The Final Project of Machine Learning - Face Mask Detector

Team Members: Jan Kalenda (24170189) & Jesse Sillman (24170088)

Introduction

This report presents the methodologies and steps undertaken to develop a machine learning system for face mask detection. The primary objective of this project was to implement an effective solution for determining whether individuals in images are wearing face masks. In practice, such systems can be deployed in setting like factors and hospitals, where wearing a face mask is madatory.

The dataset used for this project was sourced from Kaggle's "Face Mask Detection" repository, accessible at: https://www.kaggle.com/datasets/andrewmvd/face-mask-detection/code. It contained annotated images divided in three categories: "with mask", "without mask", and "mask worn incorrectly". These annotations provided the foundation for training and evaluating the model's performance.

Objective

The project's primary objectives were as follows:

1. To explore and understand the YOLO (You Only Look Once) model, a widely used for its efficiency in real-time object detection.
2. To develop and train a customised CNN (Convolutional Neural Network) model tailored for face mask detection of single person images.
3. To train and utilize RetinaNet model tailored for face mask detection and compare it with YOLO.

In addition, our aim was to successfully manipulate the dataset to filter out containing multiple persons in images and create a JSON file that includes only single-person, whether with a mask or without. This modified dataset was used to train and compare the customised CNN model.

YOLO (You Only Look Once)

YOLO is a groundbreaking object detection model introduced by Joseph Redmon in 2015. Widely known for its exceptional speed and accuracy, YOLO revolutionized the field of object detection by treating the task as a single regression problem. Unlike traditional methods that use sliding windows or region proposals, YOLO predicts bounding boxes and class probabilities for the entire image in a single forward pass of a neural network.

Yolo models generally consist of three main components:

- Backbone: The backbone is responsible for extracting features from the input image. It typically consists of convolutional layers that learn to detect patterns and features in the image.

- Neck: The neck is a series of layers that further process the features extracted by the backbone. It may include additional convolutional layers, pooling layers, and other operations to refine the features.
- Head: The head is the final part of the model that generates predictions based on the features extracted by the backbone and neck. It typically consists of convolutional layers that output bounding boxes, class probabilities, and other relevant information.

YOLO v11 [1] is the latest version of the YOLO model, which has been optimized for speed and accuracy. It is capable of detecting objects in real-time with high precision, making it ideal for applications that require fast and efficient object detection. New features of YOLOv11 include:

- C3k2 backbone, a more efficient block, replacing C2f. It is a more efficient implementation of the Cross Stage Partial (CSP [2]) Bottleneck.
- combination of SPPF (Spatial Pyramid Pooling - Fast) and a new Cross Stage Partial with Spatial Attention (C2PSA). This spatial attention mechanism allows the model to focus more effectively on important regions within the image. By pooling features spatially, the C2PSA block enables YOLO11 to concentrate on specific areas of interest, potentially improving detection accuracy for objects of varying sizes and positions. [1]
- Neck uses C3k2 block as well, with the new C2PSA module for improved spatial attention, differentiating itself from the previous YOLO v8.
- Head is responsible for the final predictions and uses the new C3k2 blocks too. Additionally, CBS (Convolution-BatchNorm-Silu) blocks are used in numerous places in the head, extracting relevant features and normalizing the data flow. Sigmoid linear unit activation function is sued to increase the performance of the model.

YOLOv11 model is capable of the following tasks:

- Object detection: Detecting objects in images and videos.
- Object tracking: Tracking objects across frames in a video.
- Pose estimation: Estimating the pose of humans or objects in images or videos.
- Instance segmentation: Segmenting images into different regions or objects.
- Image classification: Classifying images into different categories.
- Oriented object detection: Detecting objects with specific orientations.

For this project, we used YOLOv11's object detection capabilities to detect people and whether they wear a mask or not.

RetinaNet

RetinaNet is a one-stage object detection model designed to handle the problem of class imbalance in object detection. It uses a feature pyramid network (FPN) with a ResNet backbone, making it good at detecting objects of different sizes. Its key feature is the Focal Loss function, which reduces the influence of easily classified background samples and gives more weight to harder-to-detect objects.

Architecture of RetinaNet:

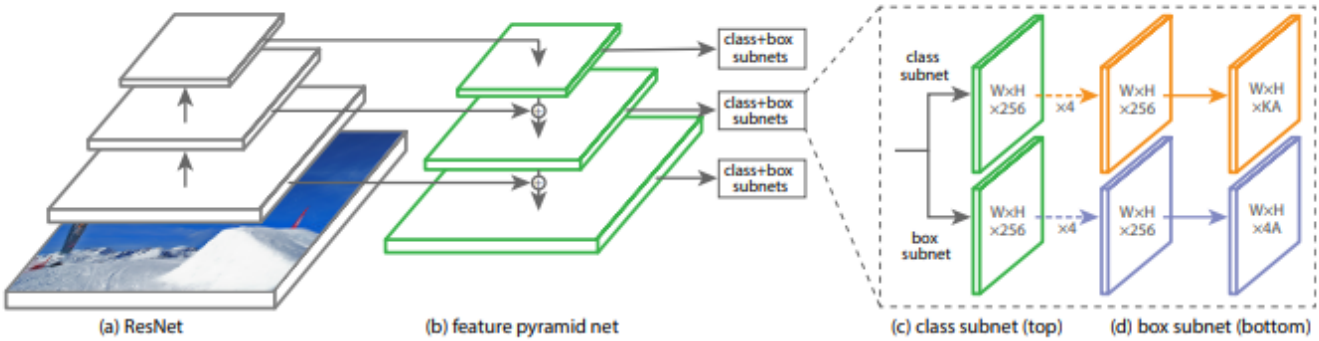


Figure 3. The one-stage **RetinaNet** network architecture uses a Feature Pyramid Network (FPN) [20] backbone on top of a feedforward ResNet architecture [16] (a) to generate a rich, multi-scale convolutional feature pyramid (b). To this backbone RetinaNet attaches two subnetworks, one for classifying anchor boxes (c) and one for regressing from anchor boxes to ground-truth object boxes (d). The network design is intentionally simple, which enables this work to focus on a novel focal loss function that eliminates the accuracy gap between our one-stage detector and state-of-the-art two-stage detectors like Faster R-CNN with FPN [20] while running at faster speeds.

Focal Loss:

- Focal Loss is a modification of the standard cross-entropy loss function that addresses the issue of class imbalance in object detection. It assigns higher weights to hard-to-detect objects, making the model focus more on these objects during training. This helps improve the model's performance on challenging cases and reduces the impact of easy-to-detect objects on the loss function. [3]

$$\mathrm{FL}(p_t) = -(1 - p_t)^{\gamma} \log(p_t).$$

Face Mask Detector using YOLO

Preprocessing steps

In the very beginning, we used the `kagglehub` library to download the dataset folder and renamed it to `face_mask_dataset` for easier access. Next, we used used Python's `xml.etree.ElementTree` module to parse the XML files, extracting the rood node for further processing:

```
import xml.etree.ElemenTree as ET

def parse_xml(xml_file):
    tree = ET.parse(xml_file)
    root = tree.getroot()
    return root
```

After parsing the XML files, the next step was to converting the bounding box into the YOLO format, which specifies each bounding box with normalized coordinates: `[class_id, x_center, y_center, width, height]` . This normalization ensures consistency across different image sizes.

The following helper function, `get_yolo_bbox` , computes the normalized values:

```
def get_yolo_bbox(xml_bbox, width, height) -> str:
    x_center = (xml_bbox[0] + xml_bbox[2]) / 2 / width
    y_center = (xml_bbox[1] + xml_bbox[3]) / 2 / height
    w = (xml_bbox[2] - xml_bbox[0]) / width
    h = (xml_bbox[3] - xml_bbox[1]) / height
    return f"{x_center} {y_center} {w} {h}"
```

Once the bounding boxes were converted, we proceeded to save the annotations in YOLO format into `.txt` files. Each file corresponds to an image and contains all the bounding boxes and their respective class IDs. The following function, `convert_to_text`, handled this process:

```
def convert_to_txt() -> None:
    data = "face_mask_dataset/annotations"
    output = "datasets/labels"
    classes = ["with_mask", "mask_weared_incorrect", "without_mask"]

    for file in os.listdir(data):
        root = ET.parse(f"{data}/{file}").getroot()
        width = int(root.find("size").find("width").text)
        height = int(root.find("size").find("height").text)
        boxes = []

        for obj in root.findall("object"):
            class_id = classes.index(obj.find("name").text)
            xml_bbox = [int(coords.text) for coords in obj.find("bndbox")]
            yolo_bbox = get_yolo_bbox(xml_bbox, width, height)
            boxes.append((class_id, yolo_bbox))

        # Write YOLO annotations to a .txt file
        with open(f"{output}/{file.split('.')[0]}.txt", "w") as f:
            for class_id, yolo_bbox in boxes:
                f.write(f"{class_id} {yolo_bbox}\n")
```

Lastly, we split the dataset into training, validation, and testing sets by creating a directory structure to organize the dataset. In this project, the dataset was splitted 70% for training, 15% for validation, and 15% for testing using the following function:

```
def create_training_set(images) -> None:
  for i, image in enumerate(images):
    if i < 0.7 * len(images):  # Training set
      shutil.move(f"face_mask_dataset/images/{image}", f"datasets/train/images/{image}")
      shutil.move(f"datasets/labels/{image.split('.')[0]}.txt", f"datasets/train/labels/{image.split('.')[0]}.t
    elif i < 0.85 * len(images):  # Validation set
      shutil.move(f"face_mask_dataset/images/{image}", f"datasets/val/images/{image}")
      shutil.move(f"datasets/labels/{image.split('.')[0]}.txt", f"datasets/val/labels/{image.split('.')[0]}.txt
    else:  # Test set
      shutil.move(f"face_mask_dataset/images/{image}", f"datasets/test/images/{image}")
      shutil.move(f"datasets/labels/{image.split('.')[0]}.txt", f"datasets/test/labels/{image.split('.')[0]}.tx
```

This function iterates through the images and moves image, alongside with its corresponding annotation file, to the appropiate folder based on the split ratio.

