class"] + [line.strip() for line in f.readlines()]


def main(_):
  channel = grpc.insecure_channel(FLAGS.server)
  stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)

  # Send request
  with Image.open(FLAGS.image) as f:
    f.load()
    # See prediction_service.proto for gRPC request/response details.
    data = np.asarray(f)
    data = np.expand_dims(data, axis=0)

    request = predict_pb2.PredictRequest()
    request.model_spec.name = 'ssdresnet'
    request.inputs['inputs'].CopyFrom(
        tf.contrib.util.make_tensor_proto(data, shape=data.shape))
    result = stub.Predict(request, 60.0)  # 10 secs timeout
    outputs = result.outputs
    detection_classes = outputs["detection_classes"]
    detection_classes = tf.make_ndarray(detection_classes)
    num_detections = int(tf.make_ndarray(outputs["num_detections"])[0])
    print("%d detection[s]" % (num_detections))
    class_label = [classes[int(x)]
                   for x in detection_classes[0][:num_detections]]
    print("SSD Prediction is ", class_label)


if __name__ == '__main__':
  tf.app.run()
```

8. Now run the script passing the server location, port, and the dog photo's filename as the parameters.

```
python ssd_resnet_client.py  --server=localhost:9000 --image 3dogs.jpg
```

Use Elastic Inference with the TensorFlow EIPredictor API

Elastic Inference TensorFlow packages for Python 2 and 3 provide an EIPredictor API. This API function provides you with a flexible way to run models on Elastic Inference accelerators as an alternative to using TensorFlow Serving. The EIPredictor API provides a simple interface to perform repeated inference on a pretrained model. The following code sample shows the available parameters.

Note
`accelerator_id` should be set to the device's ordinal number, not its `ID`.

```
ei_predictor = EIPredictor(model_dir,
           signature_def_key=None,
           signature_def=None,
           input_names=None,
           output_names=None,
           tags=None,
           graph=None,
           config=None,
           use_ei=True,
           accelerator_id=<device ordinal number>)
```

```
output_dict = ei_predictor(feed_dict)
```

EIPredictor can be used in the following ways:

```
//EIPredictor class picks inputs and outputs from default serving signature def with tag
 “serve”. (similar to TF predictor)
ei_predictor = EIPredictor(model_dir)

//EI Predictor class picks inputs and outputs from the signature def picked using the
 signtaure_def_key (similar to TF predictor)
ei_predictor = EIPredictor(model_dir, signature_def_key='predict')

// Signature_def can be provided directly (similar to TF predictor)
ei_predictor = EIPredictor(model_dir, signature_def= sig_def)

// You provide the input_names and output_names dict.
// similar to TF predictor

ei_predictor = EIPredictor(model_dir,
                           input_names,
                           output_names)


// tag is used to get the correct signature def. (similar to TF predictor)

ei_predictor = EIPredictor(model_dir, tags='serve')
```

Additional EI Predictor functionality includes:

- Support for frozen models.

```
// For Frozen graphs, model_dir takes a file name , input_names and output_names
// input_names and output_names should be provided in this case.

ei_predictor = EIPredictor(model_dir,
                           input_names=None,
                           output_names=None )
```

- Ability to disable use of Elastic Inference by using the `use_ei` flag, which defaults to `True`. This is useful for testing EIPredictor against TensorFlow Predictor.
- EIPredictor can also be created from a TensorFlow Estimator. Given a trained Estimator, you can first export a SavedModel. See the SavedModel documentation for more details. The following shows example usage:

```
saved_model_dir = estimator.export_savedmodel(my_export_dir, serving_input_fn)
ei_predictor = EIPredictor(export_dir=saved_model_dir)

// Once the EIPredictor is created, inference is done using the following:
output_dict = ei_predictor(feed_dict)
```

Use Elastic Inference with TensorFlow Predictor Example

Installing Elastic Inference TensorFlow

Elastic Inference enabled TensorFlow comes bundled in the AWS Deep Learning AMI. You can also download pip wheels for Python 2 and 3 from the Elastic Inference S3 bucket. Follow these instructions to download and install the pip package:

Choose the tar file for the Python version and operating system of your choice from the S3 bucket. Copy the path to the tar file and run the following command:

```
curl -O [URL of the tar file of your choice]
```

To open the tar the file:

```
tar -xvzf [name of tar file]
```

Try the following example to serve different models, such as ResNet, using a Single Shot Detector (SSD).

Serve and Test Inference with an SSD Model

1. Download the model. If you already downloaded the model in the Serving example, skip this step.

   ```
   curl -O https://s3-us-west-2.amazonaws.com/aws-tf-serving-ei-example/ssd_resnet.zip
   ```

2. Unzip the model. Again, you may skip this step if you already have the model.

   ```
   unzip ssd_resnet.zip -d /tmp
   ```

3. Download a picture of three dogs to your current directory.

   ```
   curl -O https://raw.githubusercontent.com/awslabs/mxnet-model-server/master/docs/
   images/3dogs.jpg
   ```

4. Use the built-in `EI Tool` to get the device ordinal number of all attached Elastic Inference accelerators. For more information on `EI Tool`, see Monitoring Elastic Inference Accelerators.

   ```
   /opt/amazon/ei/ei_tools/bin/ei describe-accelerators --json
   ```

   Your output should look like the following:

   ```
   {
     "ei_client_version": "1.5.0",
     "time": "Fri Nov  1 03:09:38 2019",
     "attached_accelerators": 2,
     "devices": [
       {
         "ordinal": 0,
         "type": "eia1.xlarge",
         "id": "eia-679e4c622d584803aed5b42ab6a97706",
         "status": "healthy"
       },
       {
         "ordinal": 1,
         "type": "eia1.xlarge",
         "id": "eia-6c414c6ee37a4d93874afc00825c2f28",
         "status": "healthy"
       }
     ]
   }
   ```

   You use the device ordinal of your desired Elastic Inference accelerator to create a Predictor.

5. Open a text editor, such as vim, and paste the following inference script. Replace the `accelerator_id` value with the device ordinal of the desired Elastic Inference accelerator. This value must be an integer. Save the file as `ssd_resnet_predictor.py`.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import sys
import numpy as np
import tensorflow as tf
import matplotlib.image as mpimg
from tensorflow.contrib.ei.python.predictor.ei_predictor import EIPredictor

tf.app.flags.DEFINE_string('image', '', 'path to image in JPEG format')
FLAGS = tf.app.flags.FLAGS

coco_classes_txt = "https://raw.githubusercontent.com/amikelive/coco-labels/master/
coco-labels-paper.txt"
local_coco_classes_txt = "/tmp/coco-labels-paper.txt"
# it's a file like object and works just like a file
os.system("curl -o %s -O %s"%(local_coco_classes_txt, coco_classes_txt))
NUM_PREDICTIONS = 5
with open(local_coco_classes_txt) as f:
  classes = ["No Class"] + [line.strip() for line in f.readlines()]


def get_output(eia_predictor, test_input):
  pred = None
  for curpred in range(NUM_PREDICTIONS):
    pred = eia_predictor(test_input)

  num_detections = int(pred["num_detections"])
  print("%d detection[s]" % (num_detections))
  detection_classes = pred["detection_classes"][0][:num_detections]
  print([classes[int(i)] for i in detection_classes])


def main(_):

  img = mpimg.imread(FLAGS.image)
  img = np.expand_dims(img, axis=0)
  ssd_resnet_input = {'inputs': img}

  print('Running SSD Resnet on EIPredictor using specified input and outputs')
  eia_predictor = EIPredictor(
      model_dir='/tmp/ssd_resnet50_v1_coco/1/',
      input_names={"inputs": "image_tensor:0"},
      output_names={"detection_classes": "detection_classes:0", "num_detections":
 "num_detections:0",
                    "detection_boxes": "detection_boxes:0"},
      accelerator_id=<device ordinal>
  )
  get_output(eia_predictor, ssd_resnet_input)

  print('Running SSD Resnet on EIPredictor using default Signature Def')
  eia_predictor = EIPredictor(
      model_dir='/tmp/ssd_resnet50_v1_coco/1/',
  )
  get_output(eia_predictor, ssd_resnet_input)


if __name__ == "__main__":
  tf.app.run()
```

6. Run the inference script.

```
python ssd_resnet_predictor.py --image 3dogs.jpg
```

For more tutorials and examples, see the TensorFlow Python API.

Use Elastic Inference with the TensorFlow Keras API

The Keras API has become an integral part of the machine learning development cycle because of its simplicity and ease of use. Keras enables rapid prototyping and development of machine learning constructs. Elastic Inference provides an API that offers native support for Keras. Using this API, you can directly use your Keras model, h5 file, and weights to instantiate a Keras-like Object. This object supports the native Keras prediction APIs, while fully utilizing Elastic Inference in the backend. The following code sample shows the available parameters:

```
EIKerasModel(model,
             weights=None,
             export_dir=None,
             ):
    """Constructs an `EIKerasModel` instance.

    Args:
      model: A model object that either has its weights already set, or will be set with
 the weights argument.
             A model file that can be loaded
      weights (Optional): A weights object, or weights file that can be loaded, and will be
 set to the model object
      export_dir: A folder location to save your model as a SavedModelBundle

    Raises:
      RuntimeError: If eager execution is enabled.
    """
```

EIKerasModel can be used as follows:

```
#Loading from Keras Model Object
from tensorflow.contrib.ei.python.keras.ei_keras import EIKerasModel
model = Model()
# Build Keras Model in the normal fashion
x = # input data
ei_model = EIKerasModel(model) # Only additional step to use EI
res = ei_model.predict(x)

#Loading from Keras h5 File
from tensorflow.contrib.ei.python.keras.ei_keras import EIKerasModel
x = # input data
ei_model = EIKerasModel("keras_model.h5") # Only additional step to use EI
res = ei_model.predict(x)

#Loading from Keras h5 File and Weights file
from tensorflow.contrib.ei.python.keras.ei_keras import EIKerasModel
x = # input data
ei_model = EIKerasModel("keras_model.json", weights="keras_weights.h5") # Only additional
 step to use EI
res = ei_model.predict(x)
```

Additionally, Elastic Inference enabled Keras includes Predict API Support:

```
tf.keras
```

```
def predict(  x,
              batch_size=None,
              verbose=0,
              steps=None,
              max_queue_size=10, #Not supported
              workers=1, #Not Supported
              use_multiprocessing=False): #Not Supported


 Native Keras
 def predict( x,
              batch_size=None,
              verbose=0,
              steps=None,
              callbacks=None) # Not supported
```

TensorFlow Keras API Example

In this example, you use a trained ResNet-50 model to classify an image of an African Elephant from ImageNet.

Test Inference with a Keras Model

1. Activate the Elastic Inference TensorFlow Conda Environment

   ```
   source activate amazonei_tensorflow_p27
   ```

2. Download an image of an African Elephant to your current directory.

   ```
   curl -O https://upload.wikimedia.org/wikipedia/commons/5/59/
   Serengeti_Elefantenbulle.jpg
   ```

3. Open a text editor, such as vim, and paste the following inference script. Save the file as `test_keras.py`.

   ```
   # Resnet Example
   from tensorflow.keras.applications.resnet50 import ResNet50
   from tensorflow.keras.preprocessing import image
   from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions
   from tensorflow.contrib.ei.python.keras.ei_keras import EIKerasModel
   import numpy as np
   import time
   import os
   ITERATIONS = 20

   model = ResNet50(weights='imagenet')
   ei_model = EIKerasModel(model)
   folder_name = os.path.dirname(os.path.abspath(__file__))
   img_path = folder_name + '/Serengeti_Elefantenbulle.jpg'
   img = image.load_img(img_path, target_size=(224, 224))
   x = image.img_to_array(img)
   x = np.expand_dims(x, axis=0)
   x = preprocess_input(x)
   # Warm up both models
   _ = model.predict(x)
   _ = ei_model.predict(x)
   # Benchmark both models
   for each in range(ITERATIONS):
     start = time.time()
     preds = model.predict(x)
     print("Vanilla iteration %d took %f" % (each, time.time() - start))
   for each in range(ITERATIONS):
   ```

```
  start = time.time()
  ei_preds = ei_model.predict(x)
  print("EI iteration %d took %f" % (each, time.time() - start))
# decode the results into a list of tuples (class, description, probability)
# (one such list for each sample in the batch)
print('Predicted:', decode_predictions(preds, top=3)[0])
print('EI Predicted:', decode_predictions(ei_preds, top=3)[0])
```

4. Run the inference script.

```
python test_keras.py
```

5. Your output should be a list of predictions, as well as their respective confidence score.

```
('Predicted:', [(u'n02504458', u'African_elephant', 0.9081173), (u'n01871265',
 u'tusker', 0.07836755), (u'n02504013', u'Indian_elephant', 0.011482777)])
('EI Predicted:', [(u'n02504458', u'African_elephant', 0.90811676), (u'n01871265',
 u'tusker', 0.07836751), (u'n02504013', u'Indian_elephant', 0.011482781)])
```

For more tutorials and examples, see the TensorFlow Python API.

TensorFlow 2 Elastic Inference with Python

With Elastic Inference TensorFlow 2 Serving, the standard TensorFlow 2 Serving interface remains unchanged. The only difference is that the entry point is a different binary named `amazonei_tensorflow2_model_server`.

TensorFlow 2 Serving and Predictor are the only inference modes that Elastic Inference supports. If you haven't tried TensorFlow 2 Serving before, we recommend that you try the TensorFlow Serving tutorial first.

This release of Elastic Inference TensorFlow Serving has been tested to perform well and provide cost-saving benefits with the following deep learning use cases and network architectures (and similar variants):

Use Case	Example Network Topology
Image Recognition	Inception, ResNet, MVCNN
Object Detection	SSD, RCNN
Neural Machine Translation	GNMT

Note
These tutorials assume usage of a DLAMI with v28 or later, and Elastic Inference enabled Tensorflow 2.

Topics

Activate the Tensorflow 2 Elastic Inference Environment

1. Choose one of the following options:
   - **Python 3** - Activate the Python 3 TensorFlow 2 Elastic Inference environment:

     ```
     $ source activate amazonei_tensorflow2_p36
     ```

   - **Python 2** - Activate the Python 2.7 TensorFlow 2 Elastic Inference environment:

     ```
     $ source activate amazonei_tensorflow2_p27
     ```

2. The remaining parts of this guide assume you are using the `amazonei_tensorflow2_p27` environment.

Use Elastic Inference with TensorFlow 2 Serving

The following is an example of serving a Single Shot Detector (SSD) with a ResNet backbone.

To serve and test inference with an inception model

1. Download the model.

   ```
   curl -O https://s3-us-west-2.amazonaws.com/aws-tf-serving-ei-example/ssd_resnet.zip
   ```

2. Unzip the model.

   ```
   unzip ssd_resnet.zip -d /tmp
   ```

3. Download a picture of three dogs to your home directory.

   ```
   curl -O https://raw.githubusercontent.com/awslabs/mxnet-model-server/master/docs/
   images/3dogs.jpg
   ```

4. Use the built-in `EI Tool` to get the device ordinal number of all attached Elastic Inference accelerators. For more information on `EI Tool`, see Monitoring Elastic Inference Accelerators.

