Student ID Cards Based Attendance System

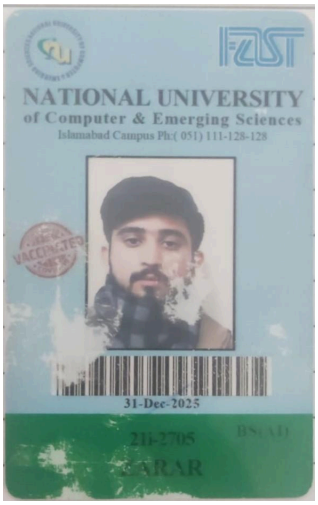
Name : Muhammad Zaraar Malik
Roll Number : 21i-2705
Section : AI-A
Course Instructor : Khadija Mehmood
Course : Computer Vision (Fall 2024)

Detailed Project Description

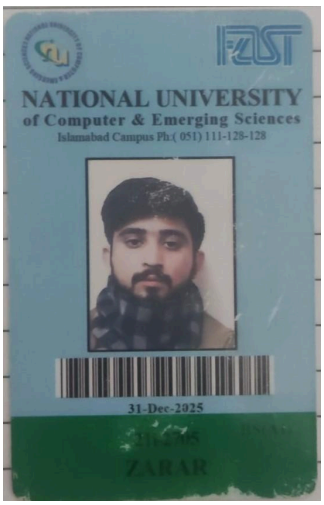
The current system of marking attendance of students is such that the teachers have to verbally ask each student's name at either the beginning or end of the class and mark their attendance.
One more problem is the “proxy system” implemented by the students such that due to an overwhelming number of students some try to mark the attendance of other students by fooling the teacher. So, an automatic system which just marks the attendance via the student's id card can be a possible solution in order to have a streamlined process for this task.

Data Collection Phase

As each university has a distinct type of ID cards for their students, I will be focused only on the Students of FAST Islamabad in order to reduce the Scope of the project. Following is a sample of how the Dataset will look like



Front Image



Back Image

As we can see that there is no differentiation between the front and back of the image however I will take both images as the condition of the ID Card is not good, Some miss printing and missing face image due to the worn out condition of the ID Card. Similarly, data will be collected from the Students mainly from AI -A and AI-B and then moving on to other students of 21st Batch. If the need arises, I will move to Students from the 22nd Batch.

Currently the collected data consists of 36 images or 18 image pairs each belonging to a particular student at FAST.

Data Exploration Phase

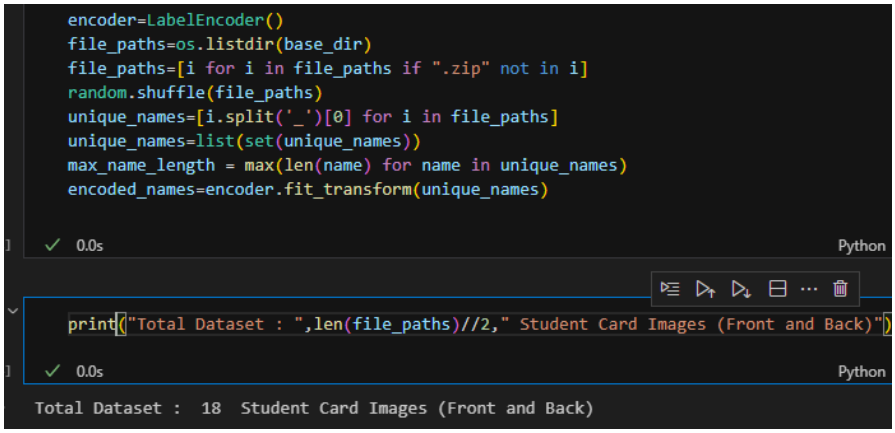
After the data was collected, the main objective was to explore the data keeping in view the following points:

- Total number of samples
- How to assign labels to the data
- Manual vs Pretrained Image Extraction

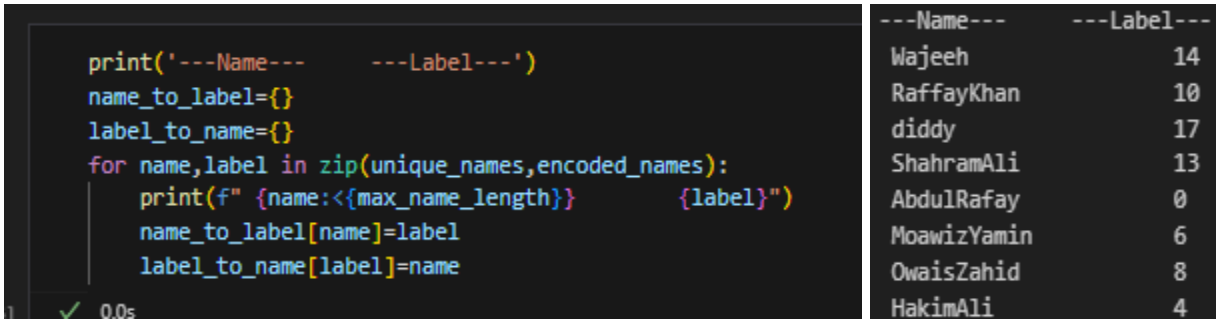
So, the image were stored in the local directory in the following format:

- FirstName_LastName_Front.jpg or FirstName_LastName_front.jpg
- FirstName_LastName_Back.jpg or FirstName_LastName_back.jpg

The images were read 1 by 1 from the local directory and then I extracted the first name and last name from the names of the files. Then I initialized a LabelEncoder. The purpose of this label encoder was to assign a unique integer value to each of the names in the files. These encoded names would serve as a label while training the model. Following are some code snippets for the above mentioned process :



Finding total number of image id in the Dataset



Encoding the names of the students

String and Vector Relationship

The next step after this was to decide how to decide what model and what approach I would be using for this project. I had the following things in mind:

- Train a Complex Architecture from scratch
- Implement Transfer learning and add my custom Model to a pretrained network
- Instead of Transfer Learning, using Fine-Tuning and train a complex architecture

Problems with "Complex Architecture" and "Fine-Tuning"

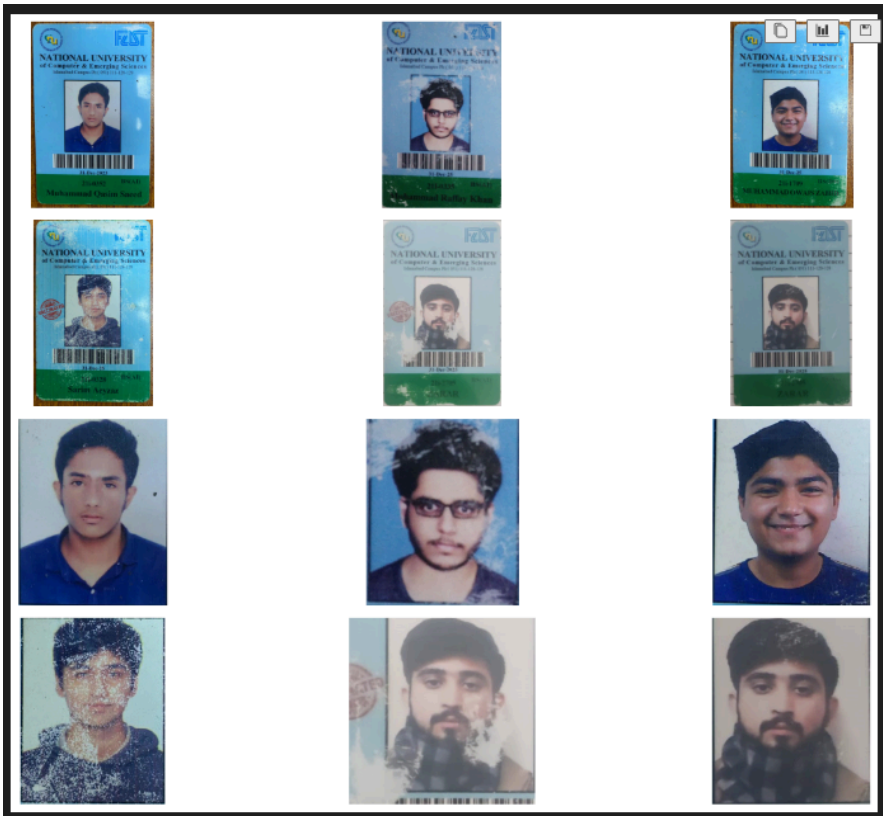
The main problem with my entire project is an extremely low dataset at hand. I mean there are a total of 36 images in the dataset, meaning there are 18 different individuals or 18 different student id cards and for each label or student there are just 2 images, the front image and the back image. So, keeping in view whichever model I choose, they have been trained on massive amounts of data which I was not able to produce during my project.

So, I decided to go with the final approach that was to use a pretrained model for a down streamed task integrated with my custom neural network that would mainly be doing the logical stuff which is meant to be implemented in this course. But I had to make it sure that it was completely

necessary to use a model and the task could not have been done using a statistical approach.
Following is are the results of human face extraction using a statistical approach:

Human Face Extraction

As you can see, the first 2 rows show the images which are inside the dataset.
The next 2 rows show the extracted human faces using a statistical approach.
Now, we can see that there are some unwanted areas in these images such as some left over part of the id card coloured as blue in the background. Now, this can serve as noise to our model during the training process. Yes, Noise is required during the training process as well so that our model does not overfit but at the same time, having such a low number of training samples, I decided not to have any unwanted noise in my dataset.
Following is the output of “Yolov9n”, this is a nano version of the base model which can be used easily for inference:



Human Face Extraction

As you can see in these images, the pretrained model was completely focused on the human faces and not anything else. Yes, we can see in the last row that due to the extremely poor quality of the image, the model has included some portion of the id card which does not consist of the human and can act as extra unwanted noise which can further lead to the destabilization of the training process.

Model Architecture

Before selecting the model architecture, i had to select the pretrained model for the human face extraction. For this, i had the following options :

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

YOLO V8

Model	size (pixels)	mAP^{val} 50-95	mAP^{val} 50	params (M)	FLOPs (B)
YOLOv9t	640	38.3	53.1	2.0	7.7
YOLOv9s	640	46.8	63.4	7.2	26.7
YOLOv9m	640	51.4	68.1	20.1	76.8
YOLOv9c	640	53.0	70.2	25.5	102.8
YOLOv9e	640	55.6	72.8	58.1	192.5

YOLO V9

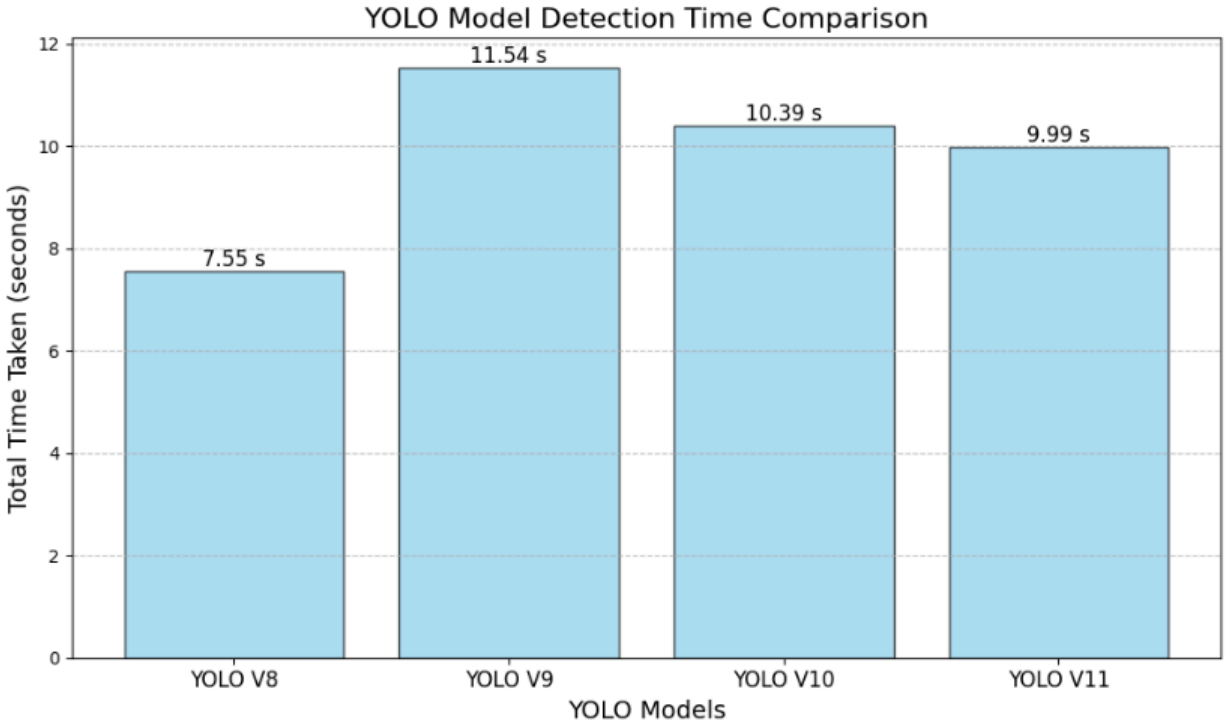
Model	Input Size	AP^{val}	FLOPs (G)	Latency (ms)
YOLOv10-N	640	38.5	**6.7**	**1.84**
YOLOv10-S	640	46.3	21.6	2.49
YOLOv10-M	640	51.1	59.1	4.74
YOLOv10-B	640	52.5	92.0	5.74
YOLOv10-L	640	53.2	120.3	7.28
YOLOv10-X	640	**54.4**	160.4	10.70

YOLO V10

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	params (M)	FLOPs (B)
YOLO11n	640	39.5	56.1 ± 0.8	1.5 ± 0.0	2.6	6.5
YOLO11s	640	47.0	90.0 ± 1.2	2.5 ± 0.0	9.4	21.5
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0
YOLO11l	640	53.4	238.6 ± 1.4	6.2 ± 0.1	25.3	86.9
YOLO11x	640	54.7	462.8 ± 6.7	11.3 ± 0.2	56.9	194.9

YOLO V11

Keeping in view some of the above mentioned parameters of each of the model, i decided to use all of these models and check their performance on my dataset as these are the metrics derived from the COCO Dataset. There can be a very high chance that the performance metrics displayed here would be the same on my dataset as during the data collection process, the images were taken using the camera of mobile phones, there is a high chance that students might not have taken care of their “Flashes” and it has been stated that YOLO is highly sensitive to Light.
Following is the inference speed of the models on my dataset :



As we can see that yolo v8 has the best inference speed so far. But another question can arise that despite having a quick speed maybe the detected images can be wrong. Following is the comparison between the detected images from each model

YOLO V8

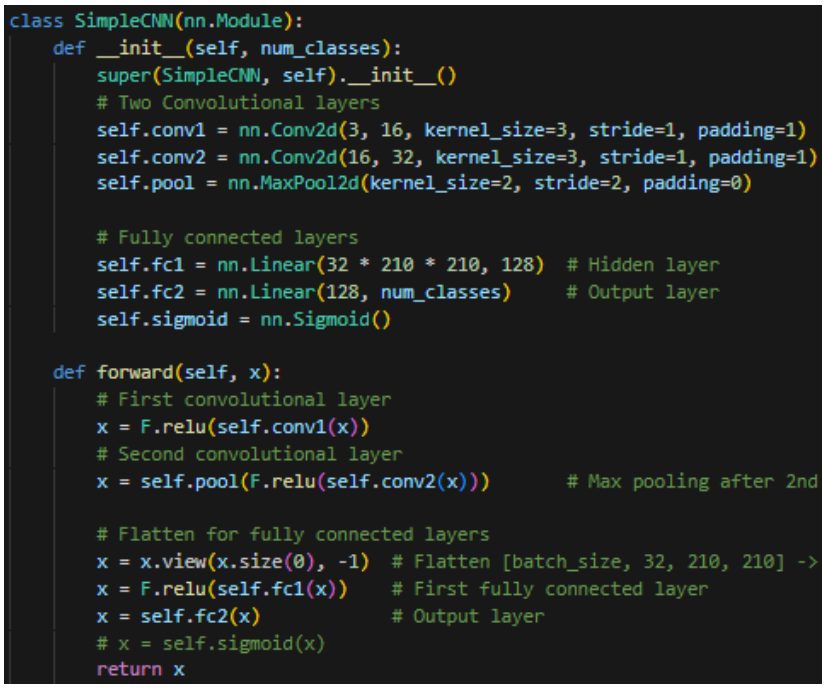
YOLO V9

YOLO V10

YOLO V11

As we can see from the above displayed image, my personal image was also acting as an edge case during the testing phase, the amazing thing about this is that despite being a better version of V9 , V10 was still unable to properly detect my face and this is quite a fascinating finding.

Now, I will talk about the Custom Models that I developed to implement the transfer learning process. I made 2 custom Neural Network classes which accepted an image of size (420,420).
The unique thing about these 2 classes is that one class contains 2 Convolution Filters followed by max pooling layers and finally a fully connected layer to make the classification.
Conversely, the other class only contains a Vanila Neural Network with just fully connected layers. The main objective of this was to experiment and prove the global context ability of the CNN network and how this can have a huge impact on the project. Moreover, another objective was to show experiment and identify that using CNN layer can decrease the model training time as there are less number of parameters to train due to the fact that weight sharing is also an intrinsic property of the CNN architecture and that linear layers will not only underperform but also it will take more time to train the model with basically no improvement in results.



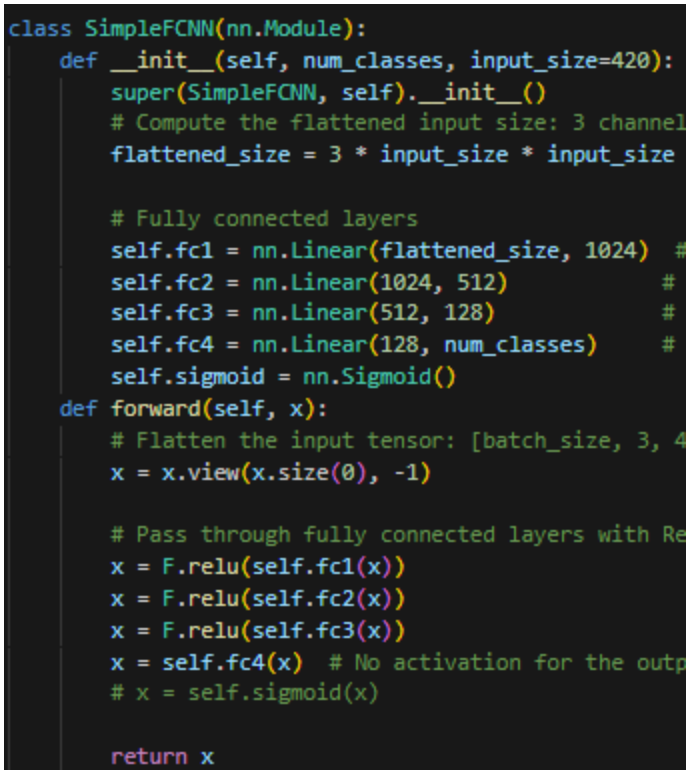
```python
class SimpleCNN(nn.Module):
    def __init__(self, num_classes):
        super(SimpleCNN, self).__init__()
        # Two Convolutional layers
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

        # Fully connected layers
        self.fc1 = nn.Linear(32 * 210 * 210, 128)  # Hidden layer
        self.fc2 = nn.Linear(128, num_classes)     # Output layer
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # First convolutional layer
        x = F.relu(self.conv1(x))
        # Second convolutional layer
        x = self.pool(F.relu(self.conv2(x)))       # Max pooling after 2nd

        # Flatten for fully connected layers
        x = x.view(x.size(0), -1)  # Flatten [batch_size, 32, 210, 210] ->
        x = F.relu(self.fc1(x))    # First fully connected layer
        x = self.fc2(x)            # Output layer
        # x = self.sigmoid(x)
        return x
```

This is the CNN architecture which I have developed.



```python
class SimpleFCNN(nn.Module):
    def __init__(self, num_classes, input_size=420):
        super(SimpleFCNN, self).__init__()
        # Compute the flattened input size: 3 channel
        flattened_size = 3 * input_size * input_size

        # Fully connected layers
        self.fc1 = nn.Linear(flattened_size, 1024)  #
        self.fc2 = nn.Linear(1024, 512)             #
        self.fc3 = nn.Linear(512, 128)              #
        self.fc4 = nn.Linear(128, num_classes)      #
        self.sigmoid = nn.Sigmoid()
    def forward(self, x):
        # Flatten the input tensor: [batch_size, 3, 4
        x = x.view(x.size(0), -1)

        # Pass through fully connected layers with Re
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = self.fc4(x)  # No activation for the outp
        # x = self.sigmoid(x)

        return x
```

This is the Fully Connected Neural Network class that I have developed.

Overall Flowchart

Conclusion

With this the semester project has been finalized, some tweaks are still left but this is as far as we can go.