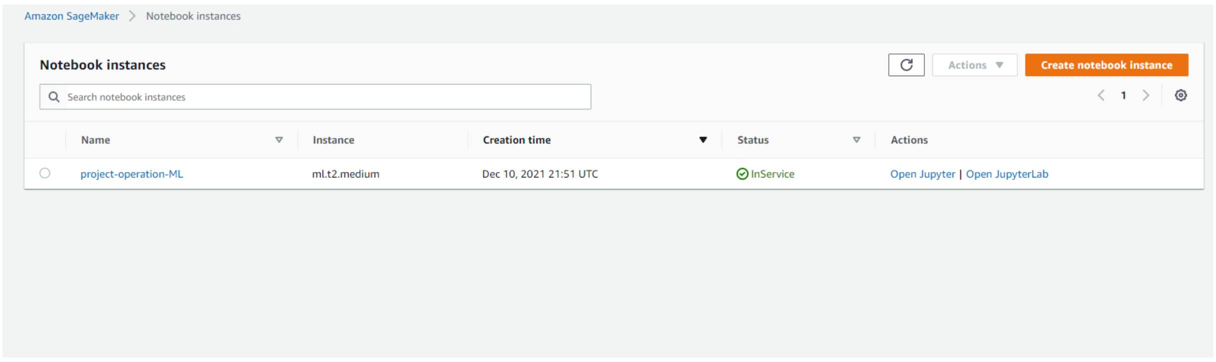
Operationsalizing an AWS ML project

The goal of this project was to train and deploy a dog breed classification model using all possible best practices related to optimization, cost and security.

Step 1: Training and deployment on Sagemaker

As model training won't be performed directly within the working notebook, a simple ml.t2.medium 5GB instance was selected for this project. The conda_python3 kernel was used to run the main Jupyter notebook.



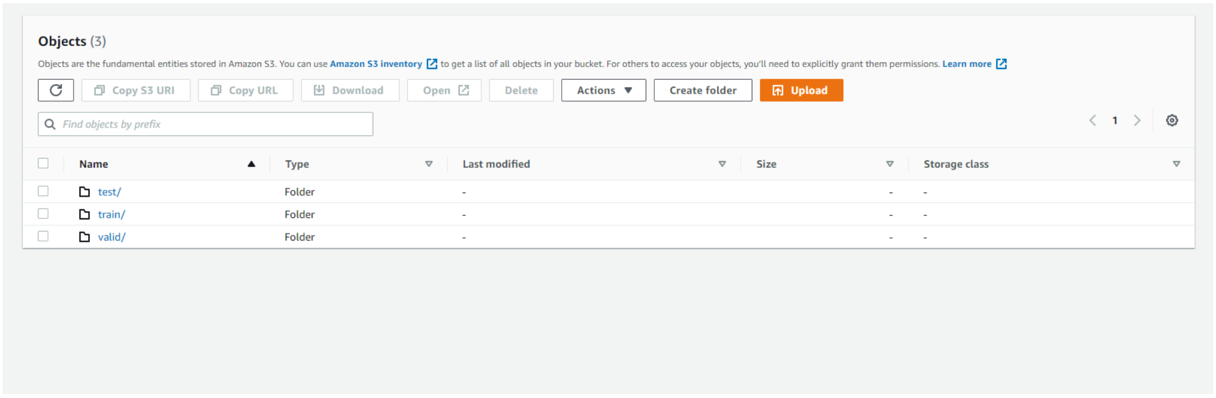
Amazon SageMaker > Notebook instances

Notebook instances

Name	Instance	Creation time	Status	Actions
project-operation-ML	ml.t2.medium	Dec 10, 2021 21:51 UTC	InService	Open Jupyter | Open JupyterLab

Figure 1: SageMaker notebook instances.

The training, validation and test datasets were first upload in S3.



