194.048 Data-intensive Computing Assignment 3 (Group 45)

Baretto Trevor Calvin (12332204), Mazanek Daniel (12005060), Taha Josef (11920555)

July 8, 2024

Contents

1 Introduction

Assignment 3 of course 194.048 is about offloading computational expensive tasks to the AWS cloud. By also implementing a local RestAPI and comparing those two solutions with various metrics, such as time, we can draw conclusions about the differences in efficiency. The results are comparable by using a pretrained YOLO-v3 neural network. In the end, we show that offloading computational intensive tasks can provide significant runtime improvements.

We also aim to explore whether the efficiency gains from offloading tasks to AWS are substantial enough to justify the architectural complexity and potential costs associated with cloud deployment and whether AWS execution is worth it for this particular object detection use case.

2 Problem Overview

The problem consisted of 3 parts.

First, a local server had to be implemented which accepts POST-requests consisting of a unique identifier and a base64 encoded image. The local server should use the YOLO model and openCV python libraries to detect images and return the detected objects (including accuracy) as json together with the unique identifier.

Part two consisted of using the resources of the AWS cloud for the object detection in the images. This had to be realized by creating AWS resources like S3 bucket, ECR registry, DynamoDB. Then a lambda function is created using a custom Docker image, which is triggered whenever an image is uploaded to the S3 bucket, which triggers the image processing and finally stores the result into DynamoDB.

Implicitly stated, a local client had to be implemented which can be triggered to perform either local or remote execution.

The third and final part of this exercise is about gathering data about the inference time for the respective executions. The inference time should be calculated per image and then averages across all images.

3 Methodology and Approach

3.1 Task 1: Local Execution

3.1.1 Set up client

At first, client.py was created to trigger either either local or remote exeuction. The argparse module was used for parsing the arguments and setting default parameters if needed.

3.1.2 Local execution

local_exec.py contains the actual logic of converting the images and sending them as POST-request to the local server. It was created to keep client.py light-weight.

local_exec.py receives the image folder, the endpoint of the local server and the output path as parameters from client.py. All images from the image folder are read in; each image is base64 encoded and sent to the local server together with a uuid by send_image_local().

Each successful response is accompanied with a time measurement; both the response and the time measure are added to a json, which then stores all results under /out/detections_local.json (default).

3.1.3 Local server

local_server.py include the local server, implemented with FLASK and openCV.

The file includes a class Yolo, which contains the method detect(). Detect() uses the pre-trained model weights to get predictions for a given image. Only predictions with a confidence > 0.5 are returned as json object with the given labels and accuracies.

The endpoint /api/object_detection expects POST-requests. If the request contains a valid input, it decodes and buffers the image. detect() from the preloaded Yolo-instance is called, the result is converted to a json and sent as response.

3.2 Task 2: AWS Execution

The aim of this part was to set up an AWS infrastructure to perform image recognition using a Lambda function triggered by S3 events. The entire process, from building the AWS infrastructure to uploading images, is executed using Python's Boto3 library, avoiding any reliance on the AWS CLI or AWS Management Console. The steps involved are outlined below.

3.2.1 AWS Infrastructure Setup

The infrastructure setup is orchestrated through the `aws_create_resources.ipynb` notebook. This notebook leverages utility functions defined in `src/aws/utils.py` to create, delete, and manage AWS resources. Below are the key steps in the notebook, along with the corresponding code snippets.

```
import src.aws.utils as aws
```

- **Create S3 Buckets**: The notebook includes functions to create S3 buckets that will store the images for recognition.

```
aws.create_s3_bucket(S3_IMAGE_BUCKET_NAME, REGION)
```

- **Create ECR Registry**: An Elastic Container Registry (ECR) is created to store Docker images for the Lambda function.

```
aws.create_ecr_repository(ECR_REPOSITORY_NAME, REGION)
aws.list_ecr_repositories(REGION) # list repositories
```

- **Create DynamoDB Table**: A DynamoDB table is set up to store the results of the image recognition. This function creates the table with a schema that stores (1) an id, (2) a time which is the time needed for image processing and (3) a hash which stores the labels and accuracies.

```
aws.create_dynamodb_table(table_name=TABLE_NAME, region=REGION)
```

