Face detection and recognition of characters from "Life with Louie" TV Show

1. Problem description

Given a set of images from "Life with Louie" TV show we need to detect characters faces for task 1 and to recognize the 4 main characters (Andy, Louie, Ora, Tommy) for task 2. For task 1 we will extract features using HOG and train a classifier that will detect if there is a face or not. We need to implement the sliding window technique and output the detections, scores and files names. For task2 we will train ResNet-18 architectures to classify between the 4 classes.

2. Data description

For training we have 4000 images having 7236 annotated faces devided in the 4 classes, so 1000 images for each class.

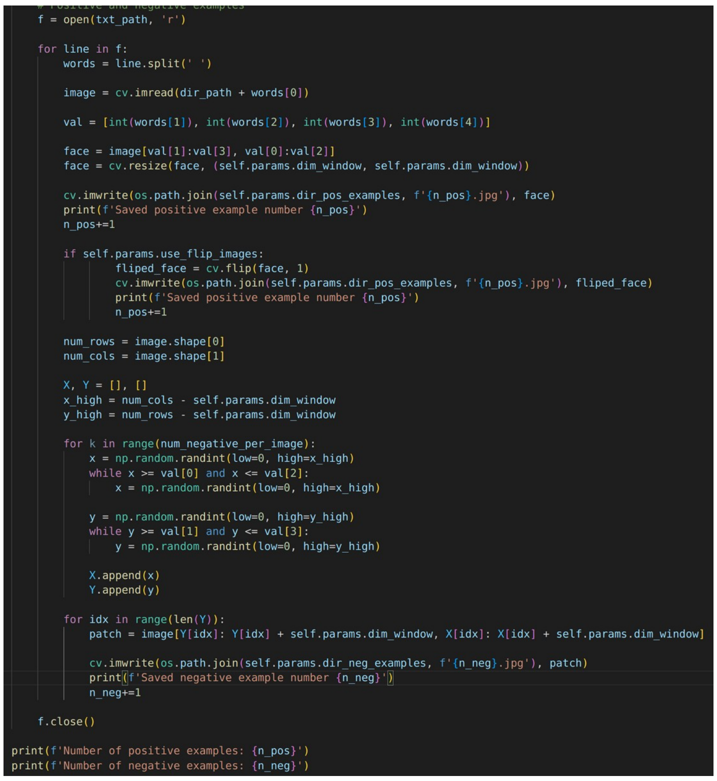
For validation we have 200 annotated images. The same for testing.

3. Detection task

The main challenge for this task was given by the fact that the faces have multiple shapes and multiple aspect ratios. We will train a classifier for 1:1 aspect ratio and for inference we will process the image and slide a 1:1 window of size 60x60.

3.1 Data preparation

We take every image and crop the faces given by the ground truth, resize them to have 60x60, flip the patch and save them to *'exemple_pozitive'* directory. For negative examples we take a random patch of size 60x60 from the image and be careful not to crop some part of a face, then we save the patches to *'exemple_negative'* directory.



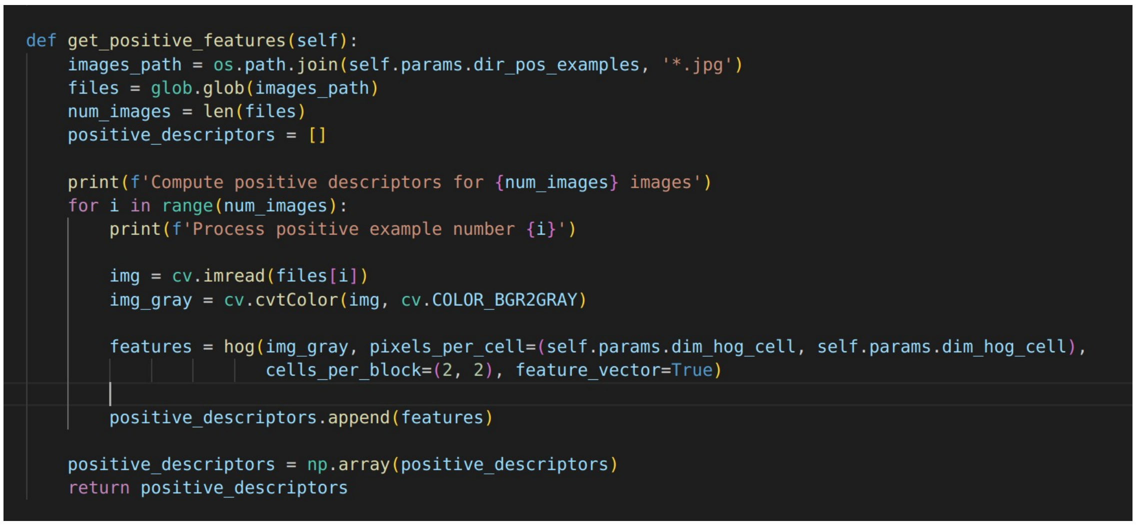
```
    f = open(txt_path, 'r')

    for line in f:
        words = line.split(' ')

        image = cv.imread(dir_path + words[0])

        val = [int(words[1]), int(words[2]), int(words[3]), int(words[4])]

        face = image[val[1]:val[3], val[0]:val[2]]
        face = cv.resize(face, (self.params.dim_window, self.params.dim_window))

        cv.imwrite(os.path.join(self.params.dir_pos_examples, f'{n_pos}.jpg'), face)
        print(f'Saved positive example number {n_pos}')
        n_pos+=1

        if self.params.use_flip_images:
                fliped_face = cv.flip(face, 1)
                cv.imwrite(os.path.join(self.params.dir_pos_examples, f'{n_pos}.jpg'), fliped_face)
                print(f'Saved positive example number {n_pos}')
                n_pos+=1

        num_rows = image.shape[0]
        num_cols = image.shape[1]

        X, Y = [], []
        x_high = num_cols - self.params.dim_window
        y_high = num_rows - self.params.dim_window

        for k in range(num_negative_per_image):
            x = np.random.randint(low=0, high=x_high)
            while x >= val[0] and x <= val[2]:
                x = np.random.randint(low=0, high=x_high)

            y = np.random.randint(low=0, high=y_high)
            while y >= val[1] and y <= val[3]:
                y = np.random.randint(low=0, high=y_high)

            X.append(x)
            Y.append(y)

        for idx in range(len(Y)):
            patch = image[Y[idx]: Y[idx] + self.params.dim_window, X[idx]: X[idx] + self.params.dim_window]

            cv.imwrite(os.path.join(self.params.dir_neg_examples, f'{n_neg}.jpg'), patch)
            print(f'Saved negative example number {n_neg}')
            n_neg+=1

    f.close()

print(f'Number of positive examples: {n_pos}')
print(f'Number of negative examples: {n_neg}')
```