   ```
   /opt/amazon/ei/ei_tools/bin/ei describe-accelerators --json
   ```

   Your output should look like the following:

   ```
   {
     "ei_client_version": "1.5.0",
     "time": "Fri Nov  1 03:09:38 2019",
     "attached_accelerators": 2,
     "devices": [
       {
         "ordinal": 0,
         "type": "eia1.xlarge",
         "id": "eia-679e4c622d584803aed5b42ab6a97706",
         "status": "healthy"
       },
       {
         "ordinal": 1,
         "type": "eia1.xlarge",
         "id": "eia-6c414c6ee37a4d93874afc00825c2f28",
         "status": "healthy"
   ```

```
    }
  ]
}
```

5. Navigate to the folder where `AmazonEI_TensorFlow_Serving` is installed and run the following command to launch the server. Set `EI_VISIBLE_DEVICES` to the device ordinal or device ID of the attached Elastic Inference accelerator that you want to use. This device will then be accessible using `id 0`. `model_base_path` must be an absolute path. For more information on `EI_VISIBLE_DEVICES`, see Monitoring Elastic Inference Accelerators.

```
EI_VISIBLE_DEVICES=<ordinal number> amazonei_tensorflow2_model_server
                --model_name=ssdresnet
                --model_base_path=/tmp/ssd_resnet50_v1_coco
                --port=9000
```

6. While the server is running in the foreground, launch another terminal session. Open a new terminal and activate the TensorFlow environment.

```
source activate amazonei_tensorflow2_p27
```

7. Use your preferred text editor to create a script that has the following content. Name it `ssd_resnet_client.py`. This script will take an image filename as a parameter and get a prediction result from the pretrained model.

```
from __future__ import print_function

import grpc
import tensorflow as tf
from PIL import Image
import numpy as np
import time
import os
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2_grpc

tf.compat.v1.app.flags.DEFINE_string('server', 'localhost:9000',
                           'PredictionService host:port')
tf.compat.v1.app.flags.DEFINE_string('image', '', 'path to image in JPEG format')
FLAGS = tf.compat.v1.app.flags.FLAGS

coco_classes_txt = "https://raw.githubusercontent.com/amikelive/coco-labels/master/
coco-labels-paper.txt"
local_coco_classes_txt = "/tmp/coco-labels-paper.txt"
# it's a file like object and works just like a file
os.system("curl -o %s -O %s"%(local_coco_classes_txt, coco_classes_txt))
NUM_PREDICTIONS = 5
with open(local_coco_classes_txt) as f:
  classes = ["No Class"] + [line.strip() for line in f.readlines()]


def main(_):
  channel = grpc.insecure_channel(FLAGS.server)
  stub = prediction_service_pb2_grpc.PredictionServiceStub(channel)

  # Send request
  with Image.open(FLAGS.image) as f:
    f.load()
    # See prediction_service.proto for gRPC request/response details.
    data = np.asarray(f)
    data = np.expand_dims(data, axis=0)

    request = predict_pb2.PredictRequest()
    request.model_spec.name = 'ssdresnet'
```

```
    request.inputs['inputs'].CopyFrom(
        tf.make_tensor_proto(data, shape=data.shape))
    result = stub.Predict(request, 60.0)  # 10 secs timeout
    outputs = result.outputs
    detection_classes = outputs["detection_classes"]
    detection_classes = tf.make_ndarray(detection_classes)
    num_detections = int(tf.make_ndarray(outputs["num_detections"])[0])
    print("%d detection[s]" % (num_detections))
    class_label = [classes[int(x)]
                   for x in detection_classes[0][:num_detections]]
    print("SSD Prediction is ", class_label)


if __name__ == '__main__':
  tf.compat.v1.app.run()
```

8. Now run the script passing the server location, port, and the dog photo's filename as the parameters.

```
python ssd_resnet_client.py  --server=localhost:9000 --image 3dogs.jpg
```

Use Elastic Inference with the TensorFlow 2 EIPredictor API

Elastic Inference TensorFlow packages for Python 2 and 3 provide an EIPredictor API. This API function provides you with a flexible way to run models on Elastic Inference accelerators as an alternative to using TensorFlow 2 Serving. The EIPredictor API provides a simple interface to perform repeated inference on a pretrained model. The following code sample shows the available parameters.

Note
`accelerator_id` should be set to the device's ordinal number, not its `ID`.

```
ei_predictor = EIPredictor(model_dir,
           signature_def_key=None,
           signature_def=None,
           input_names=None,
           output_names=None,
           tags=None,
           graph=None,
           config=None,
           use_ei=True,
           accelerator_id=<device ordinal number>)


output_dict = ei_predictor(feed_dict)
```

You can use EIPredictor in the following ways:

```
//EIPredictor class picks inputs and outputs from default serving signature def with tag
 “serve”. (similar to TF predictor)
ei_predictor = EIPredictor(model_dir)

//EI Predictor class picks inputs and outputs from the signature def picked using the
 signtaure_def_key (similar to TF predictor)
ei_predictor = EIPredictor(model_dir, signature_def_key='predict')

// Signature_def can be provided directly (similar to TF predictor)
ei_predictor = EIPredictor(model_dir, signature_def= sig_def)

// You provide the input_names and output_names dict.
// similar to TF predictor
```

```
ei_predictor = EIPredictor(model_dir,
                           input_names,
                           output_names)


// tag is used to get the correct signature def. (similar to TF predictor)

ei_predictor = EIPredictor(model_dir, tags='serve')
```

Additional EI Predictor functionality includes the following:

- Support for frozen models.

```
// For Frozen graphs, model_dir takes a file name , input_names and output_names
// input_names and output_names should be provided in this case.

ei_predictor = EIPredictor(model_dir,
                           input_names=None,
                           output_names=None )
```

- Ability to disable use of Elastic Inference by using the `use_ei` flag, which defaults to `True`. This is useful for testing EIPredictor against TensorFlow 2 Predictor.
- EIPredictor can also be created from a TensorFlow 2 Estimator. Given a trained Estimator, you can first export a SavedModel. See the SavedModel documentation for more details. The following shows example usage:

```
saved_model_dir = estimator.export_savedmodel(my_export_dir, serving_input_fn)
ei_predictor = EIPredictor(export_dir=saved_model_dir)

// Once the EIPredictor is created, inference is done using the following:
output_dict = ei_predictor(feed_dict)
```

Use Elastic Inference with TensorFlow 2 Predictor Example

Installing Elastic Inference TensorFlow 2

Elastic Inference enabled TensorFlow 2 comes bundled in the AWS Deep Learning AMI. You can also download the pip wheels for Python 2 and 3 from the Elastic Inference S3 bucket. Follow these instructions to download and install the pip package:

1. Choose the tar file for the Python version and operating system of your choice from the S3 bucket. Copy the path to the tar file and run the following command:

   ```
   curl -O [URL of the tar file of your choice]
   ```

2. To open the tar the file, run the following command:

   ```
   tar -xvzf [name of tar file]
   ```

3. Install the wheel using `pip` as shown in the following:

   ```
   pip install -U [name of untarred folder]/[name of tensorflow whl]
   ```

To serve different models, such as ResNet, using a Single Shot Detector (SSD), try the following example.

To serve and test inference with an SSD model

1. Download and unzip the model. If you already have the model, skip this step.

```
curl -O https://s3-us-west-2.amazonaws.com/aws-tf-serving-ei-example/ssd_resnet.zip
unzip ssd_resnet.zip -d /tmp
```

2. Download a picture of three dogs to your current directory.

```
curl -O https://raw.githubusercontent.com/awslabs/mxnet-model-server/master/docs/
images/3dogs.jpg
```

3. Use the built-in `EI Tool` to get the device ordinal number of all attached Elastic Inference accelerators. For more information on `EI Tool`, see Monitoring Elastic Inference Accelerators.

```
/opt/amazon/ei/ei_tools/bin/ei describe-accelerators --json
```

Your output should look like the following:

```
{
  "ei_client_version": "1.5.0",
  "time": "Fri Nov  1 03:09:38 2019",
  "attached_accelerators": 2,
  "devices": [
    {
      "ordinal": 0,
      "type": "eia1.xlarge",
      "id": "eia-679e4c622d584803aed5b42ab6a97706",
      "status": "healthy"
    },
    {
      "ordinal": 1,
      "type": "eia1.xlarge",
      "id": "eia-6c414c6ee37a4d93874afc00825c2f28",
      "status": "healthy"
    }
  ]
}
```

You use the device ordinal of your desired Elastic Inference accelerator to create a Predictor.

4. Open a text editor, such as vim, and paste the following inference script. Replace the `accelerator_id` value with the device ordinal of the desired Elastic Inference accelerator. This value must be an integer. Save the file as `ssd_resnet_predictor.py`.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import os
import sys
import numpy as np
import tensorflow as tf
import matplotlib.image as mpimg
from ei_for_tf.python.predictor.ei_predictor import EIPredictor

tf.compat.v1.app.flags.DEFINE_string('image', '', 'path to image in JPEG format')
FLAGS = tf.compat.v1.app.flags.FLAGS

coco_classes_txt = "https://raw.githubusercontent.com/amikelive/coco-labels/master/
coco-labels-paper.txt"
```

```
local_coco_classes_txt = "/tmp/coco-labels-paper.txt"
# it's a file like object and works just like a file
os.system("curl -o %s -O %s"%(local_coco_classes_txt, coco_classes_txt))
NUM_PREDICTIONS = 5
with open(local_coco_classes_txt) as f:
  classes = ["No Class"] + [line.strip() for line in f.readlines()]


def get_output(eia_predictor, test_input):
  pred = None
  for curpred in range(NUM_PREDICTIONS):
    pred = eia_predictor(test_input)

  num_detections = int(pred["num_detections"])
  print("%d detection[s]" % (num_detections))
  detection_classes = pred["detection_classes"][0][:num_detections]
  print([classes[int(i)] for i in detection_classes])


def main(_):

  img = mpimg.imread(FLAGS.image)
  img = np.expand_dims(img, axis=0)
  ssd_resnet_input = {'inputs': img}

  print('Running SSD Resnet on EIPredictor using specified input and outputs')
  eia_predictor = EIPredictor(
      model_dir='/tmp/ssd_resnet50_v1_coco/1/',
      input_names={"inputs": "image_tensor:0"},
      output_names={"detection_classes": "detection_classes:0", "num_detections":
 "num_detections:0",
                    "detection_boxes": "detection_boxes:0"},
      accelerator_id=0
  )
  get_output(eia_predictor, ssd_resnet_input)

  print('Running SSD Resnet on EIPredictor using default Signature Def')
  eia_predictor = EIPredictor(
      model_dir='/tmp/ssd_resnet50_v1_coco/1/',
  )
  get_output(eia_predictor, ssd_resnet_input)


if __name__ == "__main__":
  tf.compat.v1.app.run()
```

5. Run the inference script.

```
python ssd_resnet_predictor.py --image 3dogs.jpg
```

For more tutorials and examples, see the TensorFlow Python API.

Use Elastic Inference with the TensorFlow 2 Keras API

The Keras API has become an integral part of the machine learning development cycle because of its simplicity and ease of use. Keras enables rapid prototyping and development of machine learning constructs. Elastic Inference provides an API that offers native support for Keras. Using this API, you can directly use your Keras model, h5 file, and weights to instantiate a Keras-like Object. This object supports the native Keras prediction APIs, while fully utilizing Elastic Inference in the backend. Currently, `EIKerasModel` is only supported in Graph Mode. The following code sample shows the available parameters:

```
EIKerasModel(model,
             weights=None,
             export_dir=None,
             ):
    """Constructs an `EIKerasModel` instance.

    Args:
      model: A model object that either has its weights already set, or will be set with
 the weights argument.
             A model file that can be loaded
      weights (Optional): A weights object, or weights file that can be loaded, and will be
 set to the model object
      export_dir: A folder location to save your model as a SavedModelBundle

    Raises:
      RuntimeError: If eager execution is enabled.
    """
```

EIKerasModel can be used as follows:

```
#Loading from Keras Model Object
from ei_for_tf.python.keras.ei_keras import EIKerasModel
model = Model()
# Build Keras Model in the normal fashion
x = # input data
ei_model = EIKerasModel(model) # Only additional step to use EI
res = ei_model.predict(x)

#Loading from Keras h5 File
from ei_for_tf.python.keras.ei_keras import EIKerasModel
x = # input data
ei_model = EIKerasModel("keras_model.h5") # Only additional step to use EI
res = ei_model.predict(x)

#Loading from Keras h5 File and Weights file
from ei_for_tf.python.keras.ei_keras import EIKerasModel
x = # input data
ei_model = EIKerasModel("keras_model.json", weights="keras_weights.h5") # Only additional
 step to use EI
res = ei_model.predict(x)
```

Additionally, Elastic Inference enabled Keras includes Predict API Support as follows:

```
tf.keras
def predict(  x,
              batch_size=None,
              verbose=0,
              steps=None,
              max_queue_size=10, #Not supported
              workers=1, #Not Supported
              use_multiprocessing=False): #Not Supported


 Native Keras
 def predict( x,
              batch_size=None,
              verbose=0,
              steps=None,
              callbacks=None) # Not supported
```

TensorFlow 2 Keras API Example

In this example, you use a trained ResNet-50 model to classify an image of an African Elephant from ImageNet.

To test inference with a Keras model

1. Activate the Elastic Inference TensorFlow Conda Environment

```
source activate amazonei_tensorflow2_p27
```

2. Download an image of an African Elephant to your current directory.

```
curl -O https://upload.wikimedia.org/wikipedia/commons/5/59/
Serengeti_Elefantenbulle.jpg
```

3. Open a text editor, such as vim, and paste the following inference script. Save the file as `test_keras.py`.

```
# Resnet Example
from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictions
from ei_for_tf.python.keras.ei_keras import EIKerasModel
import numpy as np
import time
import os
import tensorflow as tf
tf.compat.v1.disable_eager_execution()


ITERATIONS = 20

model = ResNet50(weights='imagenet')
ei_model = EIKerasModel(model)
folder_name = os.path.dirname(os.path.abspath(__file__))
img_path = folder_name + '/Serengeti_Elefantenbulle.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
# Warm up both models
_ = model.predict(x)
_ = ei_model.predict(x)
# Benchmark both models
for each in range(ITERATIONS):
  start = time.time()
  preds = model.predict(x)
  print("Vanilla iteration %d took %f" % (each, time.time() - start))
for each in range(ITERATIONS):
  start = time.time()
  ei_preds = ei_model.predict(x)
  print("EI iteration %d took %f" % (each, time.time() - start))
# decode the results into a list of tuples (class, description, probability)
# (one such list for each sample in the batch)
print('Predicted:', decode_predictions(preds, top=3)[0])
print('EI Predicted:', decode_predictions(ei_preds, top=3)[0])
```

4. Run the inference script as follows:

```
python test_keras.py
```

5. Your output should be a list of predictions and their respective confidence score.

```
('Predicted:', [(u'n02504458', u'African_elephant', 0.9081173), (u'n01871265',
 u'tusker', 0.07836755), (u'n02504013', u'Indian_elephant', 0.011482777)])
('EI Predicted:', [(u'n02504458', u'African_elephant', 0.90811676), (u'n01871265',
 u'tusker', 0.07836751), (u'n02504013', u'Indian_elephant', 0.011482781)])
```

For more tutorials and examples, see the TensorFlow Python API.

Using MXNet Models with Elastic Inference

This release of Elastic Inference Apache MXNet has been tested to perform well and provide cost-saving benefits with the following deep learning use cases and network architectures (and similar variants).

Use Case	Example Network Topology
Image Recognition	Inception, ResNet, VGG, ResNext
Object Detection	SSD
Text to Speech	WaveNet

Topics

More Models and Resources

Here are some more pretrained models and examples to try with Elastic Inference.

1. MXNet Model Zoo - these Gluon models can be exported to the Symbol format and used with Elastic Inference.
2. Open Neural Network Exchange (ONNX) Models with MXNet - MXNet supports the ONNX model format, so you can use Elastic Inference with ONNX models that were exported from other frameworks.

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

MXNet Elastic Inference with Python

The Amazon Elastic Inference (Elastic Inference) enabled version of Apache MXNet lets you use Elastic Inference seamlessly, with few changes to your MXNet code. To use an existing MXNet inference script, make one change in the code. Wherever you set the context to bind your model, such as `mx.cpu()` or `mx.gpu()`, update this to use `mx.eia()` instead.

Topics

Elastic Inference Enabled Apache MXNet

For more information on MXNet set up, see Apache MXNet on AWS.

Preinstalled Elastic Inference Enabled MXNet

Elastic Inference enabled Apache MXNet is available in the AWS Deep Learning AMI.

Installing Elastic Inference Enabled MXNet

If you're not using a AWS Deep Learning AMI instance, a 'pip' package is available on Amazon S3 so you can build it in to your own Amazon Linux or Ubuntu AMIs using the following command:

```
pip install "latest-wheel"
```

Activate the MXNet Elastic Inference Environment

If you are using the AWS Deep Learning AMI, activate the Python 3 MXNet Elastic Inference environment or Python 2 MXNet Elastic Inference environment, depending on your version of Python.

For Python 3:

```
source activate amazonei_mxnet_p36
```

For Python 2:

```
source activate amazonei_mxnet_p27
```

If you are using a different AMI or a container, access the environment where MXNet is installed.

Validate MXNet for Elastic Inference Setup

Verify that you've properly set up your instance with Elastic Inference.

```
$ python ~/anaconda3/bin/EISetupValidator.py
```

If your instance is not properly set up with an accelerator, running any of the examples in this section will result in the following error:

```
Error: Failed to query accelerator metadata.
Failed to detect any accelerator
```

For detailed instructions on how to launch an AWS Deep Learning AMI with an Elastic Inference accelerator, see the Elastic Inference documentation.

Check MXNet for Elastic Inference Version

You can verify that MXNet is available to use and check the current version with the following code from the Python terminal:

```
>>> import mxnet as mx
>>> mx.__version__
'1.4.1'
```

This will return the version equivalent to the regular non-Elastic Inference version of MXNet available from GitHub

The commit hash number can be used to determine which release of the Elastic Inference-specific version of MXNet is installed using the following code:

```
import mxnet as mx
import os
path = os.path.join(mx.__path__[0],'COMMIT_HASH')
print(open(path).read())
```

You can then compare the commit hash with the Release Notes to find the specific info about the version you have.

Using Multiple Elastic Inference Accelerators with MXNet

You can run inference on MXNet when multiple Elastic Inference accelerators are attached to a single Amazon EC2 instance. The procedure for using multiple accelerators is the same as using multiple GPUs with MXNet.

Use the built-in `EI Tool` binary to get the device ordinal number of all attached Elastic Inference accelerators. For more information on `EI Tool`, see Monitoring Elastic Inference Accelerators.