- **Build and Push Docker Image**: The Dockerfile located in `src/aws/lambda/image-detection` is used to build a Docker image containing the Lambda function and necessary dependencies. More on that you can find in the next section. This image is then pushed to the ECR registry. We use the aws cli in order to authenticate to the registry.

```
subprocess.run(f"docker build -t {IMAGE_NAME} {LAMBDA_SRC_PATH}", shell=True,
    check=True)
# HINT install aws cli locally
subprocess.run(f"aws ecr get-login-password --region {REGION} | docker login
    --username AWS --password-stdin {IMAGE_NAME}", shell=True, check=True)
subprocess.run(f"docker push {IMAGE_NAME}", shell=True, check=True)
```

- **Create Lambda Function**: The Lambda function is created using the Docker image from the ECR registry. Here it is important to note that we have provided the Lambda function with the `LabRole` role, so it has enough permissions for accessing the S3 bucket and DynamoDB. Furthermore it was essential to modify the default Lambda function timeout and memory limits as these resulted into the function to fail. We set the `timeout` to 900 seconds and the `memory limit` to 1024MB which are the max limits.

```
aws.deploy_lambda_function(LAMBDA_FUNCTION_NAME, IMAGE_NAME, LAB_ROLE_ARN,
    900, 1024, region=REGION)
```

- **Add S3 Trigger to Lambda Function**: The following code adds a trigger to the Lambda function to execute it upon image upload to the S3 bucket. For this the function performs the following steps: (1) adding **lambda:InvokeFunction** to the lambda function via add_permission function, (2) setting up the bucket trigger via put_bucket_notification_configuration.

```
aws.create_s3_lambda_trigger(S3_IMAGE_BUCKET_NAME, LAMBDA_FUNCTION_ARN)
```

3.2.2 Lambda Function Implementation

The Lambda function implementation underwent several iterations to address the challenges of packaging dependencies and adhering to AWS Lambda's size limitations. Below are the key approaches and the final solution:

Initial Approach: Zip Deployment Initially, we attempted to deploy the Lambda function using a zip file. This approach involved extracting all necessary Python libraries and including them in the zip file along with the function code. However, this method presented two significant challenges:

- **Dependency Management**: Extracting and including all required libraries and binaries was complex and error-prone.
- **Size Limitation**: AWS Lambda has a 70MB limit for the deployment package. Our package exceeded this limit, making it infeasible to use the zip deployment method.

Second Approach: S3 Bucket for Binaries The second approach involved creating a separate S3 bucket to store the binaries and YOLO weights. The Lambda function would download these files at runtime. While this method overcame the size limitation, it introduced issues with reading and loading the libraries dynamically during function execution, leading to increased complexity and potential performance drawbacks.

Final Approach: Docker Container Deployment The final and most effective approach was to deploy the Lambda function as a Docker container. This method eliminated the size limitations and allowed for seamless inclusion of all dependencies. The build image had a size of 1GB. The Dockerfile `src/aws/lambda/image-detection/Dockerfile` used for this purpose is shown below:

```
# Specify AWS Lambda Python 3.11 runtime as base
FROM public.ecr.aws/lambda/python:3.11
# Install required Python packages
RUN pip install boto3 opencv-python-headless numpy
# Copy the YOLO files, which can then be locally read by handler function
COPY yolov3-tiny.weights /opt/yolov3-tiny.weights
COPY yolov3-tiny.cfg /opt/yolov3-tiny.cfg
COPY coco.names /opt/coco.names
# Copy the function code into the AWS workdir PATH
COPY lambda_function.py ${LAMBDA_TASK_ROOT}
# Set the CMD to your handler, so it gets executed
CMD ["lambda_function.lambda_handler"]
```

Lambda Function Execution The image processing code used for the Lambda function is the same as the local exectuion code. The Lambda function is triggered by an S3 bucket event whenever an image is uploaded. Key steps in the Lambda function (`src/aws/lambda/image-detection/lambda_function.py`) execution include:

- **Loading the Image**: The function retrieves the uploaded image from the S3 bucket using the boto3 client.

```
bucket_name = record['s3']['bucket']['name']
image_name = record['s3']['object']['key']
s3_client = boto3.client('s3', region_name='us-west-2')
image_obj = s3_client.get_object(Bucket=bucket_name, Key=image_name)
image_content = image_obj['Body'].read()
```