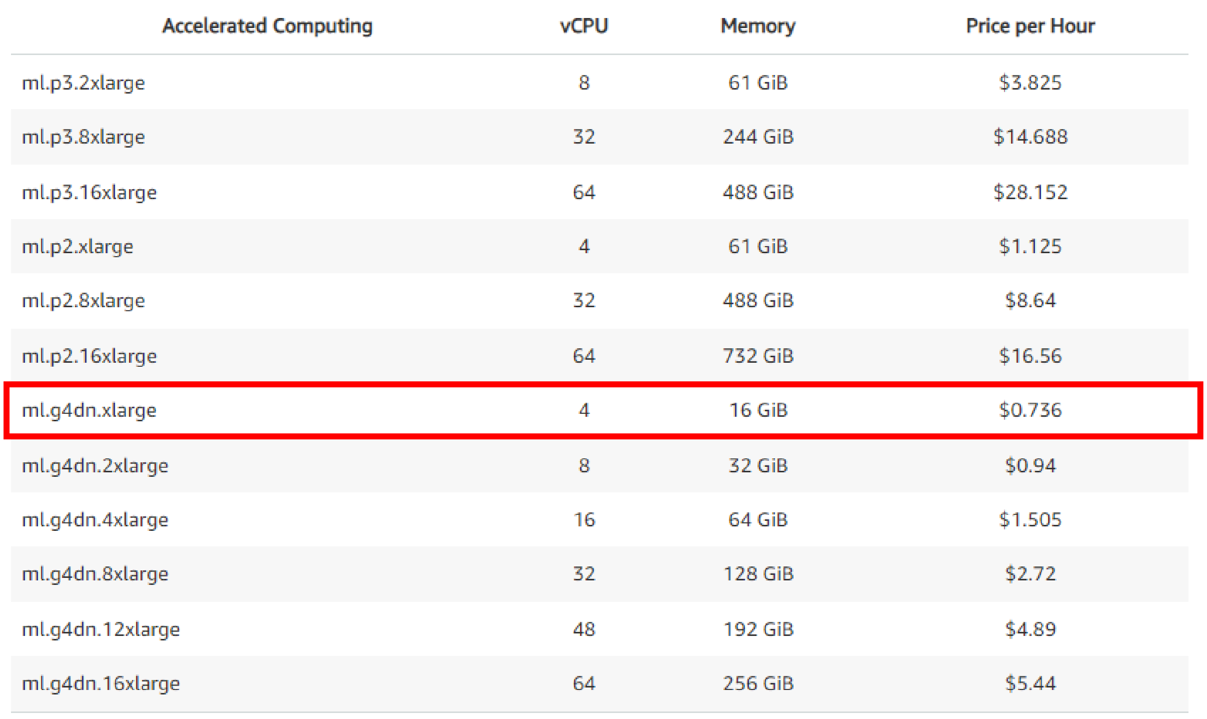
Objects (3)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
test/	Folder	-	-	-
train/	Folder	-	-	-
valid/	Folder	-	-	-

Figure 2: S3 console of the "dog-breed-project" directory.

In order to complete model training in a timely manner, an "ml.g4dn.xlarge" instance was selected. This instance type is the cheapest one within the "Accelerated computing" category.



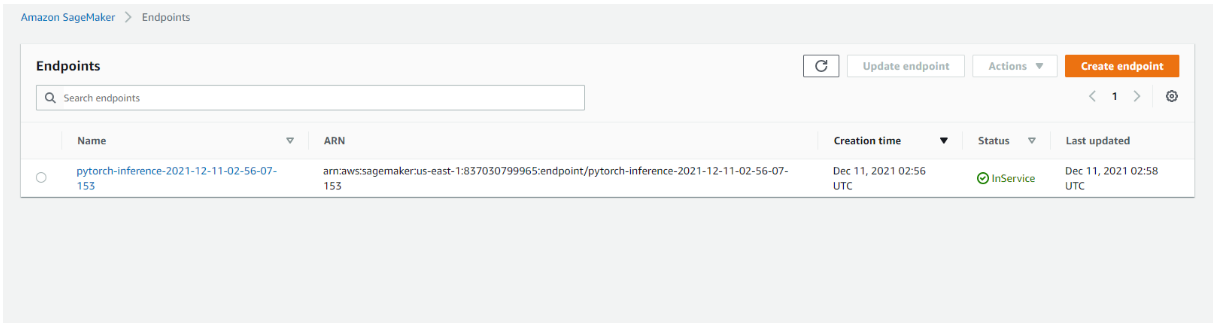
Accelerated Computing	vCPU	Memory	Price per Hour
ml.p3.2xlarge	8	61 GiB	$3.825
ml.p3.8xlarge	32	244 GiB	$14.688
ml.p3.16xlarge	64	488 GiB	$28.152
ml.p2.xlarge	4	61 GiB	$1.125
ml.p2.8xlarge	32	488 GiB	$8.64
ml.p2.16xlarge	64	732 GiB	$16.56
ml.g4dn.xlarge	4	16 GiB	$0.736
ml.g4dn.2xlarge	8	32 GiB	$0.94
ml.g4dn.4xlarge	16	64 GiB	$1.505
ml.g4dn.8xlarge	32	128 GiB	$2.72
ml.g4dn.12xlarge	48	192 GiB	$4.89
ml.g4dn.16xlarge	64	256 GiB	$5.44

