Deep Face Detection Model - Milestone #2

Brayan Leonardo Gil Guevara (C0 902422)

Eduardo Williams (C0896405)

Marzieh Mohammadi Kokaneh (C089839)

Rohit Kumar (C0895100)

Saurabh Laltaprasad Gangwar (C0894380)

Lambton College

Big Data Analytics

BDM 3035 - Big Data Capstone Project 01

PhD. Meysam Effati

Mississauga, Ontario

June 18th, 2024

Contents

Introduction

The face recognition industry has been crucial for security purposes over the years, however, once COVID hit in late 2019, more requirements for this technology have been raised.

This document explores the use of deep learning for facial recognition. The Idea comes from creating a local dataset having different photos of people on a team and training the model to detect the identity of each person. We will discuss how to collect and prepare data, as well as develop models for this combined task. The target for this project is security companies for letting people to enter the office or not!

For this specific document we are focusing on Data Splitting Process and Data Augmentation as we are getting ready for NL Model Training and Performance Analysis in future steps.

Progress Report

As mentioned before, this Milestone will focus on two essential steps for our model, data splitting and data augmentation, let's dive into it.

Data Splitting Process

For this step, we had to manually take all the jpg files from the images folder and distribute them as follows:

Train folder →70%

Test folder→ 15%

Valid folder → 15%

So, we took all our pictures (90 pictures per every team member and 10 non face related pictures) and divided them into three folders, train, test and valid. Of course we then had to move all of the .json files containing the labels, and to make efficient this step, the following code was implemented with assistance of the import os library:

```
# Move the matching labels
for folder in ['train', 'test', 'valid']:
    for file in os.listdir(os.path.join('data', folder, 'images')):
        filename = file.split('.')[0] + '.json'
        existing_filepath = os.path.join('data', 'labels', filename)
        if os.path.exists(existing_filepath):
            new_filepath = os.path.join('data', folder, 'labels', filename)
            os.replace(existing_filepath, new_filepath)
```

Figure 1. Moving the matching labels <code>

The previous code helped manually identify each picture and move its corresponding. json label file.

Having the folders properly separated allowed us to move to the next step, the augmentation process.

Data Augmentation Process

This stage will cover the data augmentation pipeline using the Albumentations library specifically for images with bounding boxes. Albumentations is a computer vision tool that boosts the performance of deep convolutional neural networks. (Buslaev et al., 2020)

This technology is being used by big tech companies like IBM, Salesforce, SONY, Hugging Face, Alibaba, Amazon, Apple, Google, Meta, NVIDIA, the list can store more names but naming them just helps understand the purpose of using a reliable tool.

But before getting there we needed to install tensorflow, opencv-python and albumentations

```
# installin tensorflow, opencv and albumentations libraries
%pip install tensorflow opencv-python albumentations
```

```
# updating tensorflow
%pip install --upgrade tensorflow
```

```
# upgrading albumentations
%pip install --upgrade albumentations
```

Figure 2. Installing and updating required libraries <code>

Next thing we want to do is declaring all of our libraries that we'll be using for this stage:

```
import pandas as pd
import numpy  as np
import os
import cv2
import json
import albumentations as alb

from matplotlib import pyplot as plt

# Importing layers and base network
import tensorflow as tf
```

Figure 3. Importing required libraries <code>

So the way we initialize this stage is by defining the paths were our images and labels are

```
# defining paths
dir_train_img = "data/train/images/"
diraug_train_img = "data_aug/train/images/"
diraug_train_lab = "data_aug/train/labels/"
diraug_test_img = "data_aug/test/images/"
diraug_test_lab = "data_aug/test/labels/"
diraug_valid_img = "data_aug/valid/images/"
diraug_valid_lab = "data_aug/valid/labels/"
```

Figure 4. Defiining paths <code>

As we will be working with lots of images, it becomes easier defining a function to load them. Same thing is going to happen with the labels made with labelme in our previous stage project:

```
# defining load_image function

def load_image(x):
    byte_img = tf.io.read_file(x)
    img = tf.io.decode_jpeg(byte_img)
    return img
```

```
# defining load_labels function

def load_labels(label_path):
    with open(label_path.numpy(), 'r', encoding = "utf-8") as f:
         label = json.load(f)

    return [label['class']], label['bbox']
```

Figure 5. load_image and load_labels functions <code>

Next step is to create a TensorFlow Dataset object that contains a list of all the images in the train_img path that we defined a prior steps back

```
# verifying images
images = tf.data.Dataset.list_files(dir_train_img + '*.jpg', shuffle = False)
```