Figure 1: Cropping positive and negative examples

The dataset has 14472 positive examples and 14472 negative examples.

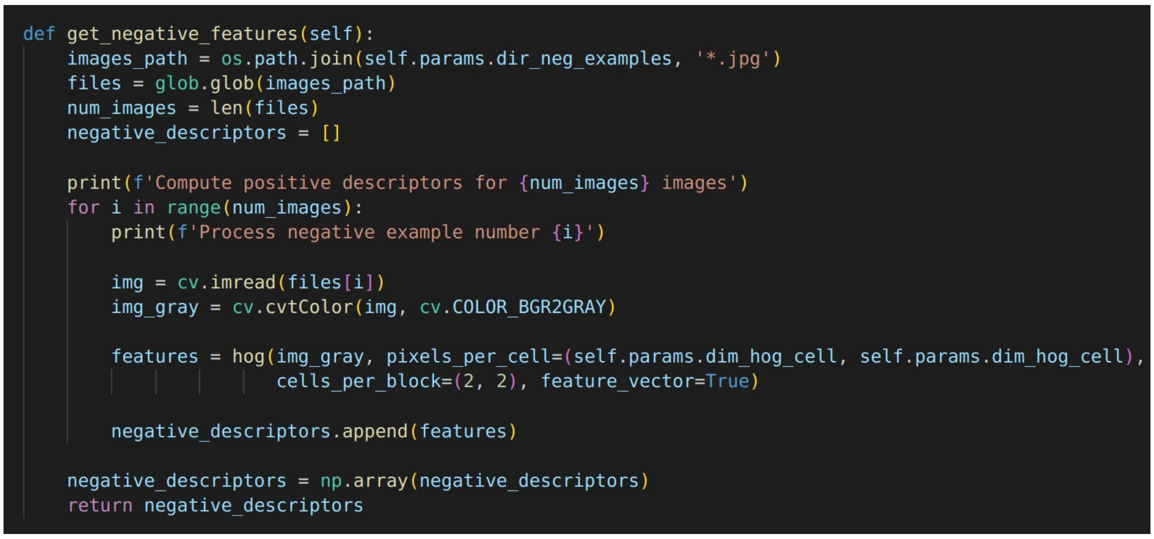
After cropping the images we need to extract hog features from them and save them on the disk as numpy arrays for training our classifier.

For 60x60 images we can choose 6x6 hog cell and 2x2 cells per block. Load every image and use **skimage.feature.hog** for extraction. We do the same for both positive and negative examples.



```python
def get_positive_features(self):
    images_path = os.path.join(self.params.dir_pos_examples, '*.jpg')
    files = glob.glob(images_path)
    num_images = len(files)
    positive_descriptors = []

    print(f'Compute positive descriptors for {num_images} images')
    for i in range(num_images):
        print(f'Process positive example number {i}')

        img = cv.imread(files[i])
        img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

        features = hog(img_gray, pixels_per_cell=(self.params.dim_hog_cell, self.params.dim_hog_cell),
                       cells_per_block=(2, 2), feature_vector=True)

        positive_descriptors.append(features)

    positive_descriptors = np.array(positive_descriptors)
    return positive_descriptors
```

Figure 2: Extract positive features



```python
def get_negative_features(self):
    images_path = os.path.join(self.params.dir_neg_examples, '*.jpg')
    files = glob.glob(images_path)
    num_images = len(files)
    negative_descriptors = []

    print(f'Compute positive descriptors for {num_images} images')
    for i in range(num_images):
        print(f'Process negative example number {i}')

        img = cv.imread(files[i])
        img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

        features = hog(img_gray, pixels_per_cell=(self.params.dim_hog_cell, self.params.dim_hog_cell),
                       cells_per_block=(2, 2), feature_vector=True)

        negative_descriptors.append(features)

    negative_descriptors = np.array(negative_descriptors)
    return negative_descriptors
```

Figure 3: Extract negative features

3.2 Training the linear classifier

We choose a Linear SVM classifier to train on different values for hyperparameter C ranging from 1e-5 to 1e0. The best value was C=1e0(=1) with an accuracy of 99.9% on training data and 94% on validation data. Figure 4 shows the linear separability of positive and negative exmaples. In the ideal case the positive examples should get a score > 0 and negative examples should get a score < 0.