To finish the preprocessing, a configurarion file named `yolo.yaml` was created to specify the paths for the training and validation datasets, the number of classes, and the class names:

```
yolo_yaml = f"""train: {pathlib.Path("datasets/train/images").resolve()}
val: {pathlib.Path("datasets/val/images").resolve()}
nc: 3
names: ['with_mask', 'mask_weared_incorrect', 'without_mask']
"""

with open("yolo.yaml", "w") as f:
    f.write(yolo_yaml)
```

Training and Inference

After completing the preprocessing procedures, we proceeded to train the YOLO model using the prepared dataset. We used the `ultralyrics` library, which simplifies the process of working with YOLO models.

The training process began by initializing the YOLO model with pre-trained weights (`yolo11m.pt`), as following:

```
model = ultralytics.YOLO('yolo11m.pt')
```

Using pre-trained weights reduces training time and improves initial performance. The model was pre-trained on COCO dataset [4] and the letter `m` signifies it as an medium sized model, balancing speed and accuracy.

The model was trained on the facemask dataset using the configuration file `yolo.yaml` . The training ran for three epochs with GPU acceleration (T4 on Google Colab) for faster computation:

```
results = model.train(data="yolo.yaml", epochs=3, save=True, device=0)
```

The best performing weights were automatically saved to `runs/detect/train/weights/best.pt` . This step prepared the model for making predictions on new, unseen data.

Next, we proceeded to evaluate the model by testing it on images from the test dataset. The trained YOLO model now equipped with the best-performing weight (`best.pt`) was loaded to perform inference on unseen image. To test the model, we selected a sample image from the test dataset and ran it through the trained YOLO model using the `predict` method:

```
filename = "maksssksksss770"
final_model.predict(f"datasets/test/images/{filename}.png", save=True)
```

We can see the results of the YOLO model's predictions displayed on the sample image:

The result highlights the following:

1. Bounding boxes

- The detected objects are enclosed within bounding boxes generated by the model

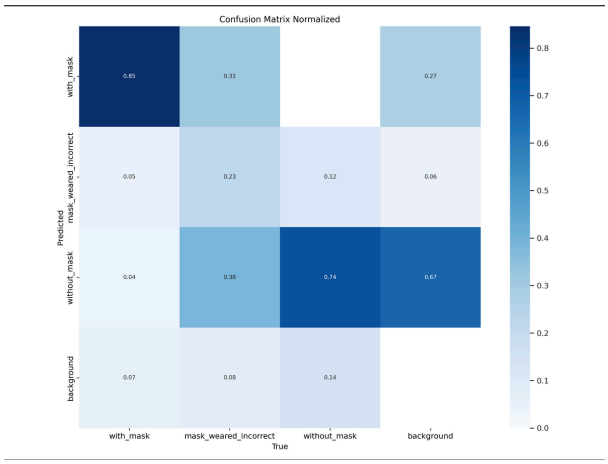
2. Class labels

- Each bounding box is annotated with a label that indicates whether the person is:
  - Wearing a mask (`with_mask`)
  - Not wearing a mask (`without mask`)
  - Wearing a mask incorrectly (`mask_weared_incorrect`).

3. Confidence scores

- The labels include confidence scores to represent the certainty about each detection.

The processed image and related prediction logs were saved in the `runs/detect/predict` directory for further review. Additionally, the `runs` directory provides further insights, such as a confusion matrix, which helps in evaluating the model's performance:



Conclusion

Using YOLO for face mask detection proved to be an efficient and practical approach, particularly due to its ability to automate many aspects of the object detection pipeline. The `ultralytics` library simplifies the process by automatically handling complex tasks such as bounding box generation, class prediction, which are essential components of object detection. The library is also augmented the data, mostly by rotation and blurring.

Face Mask Detector using a customized CNN model for a single-person.

After gaining hands-on experience with YOLO and understanding its capabilities for real-time object detection, we proceeded to tackle different tasks creating our own customized CNN and ResNet models. ResNet is a powerful model known for its ability to train networks by using residual connections to mitigate the vanishing gradient problem.

Preprocessing

Before training the customized CNN model with ResNet, we filtered out multiple persons from images and focus on images containing a single person for classification. We created a `single_files.json` file that stores a structured dataset with three categories: `single_person_files`, `with_mask`, and `without_mask`.

