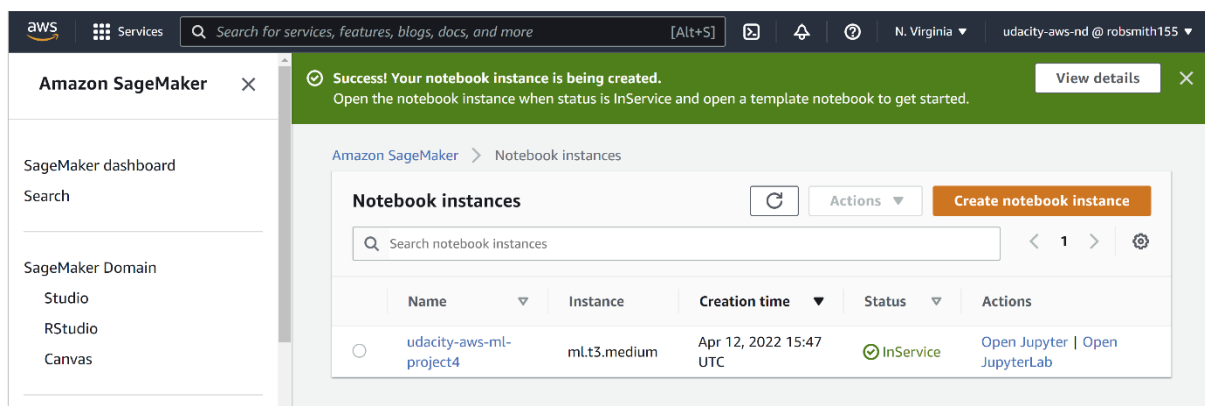


Udacity AWS ML Engineer – Project 4 report

Method 1: Training and deploying using SageMaker

Create Notebook instance

In the first approach, we train and deploy models from a Jupyter Notebook running on a Notebook instance in SageMaker. The first step was to create a Notebook instance to run the **train_and_deploy_solution.ipynb** notebook. Since the computationally expensive parts (i.e., model training) are not run using the notebook instance, we should use a cheap instance type. In this case, I used the ml.t3.medium instance. See [here](#) for more details on SageMaker instance costs.



Screenshot of Notebook instance created in SageMaker

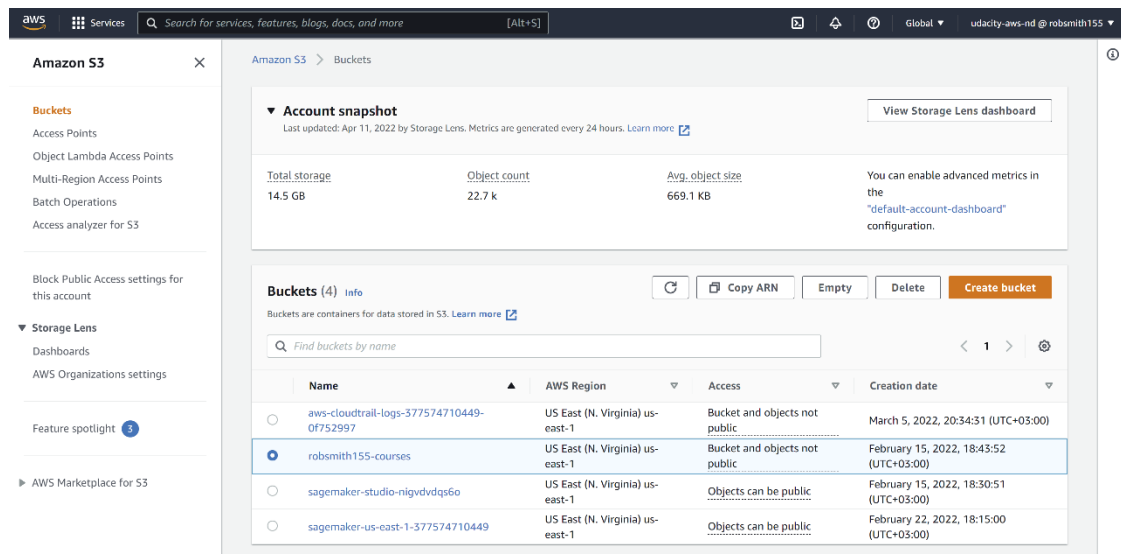
The ml.t3.medium instance has two virtual CPUs and 8GB of memory, which is more than sufficient for the work we will do in the notebook (which is mainly to download and extract some images). We submit model training as training jobs conducted on managed SageMaker instances.