- **Loading the YOLO Model**: The YOLO model configuration and weights files are loaded from the Docker containers `/opt` path. The COCO names file is also read to map the detected objects to their labels.
- **Image Processing and Object Detection**: The image is decoded using OpenCV, and the YOLO model is used to detect objects in the image. Bounding boxes, class IDs, and confidences are extracted for each detected object.
- **Storing Results in DynamoDB**: The results, including the detected objects and their confidences, are stored in a DynamoDB table along with the image ID and the processing time.

```
dynamodb_client = boto3.resource('dynamodb', region_name='us-west-2')
table = dynamodb_client.Table(DYNAMODB_TABLE_NAME)
table.put_item(Item=result)
```

3.2.3 AWS Client and Image Detection Execution

The AWS client script is designed to handle the process of uploading images to an S3 bucket, which automatically triggers the Lambda function to perform image recognition. The results are then retrieved from the DynamoDB table. The client script automates the entire workflow, allowing for concurrent execution of multiple Lambda functions to handle image recognition tasks. To execute the client script, use the following command:

```
python3 client.py --exec_type aws --input input_folder/
```

Client Process The client script performs the following steps:

- **Image Upload**: Uploads images from the specified input folder to the S3 bucket using threading to allow multiple uploads simultaneously. Each upload's time is recorded.
- **Lambda Trigger**: The S3 bucket triggers the Lambda function, enabling concurrent processing of images.
- **Result Retrieval**: After processing, the client retrieves the predictions from the DynamoDB table, including detected objects and their confidences.
- **Output Storage**: The predictions and processing/transfer times are saved to `detections_aws.json` and `detections_aws_transfer_time.json` for analysis.

Concurrent Upload and Processing The client uses threading to upload images concurrently, demonstrating AWS's ability to handle multiple Lambda function executions simultaneously. This approach ensures efficient and independent processing of each image, leveraging AWS's scalable infrastructure.

	Local	AWS
Total inference time	4.43 seconds	88.19 seconds
Average inference time per image	0.04 seconds	0.88 seconds

Table 1: Local vs AWS inference times

3.3 Task 3: Experiments- Local and Remote Execution

For the experiments, we used the Python module time to calculate the average time needed to transfer each image file in the dataset, and the average inference time on AWS.

1) The time needed to transfer each image file in the dataset The total upload time for all images on AWS was 230.33 seconds which is about 2.30 seconds on average for each image.

2) The average inference time on AWS As per table 1, the total inference time on AWS is 88.19 seconds and the average inference time per image is 0.88 seconds which is much higher compared to the inference times when the object detection was performed locally.

4 Results

We found that while the transfer time remains consistent between local and AWS execution, the inference time on AWS is significantly higher. The overhead introduced by AWS infrastructure (such as Lambda function initialization and network latency) contributes to this difference.

If scalability and parallel processing are critical, AWS could be beneficial despite the increased inference time. Some additional steps could also be taken to improve the AWS performance such as:

- Deploying the lambda function as a docker container.
- Optimizing the lambda function initialization and memory allocation.
- Batch processing to reduce overhead.

5 Conclusion

To conclude, we found that the biggest difference between local and AWS execution emerged in inference time. When running the YOLO-v3 neural network for object detection, local execution achieved an average inference time of 0.04 seconds per image, while AWS execution took an average of 0.88 seconds per image. This discrepancy can be attributed to the overhead introduced by AWS infrastructure, including Lambda function initialization and network latency.

Despite the longer inference time, AWS offers scalability and parallel processing capabilities which are especially important when handling a large number of images simultaneously. AWS is also beneficial due to its ability to distribute workloads efficiently.

Our biggest takeaway from this assignment is that it is important to consider the trade-offs between execution time and infrastructure overhead when choosing between local and cloud-based solutions and the choice between local and AWS execution depends on the specific use cases and requirements.

Furthermore, we observed thqat it is much simpler to deploy Lambda functions using docker images instead of trying to bring libraries, files and function into on functional zip file or trying to store the libraries into s3 bucket in order to overcome the zip size limit of 70MB.