For filtering the images we used YOLOv11 trained on 50 epochs, which was capable of detecting multiple objects in images. We processed the dataset in batches, predicting the images and filtering out single-person images with and without masks. The following code snippet demonstrates the process of creating the JSON file and filtering the images:

```
from dataclasses import dataclass
import json
import os

# Define the Data dataclass
@dataclass
class Data:
    single_person_files: list[str]
    with_mask: list[str]
    without_mask: list[str]

# Define the JSON file path
json_path = 'single_files.json'

# Check if the file exists, and create it if it doesn't
if not os.path.exists(json_path):
    default_data = {
        "single_person_files": [],
        "with_mask": [],
        "without_mask": []
    }
    with open(json_path, 'w') as file:
        json.dump(default_data, file, indent=2)
```

After filtering images into `single_files.json`, we proceeded to implement an iterative refinement process to ensure robustness by re-processing the dataset multiple times, allowing the model to consistently validate and refine the dataset.

```
# Define the number of iterations
num_iterations = 5
```

```
# Perform the loop
for iteration in range(num_iterations):
    print(f"Starting Iteration {iteration + 1}/{num_iterations}")

# Iterate through train, val, and test sets
for set_type in ["train", "val", "test"]:
    images = [f"datasets/{set_type}/images/{i}" for i in os.listdir(f"datasets/{set_type}/images")]
    for chunk in split_list(images, 50):  # Process images in batches of 50
        predict = final_model.predict(chunk)  # Predict using the model
        for p in predict:
            filename = p.path
            if len(p.boxes.cls) == 1:  # Ensure there is exactly one detected class
                if p.boxes.cls[0] == 0:  # Class 0: with mask
                  iteration_data["single_person_files"].append(filename)
                  iteration_data["with_mask"].append(filename)
                elif p.boxes.cls[0] == 2:  # Class 2: without mask
                  iteration_data["single_person_files"].append(filename)
                  iteration_data["without_mask"].append(filename)

print(f"Finished Iteration {iteration + 1}/{num_iterations}")

# Aggregate results into the JSON structure
aggregated_data["single_person_files"].extend(iteration_data["single_person_files"])
aggregated_data["with_mask"].extend(iteration_data["with_mask"])
aggregated_data["without_mask"].extend(iteration_data["without_mask"])
```

```
# Remove duplicates
final_data = {
    key: list(np.unique(values))  # Use np.unique to remove duplicates
    for key, values in aggregated_data.items()
}

# Save the aggregated data back to the JSON file
with open(json_path, "w") as file:
    json.dump(final_data, file, indent=2)
```

Each iteration processed batches of images from `train`, `val`, and `test` sets, and the final aggregated results were cleaned to remove duplicates and save back into the JSON file, provviding realiable and organized dataset for training the RetinaNet CNN.

Training and Inference

After processing the dataset and creating the JSON file for single-person only, we proceeded with training and evaluating with the custom CNN model for mask detection. The customized CNN was specificallly designed for binary classification tasks, distinguishing between `with_mask` and `without_mask` categories. The model's architecture consists of:

- Convolutional Layer
- Pooling Layers
- Fully Connected Layers

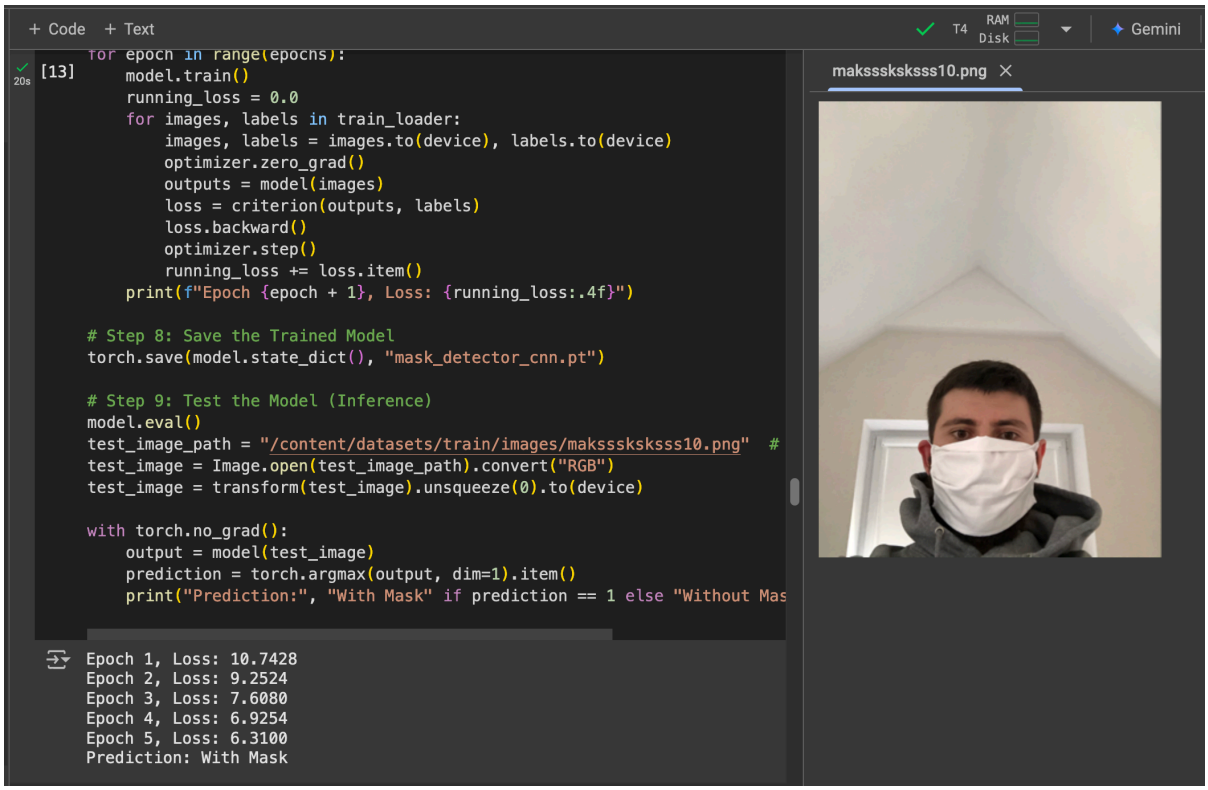
We loaded the dataset and processed it in batches using using PyTorch's `DataLoader`, which allowed for efficient handling of data during training. The model was trained for 10 epochs with a batch size of 16, using the Adam optimizer and CrossEntropyLoss function. The complete implementation of our customized CNN model is displayed below:

```
# Customized CNN model for single person mask detection
import json
import os
from PIL import Image
from torchvision import transforms
from torch.utils.data import Dataset, DataLoader
import torch
import torch.nn as nn
import torch.optim as optim
from tqdm import tqdm

# Load JSON data
json_path = "single_files.json"

with open(json_path, "r") as file:
    data = json.load(file)

# Extract data
image_paths = data["single_person_files"]
with_mask = set(data["with_mask"])
without_mask = set(data["without_mask"])

# Step 2: Define the Dataset Class
class MaskDataset(Dataset):
    def __init__(self, image_paths, with_mask, without_mask, transform=None):
        self.image_paths = image_paths
        self.with_mask = with_mask
        self.without_mask = without_mask
        self.transform = transform

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        img_name = self.image_paths[idx]
        img_path = img_name

        # Load image and put in RGB
        try:
            image = Image.open(img_path).convert("RGB")
        except FileNotFoundError:
            raise FileNotFoundError(f"Image not found: {img_path}")

        # Assign label: 1 for with_mask, 0 for without_mask
        if img_name in self.with_mask:
            label = 1
        elif img_name in self.without_mask:
            label = 0
        else:
            raise ValueError(f"Image {img_name} has no label.")

        # Apply transforms
        if self.transform:
            image = self.transform(image)
```

```
        return image, label

# Step 3: Define Image Transformations
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
])

# Step 4: Create Dataset and DataLoader
dataset = MaskDataset(image_paths, with_mask, without_mask, transform=transform)
train_loader = DataLoader(dataset, batch_size=16, shuffle=True)

# Step 5: Define the CNN Model
class MaskDetectorCNN(nn.Module):
    def __init__(self):
        super(MaskDetectorCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.fc1 = nn.Linear(64 * 30 * 30, 128)
        self.fc2 = nn.Linear(128, 2)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 30 * 30)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Step 6: Initialize Model, Loss, and Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MaskDetectorCNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Step 7: Train the Model
epochs = 10
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for images, labels in tqdm(train_loader):
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch {epoch + 1}, Loss: {running_loss:.4f}")

# Step 8: Save the Trained Model
torch.save(model.state_dict(), "mask_detector_cnn.pt")

# Step 9: Test the Model (Inference)
model.eval()
test_image_path = "/content/datasets/train/images/maksssksksss10.png"  # Example image path
test_image = Image.open(test_image_path).convert("RGB")
test_image = transform(test_image).unsqueeze(0).to(device)

with torch.no_grad():
```

```
    output = model(test_image)
    prediction = torch.argmax(output, dim=1).item()
    print("Prediction:", "With Mask" if prediction == 1 else "Without Mask")
```

After running the cell, we got the following result:



Finally, we evaluated the performance of the trained custom CNN on the test dataset. The model was loaded with saved weights and set to evaluation mode. Each test image was preprocessed and set into the model, where the predictions were compared to the actual labels.

Both correct and incorrect predictions were recorded, calculating the success rate as the percentage of accurate classifications. The complete implementation of the process is displayed below:

```
# Test of custom
model = MaskDetectorCNN().to(device)
model.load_state_dict(torch.load("mask_detector_cnn.pt"))
rate = [0,0]

model.eval()
for i, image in enumerate(images_test):
  test_image_path = image
  test_image = Image.open(test_image_path).convert("RGB")
  test_image = transform(test_image).unsqueeze(0).to(device)

  with torch.no_grad():
      output = model(test_image)
      prediction = torch.argmax(output, dim=1).item()
      if prediction == 1 and labels_test[i] == "with_mask":
        rate[0] += 1
      elif prediction == 0 and labels_test[i] == "without_mask":
        rate[0] += 1
      else:
```

```
        rate[1] += 1

# Calculate the success rate
success_rate = rate[0] / sum(rate) * 100
success_rate
```

This evaluation method is really simple and with more time, could be largely improved by implementing proper evaluation methods, instead of simple accuracy.