Figure 3: Accelerated Computing EC2 instances pricing.

To ensure that the training algorithm can utilize the provided GPU, the hpo.py script has been modified. The model and the inputs/targets are now automatically passed to the available device. With those changes, the hyperparameter optimisation took only 9 minutes.

The model was retrained using the best-hyperparameters, this time, using a spot ml.m5.large instance to drastically reduce cost.

The model was then deployed on an ml.m5.large instance and invoked from the SageMaker notebook instance, no errors were found.



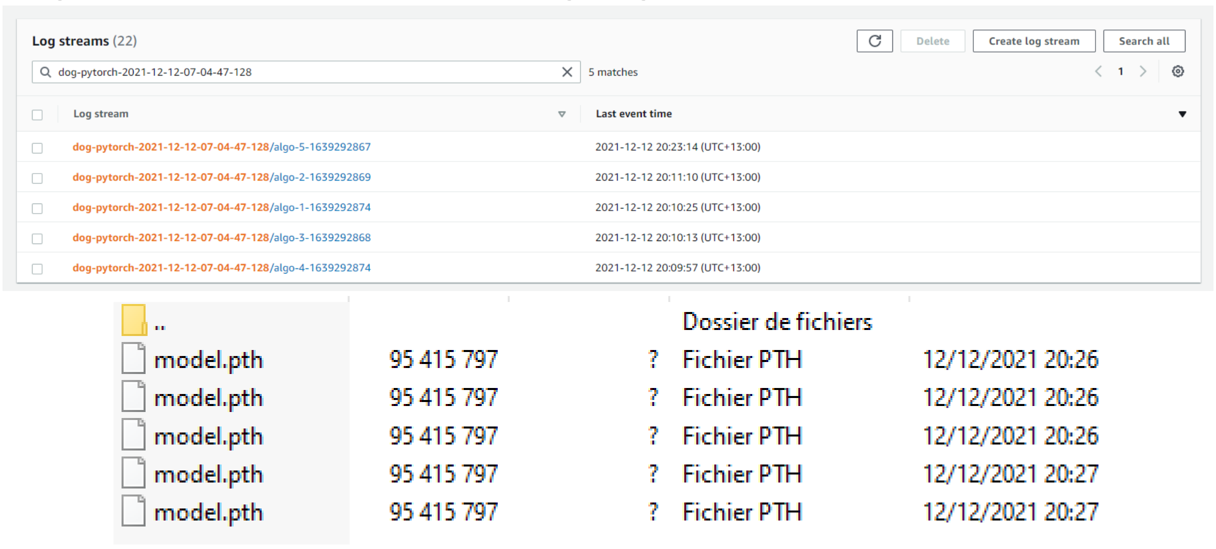
Name	ARN	Creation time	Status	Last updated
pytorch-inference-2021-12-11-02-56-07-153	arn:aws:sagemaker:us-east-1:837030799965:endpoint/pytorch-inference-2021-12-11-02-56-07-153	Dec 11, 2021 02:56 UTC	InService	Dec 11, 2021 02:58 UTC

Figure 4: Endpoints console.

The same model was then retrained using multi-instance training, more specifically, 5 ml.m5.large instances were used.

Training time with 1 instance: 22 minutes.
Training time with 5 instance: 24 minutes.

