**Project 4: AWS - Operationalizing a ML Workflow - Dog breed Image Classification using AWS SageMaker and PyTorch**

This project uses a number of tools from AWS to setup and prepare a ML model for deployment in production.

The following steps need to be carried out as part of the project:

Step 1: Training and Deployment on SageMaker

Step 2: EC2 Training

Step 3: Lambda Function Setup

Step 4: Lambda Security and Testing

Step 5: Concurrency and auto-scaling

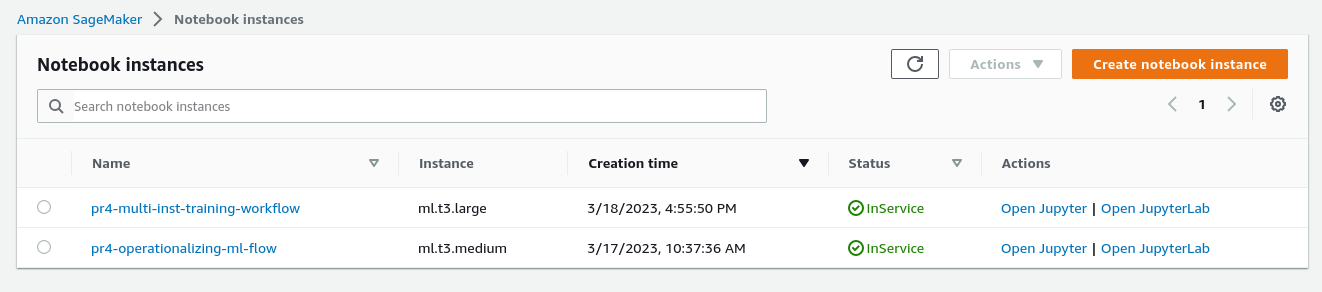
**Step 1: Training and Deployment on SageMaker**

A S3 bucket to store dog breed image files as input data was set up. A screenshot showing the set up is here:

Graphical user interface

Description automatically generated with medium confidence

First, a small instance ml.t3.medium was set up since it was one of the cheaper options, and single instance training was performed. Memory issues when re-running code for multi-instance training lead to choosing a larger instance for the next case – ml.t3.large. A screenshot of the running SageMaker instances is here:



Tuning was performed using instance type ml.g4dn.xlarge. This instance type, suggested as a default, was tried first and kept since it worked. Training jobs were run ml.m5.xlarge, also a suggested default, and it worked. Note that the number of epochs was limited and if one wanted to reduce tuning and training times and increase number of epochs then larger instances would need to be chosen.

**Single instance training details:**

hyperparameter tuning: pytorch-training-230318-1429

best values: batch\_size: 128, learning\_rate: 0.0034956970896215297'

training: dog-pytorch-2023-03-18-15-21-59-756

endpoint: pytorch-inference-2023-03-18-15-49-10-447

**Multi-instance training details:**

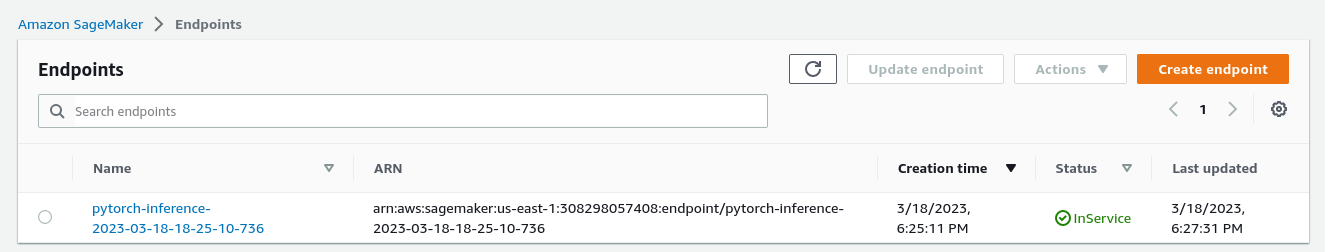
hyperparameter tuning: pytorch-training-230318-1709