```
/opt/amazon/ei/ei_tools/bin/ei describe-accelerators --json
```

Your output should look like the following:

```
{
  "ei_client_version": "1.5.0",
  "time": "Fri Nov  1 03:09:38 2019",
  "attached_accelerators": 2,
  "devices": [
    {
      "ordinal": 0,
      "type": "eia1.xlarge",
      "id": "eia-679e4c622d584803aed5b42ab6a97706",
      "status": "healthy"
    },
    {
      "ordinal": 1,
      "type": "eia1.xlarge",
      "id": "eia-6c414c6ee37a4d93874afc00825c2f28",
      "status": "healthy"
    }
  ]
}
```

Replace the device ordinal in the `mx.eia(<device ordinal>)` call with the device ordinal for your desired Elastic Inference accelerator as follows.

```
sym, arg_params, aux_params = mx.model.load_checkpoint('resnet-152', 0)

mod = mx.mod.Module(symbol=sym, context=mx.eia(<device ordinal>), label_names=None)
mod.bind(for_training=False, data_shapes=[('data', (1,3,224,224))],
    label_shapes=mod._label_shapes)
mod.set_params(arg_params, aux_params, allow_missing=True)

mod.forward(Batch([img]))
```

Use Elastic Inference with the MXNet Symbol API

Pass `mx.eia()` as the context in a call to either the `simple_bind()` or the `bind()` methods. For information, see MXNet Symbol API.

You use the `mx.eia()` context only with the bind call. The following example calls the `simple_bind()` method with the `mx.eia()` context:

```
import mxnet as mx

data = mx.sym.var('data', shape=(1,))
sym = mx.sym.exp(data)

# Pass mx.eia() as context during simple bind operation

executor = sym.simple_bind(ctx=mx.eia(), grad_req='null')
for i in range(10):

  # Forward call is performed on remote accelerator
  executor.forward(data=mx.nd.ones((1,)))
  print('Inference %d, output = %s' % (i, executor.outputs[0]))
```

The following example calls the `bind()` method:

```
import mxnet as mx
a = mx.sym.Variable('a')
b = mx.sym.Variable('b')
c = 2 * a + b
# Even for execution of inference workloads on eia,
# context for input ndarrays to be mx.cpu()
a_data = mx.nd.array([1,2], ctx=mx.cpu())
b_data = mx.nd.array([2,3], ctx=mx.cpu())
# Then in the bind call, use the mx.eia() context
e = c.bind(mx.eia(), {'a': a_data, 'b': b_data})

# Forward call is performed on remote accelerator
e.forward()
print('1st Inference, output = %s' % (e.outputs[0]))
# Subsequent calls can pass new data in a forward call
e.forward(a=mx.nd.ones((2,)), b=mx.nd.ones((2,)))
print('2nd Inference, output = %s' % (e.outputs[0]))
```

The following example calls the `bind()` method on a pre-trained real model (Resnet-50) from the Symbol API. Use your preferred text editor to create a script called mxnet_resnet50.py that has the following content. This script downloads the ResNet-50 model files (resnet-50-0000.params and resnet-50-symbol.json), list of labels(synset.txt) and an image of a cat. The cat image is used to get a prediction result from the pre-trained model. This result is looked up in the list of labels, returning a prediction result.

```
import mxnet as mx
```

```
import numpy as np

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

ctx = mx.eia()

with open('synset.txt', 'r') as f:
  labels = [l.rstrip() for l in f]

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

fname = mx.test_utils.download('https://github.com/dmlc/web-data/blob/master/mxnet/doc/
tutorials/python/predict_image/cat.jpg?raw=true')
img = mx.image.imread(fname)
# convert into format (batch, RGB, width, height)
img = mx.image.imresize(img, 224, 224) # resize
img = img.transpose((2, 0, 1)) # Channel first
img = img.expand_dims(axis=0) # batchify
img = img.astype(dtype='float32')
args['data'] = img

softmax = mx.nd.random_normal(shape=(1,))
args['softmax_label'] = softmax

exe = sym.bind(ctx=ctx, args=args, aux_states=aux, grad_req='null')

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

Then run the script, and you should see something similar to the following output. MXNet will optimize the model graph for Elastic Inference, load it on Elastic Inference accelerator, and then run inference against it:

```
(amazonei_mxnet_p36) ubuntu@ip-172-31-42-83:~$ python mxnet_resnet50.py
  [23:12:03] src/nnvm/legacy_json_util.cc:209: Loading symbol saved by previous version
 v0.8.0. Attempting to upgrade...
  [23:12:03] src/nnvm/legacy_json_util.cc:217: Symbol successfully upgraded!
  Using Amazon Elastic Inference Client Library Version: 1.2.8
  Number of Elastic Inference Accelerators Available: 1
  Elastic Inference Accelerator ID: eia-95ae5a472b2241769656dbb5d344a80e
  Elastic Inference Accelerator Type: eia2.large

  probability=0.418679, class=n02119789 kit fox, Vulpes macrotis
  probability=0.293495, class=n02119022 red fox, Vulpes vulpes
  probability=0.029321, class=n02120505 grey fox, gray fox, Urocyon cinereoargenteus
  probability=0.026230, class=n02124075 Egyptian cat
  probability=0.022557, class=n02085620 Chihuahua
```

Use Elastic Inference with the MXNet Module API

When you create the `Module` object, pass `mx.eia()` as the context. For more information, see Module API.

To use the MXNet Module API, you can use the following commands:

```
# Load saved model
sym, arg_params, aux_params = mx.model.load_checkpoint(model_path, EPOCH_NUM)

# Pass mx.eia() as context while creating Module object
mod = mx.mod.Module(symbol=sym, context=mx.eia())

# Only for_training = False is supported for eia
mod.bind(for_training=False, data_shapes=data_shape)
mod.set_params(arg_params, aux_params)

# forward call is performed on remote accelerator
mod.forward(data_batch)
```

The following script downloads two ResNet-152 model files (resnet-152-0000.params and resnet-152-symbol.json) and a labels list (synset.txt). It also downloads a cat image to get a prediction result from the pre-trained model, then looks this up in the result in labels list, returning a prediction result. Use your preferred text editor to create a script using the following content:

```
import mxnet as mx
import numpy as np
from collections import namedtuple

Batch = namedtuple('Batch', ['data'])


path='http://data.mxnet.io/models/imagenet/'
[mx.test_utils.download(path+'resnet/152-layers/resnet-152-0000.params'),
mx.test_utils.download(path+'resnet/152-layers/resnet-152-symbol.json'),
mx.test_utils.download(path+'synset.txt')]


ctx = mx.eia()


sym, arg_params, aux_params = mx.model.load_checkpoint('resnet-152', 0)
mod = mx.mod.Module(symbol=sym, context=ctx, label_names=None)
mod.bind(for_training=False, data_shapes=[('data', (1,3,224,224))],
     label_shapes=mod._label_shapes)
mod.set_params(arg_params, aux_params, allow_missing=True)

with open('synset.txt', 'r') as f:
  labels = [l.rstrip() for l in f]

fname = mx.test_utils.download('https://github.com/dmlc/web-data/blob/master/mxnet/doc/
tutorials/python/predict_image/cat.jpg?raw=true')
img = mx.image.imread(fname)


# convert into format (batch, RGB, width, height)
img = mx.image.imresize(img, 224, 224) # resize
img = img.transpose((2, 0, 1)) # Channel first
img = img.expand_dims(axis=0) # batchify


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

Save this script as test.py

Use Elastic Inference with the MXNet Gluon API

The Gluon API in MXNet provides a clear, concise, and easy-to-use API for building and training machine learning models. For more information, see the Gluon Documentation.

To use the MXNet Gluon API model for inference-only tasks, you can use the following commands:

Note
Both the model parameters and input array must be allocated with the Elastic Inference context.

```
import mxnet as mx
from mxnet.gluon import nn

def create():
    net = nn.HybridSequential()
    net.add(nn.Dense(2))
    return net

# get a simple Gluon nn model
net = create()
net.initialize(ctx=mx.cpu())

# copy model parameters to EIA context
net.collect_params().reset_ctx(mx.eia())

# hybridize the model with static alloc
net.hybridize(static_alloc=True, static_shape=True)

# allocate input array in EIA context and run inference
x = mx.nd.random.uniform(-1,1,(3,4),ctx=mx.eia())
predictions = net(x)
print(predictions)
```

You should be able to see the following output to confirm that you are using Elastic Inference:

```
Using Amazon Elastic Inference Client Library Version: xxxxxxxx
Number of Elastic Inference Accelerators Available: 1
Elastic Inference Accelerator ID: eia-xxxxxxxxxxxxxxxxxxxxxxxx
Elastic Inference Accelerator Type: xxxxxxxx
```

Loading parameters

There are a couple of different ways to load Gluon models. One way is to load model parameters from a file and specify the Elastic Inference context like the following:

```
# save the parameters to a file
net.save_parameters('params.gluon')

# create a new network using saved parameters
net2 = create()
net2.load_parameters('params.gluon', ctx=mx.eia())
net2.hybridize(static_alloc=True, static_shape=True)
predictions = net2(x)
print(predictions)
```

Loading Symbol and Parameters Files

You can also export the model's symbol and parameters to a file, then import the model as shown in the following:

```
# export both symbol and parameters to a file
net2.export('export')

# create a new network using exported network
net3 = nn.SymbolBlock.imports('export-symbol.json', ['data'],
    'export-0000.params', ctx=mx.eia())
net3.hybridize(static_alloc=True, static_shape=True)
predictions = net3(x)
```

If you have a model exported as symbol and parameter files, you can simply import those files and run inference.

```
import mxnet as mx
import numpy as np
from mxnet.gluon import nn

ctx = mx.eia()

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

with open('synset.txt', 'r') as f:
  labels = [l.rstrip() for l in f]

fname = mx.test_utils.download('https://github.com/dmlc/web-data/blob/master/mxnet/doc/
tutorials/python/predict_image/cat.jpg?raw=true')
img = mx.image.imread(fname) # convert into format (batch, RGB, width, height)
img = img.as_in_context(ctx) # image must be with EIA context
img = mx.image.imresize(img, 224, 224) # resize
img = img.transpose((2, 0, 1)) # channel first
img = img.expand_dims(axis=0) # batchify
img = img.astype(dtype='float32') # match data type

resnet50 = nn.SymbolBlock.imports('resnet-50-symbol.json',['data','softmax_label'],
    'resnet-50-0000.params',ctx=ctx) # import hybridized model symbols
label = mx.nd.array([0], ctx=ctx) # dummy softmax label in EIA context
resnet50.hybridize(static_alloc=True, static_shape=True)
prob = resnet50(img, label)
idx = prob.topk(k=5)[0]
for i in idx:
    i = int(i.asscalar())
    print('With prob = %.5f, it contains %s' % (prob[0,i].asscalar(), labels[i]))
```

Loading From Model Zoo

You can also use pre-trained models from Gluon model zoo as shown in the following:

> **Note**
> All pre-trained models expect inputs to be normalized in the same way as described in the model zoo documentation.

```
import mxnet as mx
import numpy as np
from mxnet.gluon.model_zoo import vision

ctx = mx.eia()

mx.test_utils.download('http://data.mxnet.io/models/imagenet/synset.txt')
with open('synset.txt', 'r') as f:
  labels = [l.rstrip() for l in f]
```

```
fname = mx.test_utils.download('https://github.com/dmlc/web-data/blob/master/mxnet/doc/
tutorials/python/predict_image/cat.jpg?raw=true')
img = mx.image.imread(fname) # convert into format (batch, RGB, width, height)
img = img.as_in_context(ctx) # image must be with EIA context
img = mx.image.imresize(img, 224, 224) # resize
img = mx.image.color_normalize(img.astype(dtype='float32')/255,
                               mean=mx.nd.array([0.485, 0.456, 0.406]),
                               std=mx.nd.array([0.229, 0.224, 0.225])) # normalize
img = img.transpose((2, 0, 1)) # channel first
img = img.expand_dims(axis=0) # batchify

resnet50 = vision.resnet50_v1(pretrained=True, ctx=ctx) # load model in EIA context
resnet50.hybridize(static_alloc=True, static_shape=True) # hybridize
prob = resnet50(img).softmax() # predict and normalize output
idx = prob.topk(k=5)[0] # get top 5 result
for i in idx:
    i = int(i.asscalar())
    print('With prob = %.5f, it contains %s' % (prob[0,i].asscalar(), labels[i]))
```

Troubleshooting

- MXNet Elastic Inference is built with MKL-DNN, so all operations using mx.cpu() are supported and will run with the same performance as the standard release. MXNet Elastic Inference does not support mx.gpu(), so all operations using that context will throw an error. Sample error message:

```
>>> mx.nd.ones((1),ctx=mx.gpu())
[20:35:32] src/imperative/./ imperative_utils.h:90: GPU support is disabled. Compile
 MXNet with USE_CUDA=1 to enable GPU support.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/ubuntu/deps/MXNetECL /python/mxnet/ndarray/ndarray.py", line 2421, in ones
    return _internal._ones(shape=shape, ctx=ctx, dtype=dtype, **kwargs)
  File "<string>", line 34, in _ones
  File "/home/ubuntu/deps/MXNetECL /python/mxnet/_ctypes/ndarray.py", line 92, in
 _imperative_invoke
    ctypes.byref(out_stypes)))
  File "/home/ubuntu/deps/MXNetECL /python/mxnet/base.py", line 252, in check_call
    raise MXNetError(py_str(_LIB.MXGetLastError()))
mxnet.base.MXNetError: [20:35:32] src/imperative/imperative.cc:79: Operator _ones is not
 implemented for GPU.
```

- Elastic Inference is only for production inference use cases and does not support any model training. When you use either the Symbol API or the Module API, do not call the `backward()` method or call `bind()` with `for_training=True`. This throws an error. Because the default value of `for_training` is `True`, make sure you set `for_training=False` manually in cases such as the example in Use Elastic Inference with the MXNet Module API (p. 34). Sample error using test.py:

```
Traceback (most recent call last):
  File "test.py", line 16, in <module>
    label_shapes=mod._label_shapes)
  File "/home/ec2-user/.local/lib/python3.6/site-packages/mxnet/module/module.py", line
 402, in bind
    raise ValueError("for training cannot be set to true with EIA context")
ValueError: for training cannot be set to true with EIA context
```

- For Gluon, do not call training-specific functions or you will receive the following error:

```
Traceback (most recent call last):
  File "train_gluon.py", line 44, in <module>
    output = net(data)
```

```
  File "/usr/local/lib/python2.7/dist-packages/mxnet/gluon/block.py", line 540, in
 __call__
    out = self.forward(*args)
  File "train_gluon.py", line 24, in forward
    x = self.pool1(F.relu(self.conv1(x)))
  File "/usr/local/lib/python2.7/dist-packages/mxnet/gluon/block.py", line 540, in
 __call__
    out = self.forward(*args)
  File "/usr/local/lib/python2.7/dist-packages/mxnet/gluon/block.py", line 909, in
 forward
    return self._call_cached_op(x, *args)
  File "/usr/local/lib/python2.7/dist-packages/mxnet/gluon/block.py", line 815, in
 _call_cached_op
    out = self._cached_op(*cargs)
  File "/usr/local/lib/python2.7/dist-packages/mxnet/_ctypes/ndarray.py", line 150, in
 __call__
    ctypes.byref(out_stypes)))
  File "/usr/local/lib/python2.7/dist-packages/mxnet/base.py", line 252, in check_call
    raise MXNetError(py_str(_LIB.MXGetLastError()))
mxnet.base.MXNetError: [23:10:21] /home/ubuntu/deps/MXNetECL/3rdparty/tvm/nnvm/include/
nnvm/graph.h:230: Check failed: it != attrs.end() Cannot find attribute full_ref_count in
 the graph
```

- Because training is not allowed, there is no point of initializing an optimizer for inference.
- A model trained on an earlier version of MXNet will work on a later version of MXNet Elastic Inference because it is backwards compatible (e.g. train model on MXNet 1.3 and run on MXNet Elastic Inference 1.4). However, you may run into undefined behavior if you train on a later version of MXNet (e.g. train model on MXNet Master and run on MXNet EI 1.4)
- Different sizes of Elastic Inference accelerators have different amounts of GPU memory. If your model requires more GPU memory than is available in your accelerator, you get a message that looks like the log below. If you run into this message, you should use a larger accelerator size with more memory. Stop and restart your instance with a larger accelerator.

