Operationalizing ML Workflow (Writeup)

Training in Sagemaker Notebook instance:

1. The chosen instance for creating Sagemaker notebook instance was ml.t2.medium. This has 2x vCPUs and 4GB of RAM, hence I think it will be a good fit for my project. Have chosen this instance since this is familiar to me as I have used it in my previous projects and exercises.
2. Performed single instance and multi-instance training. The multi-instance training was done using ml.m5.xlarge instance and the number of instances chosen were 4. Also noticed that both the jobs took almost same time for training. But the models created had different inference scores and found that the multi-instance training produced less loss and a better accuracy.

Code for this is provided in the ***train_and_deploy-solution.ipynb notebook***. The images related to this section are provided in "***SageMakerTrainingAndDeployment***" folder.

EC2 Instance Training:

1. The chosen instance for creating EC2 was t3.medium. With 2 vCPUs with 4GB of RAM, I think this instance will be a good fit for my project. Since micro ones are too small, and the training would take much longer time, chose t3.medium as EC2 instance.
2. The tradeoff between cost and performance enabled me to choose this instance. The training took only ~15 minutes and it succeeded by saving the model "***model.pth***" to ***/TrainedModels*** path.
3. **Challenges with default AWS console shell**: I had challenges in copy pasting the various contents - maintaining the indentation, the text etc. - from my local machine to the remote EC2 instance while working on the default AWS terminal shell console. I tried to mitigate it by ssh-ing on to the EC2 instance from my local terminal and thus leveraging all my local bash features while working on the remote machine. This enabled me to achieve the task sooner.

The image showing plot for CPU metrics and the images related to this section are in "***EC2Training***" folder

Differences between EC2 and Sagemaker:

1. In Sagemaker training, the hyperpameters and output locations are passed as arguments to the entry point script, hence used ***argparse***. But in EC2 training since these variables i.e., hyperparameters and output locations are already declared inside the script "***ec2train1.py"*** no ***argparse*** was required.
2. The model training using sagemaker notebook uses an entry point script. This means the model is trained in separate container, that is why we saved "SM_CHANNEL_TRAINING, SM_MODEL_DIR, SM_OUTPUT_DATA_DIR" in ***os.environ*** channel variables and used these in function file to get the paths of data input and output. The model training in EC2 can be thought of as a local training as both the training and code run in the same instance. Hence we did not use ***os.environ*** for EC2 training instead fetched the paths using ***os.path*** function.
3. In EC2 training, there is no estimator or tuner function to call a script, but all the code for training a pretrained model and saving it is in the code of ***ec2train1.py.*** Also noticed that creating a fully

connected layer with resnet50 pretrained model, model training and creating dataloaders are the same in both types of training.

4. We do not use any of the sagemaker specific modules in EC2 training. The drawback is that we cannot view the algorithm metrics easily from the cloudwatch console as using sagemaker. But can view the model completion status from the cloudwatch CPU metrics for the particular EC2 instance.

These are the difference I noticed after performing training in Sagemaker and EC2 instance.

Lambda Function:

1. The lambda function is used to invoke a deployed endpoint. In my lambda function code, I used the deployed endpoint: ***pytorch-inference-2021-12-21-22-47-18-583***.
2. To invoke this deployed endpoint from lambda function the ***Boto3:sagemaker-runtime.invoke_endpoint()*** is used
3. The lambda function takes the input as JSON format, pass this to the deployed endpoint to make prediction and this resultant prediction is passed to the lambda function's output body. The lambda function's return statement includes the following details: ***a status code, headers, a result type, a content type, and a body***.
4. The test event configured for testing this lambda function is an image of a dog.
5. The ***body*** contains a list of 133 numbers. These numbers represent score for each dog breed.

The images related to this section is provided in "***LambdaFunction"*** folder

Security and Testing:

1. The above step of invoking a deployed endpoint through lambda function threw an "**AccessDeniedException**" error since it did not have access to Sagemaker Endpoint. Hence attached a security policy ***"AmazonSageMakerFullAccess"*** to this lambda function's role and did the testing again.