| Region: US East (N. Virginia) ▾ | | | |
|---------------------------------|------|--------|----------------|
| Standard Instances | vCPU | Memory | Price per Hour |
| ml.t3.medium | 2 | 4 GiB | \$0.05 |
| ml.t3.large | 2 | 8 GiB | \$0.10 |
| ml.t3.xlarge | 4 | 16 GiB | \$0.20 |
| ml.t3.2xlarge | 8 | 32 GiB | \$0.40 |

Specifications and costs of the ml.t3.medium instance type on SageMaker ([Source](#))

Create S3 bucket

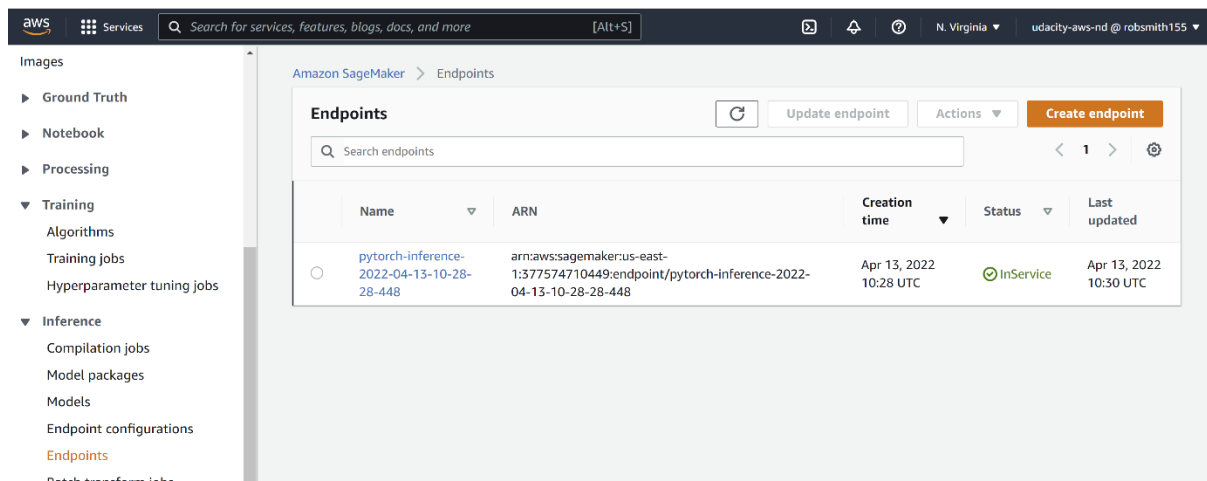
We need an S3 bucket to store the images that will be used for training and also to store the trained model artifacts. Below is a screenshot of the S3 bucket I used in my project.



Screenshot of the created S3 bucket

Deploy endpoint

After running the hyperparameter tuning and training a final model, we can deploy the model as an endpoint for real-time inference. The screenshot shows an example of a deployed endpoint.



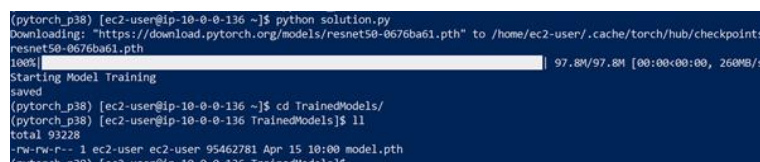
Screenshot of deployed endpoint

Method 2: Train on EC2 instance

Instance type

In the second part, we see how to train directly on an EC2 instance. The main advantage of this is that it reduces costs (~30% cheaper than SageMaker for the same instance type), but the setup process is more involved (see the README.md file in the repository for guidance on this).

