Object Classification on AWS

Gianmarco Montillo, Simone Boesso, Arianna Celli

1. Introduction

In recent times, the world has witnessed significant challenges and changes in response to the global COVID-19 pandemic.
The necessity of wearing masks to prevent the spread of the virus has become a crucial aspect of public health measures.

As a result, the development of efficient and accurate methods for identifying individuals wearing masks has gained paramount importance in various domains, including security, healthcare, and public space management.

AWS provides a wide range of services, including storage (Amazon S3), computing resources (Amazon EC2), and serverless computing (AWS Lambda). These services offer high scalability, flexibility, and reliability, enabling the architecture to effectively handle varying workloads and seamlessly accommodate spikes in demand.
The availability of comprehensive documentation, tutorials, and support from AWS simplifies the implementation and management of the architecture, ensuring a smooth development process. Additionally, AWS ensures a secure and compliant infrastructure, assuring the privacy and security of the data being processed.
Leveraging the AWS approach empowers organizations to build and deploy highly efficient, scalable, and cost-effective object classification solutions in a professional manner.

This report presents a comprehensive overview of our cloud-based architecture designed to address the task of object classification, specifically focused on identifying individuals wearing masks.
This project aims to develop a robust and scalable solution capable of accurately detecting masked individuals in real-time.

Throughout this report, we will delve into the technical aspects, methodologies, and challenges encountered during the implementation of this cloud architecture. Additionally, we will discuss the rationale behind the selected components and technologies, highlighting the advantages they offer for this specific object detection task.

2. Dataset

The dataset (Face Mask Detection | Kaggle) used for training and testing the mask-wearing behavior classification model consists of 853 images. Each image is in JPEG format and represents individuals wearing or not wearing face masks.
The images are labeled with one of the following three classes:

- *With mask*: Depicts individuals correctly wearing a face mask.
- *Without mask*: Shows individuals not wearing a face mask.
- *Mask worn incorrectly*: Illustrates individuals wearing a face mask, but not in the correct manner.

Annotations:
The dataset provides corresponding XML files for each image, containing bounding box annotations.
These annotations define the coordinates of the bounding boxes around the faces and mask regions within the images.
The bounding box annotations are used to identify the regions of interest (ROIs) for mask detection.
Each bounding box annotation corresponds to a single face in the image and provides information about the position and dimensions of the bounding box. These annotations are crucial for training the model to accurately detect and classify mask-wearing behavior.

In summary, the dataset consists of 853 JPEG images, each associated with an XML file containing bounding box annotations. The annotations provide details about the position and size of the bounding boxes, enabling the model to identify and classify mask-wearing behavior accurately.

3. Application

Our project aimed to develop a pipeline using AWS technologies to predict if a person in an image belongs to one of three classes: wearing a mask, not wearing a mask, or wearing a mask incorrectly.

Initially, we explored two approaches for achieving this task: utilizing a Yolov5 architecture and a Resnet34 architecture.
However, we encountered numerous challenges while attempting to install the dependencies required for the Yolov5 architecture.

After successfully setting up our Jupyter notebook, we faced difficulties with installing Torch, and even after resolving this issue, we were unable to proceed with the training phase due to the inability to access the already installed libraries in the environment.

Consequently, we decided to proceed with the Resnet34 architecture and performed training using the following data splits: 60% of the dataset was allocated for the training phase 20% of the dataset was used for validation 20% of the dataset was reserved for testing.
The results obtained by this approach for the training will be enlisted later in the report.

Despite the challenges encountered during the project, we were able to successfully develop the application using the Resnet34 architecture and train it on the provided dataset.

These results mark a significant milestone in achieving our goal of accurately classifying individuals based on their mask-wearing behavior.

4. Infrastructure

Infrastructure and Training Process
Our infrastructure is built upon four key services: EC2, S3, Lambda AWS, and DynamoDB.

1. **EC2**:
   We utilized EC2 instances with the appropriate roles (LabRole) to access files in our S3 bucket (*‘ourcloudcomputingbucket’*). During the training process, we faced challenges related to the resource requirements of the model.
   Specifically, we had to vertically scale our EC2 instances by increasing RAM and storage capacity to ensure successful model training. To mitigate issues such as kernel death during the loading phase, we utilized checkpoints.

2. **S3**:
   We stored our image data in an S3 bucket along with the annotations. The bucket was accessed by the EC2 instances for training purposes and ultimately for the validation. Proper permissions and roles (LabRole) were assigned to the EC2 instances to enable seamless access to the S3 storage.

3. **AWS Lambda**:
   This service played a crucial role in deploying and utilizing the trained model. We tried to configure Lambda to trigger the prediction code at specified intervals. To achieve this, we set appropriate roles to access the S3 bucket (LabRole) and configured a timeout of 1 hour for the benchmark computation.

4. **DynamoDB**:
   We explored using DynamoDB to record and store the benchmarks obtained during the training and testing phases in a table that we created called “*benchmark*”. This allowed us to track and analyze the performance of our pipeline effectively.