```
mxnet.base.MXNetError: [06:16:17] src/operator/subgraph/eia/eia_subgraph_op.cc:206: Last
 Error:
    EI Error Code: [51, 8, 31]
    EI Error Description: Accelerator out of memory. Consider using a larger accelerator.
    EI Request ID: MX-A19B0DE6-7999-4580-8C49-8EA 7ADSFFCB  --  EI Accelerator ID: eia-
cb0aasdfdfsdf2a acab7
    EI Client Version: 1.2.12
```

- For Gluon, remember that both the model and input array (image) must be allocated in the Elastic Inference context. If either the model parameters or an input are allocated in a different context, you will see one of the following errors:

```
MXNetError: [21:59:27] src/imperative/cached_op.cc:866:
    Check failed: inputs[i]->ctx() == default_ctx (eia(0) vs. cpu(0))
    CachedOp requires all inputs to live on the same context.
    But data is on cpu(0) while resnetv10_conv0_weight is on eia(0)
```

```
RuntimeError: Parameter 'resnetv10_conv0_weight' was not
    initialized on context eia(0). It was only initialized on [cpu(0)].
```

- For Gluon, make sure you hybridize the model and pass the `static_alloc=True` and `static_shape=True` options. Otherwise, MXNet will run inference in imperative mode on CPU and won't invoke any Elastic Inference functionality. In this case, you won't see Elastic Inference info messages, and may see MKLDNN info instead like the following:

```
[21:40:20] src/operator/nn/mkldnn/mkldnn_base.cc:74: Allocate 147456 bytes with malloc
 directly
```

```
[21:40:20] src/operator/nn/mkldnn/mkldnn_base.cc:74: Allocate 3211264 bytes with malloc
 directly
[21:40:20] src/operator/nn/mkldnn/mkldnn_base.cc:74: Allocate 9437184 bytes with malloc
 directly
```

- When you are using Symbol/Module API, you should always allocate arrays in the CPU context and bind with the Elastic Inference context. If you allocate arrays in the Elastic Inference context, you will see the following error when you try to bind the model:

```
Traceback (most recent call last):
    File "symbol.py", line 43, in <module>
      exe = sym.bind(ctx=ctx, args=args, aux_states=aux, grad_req='null')
    File "/home/ubuntu/.local/lib/python2.7/site-packages/mxnet/symbol/symbol.py", line
 1706, in bind
      ctypes.byref(handle)))
    File "/home/ubuntu/.local/lib/python2.7/site-packages/mxnet/base.py", line 252, in
 check_call
      raise MXNetError(py_str(_LIB.MXGetLastError()))
  mxnet.base.MXNetError: [00:05:25] src/executor/../common/exec_utils.h:516: Check
 failed: x == default_ctx Input array is in eia(0) while binding with ctx=cpu(0).
  All arguments must be in global context (cpu(0)) unless group2ctx is specified for
 cross-device graph.
```

- Calling `reshape` explicitly by using either the Module or the Symbol API, or implicitly using different shapes for input `NDArrays` in different forward passes can lead to OOM errors. Before being reshaped, the model is not cleaned up on the accelerator until the session is destroyed. In Gluon, inferring with inputs of differing shapes will result in the model re-allocating memory. For Elastic Inference, this means that the model will be re-loaded on the accelerator leading to performance degradation and potential OOM errors. MXNet does not support the reshape operation for the EIA context. Using different input data sizes or batch sizes is not supported and may result in the following error. You can either pad your data so all shapes are the same or bind the model with different shapes to use multiple executors. The latter option may result in out-of-memory errors because the model is duplicated on the accelerator.

```
mxnet.base.MXNetError: [17:06:11] src/operator/subgraph/eia/eia_subgraph_op.cc:224: Last
 Error:
     EI Error Code: [52, 3, 32]
     EI Error Description: Invalid tensor on accelerator
     EI Request ID: MX-96534015-D443-4EC2-B184-ABBBDB1B150E  --  EI Accelerator ID: eia-
a9957ab65c5f44de975944a641c86b03
     EI Client Version: 1.3.1
```

MXNet Elastic Inference with Java

Starting from Apache MXNet version 1.4, the Java API can now integrate with Amazon Elastic Inference. You can use Elastic Inference with the following MXNet Java API operations:

- MXNet Java Infer API

Topics

Install Amazon EI Enabled Apache MXNet

Amazon Elastic Inference enabled Apache MXNet is available in the AWS Deep Learning AMI. A maven repository is also available on Amazon S3. You can build this repository into your own Amazon Linux or Ubuntu AMIs, or Docker containers.

For Maven projects, Elastic Inference Java can be included by adding the following to your project's `pom.xml`:

```
<repositories>
    <repository>
      <id>Amazon Elastic Inference</id>
      <url>https://s3.amazonaws.com/amazonei-apachemxnet/scala</url>
    </repository>
</repositories>
```

In addition, add the Elastic Inference flavor of MXNet as a dependency using:

```
        <dependency>
            <groupId>com.amazonaws.ml.mxnet</groupId>
            <artifactId>mxnet-full_2.11-linux-x86_64-eia</artifactId>
            <version>[1.4.0,)</version>
        </dependency>
```

Check MXNet for Java Version

You can use the commit hash number to determine which release of the Java-specific version of MXNet is installed using the following code:

```
// Imports
import org.apache.mxnet.javaapi.*;

// Lines to run
Version$ version$ = Version$.MODULE$;
System.out.println(version$.getCommitHash());
```

You can then compare the commit hash with the Release Notes to find the specific info about the version you have.

Use Amazon Elastic Inference with the MXNet Java Infer API

To use Amazon Elastic Inference with the MXNet Java Infer API, pass `Context.eia()` as the context when creating the Infer Predictor object. See the MXNet Infer Reference for more information. The following example uses the pre-trained real model (Resnet-152):

```
package mxnet;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.util.Arrays;
import java.util.Comparator;
```

```
import java.util.List;
import java.util.stream.IntStream;
import org.apache.commons.io.FileUtils;
import org.apache.mxnet.infer.javaapi.ObjectDetector;
import org.apache.mxnet.infer.javaapi.Predictor;
import org.apache.mxnet.javaapi.*;

public class Example {
    public static void main(String[] args) throws IOException {
        String urlPath = "http://data.mxnet.io/models/imagenet";
        String filePath = System.getProperty("java.io.tmpdir");

        // Download Model and Image
        FileUtils.copyURLToFile(new URL(urlPath + "/resnet/152-layers/
resnet-152-0000.params"),
                new File(filePath + "resnet-152/resnet-152-0000.params"));
        FileUtils.copyURLToFile(new URL(urlPath + "/resnet/152-layers/resnet-152-
symbol.json"),
                new File(filePath + "resnet-152/resnet-152-symbol.json"));
        FileUtils.copyURLToFile(new URL(urlPath + "/synset.txt"),
                new File(filePath + "resnet-152/synset.txt"));
        FileUtils.copyURLToFile(new URL("https://github.com/dmlc/web-data/blob/master/
mxnet/doc/tutorials/python/predict_image/cat.jpg?raw=true"),
                new File(filePath + "cat.jpg"));

        List<Context> contexts = Arrays.asList(Context.eia());
        Shape inputShape = new Shape(new int[]{1, 3, 224, 224});
        List<DataDesc> inputDesc = Arrays.asList(new DataDesc("data", inputShape,
 DType.Float32(), "NCHW"));
        Predictor predictor = new Predictor(filePath + "resnet-152/resnet-152", inputDesc,
 contexts, 0);

        BufferedImage originalImg = ObjectDetector.loadImageFromFile(filePath + "cat.jpg");
        BufferedImage resizedImg = ObjectDetector.reshapeImage(originalImg, 224, 224);
        NDArray img = ObjectDetector.bufferedImageToPixels(resizedImg, new Shape(new int[]
{1, 3, 224, 224}));

        List<NDArray> predictResults = predictor.predictWithNDArray(Arrays.asList(img));
        float[] results = predictResults.get(0).toArray();

        List<String> synsetLines = FileUtils.readLines(new File(filePath + "resnet-152/
synset.txt"));

        int[] best = IntStream.range(0, results.length)
                .boxed().sorted(Comparator.comparing(i -> -results[i]))
                .mapToInt(ele -> ele).toArray();

        for (int i = 0; i < 5; i++) {
            int ind = best[i];
            System.out.println(i + ": " + synsetLines.get(ind) + " - " + best[ind]);
        }

    }
}
```

More Models and Resources

For more tutorials and examples, see:

- the framework's official Java documentation
- Java API Reference
- Apache MXNet website

Troubleshooting

- MXNet EI is built with MKL-DNN. All operations using Context.cpu() are supported and will run with the same performance as the standard release. MXNet EI does not support Context.gpu(). All operations using that context will throw an error.
- You cannot allocate memory for NDArray on the remote accelerator by writing something like this:

```
x = NDArray.array(Array(1,2,3),
            ctx=Context.eia())
```

  This throws an error. Instead you should use `Context.cpu()`. Look at the previous `bind()` example to see how MXNet automatically transfers your data to the accelerator as necessary. Sample error message:
- Elastic Inference is only for production inference use cases and does not support any model training. When you use either the Symbol API or the Module API, do not call the `backward()` method or call `bind()` with `forTraining=True`. This throws an error. Because the default value of `forTraining` is `True`, make sure you set `for_training=False` manually in cases such as the example in Use Elastic Inference with the MXNet Module API (p. 34). Sample error using test.py:
- Because training is not allowed, there is no point of initializing an optimizer for inference.
- A model trained on an earlier version of MXNet will work on a later version of MXNet EI because it is backwards compatible. For example, you can train a model on MXNet 1.3 and run it on MXNet EI 1.4. However, you may run into undefined behavior if you train on a later version of MXNet. For example, training a model on MXNet Master and running on MXNet EI 1.4.
- Different sizes of EI accelerators have different amounts of GPU memory. If your model requires more GPU memory than is available in your accelerator, you get a message that looks like the log below. If you run into this message, you should use a larger accelerator size with more memory. Stop and restart your instance with a larger accelerator.
- Calling `reshape` explicitly by using either the Module or the Symbol API can lead to OOM errors. Implicitly using different shapes for input `NDArrays` in different forward passes can also lead to OOM errors. Before being reshaped, the model is not cleaned up on the accelerator until the session is destroyed.

MXNet Elastic Inference with Scala

Starting from Apache MXNet version 1.4, the Scala API can now integrate with Amazon Elastic Inference. You can use Elastic Inference with the following MXNet Scala API operations:

- MXNet Scala Symbol API
- MXNet Scala Module API
- MXNet Scala Infer API

Topics

Install Elastic Inference Enabled Apache MXNet

Elastic Inference enabled Apache MXNet is available in the AWS Deep Learning AMI. A maven repository is also available on Amazon S3. You can build it in to your own Amazon Linux or Ubuntu AMIs, or Docker containers.

For Maven projects, Elastic Inference with Scala can be included by adding the following to your project's `pom.xml`:

```
<repositories>
    <repository>
      <id>Amazon Elastic Inference</id>
      <url>https://s3.amazonaws.com/amazonei-apachemxnet/scala</url>
    </repository>
</repositories>
```

In addition, add the Elastic Inference flavor of MXNet as a dependency using:

```
        <dependency>
            <groupId>com.amazonaws.ml.mxnet</groupId>
            <artifactId>mxnet-full_2.11-linux-x86_64-eia</artifactId>
            <version>[1.4.0,)</version>
        </dependency>
```

Check MXNet for Scala Version

You can use the commit hash number to determine which release of the Scala-specific version of MXNet is installed using the following code:

```
// Imports
import org.apache.mxnet.util.Version

// Line to run
println(Version.getCommitHash)
```

You can then compare the commit hash with the Release Notes to find the specific info about the version you have.

Use Amazon Elastic Inference with the MXNet Symbol API

To use Elastic Inference with the MXNet Symbol API, pass `Context.eia()` as the context in a call to either the `Symbol.bind` or `Symbol.simpleBind` methods. See the MXNet Symbol Reference for more information.

The following is an example using `Context.eia()` in a call to simpleBind:

```
import org.apache.mxnet._

object Example {

  def main(args: Array[String]): Unit = {
    val data = Symbol.Variable("data", shape=Shape(1))
    val sym = Symbol.api.exp(Some(data))
```

```
    // Pass mx.eia() as context during simple bind operation
    val executor = sym.simpleBind(Context.eia(), gradReq = "null", shapeDict = Map("data" -
> Shape(1)))
    for( i  <- 1 to 10) {
      executor.forward(false, ("data", NDArray.ones(1)))
      println(s"Inference ${i}, output = ${executor.outputs.head}")
    }
  }
}
```

Note, the GPU context is not supported. All values and computations that are not Elastic Inference should use the CPU context. Use the Elastic Inference context only with the bind call.

The following is an example using bind. Note, you cannot use the Elastic Inference context to allocate memory or it will throw an error.

```
import org.apache.mxnet._

object Example {

  def main(args: Array[String]): Unit = {
    val a = Symbol.Variable("a")
    val b = Symbol.Variable("b")
    val c = a + b

    // Even for EIA workloads, declare NDArrays on the CPU
    val aData = NDArray.array(Array(1f,2f), Shape(2), Context.cpu())
    val bData = NDArray.array(Array(2f,3f), Shape(2), Context.cpu())

    // Then in the bind call, use Context.eia()
    val executor = c.bind(Context.eia(), Map("a" -> aData, "b" -> bData))

    // The forward call is performed on the remote accelerator
    executor.forward()
    println(s"1st Inference, output = ${executor.outputs.head}")

    // Subsequent calls can pass new data in a forward call
    executor.forward(false, ("a", NDArray.ones((2))), ("b", NDArray.ones((2))))
    println(s"2nd Inference, output = ${executor.outputs.head}")
  }
}
```

Use Amazon Elastic Inference with the MXNet Module API

To use Elastic Inference with the MXNet Module API, pass `Context.eia()` as the context when creating the `Module` object. See the MXNet Module Reference for more information.

The following is an example using Elastic Inference with the Module API on a pre-trained real model (Resnet-152).

```
import java.io.File
import java.net.URL

import org.apache.commons.io.FileUtils
import org.apache.mxnet._
import org.apache.mxnet.infer.ImageClassifier
import org.apache.mxnet.module.Module
```

```
import scala.io.Source

object Example {

  def main(args: Array[String]): Unit = {
    val urlPath = "http://data.mxnet.io/models/imagenet"
    val filePath = System.getProperty("java.io.tmpdir")

    // Download Model and Image
    FileUtils.copyURLToFile(new URL(s"${urlPath}/resnet/152-layers/
resnet-152-0000.params"),
      new File(s"${filePath}resnet-152/resnet-152-0000.params"))
    FileUtils.copyURLToFile(new URL(s"${urlPath}/resnet/152-layers/resnet-152-
symbol.json"),
      new File(s"${filePath}resnet-152/resnet-152-symbol.json"))
    FileUtils.copyURLToFile(new URL(s"${urlPath}/synset.txt"),
      new File(s"${filePath}resnet-152/synset.txt"))
    FileUtils.copyURLToFile(new URL("https://github.com/dmlc/web-data/blob/master/mxnet/
doc/tutorials/python/predict_image/cat.jpg?raw=true"),
      new File(s"${filePath}cat.jpg"))

    // Load model
    val (symbol, argParams, auxParams) = Model.loadCheckpoint(s"${filePath}resnet-152/
resnet-152", 0)
    val mod = new Module(symbol, contexts = Context.eia(), labelNames = IndexedSeq())
    mod.bind(dataShapes=IndexedSeq(DataDesc("data", Shape(1, 3, 224, 224))), forTraining =
 false)
    mod.setParams(argParams, auxParams, allowMissing = true)
    val labels = Source.fromFile(s"${filePath}resnet-152/
synset.txt").getLines().map(_.trim).toIndexedSeq

    // Load image
    val originalImg = ImageClassifier.loadImageFromFile(s"${filePath}cat.jpg")
    val resizedImg = ImageClassifier.reshapeImage(originalImg, 224, 224)
    val img = ImageClassifier.bufferedImageToPixels(resizedImg, Shape(1, 3, 224, 224))

    mod.forward(new DataBatch(IndexedSeq(img), IndexedSeq(), IndexedSeq(), 0))

    val probabilities = mod.getOutputs().head.head.toArray
    val best = probabilities.zipWithIndex.sortBy(-_._1).take(5)
    best.zipWithIndex.foreach {
      case ((prob, nameIndex), i) => println(s"Option ${i}: ${labels(nameIndex)} -
 ${prob}")
    }
  }
}
```

Use Amazon Elastic Inference with the MXNet Infer API

To use Elastic Inference with the MXNet Infer API, pass `Context.eia()` as the context when creating the Infer Predictor object. See the MXNet Infer Reference for more information. The following example also uses the pre-trained real model (Resnet-152).

```
import java.io.File
import java.net.URL

import org.apache.commons.io.FileUtils
import org.apache.mxnet._
import org.apache.mxnet.infer.ImageClassifier

object Example {

  def main(args: Array[String]): Unit = {
```

```
    val urlPath = "http://data.mxnet.io/models/imagenet"
    val filePath = System.getProperty("java.io.tmpdir")

    // Download Model and Image
    FileUtils.copyURLToFile(new URL(s"${urlPath}/resnet/152-layers/
resnet-152-0000.params"),
      new File(s"${filePath}resnet-152/resnet-152-0000.params"))
    FileUtils.copyURLToFile(new URL(s"${urlPath}/resnet/152-layers/resnet-152-
symbol.json"),
      new File(s"${filePath}resnet-152/resnet-152-symbol.json"))
    FileUtils.copyURLToFile(new URL(s"${urlPath}/synset.txt"),
      new File(s"${filePath}resnet-152/synset.txt"))
    FileUtils.copyURLToFile(new URL("https://github.com/dmlc/web-data/blob/master/mxnet/
doc/tutorials/python/predict_image/cat.jpg?raw=true"),
      new File(s"${filePath}cat.jpg"))

    val inputShape = Shape(1, 3, 224, 224)
    val inputDesc = IndexedSeq(DataDesc("data", inputShape, DType.Float32, "NCHW"))
    val imgClassifier = new ImageClassifier(s"${filePath}resnet-152/resnet-152", inputDesc,
 Context.eia())

    val img = ImageClassifier.loadImageFromFile(s"${filePath}cat.jpg")
    val topK = 5
    val output = imgClassifier.classifyImage(img, Some(topK)).head

    output.zipWithIndex.foreach{
      case ((name, prob), i) => println(s"Option ${i}: ${name} - ${prob}")
    }
  }
}
```

More Models and Resources

For more tutorials and examples, see:

- The framework's official Scala documentation
- Scala API Reference
- Apache MXNet website

Troubleshooting

- MXNet Elastic Inference is built with MKL-DNN, so all operations using Context.cpu() are supported and runs with the same performance as the standard release. MXNet Elastic Inference does not support Context.gpu(), so all operations using that context will throw an error.
- You cannot allocate memory for NDArray on the remote accelerator by writing something like the following.