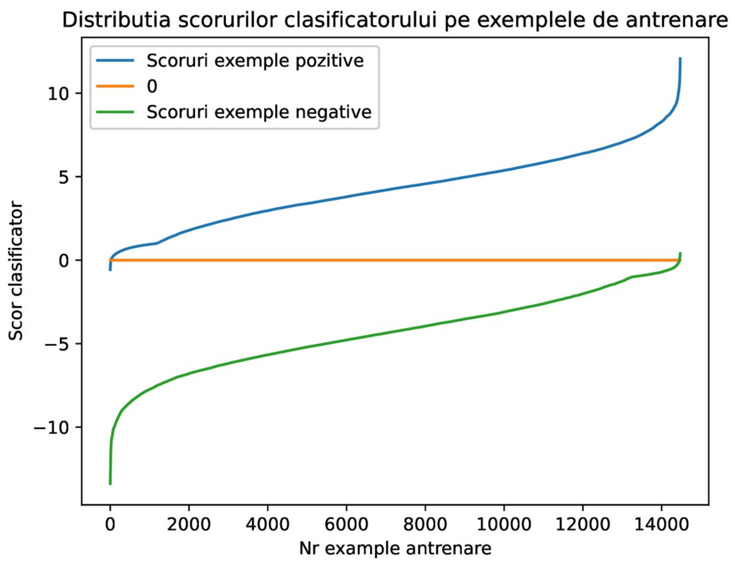
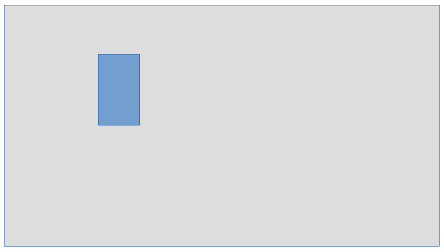
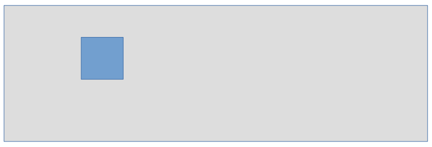


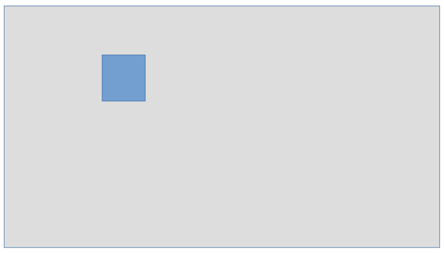
Figure 4: Linear separability of training data

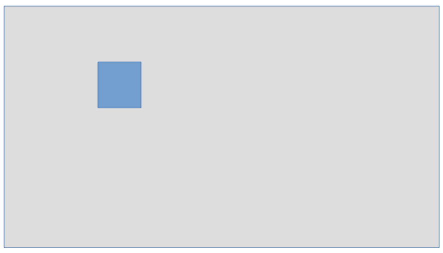
3.3 Sliding window

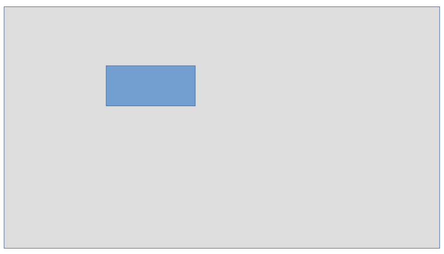
With our classifier we start to slide a window across the image and get a classification score for each patch. If the score is higher than a threshold we take that patch as a detection and save it. We will have multiple window ratios, one for each possible head ratio, and multiple scales for every size of head. We want to resize the image such that every head with ratio 3:4 (0.75) will have ratio 1:1 (1.0), then we calculate the score for that resized patch. If we get a detection the coordonates need to be transformed back so that they match the head of the character in the original image. Figure 5 shows the code that resize the image, the sliding window and the recovery of the coordonates.