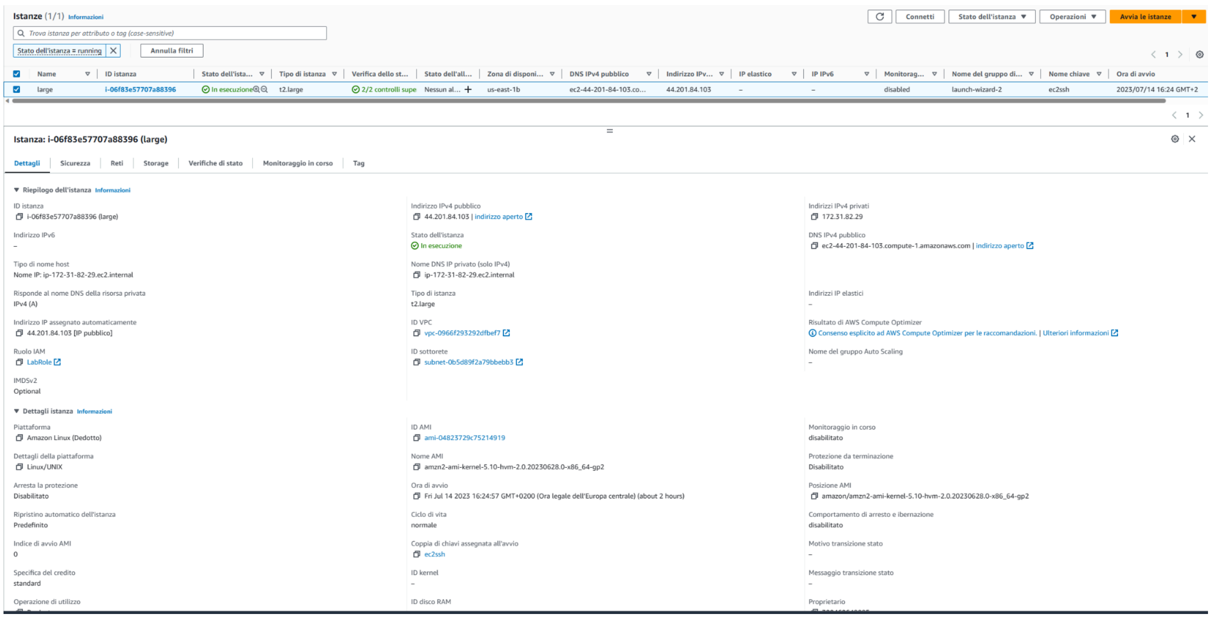
5. Development

For the development of our cloud application we proceeded in many steps with different turnbacks and errors encountered, here we will talk about the process for each of the services used

- **EC2 Instance**:
  After many trials with different kind of machines, we opter to set up an EC2 instance with the following specifications:
  - Instance Type: **t2.large**
  - RAM: 8 GB
  - vCPUs: 2

In this way we managed to surpass many problems that we had in installing the dependencies given the low RAM of the other instances given that we tried to be cost efficient at most.

Below the details of our chosen instance:



Istanze (1/1) Informazioni

Trova istanza per attributo o tag (case-sensitive)

Stato dell'istanza = running

Name	ID istanza	Stato dell'ista...	Tipo di istanza	Verifica dello st...	Stato dell'all...	Zona di disponi...	DNS IPv4 pubblico	Indirizzo IPv...	IP elastico	IP IPv6	Monitorag...	Nome del gruppo di...	Nome chiave	Ora di avvio
large	i-06f83e57707a88396	In esecuzione	t2.large	2/2 controlli supe	Nessun al...	us-east-1b	ec2-44-201-84-103.co...	44.201.84.103	–	–	disabled	launch-wizard-2	ec2ssh	2023/07/14 16:24 GMT+2

Istanza: i-06f83e57707a88396 (large)

Riepilogo dell'istanza Informazioni

ID istanza i-06f83e57707a88396 (large)	Indirizzo IPv4 pubblico 44.201.84.103 | indirizzo aperto	Indirizzi IPv4 privati 172.31.82.29
Indirizzo IPv6 –	Stato dell'istanza In esecuzione	DNS IPv4 pubblico ec2-44-201-84-103.compute-1.amazonaws.com | indirizzo aperto
Tipo di nome host Nome IP: ip-172-31-82-29.ec2.internal	Nome DNS IP privato (solo IPv4) ip-172-31-82-29.ec2.internal	
Risponde al nome DNS della risorsa privata IPv4 (A)	Tipo di istanza t2.large	Indirizzi IP elastici –
Indirizzo IP assegnato automaticamente 44.201.84.103 [IP pubblico]	ID VPC vpc-0966f293292dfbef7	Risultato di AWS Compute Optimizer Consenso esplicito ad AWS Compute Optimizer per le raccomandazioni. | Ulteriori informazioni
Ruolo IAM LabRole	ID sottorete subnet-0b5d89f2a79bbebb3	Nome del gruppo Auto Scaling –
IMDSv2 Optional		

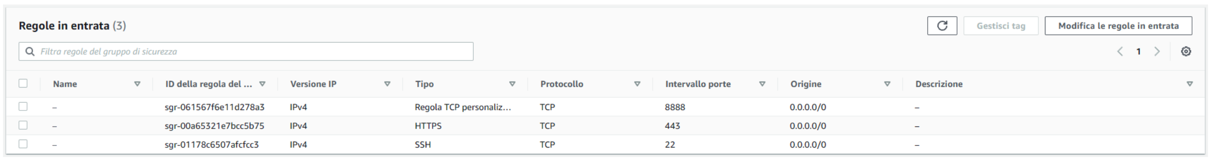
Dettagli istanza Informazioni

Piattaforma Amazon Linux (Dedotto)	ID AMI ami-04823729c75214919	Monitoraggio in corso disabilitato
Dettagli della piattaforma Linux/UNIX	Nome AMI amzn2-ami-kernel-5.10-hvm-2.0.20230628.0-x86_64-gp2	Protezione da terminazione Disabilitato
Arresta la protezione Disabilitato	Ora di avvio Fri Jul 14 2023 16:24:57 GMT+0200 (Ora legale dell'Europa centrale) (about 2 hours)	Posizione AMI amazon/amzn2-ami-kernel-5.10-hvm-2.0.20230628.0-x86_64-gp2
Ripristino automatico dell'istanza Predefinito	Ciclo di vita normale	Comportamento di arresto e ibernazione disabilitato
Indice di avvio AMI 0	Coppia di chiavi assegnata all'avvio ec2ssh	Motivo transizione stato –
Specifica del credito standard	ID kernel –	Messaggio transizione stato –
Operazione di utilizzo	ID disco RAM	Proprietario

- **Environment**:
  To train the model that is used as the final validator we had to create an empty ***conda environment*** (*py_env*) and install on it the dependencies needed:
    - **Torch** 2.0.1
      The most important package that we needed to install in order to train, test and then validate the model, the whole concept of the project revolves around this.
    - Pandas 2.0.3
    - Numpy 1.25.21
    - Seaborn
    - Matplotlib
    - **Boto3**
      Which allowed us to communicate both with the S3 bucket and the DynamoDB.
    - Pickle

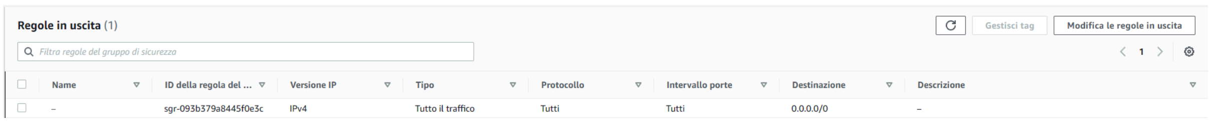
We installed **Jupyter Notebook** thanks to *conda* to have a better user experience in the development of the code to train our model.

To connect the EC2 to an external URL we created a new security group with different rules to enable ingoing…


Regole in entrata (3)
Gestisci tag
Modifica le regole in entrata
Filtra regole del gruppo di sicurezza
1

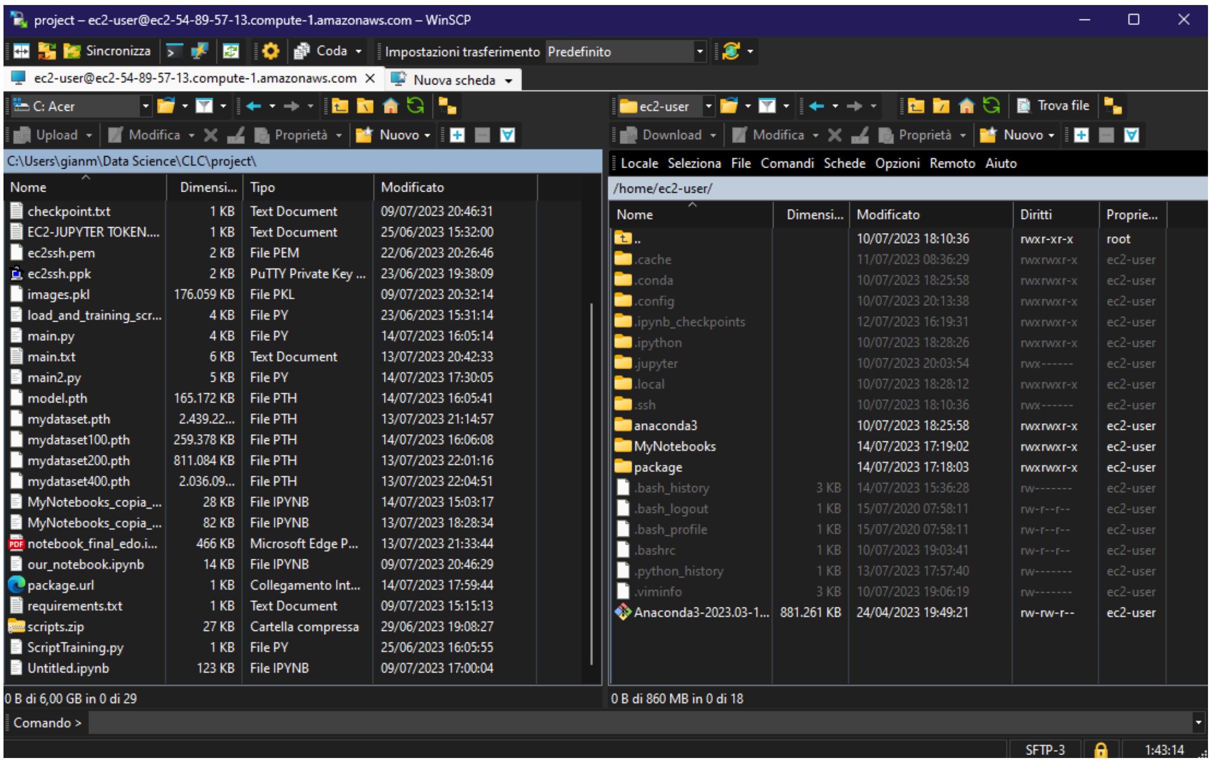
Name	ID della regola del ...	Versione IP	Tipo	Protocollo	Intervallo porte	Origine	Descrizione
–	sgr-061567f6e11d278a3	IPv4	Regola TCP personaliz...	TCP	8888	0.0.0.0/0	–
–	sgr-00a65321e7bcc5b75	IPv4	HTTPS	TCP	443	0.0.0.0/0	–
–	sgr-01178c6507afcfcc3	IPv4	SSH	TCP	22	0.0.0.0/0	–

…and outgoing traffic


Regole in uscita (1)
Gestisci tag
Modifica le regole in uscita
Filtra regole del gruppo di sicurezza
1

Name	ID della regola del ...	Versione IP	Tipo	Protocollo	Intervallo porte	Destinazione	Descrizione
–	sgr-093b379a8445f0e3c	IPv4	Tutto il traffico	Tutti	Tutti	0.0.0.0/0	–

To make it easy to manage files on the instance we used a third party software called **WinSCP**:


project – ec2-user@ec2-54-89-57-13.compute-1.amazonaws.com – WinSCP
Sincronizza
Coda
Impostazioni trasferimento
Predefinito
ec2-user@ec2-54-89-57-13.compute-1.amazonaws.com
Nuova scheda
C: Acer
Upload
Modifica
Proprietà
Nuovo
ec2-user
Trova file
Download
Locale Seleziona File Comandi Schede Opzioni Remoto Aiuto
Comando >
SFTP-3
1:43:14

C:\Users\gianm\Data Science\CLC\project\

Nome	Dimensi...	Tipo	Modificato
checkpoint.txt	1 KB	Text Document	09/07/2023 20:46:31
EC2-JUPYTER TOKEN....	1 KB	Text Document	25/06/2023 15:32:00
ec2ssh.pem	2 KB	File PEM	22/06/2023 20:26:46
ec2ssh.ppk	2 KB	PuTTY Private Key ...	23/06/2023 19:38:09
images.pkl	176.059 KB	File PKL	09/07/2023 20:32:14
load_and_training_scr...	4 KB	File PY	23/06/2023 15:31:14
main.py	4 KB	File PY	14/07/2023 16:05:14
main.txt	6 KB	Text Document	13/07/2023 20:42:33
main2.py	5 KB	File PY	14/07/2023 17:30:05
model.pth	165.172 KB	File PTH	14/07/2023 16:05:41
mydataset.pth	2.439.22...	File PTH	13/07/2023 21:14:57
mydataset100.pth	259.378 KB	File PTH	14/07/2023 16:06:08
mydataset200.pth	811.084 KB	File PTH	13/07/2023 22:01:16
mydataset400.pth	2.036.09...	File PTH	13/07/2023 22:04:51
MyNotebooks_copia_...	28 KB	File IPYNB	14/07/2023 15:03:17
MyNotebooks_copia_...	82 KB	File IPYNB	13/07/2023 18:28:34
notebook_final_edo.i...	466 KB	Microsoft Edge P...	13/07/2023 21:33:44
our_notebook.ipynb	14 KB	File IPYNB	09/07/2023 20:46:29
package.url	1 KB	Collegamento Int...	14/07/2023 17:59:44
requirements.txt	1 KB	Text Document	09/07/2023 15:15:13
scripts.zip	27 KB	Cartella compressa	29/06/2023 19:08:27
ScriptTraining.py	1 KB	File PY	25/06/2023 16:05:55
Untitled.ipynb	123 KB	File IPYNB	09/07/2023 17:00:04

0 B di 6,00 GB in 0 di 29

/home/ec2-user/

Nome	Dimensi...	Modificato	Diritti	Proprie...
..		10/07/2023 18:10:36	rwxr-xr-x	root
.cache		11/07/2023 08:36:29	rwxrwxr-x	ec2-user
.conda		10/07/2023 18:25:58	rwxrwxr-x	ec2-user
.config		10/07/2023 20:13:38	rwxrwxr-x	ec2-user
.ipynb_checkpoints		12/07/2023 16:19:31	rwxrwxr-x	ec2-user
.ipython		10/07/2023 18:28:26	rwxrwxr-x	ec2-user
.jupyter		10/07/2023 20:03:54	rwx------	ec2-user
.local		10/07/2023 18:28:12	rwxrwxr-x	ec2-user
.ssh		10/07/2023 18:10:36	rwx------	ec2-user
anaconda3		10/07/2023 18:25:58	rwxrwxr-x	ec2-user
MyNotebooks		14/07/2023 17:19:02	rwxrwxr-x	ec2-user
package		14/07/2023 17:18:03	rwxrwxr-x	ec2-user
.bash_history	3 KB	14/07/2023 15:36:28	rw-------	ec2-user
.bash_logout	1 KB	15/07/2020 07:58:11	rw-r--r--	ec2-user
.bash_profile	1 KB	15/07/2020 07:58:11	rw-r--r--	ec2-user
.bashrc	1 KB	10/07/2023 19:03:41	rw-r--r--	ec2-user
.python_history	1 KB	13/07/2023 17:57:40	rw-------	ec2-user
.viminfo	3 KB	10/07/2023 19:06:19	rw-------	ec2-user
Anaconda3-2023.03-1...	881.261 KB	24/04/2023 19:49:21	rw-rw-r--	ec2-user

0 B di 860 MB in 0 di 18

To connect we used the keys stored in the “*.pem*” file extracted upon the creation of our EC2.

- **S3 Bucket**:
  The creation of the S3 Bucket was very smooth, in this we uploaded the images and the annotations as well as the model and the dataset (image preprocessed).

  To access the S3 bucket from our newly created instance we had to grant the appropriate permissions (**LabInstanceProfile**) to the instance and modify its policy to allow access to the files.

Now that our environment is ready and the services are well aligned we can proceed to the pre-processing of the data and the training of our model.

- **Training**

  We made the decision to switch from using the Yolov5 architecture to the Resnet34 architecture as our pre-trained model.

  The primary reason for this change was the difficulties we encountered during the installation of the dependencies required for Yolov5. Despite following the installation guidelines, the Yolov5 package was unable to locate the installed dependencies.

  We suspect that this issue stemmed from a conflict between library different versions.

  To proceed with our project and ensure a viable alternative, we opted to explore the Resnet34 architecture. Resnet34 has been widely used in the literature for similar tasks and has proven to be effective in various image classification tasks, including mask-wearing behavior classification.

  By switching to Resnet34, we aimed to leverage the pre-trained model's capabilities and benefit from its established performance. This decision was driven by the need to overcome the challenges faced with Yolov5 and continue our project smoothly.

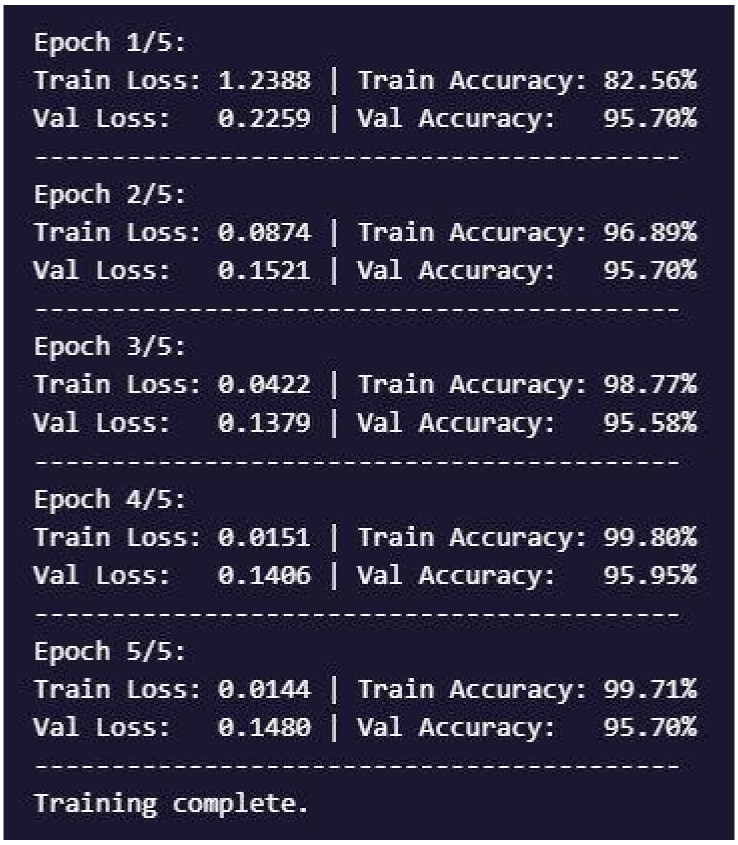
  Thus the problem is not of object detection but of object classification, where in each image with multiple masks, multiple objects to classify are derived.

Once we obtain the dataset with the tuples of tensors representing the object to identify, we proceed to train our model as follows:

```python
import torch
import torch.nn as nn
import torch.optim as optim

model=models.resnet34(pretrained=True)
criterion=torch.nn.CrossEntropyLoss()
optimizer=optim.SGD(model.parameters(),lr=0.001,momentum=0.9)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def train(model, criterion, optimizer, train_loader, val_loader, num_epochs):
    model = model.to(device)
    criterion = criterion.to(device)

    for epoch in range(num_epochs):
        model.train()
        train_loss = 0.0
        train_correct = 0
        train_total = 0

        for inputs, labels in train_loader:
            inputs = inputs.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            # forward pass
            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # backward pass and optimization
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            train_total += labels.size(0)
            train_correct += (predicted == labels).sum().item()

        train_loss /= len(train_loader)
        train_accuracy = 100.0 * train_correct / train_total

        # validation
        model.eval()
        val_loss = 0.0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs = inputs.to(device)
                labels = labels.to(device)

                outputs = model(inputs)
                loss = criterion(outputs, labels)

                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                val_total += labels.size(0)
                val_correct += (predicted == labels).sum().item()

            val_loss /= len(val_loader)
            val_accuracy = 100.0 * val_correct / val_total

        # print epoch statistics
        print(f"Epoch {epoch+1}/{num_epochs}:")
        print(f"Train Loss: {train_loss:.4f} | Train Accuracy: {train_accuracy:.2f}%")
        print(f"Val Loss:   {val_loss:.4f} | Val Accuracy:   {val_accuracy:.2f}%")
        print("------------------------------------------")

    print("Training complete.")

num_epochs = 5
learning_rate = 0.0001
optimizer = optim.SGD(model.parameters(),lr=learning_rate,momentum=0.9)

start_time = time.time()
train(model, criterion, optimizer, train_loader, val_loader, num_epochs)
elapsed_time = time.time() - start_time
```

Given that the pre-trained was already giving high accuracy in the first epoch, we chose to train it for 5 epochs overall, obtaining very satisfying results given a train set of 60%, test and val set both of 20%:

Throughout this phase of training, we encountered challenges related to resource limitations, which necessitated scaling the EC2 instance. We also implemented checkpoints to ensure progress could be saved in case of kernel death.
We trained the model using the concept of transfer knowledge, so we took the pretrained model, we froze the first 6 layers and finally we fine tuned the model.



```
Epoch 1/5:
Train Loss: 1.2388 | Train Accuracy: 82.56%
Val Loss:   0.2259 | Val Accuracy:   95.70%
------------------------------------------
Epoch 2/5:
Train Loss: 0.0874 | Train Accuracy: 96.89%
Val Loss:   0.1521 | Val Accuracy:   95.70%
------------------------------------------
Epoch 3/5:
Train Loss: 0.0422 | Train Accuracy: 98.77%
Val Loss:   0.1379 | Val Accuracy:   95.58%
------------------------------------------
Epoch 4/5:
Train Loss: 0.0151 | Train Accuracy: 99.80%
Val Loss:   0.1406 | Val Accuracy:   95.95%
------------------------------------------
Epoch 5/5:
Train Loss: 0.0144 | Train Accuracy: 99.71%
Val Loss:   0.1480 | Val Accuracy:   95.70%
------------------------------------------
Training complete.
```

Thanks to torch utilities we saved our model in a file called "*model.pth*" to keep track of it and to easily pass it and be used by different sources (AWS Lambda).

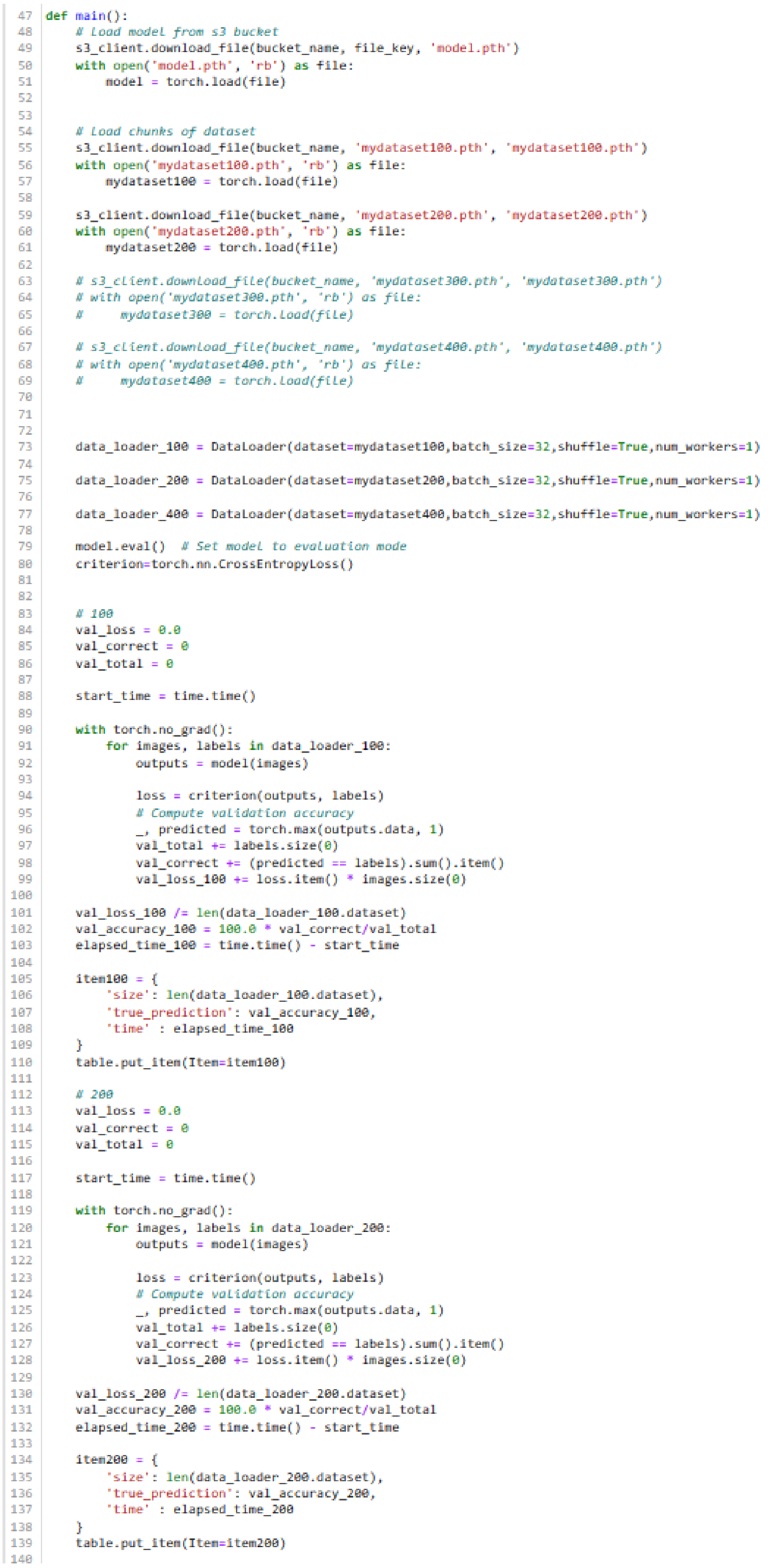
- **AWS Lambda**:
  The main idea in our project was to use AWS Lambda to test our model by triggering the activation thanks to an **EventBridge** and to perform an identification for a given set of images and to store some values inside a **DynamoDB** directly from the AWS Lambda function.

  Unfortunately we encountered many problems in this part of the development and couldn't resolve them all, but we still want to illustrate the main idea behind this piece of our architecture and the work done.

