Project 4 AWS Machine Learning Engineer Nanodegre

This is the fourth project of the nanodegree **AWS Machine Learning Engineer**. In this project we are going to set up the right configuration of a machine learning project involving computer resources, production configuration, deployment configuration, security, latency and concurrency.

Notebook Instance Setup

Firstly, we will create a notebook instance from Sagemaker » *Notebook* » *Notebook Instances* » *Create notebook instance*

For this project we will user for our instance a ml.t3.medium type that we will name "project-4-udacity".

S3 Setup

Next step is to create a new S3 Bucket:

From S3, we will click on Buckets and Create bucket to create our new bucket

Then upload the data to our bucket using wget command:

```
%%capture
!wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
!unzip dogImages.zip
!aws s3 cp dogImages s3://project-4-udacity-ml/data/ --recursive
```

After that, we can start our hyperparameter tunning job, setting parameters, estimators and tunners first. Our estimator will use our hpo.py script as entry point to train the model with the different hyperparameters values that we set up:

```
# Hyperparameters: learning rate and batch size.
hyperparameter_ranges = {
    "learning_rate": ContinuousParameter(0.001, 0.1),
    "batch_size": CategoricalParameter([32, 64, 128, 256, 512]),
}

role = sagemaker.get_execution_role()

objective_metric_name = "Test Loss"
objective_type = "Minimize"

# Estimator
estimator = PyTorch(
```

```
    entry_point="hpo.py",
    base_job_name='pytorch_dog_hpo',
    role=role,
    framework_version="1.4.0",
    instance_count=1,
    instance_type="ml.m5.xlarge",
    py_version='py3'
)

#Tuner
tuner = HyperparameterTuner(
    estimator,
    objective_metric_name,
    hyperparameter_ranges,
    metric_definitions,
    max_jobs=2,
    max_parallel_jobs=2,
    objective_type=objective_type
)
```

After that we can set our enviroments variables and star the hyperparameter tunning job:

```
# Environment variables
os.environ['SM_CHANNEL_TRAINING']='s3://project-4-udacity-ml/data/'  # data in S3 to train
os.environ['SM_MODEL_DIR']='s3://project-4-udacity-ml/' # output of the model artifact
os.environ['SM_OUTPUT_DATA_DIR']='s3://project-4-udacity-ml/output/' # location of the outp

# hpo.py script access to our environment variables
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()

# Hyperparameter job:
tuner.fit({"training": "s3://project-4-udacity-ml/data/"})
```

Hyperparameter Tunning Job:

Training Jobs: Hyperparameter Tunning

Estimators:

One Instance:

Five Instances:

Training Jobs: Estimators

All Training Jobs Completed

Deployed Model

EC2 Setup

We will first navigate using the console to EC2 and will click on "Launch instaces"

We give the name *project-4-aws-ml* and select *Deep Learning AMI GPU PyTorch* as AMI:

Then we have to select an instance type, in this case we select t3.medium as a reasonable balance of performance and affordability.

We will create also a key pair, so we can access, if we needed, from Cloud9:

Key pair: *project4*

After that, we can launch our EC2 instance:

Now we can connect our instance:

We run our first command:

```
wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
unzip dogImages.zip
```

and the command to create the directory for the model

```
mkdir TrainedModels
```

Next step is to create a blank Python file in order to paste the training code:

```
vim solution.py
```

We will use the following command to paste the code in ec2train1.py to the file and press enter:

```
:set paste
```

After pasting the code, we need to type the following command:

```
:wq!
```

Now we are ready to train the model, by typing the following command:

```
python solution.py
```

Training model

Model saved

Metrics

Comparision SageMaker vs EC2

Training models on EC2 offers extensive customization and control, allowing users to manually manage the entire machine learning workflow, optimizing it based on specific needs. On the other hand, Amazon SageMaker provides a more integrated and automated environment, focusing on streamlining the machine learning process from training to deployment, making it user-friendly and efficient.

EC2: - High Customization & Control - Manual Management of Workflow

SageMaker: - Integrated & Automated Workflow - User-friendly & Efficient

Lambda Functions Setup

To create our Lambda function, we will navigate to AWS Lambda in our console and will click on "Create a function" that uses Python 3.8:

Once paste the code we have from the file lambdafunction.py and click on "Deploy":

Now, if we try to test the Lambda function with the following json body:

```
{ "url": "https://s3.amazonaws.com/cdn-origin-etr.akc.org/wp-content/uploads/2017/11/201133
```

We will get an error:

```
[ERROR] ClientError: An error occurred (AccessDeniedException) when calling the InvokeEndpoi
```

Basically, the role created when setting our Lambda has not rights to perform the action. To fix this, we will update the rights adding a new policy to our Lambda function so that it can access Sagemaker. We will do this through IAM » Roles » searching our Lambda in the search bar:

Now we will add the policy in order to give rights to trigger the test clicking on the role » Attach policies and searching for the policy *AmazonSageMakerFullAccess*, selecting it and adding:

Adding this policy will allow our lambda to access Sagemaker and our Endpoint. Also, by default, our Lambda function can only handle one request at one. To change this, we will use concurrency in order to give our Lambda the power to trigger multiple times at once.

To add concurrency we would need to configure a new version we will click on Actions » Publish new version:

Now we will click on "Provisioned concurrency" and "Edit":

We will set the concurrency to 3, so our Lambda function can handle 3 requests at once:

New concurrency can take a few minutes to be ready:

Auto-Scaling

Last step for our excersise is to implement Auto-scaling. Auto-scaling referrs to the dynamic adjustment of the instances used with deployed models. This adjustment is based on workload and it's increased or decreased based on this. This is useful in order to ensure that we are charged only for the active instances.

To add Auto-scaling, we will click on Sagemaker » Endpoints » Select our Endpoint » Configure auto scaling

First, we will increase up to 3 instances as max number of instances:

Secondly, we will define the scaling policy. This would referr to the number of simultaneous requests that our endpoint would receive in order to trigger the auto-scaling feature and increase the number of instances. This value would be the *Target* and we will set it to 30 requests. Then we have *Scale in cool down* and *Scale our cool down*:

- **Scale in cool down:** Number of seconds that the auto-scaling must wait before increasing the number of instances
- **Scale out cool dow:** Number of seconds that the auto-scaling must wait before decreasing the number of instances

We wil set this to 45 seconds:

And our endpoint would be ready:

In this fourth project for the AWS Machine Learning Engineer Nanodegree, the focus has been on configuring and deploying a machine learning project covering aspects like computer resources, security, and concurrency. Initially, a SageMaker notebook instance was set up, followed by creating an S3 bucket for storing project data and subsequently configuring a model with hyperparameters. Training was conducted on both EC2 and SageMaker, emphasizing EC2's customization and SageMaker's user-friendly automation. Lambda functions were implemented to invoke the model endpoint, which involved resolving permission issues and configuring concurrency for handling multiple requests. Finally, auto-scaling was applied to the deployed model to dynamically adjust instances based on the workload, enabling an efficient utilization of resources.