Figure 6. TensorFlow Dataset object creation <code>

We then try to access the recently created TensorFlow Dataset object and try to get the filet path of the first image in the training dataset as a byte string

```
images.as_numpy_iterator().next()
```

```
Out[51]:
b'data\\train\\images\\EDU_0001.JPG'
```

Figure 7. Fist element in our TensorFlow Dataset object <code>

Then, we apply the load_image function declared before to each element un our *images* object by using a map function.

```
# loading images
images = images.map(load_image)
```

Figure 8. Load images using the function created <code>

Our next step is to covert the Dataset object (images) into a Numpy iterator by using *.as_numpy_iterator()* and show the first decoded image tensor from the dataset:

```
images.as_numpy_iterator().next()
```

Figure 9. Converting Dataset object into a NumPy iterator. <code>

The output obtained here is an array that represents a single decoded image from the dataset, the array has:

- Height of the image (number of rows)
- Width of the image (number of columns)
- Number of color channels (3 for RGB)

Next thing done here was viewing the batches of images from our TensorFlow dataset using matplotlib. We specified a batch size of 4, meaning that our whole dataset (images) will be split into groups of 4 images and the iterator will iterate through these batches

```
# viewing raw images with Matplotlib
image_generator = images.batch(4).as_numpy_iterator()
```

Figure 10. Viewing raw images with Matplotlib <code>

We then retrieve a batch of images from the dataset using the iterator we created before, create a suitable layout with subplots using Matplotlib, and then display each image in the batch on its own subplot within the figure

```
plot_images = image_generator.next()

fig, ax = plt.subplots(ncols=4, figsize=(20,20))

for idx, image in enumerate(plot_images):
    ax[idx].imshow(image)

plt.show()
```

Figure 11. Displaying each image in the batch <code>

Now we are ready to jump into the augmentor pipeline creation

```
# setup albumentations transform pipeline

augmentor = alb.Compose([alb.RandomCrop(width=480, height=480),
            alb.HorizontalFlip(p=0.5),
            alb.RandomBrightnessContrast(p=0.2),
            alb.RandomGamma(p=0.2),
            alb.RGBShift(p=0.2),
            alb.VerticalFlip(p=0.5)],
            bbox_params = alb.BboxParams(format='albumentations',
                                         label_fields=['class_labels']))
```

Figure 12. Albumentations transform pipeline

How does this small piece of code works? *alb* refers to this library and it is being initialized by creating a sequence of image transformation steps by calling *alb.Compose*. Now, every step inside Compose becomes a sequence, it can be seen as it was defined by squared brackets [].

alb.RandomCrop(width=480, height=480) This will take the image and crops it to a size of 480x480 randomly. This becomes useful for generating different image patches .

alb.HorizontalFlip(p=0.5) This will flip the image horizontally with a probability of 50%. In more simple words, this step will make with a probability of 0.5 look like if the image was been seen from a mirror.

alb.RandomBrightnessContrast(p=0.2) With a probability of 20%, the image will get a different brightness and contrast.

alb.RandomGamma(p=0.2) This piece of code will randomly change the gamma value of the image with a probability of 20%

alb.RGBShift(p=0.2) This will shift the values of the RGB channels with a probability of 20 %. This will alter in other words the balance of color of the image.

alb.VerticalFlip(p=0.5) This last piece will flip the images vertically with a probability of 50%.

bbox_params = alb.BboxParams(format='albumentations',label_fields=['class_labels']) This last piece will give parameters for handling bounding boxes, This is relevant when doing object detection. First the format=’albumentations’ will specify the format of the bounding boxes. On the other hand ‘label_fields’ will specify that the bounding boxes have associated labels stored in the ‘class_labels’ field. This will guarantee that the transformations applied to the bounding boxes are also applied to their corresponding labels.

Following next, the below piece of code manages to load a test image and its corresponding annotation from disk using OpenCV for image processing and the json library for parsing JSON data.

```
# Load a Test image and annotation with OpenCV and JSON
img = cv2.imread(os.path.join('data', 'train', 'images', 'EDU_0005.jpg'))

with open(os.path.join('data', 'train', 'labels', 'EDU_0005.json'), 'r') as f:
    label = json.load(f)

label
```

Figure 13. Load a test image and annotation with OpenCV and JSON

We also wanted to test if representing the bounding box coordinates for the first shape in the annotations we loaded from the JSON file before.

```
label['shapes'][0]['points']
```