On a personal level, I am quite unsure of the benefits of just multiplying the number of training instances. All it does is to create 5 instances, perform the same training independently and save 5 different models without assembling/aggregating the model weights. As demonstrated in the following images:



dog-pytorch-2021-12-12-07-04-47-128

Log stream	Last event time
dog-pytorch-2021-12-12-07-04-47-128/algo-5-1639292867	2021-12-12 20:23:14 (UTC+13:00)
dog-pytorch-2021-12-12-07-04-47-128/algo-2-1639292869	2021-12-12 20:11:10 (UTC+13:00)
dog-pytorch-2021-12-12-07-04-47-128/algo-1-1639292874	2021-12-12 20:10:25 (UTC+13:00)
dog-pytorch-2021-12-12-07-04-47-128/algo-3-1639292868	2021-12-12 20:10:13 (UTC+13:00)
dog-pytorch-2021-12-12-07-04-47-128/algo-4-1639292874	2021-12-12 20:09:57 (UTC+13:00)

..			Dossier de fichiers	
model.pth	95 415 797	?	Fichier PTH	12/12/2021 20:26
model.pth	95 415 797	?	Fichier PTH	12/12/2021 20:26
model.pth	95 415 797	?	Fichier PTH	12/12/2021 20:26
model.pth	95 415 797	?	Fichier PTH	12/12/2021 20:27
model.pth	95 415 797	?	Fichier PTH	12/12/2021 20:27

Figure 5: Multi-instance training output.

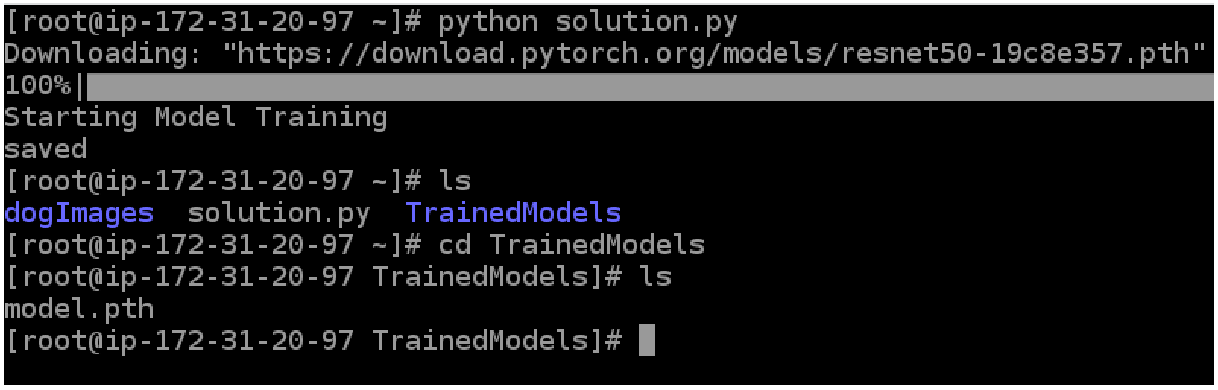
To perform proper data distributed training (and so increase the training speed) on Deep learning models, one should pick a multi-GPU instance(s) and follow the SageMaker documentation.

Step 2: EC2 Training

The same model was trained again, this time, using a manually deployed EC2 instance.
To be consistent with the step 1 of this project, a spot m5.large instance with an Deeplearning Amazon Linux 2 AMI was selected.

Unlike creating a training job from a notebook instance, which requires to provide the training data location and where the model's artifacts will be saved (as S3 paths), training a model inside an EC2 instance is similar to training a model on a local computer. Everything happen locally and the respective data/model folders must be manually created.

The following image shows the last bash commands used to train the model and access its artifacts.



```
[root@ip-172-31-20-97 ~]# python solution.py
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth"
100%|
Starting Model Training
saved
[root@ip-172-31-20-97 ~]# ls
dogImages  solution.py  TrainedModels
[root@ip-172-31-20-97 ~]# cd TrainedModels
[root@ip-172-31-20-97 TrainedModels]# ls
model.pth
[root@ip-172-31-20-97 TrainedModels]#
```

Figure 6: EC2 instance CLI.

BONUS: As an extra-challenge, I created an endpoint from this model artifact.

Step 1: Extract model.pth from the EC2 instance.
The model artifact was first moved to S3 using the following command:

```
aws s3 cp model.pth s3://sagemaker-us-east-1-836049028555593
```

Step 2: Transform to .tar.gz and upload to S3.
The model artifact was first download from S3.
Then, it was transformed in the appropriate format using the following command:

```
!tar -cvsf model.tar.gz model.pth
```

Finally, the compressed model was uploaded back to S3.