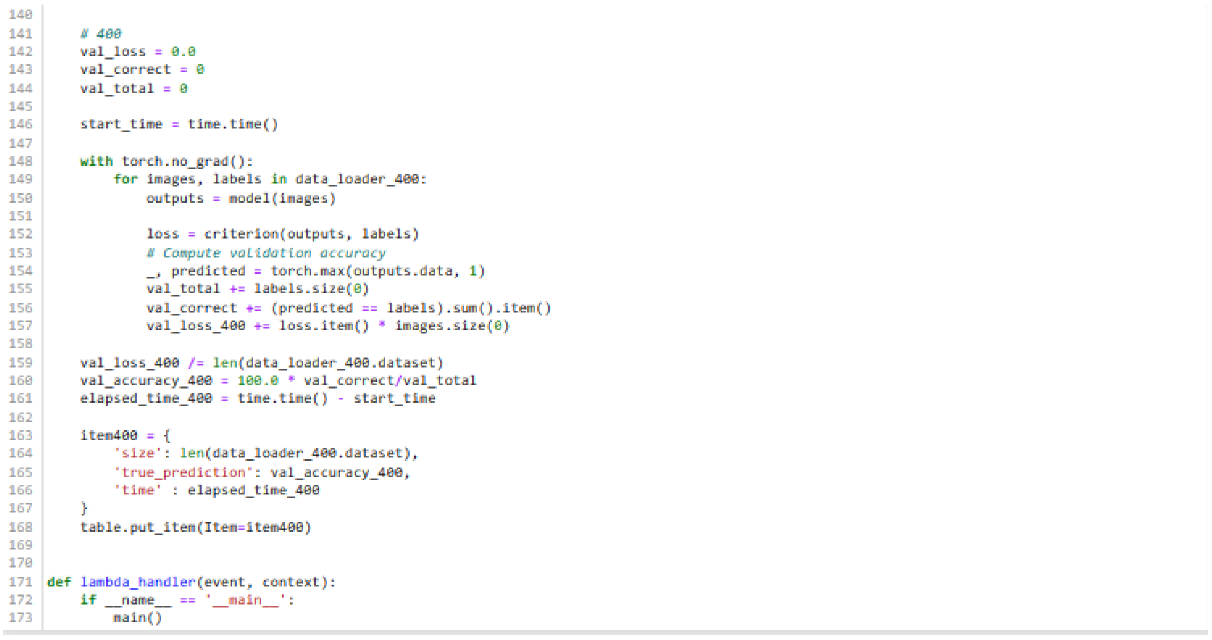
  Given that the code we wanted to be executed on the AWS Lambda needed dependencies such as torch, we had to deploy a package with a 'ready-to-go' environment (*landa_env*) and a main file with the testing in clear.
  To do so we created a folder called 'package' with the structure:

  ```
  | code
     | main.py
     | unzip_torch.py
  | lambda_env
  | deploy.sh
  ```

  In the code repository we can find the **main.py** which contains the famous **lambda_handler**, i.e. the function that will be executed when the EventBridge is triggered:



```python
def main():
    # Load model from s3 bucket
    s3_client.download_file(bucket_name, file_key, 'model.pth')
    with open('model.pth', 'rb') as file:
        model = torch.load(file)


    # Load chunks of dataset
    s3_client.download_file(bucket_name, 'mydataset100.pth', 'mydataset100.pth')
    with open('mydataset100.pth', 'rb') as file:
        mydataset100 = torch.load(file)

    s3_client.download_file(bucket_name, 'mydataset200.pth', 'mydataset200.pth')
    with open('mydataset200.pth', 'rb') as file:
        mydataset200 = torch.load(file)

    # s3_client.download_file(bucket_name, 'mydataset300.pth', 'mydataset300.pth')
    # with open('mydataset300.pth', 'rb') as file:
    #     mydataset300 = torch.load(file)

    # s3_client.download_file(bucket_name, 'mydataset400.pth', 'mydataset400.pth')
    # with open('mydataset400.pth', 'rb') as file:
    #     mydataset400 = torch.load(file)



    data_loader_100 = DataLoader(dataset=mydataset100,batch_size=32,shuffle=True,num_workers=1)

    data_loader_200 = DataLoader(dataset=mydataset200,batch_size=32,shuffle=True,num_workers=1)

    data_loader_400 = DataLoader(dataset=mydataset400,batch_size=32,shuffle=True,num_workers=1)

    model.eval()  # Set model to evaluation mode
    criterion=torch.nn.CrossEntropyLoss()


    # 100
    val_loss = 0.0
    val_correct = 0
    val_total = 0

    start_time = time.time()

    with torch.no_grad():
        for images, labels in data_loader_100:
            outputs = model(images)

            loss = criterion(outputs, labels)
            # Compute validation accuracy
            _, predicted = torch.max(outputs.data, 1)
            val_total += labels.size(0)
            val_correct += (predicted == labels).sum().item()
            val_loss_100 += loss.item() * images.size(0)

    val_loss_100 /= len(data_loader_100.dataset)
    val_accuracy_100 = 100.0 * val_correct/val_total
    elapsed_time_100 = time.time() - start_time

    item100 = {
        'size': len(data_loader_100.dataset),
        'true_prediction': val_accuracy_100,
        'time' : elapsed_time_100
    }
    table.put_item(Item=item100)

    # 200
    val_loss = 0.0
    val_correct = 0
    val_total = 0

    start_time = time.time()

    with torch.no_grad():
        for images, labels in data_loader_200:
            outputs = model(images)

            loss = criterion(outputs, labels)
            # Compute validation accuracy
            _, predicted = torch.max(outputs.data, 1)
            val_total += labels.size(0)
            val_correct += (predicted == labels).sum().item()
            val_loss_200 += loss.item() * images.size(0)

    val_loss_200 /= len(data_loader_200.dataset)
    val_accuracy_200 = 100.0 * val_correct/val_total
    elapsed_time_200 = time.time() - start_time

    item200 = {
        'size': len(data_loader_200.dataset),
        'true_prediction': val_accuracy_200,
        'time' : elapsed_time_200
    }
    table.put_item(Item=item200)
```



```python

    # 400
    val_loss = 0.0
    val_correct = 0
    val_total = 0

    start_time = time.time()

    with torch.no_grad():
        for images, labels in data_loader_400:
            outputs = model(images)

            loss = criterion(outputs, labels)
            # Compute validation accuracy
            _, predicted = torch.max(outputs.data, 1)
            val_total += labels.size(0)
            val_correct += (predicted == labels).sum().item()
            val_loss_400 += loss.item() * images.size(0)

    val_loss_400 /= len(data_loader_400.dataset)
    val_accuracy_400 = 100.0 * val_correct/val_total
    elapsed_time_400 = time.time() - start_time

    item400 = {
        'size': len(data_loader_400.dataset),
        'true_prediction': val_accuracy_400,
        'time' : elapsed_time_400
    }
    table.put_item(Item=item400)


def lambda_handler(event, context):
    if __name__ == '__main__':
        main()
```

The **unzip_torch.py** is a file thinked to fit the package in the size limit (250MB) of deployment package, the idea was to compress the torch library and at runtime decompress it under a '/tmp' directory which grants an additional 500MB of storage.