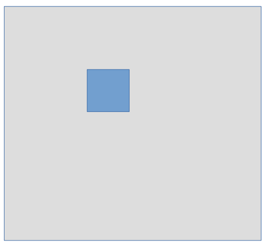

Drawing 1: Window ratio 1:2

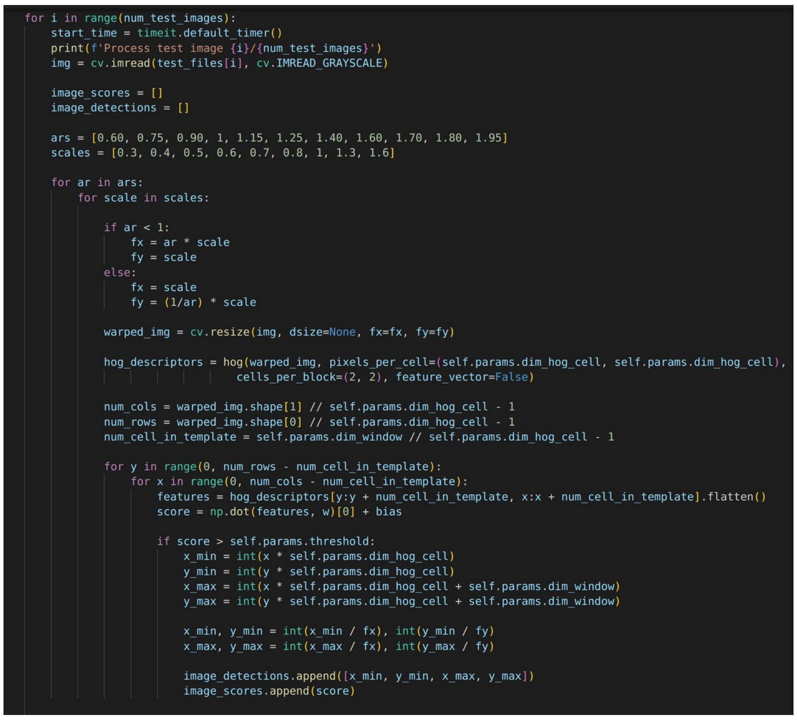

Drawing 2: Window ratio 1:1


Drawing 4: Window ratio 1:1


Drawing 3: Window ratio 1:1


Drawing 6: Window ratio 2:1

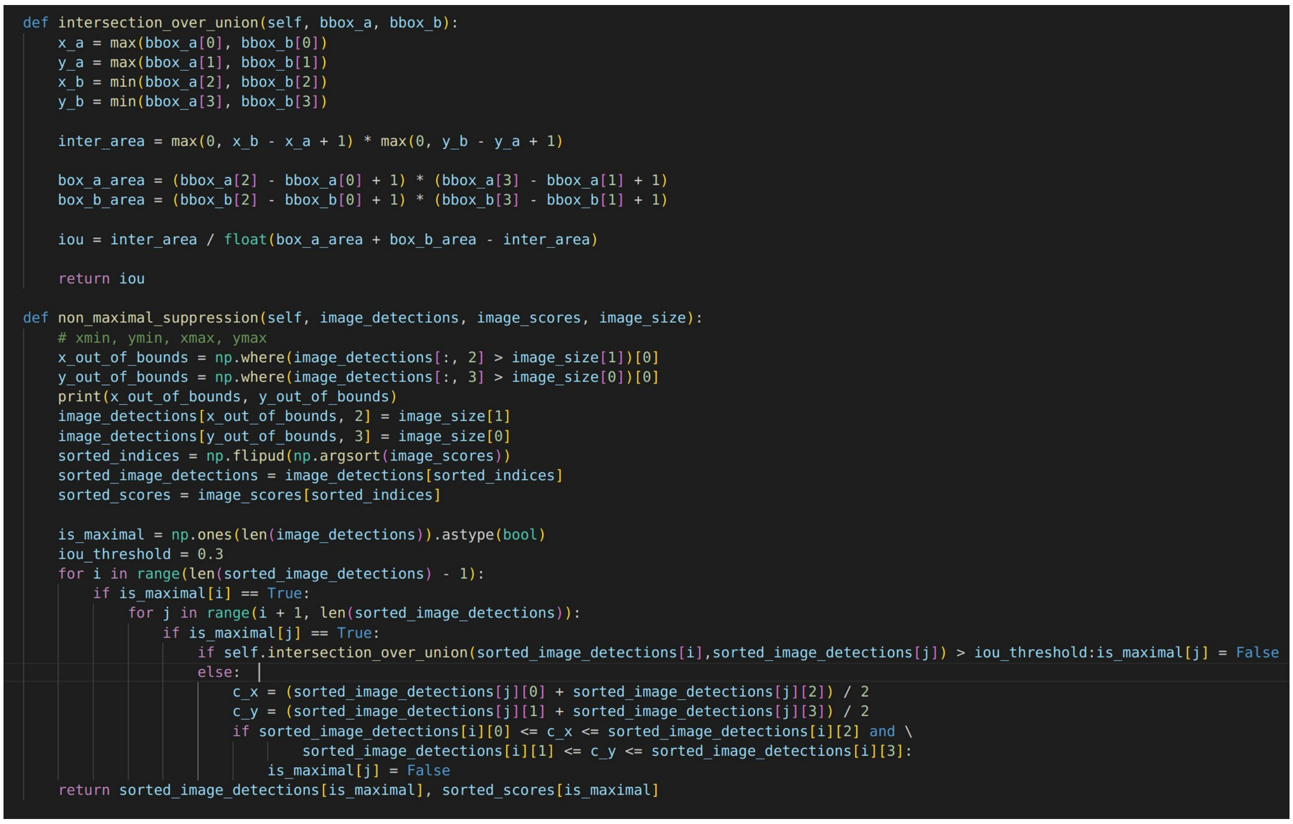

Drawing 5: Window ratio 1:1



```python
for i in range(num_test_images):
    start_time = timeit.default_timer()
    print(f'Process test image {i}/{num_test_images}')
    img = cv.imread(test_files[i], cv.IMREAD_GRAYSCALE)

    image_scores = []
    image_detections = []

    ars = [0.60, 0.75, 0.90, 1, 1.15, 1.25, 1.40, 1.60, 1.70, 1.80, 1.95]
    scales = [0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 1, 1.3, 1.6]

    for ar in ars:
        for scale in scales:

            if ar < 1:
                fx = ar * scale
                fy = scale
            else:
                fx = scale
                fy = (1/ar) * scale

            warped_img = cv.resize(img, dsize=None, fx=fx, fy=fy)

            hog_descriptors = hog(warped_img, pixels_per_cell=(self.params.dim_hog_cell, self.params.dim_hog_cell),
                                       cells_per_block=(2, 2), feature_vector=False)

            num_cols = warped_img.shape[1] // self.params.dim_hog_cell - 1
            num_rows = warped_img.shape[0] // self.params.dim_hog_cell - 1
            num_cell_in_template = self.params.dim_window // self.params.dim_hog_cell - 1

            for y in range(0, num_rows - num_cell_in_template):
                for x in range(0, num_cols - num_cell_in_template):
                    features = hog_descriptors[y:y + num_cell_in_template, x:x + num_cell_in_template].flatten()
                    score = np.dot(features, w)[0] + bias

                    if score > self.params.threshold:
                        x_min = int(x * self.params.dim_hog_cell)
                        y_min = int(y * self.params.dim_hog_cell)
                        x_max = int(x * self.params.dim_hog_cell + self.params.dim_window)
                        y_max = int(y * self.params.dim_hog_cell + self.params.dim_window)

                        x_min, y_min = int(x_min / fx), int(y_min / fy)
                        x_max, y_max = int(x_max / fx), int(y_max / fy)

                        image_detections.append([x_min, y_min, x_max, y_max])
                        image_scores.append(score)
```

Figure 5: Sliding window

After sliding all the windows at specified aspect ratios and scale we have multiple detection for one face that are overlapping. We need to apply non-maximum suppression to get the detection that fits the characters face the best. Also we need to implement intersection over union (IoU). Then we save the remaining detections with their scores and file names.



```python
def intersection_over_union(self, bbox_a, bbox_b):
    x_a = max(bbox_a[0], bbox_b[0])
    y_a = max(bbox_a[1], bbox_b[1])
    x_b = min(bbox_a[2], bbox_b[2])
    y_b = min(bbox_a[3], bbox_b[3])

    inter_area = max(0, x_b - x_a + 1) * max(0, y_b - y_a + 1)

    box_a_area = (bbox_a[2] - bbox_a[0] + 1) * (bbox_a[3] - bbox_a[1] + 1)
    box_b_area = (bbox_b[2] - bbox_b[0] + 1) * (bbox_b[3] - bbox_b[1] + 1)

    iou = inter_area / float(box_a_area + box_b_area - inter_area)

    return iou

def non_maximal_suppression(self, image_detections, image_scores, image_size):
    # xmin, ymin, xmax, ymax
    x_out_of_bounds = np.where(image_detections[:, 2] > image_size[1])[0]
    y_out_of_bounds = np.where(image_detections[:, 3] > image_size[0])[0]
    print(x_out_of_bounds, y_out_of_bounds)
    image_detections[x_out_of_bounds, 2] = image_size[1]
    image_detections[y_out_of_bounds, 3] = image_size[0]
    sorted_indices = np.flipud(np.argsort(image_scores))
    sorted_image_detections = image_detections[sorted_indices]
    sorted_scores = image_scores[sorted_indices]

    is_maximal = np.ones(len(image_detections)).astype(bool)
    iou_threshold = 0.3
    for i in range(len(sorted_image_detections) - 1):
        if is_maximal[i] == True:
            for j in range(i + 1, len(sorted_image_detections)):
                if is_maximal[j] == True:
                    if self.intersection_over_union(sorted_image_detections[i],sorted_image_detections[j]) > iou_threshold:is_maximal[j] = False
                    else: 
                        c_x = (sorted_image_detections[j][0] + sorted_image_detections[j][2]) / 2
                        c_y = (sorted_image_detections[j][1] + sorted_image_detections[j][3]) / 2
                        if sorted_image_detections[i][0] <= c_x <= sorted_image_detections[i][2] and \
                                sorted_image_detections[i][1] <= c_y <= sorted_image_detections[i][3]:
                            is_maximal[j] = False
    return sorted_image_detections[is_maximal], sorted_scores[is_maximal]
```

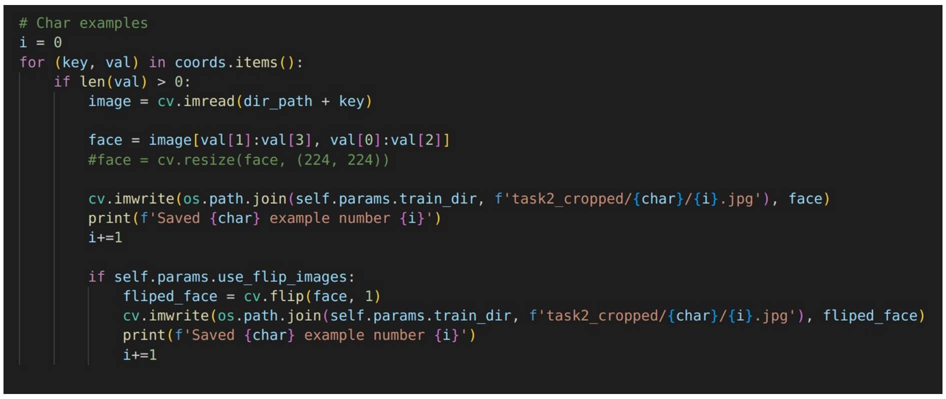
Figure 6: IoU and Non-maximal suppression

4. Recognition task

For the recognition task we need to load the output saved by detection task and iterate through them. We will train a network to classify between the 4 characters and then pass every detection to the model. The output of the model will be a 1x4 tensor that have the probability of each class prediction.

4.1 Data preparation

We take every image from the dataset and crop the faces like we did on detection task, then save them with the original size and aspect ratio. Also we flip the face to augment out data. We save the faces in directories named after the character's name. We do the same process for validation data.



```
# Char examples
i = 0
for (key, val) in coords.items():
    if len(val) > 0:
        image = cv.imread(dir_path + key)

        face = image[val[1]:val[3], val[0]:val[2]]
        #face = cv.resize(face, (224, 224))

        cv.imwrite(os.path.join(self.params.train_dir, f'task2_cropped/{char}/{i}.jpg'), face)
        print(f'Saved {char} example number {i}')
        i+=1

        if self.params.use_flip_images:
            fliped_face = cv.flip(face, 1)
            cv.imwrite(os.path.join(self.params.train_dir, f'task2_cropped/{char}/{i}.jpg'), fliped_face)
            print(f'Saved {char} example number {i}')
            i+=1
```

Figure 7: Cropping the dataset for training

4.2 Convolutional Neural Network

For this task we choose ResNet-18 architecture and implemented it from scratch using Pytorch.

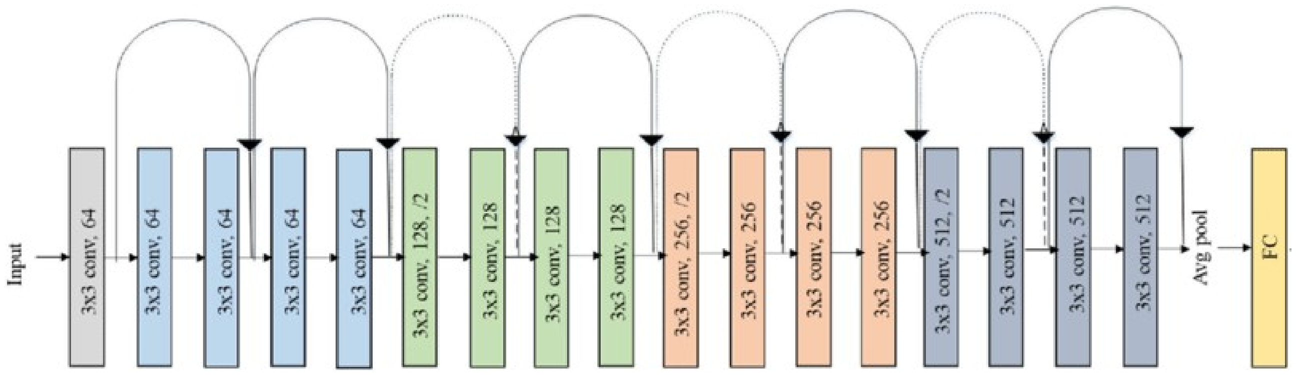
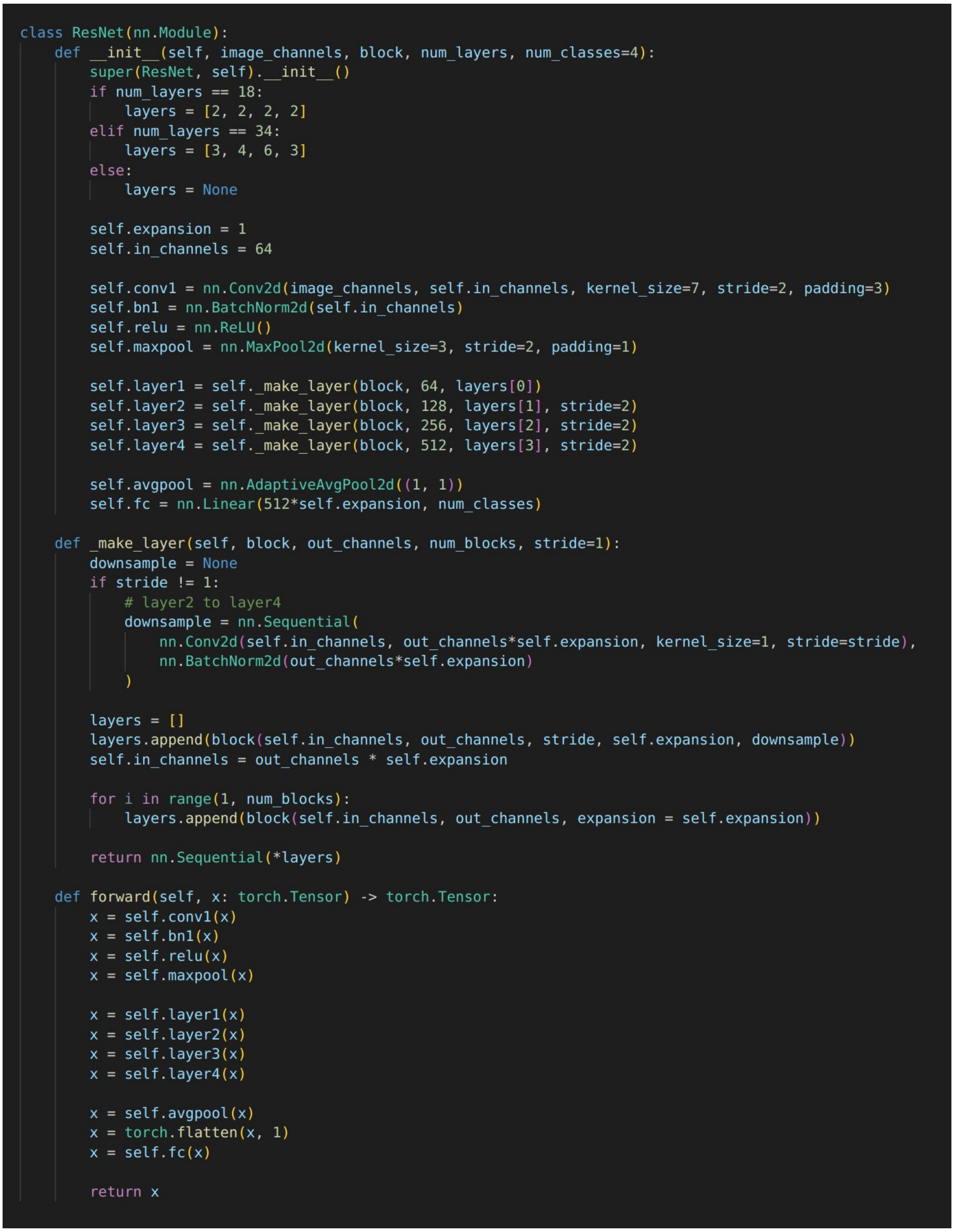


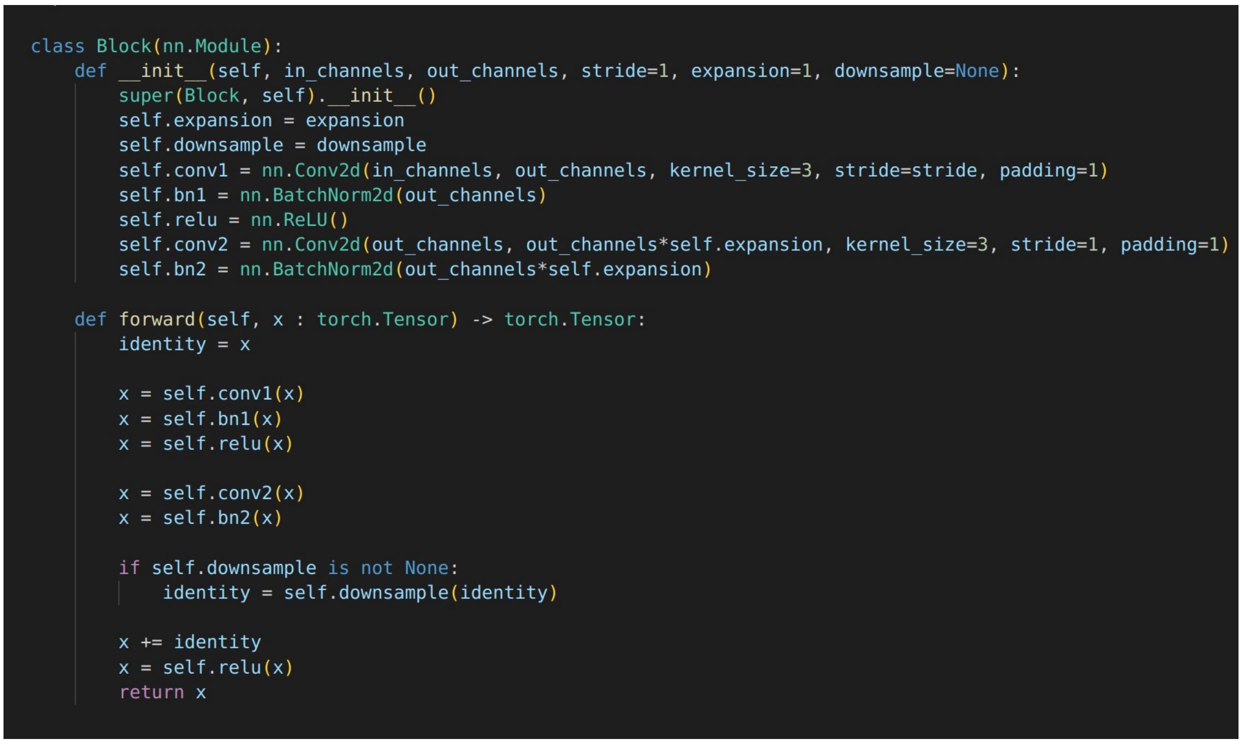
Figure 8: ResNet-18 architecture

We implemented a class that defines the Residual Block that we will use in the network. Then we implemented the main class of the network. Below are some samples from the code.



