Project Report
Facial detection & recognition of Dexter's Laboratory characters

Introduction and Objectives

This project is a homework for the *Computer Vision* class and focuses on implementing various algorithms to **detect and recognize** faces of different characters from Dexter's Laboratory animated series.

The project is composed of **2 tasks**:

i. Detecting the faces of every character;
ii. Recognizing the faces of Dexter, DeeDee and their parents.

I tried **2 different approaches** for solving the problem:

1. **Sliding window technique**, feature extraction using **Histogram of Oriented Gradients** and classifying with a **neural network**;
2. **Transfer learning** on a **YOLO model** architecture using **pretrained weights.**

1. Sliding window, HOG & NN

This detailed description is for task i. Task ii will be described at the end of this part.

The functions *get_positive_hog* and *get_negative_hog* are implemented in the *feature_extraction.py* file and are used to get the **positive and negative descriptors** from grayscale training images for creating the classifier.

get_positive_hog has 3 parameters:

- **dim_window** (integer) - the annotated face is resized to a size of dim_window x dim_window;
- **dim_hog_cell** (integer) - the size of a cell for the HoG feature extraction is dim_hog_cell x dim_hog_cell**;**
- **augment** (boolean) - if True then we also create horizontal flipped versions of all the faces to add to the positive descriptor array.

get_negative_hog has 6 parameters:

- **dim_window** (integer) - same as in *get_positive_hog;*
- **dim_hog_cell** (integer) - same as in *get_positive_hog;*
- **neg_per_image** (integer) - the number of negative examples to extract from every image in the training set;
- **min_patch_size** (integer) - the minimum dimension of the random patch chosen;
- **max_patch_size** (integer) - the maximum dimension of the random patch chosen;
- **max_iou** (float) - the maximum ratio of Intersection over Union between the random patch and an annotated face.

The functions work in the following way:

get_positive_hog - Iterate over training character directories, loads images and associated bounding box annotations. For each image, crop regions corresponding to the bounding boxes. Resize the cropped region to dim_window x dim_window. Compute the HOG feature vector for the cropped face. If augment is True, compute HOG features for the horizontally flipped version of the face. Return a NumPy array of positive HOG descriptors. Each row represents the feature vector of a face.

get_negative_hog - Iterate over training character directories, loads images and associated bounding box annotations. Randomly select square patches from each image of a random size between min_patch_size and

max_patch_size. Ensure the selected patch does not overlap significantly with annotated bounding boxes by calculating IoU value for each annotated face in the image. If IoU ≤ max_iou for every face, resize the patch to dim_window x dim_window and compute the HOG feature vector for the resized patch. Return a NumPy array of negative HOG descriptors. Each row represents the feature vector of a non-face region.

For the classification I used a **feedforward neural network** with the following architecture:

```
def __init__(self, input_length):
    super().__init__()
    self.flatten = nn.Flatten()
    self.first_layer = nn.Linear(input_length, out_features: 256)
    self.second_layer = nn.Linear( in_features: 256, out_features: 64)
    self.output_layer = nn.Linear( in_features: 64, out_features: 2)
    self.device = "cuda" if torch.cuda.is_available() else "cpu"
    self.to(self.device)

def forward(self, x):
    if x.ndim == 1:
        x = x.unsqueeze(0)
    x = self.flatten(x)
    x = F.relu(self.first_layer(x))
    x = F.relu(self.second_layer(x))
    x = self.output_layer(x)
    return x
```

It accepts a 1D input of length N and passes it through three fully connected layers. The first two layers, with 256 and 64 neurons respectively, use ReLU activation to introduce non-linearity. The final layer outputs scores for two classes.

The loss function is **CrossEntropy** and the optimizer is **SGD** with a *learning rate* of 0.01 and *momentum* of 0.9.

The implementation of the network can be found in the *neural_network.py* file, along with the following **dataset class** which uses the positive and negative descriptor files to create a dataloader so we can train the model with batches.

```python
class HOGDataset(Dataset):
    def __init__(self, negative_file, positive_file):
        self.negative_data = np.load(negative_file)
        self.positive_data = np.load(positive_file)

        self.negative_labels = np.zeros(self.negative_data.shape[0], dtype=np.float32)
        self.positive_labels = np.ones(self.positive_data.shape[0], dtype=np.float32)

        self.data = np.concatenate( arrays: [self.negative_data, self.positive_data], axis=0)
        self.labels = np.concatenate( arrays: [self.negative_labels, self.positive_labels], axis=0)

        self.data = torch.tensor(self.data, dtype=torch.float32)
        self.labels = torch.tensor(self.labels, dtype=torch.long)

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

    def __getitem__(self, idx):
        return self.data[idx], self.labels[idx]
```

The *run_project.py* file contains the *run* function that iterates through each image, using a sliding window approach to detect faces. After each iteration, we reduce the dimension of the image by a scale factor, thus being able to detect bigger faces. For each picture we reduce overlapping detections by applying the *non_maximal_suppression* function from the *helpers.py* file.

The *run* function has 5 parameters:

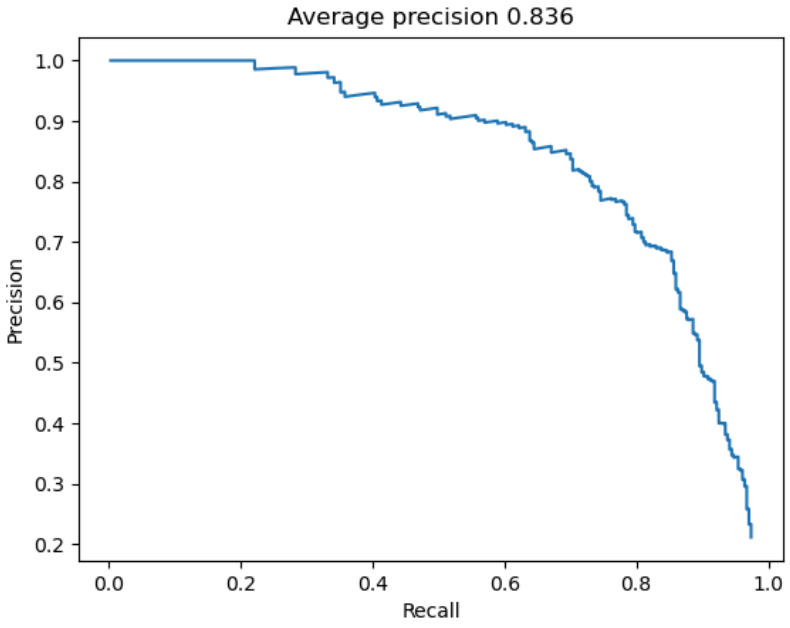
- **model** - the model used for getting the score for each window;
- **dim_hog_cell** (integer) - same as in *get_positive_hog;*
- **dim_window** (integer) - same as in *get_positive_hog;*
- **score_threshold** (float) - minimum threshold for a detection to be considered valid, detections with scores below this value are discarded;
- **scale_factor** (float) - factor by which images are resized at each step.

For the **task i**, I used the following parameters:

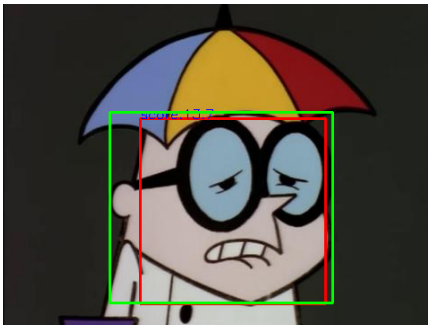
- General
  - dim_window = 64
  - dim_hog_cell = 8
- For positive & negative descriptors
  - augment = True
  - neg_per_image = 100
  - min_patch_size = 10
  - max_patch_size = 210
  - max_iou = 0.35
- Neural network
  - batch_size = 128
  - num_epochs = 18

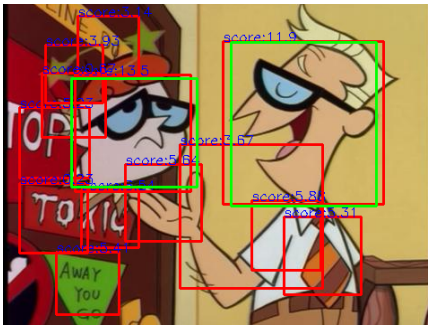
- Sliding window
    - score_threshold = 0
    - scale_factor = 0.98

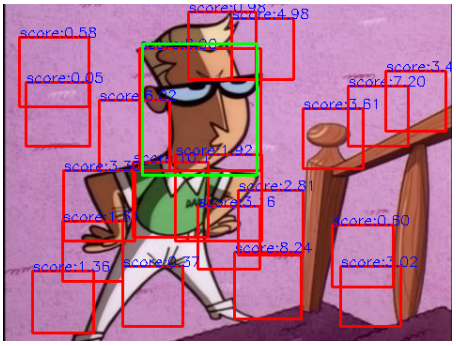
The result of testing on the validation set (it contains 200 annotated images that were not used for training) is the following:



Some results of task i with this approach look like this (green are the annotated faces and red are the predictions of the model):

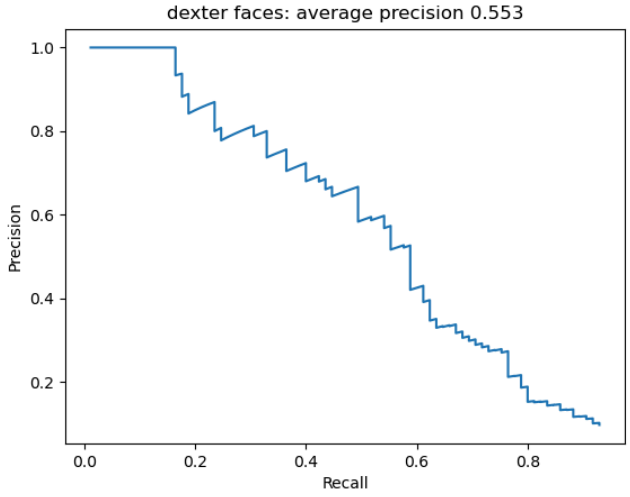


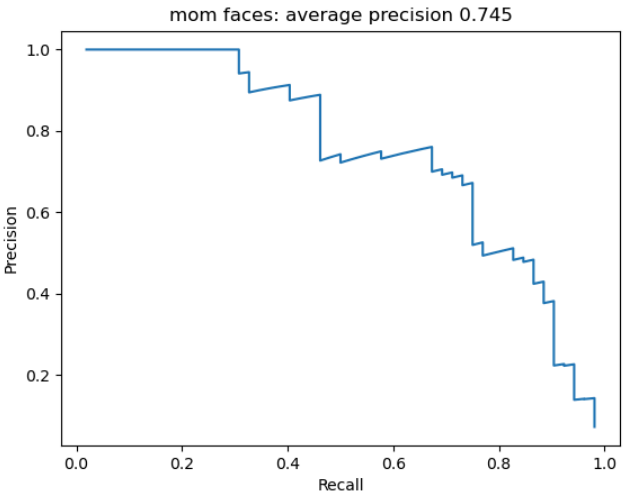


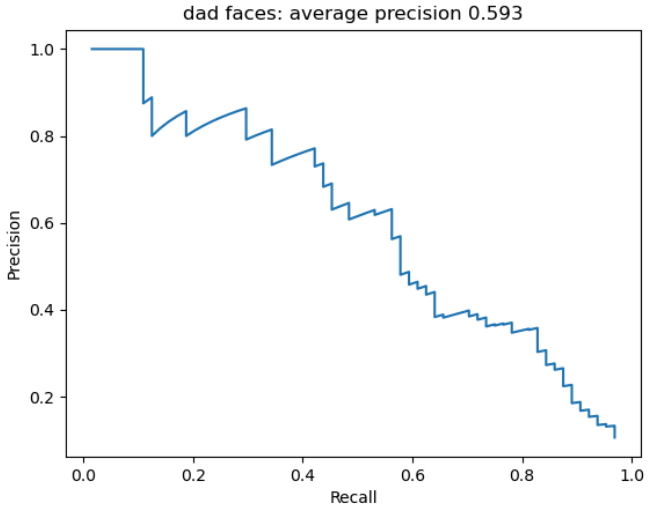


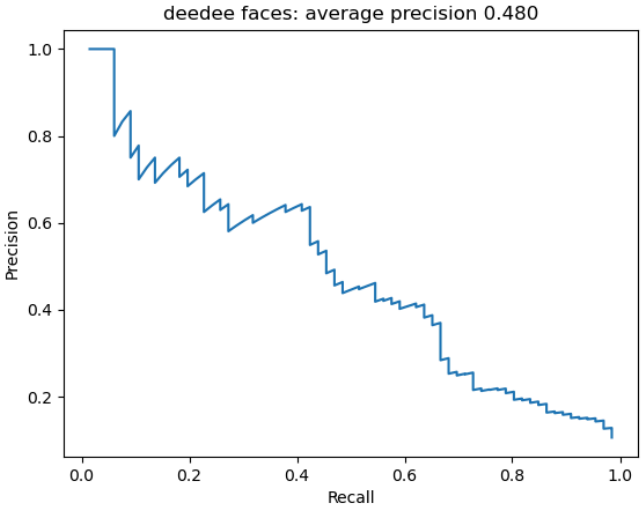
We can see that the model has a good recall, being able to detect the faces of characters even if there are some false positives.

For task ii (character recognition) I used the same approach, the only difference being that the negative descriptors for a character (for example, dad) also contain the positive descriptors for the other 3 characters (deedee, dexter and mom). The results are as follows:









This way, I get a mean Average Precision of 0.592 for task ii on the validation data.

2. Transfer Learning on a YOLO architecture

For this part I used Ultralytics YOLOv5. Both tasks will be detailed here since they are pretty similar.

First, I created scripts that convert the training and validation annotated data to a format that is compatible with the YOLO algorithm. It uses the format *<object-class> <x_center> <y_center> <width> <height>*. The scripts are the *convert_annotations_train.py* and *convert_annotations_val.py*. To be noted that

the validation data is not necessary since it is not used to update the weights of the model.

Creating the setup was easy, the commands I used are the following:

- git clone https://github.com/ultralytics/yolov5.git
- cd yolov5
- pip install -r requirements.txt

After that, I created an *dataset.yaml* file that looks like this (for task ii).

```
train: ../images/train
val: ../images/val

nc: 4

names: ['dad', 'deedee', 'dexter', 'mom']
```

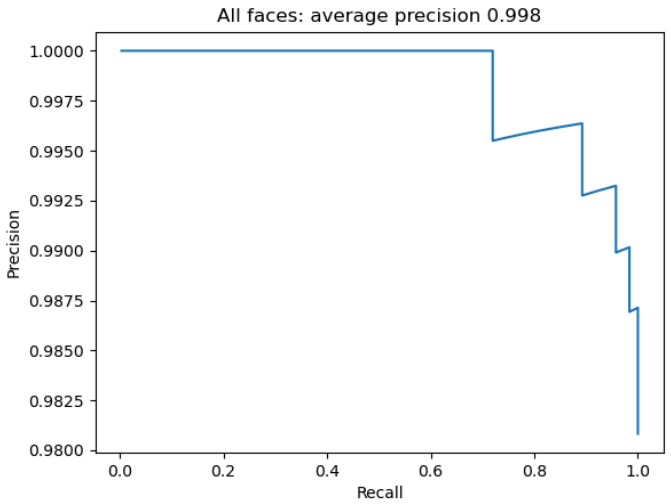
The following command was used to get weights pretrained on COCO:

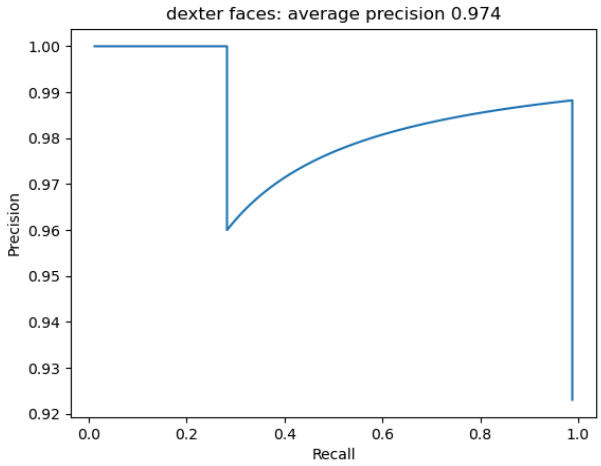
- *wget https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.pt*

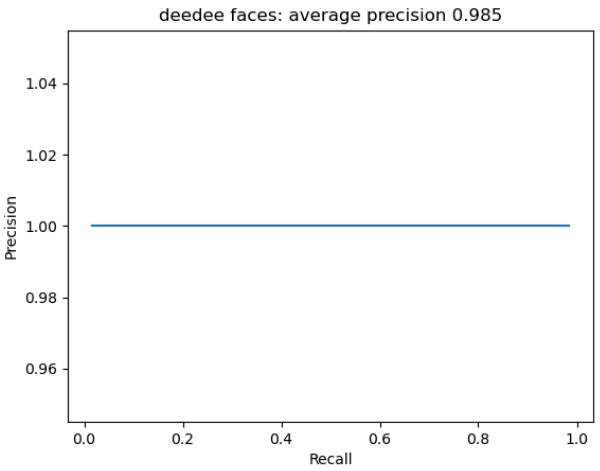
The models for both tasks were trained with the following command:

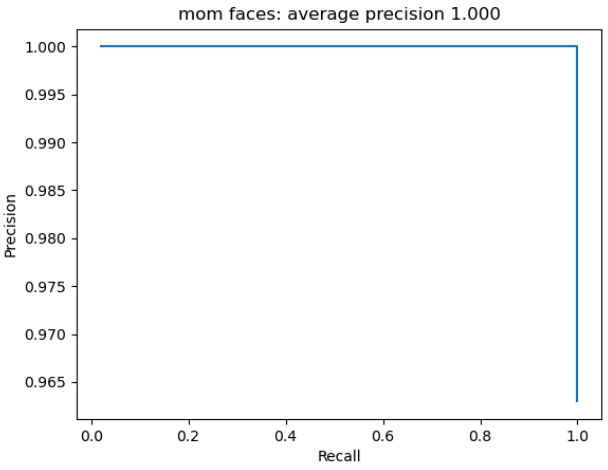
- *python train.py --img 480 --batch 16 --epochs 50 --data ../dataset.yaml --weights yolov5s.pt*

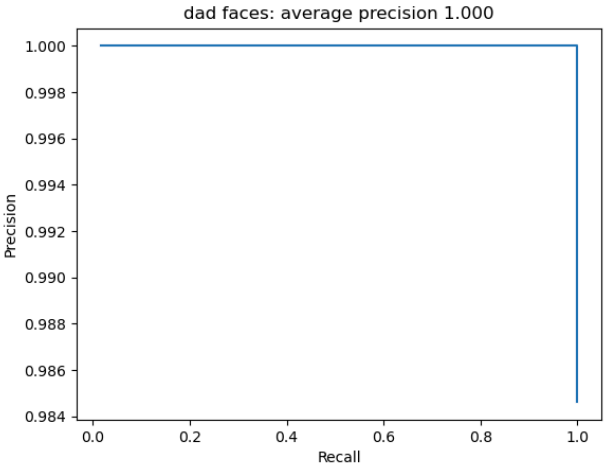
In the end, I created files named *run_yolo.py* for both tasks that runs inference on the model on an image directory and save the results in numpy files compatible with the evaluation script. The results of the YOLO on the validation are the following:




dexter faces: average precision 0.974
Precision
Recall
1.00
0.99
0.98
0.97
0.96
0.95
0.94
0.93
0.92
0.0
0.2
0.4
0.6
0.8
1.0


deedee faces: average precision 0.985
Precision
Recall
1.04
1.02
1.00
0.98
0.96
0.0
0.2
0.4
0.6
0.8
1.0


mom faces: average precision 1.000
Precision
Recall
1.000
0.995
0.990
0.985
0.980
0.975
0.970
0.965
0.0
0.2
0.4
0.6
0.8
1.0


dad faces: average precision 1.000
Precision
Recall
1.000
0.998
0.996
0.994
0.992
0.990
0.988
0.986
0.984
0.0
0.2
0.4
0.6
0.8
1.0