Training and testing with ResNet-50

After training and evaluating the customized CNN model for binary classification of face mask detection, we proceeded to gain hands-on experience using ResNet-based architecture to further analyze its performance on the dataset. ResNet-50 is a deep convolutional neural network with 50 layers, known for its ability to effectively train deep models by utilizing skip connections to address the vanishing gradient problems.

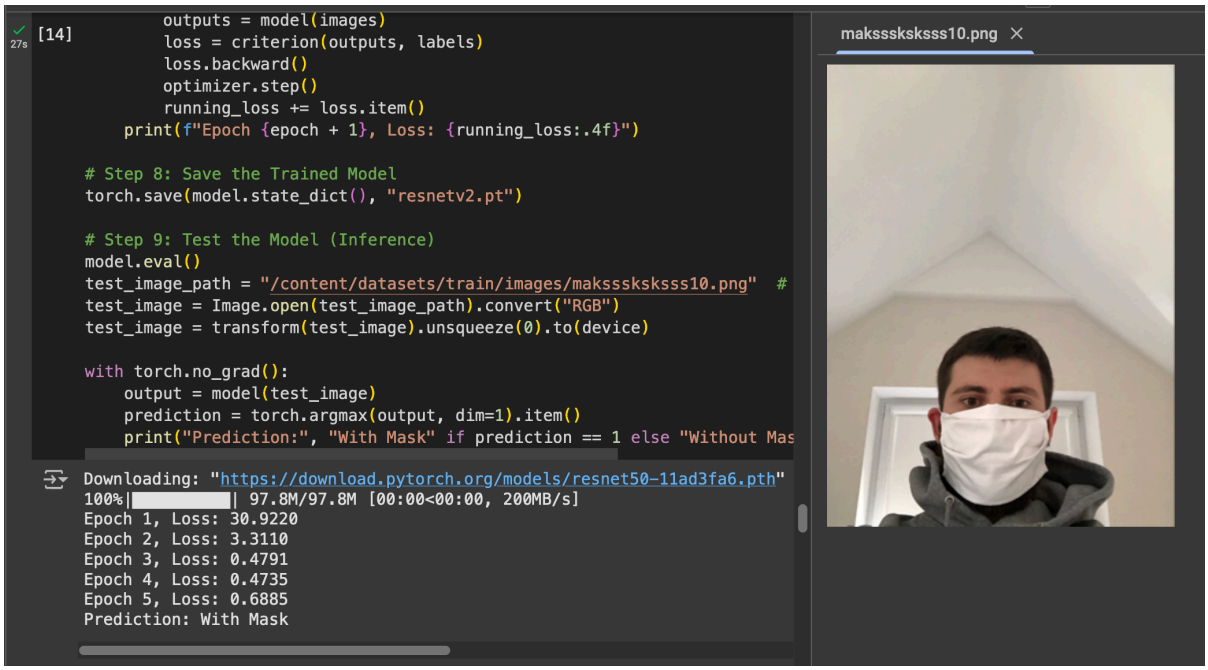
We initialized the ResNet-50 model with pre-trained weights (`ResNet50_Weights.DEFAULT`) to leverage transfer learning. the model was fine-tuned on the single-person dataset, processed in batches using PyTorch's `DataLoader` . We trained the model for 10 epochs using the Adam optimizer and `CrossEntropyLoss` to minimize the classification error, and the weights were saved as `resnetv2.pt` .

The training and testing procedures for ResNet-50 were implemented as follows:

```
from torchvision.models import resnet50, ResNet50_Weights

# Step 6: Initialize Model, Loss, and Optimizer
weights = ResNet50_Weights.DEFAULT
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = resnet50(weights=weights).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Step 7: Train the Model
epochs = 10
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch {epoch + 1}, Loss: {running_loss:.4f}")

# Step 8: Save the Trained Model
torch.save(model.state_dict(), "resnetv2.pt")

# Step 9: Test the Model (Inference)
model.eval()
test_image_path = "/content/datasets/train/images/maksssksksss10.png"  # Example image path
```

```
test_image = Image.open(test_image_path).convert("RGB")
test_image = transform(test_image).unsqueeze(0).to(device)

with torch.no_grad():
    output = model(test_image)
    prediction = torch.argmax(output, dim=1).item()
    print("Prediction:", "With Mask" if prediction == 1 else "Without Mask")
```

After running the cell, we got the following result:



Then, we evaluated its performance on the entire test set by preprocessing each image, running it through the model, and comparing predictions to the ground truth (`with_mask` or `without_mask`). Correct and incorrect classifications were recorded to calculate overall accuracy as a percentage.

