Operationalizing ML workflow on Sagemaker

Step 1: Training and deployment on Sagemaker

- I chose ml.t3.medium as the running instance for my jupyter environment. I chose it because I have used it in my previous projects and and it's a good instance with 2x vCPUs and 4GB of RAM that has the capability to run my lines of codes in an acceptable speed and also has a fair amount of hourly costs.

- I created a bucket named "operationalizing-ml-bucket" to save my model and outputs of my training jobs in it. I chose the AWS Region as "US East (N. Virginia)", because my AWS account is now active to work in this region.

- I ran the cells up to 8th cell to fit my tuner to find the best hyperparameters. Then I can use these best hyperparameters to fit the estimator and train my model with them in the 16th cell.

- Using instance type "ml.m5.xlarge", it fails on training with batch size of 512. So I fitted the Tuner on the 8th cell using batch size in range of [32, 64, 128] (or I must use other stronger instances with more computing power that using them cost more!).

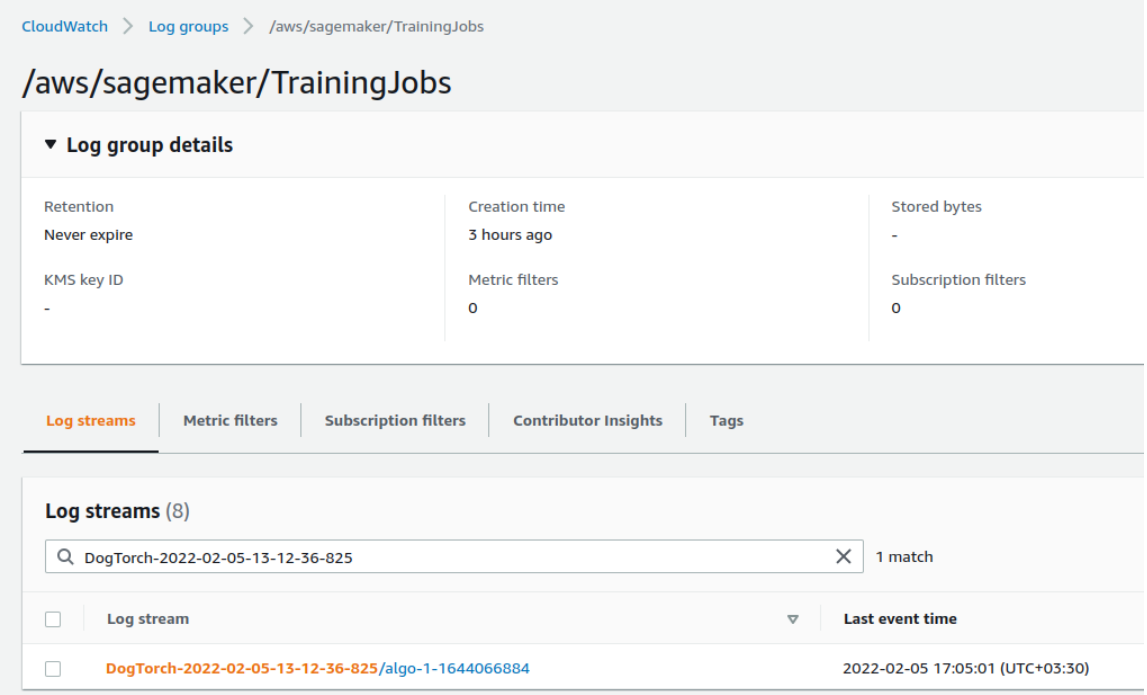
```
In [30]: from sagemaker.analytics import HyperparameterTuningJobAnalytics

exp = HyperparameterTuningJobAnalytics(
  hyperparameter_tuning_job_name='pytorch-training-220205-1201')

jobs = exp.dataframe()

jobs.sort_values('FinalObjectiveValue', ascending=0)
```

Out[30]:

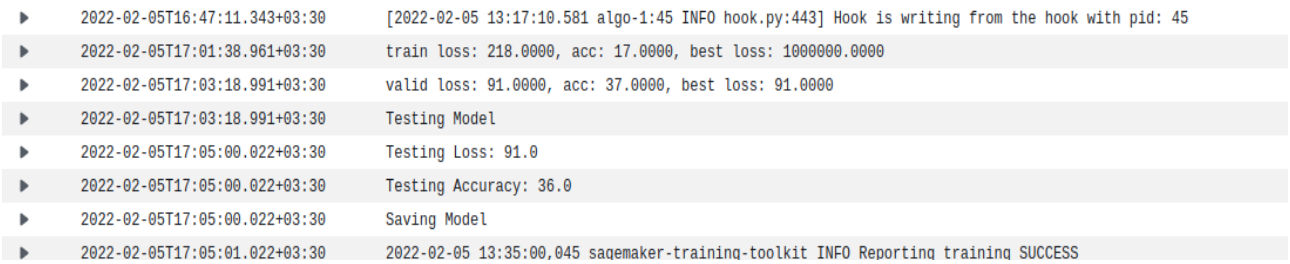
	batch_size	learning_rate	TrainingJobName	TrainingJobStatus	FinalObjectiveValue	TrainingStartTime	TrainingEndTime	TrainingElapsedTimeSeconds
0	"64"	0.002033	pytorch-training-220205-1201-002-1d2cce22	Completed	82.0	2022-02-05 12:04:25+00:00	2022-02-05 12:24:33+00:00	1208.0
1	"512"	0.066212	pytorch-training-220205-1201-001-cdaaa252	Failed	NaN	2022-02-05 12:03:42+00:00	2022-02-05 12:07:10+00:00	208.0

failed training job on batch size of 512

- I have done the training with best hyperparameters on the 16th cell and then I created an endpoint and got the response of the classification of an image from it. The training job using an instance lasts for 23 minutes. The logs that show the Testing Loss and Accuracy is shown here:



Single instance training log stream



Single instance training Testing Loss and Accuracy

The profiler report of this job is named “profiler-report-one-instance-training.html” and is existed in the “Sagemaker Run” folder. Here is the screenshot of getting the image and its predicted class:

```
import requests
#request_dict={ "url": "https://cdn1-www.cattime.com/assets/uploads/2011/12/file_2744_british-shorthair-460x290-460
request_dict={ "url": "https://s3.amazonaws.com/cdn-origin-etr.akc.org/wp-content/uploads/2017/11/20113314/Carolina

img_bytes = requests.get(request_dict['url']).content
type(img_bytes)
```

```
bytes
```

```
from PIL import Image
import io
Image.open(io.BytesIO(img_bytes))
```

show the test image

```
response=predictor.predict(img_bytes, initial_args={"ContentType": "image/jpeg"})
```

```
import json
response2=predictor.predict(json.dumps(request_dict), initial_args={"ContentType": "application/json"})
```

```
type(response2[0][0])
```

```
float
```

```
print(response2[0])
```

```
[-6.813735485076904, -2.314234733581543, -2.4530599117279053, -0.33914512395858765, -2.8606839179992676, -3.112207
8895568848, -2.220827102661133, -2.298513650894165, -5.290785789489746, -0.7099820375442505, 0.08821891248226166,
-3.1122288703918457, -2.582509994506836, 0.681918740272522, -2.8768880367279053, -1.4713103771209717, -7.354936122
894287, -3.962596893310547, -2.3961808681488037, 0.8976378440856934, -1.867553949356079, -0.058517903089523315, -
5.335705757141113, -4.2153425216674805, -3.741987705230713, -4.336663246154785, -2.1092474460601807, -4.2316789627
075195, -5.577977180480957, -1.662833333015442, -2.8958067893981934, -4.032334804534912, -5.103762626647949, -2.65
1531219482422, -6.886448860168457, -3.2887139320373535, -3.31431245803833, -2.390284538269043, -0.387097746133804
3, -4.738138198852539, -2.891908645629883, -1.413643479347229, -0.7013052701950073, -4.9064717292785645, -0.874709
6657752991, -6.665121078491211, -1.6694062948226929, -1.8764053583145142, -1.0472358465194702, -1.639080166816711
4, -4.352616310119629, -6.988555908203125, -4.930066108703613, -2.582667112350464, -4.983554363250732, -1.48646175
86135864, -5.0205488204956055, -2.18139910697937, -2.2798972129821777, -1.9204773902893066, -7.4354109764099912, -
6.472367763519287, -7.7789483070373535, -7.328060150146484, -1.6872206926345825, -7.342635154724121, -0.3452627658
843994, -5.989748954772949, -2.5607922077178955, -2.2447502613067627, 0.33085912466049194, -5.575009822845459, -3.
4133946895599365, -4.608648300170898, -3.188641309738159, -2.747288942337036, -6.104413032531738, -1.1470075845718
384, -4.789041996002197, -1.5678908824920654, 1.2766193151474, -5.887208938598633, -1.5808424949645996, 0.33793711
66229248, -6.66790771484375, -8.003376960754395, -2.3290553092956543, -6.040937900543213, -2.7056031227111816, -0.
603486180305481, -5.903278350830078, -4.089191913604736, -3.4990732669830322, -3.0671725273132324, -3.342291116714
4775, -3.792228937149048, 0.18990904092788696, -3.626432418823242, -4.9088945388793945, -5.2451276779174805, -7.32
3543071746826, -2.1710963249206543, -4.08787202835083, -4.750383377075195, -3.4247186183929443, -10.41490745544433
6, -2.985180377960205, -1.2440528869628906, -2.0715267658233643, -0.1557144969701767, -2.155757188796997, -1.09689
1164779663, -3.184666872024536, -3.3269786834716797, -4.468642711639404, -1.218206524848938, -6.368831157684326,
1.0161904096603394, -5.429952621459961, -1.9429376125335693, -2.106492042541504, -4.268537521362305, -3.5804700851
44043, -2.394174337387085, -5.249018669128418, -3.9260332584381104, -2.9964399337768555, -1.1166750192642212, -3.8
6635422706604, -6.922861099243164, -4.6553635597229, -3.5067830085754395, -3.050952196121216]
```

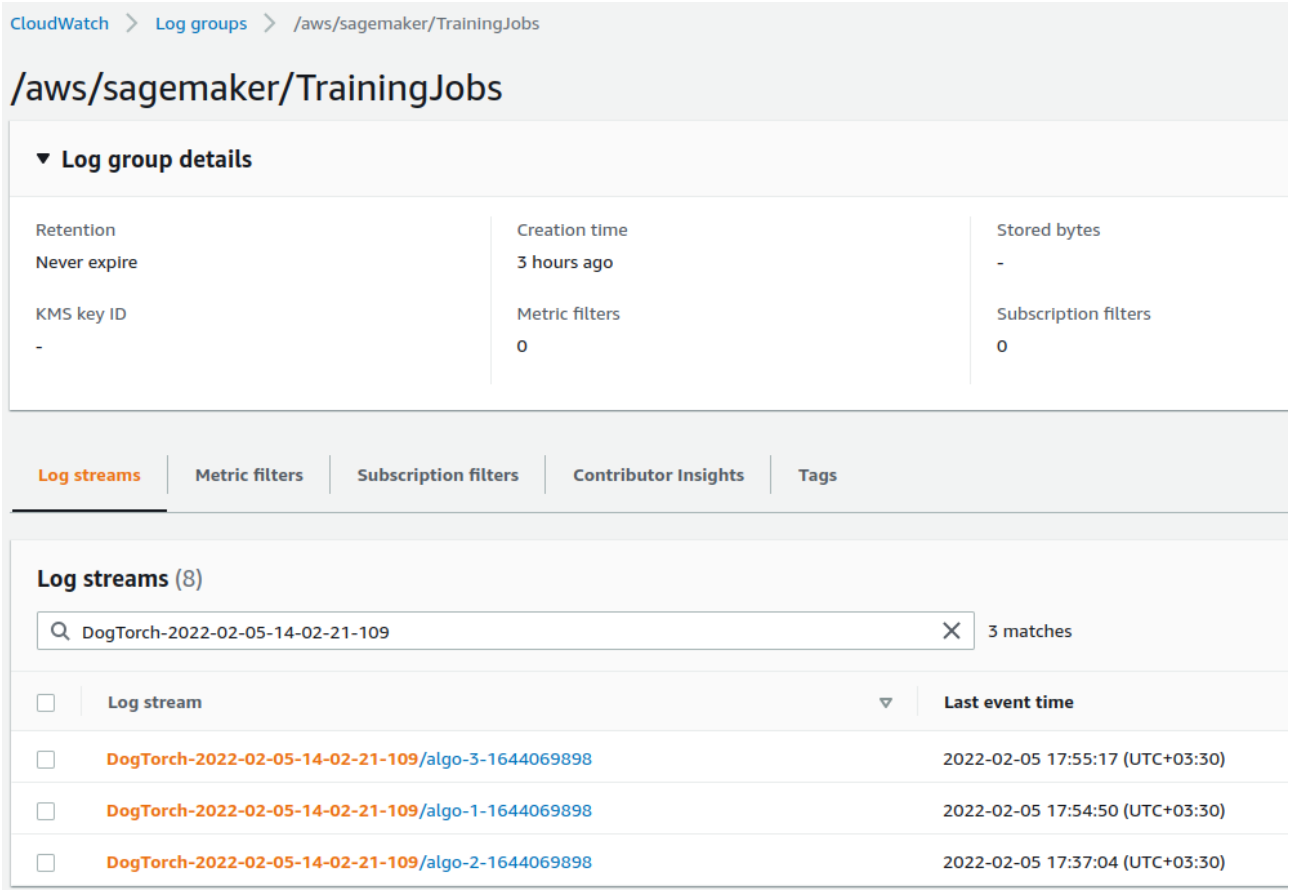
```
import torch
import numpy as np
print("The image is in class ", np.argmax(response, 1)[0])
```

```
The image is in class  80
```

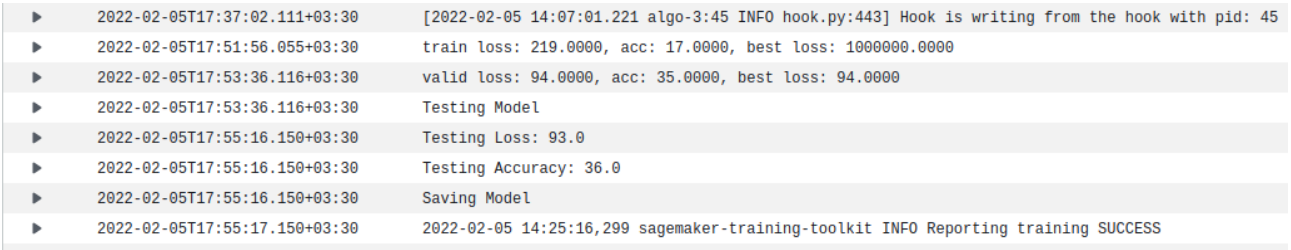
First model's result (model achieved on single instance training)

Writeup by Ensiyeh Raoufi

- I’ve done both single instance training and multi instance training. Because of the same S3 data distribution type that was automatically set on “FullyReplicated”, these two training jobs took almost the same time to train (23 minutes). However the first training job just trained one model and the second (multi_instance training job that is using 3 instances on mine), trained 3 models that are all different with Testing Loss and Accuracies. This resulted in different classification of the same previous image. The log streams of this training job and the logs that show the Testing Loss and Accuracy is shown here:



Multi instance training log streams



algo-3 Testing Loss and Accuracy

```
2022-02-05T17:37:04.270+03:30  [2022-02-05 14:07:04.068 algo-2:45 INFO hook.py:443] Hook is writing from the hook with pid: 45
2022-02-05T17:51:37.756+03:30  train loss: 216.0000, acc: 18.0000, best loss: 1000000.0000
2022-02-05T17:53:17.792+03:30  valid loss: 90.0000, acc: 38.0000, best loss: 90.0000
2022-02-05T17:53:17.792+03:30  Testing Model
2022-02-05T17:54:57.820+03:30  Testing Loss: 90.0
2022-02-05T17:54:57.820+03:30  Testing Accuracy: 37.0
2022-02-05T17:54:57.820+03:30  Saving Model
2022-02-05T17:54:58.820+03:30  2022-02-05 14:24:57,878 sagemaker-training-toolkit INFO Reporting training SUCCESS
```

algo-1 Testing Loss and Accuracy

```
2022-02-05T17:51:30.442+03:30  train loss: 214.0000, acc: 18.0000, best loss: 1000000.0000
2022-02-05T17:53:10.486+03:30  valid loss: 87.0000, acc: 38.0000, best loss: 87.0000
2022-02-05T17:53:10.486+03:30  Testing Model
2022-02-05T17:54:50.514+03:30  Testing Loss: 88.0
2022-02-05T17:54:50.514+03:30  Testing Accuracy: 38.0
2022-02-05T17:54:50.514+03:30  Saving Model
2022-02-05T17:54:50.514+03:30  2022-02-05 14:24:50,436 sagemaker-training-toolkit INFO Reporting training SUCCESS

No newer events at this moment. Auto retry paused. Resume
```

algo-2 Testing Loss and Accuracy

The profiler report of this job is named “profiler-report-multi-instance-training.html” and is existed in the “Sagemaker Run” folder. Here is the screenshot of the predicted class of this job’s best model:

```
In [75]: print(response2[0])

[-8.883174896240234, -3.883585214614868, -3.3491785526275635, -0.8631643056869507, -4.061888694763184, -4.97641849
5178223, -3.9073526859283447, -2.4597530364990234, -5.314611911773682, -1.096168875694275, -1.2172263860702515, -
3.8898112773895264, -2.162242889404297, -0.3464895784854889, -6.300234317779541, -4.7197418212890625, -4.081475257
873535, -1.3907724618911743, -4.068776607513428, 1.2770224809646606, -3.385958671569824, -1.8627971410751343, -4.5
886712074279785, -6.463838577270508, -3.7184276580810547, -7.741989612579346, -2.8654041290283203, -5.062293052673
34, -4.232786178588867, -1.1377196311950684, -4.064356327056885, -4.316063404083252, -6.894093036651611, -5.452674
388885498, -7.543737888336182, -6.777099609375, -5.5216169357299805, -3.3047757148742676, -2.3946547508239746, -6.
352634429931641, -3.000173330307007, -0.8704586625099182, 0.58835768699646, -5.024086952209473, -1.87663388252258
3, -7.184225559234619, -3.548203945159912, -2.7504050731658936, -0.5375791788101196, -1.0459372997283936, -6.34315
824508667, -5.563941478729248, -6.127568244934082, -2.835742950439453, -6.555300712585449, -2.0625016689300537, -
3.2226529121398926, -3.6401891708374023, -2.66184139251709, -2.266871452331543, -3.9799418449401855, -4.4138526916
50391, -6.682000160217285, -7.64078950881958, -4.996469497680664, -6.860738754272461, -1.833235740661621, -3.99236
96517944336, -5.349564552307129, -0.8875131011009216, -1.4309957027435303, -2.6444950103759766, -2.44771432876586
9, -3.581684112548828, -2.720899820327759, -4.854140758514404, -6.685421943664551, -1.4868019819259644, -4.6282877
922058105, -5.350033760070801, 0.7991379499435425, -3.797375440597534, -1.631448745727539, -1.4913371801376343, -
9.229757308959961, -6.2724080085754395, -0.8190008401870728, -6.773999214172363, -2.2026596069335938, -0.998181521
8925476, -6.440763473510742, -3.821532964706421, -4.463817596435547, -5.093514442443848, -5.168546676635742, -2.50
40743350982666, -3.003063678741455, -3.744699716567993, -2.578249216079712, -2.367434501647949, -5.58324861526489
3, -1.108337640762329, -3.2251598834991455, -3.333162784576416, -3.5010733604431152, -5.996979713439941, -2.060209
274291992, -1.1310911178588867, -2.781672477722168, -1.6836073398590088, -1.9843217134475708, -4.745124340057373,
-6.583361625671387, -2.5990943908691406, -4.295409679412842, -3.9293973445892334, -6.917832374572754, -2.038593769
0734863, -4.7636919021606445, 0.7190966606140137, -3.961813449859619, -4.331568717956543, -4.043440341949463, -5.0
49510478973389, -5.250204086303711, -5.686392307281494, -1.95003080368042, -0.29580986499786377, -6.71958017349243
2, -5.922369003295898, -4.479983806610107, -1.6971362829208374, -4.570926666259766]
```

```
In [76]: import torch
         import numpy as np
         print("The image is in class ", np.argmax(response, 1)[0])

         The image is in class  19
```

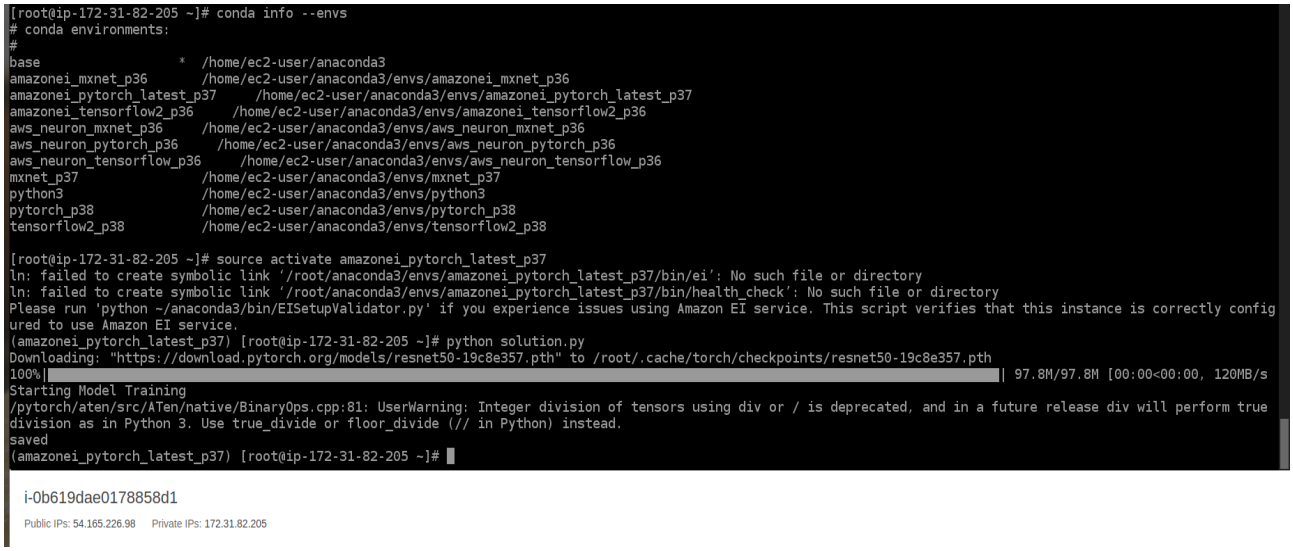
second model’s result on the same image (model achieved on multi instance training)

The screenshots of creating a Notebook Instance, creating a s3 bucket, the predictor’s results, and the other images related to this part (step 1) are existed in the “Sagemaker Run” folder. The names of the images explain the details of them.

Writeup by Ensiyeh Raoufi

Step 2: EC2 Training

- I navigated to the EC2 in my AWS account and then I clicked on “launch instances” to create an EC2 instance. I selected the "Amazon Deep Learning AMI" as an operating system for my EC2 instance, so that I can use its libraries. Since I’m doing pretty moderate computing here, I selected t2.micro which doesn’t have much memory that is enough for my job and also this type is eligible for the free tier.
- I created the solution.py file in my EC2 instance environment and pasted the lines of codes of the ec2train1.py file in it. Then I ran the solution.py and saved the model. Here is the screenshot of the done job:



EC2 training job and saving the model

- I found these differences between the code in ec2train1.py and the code I used in Step 1:
  - EC2 Instances are computing resources that are less expensive than SageMaker instances, but offer fewer managed services. They are not supporting advanced graphical options and you have to write and run all of your codes in an environment like a command window.
  - In the first step’s code we use a different script (hpo.py) as the entry point for training our model, because the model is training in a separate container. That’s why we are obliged to assign os.environ channel variables specified for sagemaker containers like

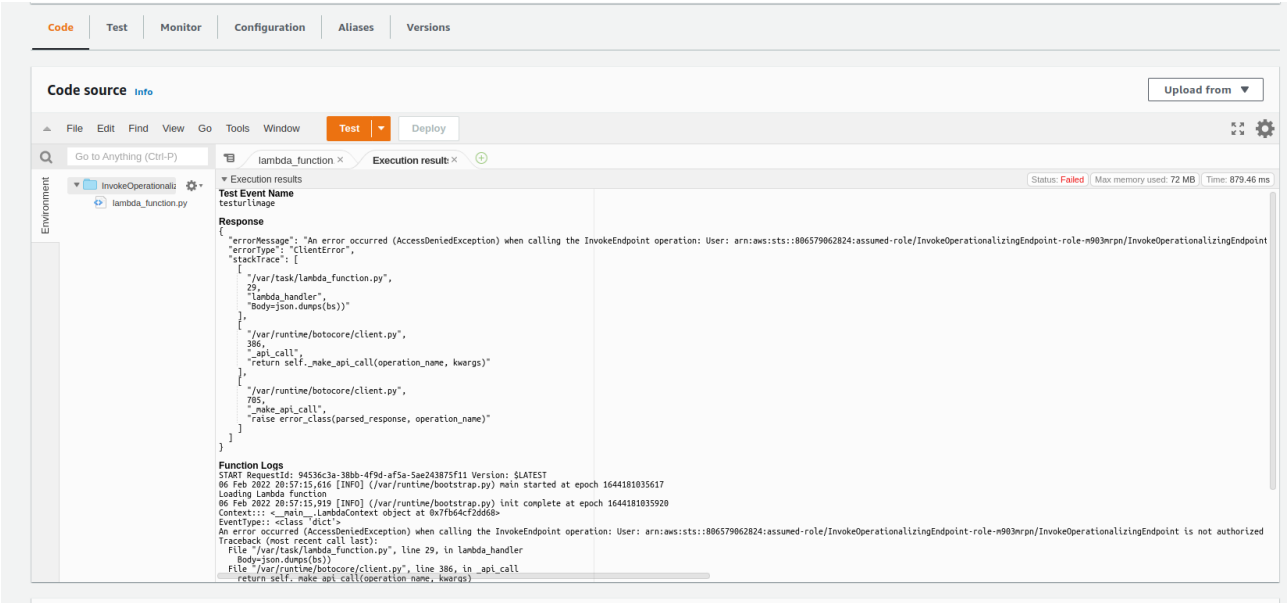
"SM_CHANNEL_TRAINING", and define these variables as arguments to our parser. We do this to interpreter be able to recognize the paths of the data inputs. While in the EC2 training we don't need to call the os.environ variables with channel names (training channel) and the path can be specify by only using os.path function.

- In the step 1, any hyperparameters (e.g. batch_size) provided by the training job will be passed by the interpreter to the entry point as script arguments, but in EC2 training all the hyperparameters are defined inside the ec2train1.py.
- In the first step one can use the Sagemaker debugger and profiler to plot the train/validation loss, check the vanishing gradient, GPU optimization, etc. during the training, while in EC2 training there are no many options for visual analysis.

The screenshots of creating an EC2 Instance, launching it, and the other images related to this part (step 2) are existed in the "EC2" folder. The names of the images explain the details of them.

Step 3: Lambda function setup

- We create a lambda function to invoke our endpoint when we call it. This lambda function will invoke our endpoint by giving the URL of an image in a JSON format as an input. After giving proper permissions to this function, it returns the output in a JSON format and by reading the result string using json.dumps, we could see a list of 133 float numbers each of which shows the tendency of the image to be in the related class (we have 133 dog breeds in this image dataset). The index of the maximum value of this list, leads us to the predicted class of the given image. At first and without attaching the necessary policies to our lambda function we will receive some errors like this:

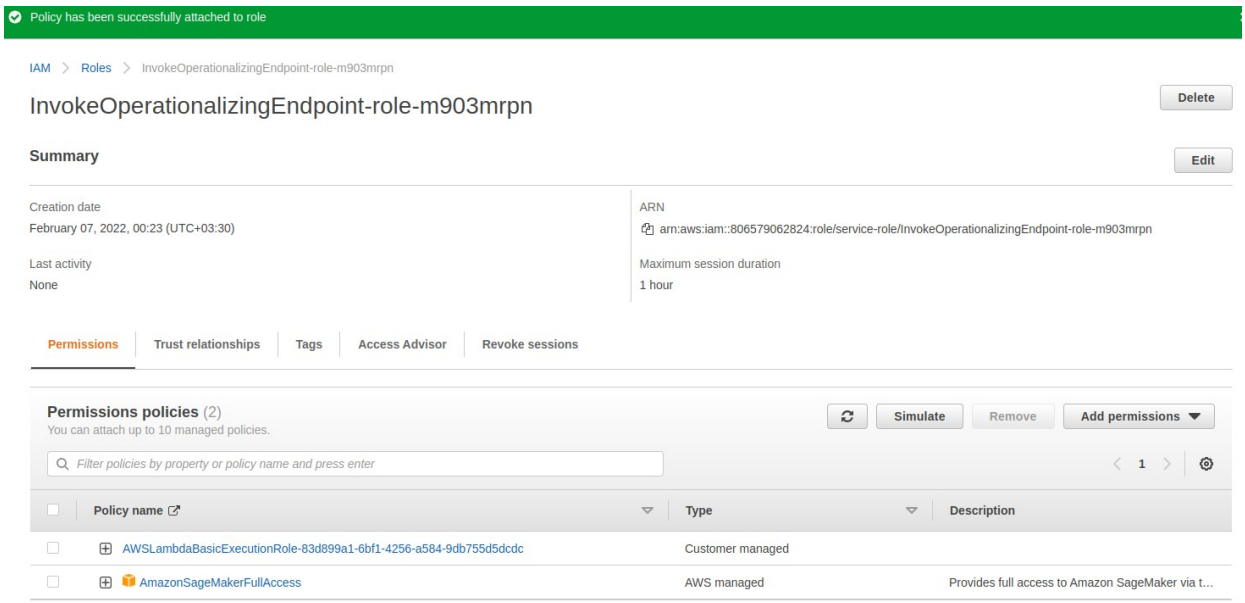


Authorization error

The screenshots of Lambda function codes, testing it, and the other images related to this part (step 3) are existed in the "Lambda" folder. The names of the images explain the details of them.

Step 4: Security and testing

- By adding AmazonSageMakerFullAccess policy to my lambda function role, it's now authorized to invoke my sagemaker endpoint. Although roles that have "FullAccess" policies attached may be too permissive and may lead to problems, but because here in this project I'm the only contributor and also the user of this product, there's no more concern to give this permission. After resolving the lambda function security issue, it's is able to return the result of the test image. Here is a screenshot of attaching proper policy to our function:



Attach policy to the lambda function

- The result of invoking my endpoint on the test image is:

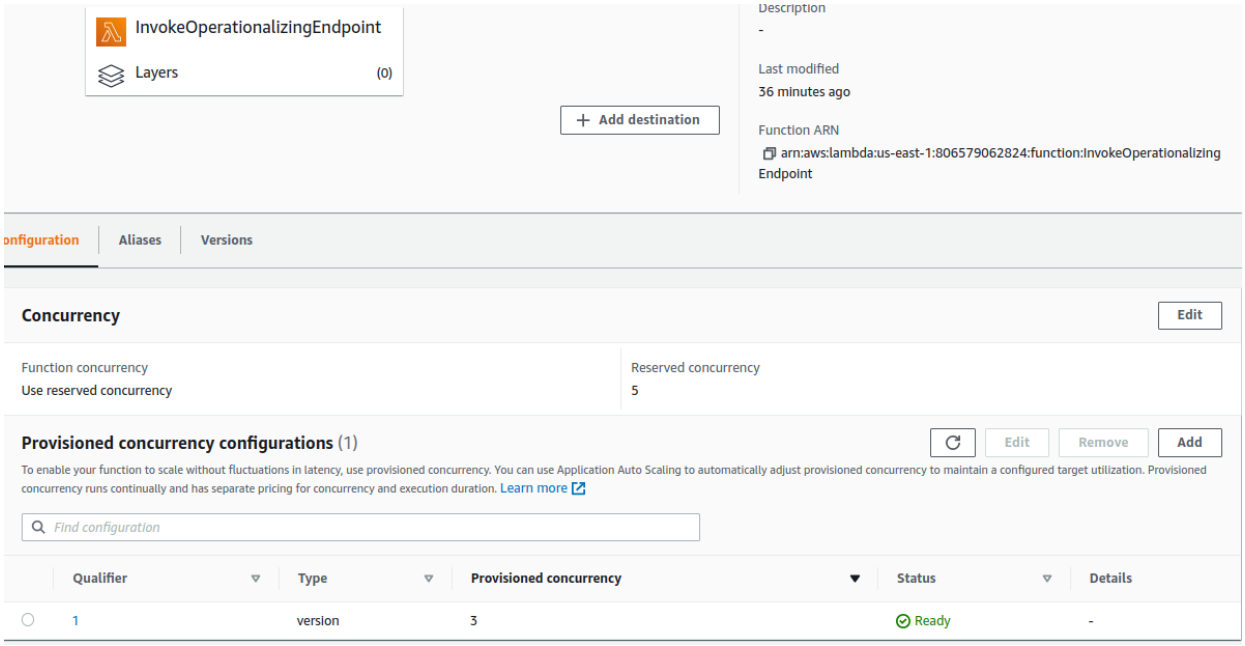
[-6.813735485076904, -2.314234733581543, -2.4530599117279053, -0.3391451239585876, -2.8606839179992676, -3.1122078895568848, -2.220827102661133, -2.298513650894165, -5.290785789489746, -0.7099820375442505, 0.08821891248226166, -3.1122288703918457, -2.582509994506836, 0.681918740272522, -2.8768880367279053, -1.4713103771209717, -7.354936122894287, -3.962596893310547, -2.3961808681488037, 0.8976378440856934, -1.867553949356079, -0.058517903089523315, -5.335705757141113, -4.2153425216674805, -3.741987705230713, -4.336663246154785, -2.1092474460601807, -4.2316789627075195, -5.577977180480957, -1.662833333015442, -2.8958067893981934, -4.032334804534912, -5.103762626647949, -2.651531219482422, -6.886448860168457, -3.2887139320373535, -3.31431245803833, -2.390284538269043, -0.3870977461338043, -4.738138198852539, -2.891908645629883, -1.413643479347229, -0.7013052701950073, -4.9064717292785645, -0.8747096657752991, -6.665121078491211, -1.6694062948226929, -1.8764053583145142, -1.0472358465194702, -1.6390801668167114, -4.352616310119629, -6.988555908203125, -4.930066108703613, -2.582667112350464, -4.983554363250732, -1.4864617586135864, -5.0205488204956055, -2.18139910697937, -2.2798972129821777, -1.9204773902893066, -7.435410976409912, -6.472367763519287, -7.7789483070373535, -7.328060150146484, -1.6872206926345825, -7.342635154724121, -0.3452627658843994, -5.989748954772949, -2.5607922077178955, -2.2447502613067627, 0.33085912466049194, -5.575009822845459, -3.4133946895599365, -4.608648300170898, -3.188641309738159, -2.747288942337036, -6.104413032531738, -1.1470075845718384, -4.789041996002197, -1.5678908824920654, 1.2766193151474, -5.887208938598633, -1.5808424949645996, 0.3379371166229248, -6.66790771484375, -8.003376960754395, -2.3290553092956543, -6.040937900543213, -2.7056031227111816, -0.603486180305481, -5.903278350830078, -4.089191913604736, -3.4990732669830322, -3.0671725273132324, -3.3422911167144775, -3.792228937149048,