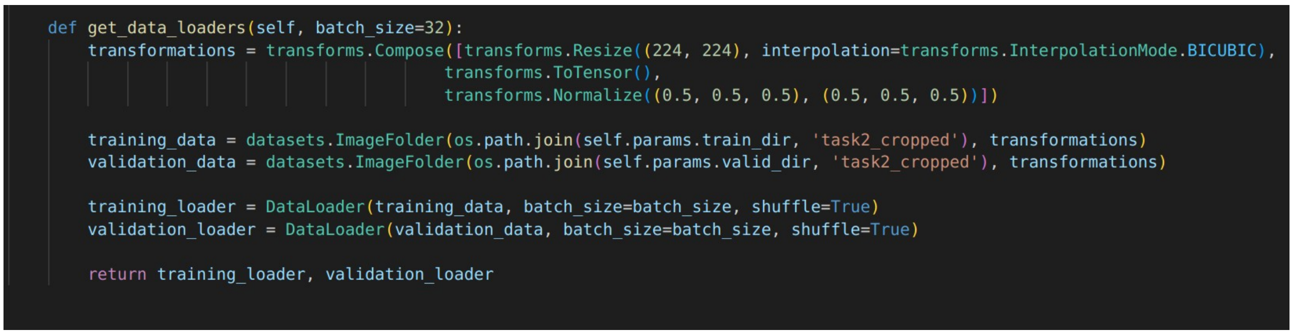
```python
class ResNet(nn.Module):
    def __init__(self, image_channels, block, num_layers, num_classes=4):
        super(ResNet, self).__init__()
        if num_layers == 18:
            layers = [2, 2, 2, 2]
        elif num_layers == 34:
            layers = [3, 4, 6, 3]
        else:
            layers = None

        self.expansion = 1
        self.in_channels = 64

        self.conv1 = nn.Conv2d(image_channels, self.in_channels, kernel_size=7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm2d(self.in_channels)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512*self.expansion, num_classes)

    def _make_layer(self, block, out_channels, num_blocks, stride=1):
        downsample = None
        if stride != 1:
            # layer2 to layer4
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, out_channels*self.expansion, kernel_size=1, stride=stride),
                nn.BatchNorm2d(out_channels*self.expansion)
            )

        layers = []
        layers.append(block(self.in_channels, out_channels, stride, self.expansion, downsample))
        self.in_channels = out_channels * self.expansion

        for i in range(1, num_blocks):
            layers.append(block(self.in_channels, out_channels, expansion = self.expansion))

        return nn.Sequential(*layers)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x
```

Figure 9: ResNet-18


```python
class Block(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, expansion=1, downsample=None):
        super(Block, self).__init__()
        self.expansion = expansion
        self.downsample = downsample
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(out_channels, out_channels*self.expansion, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels*self.expansion)

    def forward(self, x : torch.Tensor) -> torch.Tensor:
        identity = x

        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.conv2(x)
        x = self.bn2(x)

        if self.downsample is not None:
            identity = self.downsample(identity)

        x += identity
        x = self.relu(x)
        return x
```
Figure 10: Residual Block

4.3 Training the network

For the training part we need to prepare the data we just cropped. Figure 11 shows the function used for create the DataLoader objects for training and validation.


```python
def get_data_loaders(self, batch_size=32):
    transformations = transforms.Compose([transforms.Resize((224, 224), interpolation=transforms.InterpolationMode.BICUBIC),
                                          transforms.ToTensor(),
                                          transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

    training_data = datasets.ImageFolder(os.path.join(self.params.train_dir, 'task2_cropped'), transformations)
    validation_data = datasets.ImageFolder(os.path.join(self.params.valid_dir, 'task2_cropped'), transformations)

    training_loader = DataLoader(training_data, batch_size=batch_size, shuffle=True)
    validation_loader = DataLoader(validation_data, batch_size=batch_size, shuffle=True)

    return training_loader, validation_loader
```
Figure 11: Create DataLoaders

We need to apply some transformations of the data. First we resize the faces to 224x224 because this is the input size for the architecture, then convert the images into tensors and apply normalization.

Function 'torchvision.datasets.ImageFolder' takes the path to images and apply the transformation. The classes will be the directories where the images are saved.

The optimizer we chose is Adam and the loss function is Cross Entropy Loss. The network was trained in 50 and 70 epochs and on many combinations of batch_size and learning_rate hyperparameters. The best was epochs=50, batch_size=32 and learning_rate=0.001. The training was done on Nvidia GTX 1050 with 4GB of VRAM. The total number of trainable parameters was about 11mil.