Step 3: Create and deploy a Pytorch model.
The same methodology as in the first step was used to deploy the model.

For more details on how the model has been transformed and deployed, look at the "Bonus section: Deploy the model trained on EC2" section of the notebook "train_and_deploy-solution".

Step 3: Lambda function setup

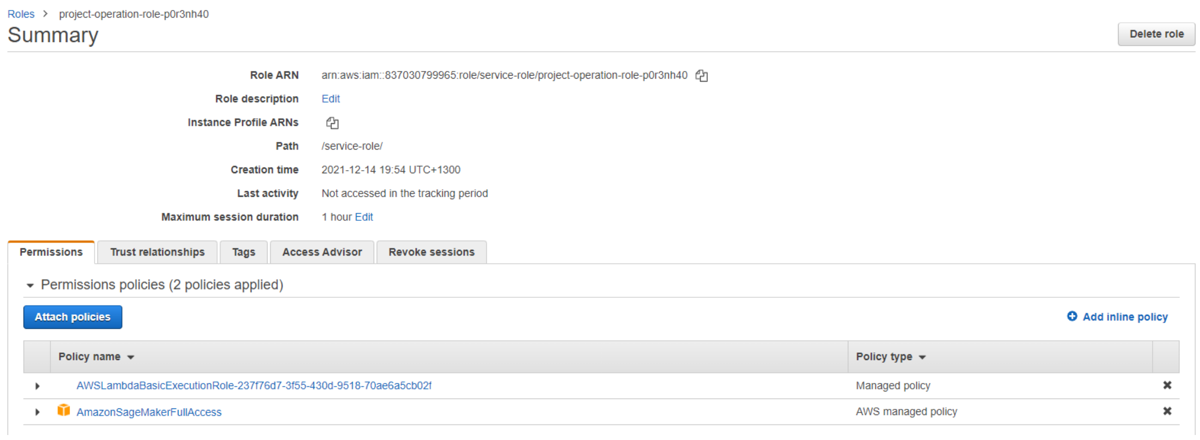
A lambda function was created in order to easily perform inference from other applications/services.

Lambda function do not automatically come with the sagemaker library, however, one can invoke a deployed endpoint via a boto3 runtime instance. First, the data is extracted from the received event, then, it is passed to the invoke_endpoint function with its type.

The prediction output is an utf-8 encoded json object. It is decoded, then send back as a json object within the "body" section of the function output.

Step 4: Security and testing

The sagemakerfullaccess policy was added to the function IAM role in order to let it access the deployed endpoint. I did not find a SageMaker policy that would only give access to deployed endpoints. Therefore, I do not think the security can be improved through just the Lambda IAM role.



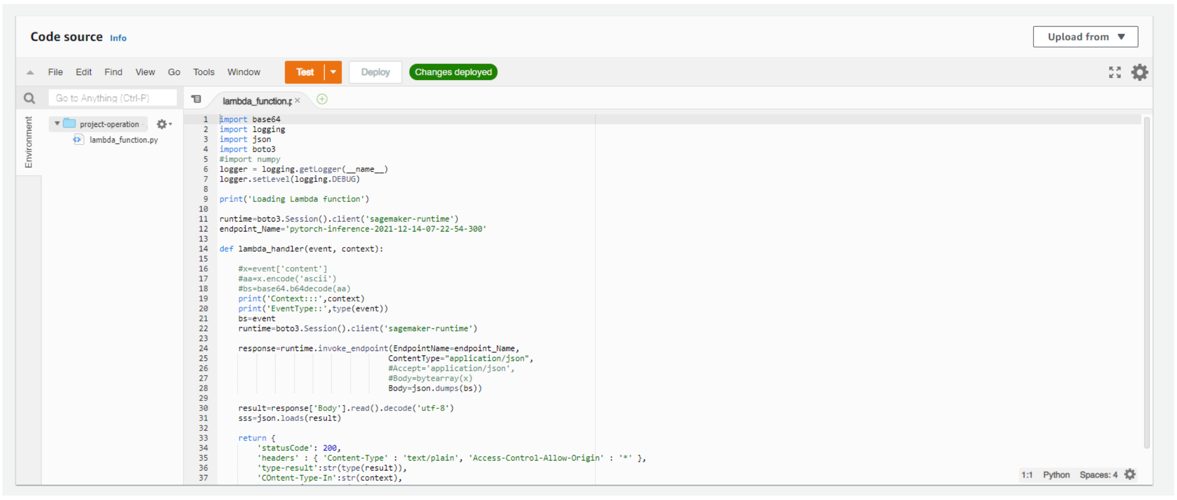
Role ARN	arn:aws:iam::837030799965:role/service-role/project-operation-role-p0r3nh40
Role description	Edit
Instance Profile ARNs	
Path	/service-role/
Creation time	2021-12-14 19:54 UTC+1300
Last activity	Not accessed in the tracking period
Maximum session duration	1 hour Edit