The implementation for testing is displayed below:

```
# test of resnet
model = resnet50().to(device)
model.load_state_dict(torch.load("resnetv2.pt"))
rate = [0,0] # correct, incorrect

model.eval() # Set model to evaluation mode
for i, image in enumerate(images_test):
  test_image_path = image
  test_image = Image.open(test_image_path).convert("RGB")
  test_image = transform(test_image).unsqueeze(0).to(device)

  with torch.no_grad(): # Disable gradient tracking
      output = model(test_image)
      prediction = torch.argmax(output, dim=1).item()
      if prediction == 1 and labels_test[i] == "with_mask":
        rate[0] += 1
      elif prediction == 0 and labels_test[i] == "without_mask":
        rate[0] += 1
      else:
```

```
        rate[1] += 1

success_rate = rate[0] / sum(rate) * 100
success_rate
```

Face Mask Detector using RetinaNet

After implementing our customized CNN model, we proceeded to train and test the RetinaNet model. Unlike the customized CNN, which focused on single-person images, RetinaNet was trained using the full dataset, which included all annotated images with multiple bounding boxes for objects.

The task was also changed as we no longer focused on simple classification but on object detection, where the model needed to detect and classify objects in images. The RetinaNet model was trained to detect face masks in images and classify them into three categories: `with_mask` , `mask_weared_incorrect` , and `without_mask` .

Preprocessing the model

Before training the RetinaNet model, we proceeded to prepare the dataset to ensure it was suitable for object detection. This involved loading images, extracting bounding boxes and class labels from annotations, and normalizing the data.

We implemented the `MaskDatasetSegmentation` class to handle the preprocessing. This class loaded the images, read the annotation files, and converted the bounding box coordinates into the format required by RetinaNet. The dataset was then organized into batches using PyTorch's `DataLoader` to streamline training.

The implementation of the preprocessing pipeline is displayed below:

```
from PIL import Image
from torch.utils.data import Dataset
import xml.etree.ElementTree as ET
import torch

class MaskDatasetSegmentation(Dataset):
    def __init__(self, image_paths, transform=None):
        self.image_paths = image_paths
        self.transform = transform

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        img_name = self.image_paths[idx]
        label_name = self.image_paths[idx].split("/")[-1].replace(".png", ".xml")


        try:
            img = Image.open(img_name)
            image = Image.open(img_name).convert("RGB")
        except FileNotFoundError:
            raise FileNotFoundError(f"Image not found: {img_name}")
```

```
        # Apply transforms
        if self.transform:
            img = self.transform(img)
            img_bbox = img.size() # Get image size after transformation
            image = self.transform(image)

        data = "face_mask_dataset/annotations"
        classes = ["with_mask", "mask_weared_incorrect", "without_mask"]

        # Load annotations
        root = ET.parse(f"{data}/{label_name}").getroot()
        # Extract image size
        width = int(root.find("size").find("width").text)
        height = int(root.find("size").find("height").text)

        labels = []
        boxes = []

        for i in root.findall("object"):
            class_id = classes.index(i.find("name").text)
            xml_bbox = [float(coords.text) for coords in i.find("bndbox")]
            # Normalize boundary box for the transformed image
            xml_bbox[0] = xml_bbox[0] / width  * img_bbox[1]
            xml_bbox[2] = xml_bbox[2] / width * img_bbox[1]
            xml_bbox[1] = xml_bbox[1] / height * img_bbox[2]
            xml_bbox[3] = xml_bbox[3] / height * img_bbox[2]
            labels.append(class_id)
            boxes.append(xml_bbox)


        boxes = torch.tensor(boxes)
        labels = torch.tensor(labels)


        return image, {'boxes' : boxes, "labels" : labels}
```

Training the model

The RetinaNet model was initialized using ResNet-50 as its backbone, with pre-trained weights (`RetinaNet_ResNet50_FPN_Weights.DEFAULT`) to enhance performance. The dataset was downloaded from Kaggle, and prepared using the `MaskDatasetSegmentation` class. The model was trained for one epoch using the Adam optimizer. During training, the losses were calculated and backpropagated to update the weights. The running loss was monitored to track convergence, and the final model weights were saved as `retina2.pt` .