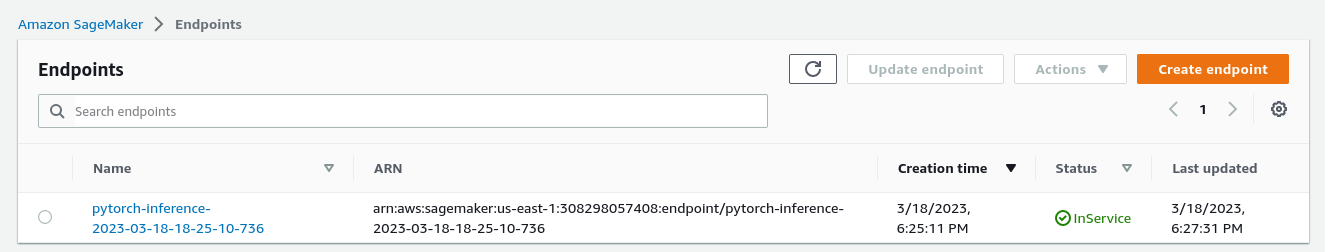
best values: batch\_size: 64, learning\_rate: 0.030884864336889317

training: dog-pytorch-2023-03-18-18-01-30-589

endpoint: pytorch-inference-2023-03-18-18-25-10-736

**Screenshots:** details of tuning and training jobs are in the “screenshots” folder, here are screenshots of deployed endpoints:

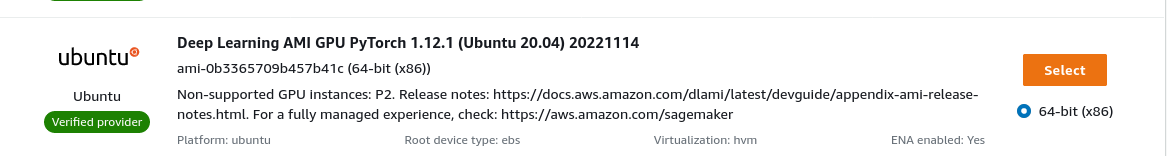


****

**Step 2: EC2 Training**

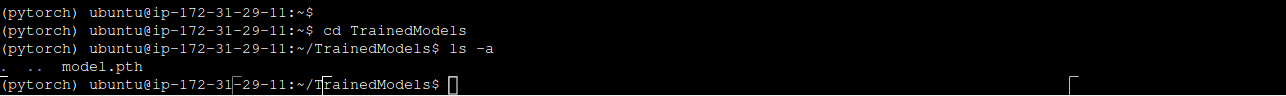
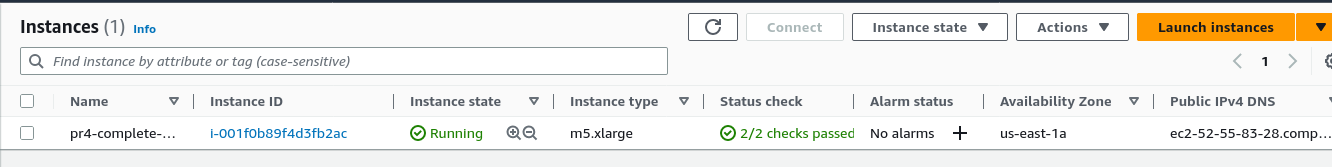
Instead of using a SageMaker instance for model training the model can also be trained using an EC2 instance. A script called “ec2train1.py” was supplied, which was adapted from the notebook train\_and\_deploy.ipynb to run on EC2. It was not possible to select ml.g4dn.xlarge as an instance due to resource limits on GPUs, and ml.m5.xlarge was selected instead.

Due to having no provision of GPUs it was not possible to launch the Amazon Linux DL AMI PyTorch 1.13.1. Ubuntu DL AMI was used instead:



The number of epochs was limited to 5 in the training process to conserve resources since the aim was to illustrate the use of EC2 for model training.

The model was successfully trained on the machine in a much shorter time than it took on SageMaker. Here are screenshots of the EC2 instance setup and of the model output:

In addition to that an Amazon Linux 2 AMI was also launched as another EC2 instance. A conda environment containing the required packages, such as pytorch, torchvision and others was set up, and the model was also successfully trained on that machine.

