writeup.md

Student justifies the type of Sagemaker instance they created

```
Text in the writeup describes and justifies the student's choice
```

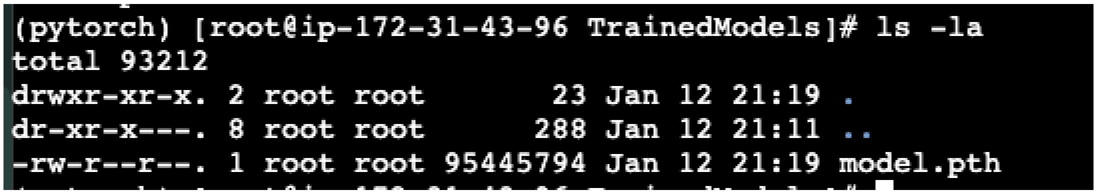
Since the heavy computing (training, batching, and inference) will be conducted using the Estimator, which will handle the creation of new instances, the instance selected was a small one: ml.t3.medium. This will minimize the cost of implementation.

Create and open an EC2 instance

```
Writeup contains a description of the EC2 instance created, including justifi
```

The EC2 instance selected was a m5.large. This instance was selected because it is a general-purpose instance that provides a balance of compute, memory, and networking resources. Since there is no much data to train on GPU, the m5.large instance was selected to minimize the cost of implementation. And compared to a t5 instance, the m5 instance provides better performance.

Here is the screenshot of the model saved in the EC2 instance:



```
(pytorch) [root@ip-172-31-43-96 TrainedModels]# ls -la
total 93212
drwxr-xr-x. 2 root root       23 Jan 12 21:19 .
dr-xr-x---. 8 root root      288 Jan 12 21:11 ..
-rw-r--r--. 1 root root 95445794 Jan 12 21:19 model.pth
```

Student identified potential vulnerabilities in the project's IAM configuration.

```
The writeup contains a description of potential vulnerabilities in the IAM se
```

The IAM role created for the Lambda function, IAM Role `LambdaFunctionUdacity5-role-m9k4xfwe`, has the `AmazonSageMakerFullAccess` policy attached. This policy provides full access to all SageMaker resources. This is a vulnerability because the Lambda function only needs read access to the SageMaker endpoint. The policy should be updated to provide only the necessary permissions. A bad actor could potentially use the Lambda function to delete SageMaker resources or create new ones, which could result in financial loss or data leakage.

Roles that are old or inactive can also be a vulnerability. If a role is no longer in use, it should be deleted to prevent unauthorized access to resources. Bad actors could potentially use old or inactive roles to gain access to resources that they should not have access to.

Roles for functions that the project is no longer using can also be a vulnerability. If a role is no longer in use, the policies attached to that role should be reviewed and removed if they are no longer needed. Bad actors could potentially use these policies to gain unauthorized access to resources.

Student clearly describes the configuration of concurrency (on the Lambda function) and auto-scaling (for the deployed endpoint)

```
The writeup clearly describes traffic, cost, and efficiency considerations fc
```

A Lambda function has a concurrency limit of 1000. This means that the function could handle up to 1000 requests at the same time. Reserve concurrency can be set from 1 to 1000. Increasing the reserve concurrency will reduce the latency of the function, but it will also increase the cost. Any request that exceeds the reserve concurrency will be throttled. This follows the serverless architecture principle of "cold start" where the function is initialized when a request is received. The Lambda function will scale automatically based on the number of requests received. This will help to optimize the cost and efficiency of the function.

There is also the option to set up provisioned concurrency for the Lambda function. Provisioned concurrency will keep the function warm and ready to handle requests. This will reduce the latency of the function, but it will also increase the cost since the function will be running even when there are no requests. Provisioned concurrency can be set up to the reserve concurrency limit. This is not fully serverless since the function will be running all the time, but it can be useful for applications that require low latency.

Test the Lambda function by invoking the endpoint with the "lab.jpg" input.

```
Student provides the Lambda function response in the writeup
```

I could not find the lab.jpg image but I used the Carolina-Dog-standing-outdoors.jpg image instead.

```
Here is the output of the Lambda function:
```

```
Status: Succeeded
Test Event Name: ImageTest

Response:
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "text/plain",
    "Access-Control-Allow-Origin": "*"
  },
  "type-result": "<class 'str'>",
  "Content-Type-In": "LambdaContext([aws_request_id=07f7761d-3fb2-41f0-a7
  "body": "[[-8.520931243896484, -5.1566548347473145, -2.462192535400390
}

Function Logs:
Loading Lambda function
START RequestId: 07f7761d-3fb2-41f0-a790-9debc31f308b Version: $LATEST
Context::: LambdaContext([aws_request_id=07f7761d-3fb2-41f0-a790-9debc31f
EventType:: <class 'dict'>
Event {'url': 'https://s3.amazonaws.com/cdn-origin-etr.akc.org/wp-content
END RequestId: 07f7761d-3fb2-41f0-a790-9debc31f308b
REPORT RequestId: 07f7761d-3fb2-41f0-a790-9debc31f308b      Duration: 207

Request ID: 07f7761d-3fb2-41f0-a790-9debc31f308b
```

Data sharding on EC2. Adjust your EC2 training code so that it accomplishes multi-instance training, including data sharding.

The file `ec2train-multi.py` contains the code for multi-instance training with data sharding. The code reads the data from the S3 bucket and shards it into multiple parts. Each instance reads a different shard of the data and trains on it. It uses torch.distributed to coordinate the training across instances. The code also saves the model to the S3 bucket after training is complete. It is training on CPU using gloo instead of nccl because the data is small and GPU clusters are expensive.

API setup. Use the API gateway to set up an API that allows access to your Lambda function by the public.

The API Gateway was set up to expose the Lambda function to the public. The API Gateway has a POST method that triggers the Lambda function. You can see the API Gateway call in the train_and_deploy.ipynb notebook. The lambdafunction.py was updated to follow the proxy integration for the response from the Lambda function.