The complete implementation of the training process is displayed below:

```
from torchvision.io.image import decode_image
from torchvision.models.detection import retinanet_resnet50_fpn, RetinaNet_ResNet50_FPN_Weights
from torchvision.utils import draw_bounding_boxes
from torchvision.transforms.functional import to_pil_image
from torchvision.ops import sigmoid_focal_loss
import xml.etree.ElementTree as ET
import json
```

```
import tqdm
import os
import pathlib
!pip install kagglehub
!rm -rf face_mask_dataset/
import kagglehub
import torch
import torch.nn as nn
import torch.optim as optim

path = kagglehub.dataset_download("andrewmvd/face-mask-detection")
os.rename(path, "face_mask_dataset")

dataset = MaskDatasetSegmentation(["face_mask_dataset/images/" + i for i in os.listdir("face_mask_dataset/image
train_loader = DataLoader(dataset, batch_size=12, shuffle=True, collate_fn=lambda x: tuple(zip(*x)))

# Step 1: Initialize model with the best available weights
weights = RetinaNet_ResNet50_FPN_Weights.DEFAULT
try:
    model = torch.load("retina_best.pt").to(device)
except:
  model = retinanet_resnet50_fpn(weights=None, box_score_thresh=0.7).to(device) # The threshold is the most imp
optimizer = optim.Adam(model.parameters(), lr=0.001)


# Step 7: Train the Model
epochs = 1
for epoch in range(epochs):
    model.train()
    running_loss = 0.0
    for images, targets in tqdm.tqdm(train_loader):
        # Move data to device
        images = [img.to(device) for img in images]
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        optimizer.zero_grad()
        outputs = model(images, targets)

        losses = sum(loss for loss in outputs.values())
        losses.backward()
        optimizer.step()
        running_loss += losses.item()
    print(f"Epoch {epoch + 1}, Loss: {running_loss:.4f}")

# Step 8: Save the Trained Model
torch.save(model, "retina.pt")
```

The final step tested the RetinaNet model on a sample image, displaying predictions with bounding boxes and class labels directly on the image to verify its accuracy. The full implementation is displayed below:

```
model = torch.load("retina_best.pt").to(device)

classes = ["with_mask", "mask_weared_incorrect", "without_mask"]

model.eval()
test_image_path = "face_mask_dataset/images/maksssksksss19.png"  # Example image path
test_image = Image.open(test_image_path).convert("RGB")
test_image = transform(test_image).unsqueeze(0).to(device)
```

```
with torch.no_grad():
    output = model(test_image)[0]
    i = test_image.squeeze()
    filtered_boxes = [
        (box, label, score)
        for box, label, score in zip(output["boxes"], output["labels"], output["scores"])
        if score > 0.95  # Changing the threshold affects the number of drawn boxes
    ]
    a = [i[0] for i in filtered_boxes]
    only_boxes = torch.stack(a)
    labels = [classes[i[1]] for i in filtered_boxes]
    box = draw_bounding_boxes(
        i,
        boxes=only_boxes,
        labels=labels,
        colors="red",
        width=4, font_size=30)
    im = to_pil_image(box.detach())
im
```

After testing the RetinaNet model using sample images from the dataset, we got the following results:

The results demonstrate that the RetinaNet model successfully detected face masks with bounding boxes. However, multiple bounding boxes appeared on some individuals wearing a face mask, which cause noise in the results. This noise can negatively impact performance metrics, such as precision and recall, by inflating the number of detected objects. To address this, applying Non-Maximum Suppression (NMS) can help remove redundant boxes by retaining only the most confident bounding box for each detected object.

Additionally as the model was trained on COCO dataset [4], it may not be optimized for detecting face masks. Training the model only on face mask images would reduce the amount of noise carried over from the trained weights from COCO dataset.

The model itself enables setting our own box scoring threshold, which can be adjusted to improve the model's performance. By setting a higher threshold, we can filter out less confident predictions, reducing the number of false

positives and improving the model's precision. However, during training we discovered that setting the score too high results in no boxes, making the model not learn, and when setting the score too low, making the focal loss activation explode, leading in catastrophic increase in loss.

Conclusion

The YOLOv11 model made the early stages of the project easier due to its pre-trained capabilities and streamlined implementation using the Ultralytics library. It enabled fast training, testing, and deployment, making it ideal for real-time detection scenarios. However, as we proceeded to develop customized CNNs and train RetinaNet, the process became significantly more challenging. These models required a deeper understanding of CNN concepts to successfully manage and build an effective face mask detection system.

Despite these challenges, the project provided valuable hands-on experience using machine learning for object detection. It also highlighted the importance of understanding the concepts of CNN when creating customized solutions. Improvements, such as fine-tuning the models and optimizing preprocessing pipelines can further enhance performance and noise, ensuring more accurate and reliabe face mask detection results.

References

1. YOLOv11: An Overview of the Key Architectural Enhancements, Rahima Khanam and Muhammad Hussain, 2024, https://arxiv.org/abs/2410.17725v1
2. CSPNet: A New Backbone That Can Enhance Learning Capability of CNN, Chao Wang, Hongqiao Wang, and Shilin Zhang, 2020, https://openaccess.thecvf.com/content_CVPRW_2020/papers/w28/Wang_CSPNet_A_New_Backbone_That_Can_Enhance_Learning_Capability_of_CVPRW_2020_paper.pdf
3. Focal Loss for Dense Object Detection, Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár, 2017, https://arxiv.org/abs/1708.02002
4. COCO Dataset, https://cocodataset.org/#home