Figure 14. Extracting the bounding box points for the first shape

Next by using indexing, we assign the individual coordinates for the box coordinates of our first object in our annotations. The coordinates are store in a list initialized with 0s, here the top-left and bottom-right corner coordinates are stored.

```
coords = [0,0,0,0]
coords[0] = label['shapes'][0]['points'][0][0]
coords[1] = label['shapes'][0]['points'][0][1]
coords[2] = label['shapes'][0]['points'][1][0]
coords[3] = label['shapes'][0]['points'][1][1]
coords
```

Figure 15. Storing the bounding box coordinates

This will be used later to normalize those coordinates to a range between 0 and 1 by performing element-wise division:

```
coords = list(np.divide(coords, [480,480,480,480]))
coords
```

Figure 16. Normalizing the coordinates

Now we finally apply the defined augmentation pipeline (augmentor) to our images and modify its properties

```
# Applying augmentations just for one image to view results
augmented = augmentor(image=img, bboxes=[coords], class_labels=['Eduardo'])
augmented
```

Figure 17. Applying augmentation just for one image to view results

We use next *augmented.keys()* to explore the output of the augmentation pipeline. This will display the available keys in the dictionary returned by the augmentator. We then view the NumPy array that represents the images after applying the augmentation pipeline to our original *image* by calling *augmented['image']*. We also tested that we were able to represent the coordinates of the bounding box for the test image that we transformed after applying the augmentation pipeline by using *augmented['bboxes']*

The next step includes combining OpenCV for image manipulation and Matplotlib for visualization to display the augmented image with the transformed bounding box

```
cv2.rectangle(augmented['image'],
              tuple(np.multiply(augmented['bboxes'][0][:2], [480,480]).astype(int)),
              tuple(np.multiply(augmented['bboxes'][0][2:], [480,480]).astype(int)),
              (255,0,0), 2)

plt.imshow(augmented['image'])
```

Figure 18. Visualizing the augmented image with the transformed bounding box

Results:

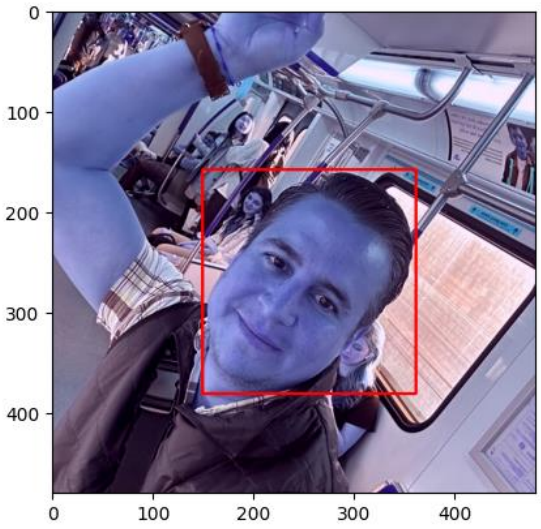

Figure 19. Results of augmentation for one image

Now we finally apply our augmentation pipeline for all the images across different partitions (train, test, valid) in our dataset.

- The code iterates through three partitions ('train', 'test', 'valid') using a nested loop.
- Within each partition, it iterates over all image files in the 'images' folder using os.listdir.
- • cv2.imread reads the current image (image) from the 'images' folder.
- • It initializes coords with a placeholder value

- The code attempts to load the corresponding annotation file (.json) from the 'labels' folder

If the annotation file exists:

- It opens the file, loads the JSON content using json.load, and assigns it to label.
- It extracts the bounding box coordinates from the loaded annotations and assigns them to the coords list.
- The coordinates are then normalized by dividing by the image dimensions (assumed to be 480x480).

Image augmentation is done as explained below:

An *try-except* block is used to handle potential errors during augmentation. Inside the *try* block, there's a loop that iterates 60 times *(for x in range(60*)) to perform augmentation on each image multiple times, meaning that for every image, we are making 60 annotations, obtaining in total *60*100=6000* images per class.

- A class number (*classNumber*) is initialized to 0
- The *augmentor* function is called to apply image augmentation to the *img* with the provided *coords* and a class label string combining partition and image name.
- The augmented image is saved using *cv2.imwrite* with a new filename that includes the original filename and an augmentation counter (`x`).