  ```
  x = NDArray.array(Array(1,2,3),
            ctx=Context.eia())
  ```

  This throws an error. Instead you should use `Context.cpu()`. Look at the previous `bind()` example to see how MXNet automatically transfers your data to the accelerator as necessary. Sample error message:
- Amazon Elastic Inference is only for production inference use cases and does not support any model training. When you use either the Symbol API or the Module API, do not call the `backward()` method or call `bind()` with `forTraining=True`. This throws an error. Because the default value of `forTraining` is `True`, make sure you set `for_training=False` manually in cases such as the example in Use Elastic Inference with the MXNet Module API (p. 34). Sample error using test.py:

- Because training is not allowed, there is no point tp initializing an optimizer for inference.
- A model trained on an earlier version of MXNet will work on a later version of MXNet EI because it is backwards compatible. For example, you can train a model on MXNet 1.3 and run it on MXNet EI 1.4. However, you may run into undefined behavior if you train on a later version of MXNet. For example training a model on MXNet Master and running it on MXNet Elastic Inference 1.4.
- Different sizes of Elastic Inference accelerators have different amounts of GPU memory. If your model requires more GPU memory than is available in your accelerator, you get a message that looks like the log below. If you run into this message, you should use a larger accelerator size with more memory. Stop and restart your instance with a larger accelerator.
- Calling `reshape` explicitly by using either the Module or the Symbol API can lead to OOM errors. Implicitly using different shapes for input `NDArrays` in different forward passes can also lead to OOM errors. Before being reshaped, the model is not cleaned up on the accelerator until the session is destroyed.

Using PyTorch Models with Elastic Inference

This release of Elastic Inference enabled PyTorch has been tested to perform well and provide cost-saving benefits with the following deep learning use cases and network architectures (and similar variants).

> **Note**
> Elastic Inference enabled PyTorch is only available with Amazon Deep Learning Containers v27 and later.

Use Case	Example Network Topology
Image Recognition	Inception, ResNet, VGG
Semantic Segmentation	UNet
Text Embeddings	BERT
Transformers	GPT

Topics

Compile Elastic Inference-enabled PyTorch models

Elastic Inference-enabled PyTorch only supports TorchScript compiled models. You can compile a PyTorch model into TorchScript using either tracing or scripting. Both produce a computation graph, but differ in how they do so.

Scripting a model is the preferred way of compiling to TorchScript because it preserves all model logic. However, the set of models that can be scripted is smaller than the set of traceable models. Your model might be traceable, but not scriptable, or not traceable at all. You may need to modify your model code to make it TorchScript compatible.

Because of the way that Elastic Inference handles control-flow operations in PyTorch 1.3.1, inference latency might be noticeable for scripted models that contain many conditional branches. Try both

tracing and scripting to see how your model performs with Elastic Inference. With PyTorch 1.3.1, it is likely that a traced model performs better than its scripted version.

Scripting

Scripting performs direct analysis of the source code to construct a computation graph and preserve control flow.

The following example code shows how to compile a model using scripting. It uses the TorchVision pretrained weights for ResNet18. The resulting scripted model can still be saved to a file, then loaded with `torch.jit.load` using Elastic Inference-enabled PyTorch.

```
import torchvision, torch
# ImageNet pretrained models take inputs of this size.
x = torch.rand(1,3,224,224)
# Call eval() to set model to inference mode
model = torchvision.models.resnet18(pretrained=True).eval()
scripted_model = torch.jit.script(model)
```

Tracing

Tracing takes a sample input and records the operations performed when executing the model on that particular input. This means that control flow may be erased because the graph is compiled by tracing the code with just one input. For example, a model definition might have code to pad images of a particular size x. If the model is traced with an image of a different size y, then future inputs of size x fed to the traced model will not be padded. This happens because the code path was never run while tracing with the sample input.

The following example shows how to compile a model using tracing. It uses the TorchVision pretrained weights for ResNet18. The `torch.jit.optimized_execution` context block is required to use traced models with Elastic Inference. This function is only available through the Elastic Inference enabled PyTorch framework.

If you are tracing your model with the basic PyTorch framework, don't include the `torch.jit.optimized_execution` context. The resulting traced model can still be saved to a file, then loaded with `torch.jit.load` using Elastic Inference-enabled PyTorch.

```
import torchvision, torch
# ImageNet pretrained models take inputs of this size.
x = torch.rand(1,3,224,224)
# Call eval() to set model to inference mode
model = torchvision.models.resnet18(pretrained=True).eval()

# Required when using Elastic Inference
with torch.jit.optimized_execution(True, {'target_device': 'eia:0'}):
    traced_model = torch.jit.trace(model, x)
```

Saving and loading a compiled model

The output of tracing and scripting is a ScriptModule, the TorchScript version of the basic PyTorch nn.Module. Serializing and de-serializing a TorchScript module is as easy as calling torch.jit.save() and torch.jit.load() respectively. This is the JIT version of saving and loading a basic PyTorch model using `torch.save()` and `torch.load()`.

```
torch.jit.save(traced_model, 'resnet18_traced.pt')
torch.jit.save(scripted_model, 'resnet18_scripted.pt')
```

```
traced_model = torch.jit.load('resnet18_traced.pt')
scripted_model = torch.jit.load('resnet18_scripted.pt')
```

Saved TorchScript models are not bound to specific classes and code directories, unlike basic PyTorch models. You can directly load saved TorchScript models without instantiating the model class first.

CPU training requirement

PyTorch does not save models in a device-agnostic way. Model training frequently happens in a CUDA context on a GPU. However, the Elastic Inference enabled PyTorch framework is CPU-only on the client side, even though your model runs in a CUDA context on the server.

Tracing models may lead to tensor creation on a specific device. When this happens, you may get errors when loading the model onto a different device. To avoid device-related errors, load your model by explicitly specifying the CPU device using `torch.jit.load(model, map_location=torch.device('cpu'))`. This forces all model tensors to CPU. If you still get an error, cast your model to CPU before saving it. This can be done on any instance type, including GPU instances. For more information, see TorchScript's Frequently Asked Questions.

Additional Requirements and Considerations

Framework Paradigms: Dynamic versus Static Computational Graphs

All deep learning frameworks view models as directed acyclic graphs. However, the frameworks differ in how they allow you to specify models. TensorFlow and MXNet use static computation graphs, meaning that the computation graph must be defined and built before it's run. In contrast, PyTorch uses dynamic computational graphs. This means that models are imperatively specified by using idiomatic Python code, and then the computation graph is built at execution time. Rather than being predetermined, the graph's structure can change during execution.

Productionizing PyTorch with TorchScript

TorchScript addresses the limitations of the computation graph being built at execution time with JIT. JIT is a just-in-time compiler that compiles and exports models to a Python-free representation. By converting PyTorch models into TorchScript, users can run their models in any production environment. JIT also performs graph-level optimizations, providing a performance boost over basic PyTorch.

To use Elastic Inference enabled PyTorch, you must convert your models to the TorchScript format.

Model Format

Basic PyTorch uses dynamic computational graphs. This means that models are specified with idiomatic Python code and the computation graph is built at execution time. Elastic Inference supports TorchScript saved models. TorchScript uses Torch.JIT, a just-in-time compiler, to produce models that can be serialized and optimized from PyTorch code. These models can be run anywhere, including environments without Python. Torch.JIT offers two ways to compile a PyTorch model: tracing and scripting. Both produce a computation graph, but differ in how they do so. For more information on compiling using Torch.JIT, see Compile Elastic Inference-enabled PyTorch models (p. 48). For more information about running inference using TorchScript, see Use Elastic Inference with PyTorch for inference (p. 51).

Additional Resources

For more information about using TorchScript, see the TorchScript tutorial.

The following pretrained PyTorch models can be used with Elastic Inference:

- TorchVision
- Torch.Hub

PyTorch Elastic Inference with Python

The Amazon Elastic Inference enabled version of PyTorch lets you use Elastic Inference seamlessly, with few changes to your PyTorch code. The following tutorial shows how to perform inference using an Elastic Inference accelerator.

> **Note**
> Elastic Inference enabled PyTorch is only available with Amazon Deep Learning Containers version 27 and later.

Topics

Install Elastic Inference Enabled PyTorch

Preinstalled Elastic Inference Enabled PyTorch

The Elastic Inference enabled packages are available in the AWS Deep Learning AMI. You also have Docker container options through the Amazon Deep Learning Containers.

Installing Elastic Inference Enabled PyTorch

If you're not using a AWS Deep Learning AMI instance, you can download the packages from the Amazon S3 bucketto build it into your own Amazon Linux or Ubuntu AMIs.

Activate the PyTorch Elastic Inference Environment

If you are using the AWS Deep Learning AMI, activate the Python 3 Elastic Inference enabled PyTorch environment. Python 2 is not supported for Elastic Inference enabled PyTorch.

For Python 3, run the following to activate the environment:

```
source activate amazonei_pytorch_p36
```

If you are using a different AMI or a container, access the environment where PyTorch is installed.

The remaining parts of this guide assume you are using the `amazonei_pytorch_p36` environment. If you are switching from MXNet or TensorFlow Elastic Inference environments, you must stop and then start your instance in order to reattach the Elastic Inference accelerator. Rebooting is not sufficient since the process of switching frameworks requires a complete shut down.

Use Elastic Inference with PyTorch for inference

With Elastic Inference enabled PyTorch, the inference API is largely unchanged. However, you must use the `with torch.jit.optimized_execution()` context to trace or script your models into TorchScript, then perform inference.

Run Inference with a ResNet-50 Model

To run inference using Elastic Inference enabled PyTorch, do the following.

1. Download a picture of a cat to your current directory.

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

2. Download a list of ImageNet class mappings to your current directory.

```
wget https://aws-dlc-sample-models.s3.amazonaws.com/pytorch/imagenet_classes.txt
```

3. Use the built-in `EI Tool` to get the device ordinal number of all attached Elastic Inference accelerators. For more information on `EI Tool`, see Monitoring Elastic Inference Accelerators.

```
/opt/amazon/ei/ei_tools/bin/ei describe-accelerators --json
```

Your output should look like the following:

```
{
  "ei_client_version": "1.5.0",
  "time": "Fri Nov  1 03:09:38 2019",
  "attached_accelerators": 2,
  "devices": [
    {
      "ordinal": 0,
      "type": "eia1.xlarge",
      "id": "eia-679e4c622d584803aed5b42ab6a97706",
      "status": "healthy"
    },
    {
      "ordinal": 1,
      "type": "eia1.xlarge",
      "id": "eia-6c414c6ee37a4d93874afc00825c2f28",
      "status": "healthy"
    }
  ]
}
```

You use the device ordinal of your desired Elastic Inference accelerator to run inference.

4. Use your preferred text editor to create a script that has the following content. Name it `pytorch_resnet50_inference.py`. This script uses ImageNet pretrained TorchVision model weights for ResNet-50, a popular convolutional neural network for image classification. It traces the weights with an image tensor and saves it. The script then loads the saved model, performs inference on the input, and prints out the top predicted ImageNet classes.

   This script uses the `torch.jit.optimized_execution` context, which is necessary to use the Elastic Inference accelerator. If you don't use the `torch.jit.optimized_execution` context correctly, then inference will run entirely on the client instance and won't use the attached accelerator. The Elastic Inference enabled PyTorch framework accepts two parameters for this context, while the vanilla PyTorch framework accepts only one parameter. The second parameter is used to specify the accelerator device ordinal. `target_device` should be set to the device's ordinal number, not its ID. Ordinals are numbered beginning with 0.

   Note
   This script specifies the CPU device when loading the model. This avoids potential problems if the model was traced and saved using a GPU context.

```
import torch, torchvision
import PIL
from torchvision import transforms
from PIL import Image

def get_image(filename):
```

```
  im = Image.open(filename)
  # ImageNet pretrained models required input images to have width/height of 224
  # and color channels normalized according to ImageNet distribution.
  im_process = transforms.Compose([transforms.Resize([224, 224]),
                                   transforms.ToTensor(),
                                   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                        std=[0.229, 0.224, 0.225])])
  im = im_process(im) # 3 x 224 x 224
  return im.unsqueeze(0) # Add dimension to become 1 x 3 x 224 x 224

im = get_image('kitten.jpg')

# eval() toggles inference mode
model = torchvision.models.resnet50(pretrained=True).eval()
# Compile model to TorchScript via tracing
# Here want to use the first attached accelerator, so we specify ordinal 0.
with torch.jit.optimized_execution(True, {'target_device': 'eia:0'}):
  # You can trace with any input
  model = torch.jit.trace(model, im)

# Serialize model
torch.jit.save(model, 'resnet50_traced.pt')

# Deserialize model
model = torch.jit.load('resnet50_traced.pt', map_location=torch.device('cpu'))

# Perform inference. Make sure to disable autograd and use EI execution context
with torch.no_grad():
  with torch.jit.optimized_execution(True, {'target_device': 'eia:device ordinal'}):
    probs = model(im)

# Torchvision implementation doesn't have Softmax as last layer.
# Use Softmax to convert activations to range 0-1 (probabilities)
probs = torch.nn.Softmax(dim=1)(probs)

# Get top 5 predicted classes
classes = eval(open('imagenet_classes.txt').read())
pred_probs, pred_indices = torch.topk(probs, 5)
pred_probs = pred_probs.squeeze().numpy()
pred_indices = pred_indices.squeeze().numpy()

for i in range(len(pred_indices)):
  curr_class = classes[pred_indices[i]]
  curr_prob = pred_probs[i]
  print('{}: {:.4f}'.format(curr_class, curr_prob))
```

Note
You don't have to save and load your model. You can compile your model, then directly do inference with it. The benefit to saving your model is that it will save time for future inference jobs.

5. Run the inference script.

```
python pytorch_resnet50_inference.py
```

Your output should be similar to the following. The model predicts that the image is most likely to be a tabby cat, followed by a tiger cat.

```
Using Amazon Elastic Inference Client Library Version: 1.6.2
Number of Elastic Inference Accelerators Available: 1
Elastic Inference Accelerator ID: eia-53ab0670550948e88d7aac0bd331a583
Elastic Inference Accelerator Type: eia2.medium
Elastic Inference Accelerator Ordinal: 0
```

```
tabby, tabby cat: 0.4674
tiger cat: 0.4526
Egyptian cat: 0.0667
plastic bag: 0.0025
lynx, catamount: 0.0007
```

Monitoring Elastic Inference Accelerators

The following tools are provided to monitor and check the status of your Elastic Inference accelerators.

EI_VISIBLE_DEVICES

`EI_VISIBLE_DEVICES` is an environment variable that you use to control which Elastic Inference accelerator devices are visible to the deep learning frameworks. `EI_VISIBLE_DEVICES` can also be used with `EI Tool`. The variable is a comma-separated list of device ordinal numbers or device IDs. Use `EI Tool` to see all attached Elastic Inference accelerator devices.

`EI_VISIBLE_DEVICES` is used as follows. In this example, only the device with the ordinal number value `3` will be used when starting the server.

```
EI_VISIBLE_DEVICES=3 amazonei_tensorflow_model_server --port=8502 --rest_api_port=8503 --
model_name=ssdresnet --model_base_path=/home/ec2-user/models/ssdresnet
```

If `EI_VISIBLE_DEVICES` is not set, then all attached devices are visible. If `EI_VISIBLE_DEVICES` is set to an empty string, then none of the devices are visible.

Using EI_VISIBLE_DEVICES with Multiple Devices

To pass multiple devices with `EI_VISIBLE_DEVICES`, use a comma-separated list. This list can contain device ordinal numbers or device IDs. The following command shows the use of multiple devices with `EI Tool`:

```
EI_VISIBLE_DEVICES=1,3 /opt/amazon/ei/ei_tools/bin/ei describe-accelerators -j
```

When using multiple Elastic Inference accelerators with `EI_VISIBLE_DEVICES`, the devices visible to the framework take on new ordinal numbers within the process. They will be labeled within the process starting from zero. This change only happens within the process. It does not have any impact on the ordinal numbers of the devices outside of the process. It also does not impact devices that are not included in `EI_VISIBLE_DEVICES`.

Exporting EI_VISIBLE_DEVICES

To set the `EI_VISIBLE_DEVICES` variable for use with all child processes of the current shell process, use the following command:

```
export EI_VISIBLE_DEVICES=1,3
```

All subsequently launched processes use this value. You must override or update the `EI_VISIBLE_DEVICES` value to change this behavior.

EI Tool

The `EI Tool` is a binary that comes with the latest version, v26.0, of the Conda DLAMI. You can also download it from the Amazon S3 Bucket. It can be used to monitor the status of multiple Elastic Inference accelerators.

By default, running `EI Tool` as follows prints basic information about the Elastic Inference accelerators attached to the Amazon Elastic Compute Cloud instance.

