Operationalizing Sagemaker Workflows Using AWS Sagemaker.

Introduction

This is a project to classify dog images using deep learning in aws using MLops best practices i.e optimization, security and cost.

Step 1: Initial setup, training and deployment

In this task we set up a new notebook instance for use in out project. The instance type selected for this project is `ml.t2.medium` . The choice of this instance type is because of low cost and fast launch characteristics. Below is the image of the set instance.

After setting up a notebook instance we the create a bucket and alter the first few cells of the `train_and_deploy-solution.ipynb` notebook. We run the cells inorder to upload the data to our created bucket ready for training. Below is a screen capture of the bucket with data uploaded.

Step 2: EC2 Training.

In this step we set up an EC2 instance and accomplish model training. We use a "Amazon Deep Learning AMI" for this. Here is a screenshot of the model after training with the EC2 instance.

Differences between `ec2train1.py` and `train_and_deploy-solution.ipynb` notebook ared;

1. We do not import sagemaker and its modules in the EC2 instance.
2. The EC2 script has no `argparse`

Step 3: Setting up a Lambda function

According to AWS documentation AWS Lambda is a serverless, event-driven compute service that lets you run code for virtually any type of application or backend service without provisioning or managing servers. You can trigger Lambda from over 200 AWS services and software as a service (SaaS) applications, and only pay for what you use.

Lambda runs your function only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the compute time that you consume—there is no charge when your code is not running.

Checkout the screen capture of the lambda function set below.

How it Works.

The function takes json input passes it to a configured input and passes out the result. Our function has `runtime` that is initiated by `boto3` and and `endpoint_name` that holds the name of the endpoint that the function will be invoking.

The function invokes the endpoint using runtime's `.invoke_endpoint()` method that takes `endpoint_name` as one of its parameters and returns a json data format as a result.

-

Step 4: Security and Testing.

When we created the lambda function in the previous step, its test results were erroneous. The error was due to insufficient permissions before invoking the specified endpoint. Inorder to rectify this we have to configure the correct security policiy in te 'IAM role' console to permit the invokation. We give our funtion **AmazonSagemakerFullAccess** inorder to succesifully invoke our specified endpoint.

Lambda test Success.

Security policy role adjustment.

Step 5: Concurrency and Auto-Scaling.

Concurency

```
Reserved concurrency – Reserved concurrency guarantees the maximum number of concurrent
instances for the function. When a function has reserved concurrency, no other function can use
that concurrency. There is no charge for configuring reserved concurrency for a function.

Provisioned concurrency – Provisioned concurrency initializes a requested number of execution
environments so that they are prepared to respond immediately to your function's invocations.
Note that configuring provisioned concurrency incurs charges to your AWS account.
```

In this project used reserved concurrency of default value 50. Observe below.

Autoscaling

Autoscaling is used, with *scale in* of 300 seconds and a *scale out* of 300 seconds. This is inorder to cut and minimize costs in when the resource is much more idle.

In [ ]:

In [ ]: