Real-Time Face Recognition System with Custom-Trained Deep Learning Models

Team Members:

- **[Ahmed Kamal FathAllah]**, **ID:** [17010210]
- **[George Seleim Abd-Allah Khaleel]**, **ID:** [20010436]

Department:

- **Computer Science and Engineering Department (CSED)**, **Alexandria University**

supervisors:

- **[Dr. Salah Selim]**, Email: **[salahss9@yahoo.com]**

Co-Advisor:

- **[Dr. Magdy Abdel-Azeem]**, Email: **[magdy_aa@hotmail.com]**

Project Details:

- **Graduation Project**

This report provides a comprehensive overview of the facial recognition system. The system encompasses various components, including face detection, face recognition, and a user interface, all integrated into a cohesive application. The report details the models used, the project structure, installation procedures, usage examples, and underlying code implementations.

System Overview

The facial recognition system is built using Python and integrates several libraries for deep learning, computer vision, and GUI development. It leverages:

- **CustomTkinter:** For creating a graphical user interface.
- **PyTorch:** As the deep learning framework for model training and inference.
- **OpenCV:** For computer vision tasks such as image capture and processing.
- **SQLite:** As the database for storing user information.
- **ESP32-CAM:** For capturing images from a camera. The system features both a custom-trained face detection model and an alternative implementation using InsightFace. The primary face recognition model is based on a ResNet architecture with modifications for enhanced performance.

Models

Face Detection

- The system employs a **custom-trained face detection model** built upon a Convolutional Neural Network (CNN) architecture.
- The model was trained using the **CelebA dataset**, consisting of over 200,000 images with bounding box annotations.
- The CNN architecture contains **multiple convolutional and fully connected layers**.
- The model takes images of size **224x224 pixels** as input.
- The model outputs **four coordinates** representing the bounding box: x, y, width, and height.
- **An alternative implementation utilizes the YuNet face detector,** a model specifically chosen for its efficiency and accuracy in face detection, especially in scenarios with varying image quality, which makes it suitable for processing images from the ESP32-CAM. The **YuNet model is integrated through OpenCV's `cv2.FaceDetectorYN` class**, allowing for streamlined implementation within the project. This model was preferred over the custom-trained bounding box model (v5) because the latter was trained on high quality images of celebrity faces from the CelebA dataset and was found to be less robust when used with the low-quality images generated by the ESP32-CAM.
- **Key Features of YuNet:** The YuNet face detector is configured with specific parameters to optimize its performance. These parameters include an **input size of 320x320 pixels**, which is the size to which input frames are resized before being processed by the detector, a **confidence threshold of 0.9** which determines

the minimum confidence score for a detection to be considered valid, a **non-maximum suppression (NMS) threshold of 0.3** which filters out overlapping detections and a **top-k value of 5000**, which specifies the maximum number of detections to retain after NMS.

Face Recognition

- The core face recognition model is based on a **ResNet architecture** with residual connections, referred to as **ResArkSGD**.
- Several versions of this model exist, each with varying accuracy:
    - **88% accuracy:** A version not augmented.
    - **95% accuracy:** A version not augmented.
    - **94% accuracy:** An augmented version.
- The model is trained using **VGGFace2_hq_cropped** and tested on **Pins** datasets.
- The model outputs embeddings, which are vector representations of facial features used for comparison during recognition.
- The model uses **ArcFace Loss** for training, which enhances feature discrimination by applying an angular margin penalty.

Model Architectures

- **FaceBBoxModel (Bounding Box Detector):**
- The `FaceBBoxModel` is a custom-designed Convolutional Neural Network (CNN) that is responsible for detecting faces within an image and predicting their bounding box coordinates. The model's architecture is structured as a series of **convolutional layers for feature extraction**, followed by fully connected layers for bounding box regression.
    - **Convolutional Layers**: The model begins with **five convolutional layers**, each playing a critical role in feature extraction. These layers are organized as follows:
        - **Conv1**: The first convolutional layer applies **16 filters** with a **kernel size of 3x3**, a stride of 1, and padding of 1 to the input image, which has 3 color channels (RGB). This layer extracts the initial low-level features from the image.
        - **Conv2**: The second convolutional layer uses **32 filters**, also with a **3x3 kernel**, stride of 1, and padding of 1. It builds on the features learned in the previous layer to extract more complex patterns.

        - **Conv3**: The third convolutional layer employs **64 filters** with the same kernel size, stride, and padding parameters. It continues to refine the feature maps, capturing increasingly intricate details.
        - **Conv4**: The fourth convolutional layer utilizes **128 filters** with the same kernel size, stride, and padding. This layer extracts more complex features that are essential for accurate bounding box prediction.
        - **Conv5**: The fifth and final convolutional layer uses **256 filters** with a **3x3 kernel**, stride of 1, and padding of 1. This layer extracts the deepest features used by the model.
    - **Batch Normalization:** Each convolutional layer is immediately followed by a **batch normalization layer** (`nn.BatchNorm2d`). Batch normalization is used to normalize the output of the previous layer by adjusting and scaling the activations, resulting in more stable and faster training. It does this by calculating the mean and variance of each channel in the mini-batch and then using those values to normalize the layer's output.
    - **Max Pooling:** After each convolutional and batch normalization layer, a **max pooling layer** (`nn.MaxPool2d`) with a **kernel size of 2x2** and a stride of 2 is applied. **Max pooling reduces the spatial dimensions of the feature maps**, which helps to decrease the number of parameters, control overfitting, and make the model more robust to small variations in the input images. It does this by taking the max value of each 2x2 region.
    - **Fully Connected Layers:** Following the convolutional layers, the feature maps are **flattened** into a vector and passed through **three fully connected (linear) layers** (`nn.Linear`) for bounding box prediction.
        - The first fully connected layer transforms the flattened feature map to a **512-dimensional vector**, followed by a ReLU activation function.
        - A **dropout layer** (`nn.Dropout`) with a dropout rate of **0.3** is applied to prevent overfitting. Dropout works by randomly setting a fraction of the neurons to zero during training, forcing the network to learn more robust features.
        - The second fully connected layer transforms the 512-dimensional vector to a **128-dimensional vector**, also followed by a ReLU activation.
        - The third fully connected layer reduces the vector to **4 dimensions**, corresponding to the **bounding box coordinates (x, y, width, height)**.

- **Output**: The model outputs the bounding box coordinates, which are then used to crop the face from the original image, for further processing by the face recognition model.

- **EnhancedFaceRecognitionModel (Face Recognition):**
- The `EnhancedFaceRecognitionModel` is a sophisticated deep learning model designed for **generating high-quality embeddings** for face recognition. This model employs a deep architecture comprising initial convolutional layers, followed by multiple **residual blocks and deep residual blocks**, and finally fully connected layers to produce the face embeddings.
  - **Initial Convolutional Layers**:
    - The model starts with a sequence of initial convolutional layers (`self.conv1`). This sequence includes a **2D convolutional layer (`nn.Conv2d`) with 64 filters**, a **kernel size of 7x7**, a stride of 2, and padding of 3. This layer is responsible for extracting initial, low-level features from the input image, which has three color channels (RGB).
    - The output of the convolutional layer is then passed through a **batch normalization layer (`nn.BatchNorm2d`) with 64 channels**, which normalizes the output and helps in faster training.
    - A **ReLU (Rectified Linear Unit) activation function** is applied to introduce non-linearity.
    - Finally, a **max pooling layer (`nn.MaxPool2d`) with a kernel size of 3x3**, a stride of 2, and padding of 1 is used to downsample the feature maps, reducing their spatial dimensions and computational load while retaining the most important information.
  - **Deep Residual Layers**: The core of the model consists of four distinct sets of residual layers:
    - **Layer 1**: This layer (`self.layer1`) consists of **3 standard residual blocks**, each with 64 input and output channels, and a stride of 1. These blocks learn feature mappings and add residual connections to avoid the vanishing gradient problem.
    - **Layer 2**: This layer (`self.layer2`) is composed of **4 standard residual blocks**, each with 64 input channels and 128 output channels, and a stride of 2 for the first block.
    - **Layer 3**: This layer (`self.layer3`) contains **6 standard residual blocks**, each with 128 input channels and 256 output channels, and a stride of 2 for the first block.

        - **Layer 4**: This layer (`self.layer4`) is made up of **3 deep residual blocks**, each with 256 input channels and 512 output channels, and a stride of 2 for the first block. This deep residual block is different from standard residual blocks because it adds an extra convolutional layer in the block, which gives the model more capacity to learn complex patterns in the data.
    - **Residual Blocks**:
        - The **standard residual block** (`ResidualBlock`) contains two convolutional layers (`nn.Conv2d`), each with a kernel size of 3, padding of 1, and batch normalization (`nn.BatchNorm2d`) after each convolution and ReLU activation after the first convolution. A shortcut connection is added to the output of the second convolutional layer. The shortcut connection either directly passes the input or uses a 1x1 convolution with batch norm if the input and output feature maps have different dimensions.
    - The **deep residual block** (`DeepResidualBlock`) is similar to the standard residual block but has an additional convolutional layer (`nn.Conv2d`) and batch normalization layer, which allows the model to learn more complex features.
    - **Fully Connected Layers**: After the deep residual layers, the feature maps are flattened and passed through two fully connected layers (`self.fc_layers`).
        - The first fully connected layer (`nn.Linear`) transforms the flattened feature map to a **1024-dimensional vector** and is followed by batch normalization (`nn.BatchNorm1d`) and a ReLU activation.
        - The second fully connected layer (`nn.Linear`) maps the 1024-dimensional vector to a **512-dimensional embedding vector**, also followed by batch normalization.
    - **Output**: The model outputs a **512-dimensional normalized embedding vector** that represents the unique facial features of the input. This embedding can then be used to determine the similarity between different faces.
    - **ArcFace Loss**: The model is trained using an **ArcFace loss function** (`ArcFaceLoss`), which is specifically designed to improve the discriminative power of face embeddings. The ArcFace loss applies an angular margin penalty to the target logit, enhancing feature discrimination. During training, the ArcFace loss uses a scale and margin

for the first 40 epochs (s=30, m=0.5), and then increases them for the remaining epochs (s=45, m=0.7).
    - **Weight Initialization**: The model's weights are initialized using Kaiming Normal initialization for convolutional layers, Xavier Normal initialization for linear layers and batchnorm layers are initialized to 1 for weights and 0 for biases.

Project Structure

The project is organized into several Python files and directories:

- **`app.py`:** The main application file that uses custom models for facial recognition. It includes the GUI and the main logic of the application. It is now using YuNet face detector model as it is more robust on esp32-cam
- **`insightface_app.py`:** An alternative implementation using the InsightFace library for facial recognition.
- **`embeddings.py`:** Contains the implementation and loader class of the face recognition model.
- **`bounding_box.py`:** Contains the implementation and loader class of the custom face detection model.
- **`bounding_box_yunet.py`:** Contains the implementation and loader class of the YuNet face detection model.
- **`models/`:** A directory containing all the trained models.
    - **`bbox_models/`:** Contains face detection models.
    - **`v5/`:** Contains custom-trained models.
    - **`YuNet/`:** Contains models for the YuNet face detector.
    - **`resarksgd/`:** Contains face recognition models.

Installation

The installation process involves the following steps:

1. **Cloning the Repository:** The project is cloned from a GitHub repository using the following command: `git clone https://github.com/AhmedKamal75/Graduation_Project.git`

2. **Navigating to the Project Directory:** The user navigates into the cloned project directory: `cd Graduation_Project`

    a. **Creating and Activating a Virtual Environment:** A virtual environment is created and activated to manage the project's dependencies:

```
python3 -m venv .venv
source .venv/bin/activate # Linux/macOS
.venv\Scripts\activate # Windows
```

3. **Installing Dependencies:** The required Python packages are installed using pip:
```
pip install -r requirements.txt
```

4. **Downloading the Pre-trained Model:** The `resarksgd95.pth` model file is downloaded from the provided Kaggle link and placed in the `models/resarksgd/` directory. model with accuracy of 95%

5. **Updating the Embedding Predictor:** The application code (`app.py`) is modified to point to the correct path of the downloaded model file. For example:
```
embedding_predictor =
EmbeddingPredictor(model_path='models/resarksgd/resarksgd95.pth',
device='cpu')
```

Usage

The application supports two main processes: registration and authentication.

Registration Process

1. The main application is run using the command: `python app.py`

2. The user clicks the "Register Person" button in the GUI.
3. The user enters their name and sets access permissions.
4. The system automatically captures 10 facial samples of the user.
5. A verification message indicates whether the registration was successful.

Authentication Process

1. The user stands in front of the camera.
2. The user clicks the "Login Person" button in the GUI.
3. The system displays the recognition results, indicating whether the user is successfully authenticated. The threshold for recognition can be adjusted to control the sensitivity of the recognition system. The note in the README.md indicates the threshold can be as low as 0.1.

Code Implementation

GUI Elements (`app.py` and `insightface_app.py`)

The GUI is constructed using CustomTkinter, featuring frames for login, results, and control panels.

- **Login Frame:** A frame containing a label "Login" and a button "Login Person".
- **Results Frame:** A frame for displaying the recognition results. This frame includes a label "Results" and a text box for output.
- **Main Loop and Program Exit:** The main application runs in a loop using `self.root.mainloop()`. The program exits when the window is closed by calling `self.exit_program()`.
- **Capture Thread:** If active, the capture thread is joined, and the application terminates when the window is closed.
- **Camera and Database Paths:** The camera URL and database path are initialized in the main section of the application.
- **Embedding Predictor Initialization:** An embedding predictor is initialized with the specified model and device (`cpu`).
- **The `run` Function**: The `run` method sets up the application's main event loop, ensures proper handling of window closing events, and starts the GUI.
- **`login_btn` Widget:** The login button is created with the `CTkButton` class, and the button is set to execute the `self.login_person` command when clicked.

Face Recognition Model (`arkface_residual_connections_part_2.ipynb`)

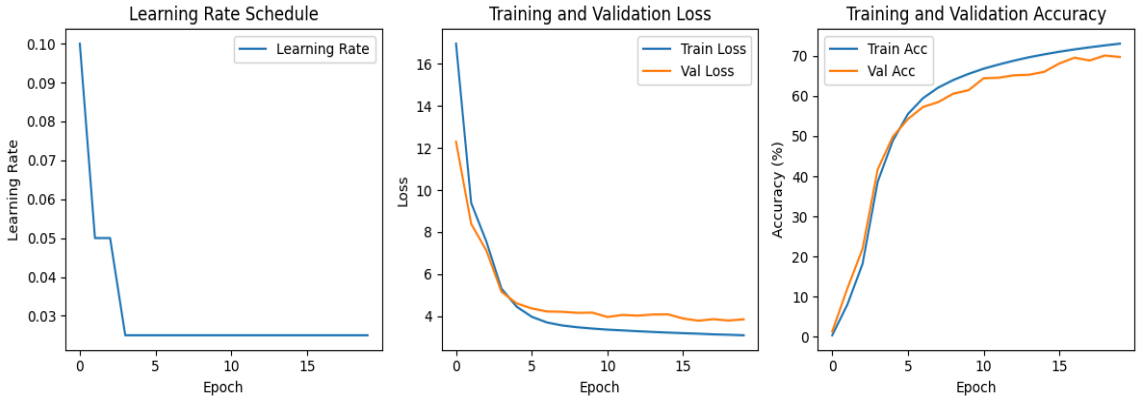
The face recognition model is implemented with PyTorch, incorporating several advanced techniques and layers:

- **Data Loading and Preprocessing:** Dataset such as VGGFace2 is used for training the model, Pins is used for testing the model. The datasets are preprocessed using transformations that include resizing, normalization, and converting to tensors. Albumentations is used for applying image augmentations.
- **Dataset Classes:** Custom dataset classes (`PinsDataset` and `VGGFace2Dataset`) are implemented to manage image loading and labeling. These classes handle the retrieval of images and their corresponding labels and also allow for the creation of subsets.
- **ArcFace Loss:** The `ArcFaceLoss` class implements the ArcFace loss function, which is designed to enhance feature discrimination during training. It does this by adding an angular margin penalty to the target logit.
- **Residual Blocks:** The model uses residual blocks (`ResidualBlock` and `DeepResidualBlock`) to build deep convolutional layers. These blocks help with training deep neural networks.
- **Model Architecture:** The `EnhancedFaceRecognitionModel` class defines the main structure of the face recognition network. It uses a combination of convolutional layers, residual blocks, and fully connected layers. The model includes initial convolutional layers, deep residual layers, simplified fully connected layers, and the ArcFace loss.
- **Model Training Process for `EnhancedFaceRecognitionModel`:**
- The training of the `EnhancedFaceRecognitionModel` is orchestrated within the `train_model` function. This process encompasses data handling, optimization, and performance evaluation, and includes detailed steps as below:
- **Data Loading**:
    - The training process starts by loading the training and validation datasets using PyTorch's `DataLoader` class.
    - The datasets are split into batches of a defined `batch_size`. The batch size is set to 64 in the provided configuration.
    - The data loaders also handle shuffling of the data to ensure that the model is exposed to a diverse set of samples during each epoch.
    - The data loaders also use `num_workers` parameter to load the data in parallel. `pin_memory=True` is used when using a CUDA device to load the data into the GPU memory more efficiently. The number of workers is 0 when using CPU and 4 when using a CUDA device.
- **Optimizer and Loss Function Setup:**
    - The model employs the **Stochastic Gradient Descent (SGD) optimizer** with **Nesterov momentum**. This optimizer is selected for its effectiveness in training deep neural networks.

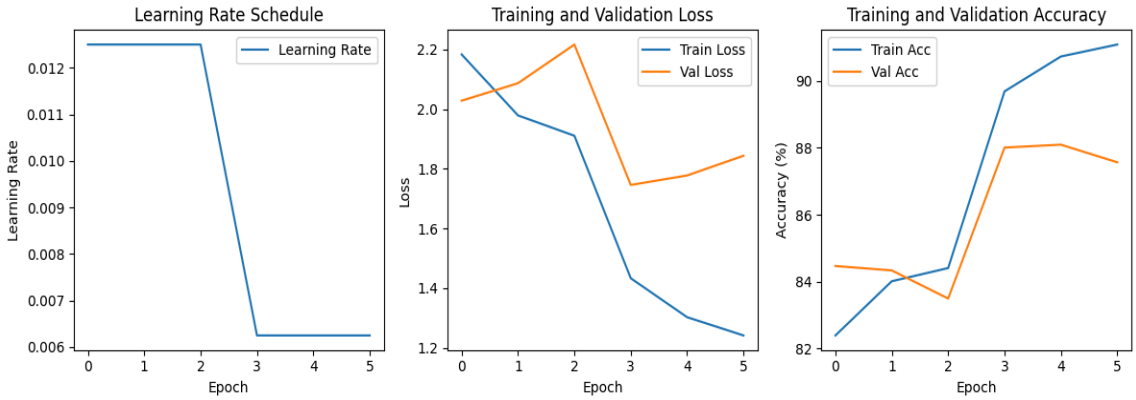
    - The optimizer parameters include a learning rate (`learning_rate`), momentum (`momentum`), and weight decay (`weight_decay`).
    - The learning rate is initially set to 0.1 in the provided configuration.
    - The momentum is set to 0.9 and weight decay is set to 1e-4.
    - A **MultiStepLR scheduler** is used to decay the learning rate during training. This scheduler decreases the learning rate by a factor (`gamma`) at predefined epochs (`milestones`). The learning rate decreases by a factor of 0.5 at epochs.
    - A **CrossEntropyLoss** is used as the loss function for training the model. This is a standard loss function for multi-class classification problems.
- **Training Loop:**
    - The training process is carried out for a specified number of epochs.
    - In each epoch, the model iterates through the training dataset in batches.
    - During the training phase, the model is set to training mode (`model.train()`), and for each batch, the following steps are executed:
        - The input images and labels are transferred to the appropriate device (CPU or GPU).
        - The model performs a forward pass, producing the outputs and embeddings.
        - The loss is calculated using the `CrossEntropyLoss`.
        - The gradients are calculated and backpropagated.
        - The model parameters are updated using the optimizer with clipped gradients to prevent exploding gradients.
        - The training loss and accuracy for the batch are recorded.
    - After each epoch, the average training loss and accuracy are calculated and recorded.
- **Validation Phase:**
    - After the training phase in each epoch, the model's performance is evaluated on a validation dataset.
    - The model is set to evaluation mode (`model.eval()`), which turns off operations like dropout and batch normalization that are only used during training.
    - The model iterates through the validation dataset, and for each batch, it performs a forward pass, calculates the loss, and records the validation loss and accuracy.
    - The gradients are not computed during the validation phase to save memory and time.

    - After each epoch, the average validation loss and accuracy are calculated and recorded.
- **Learning Rate Scheduling:**
    - The learning rate scheduler is stepped after each epoch, adjusting the learning rate according to the specified milestones.
    - The current learning rate is printed at the beginning of each epoch.
- **Model Saving:**
    - The model with the best validation loss is saved during training.

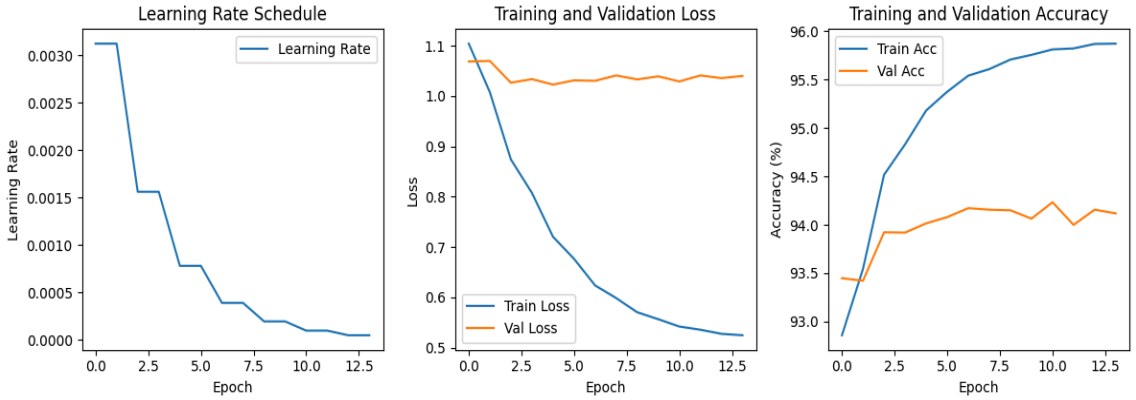
- **Performance Monitoring:**
    - The training and validation losses and accuracies are tracked across epochs using python lists and plotted at the end of training to visualize the training process.
    - The learning rate schedule is also tracked and plotted.
- These plots help assess the convergence and identify potential problems like overfitting.
- **Training Results**
    - 5 sets of training results are done, and the result is plotted:
        - The first result after 20 epochs of training:



        - The second result after 6 epochs of training:

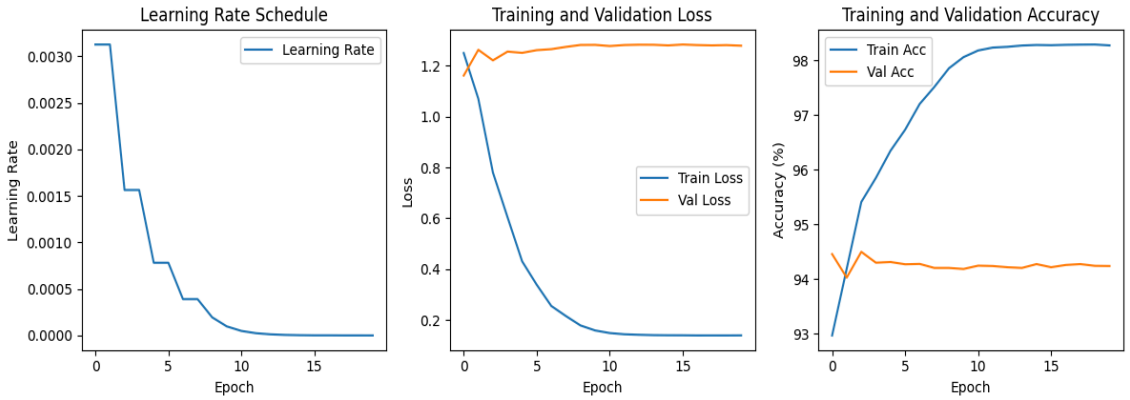


- The third result after 14 epochs of training:



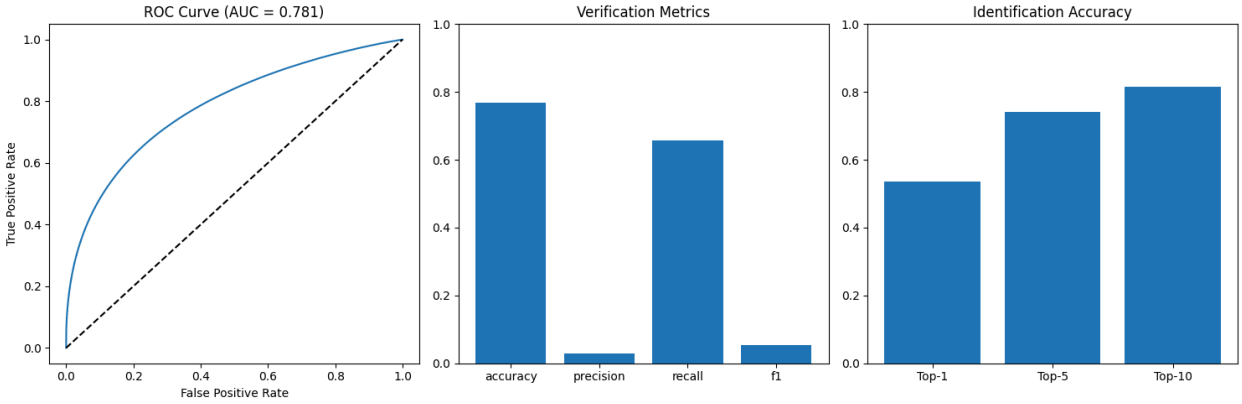
- The fourth result after 3 epochs of training:

- The fifth result after 20 epochs of training with heavy augmentations:



- These results show a trend that as the training goes on the training and validation loss decreases and the training and validation accuracy increases.

- **Testing Results:**
  - The testing results are obtained from running the trained model of 95% accuracy on the entire PINS dataset. The dataset has 105 people and 17534 images. These tests include:
  - **Verification Results:**



- The verification results include **accuracy: 0.7825, precision: 0.0292, recall: 0.6485, f1: 0.0558, auc: 0.7837, best_threshold: 0.1122**
- These results are calculated by comparing embeddings from images of the same person with embeddings from images of different people.
- The **accuracy** metric provides a general measure of the verification performance, showing that the model can classify the images correctly 78% of the time, but when looking at the other metric there seems to be a big imbalance between **precision and recall**.
- The **precision** metric is at 0.0292 indicating that out of all the pairs classified as being of the same person, only 2.92% are the same person. This is very low and indicates many false positives.
- The **recall** metric is at 0.6485, indicating that the model can identify about 65% of the images that are of the same person.
- The **F1 score** is only 0.0558, which reflects the imbalance between precision and recall.
- The **AUC** which is the area under the ROC curve, is 0.7837 which is better than random performance.
- The best threshold was found to be 0.1122 which is the optimal threshold when using the ROC curve.

o **Identification Results:**

- **Top 1 Accuracy: 0.5450.**
- **Top 5 Accuracy: 0.7459.**
- **Top 10 Accuracy: 0.8223**

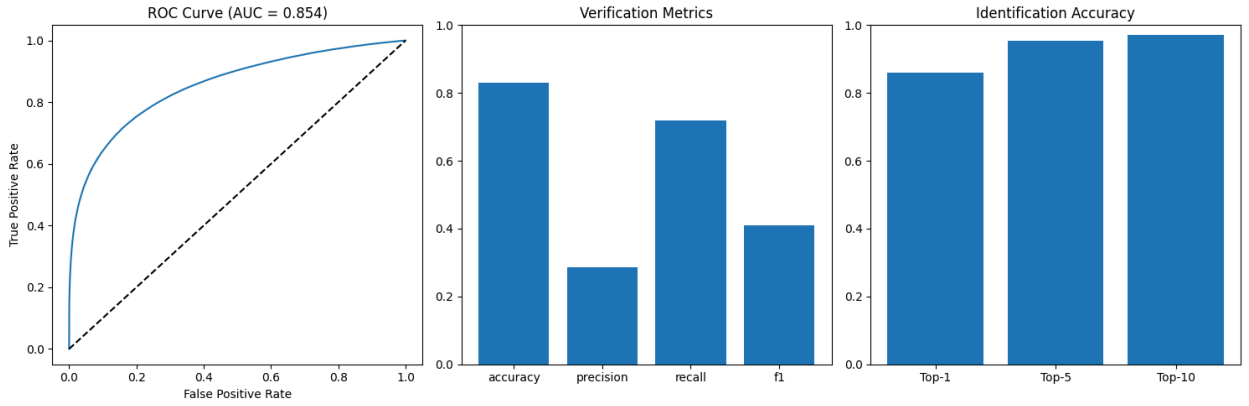
o **Optimal Threshold Results:**

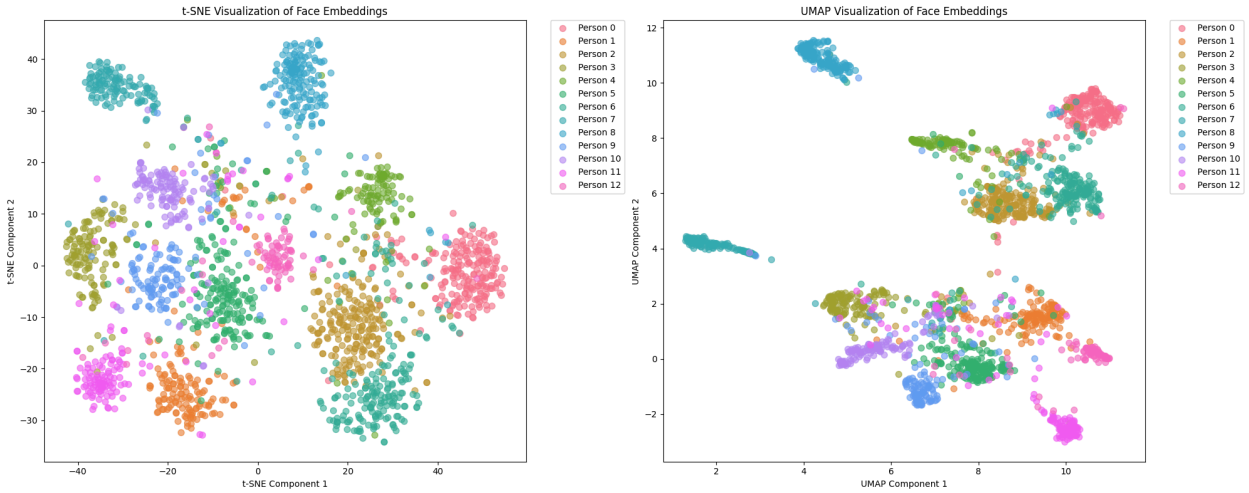
- The model finds the best threshold that maximizes precision while maintaining an acceptable recall (at least 0.2 in this case). The

optimal threshold was found to be 0.31, at which the precision is 0.153, recall is 0.204 and f1 is 0.175.

- **Unconditional Optimal Threshold Results:**
  - The model also finds the optimal threshold that maximizes precision unconditionally. The unconditional optimal threshold was found to be 0.99, at which the precision is 1.0, recall is 0.006 and f1 is 0.01.

- Another test was performed on a subset of the PINS dataset consisting of **2000 images** from **13 different identities**.





- **Verification Results:**
  - **Accuracy: 0.8292**
  - **Precision: 0.2865**
  - **Recall: 0.7179**
  - **F1 Score: 0.4095**
  - **AUC: 0.8540**
  - **Best Threshold: 0.1263**

- **Identification Results:**
  measure how often the **correct identity** appears within the **top X most likely predictions** made by the model
  - **Top-1 Accuracy: 0.8610**
  - **Top-5 Accuracy: 0.9535**
  - **Top-10 Accuracy: 0.9720**
- **Optimal Threshold Results (for max precision with recall >= 0.2):**
  - **Optimal Threshold: 0.4**
  - **Precision: 0.9121**
  - **Recall: 0.2111**
  - **F1 Score: 0.3428**
- **Unconditional Optimal Threshold Results (for max precision):**
  - **Unconditional Optimal Threshold: 0.99**
  - **Unconditional Precision: 1.0000**
  - **Unconditional Recall: 0.0061**
  - **Unconditional F1 Score: 0.0121**

Comparison with Full PINS Dataset Results

- **Verification:**
  - The subset shows a significantly higher accuracy (0.8292 vs 0.7825), precision (0.2865 vs 0.0292) and F1 score (0.4095 vs 0.0558), and AUC (0.8540 vs 0.7837) compared to the full dataset test. The subset and full dataset recall values are also different at (0.7179 vs 0.6485).
  - **This indicates that the model performs much better on the subset for face verification, with fewer false positives and better overall classification performance.**
- **Identification:**
  - The subset shows a significantly higher accuracy across the board for identification results; (top 1: 0.8610 vs 0.5450, top 5: 0.9535 vs 0.7459, top 10: 0.9720 vs 0.8223).
  - **This shows that the model is much more accurate at identifying faces in the top-k positions in the subset compared to the full dataset**.
- **Optimal Threshold:**
  - The subset produces a much higher precision with a good F1 score when finding the optimal threshold on the condition that recall is above 0.2 (precision: 0.9121 vs 0.153, F1: 0.3428 vs 0.175). However, the recall is much lower (0.2111 vs 0.204) with the subset test, likely as a consequence of the optimization parameters.

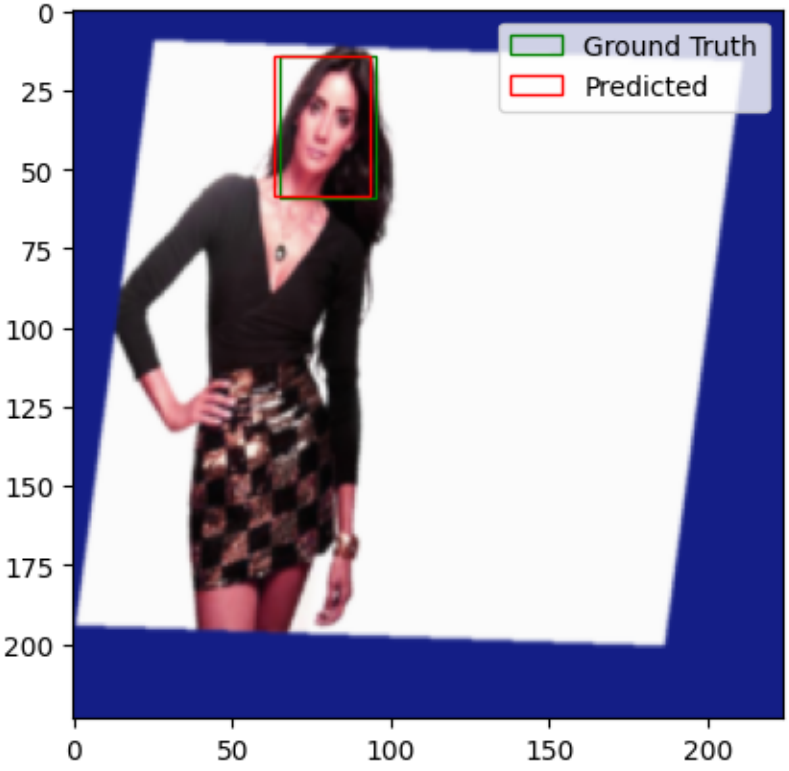
Opinion on the Differences

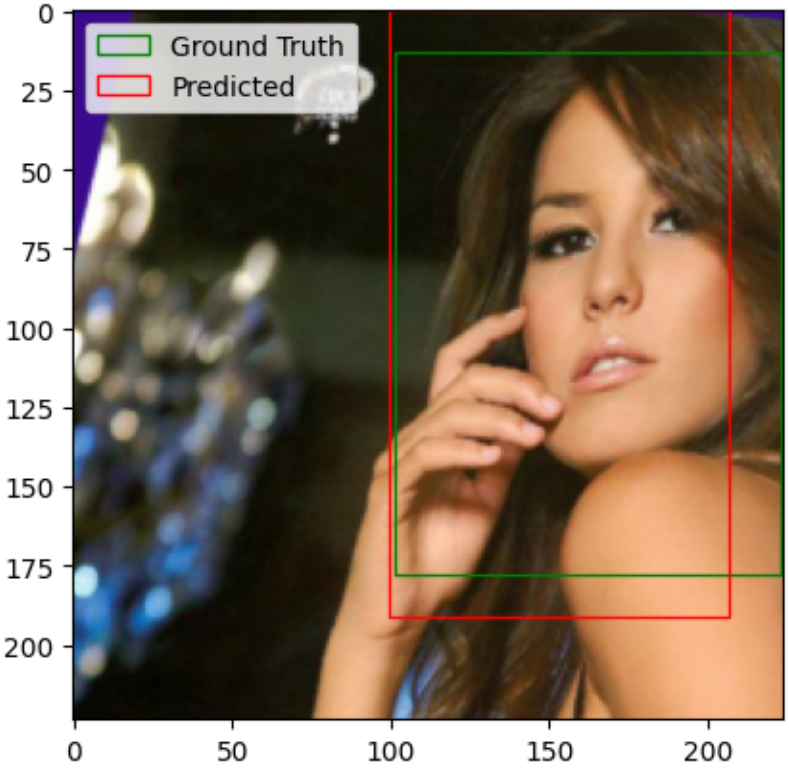
- **Dataset Complexity:** The most significant factor contributing to the difference is the complexity of the dataset.
    - The full PINS dataset contains 105 different identities and over 17,000 images, while the subset has only 13 identities and 2000 images.
    - **The reduced number of classes (identities) and samples in the subset makes the face recognition and verification tasks much easier for the model to handle compared to the full dataset.**
- The model performs significantly better on the subset of the PINS dataset than on the full dataset, as indicated by the verification and identification metrics.
- The difference is likely due to the reduced complexity of the subset.
- It's crucial to evaluate models on diverse and representative datasets to get an accurate assessment of their real-world performance.

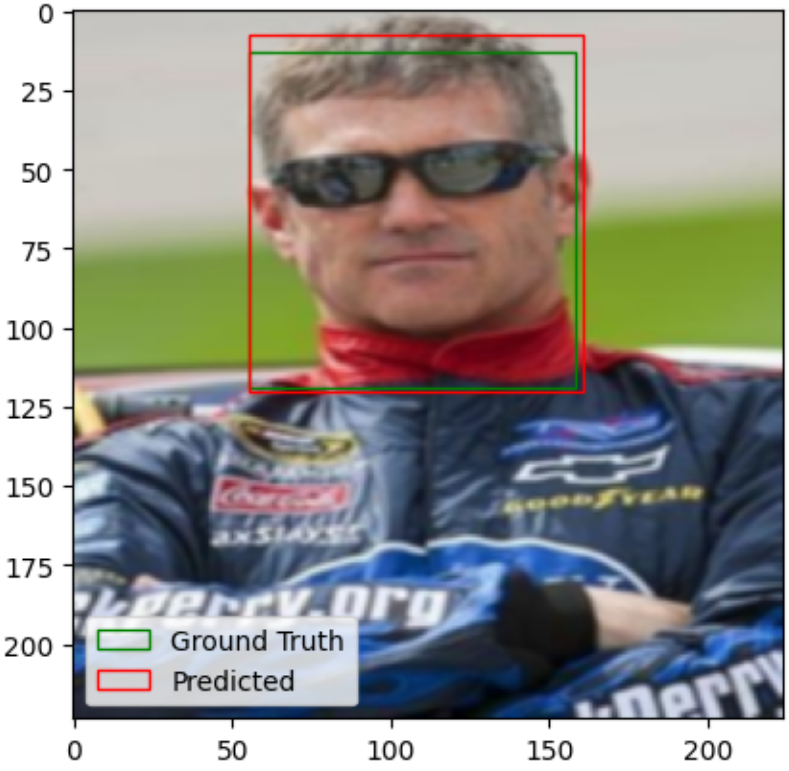
Face Detection Model (`bounding_box.py` and `facial_detection_celeba_bbox_optimized.py`)

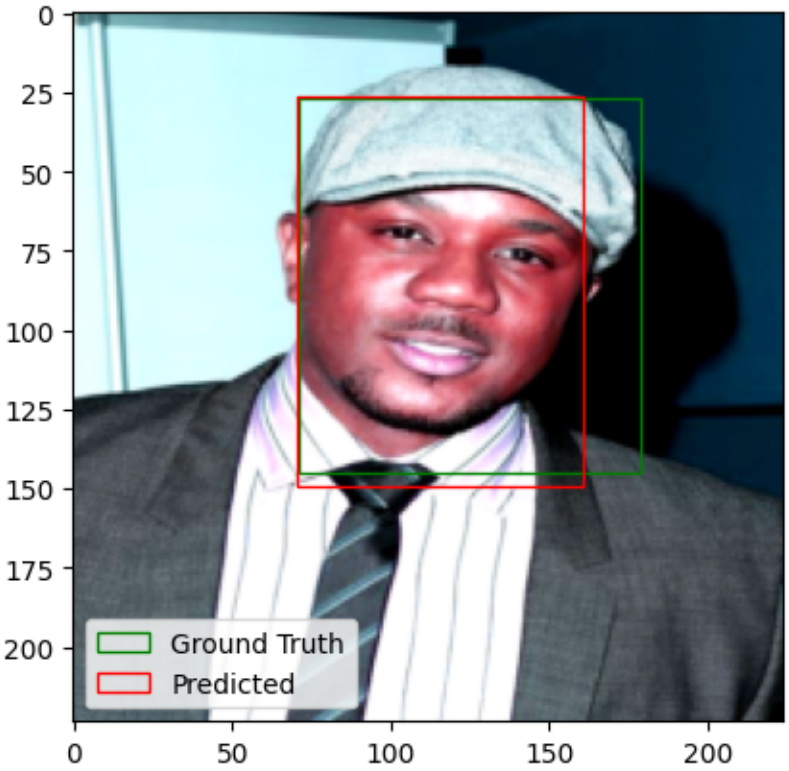
The face detection model implementation involves:

- **Model Architecture:** The `FaceBBoxModel` class defines the structure of the face detection network. It utilizes convolutional layers, batch normalization, max pooling, and fully connected layers for bounding box prediction.
- **Data Loading:** The CelebA dataset is loaded from Hugging Face for training the bounding box model. This dataset contains images of celebrities with bounding box annotations.
- **Dataset Class:** A custom dataset class is created to manage the images and their bounding box information and applies transformations to the images and bounding boxes such as resizing and normalization.
- **Training Process**: The training process involves iterating through the dataset and updating the model parameters using an optimizer and the loss function. The training function includes gradient scaling for faster training and utilizes the Adam optimizer with a learning rate scheduler.
- **Data Visualization:** A function is implemented to display images with their bounding box annotations. This visualization is useful for confirming the dataset and the model's outputs.
- These are some of the outputs of the model on the testing set of the celebA:


Ground Truth
Predicted
0
25
50
75
100
125
150
175
200
0
50
100
150
200


Ground Truth
Predicted
0
25
50
75
100
125
150
175
200
0
50
100
150
200


0
25
50
75
100
125
150
175
200
0
50
100
150
200
Ground Truth
Predicted


0
25
50
75
100
125
150
175
200
0
50
100
150
200
Ground Truth
Predicted

Key Components and Functions:

- **`app.py`**: This file serves as the central hub for the facial recognition application, integrating the GUI, camera interface, and deep learning models for face detection and recognition. It controls the overall flow of the application. It manages the user interface, processes user actions, captures camera data, performs face detection, extracts embeddings, and manages registration and authentication.
- **`insightface_app.py`**: This alternatively implements the core functionalities by utilizing the `InsightFace` library. This file offers a way to implement facial recognition tasks using different tools and models from the main application.
- **`embeddings.py`**: This file contains the implementation of the face recognition model based on the `EnhancedFaceRecognitionModel` class. This includes the forward pass through the network to extract embeddings, the ArcFace loss function, and other components specific to the model.
- **`bounding_box.py`**: This is the implementation of the custom face detection model. This file includes the convolutional architecture and training process of the bounding box model. It is an implementation that predicts the bounding box coordinates of detected faces.
- **`bounding_box_yunet.py`**: This is another face detection implementation using the `YuNet` face detection model, which is used as an alternative face detection approach.

Summary

The described facial recognition system is a sophisticated application designed for robust face detection and recognition, integrating custom-trained and pre-trained deep learning models. The system is implemented using Python, leveraging a range of specialized libraries to ensure functionality and performance. The application aims to provide a complete, user-friendly solution, from model training to practical deployment via a Graphical User Interface (GUI). The system uses advanced techniques in deep learning, loss function optimization, and rigorous evaluation metrics to achieve its goals.

Key Components and Features:

- **Real-time face detection and recognition:** The system performs face detection and recognition in real-time, making it suitable for interactive applications.

- **Custom-trained models:** The system utilizes custom-trained deep learning models for both face detection and recognition, taking into account that the custom face detector is not as optimized for the specific task of esp32-cam, so we are using YuNet instead.
- **User registration and authentication:** The system supports user registration, which includes capturing multiple face samples to create a user profile and provides authentication by matching detected faces with registered users.
- **Access control management:** The system allows for access permissions to be assigned to registered users.
- **GUI Interface:** A graphical user interface, built with CustomTkinter, provides an easy-to-use way to interact with the system.
- **IP Camera Integration:** The system is compatible with IP cameras, and tested using an ESP32-CAM, for image input.
- **Queue-Based Embedding Averaging:** The system averages multiple face embeddings to provide more accurate recognition results.
- **Thread-Safe Database Operations:** The system implements thread-safe database operations to ensure smooth, concurrent functioning.

Model Details:

- **Face Detection Model:**
    - A custom-trained Convolutional Neural Network (CNN) is used for face detection.
    - The model is trained on the CelebA dataset, comprising over 200,000 facial images.
    - Input size: 224x224 pixels; Output: four coordinates (x, y, width, height).
    - Alternatively, the YuNet face detector model is implemented for face detection
- **Face Recognition Model:**
    - An **ArcFace-based architecture** with residual connections is utilized for face recognition.
    - The model's **embedding dimension is 512**.
    - The model is trained with an **ArcFace loss function with dynamic scaling and margin parameters**.
    - The model uses face alignment as a preprocessing step.
    - The model is trained on the VGGFace2-HQ cropped dataset and tested on the PINS face recognition dataset.

Project Structure:

The project is organized as follows:

- `app.py`: Main application using custom models.
- `insightface_app.py`: Alternative implementation using InsightFace.
- `embeddings.py`: Contains the face recognition model implementation.
- `bounding_box.py`: Contains the custom face detection model implementation.
- `bounding_box_yunet.py`: Contains the YuNet face detector implementation.
- `models/`: Directory containing the trained models.
    - `bbox_models/`: Contains face detection models.
    - `v5/`: Contains custom-trained face detection models.
    - `YuNet/`: Contains YuNet face detector models.
    - `resarksgd/`: Contains face recognition models.

Installation and Usage:

- The system requires Python 3.8+ and CUDA-capable GPU is recommended.
- The repository can be cloned from GitHub, and a virtual environment can be set up.
- Required dependencies can be installed using `pip`.
- Pre-trained models must be downloaded and placed in the correct directory.
- The main application can be launched by running `app.py`, or alternatively, `insightface_app.py`.
- The system supports user registration with multiple face samples (preferably not more than 15) and authentication via a live camera feed.

Resources and Citations

- **CelebA Dataset:** The CelebA dataset is used for training the custom face detection model. This dataset contains over 200,000 celebrity images with bounding box and facial landmark annotations:
    - https://huggingface.co/datasets/hfaus/CelebA_bbox_and_facepoints
- **VGGFace2-HQ Cropped Dataset:** The VGGFace2-HQ Cropped dataset is used for training the face recognition model:
    - https://www.kaggle.com/datasets/zenbot99/vggface2-hq-cropped
- **PINS Face Recognition Dataset:** The PINS dataset is used for testing the face recognition model:
    - https://www.kaggle.com/datasets/hereisburak/pins-face-recognition
- **Training Notebooks:** The following notebooks contain code for training the face detection and recognition models:

    - **Face Detection (CelebA):**
        - https://www.kaggle.com/code/ahmedkamal75/facial-detection-celeba-bbox-optimized
    - **Face Recognition (ArcFace):**
        - `models/resarksgd/arkface-residual-connections-part-2.ipynb`
        - https://www.kaggle.com/code/ahmedkamal75/arkface-residual-connections-part-2
- **GitHub Repository:** The code for the face recognition system can be found at:
    - https://github.com/AhmedKamal75/Graduation_Project
- **Pre-trained Models:** The pre-trained models can be downloaded from the following links:
    - https://www.kaggle.com/models/ahmedkamal75/bbox_model_v5_epoch_8/
    - https://www.kaggle.com/models/ahmedkamal75/bbox_v5_augmented_epoch_50
    - https://www.kaggle.com/models/ahmedkamal75/resarksgd-acc-88/
    - https://www.kaggle.com/models/ahmedkamal75/resarksgd95/
    - resarksgdaug94