We also create and save annotations. An empty dictionary *annotation* is created to store annotation information for the augmented image. It sets the `"image"` key to the original image filename. The code checks again if the annotation file existed for the original image:

If it existed:

- It checks if any bounding boxes were present in the augmented result (*len(augmented['bboxes'])).*
- If no bounding boxes were found after augmentation, it sets a default bounding box *([0,0,0,0])* and class number (*classNumber*) in the annotation.
- Otherwise, it assumes the first bounding box *(augmented['bboxes'][0])* and potentially assigns a class number based on the original label

If the annotation file didn't exist for the original image:

- It sets default values for bounding box and class number in the annotation.

Finally, the *annotation* dictionary is saved as a JSON file using *json.dump* with a filename similar to the augmented image.

Next Steps

The next steps include working with the NL Model Training and working in the performance Analysis. This will provide all the necessary resources as we get ready for the next stage with predictions on the TEST dataset and making real time prediction testing.

Faced Challenges

The code faces several challenges that can affect its performance and accuracy. One significant assumption is that annotation files (JSON) exist for all images. If some images are missing annotations, the script might fail or generate incorrect annotations, leading to errors or inaccuracies in the data processing workflow. This issue can compromise the quality of the training data, affecting the overall performance of the model.

Another challenge is the assumption regarding specific class labels. The code assigns class numbers based on predefined labels. If the dataset contains different class labels, the logic for assigning class numbers must be adjusted accordingly. This mismatch can result in mislabeling or errors, undermining the reliability of the dataset.

Additionally, the code relies on a predefined augmentor function, making it difficult to evaluate the types of augmentations applied and their suitability without seeing the implementation. For more granular control over the augmentation process, using libraries like imgaug or exploring Albumentations configuration options could be beneficial. These alternatives offer a wider range of augmentation techniques and customization, better catering to specific needs.

Furthermore, the code performs 60 augmentations per image, which can be computationally expensive. This high number of augmentations might strain resources and extend processing time, making the augmentation process less efficient, especially with large datasets. Balancing the number of augmentations with computational efficiency is crucial to optimizing the overall workflow.

Lessons Learned

Image augmentation can be a powerful tool for expanding our dataset by creating variations of existing images. This technique is particularly valuable in computer vision tasks, where limited training data can be a significant challenge. By generating new, diverse images from the original dataset, augmentation helps improve the robustness and performance of machine learning models.

A key aspect of the code review for this milestone is its use of existing annotations, such as bounding boxes and class labels, to create corresponding annotations for the augmented images. This ensures that the association between objects in the original images and their transformed versions is preserved, maintaining the integrity of the dataset.

The code underscores the importance of carefully designing the augmentation pipeline. The choice of transformations and their specific parameters plays a crucial role in the effectiveness of the augmentation process. Well-designed augmentations can significantly enhance model performance by providing a more varied and comprehensive training dataset.

Moreover, the code highlights the necessity of robust error handling to catch potential issues during the augmentation process. Effective error handling ensures that problems are identified and addressed promptly, preventing them from compromising the quality of the augmented data. Additionally, considering the efficiency of the augmentation process is vital, especially when working with large datasets. Efficient augmentation pipelines help manage computational resources and processing time, making the overall process more practical and scalable.

In summary, the code illustrates how image augmentation can enhance our dataset, the importance of maintaining accurate annotations, the need for thoughtful design in the

augmentation pipeline, and the critical role of error handling and efficiency. These elements collectively contribute to a more effective and reliable approach to improving model performance in computer vision tasks.

Conclusion

In conclusion, we reviewed code that effectively demonstrates the application of image augmentation to artificially create variations of existing images. This technique addresses the common challenge of limited training data in computer vision tasks by expanding the dataset, enhancing model robustness and performance.

We dissected the code into its components, gaining insights into how it loads images and annotations, applies augmentations, and generates new annotations for the augmented data. This detailed examination highlighted several challenges, such as handling missing annotations, the limited control over augmentation types, and the necessity for more specific error handling.

The importance of carefully designing the augmentation pipeline to suit your dataset and achieve desired outcomes was a key takeaway. Leveraging existing annotations and ensuring data validation were also emphasized as crucial steps to maintain the quality of the augmented data.

Furthermore, we discussed how libraries like imgaug and Albumentations can offer more granular control over the augmentation process, providing a broader range of techniques and customization options. Lastly, we considered strategies to improve efficiency and error handling to better suit real-world applications, ensuring a more practical and scalable approach to image augmentation.

References

Buslaev, A., Iglovikov, V. I., Khvedchenya, E., Parinov, A., Druzhinin, M., & Kalinin, A. A. (2020). Albumentations: Fast and flexible image augmentations. *Information (Switzerland)*, *11*(2), 1–20. https://doi.org/10.3390/info11020125