**Step 3: Lambda Function Setup**

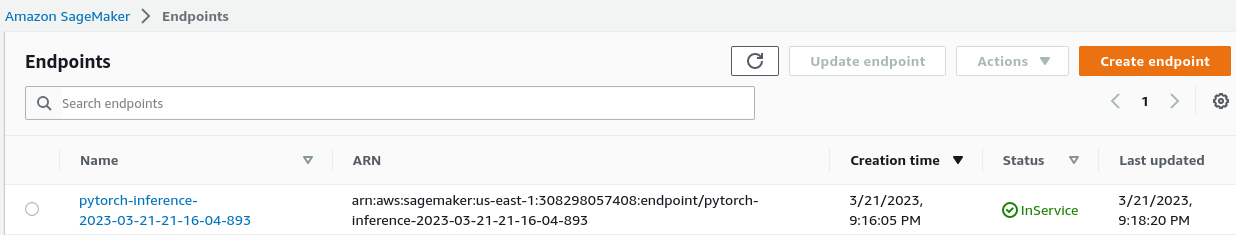
Having a model available to make inferences, we used a lambda function, using the script lambdafunction.py provided, and adapted for our model.

Lambda functions are serverless compute service which can be adapted and scaled to match demand. In the case used here the lambda function links inference input data with a deployed model and performs inferences via deployed endpoints.

The setup of the lambda function code is as follows:

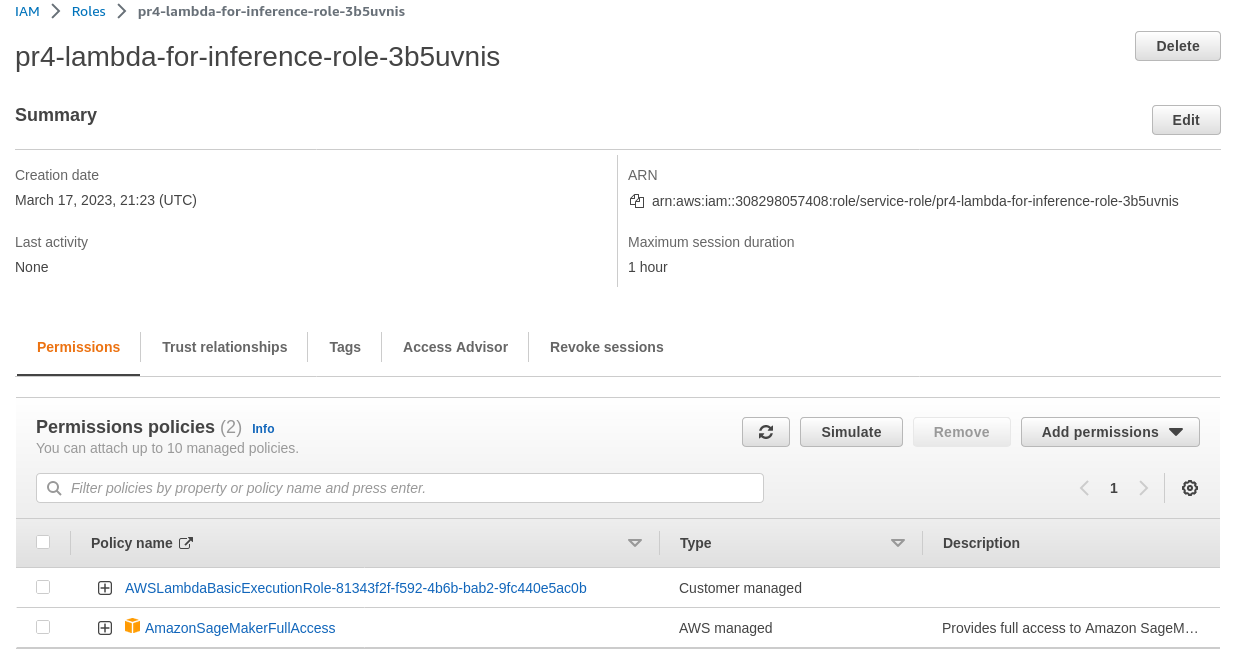
It imports packages, sets logging and then also used sagemaker runtime session using boto3 client. The endpoint to be used for inference is specified next. Then, the lambda\_handler function is defined as the main function, which takes two input arguments (event, context). The handler starts up SageMaker at runtime, and invokes the endpoint to make an inference for the function input arguments. The result of the inference is then returned as part of a json file by the handler.