2. This time the testing produced the result which is an array of 133 values.

```
[[-2.752485752105713, -3.0742406845092773, -2.7072434425354004, -
0.4588577151298523, -3.2295820713043213, -5.153140544891357, -
0.5385732054710388, -0.4383907914161682, -3.088043212890625, -
1.0321199893951416, 0.9932408332824707, -3.7188267707824707,
0.7552751302719116, 3.1842782497406006, -5.019706726074219, -
0.3598641753196716 3, -3.588852882385254, -0.8110900521278381, -
2.8695971965789795, 1.5904120206832886, -0.9241980910301208,
2.5600669384002686, -3.9685468673706055, -6.84370756149292, -
1.5361120700836182, -6.38437557220459, -0.4377124011516571, -
4.232139587402344, -4.49371337890625, -0.7085245847702026, -
0.4279719889163971, -1.6521046161651611, -4.066995143890381, -
2.4037914276123047, -5.823481559753418, -3.4529855251312256, -
3.157111644744873, -2.183858871459961, -0.0696466863155365, -
3.3678386211395264, -1.826881766319275, 0.3568563461303711,
2.181180238723755, -0.7849706411361694, -0.22415950894355774, -
6.7136101722717285, -1.6692124605178833, 0.6472925543785095, -
2.953855276107788, -2.9273879528045654, -3.1917378902435303, -
6.3375701904296875, -3.787379026412964, -2.664496421813965, -
3.166351556777954, -0.478836327791214, -3.03336501121521, -
4.0597686767578125, -2.4542789459228516, -1.3422201871871948, -
6.038204669952393, -3.1193830966949463, -2.7200071811676025, -
3.641798973083496, -3.90024471282959, -4.172355175018310 5, -
0.9714481830596924, -2.2529494762420654, 0.689486026763916,
0.8333936929702759, 2.6098477840423584, -3.0068612098693848, -
1.9449782371520996, -3.5508840084075928, -4.624110698699951, -
0.3362393379211426, -3.1156046390533447, -2.3471200466156006, -
5.647727966308594, -3.991274356842041, 1.399116039276123, -4.958477020263672,
0.18982377648353577, 0.10864777863025665, -1.2333266735076904, -
3.9756884574890137, -0.09176893532276154, -5.387872219085693, -
2.253798723220825, 1.3653414249420166, -7.478715419769287, -
5.3705596923828125, -2.272390365600586, -4.324128150939941, -
2.624507427215576, -1.4737805128097534, -2.77466082572937, -
2.2193846702575684, -3.445040464401245, -3.8443679809570312, -
9.351794242858887, -2.364043951034546, -3.1240758895874023, -
2.924889326095581, -2.1774723529815674, -2.983811140060425, -
1.0065370798110962, -1.57297945022583, -2.9210076332092285,
0.867063581943512, 1.6127315759658813, -0.8038678169250488, -
4.947366237640381, -3.357792615890503, -4.912656784057617, -
0.12306907773017883, -3.457397937774658, 0.15787291526794434, -
3.896953582763672, 1.8178815841674805, -1.0908938646316528, -
2.119222640991211, -4.131339073181152, -4.940979957580566, -
7.479795932769775, -4.8099751472473145, -1.8646808862686157, -
0.7822846174240112, -2.3617258071899414, -2.739793539047241, -
0.8559655547142029, -0.28576117753982544, -0.8008557558059692]]
```

3. From the above result it is evident that the best result is at index **13** with a value of **3.1842782497406006**
4. As an IAM user, it is a safe practice to enable the multi factor authentication to ensure that even if we mismanage our credentials, the unscrupulous elements cannot gain access to our AWS account as it demands an MFA credential.
5. It is preferable to rotate the access credentials and keys frequently to ensure extra level of security.

6. As a security best practice, the industry demands that each IAM user should have the least privileged access configured to ensure that the user is just able to fulfill his/her responsibility according to his/her role. All the unnecessary IAM roles/policies should be revoked for the IAM user.
7. There is a security vulnerability with respect to using the ***AmazonSageMakerFullAccess*** policy in lambda function's role for using it just to invoke(get/list/watch) the endpoint. But since there are no other predefined policies related to endpoint, except for ***AmazonMachineLearningManageRealTimeEndpointOnlyAccess***, available in the AWS console, I had to add the ***AmazonSageMakerFullAccess*** policy. The issue with using ***AmazonMachineLearningManageRealTimeEndpointOnlyAccess*** policy is that the specific policy has a ***Write*** access enabled on it by default. The only other way is to try and create a custom IAM policy to provide the user with get/list/watch-only access for the endpoint.

The images related to this section are in "***Security***" folder.

Concurrency and Autoscaling:

1. Once we have deployed endpoint and created lambda function successfully, need to set up concurrency for the lambda function and autoscaling for the deployed endpoint.
2. To set up concurrency for my lambda function, I set up ***reserved concurrency to 5*** which means my lambda function can handle up to 5 requests simultaneously. To reduce the latency, also configured ***always-ready provisioned concurrency to 2.***
3. To handle the time when there will be high traffic, autoscaling was set up for the deployed endpoint. I have set the ***Scale in cool down to 300*** and **Scale out cool down to 30** to minimize the cost. After doing the autoscaling configuration for deployed endpoint, the screenshot of **Endpoint runtimes settings** dashboard shows ***instance min-max -- 1-2***

The images related to this section is in the ***"Concurrency_and_AutoScaling"*** folder.