Permissions policies (2 policies applied)

Policy name	Policy type
AWSLambdaBasicExecutionRole-237f76d7-3f55-430d-9518-70ae6a5cb02f	Managed policy
AmazonSageMakerFullAccess	AWS managed policy

Figure 7: Lambda function role summary.

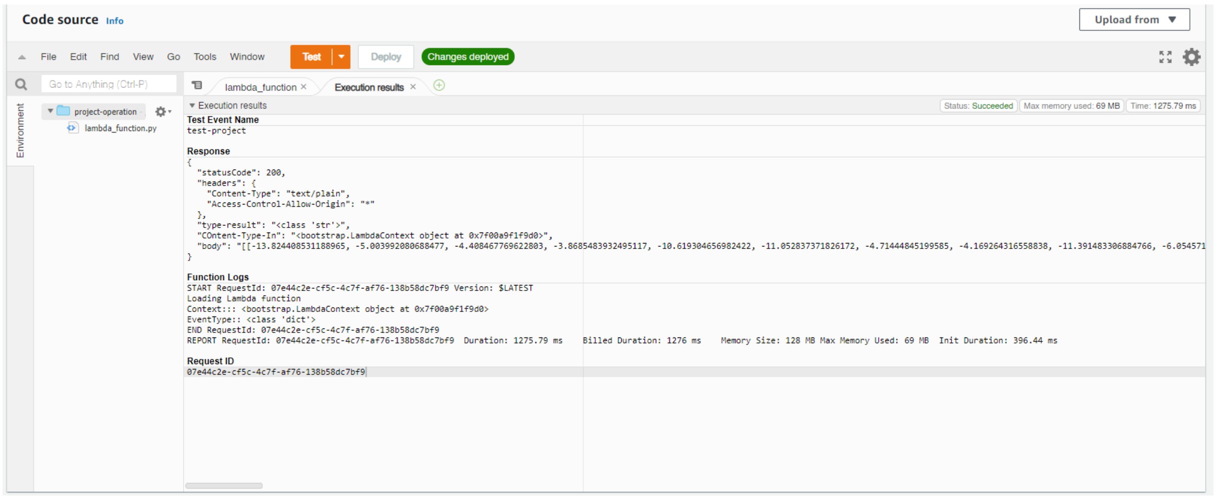
The lambda function was then written and deployed as shown in the following figure.



```python
import base64
import logging
import json
import boto3
#import numpy
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

print('Loading Lambda function')

runtime=boto3.Session().client('sagemaker-runtime')
endpoint_Name='pytorch-inference-2021-12-14-07-22-54-300'

def lambda_handler(event, context):

    #x=event['content']
    #aa=x.encode('ascii')
    #bs=base64.b64decode(aa)
    print('Context:::',context)
    print('EventType::',type(event))
    bs=event
    runtime=boto3.Session().client('sagemaker-runtime')

    response=runtime.invoke_endpoint(EndpointName=endpoint_Name,
                                     ContentType="application/json",
                                     #Accept='application/json',
                                     #Body=bytearray(x)
                                     Body=json.dumps(bs))

    result=response['Body'].read().decode('utf-8')
    sss=json.loads(result)

    return {
        'statusCode': 200,
        'headers' : { 'Content-Type' : 'text/plain', 'Access-Control-Allow-Origin' : '*' },
        'type-result':str(type(result)),
        'COntent-Type-In':str(context),
```

Figure 8: Lambda function code.