For the lambda function to work, a model endpoint configuration needed to be re-deployed, which was done running a few lines of code to deploy the saved single-instance model in SageMaker notebook but this can also be done by simply creating an endpoint again directly from the saved endpoint configuration using the AWS SageMaker Console. Endpoint was:



S**tep 4: Lambda Security and Testing**

The lambda function then needed to be set up, and the name of the endpoint needed to be added to the lambdafunction.py script. The lambda function also needed to have a policy added to its role to be able to access AWS SageMaker. Full access was given for the lambda function:

It is worth pointing out though that it is good practice to:

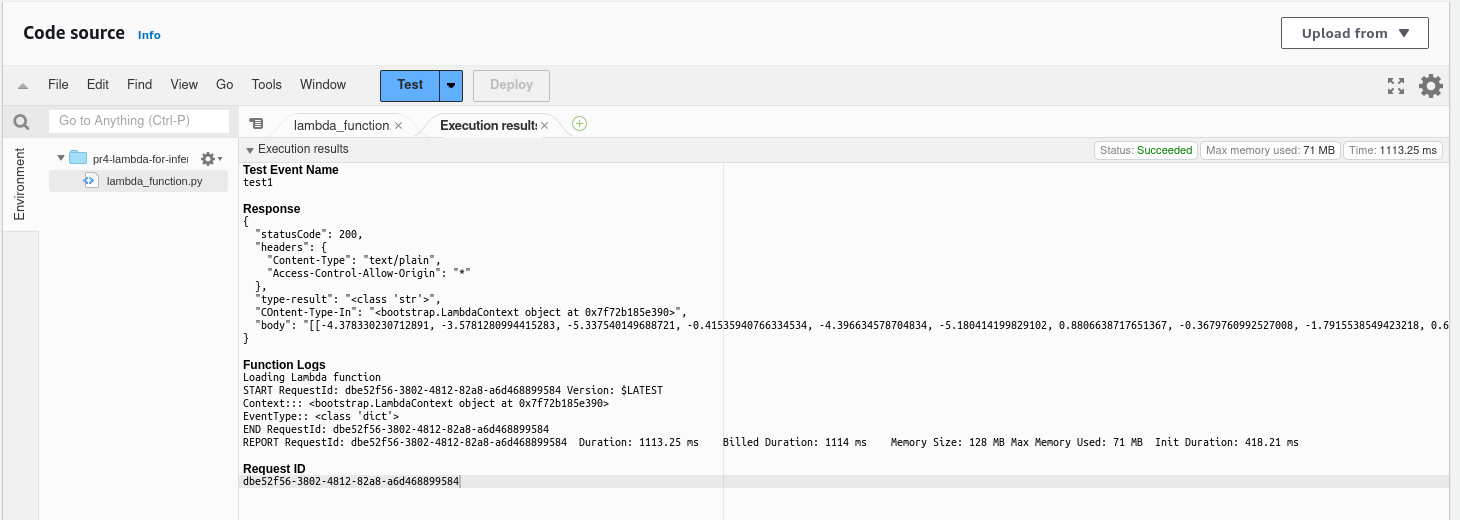
- limit role permissions to what is strictly necessary

- delete roles that are no longer active

This means that setting up a tailored policy instead would be more secure.