Finally, the **deploy.sh** is a script built to (of course) deploy the package to the AWS Lambda function.

The main problem we found here was the size of the package, although the management of the memory of torch's dependencies we couldn't diminish the size of our package even after compressing it to a zip.
And when we tried to download it or to put it on the S3 bucket to be accessible the AWS couldn't respond in any way resulting in the interruption of the connection.

- **Alternative solution**:

  Instead of using the AWS Lambda, we used the same code (except for the lambda_handler function of course) in a python file inside the EC2 and through the notebook we managed to call the testing for the given set of images.

The set of images we chose to make the test on are three of the size, respectively, of 100, 200 and 400 images.

Each image could contain more masks than just one, so the detection can be performed more than once per image to detect multiple masks.

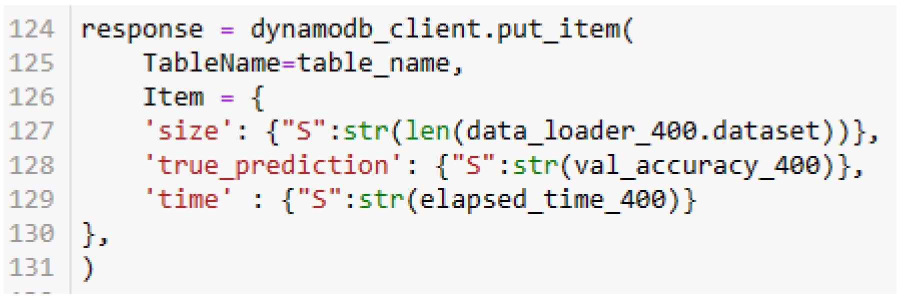
For every set we want to perform a test and gather information about it using the model we talked about earlier.

The properties in which we are interested are:

- **Size** of the set (represented in number of objects to detect, not in images)
- **Time** took by the model to analyze the whole set
- Percentage of **predictions** made by the model, indicating its efficiency

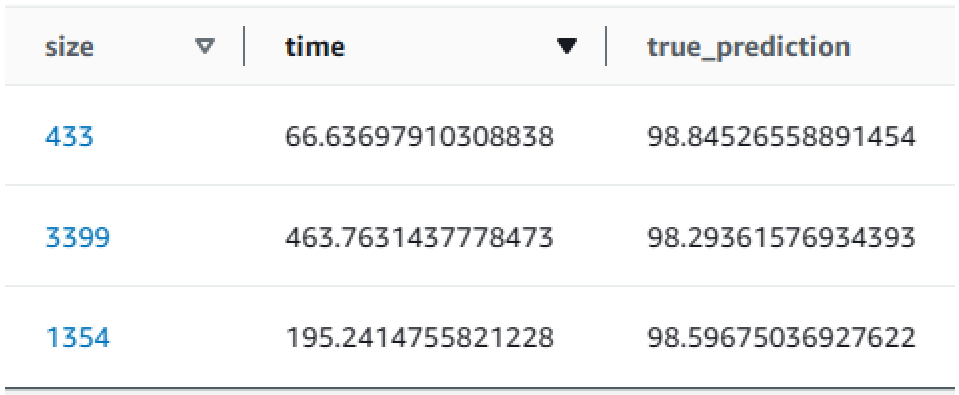
These will be stored in different records inside the **DynamoDB** table that we early created, called *benchmark*, where the partition key is the size of the set.

To access the DynamoDB was sufficient to instantiate a client thanks to boto3 and then upload the records using the JSON format:



```
response = dynamodb_client.put_item(
    TableName=table_name,
    Item = {
    'size': {"S":str(len(data_loader_400.dataset))},
    'true_prediction': {"S":str(val_accuracy_400)},
    'time' : {"S":str(elapsed_time_400)}
},
)
```

The results obtained can be seen in the table below, extracted directly from our DynamoDB “benchmark”.



size	time	true_prediction
433	66.63697910308838	98.84526558891454
3399	463.7631437778473	98.29361576934393
1354	195.2414755821228	98.59675036927622

As we can see the results obtained are in line with what we could expect especially for the prediction.

6. Conclusion

Taking a look at the results obtained it seems that they are in line with what we expected, the increment in time of execution scales linearly with the size of each set of analysis.
Moreover the percentage of correct prediction is very high and remains constant even if the samples to analyze are much bigger, implying a good reliability of the model and the project in general.

The cost efficiency has also been a key aspect of our development and we managed to always stay at low costs for the requested services, also thanks to the free tier plan that allowed us to use many resources for free.

Some limitations of the Learner Lab have been a bit harsh to handle, such as the impossibility of creating an IAM role that we wanted to use to make the instances and services available for different users.

But these kind of obstacles have been easily overridden and we can affirm that the Learner Lab is an excellent tool to take confidence with the most used cloud infrastructures.

In conclusion we can say that this project provided for us a valuable opportunity to acquire and apply essential tools in the realm of cloud technologies, which have a significant impact on our professional lives.

Our choice to undertake this challenging project aimed at gaining a comprehensive understanding of potential obstacles and improving our skills. Even the challenges we faced during the Lambda AWS segment served as valuable learning experiences, helping us grasp the associated difficulties and identify areas for further improvement and learning.