OPERATIONALIZING MACHINE LEARNING ON SAGEMAKER
Course Project
Student: Oscar Chavarriaga, ochavarriaga@grupobit.net
Analytics Leader – Grupo bit www.grupobit.net

Step 1: SageMaker instances type for Jupyter Notebook.

I selected the same sagemaker jupyter notebook instance I have been using for the other activities in the course, I chose this because it is a cheap instance and all the heavy computations like training and deployment will be using on-demand provisioned instances. These instances types are already configured in the notebook code, ml.m5.xlarge for hyperparameter tuning and training, and ml.m5.large for deploying the endpoint.

Step 2: EC2 Training

EC2 Instance type

In this case, a chose the instance m5.xlarge which is the equivalent for the one that I used for training when using the SageMaker estimator ml.m5.xlarge, I made this choice in order to compare performance in both exercises, sagemaker-training-job and ec2-training.

EC2 code

The main difference between the code for training job in SageMaker and the equivalent code for training a model within a EC2 instance is while setting the parameters for the job. In the case of SageMaker training job, parameters as data sources and hyperparameters for the PyTorch model are set using environmental variables, this is because it is necessary to be configured in this way for the SageMaker training job works properly; on the other hand, for the EC2-training all these parameters and hyperparameters are hard-coded. Furthermore, considering SageMaker training jobs, it is used classes as Estimators for defining the hardware and software that will be used for the training job, otherwise, in the case of EC2-training, the execution of the code is achieved via the command line, and the software and hardware choices were previously made during the EC2 instance creation.

Step 3: Lambda function – How it works

The lambda function takes an image url within a json payload and then it invokes a prediction from the endpoint, it is done used a sagemaker runtime client from the library boto3. Then, as we can see in the input_fn defined at inference2.py file, the entry point for the endpoint, the inference process downloads the image from the url and then parse it as a byte array using io.BytesIO method. This transformed image will be used for making the inference with the deployed model.

Step 4: Security and testing – Possible vulnerabilities

In my case, I attached the “AmazonSageMakerFullAccess” policy to the lambda function’s execution role. I think this is the case of a very permissive policy for a lambda function. This could be considered as a security vulnerability. A good solution for improving this security vulnerability could be unattaching this policy, and then, creating a policy which only involves the needed permissions

for invoking an endpoint from a lambda function, obviously we have to attach this policy to the execution role after creation.

Step 5: Concurrency and auto-scaling

For lambda function's concurrency, I set up a provisioned concurrency of three instances, this choice was made for having tripled the simultaneous requests that the lambda function can handle. Furthermore, in the case of the auto-scaling, I configured 30 seconds in both the scale in and scale out parameter, this decision was made to auto-scale as soon as the traffic shows an increment. Other established parameter was the target value for the "SageMakerVariantInvocationsPerInstance" metric, it was set as 10, this means sagemaker will launch a new endpoint instance as soon as the average number of times per minute that each instance has overcomes 10.