**Lambda Testing:** The lambda function needed to be tested to check it worked. Test input and the output results of the test are shown below:

**Lambda Test1 - input:** "https://s3.amazonaws.com/cdn-origin-etr.akc.org/wp-content/uploads/2017/11/20113314/Carolina-Dog-standing-outdoors.jpg"



**Lambda Test1 - output:**

{"statusCode": 200,

"headers": {

"Content-Type": "text/plain",

"Access-Control-Allow-Origin": "\*"

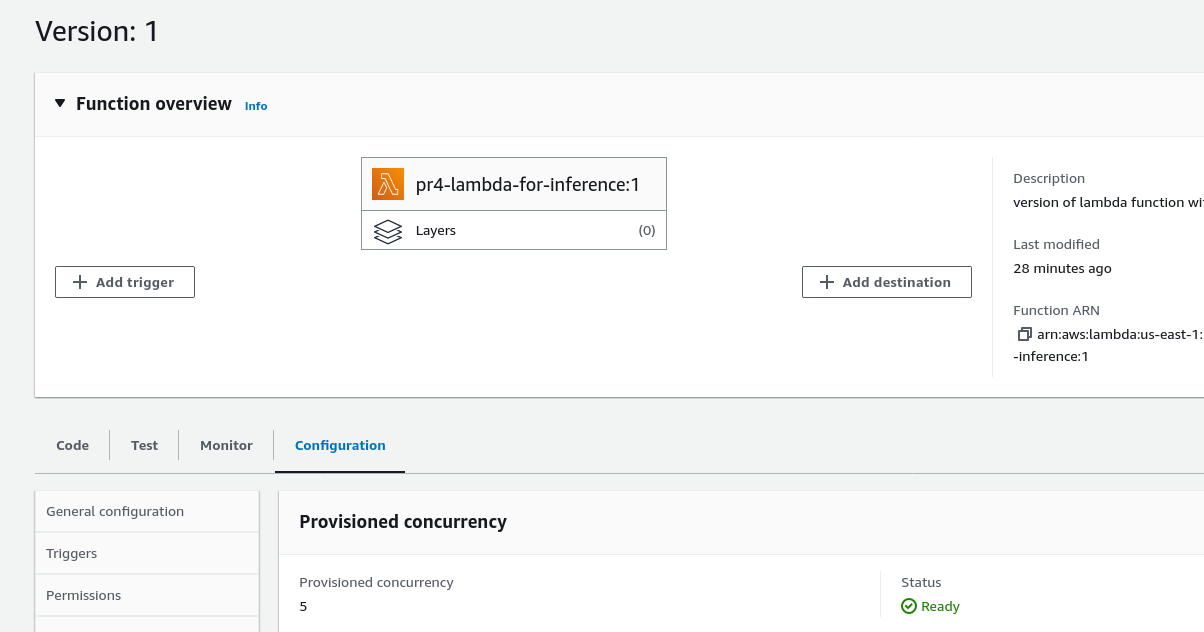
},

"type-result": "<class 'str'>",

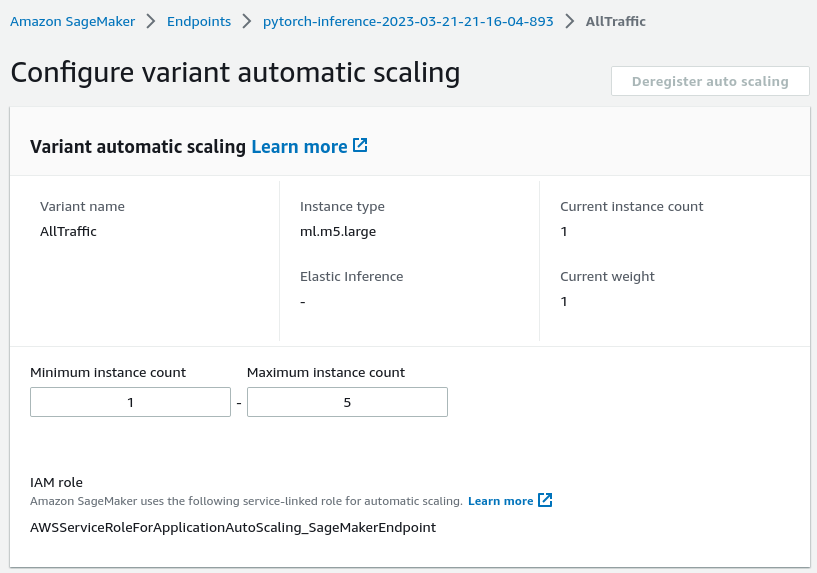
"COntent-Type-In": "<bootstrap.LambdaContext object at 0x7f72af192f50>",