Writeup by Ensiyeh Raoufi

0.18990904092788696, -3.626432418823242, -4.9088945388793945, -5.2451276779174805, -7.323543071746826, -2.1710963249206543, -4.08787202835083, -4.750383377075195, -3.4247186183929443, -10.414907455444336, -2.985180377960205, -1.2440528869628906, -2.0715267658233643, -0.1557144969701767, -2.155757188796997, -1.096891164779663, -3.184666872024536, -3.3269786834716797, -4.468642711639404, -1.218206524848938, -6.368831157684326, 1.0161904096603394, -5.429952621459961, -1.9429376125335693, -2.106492042541504, -4.268537521362305, -3.580470085144043, -2.394174337387085, -5.249018669128418, -3.9260332584381104, -2.9964399337768555, -1.1166750192642212, -3.86635422706604, -6.922861099243164, -4.6553635597229, -3.5067830085754395, -3.050952196121216]

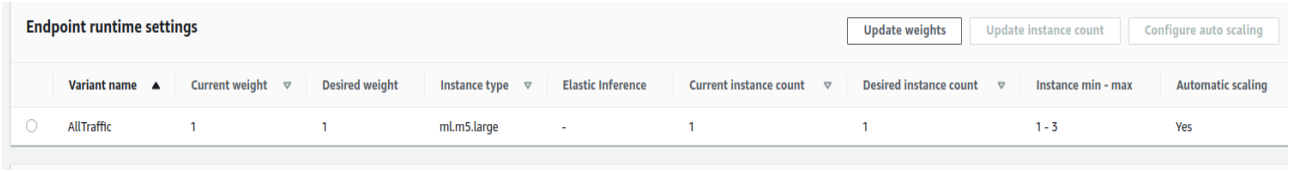
Step 5: Concurrency and auto-scaling

- I set the reserved concurrency of my lambda function to 5, which means my lambda function is able to access 5 instances to reply to multiple requests simultaneously.
- I published a new version of my lambda function and configured provisioned concurrency to enable my function to scale without fluctuations in latency. Although Provisioned concurrency has a higher cost than reserved concurrency, but it turns on instances that are always ready to respond to traffic (3 instances in my case). Because we don't know exactly how much traffic we expect to get, we need resources to be automatically provisioned based on whatever traffic comes. Setting this type of concurrency, we can achieve low latency even in very high traffic scenarios. Here is the screenshot of the configured provisioned concurrency:



Provisioned concurrency configurations is ready

- I set the maximum instance count to 3, scale in cool down to 30 (the time period, measured in seconds that the automatic scaling waits to delete instances when the traffic decreases), scale out cool down to 30 (the waiting period, measured in seconds, of an endpoint with autoscaling before decreasing to fewer instances). Because at the times when traffic increases, it's very important that users have low latency and get responses from our endpoints very quickly, it’s vital to set auto-scaling to the variants of our endpoints. Here you can see that the auto-scaling is set to the single variant of our endpoint:



Automatic scaling is configured

The screenshots of configuring concurrency for Lambda function, configuring auto-scaling for the endpoint, and the other images related to this part (step 5) are existed in the “Concurrency and Auto-scaling” folder. The names of the images explain the details of them.

Writeup by Ensiyeh Raoufi