In this case, I selected the m5.xlarge instance type. We used this to train the models in the SageMaker part above. The output reports did not pick up any issues, so this seems okay. It would be better to use a GPU instance since we use a convolutional neural network and image data. However, the training code provided by Udacity is not set up to work on GPU, and I don't want to spend time on this. You can see an example of how to write the code to work on GPU (if available) [here](#).



```
(pytorch_p38) [ec2-user@ip-10-0-0-136 ~]$ python solution.py
Downloading: "https://download.pytorch.org/models/resnet50-0676ba61.pth" to /home/ec2-user/.cache/torch/hub/checkpoints/resnet50-0676ba61.pth
100% |#####| 97.8M/97.8M [00:00:00:00, 260MB/s]
Starting Model Training
saved
(pytorch_p38) [ec2-user@ip-10-0-0-136 ~]$ cd TrainedModels/
(pytorch_p38) [ec2-user@ip-10-0-0-136 TrainedModels]$ ll
total 93228
-rw-rw-r-- 1 ec2-user ec2-user 95462781 Apr 15 10:00 model.pth
(pytorch_p38) [ec2-user@ip-10-0-0-136 TrainedModels]$
```

Screenshot of running the training script on an EC2 instance and model output

Code differences

Some modifications were made to the code used in SageMaker to make it suitable for running on an EC2 instance. Obviously, we can't make use of the SageMaker SDK outside of SageMaker, so rather than creating an instance of the PyTorch Estimator class and passing our code as the entry point, we now just run the script directly. This means that we don't need to define a main() function or use the if __name__=='__main__' block at the end of the script (see below). This is needed in SageMaker so that the script will run and we can pass hyperparameters from the Estimator class. In the code for the EC2 instance, we simply hard code the hyperparameters as part of the script.



```
if __name__ == '__main__':
    parser=argparse.ArgumentParser()
    parser.add_argument('--learning_rate', type=float)
    parser.add_argument('--batch_size', type=int)
    parser.add_argument('--data', type=str, default=os.environ['SM_CHANNEL_TRAINING'])
    parser.add_argument('--model_dir', type=str, default=os.environ['SM_MODEL_DIR'])
    parser.add_argument('--output_dir', type=str, default=os.environ['SM_OUTPUT_DATA_DIR'])

    args=parser.parse_args()
    print(args)
    |
    main(args)
```

Argparse arguments required in SageMaker but not on EC2 instance

Additionally, in the SageMaker code we needed to make use of logging to ensure important metrics are logged to AWS CloudWatch. SageMaker would then monitor these logs for our validation metric which would be used for things such as the hyperparameter search. When running on an EC2 instance, we can't make use of the SageMaker managed hyperparameter tuning, so we don't explicitly need the logging (although I see no reason why we couldn't include it). If we want to run a hyperparameter search on an EC2 instance, we will need to write the code for this ourselves (possibly using a package such as Optuna).

Lambda function

The code in the **lambdafunction.py** file was used to create a Lambda function in AWS. The role of this function is to send requests to the model endpoint that we deployed in an earlier step. For this we make

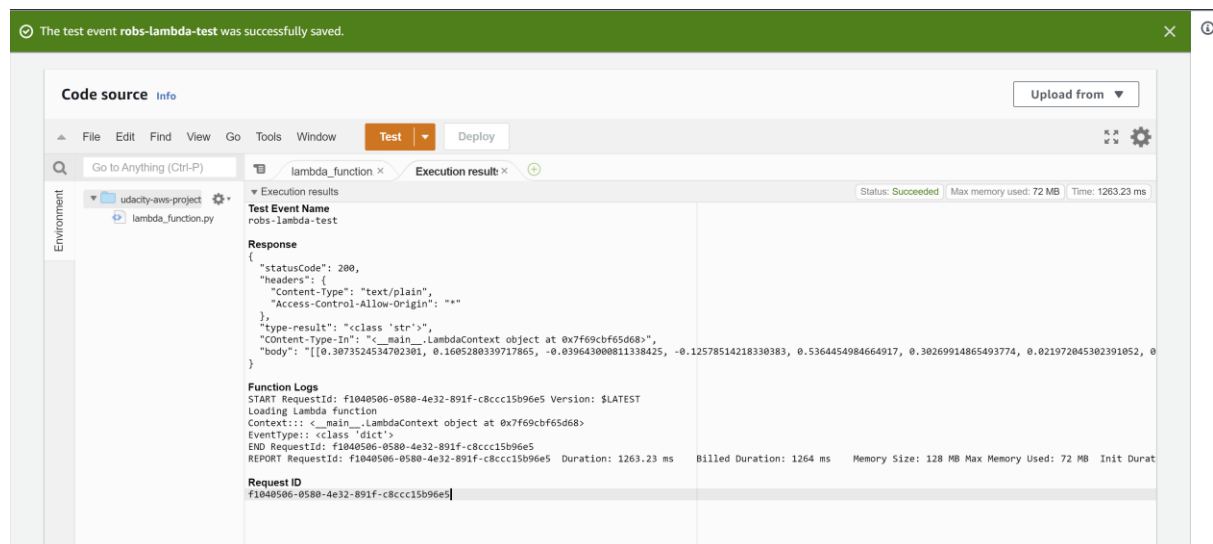
use of the **boto3** SDK and specifically the **invoke_endpoint()** method (see the [documentation](#)). Here we need to pass the following information:

- **EndpointName:** Name of the endpoint in SageMaker that we want to use to make predictions.
- **Body:** The input data we want to make a prediction with (in this case an image).
- **ContentType:** The Multipurpose Internet Mail Extension (MIME) data type of the input data in the request body.
- **Accept:** The desired MIME type of the response.

The **lambda_handler()** function is triggered when it receives a JSON object containing the URL path to an image. This is passed to the **invoke_endpoint()** method which sends the data to the endpoint as a POST request. In SageMaker, four functions are required for inference. Three of these are defined in the **inference2.py** module, which we used as the entry point for the deployed model. The **input_fn** function has an option to receive an image in JSON format which is then loaded as a Pillow image ready for passing to the model. The final prediction is sent back as a JSON object which is read and decoded in the Lambda function. The result is contained in the **body** key of the returned dictionary.

Endpoint response

A screenshot of testing the Lambda function in AWS is shown below.

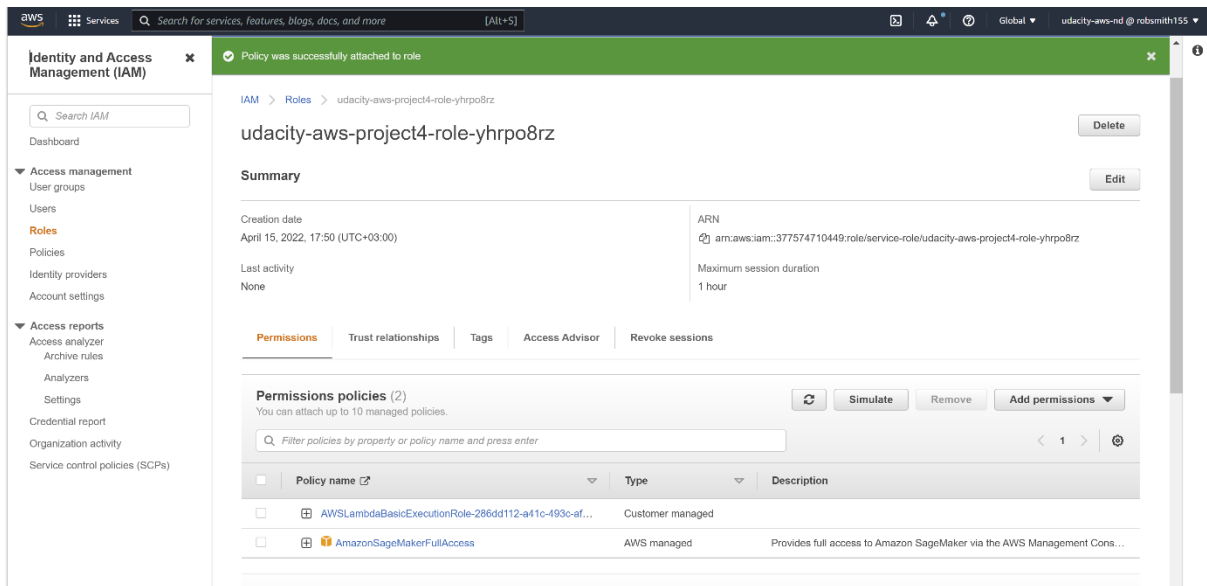


```
0.10215362906455994, 0.5603410005569458, 0.011395391076803207, 0.0859059989452362,
0.38507866859436035, -0.08101306110620499, 0.06991083174943924, 0.12087947130203247, -
0.06042066961526871, 0.014421327970921993, 0.20035959780216217, 0.5067268013954163,
0.08291489630937576, 0.44622117280960083, 0.3138309717178345, 0.03366061672568321,
0.3719773292541504, 0.08885599672794342, 0.2828841507434845, 0.21656250953674316, -
0.0393398143351078, 0.03334985673427582, 0.05661756545305252, 0.1356087177991867, -