"body": "[[-4.378330230712891, -3.5781280994415283, -5.337540149688721, -0.41535940766334534, -4.396634578704834, -5.180414199829102, 0.8806638717651367, -0.3679760992527008, -1.7915538549423218, 0.629810631275177, 0.7010319232940674, -3.4363083839416504, -1.472936987876892, -0.172869011759758, -3.5405614376068115, -2.037381887435913, -4.298142433166504, 1.1769710779190063, -5.919622421264648, 1.4042762517929077, -2.274519443511963, -1.2977726459503174, -4.813256740570068, -4.985188007354736, -0.8374140858650208, -6.628030776977539, -0.703967273235321, -1.5470912456512451, -3.9687652587890625, -3.1108357906341553, -2.4022412300109863, -0.5873315930366516, -5.076338291168213, -1.2679113149642944, -4.058257579803467, -4.155582904815674, -6.397622585296631, -3.911611318588257, -1.8280497789382935, -2.634091377258301, -3.4441540241241455, -1.1431111097335815, 0.11599510163068771, -2.3349504470825195, -0.871257483959198, -8.030259132385254, -2.2862918376922607, 0.4228699803352356, -2.121669054031372, -0.9147700667381287, -1.2141417264938354, -3.7804627418518066, -4.970803260803223, -2.4628045558929443, -4.684655666351318, -2.001572847366333, -1.4202160835266113, -6.756036758422852, 0.3442229926586151, -1.9132966995239258, -5.728144645690918, -6.5689473152160645, -3.559098720550537, -3.785905122756958, -1.2691550254821777, -4.098543643951416, -0.3214186429977417, -3.996256113052368, -0.13830961287021637, -0.9754683971405029, 1.2067346572875977, -2.930324077606201, -4.8984246253967285, -4.274216651916504, -4.476498603820801, -2.503283739089966, -3.986737012863159, -0.2968461215496063, -3.3737401962280273, -0.7913339138031006, 0.4878810942173004, -6.566642761230469, 0.3498344421386719, -1.9287998676300049, -5.601408004760742, -5.214552402496338, -3.5977296829223633, -4.7592668533325195, -5.209334373474121, -0.26540127396583557, -4.089554309844971, -3.534688711166382, -4.515682697296143, -5.6309309005737305, -4.654447555541992, -0.2431345283985138, -3.915580987930298, -0.050248511135578156, -5.438633441925049, -4.732776641845703, -5.053985595703125, -0.22484064102172852, -1.409592866897583, -5.626407623291016, -4.166421413421631, -4.242527961730957, -2.6510963439941406, -1.172747254371643, -0.9994869232177734, -0.10661790519952774, -1.6325105428695679, -1.285609245300293, -5.866470813751221, -4.832592487335205, -3.833681344985962, -0.7599971890449524, -4.145536422729492, -0.19713421165943146, -6.2762837409973145, 0.8528107404708862, -2.024597644805908, -2.057677745819092, -2.2029531002044678, -2.8904268741607666, -5.083116054534912, -1.9469395875930786, -3.0945799350738525, -0.48071593046188354, -1.176081895828247, -6.51193904876709, -3.3230302333831787, -1.8079626560211182, -3.352330207824707]]"}

**Step 5: Concurrency and Auto-scaling**

Provisioned concurrency can be set up for a lambda function to enable it to be able to respond to multiple requests at once with low latency. To demonstrate the principle provisioned concurrency was set at 5 concurrent requests. In practice it is good to have some historic endpoint access request data to establish a good provisioned concurrency value.

The deployed endpoint can also be adapted to be more flexible for responding to requests as auto-scaling can be introduced. A maximum instance count was set at 5 to illustrate the setup meaning that the endpoint can use a maximum of 5 instances if traffic increases:



One can also set a target value of when to start auto-scaling, and times (below in seconds) of how long to wait before scaling up and down. Analysis of typical historic endpoint traffic can be used together AWS pricing data to find an optimal combination in terms of latency and cost.