The test event did not raise any errors, and an array of 133 probabilities was obtained.



Test Event Name
```
test-project
```

Response
```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "text/plain",
    "Access-Control-Allow-Origin": "*"
  },
  "type-result": "<class 'str'>",
  "COntent-Type-In": "<bootstrap.LambdaContext object at 0x7f00a9f1f9d0>",
  "body": "[[-13.824408531188965, -5.003992080688477, -4.408467769622803, -3.8685483932495117, -10.619304656982422, -11.052837371826172, -4.71444845199585, -4.169264316558838, -11.391483306884766, -6.054571
}
```

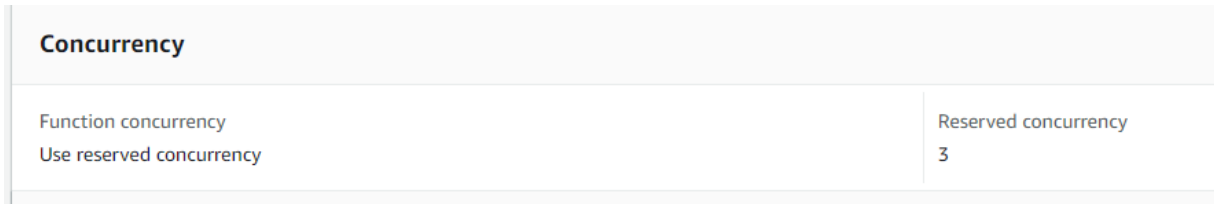
Function Logs
```
START RequestId: 07e44c2e-cf5c-4c7f-af76-138b58dc7bf9 Version: $LATEST
Loading Lambda function
Context::: <bootstrap.LambdaContext object at 0x7f00a9f1f9d0>
EventType:: <class 'dict'>
END RequestId: 07e44c2e-cf5c-4c7f-af76-138b58dc7bf9
REPORT RequestId: 07e44c2e-cf5c-4c7f-af76-138b58dc7bf9  Duration: 1275.79 ms    Billed Duration: 1276 ms    Memory Size: 128 MB Max Memory Used: 69 MB  Init Duration: 396.44 ms
```

Request ID
```
07e44c2e-cf5c-4c7f-af76-138b58dc7bf9
```

Figure 9: Lambda function execution result.

Step 5: Concurrency and auto-scaling

To reduce potential latency due to the lambda function, reserved concurrency was set up. Up to 3 functions can now be instantiated simultaneously in order to meet traffic demand.



Concurrency

Function concurrency	Reserved concurrency
Use reserved concurrency	3

Figure 10: Lambda concurrency parameters.

To reduce potential latency due to the model endpoint, a custom scaling policy was set up. Up to 3 instances can now be instantiated simultaneously in order to meet traffic demand. More specifically, an additional instance will be created if the average CPU usage is greater than 70% for more than 30 seconds.