0.027538329362869263, -0.01635069213807583, 0.2623848021030426, 0.026232119649648666,
0.3068734109401703, 0.4680444598197937, 0.0725063905119896, 0.03207611292600632,
0.23648276925086975, 0.2683831453323364, 0.14146173000335693, 0.08591248095035553, -
0.07255176454782486, -0.057108331471681595, -0.31853920221328735, -0.22579054534435272,
0.27474525570869446, 0.17421916127204895, -0.004210913088172674, 0.19220998883247375, -
0.01670219376683235, -0.06526736915111542, -0.16836169362068176, -0.015632934868335724,
0.33823859691619873, -0.0024862070567905903, -0.06596869975328445, 0.2977391183376312, -
0.08674783259630203, 0.2315887063741684, 0.2616405785083771, 0.02022174745798111,
0.16473010182380676, -0.18815717101097107, 0.21177102625370026, 0.2499392181634903,
0.10135456174612045, 0.18729324638843536, 0.2522779405117035, 0.38027429580688477,
0.0204362440854311, -0.24850353598594666, 0.07295502722263336, 0.09496000409126282,
0.01672663539648056, -0.0006885593757033348, -0.01095852255821228, -0.087455153465271, -
0.2082706242799759, -0.07572585344314575, -0.34130778908729553, 0.23969976603984833, -
0.10227128118276596, -0.36426258087158203, 0.06865102797746658, -0.017005393281579018, -
0.5326515436172485, -0.07029230892658234, -0.3015958070755005, 0.02527150698006153,
0.14810319244861603, -0.222121000289917, -0.23108166456222534, 0.2554415166378021, -
0.31914690136909485, 0.13121724128723145, 0.05213861167430878, -0.3299119770526886, -
0.15617701411247253, -0.47054558992385864, -0.40627631545066833, -0.10191944986581802,
0.18232806026935577, -0.31792423129081726, -0.3332754671573639, -0.03458722308278084, -
0.3782341778278351, 0.05443983152508736, 0.004122019745409489, -0.5084680914878845, -
0.5100063681602478, -0.3905940055847168]]"
```

```
}
```