```
ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ ./ei describe-accelerators
EI Client Version: 1.5.0Time: Fri Nov  1 03:09:15 2019
Attached accelerators: 2
Device 0:
    Type: eia1.xlarge
    Id: eia-679e4c622d584803aed5b42ab6a97706
    Status: healthy
Device 1:
    Type: eia1.xlarge
    Id: eia-6c414c6ee37a4d93874afc00825c2f28
    Status: healthy
```

The following topic describes options for using `EI Tool` from the command line.

Getting Help

There are two ways to get help when using `EI Tool`. The following are the two methods for accessing help.

- The `EI Tool` will output usage information if a command is not provided.

```
ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ ./ei
Usage: ei describe-accelerators [options]
Print description of attached accelerators.
Options:
-j, --json    Print description of attached accelerators in JSON format.
-h, --help    Print this help instructions and exit.
ubuntu@ip-10-0-0-98:~/ei_tools/bin$ echo $?
1
```

- You can use the `-h` and `—help` switches to output the same information.

```
ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ ./ei describe-accelerators -h
Usage: ei describe-accelerators [options]
Print description of attached accelerators.
Options:
-j, --json    Print description of attached accelerators in JSON format.
-h, --help    Print this help instructions and exit.

ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ ./ei describe-accelerators --help
Usage: ei describe-accelerators [options]
Print description of attached accelerators.
Options:
-j, --json    Print description of attached accelerators in JSON format.
-h, --help    Print this help instructions and exit.
```

JSON

The `EI Tool` supports JSON output when describing attached Elastic Inference accelerators. The **`-j/--json`** switches can be used to print the accelerator state description as a JSON object.

```
ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ ./ei describe-accelerators -j
{
  "ei_client_version": "1.5.0",
  "time": "Fri Nov  1 03:09:38 2019",
  "attached_accelerators": 2,
  "devices": [
    {
      "ordinal": 0,
      "type": "eia1.xlarge",
      "id": "eia-679e4c622d584803aed5b42ab6a97706",
      "status": "healthy"
    },
    {
      "ordinal": 1,
      "type": "eia1.xlarge",
      "id": "eia-6c414c6ee37a4d93874afc00825c2f28",
      "status": "healthy"
    }
  ]
}

ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ ./ei describe-accelerators --json
{
  "ei_client_version": "1.5.0",
  "time": "Fri Nov  1 03:10:15 2019",
  "attached_accelerators": 2,
  "devices": [
    {
      "ordinal": 0,
      "type": "eia1.xlarge",
      "id": "eia-679e4c622d584803aed5b42ab6a97706",
      "status": "healthy"
    },
    {
      "ordinal": 1,
      "type": "eia1.xlarge",
      "id": "eia-6c414c6ee37a4d93874afc00825c2f28",
      "status": "healthy"
    }
  ]
}
```

Errors

Errors encountered when running `EI Tool` are output to `stderr`. The following illustrates an error encountered due to blocked outgoing traffic.

```
ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ ./ei describe-accelerators
[Fri Nov  1 03:20:29 2019, 046923us] [Connect] Failed. Error message - Last Error:
    EI Error Code: [1, 4, 1]
    EI Error Description: Internal error
    EI Request ID: MX-EFBD3C87-6E8E-4E99-
A855-949CB2A24E7F  --  EI Accelerator ID: eia-679e4c622d584803aed5b42ab6a97706
    EI Client Version: 1.5.0
[Fri Nov  1 03:20:44 2019, 055905us] [Connect] Failed. Error message - Last Error:
```

```
    EI Error Code: [1, 4, 1]
    EI Error Description: Internal error
    EI Request ID: MX-BD40C53D-6BBC-49A8-
AF6D-27FF542DA38A  --  EI Accelerator ID: eia-6c414c6ee37a4d93874afc00825c2f28
    EI Client Version: 1.5.0
EI Client Version: 1.5.0Time: Fri Nov  1 03:20:44 2019
Attached accelerators: 2
Device 0:
    Type: eia1.xlarge
    Id: eia-679e4c622d584803aed5b42ab6a97706
    Status: not reachable
Device 1:
    Type: eia1.xlarge
    Id: eia-6c414c6ee37a4d93874afc00825c2f28
    Status: not reachable
ubuntu@ip-10-0-0-98:~/ei_tools/bin$ echo $?
0
```

It's important to note that a JSON object is also output when the `-j/--json` switches are set. Even though errors encountered when running `EI Tool` are output to `stderr`, the `stdout` can still be parsed as a `JSON` object.

```
ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ ./ei describe-accelerators -j
E1101 03:54:54.084712 25091 log_stream.cpp:232] [Connect] Failed. Error message - Last Error:
    EI Error Code: [1, 4, 1]
    EI Error Description: Internal error

  EI Request ID: MX-192D16B1-65CD-43AA-9CA8-0D717D134C0E  --  EI Accelerator ID: eia-679e4c622d584803ae
    EI Client Version: 1.5.0
E1101 03:55:09.096704 25091 log_stream.cpp:232] [Connect] Failed. Error message - Last Error:
    EI Error Code: [1, 4, 1]
    EI Error Description: Internal error
    EI Request ID: MX-A4C4C90E-FC13-4D58-
AA4F-54382222E8D7  --  EI Accelerator ID: eia-6c414c6ee37a4d93874afc00825c2f28
    EI Client Version: 1.5.0
{
  "ei_client_version": "1.5.0",
  "time": "Fri Nov  1 03:55:09 2019",
  "attached_accelerators": 2,
  "devices": [
    {
      "ordinal": 0,
      "type": "eia1.xlarge",
      "id": "eia-679e4c622d584803aed5b42ab6a97706",
      "status": "not reachable"
    },
    {
      "ordinal": 1,
      "type": "eia1.xlarge",
      "id": "eia-6c414c6ee37a4d93874afc00825c2f28",
      "status": "not reachable"
    }
  ]
}
```

Using EI Tool with LD_LIBRARY_PATH

If there has been a change to your local `LD_LIBRARY_PATH` variable, you may have to modify your use of `EI_Tool`. Include the following `LD_LIBRARY_PATH` value when using `EI_Tool`:

```
LD_LIBRARY_PATH=/opt/amazon/ei/ei_tools/lib
```

The following example uses this value with a single Elastic Inference accelerator:

```
EI_VISIBLE_DEVICES=1 LD_LIBRARY_PATH=/opt/amazon/ei/ei_tools/lib /opt/amazon/ei/ei_tools/
bin/ei describe-accelerators -j
{
  "ei_client_version": "1.5.3",
  "time": "Tue Nov 19 16:57:21 2019",
  "attached_accelerators": 1,
  "devices": [
    {
      "ordinal": 0,
      "type": "eia1.xlarge",
      "id": "eia-7f127e2640e642d48a7d4673a57581be",
      "status": "healthy"
    }
  ]
}
```

Health Check

You can use `Health Check` to monitor the health of your Elastic Inference accelerators. The exit code of the `Health Check` command is `0` if all accelerators are healthy and reachable. If they are not, then the exit code is `1`.

```
ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ ./health_check
EI Client Version: 1.5.0
Device 0: healthy
Device 1: healthy
ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ echo $?
0
```

The following illustrates an error due to blocked traffic received when running `Health Check`.

```
ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ ./health_check
[Fri Nov  1 07:00:47 2019, 134735us] [Connect] Failed. Error message - Last Error:
    EI Error Code: [1, 4, 1]
    EI Error Description: Internal error
    EI Request ID: MX-
A0558121-49D8-48DB-8CCB-9322D78BFCA5  --  EI Accelerator ID: eia-679e4c622d584803aed5b42ab6a97706
    EI Client Version: 1.5.0
Device 0: not reachable
[Fri Nov  1 07:01:02 2019, 143732us] [Connect] Failed. Error message - Last Error:
    EI Error Code: [1, 4, 1]
    EI Error Description: Internal error
    EI Request ID: MX-AC879033-FB46-46EE-B2B6-A76F5E674E0D
  --  EI Accelerator ID: eia-6c414c6ee37a4d93874afc00825c2f28
    EI Client Version: 1.5.0
Device 1: not reachable
ubuntu@ip-10-0-0-98:/opt/amazon/ei/ei_tools/bin$ echo $?
1
```

MXNet Elastic Inference with SageMaker

By using Amazon Elastic Inference, you can speed up the throughput and decrease the latency of getting real-time inferences from your deep learning models that are deployed as Amazon SageMaker hosted models, but at a fraction of the cost of using a GPU instance for your endpoint.

For more information, see the Amazon SageMaker Elastic Inference Documentation

Using Amazon Deep Learning Containers With Elastic Inference

Amazon Deep Learning Containers with Amazon Elastic Inference (Elastic Inference) are a set of Docker images for serving models in TensorFlow, Apache MXNet (MXNet), and PyTorch. Deep Learning Containers can include a wide variety of options for deep learning. These containers are only available for inference jobs and should not be used for training. See Deep Learning Containers Images for training images. For community discussion, see the Deep Learning Containers Discussion Forum

You can use Deep Learning Containers with Elastic Inference on Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Elastic Container Service (Amazon ECS).

Topics

Using Amazon Deep Learning Containers with Amazon Elastic Inference on Amazon EC2

Amazon Deep Learning Containers with Amazon Elastic Inference (Elastic Inference) are a set of Docker images for serving models in TensorFlow, Apache MXNet (MXNet), and PyTorch on Amazon Elastic Compute Cloud (Amazon EC2). Deep Learning Containers provide optimized environments with TensorFlow, MXNet, and PyTorch. They are available in the Amazon Elastic Container Registry (Amazon ECR).

These tutorials describe how to use Deep Learning Containers with Elastic Inference on Amazon Elastic Compute Cloud (Amazon EC2).

Topics

Prerequisites

Before you start this tutorial, set up the following resources in the AWS Management Console.

1. Create an AWS Identity and Access Management (IAM) user and attach the following policies:
   - AmazonECS_FullAccess Policy
   - AmazonEC2ContainerRegistryFullAccess
2. Follow the instructions for Setting Up EI with the following modification:

   Create a security group (use the default VPC, or create a VPC with an internet gateway) and open the ports necessary for your desired inference server:

- All frameworks require: 22 for SSH and 443 for HTTPS
- TensorFlow inference: 8500 and 8501 open to TCP traffic
- MXNet and PyTorch inference: 80 and 8081 open to TCP traffic

3. Launch an Amazon EC2 instance with the Elastic Inference role using the AWS Deep Learning Base Amazon Machine Image (AMI). Because you need only the AWS Command Line Interface (AWS CLI) and Docker, this is the best AMI.
4. SSH into the Amazon EC2 instance.
5. On the instance, run the following commands using the keys associated with the user created in Step1. Confirm that Elastic Inference is available in your region.

```
aws configure set aws_access_key_id <access_key_id>

aws configure set aws_secret_access_key <secret_access_key>

aws configure set region <region>

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

Using TensorFlow Elastic Inference accelerators on EC2

When using Elastic Inference, you can use the same Amazon EC2 instance for models on two different frameworks. To do so, use the console to stop the Amazon EC2 instance and restart it, instead of rebooting it. The Elastic Inference accelerator doesn't detach when you reboot the instance.

To use the Elastic Inference accelerator with TensorFlow

1. From the command line of your Amazon EC2 instance, pull the TF-EI image from Amazon Elastic Container Registry (Amazon ECR) with the following code. To select an image, see Deep Learning Containers Images.

```
docker pull 763104351884.dkr.ecr.<region>.amazonaws.com/tensorflow-inference-
eia:<image_tag>
```

2. Clone the GitHub Tensorflow repository for serving the half_plus_three model.

```
https://github.com/tensorflow/serving.git
```

3. Run the container with entry point for TF-half-plus-three. You can get the <image_id> by running the `docker images` command.

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

4. Begin inference on the same instance using a query with the REST API.

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

5. Optionally, query from another Amazon EC2 instance. Make sure that the 8500 and 8501 ports are exposed in the security group.

```
curl -d '{"instances": [1.0, 2.0, 5.0]}' -X POST http://<ec2_public_ip_address>:8501/
v1/models/saved_model_half_plus_three:predict
```

6. The results should look something like the following.

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

Using MXNet Elastic Inference accelerators on Amazon EC2

When using Elastic Inference, you can use the same Amazon EC2 instance for models on two different frameworks. To do so, use the console to top the Amazon EC2 instance and restart it, instead of rebooting it. The Elastic Inference accelerator doesn't detach when you reboot the instance.

To use the Elastic Inference accelerator with MXNet

1. Pull the MXNet-Elastic Inference image from Amazon Elastic Container Registry (Amazon ECR). To select an image, see Deep Learning Containers Images.

```
docker pull 763104351884.dkr.ecr.<region>.amazonaws.com/mxnet-inference-eia:<image_tag>
```

2. Run the container with the following command. You can get the `<image_id>` by running the `docker images` command.

```
docker run -itd --name mxnet_inference -p 80:8080 -p 8081:8081 <image_id> \
        mxnet-model-server --start --foreground \
        --mms-config /home/model-server/config.properties \
        --models resnet-152-eia=https://s3.amazonaws.com/model-server/
model_archive_1.0/resnet-152-eia.mar
```

3. Download the input image for the test.

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

4. Begin inference using a query with the REST API.

```
curl -X POST http://127.0.0.1:80/predictions/resnet-152-eia -T kitten.jpg
```

5. The results should look something like the following.

```
[
  {
    "probability": 0.8582226634025574,
    "class": "n02124075 Egyptian cat"
  },
  {
    "probability": 0.09160050004720688,
    "class": "n02123045 tabby, tabby cat"
  },
  {
    "probability": 0.037487514317035675,
    "class": "n02123159 tiger cat"
```

```
  },
  {
    "probability": 0.0061649843119084835,
    "class": "n02128385 leopard, Panthera pardus"
  },
  {
    "probability": 0.003171598305925727,
    "class": "n02127052 lynx, catamount"
  }
]
```

Using PyTorch Elastic Inference accelerators on Amazon EC2

When using Elastic Inference, you can use the same Amazon EC2 instance for models on multiple frameworks. To do so, use the console to stop the Amazon EC2 instance and restart it, instead of rebooting it. The Elastic Inference accelerator doesn't detach when you reboot the instance.

To use the Elastic Inference accelerators with PyTorch

1. From the terminal of your Amazon EC2 instance, pull the Elastic Inference enabled PyTorch image from Amazon Elastic Container Registry (Amazon ECR) with the following code. To select an image, see Deep Learning Containers Images.

```
docker pull 763104351884.dkr.ecr.<region>.amazonaws.com/pytorch-inference-
eia:<image_tag>
```

2. Run the container with the following command. You can get the `<image_id>` by running the `docker images` command.

```
docker run -itd --name pytorch_inference_eia -p 80:8080 -p 8081:8081 <image_id> \
        mxnet-model-server --start --foreground \
        --mms-config /home/model-server/config.properties \
        --models densenet-eia=https://aws-dlc-sample-models.s3.amazonaws.com/pytorch/
densenet_eia/densenet_eia.mar
```

3. Download an image of a flower to use as the input image for the test.

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

4. Begin inference using a query with the REST API.

```
curl -X POST http://127.0.0.1:80/predictions/densenet-eia -T flower.jpg
```

5. The results should look something like the following.

```
[
  [
    "pot, flowerpot",
    14.690367698669434
  ],
  [
    "sulphur butterfly, sulfur butterfly",
    9.29893970489502
  ],
  [
    "bee",
```

```
    8.29178237915039
  ],
  [
    "vase",
    6.987090587615967
  ],
  [
    "hummingbird",
    4.341294765472412
  ]
]
```

Using Deep Learning Containers with Amazon Deep Learning Containers on Amazon ECS

Amazon Deep Learning Containers are a set of Docker images for training and serving models in TensorFlow, Apache MXNet (MXNet), and PyTorch on Amazon Elastic Container Service (Amazon ECS). Deep Learning Containers provide optimized environments with TensorFlow, MXNet, and PyTorch. They are available in Amazon Elastic Container Registry (Amazon ECR).

This tutorial describes how to use Deep Learning Containers with Elastic Inference on Amazon ECS.

Topics

Prerequisites

Your Amazon ECS container instances require at least version 1.30.0 of the container agent. For information about checking your agent version and updating to the latest version, see Updating the Amazon ECS Container Agent.