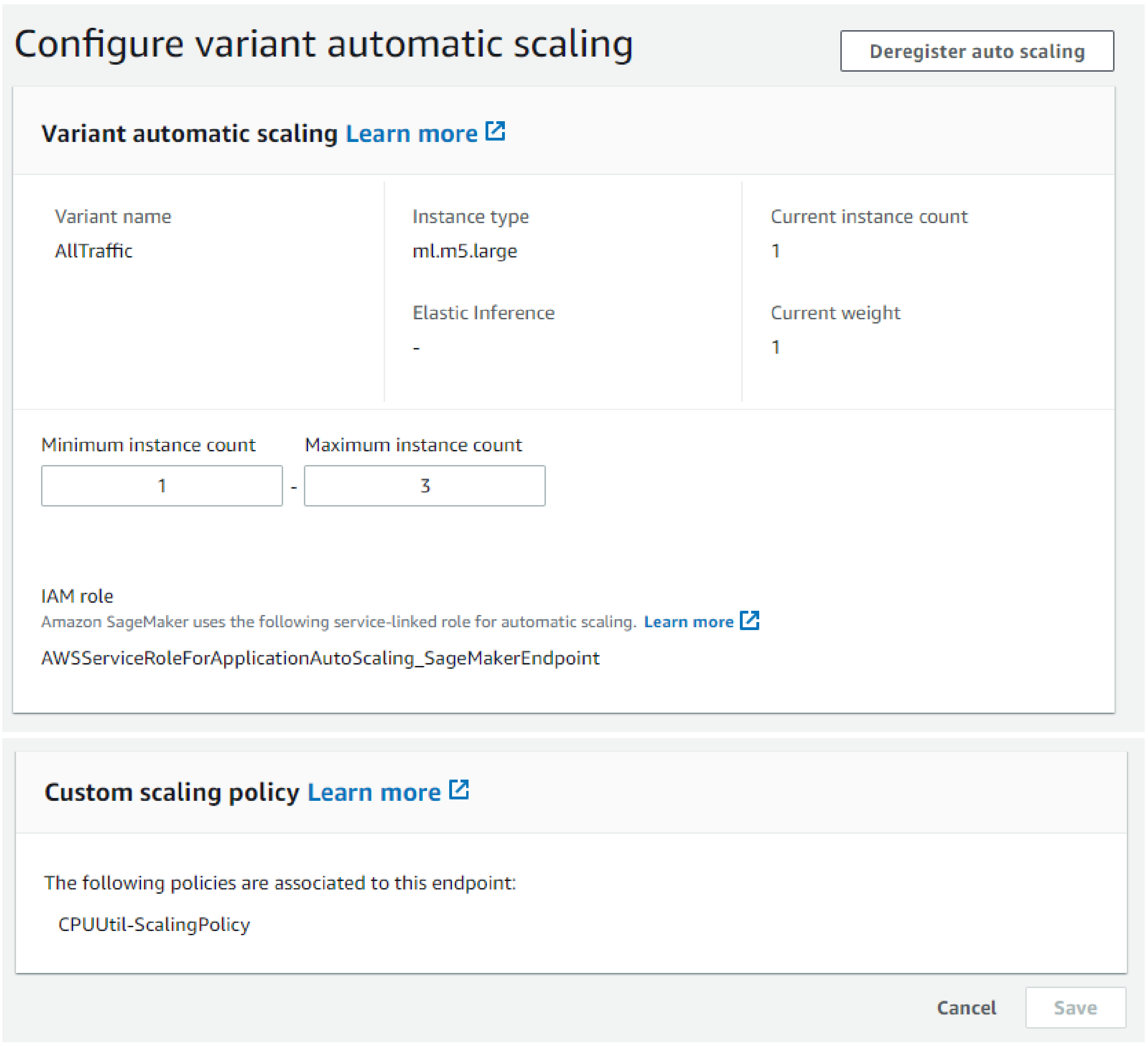
This custom policy was created from the notebook instance in two steps:

1) Register the endpoint and its associated variant.

2) Set up the scaling with the client.put_scaling_policy() function.

For more details on how the policy has been created and attached to the enpoint, look at the "Configure autoscaling policy" section (the last one) of the notebook "train_and_deploy-solution".

Here are screenshots of the endpoint console that prove the policy has been correctly attached:


Configure variant automatic scaling

Variant automatic scaling

Variant name: AllTraffic

Instance type: ml.m5.large

Elastic Inference: -

Current instance count: 1

Current weight: 1

Minimum instance count: 1 - Maximum instance count: 3

IAM role

Amazon SageMaker uses the following service-linked role for automatic scaling.

AWSServiceRoleForApplicationAutoScaling_SageMakerEndpoint

Custom scaling policy

The following policies are associated to this endpoint:

CPUUtil-ScalingPolicy

Figure 11: Endpoint console.

In a real world situation, one should adapt concurrency/auto-scalling configuration depending on the observed traffic. For example, if after several days I realise that my endpoint instance CPU utilisation tend to stagnate around 80%, I would consider raising the threshold to 90%, otherwise I would be paying another instance for nothing.

One should also look out for where the throughput bottleneck is. If the Lambda function is responsible for the latency, increasing the endpoint auto-scaling capabilities would induce useless additional cost.

Finally, the initial concurrency/auto-scalling configuration should be decided depending on the latency tolerance decided by the stakeholders. If the application is very latency sensitive, I would consider provisioned concurrency for the lambda function and a very short scale-out cooldown for the model endpoint. Only after having properly quantify the traffic patterns, I would consider cost optimistation/strategies.