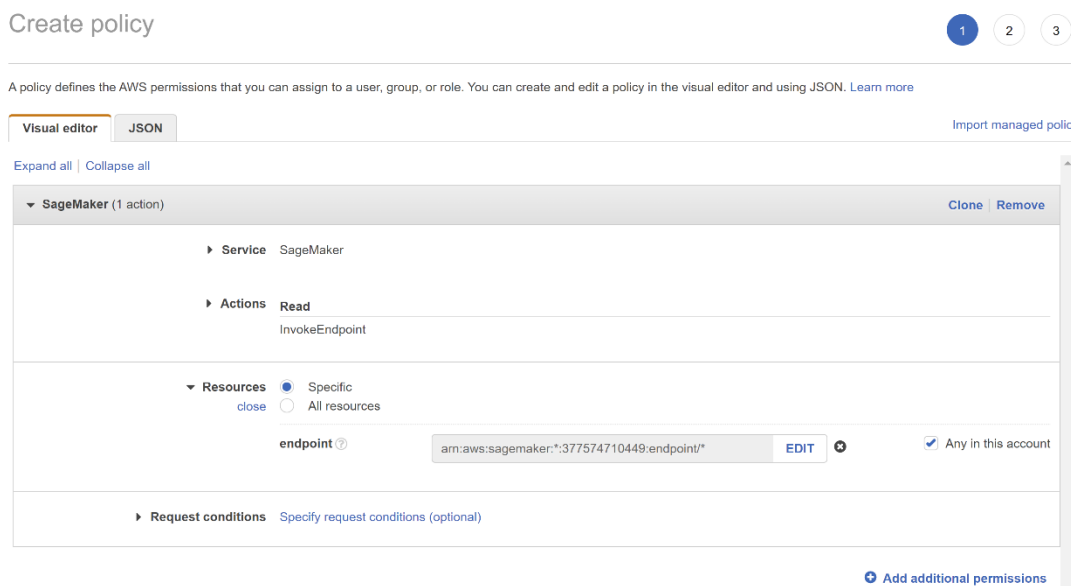
Security considerations

For the Lambda function to send requests to the model endpoint it needs access to SageMaker. Below you can see that I added full SageMaker access to the Lambda function. However, we shouldn't really do this as it gives the Lambda function more rights than it needs, which may be a security issue.



Screenshot of policies attached to the Lambda function

Instead, we should assign the minimum rights it needs to run. In this case, all that is required is permission to invoke an endpoint. I couldn't find this in the available policies so we need to use the option to create our own and select the **InvokeEndpoint** action under SageMaker (See figure below).



Lambda function only requires InvokeEndpoint access in this case

Concurrency and autoscaling

Concurrency refers to the ability of a Lambda function to serve multiple requests simultaneously. This is useful to reduce latency during high traffic. We haven't been given a use case for this project, so we can't really make an informed decision. In this case I used provisioned concurrency of 10, which means 10

Lambda functions can run simultaneously. Using more concurrency uses more compute which will increase our costs, which is something that we need to consider. Here I have assigned provisioned concurrency, which means they are always on. The advantage is that it will very quickly respond to high traffic and keep our latency low. A cheaper alternative is reserved concurrency, but here we may have to wait for instances to start which means we may still have latency issues.

The screenshot shows the 'Configure provisioned concurrency' page in the AWS Lambda console. The breadcrumb trail at the top reads: 'Lambda > Functions > udacity-aws-project4 > Version: 1 > Configure provisioned concurrency'. The main heading is 'Configure provisioned concurrency'. Below this, there is a section titled 'Provisioned concurrency'. Inside this section, it shows 'Version: 1' and 'Aliases: -'. A descriptive paragraph explains that provisioned concurrency enables scaling without latency fluctuations by running instances continuously. Below the text, it states '\$13.95 per month in addition to pricing for duration and requests.' with a link to 'Pricing'. A numeric input field is set to '10', with a note '900 available' below it. At the bottom right of the configuration box are 'Cancel' and 'Save' buttons.

Use of provisioned concurrency

While concurrency allows Lambda functions to respond to multiple requests at the same time, autoscaling does the same thing but for deployed endpoints (i.e., the endpoint can respond to multiple requests at the same time). Here we need to specify the maximum number of instances we want to use during peak traffic, where again we need to weight up latency vs. cost. In this case I allow up to four instance to run at the same time. A new instance is started when we get 20 almost simultaneous requests to each running instance.

The screenshot shows the 'Variant automatic scaling' page in the AWS SageMaker console. The breadcrumb trail at the top reads: 'SageMaker > Endpoints > SageMakerEndpoint > Variant automatic scaling'. The main heading is 'Variant automatic scaling' with a 'Learn more' link. Below this, there is a table with three columns: 'Variant name', 'Instance type', and 'Current instance count'. The first row shows 'AllTraffic' as the variant name, 'ml.m5.large' as the instance type, and '1' as the current instance count. Below the table, there are two input fields: 'Minimum instance count' set to '1' and 'Maximum instance count' set to '4'. At the bottom, there is a section for 'IAM role' which states 'Amazon SageMaker uses the following service-linked role for automatic scaling.' with a 'Learn more' link. The role name is 'AWSServiceRoleForApplicationAutoScaling_SageMakerEndpoint'.

Setting autoscaling for model endpoint