Resource Setup

To complete the tutorial, set up the following resources in the AWS Console

1. Perform the steps for Setting Up Deep Learning Containers on ECS.
   - If you create a VPC and security group, ensure that there is a subnet in an Availability Zone with Elastic Inference available.
2. Follow the instructions for Setting Up EI with the following modifications:

   Complete all sections except Launching an Instance with Elastic Inference.

   Create a security group (use the default VPC, or create a VPC with an internet gateway) and open the ports necessary for your desired inference server:

   - Both frameworks require: 22 for SSH and 443 for HTTPS
   - TensorFlow inference: 8500 and 8501 open to TCP traffic
   - MXNet and PyTorch inference: 80 and 8081 open to TCP traffic

Add the ec2-role-trust-policy.json IAM policy described in the Elastic Inference setup instructions to the ecsInstanceRole IAM role.

Using TensorFlow Elastic Inference accelerators on Amazon ECS

To use the Elastic Inference accelerator with TensorFlow

1. Create an Amazon ECS cluster named **`tensorflow-eia`** on AWS in an AWS Region that has access to Elastic Inference.

```
aws ecs create-cluster --cluster-name tensorflow-eia \
                       --region <region>
```

2. Create a text file called `tf_script.txt` and add the following text.

```
#!/bin/bash
echo ECS_CLUSTER=tensorflow-eia >> /etc/ecs/ecs.config
```

3. Create a text file called `my_mapping.txt` and add the following text.

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

4. Launch an Amazon EC2 instance in the cluster that you created in Step 1 without attaching an Elastic Inference accelerator. Use Amazon ECS-optimized AMIs to get an image-id.

```
aws ec2 run-instances --image-id <ECS_Optimized_AMI> \
                      --count 1 \
                      --instance-type <cpu_instance_type> \
                      --key-name <name_of_key_pair_on_ec2_console>
                      --security-group-ids <sg_created_with_vpc> \
                      --iam-instance-profile Name="ecsInstanceRole" \
                      --user-data file://tf_script.txt \
                      --block-device-mapping file://my_mapping.txt \
                      --region <region> \
                      --subnet-id <subnet_with_ei_endpoint>
```

5. For all Amazon EC2 instances that you launch, use the ecsInstanceRole IAM role. Make note of the public IPv4 address when the instance is started.
6. Create a TensorFlow inference task definition with the name tf_task_def.json. Set `"image"` to any TensorFlow image name. To select an image, see Prebuilt Amazon SageMaker Docker Images. For "deviceType" options, see Launching an Instance with Elastic Inference.

```
{
    "requiresCompatibilities":[
        "EC2"
    ],
    "containerDefinitions":[
```

```
        {
            "entryPoint":[
                "/bin/bash",
                "-c",
                "mkdir -p /test && cd /test && git clone -b r1.14 https://github.com/
tensorflow/serving.git && cd / && /usr/bin/tensorflow_model_server --port=8500 --
rest_api_port=8501 --model_name=saved_model_half_plus_three --model_base_path=/test/
serving/tensorflow_serving/servables/tensorflow/testdata/saved_model_half_plus_three"
            ],
            "name":"tensorflow-inference-container",
            "image":"<tensorflow-image-uri>",
            "memory":8111,
            "cpu":256,
            "essential":true,
            "portMappings":[
                {
                    "hostPort":8500,
                    "protocol":"tcp",
                    "containerPort":8500
                },
                {
                    "hostPort":8501,
                    "protocol":"tcp",
                    "containerPort":8501
                },
                {
                    "containerPort":80,
                    "protocol":"tcp"
                }
            ],
            "healthCheck":{
                "retries":2,
                "command":[
                    "CMD-SHELL",
                    "LD_LIBRARY_PATH=/opt/ei_health_check/lib /opt/ei_health_check/
health_check"
                ],
                "timeout":5,
                "interval":30,
                "startPeriod":60
            },
            "logConfiguration":{
                "logDriver":"awslogs",
                "options":{
                    "awslogs-group":"/ecs/tensorflow-inference-eia",
                    "awslogs-region":"<region>",
                    "awslogs-stream-prefix":"half-plus-three",
                    "awslogs-create-group":"true"
                }
            },
            "resourceRequirements":[
                {
                    "type":"InferenceAccelerator",
                    "value":"device_1"
                }
            ]
        }
     ],
     "inferenceAccelerators":[
         {
             "deviceName":"device_1",
             "deviceType":"<EIA_instance_type>"
         }
    ],
    "volumes":[
```

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

     ],
     "family":"tensorflow-eia"
}
```

7. Register the TensorFlow inference task definition. Note the task definition family and revision number from the output of the following command.

```
aws ecs register-task-definition --cli-input-json file://tf_task_def.json --
region <region>
```

8. Create a TensorFlow inference service.

```
aws ecs create-service --cluster tensorflow-eia --service-name tf-eia1 --task-
definition tensorflow-eia:<revision_number> --desired-count 1 --scheduling-
strategy="REPLICA" --region <region>
```

9. Begin inference using a query with the REST API.

```
curl -d '{"instances": [1.0, 2.0, 5.0]}' -X POST http://<public-ec2-ip-address>:8501/
v1/models/saved_model_half_plus_three:predict
```

10. The results should look something like the following.

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

Using MXNet Elastic Inference accelerators on Amazon ECS

To use the Elastic Inference accelerator with MXNet

1. Create an Amazon ECS cluster with named ***mxnet-eia*** on AWS in an AWS Region that has access to Elastic Inference.

```
aws ecs create-cluster --cluster-name mxnet-eia \
                       --region <region>
```

2. Create a text file called `mx_script.txt` and add the following text.

```
#!/bin/bash
echo ECS_CLUSTER=mxnet-eia >> /etc/ecs/ecs.config
```

3. Create a text file called `my_mapping.txt` and add the following text.

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

```
]
```

4. Launch an Amazon EC2 instance in the cluster that you created in Step 1 without attaching an Elastic Inference accelerator. To select an AMI, see Amazon ECS-optimized AMIs.

```
aws ec2 run-instances --image-id <ECS_Optimized_AMI> \
                      --count 1 \
                      --instance-type <cpu_instance_type> \
                      --key-name <name_of_key_pair_on_ec2_console>
                      --security-group-ids <sg_created_with_vpc> \
                      --iam-instance-profile Name="ecsInstanceRole" \
                      --user-data file://mx_script.txt \
                      --block-device-mapping file://my_mapping.txt \
                      --region <region> \
                      --subnet-id <subnet_with_ei_endpoint>
```

5. Create an MXNet inference task definition named mx_task_def.json. Set "`image`" to any MXNet image name. To select an image, see Prebuilt Amazon SageMaker Docker Images. For "deviceType" options, see Launching an Instance with Elastic Inference.

```
{
   "requiresCompatibilities":[
      "EC2"
   ],
   "containerDefinitions":[
      {
         "entryPoint":[
                "/bin/bash",
                "-c",
           "/usr/local/bin/mxnet-model-server --start --foreground --mms-config /home/
model-server/config.properties --models resnet-152-eia=https://s3.amazonaws.com/model-
server/model_archive_1.0/resnet-152-eia.mar"],
         "name":"mxnet-inference-container",
         "image":"<mxnet-image-name>",
         "memory":8111,
         "cpu":256,
         "essential":true,
         "portMappings":[
            {
               "hostPort":80,
               "protocol":"tcp",
               "containerPort":8080
            },
            {
               "hostPort":8081,
               "protocol":"tcp",
               "containerPort":8081
            }
         ],
         "healthCheck":{
            "retries":2,
            "command":[
               "CMD-SHELL",
               "LD_LIBRARY_PATH=/opt/ei_health_check/lib /opt/ei_health_check/bin/
health_check"
            ],
            "timeout":5,
            "interval":30,
            "startPeriod":60
         },
         "logConfiguration":{
            "logDriver":"awslogs",
            "options":{
               "awslogs-group":"/ecs/mxnet-inference-eia",
```

```
                "awslogs-region":"<region>",
                "awslogs-stream-prefix":"squeezenet",
                "awslogs-create-group":"true"
             }
          },
          "resourceRequirements":[
                 {
                     "type":"InferenceAccelerator",
                     "value":"device_1"
                 }
          ]
       }
   ],
   "inferenceAccelerators":[
      {
         "deviceName":"device_1",
         "deviceType":"<EIA_instance_type>"
      }
   ],
   "volumes":[

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

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

6. Register the MXNet inference task definition. Note the task definition family and revision number in the output.

```
aws ecs register-task-definition --cli-input-json file://mx_task_def.json --
region <region>
```

7. Create an MXNet inference service.

```
aws ecs create-service --cluster mxnet-eia --service-name mx-eia1 --task-definition
 mxnet-eia:<revision_number> --desired-count 1 --scheduling-strategy="REPLICA" --
region <region>
```

8. Download the input image for the test.

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

9. Begin inference using a query with the REST API.

```
curl -X POST http://<ec2_public_ip_address>:80/predictions/resnet-152-eia -T kitten.jpg
```

10. The results should look something like the following.

```
[
  {
    "probability": 0.8582226634025574,
    "class": "n02124075 Egyptian cat"
  },
  {
    "probability": 0.09160050004720688,
    "class": "n02123045 tabby, tabby cat"
  },
  {
    "probability": 0.037487514317035675,
```

```
    "class": "n02123159 tiger cat"
  },
  {
    "probability": 0.0061649843119084835,
    "class": "n02128385 leopard, Panthera pardus"
  },
  {
    "probability": 0.003171598305925727,
    "class": "n02127052 lynx, catamount"
  }
]
```

Using PyTorch Elastic Inference accelerators on Amazon ECS

To use Elastic Inference accelerators with PyTorch

1. From your terminal, create an Amazon ECS cluster named **`pytorch-eia`** on AWS in an AWS Region that has access to Elastic Inference.

```
aws ecs create-cluster --cluster-name pytorch-eia \
                       --region <region>
```

2. Create a text file called `pt_script.txt` and add the following text.

```
#!/bin/bash
echo ECS_CLUSTER=pytorch-eia >> /etc/ecs/ecs.config
```

3. Create a text file called `my_mapping.txt` and add the following text.

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

4. Launch an Amazon EC2 instance in the cluster that you created in Step 1 without attaching an Elastic Inference accelerator. To select an AMI, see Amazon ECS-optimized AMIs.

```
aws ec2 run-instances --image-id <ECS_Optimized_AMI> \
                      --count 1 \
                      --instance-type <cpu_instance_type> \
                      --key-name <name_of_key_pair_on_ec2_console>
                      --security-group-ids <sg_created_with_vpc> \
                      --iam-instance-profile Name="ecsInstanceRole" \
                      --user-data file://pt_script.txt \
                      --block-device-mapping file://my_mapping.txt \
                      --region <region> \
                      --subnet-id <subnet_with_ei_endpoint>
```

5. Create a PyTorch inference task definition named pt_task_def.json. Set "`image`" to any PyTorch image name. To select an image, see Prebuilt Amazon SageMaker Docker Images. For "deviceType" options, see Launching an Instance with Elastic Inference.

```
{
```

```
   "requiresCompatibilities":[
      "EC2"
   ],
   "containerDefinitions":[
      {
         "entryPoint":[
                "/bin/bash",
                "-c",
           "mxnet-model-server --start --foreground --mms-config /home/model-server/
config.properties --models densenet-eia=https://aws-dlc-sample-models.s3.amazonaws.com/
pytorch/densenet_eia/densenet_eia.mar"],
         "name":"pytorch-inference-container",
         "image":"<pytorch-image-name>",
         "memory":8111,
         "cpu":256,
         "essential":true,
         "portMappings":[
            {
               "hostPort":80,
               "protocol":"tcp",
               "containerPort":8080
            },
            {
               "hostPort":8081,
               "protocol":"tcp",
               "containerPort":8081
            }
         ],
         "healthCheck":{
            "retries":2,
            "command":[
               "CMD-SHELL",
               "LD_LIBRARY_PATH=/opt/ei_health_check/lib /opt/ei_health_check/bin/
health_check"
            ],
            "timeout":5,
            "interval":30,
            "startPeriod":60
         },
         "logConfiguration":{
            "logDriver":"awslogs",
            "options":{
               "awslogs-group":"/ecs/pytorch-inference-eia",
               "awslogs-region":"<region>",
               "awslogs-stream-prefix":"densenet-eia",
               "awslogs-create-group":"true"
            }
         },
         "resourceRequirements":[
                {
                    "type":"InferenceAccelerator",
                    "value":"device_1"
                }
         ]
      }
   ],
   "inferenceAccelerators":[
      {
         "deviceName":"device_1",
         "deviceType":"<EIA_instance_type>"
      }
   ],
   "volumes":[

   ],
   "networkMode":"bridge",
```

```
   "placementConstraints":[

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

6. Register the PyTorch inference task definition. Note the task definition family and revision number in the output.

```
aws ecs register-task-definition --cli-input-json file://pt_task_def.json --
region <region>
```

7. Create a PyTorch inference service.

```
aws ecs create-service --cluster pytorch-eia --service-name pt-eia1 --task-definition
 pytorch-eia:<revision_number> --desired-count 1 --scheduling-strategy="REPLICA" --
region <region>
```

8. Download the input image for the test.

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

9. Begin inference using a query with the REST API.

```
curl -X POST http://<ec2_public_ip_address>:80/predictions/densenet-eia -T flower.jpg
```

10. The results should look something like the following.

```
[
  [
    "pot, flowerpot",
    14.690367698669434
  ],
  [
    "sulphur butterfly, sulfur butterfly",
    9.29893970489502
  ],
  [
    "bee",
    8.29178237915039
  ],
  [
    "vase",
    6.987090587615967
  ],
  [
    "hummingbird",
    4.341294765472412
  ]
]
```

Using Amazon Deep Learning Containers with Elastic Inference on Amazon SageMaker

Amazon Deep Learning Containers with Elastic Inference are a set of Docker images for serving models in TensorFlow, Apache MXNet (MXNet), and PyTorch on Amazon SageMaker. Deep Learning Containers

provide optimized environments with TensorFlow, MXNet, and PyTorch. They are available in the Amazon Elastic Container Registry (Amazon ECR).

For more information, see Amazon SageMaker Elastic Inference.

Security in Amazon Elastic Inference

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 Elastic Inference, 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 Elastic Inference. The following topics show you how to configure Elastic Inference to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Elastic Inference resources.

Topics

Identity and Access Management for Amazon Elastic Inference

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 resources. IAM is an AWS service that you can use with no additional charge.

Your Identity and Access Management options may vary depending on what your Elastic Inference accelerator is attached to. For more information on Identity and Access Management, see:

- Identity and Access Management for Amazon EC2
- Identity and Access Management for Amazon Elastic Container Service
- Identity and Access Management for Amazon SageMaker

Topics

Authenticating With Identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see Signing in to the AWS Management Console as an IAM user or root user in the *IAM User Guide*.

You must be *authenticated* (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email address or your IAM user name. You can access AWS programmatically using your root user or IAM users access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 signing process in the *AWS General Reference*.

Regardless of the authentication method that you use, you might also be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM Users and Groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing access keys for IAM users in the *IAM User Guide*. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to create an IAM user (instead of a role) in the *IAM User Guide*.

IAM Roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS

Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM roles in the *IAM User Guide*.

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

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated users and roles in the *IAM User Guide*.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see in the *Service Authorization Reference*.
  - **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. Service roles provide access only within your account and cannot be used to grant access to services in other accounts. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.
  - **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) in the *IAM User Guide*.

Managing Access Using Policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. You can sign in as the root user or an IAM user, or you can assume an IAM role. When you then make a request, AWS evaluates the related identity-based or resource-based policies. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON

documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

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

Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

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

Identity-Based Policies

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

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing between managed policies and inline policies in the *IAM User Guide*.

Resource-Based Policies

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

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

Access Control Lists (ACLs)

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

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

Other Policy Types

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

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role).

You can set a permissions boundary for an entity. The resulting permissions are the intersection of entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.

- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see How SCPs work in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple Policy Types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

Logging and Monitoring in Amazon Elastic Inference

Your Amazon Elastic Inference instance comes with tools to monitor the health and status of your accelerators. You can also monitor your Elastic Inference accelerators using Amazon CloudWatch, which collects metrics about your usage and performance. For more information, see Monitoring Elastic Inference Accelerators and Using CloudWatch Metrics to Monitor Elastic Inference.

Compliance Validation for Amazon Elastic Inference

Third-party auditors assess the security and compliance of Amazon Elastic Inference as part of multiple AWS compliance programs. The supported compliance programs may vary depending on what your Elastic Inference accelerator is attached to. For information on the supported compliance programs, see:

- Compliance Validation for Amazon EC2
- Compliance Validation for Amazon SageMaker
- Compliance Validation for Amazon Elastic Container Service

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 Elastic Inference 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 Amazon Elastic Inference

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.

Features to support you data resiliency needs may vary depending on what your Elastic Inference accelerator is attached to. For information on features to help support your data resiliency and backup needs, see:

- Resilience in Amazon EC2
- Resilience in Amazon SageMaker

Infrastructure Security in Amazon Elastic Inference

The infrastructure security of Amazon Elastic Inference may vary depending on what your Elastic Inference accelerator is attached to. For more information, see:

- Infrastructure Security in Amazon EC2
- Infrastructure Security in Amazon SageMaker
- Infrastructure Security in Amazon Elastic Container Service

Configuration and Vulnerability Analysis in Amazon Elastic Inference

AWS supplies updates for Amazon Elastic Inference that do not require any customer action. These updates to Elastic Inference and the supported frameworks are built into the latest Elastic Inference version. Because AWS provides new libraries for Elastic Inference, you must launch a new instance with an Elastic Inference accelerator attached. It is your responsibility to ensure that you are using the latest

Elastic Inference accelerator version. For more setup information, see Setting Up to Launch Amazon EC2 with Elastic Inference and Using AWS Deep Learning Containers With Elastic Inference.

Using CloudWatch Metrics to Monitor Elastic Inference

You can monitor your Elastic Inference accelerators using Amazon CloudWatch, which collects metrics about your usage and performance. Amazon CloudWatch records these statistics for a period of two weeks. You can access historical information and gain a better perspective of how your service is performing.

By default, Elastic Inference sends metric data to CloudWatch in 5-minute periods.

For more information, see the Amazon CloudWatch User Guide.

Topics

Elastic Inference Metrics and Dimensions

The client instance connects to one or more Elastic Inference accelerators through a PrivateLink endpoint. The client instance then inspects the input model's operators. If there are any operators that cannot run on the Elastic Inference accelerator, the client code partitions the execution graph. Only subgraphs with supported operators are loaded and run on the accelerator. The rest of the subgraphs run on the client instance. In the case of graph partitioning, each inference call on the client instance can result in multiple inference requests on an accelerator. This happens because evaluating each subgraph on the accelerator requires a separate inference call. Some CloudWatch metrics collected on the accelerator give you subgraph metrics and are called out accordingly.

Metrics are grouped first by the service namespace, then by the various dimension combinations within each namespace. You can use the following procedures to view the metrics for Elastic Inference.

To view metrics using the CloudWatch console

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. If necessary, change the Region. From the navigation bar, select the region where Elastic Inference resides. For more information, see Regions and Endpoints.
3. In the navigation pane, choose **Metrics**.
4. Under **All metrics**, select a metrics category, and then scroll down to view the full list of metrics.

To view metrics (AWS CLI)

- At a command prompt, enter the following command:

```
aws cloudwatch list-metrics --namespace " AWS/ElasticInference "
```

CloudWatch displays the following metrics for Elastic Inference.

Metric	Description
AcceleratorHealthCheckFailed	Reports whether the Elastic Inference accelerator has passed a status health check in the last minute. A value of zero (0) indicates that the status check passed. A value of one (1) indicates a status check failure. Units: Count
ConnectivityCheckFailed	Reports whether connectivity to the Elastic Inference accelerator is active or has failed in the last minute. A value of zero (0) indicates that a connection from the client instance was received in the last minute. A value of one (1) indicates that no connection was received from the client instance in the last minute. Units: Count
AcceleratorMemoryUsage	The memory of the Elastic Inference accelerator used in the last minute. Units: Bytes
AcceleratorUtilization	The percentage of the Elastic Inference accelerator used for computation in the last minute. Units: Percent
AcceleratorTotalInferenceCount	The number of inference requests reaching the Elastic Inference accelerator in the last minute. The requests represent the total number of separate calls on all subgraphs on the Elastic Inference accelerator. Units: Count
AcceleratorSuccessfulInferenceCount	The number of successful inference requests reaching the Elastic Inference accelerator in the last minute. The requests represent the total number of separate calls on all subgraphs on the Elastic Inference accelerator. Units: Count
AcceleratorInferenceWithClientErrorCount	The number of inference requests reaching the Elastic Inference accelerator in the last minute that resulted in a 4xx error. The requests represent the total number of separate calls on all subgraphs on the Elastic Inference accelerator. Units: Count
AcceleratorInferenceWithServerErrorCount	The number of inference requests reaching the Elastic Inference accelerator in the last minute that resulted in a 5xx error. The requests

Metric	Description
	represent the total number of separate calls on all subgraphs on the Elastic Inference accelerator. Units: Count

You can filter the Elastic Inference data using the following dimensions.

Dimension	Description
ElasticInferenceAcceleratorId	This dimension filters the data by the Elastic Inference accelerator.
InstanceId	This dimension filters the data by instance to which the Elastic Inference accelerator is attached.

Creating CloudWatch Alarms to Monitor Elastic Inference

You can create a CloudWatch alarm that sends an Amazon SNS message when the alarm changes state. An alarm watches a single metric over a time period that you specify. It sends a notification to an SNS topic based on the value of the metric relative to a given threshold. This takes place over a number of time periods.

For example, you can create an alarm that monitors the health of an Elastic Inference accelerator. It sends a notification when the Elastic Inference accelerator fails a status health check for three consecutive 5-minute periods.

To create an alarm for Elastic Inference accelerator health status

1. Open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. In the navigation pane, choose **Alarms**, **Create Alarm**.
3. Choose **Amazon EI Metrics**.
4. Select the **Amazon EI** and the **AcceleratorHealthCheckFailed** metric and choose **Next**.
5. Configure the alarm as follows, and then choose **Create Alarm**:

   - Under **Alarm Threshold**, enter a name and description. For **Whenever**, choose **=>** and enter **1**. For the consecutive periods, enter **3**.
   - Under **Actions**, select an existing notification list or choose **New list**.
   - Under **Alarm Preview**, select a period of 5 minutes.

Troubleshooting

The following are common Amazon Elastic Inference errors and troubleshooting steps.

Topics

Issues Launching Accelerators

Ensure that you are launching in a Region where Elastic Inference accelerators are available. For more information, see the Region Table.

Resolving Configuration Issues

If you launched your instance with the Deep Learning AMI (DLAMI), run `python ~/anaconda3/bin/EISetupValidator.py` to verify that the instance is correctly configured. You can also download the EISetupValidator.py script and run `'python EISetupValidator.py`.

Issues Running AWS Batch

Running an AWS Batch job from an Amazon EC2 instance with Elastic Inference may throw the following error:

```
[Sat Nov 23 20:21:11 2019, 792775us] Error during accelerator discovery
[Sat Nov 23 20:21:11 2019, 792895us] Failed to detect any accelerator
[Sat Nov 23 20:21:11 2019, 792920us] Warning - Preconditions not met for reaching
 Accelerator
```

To fix this issue, unset the `ECS_CONTAINER_METADATA_URI` environment variable for the processes using Elastic Inference enabled frameworks. The `ECS_CONTAINER_METADATA_URI` environment variable is automatically set for containers launched as Amazon Elastic Container Service tasks. AWS Batch uses Amazon ECS to run containerized jobs. The following shows how to unset the `ECS_CONTAINER_METADATA_URI` variable.

```
env -u ECS_CONTAINER_METADATA_URI python script_using_tf_predictor_api.py
env -u ECS_CONTAINER_METADATA_URI amazonei_tensorflow_model_server
env -u ECS_CONTAINER_METADATA_URI python script_using_ei_mxnet.py
```

This does not unset `ECS_CONTAINER_METADATA_URI` globally. It only unsets it for the relevant processes, so unsetting it will not have any undesirable side-effects. Once `ECS_CONTAINER_METADATA_URI` is no longer set, Elastic Inference should work with AWS Batch.

Resolving Permission Issues

If you are unable to successfully connect to accelerators, verify that you have completed the following:

- Set up a Virtual Private Cloud (VPC) endpoint for Elastic Inference for the subnet in which you have launched your instance.
- Configure security groups for the instance and VPC endpoints with outbound rules that allow communications for HTTPS (Port 443). Configure the VPC endpoint security group with an inbound rule that allows HTTPS traffic.
- Add an IAM instance role with the **elastic-inference:Connect** permission to the instance from which you are connecting to the accelerator.
- Check CloudWatch Logs to verify that your accelerator is healthy. The Amazon EC2 instance details from the console contain a link to CloudWatch, which allows you to view the health of its associated accelerator.

Stop and Start the Instance

If your Elastic Inference accelerator is in an unhealthy state, stopping and starting it again is the simplest option. For more information, see Stopping and Starting Your Instances.

Warning
When you stop an instance, the data on any instance store volumes is erased. If you have any data to preserve on instance store volumes, make sure to back it up to persistent storage.

Troubleshooting Model Performance

Elastic Inference accelerates operations defined by frameworks like TensorFlow and MXNet. While Elastic Inference accelerates most:

- neural network
- math
- array manipulation
- control flow

operators, there are many operators that Elastic Inference does not accelerate. These include

- training-related operators
- input/output operators
- operators in contrib

When a model contains operators that Elastic Inference does not accelerate, the framework runs them on the instance. The frequency and location of these operators within a model graph can have an impact on the model's inference performance with Elastic Inference accelerators. If your model is known to benefit from GPU acceleration and does not perform well on Elastic Inference, contact AWS Support or amazon-ei-feedback@amazon.com.

Submitting Feedback

Contact AWS Support or send feedback to: amazon-ei-feedback@amazon.com.

Amazon Elastic Inference Error Codes

The Amazon Elastic Inference service manages the lifecycle of Elastic Inference accelerators and is accessible as an AWS PrivateLink endpoint service. The client instance (Amazon Elastic Compute Cloud (Amazon EC2), Amazon SageMaker or the Amazon Elastic Container Service (Amazon ECS) container instance) connects to an AWS PrivateLink endpoint to reach an Elastic Inference accelerator. The Elastic Inference version of the framework code includes an Elastic Inference client library (ECL) that is compatible with machine learning frameworks including TensorFlow, Apache MXNet, and PyTorch. ECL communicates with the Elastic Inference accelerator through AWS PrivateLink. The Elastic Inference-enabled framework running on the client instance maintains a persistent connection to the Elastic Inference accelerator via a keep-alive thread using ECL. You can see the health status of the accelerator on your Amazon CloudWatch metrics for Elastic Inference.

When you make the first inference call to an accelerator after you provision any service instance, it takes longer than subsequent infer calls. During this time, the Elastic Inference service sets up a session between the client instance and the Elastic Inference accelerator. The client code also inspects the model's operators. If there are any operators that cannot run on the Elastic Inference accelerator, the client code partitions the execution graph and only loads the subgraphs with operators that are supported on the accelerator. This implies that some of the subgraphs are run on the accelerator and the others are run locally. Any subsequent inference calls take less time to run because they use the already-initialized sessions. They also run on an Elastic Inference accelerator that has already loaded the model. If your model includes any operator that is not supported on Elastic Inference, the inference calls have higher latency. You can see CloudWatch metrics for the subgraphs that run on the Elastic Inference accelerator.

A list of these errors is provided in the following table. When you set up the Elastic Inference accelerators and make inference calls in the different components described, you might have errors that provide three comma-delimited numbers. For example `[21, 5, 28]`. Look up the third error code number (in the example, `28`), which is an ECL status code, in the table here to learn more. The first two numbers are internal error codes that help Amazon investigate issues and are represented with an `x` and `y` in the following table.

[x, y, ECL STATUS CODE]	Error Description
[x,y,1]	The Elastic Inference accelerator had an error. Retry the request. If this doesn't work, upgrade to a larger Elastic Inference accelerator size.
[x,y,6]	Failed to parse the model. First, update to the latest client library version and retry. If this doesn't work, contact `<amazon-ei-feedback@amazon.com>`.
[x,y,7]	This typically happens when Elastic Inference has not been set up correctly. Use the following resources to check your Elastic Inference setup: For SageMaker: Set Up to Use EI For Amazon EC2: Setting Up to Launch Amazon EC2 with Elastic Inference (p. 6) For Amazon ECS: Verify that your ecs-ei-task-role has been created correctly for the Amazon ECS container instance. For an example, see Setting up an ECS container instance for Elastic Inference in the blog post *Running Amazon Elastic Inference Workloads on Amazon ECS*.

[x, y, ECL STATUS CODE]	Error Description
[x,y,8]	The client instance or Amazon ECS task is unable to authenticate with the Elastic Inference accelerator. To configure the required permissions, see Configuring an Instance Role with an Elastic Inference Policy (p. 8).
[x,y,9]	Authentication failed during SigV4 signing. Contact <amazon-ei-feedback@amazon.com>.
[x,y,10]	Stop the client instance, then start it again. If this doesn't work, provision a new client instance with a new accelerator. For Amazon ECS, stop the current task and launch a new one.
[x,y,12]	Model not loaded on the Elastic Inference accelerator. Retry your inference request. If this doesn't work, contact <amazon-ei-feedback@amazon.com>.
[x,y,13]	An inference session is not active for the Elastic Inference accelerator. Retry your inference request. If this doesn't work, contact <amazon-ei-feedback@amazon.com>.
[x,y,15]	An internal error occurred on the Elastic Inference accelerator. Retry your inference request. If this doesn't work, contact <amazon-ei-feedback@amazon.com>.
[x,y,16]	An internal error occurred. Retry your inference request. If this doesn't work, contact <amazon-ei-feedback@amazon.com>.
[x,y,17]	An internal error occurred. Retry your inference request. If this doesn't work, contact <amazon-ei-feedback@amazon.com>.
[x,y,19]	Typically indicates there are no accelerators attached to the Amazon EC2 or SageMaker instance. Also the client is run outside of the Amazon ECS task container. Verify your setup according to Setting Up to Launch Amazon EC2 with Elastic Inference (p. 6). If this doesn't work, contact<amazon-ei-feedback@amazon.com>.
[x,y,23]	An internal error occurred. Contact <amazon-ei-feedback@amazon.com>.
[x,y,24]	An internal error occurred. Contact <amazon-ei-feedback@amazon.com>.
[x,y,25]	An internal error occurred. Contact <amazon-ei-feedback@amazon.com>.
[x,y,26]	An internal error occurred. Contact <amazon-ei-feedback@amazon.com>.

[x, y, ECL STATUS CODE]	Error Description
[x,y,28]	Configure your client instance and Elastic Inference AWS PrivateLink endpoint in the same subnet. If they already are in the same subnet, contact <amazon-ei-feedback@amazon.com>
[x,y,29]	An internal error occurred. Retry your inference request. If this doesn't work, contact <amazon-ei-feedback@amazon.com>.
[x,y,30]	Unable to connect to the Elastic Inference accelerator. Stop and restart the client instance. For Amazon ECS, stop the current task and launch a new one. If this doesn't work, contact <amazon-ei-feedback@amazon.com>.
[x,y,31]	Elastic Inference accelerator is out of memory. Use a larger Elastic Inference accelerator.
[x,y,32]	Tensors that are not valid were passed to the Elastic Inference accelerator. Using different input data sizes or batch sizes is not supported and might result in this error. You can either pad your data so all shapes are the same or bind the model with different shapes to use multiple executors. The latter option may result in out-of-memory errors because the model is duplicated on the accelerator.
[x,y,34]	An internal error occurred. Contact <amazon-ei-feedback@amazon.com>.
[x,y,35]	Unable to locate SSL certificates on the client instance. Check /etc/ssl/certs for the following certificates: ca-bundle.crt, Amazon_Root_CA_#.pem. If they are present, contact <amazon-ei-feedback@amazon.com>.
[x,y,36]	Your Elastic Inference accelerator is not set up properly or the Elastic Inference service is currently unavailable. First, verify that the accelerator has been set up correctly using Setting Up to Launch Amazon EC2 with Elastic Inference (p. 6) and retry your request after 15 seconds. If it still doesn't work, contact <amazon-ei-feedback@amazon.com>.
[x,y,39]	The model type that was received does not match the model type that was expected. For example, you sent an MXNet model when the accelerator was expecting a TensorFlow model. Stop and then restart the client instance to load the correct model and retry the request. For Amazon ECS, stop the current task and launch a new one with the correct model and retry the request.

[x, y, ECL STATUS CODE]	Error Description
[x,y,40]	An internal error occurred. Contact <amazon-ei-feedback@amazon.com>.
[x,y,41]	An internal error occurred. Contact <amazon-ei-feedback@amazon.com>.
[x,y,42]	Elastic Inference accelerator provisioning is in progress. Please retry your request in a few minutes.
[x,y,43]	This typically happens when the load model request took longer than the default client timeout. Retry the request to see if this resolves the issue. If it does not, contact <amazon-ei-feedback@amazon.com>.
[x,y,45]	The Elastic Inference accelerator state is unknown. Stop and restart the client instance. For Amazon ECS, stop the current task and launch a new one.
[x,y,46]	If you were unable to resolve the issue using the Elastic Inference error message provided, please contact <amazon-ei-feedback@amazon.com>. If applicable, please provide any Elastic Inference error codes and error messages that you received.

Document History for Developer Guide

The following table describes the documentation for this release of Amazon Elastic Inference.

- **API version: latest**
- **Latest documentation update:** April 16, 2019

update-history-change	update-history-description	update-history-date
Multiple Accelerators (p. 1)	Information on attaching multiple Elastic Inference accelerators was added to the setup guide.	December 2, 2019
Amazon Elastic Inference (p. 1)	Elastic Inference prerequisites and related info were added to the setup guide.	April 16, 2019

AWS glossary

For the latest AWS terminology, see the AWS glossary in the *AWS General Reference*.