4.4 Training results

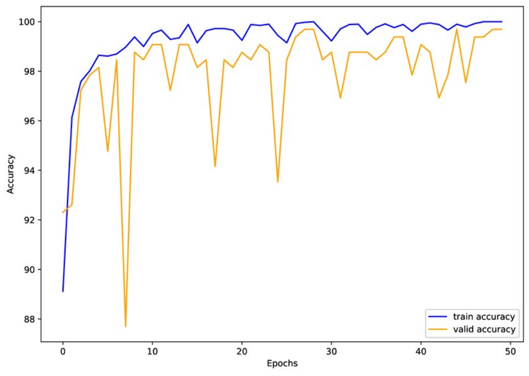


Figure 12: Accuracy

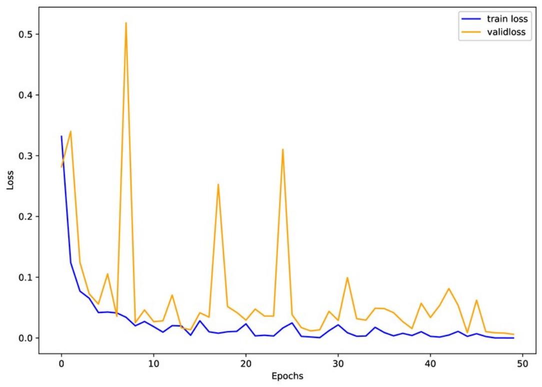
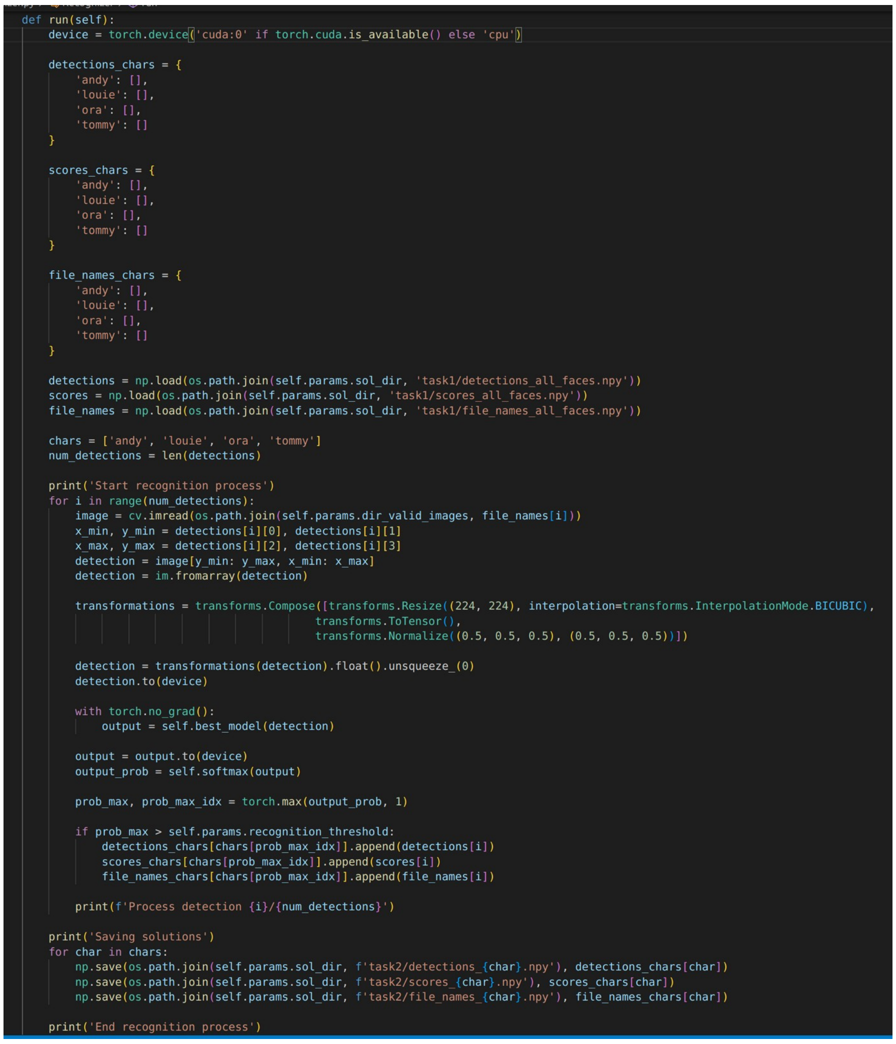


Figure 13: Loss

The best validation loss is 0.008 and accuracy about 98-99%. The training loss is about 0.0009 and accuracy about 99-100%

4.5 Inference



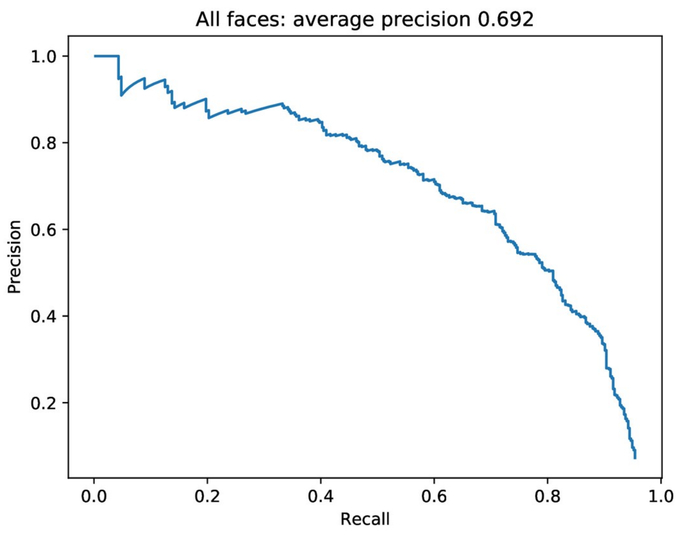
```python
def run(self):
    device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

    detections_chars = {
        'andy': [],
        'louie': [],
        'ora': [],
        'tommy': []
    }

    scores_chars = {
        'andy': [],
        'louie': [],
        'ora': [],
        'tommy': []
    }

    file_names_chars = {
        'andy': [],
        'louie': [],
        'ora': [],
        'tommy': []
    }

    detections = np.load(os.path.join(self.params.sol_dir, 'task1/detections_all_faces.npy'))
    scores = np.load(os.path.join(self.params.sol_dir, 'task1/scores_all_faces.npy'))
    file_names = np.load(os.path.join(self.params.sol_dir, 'task1/file_names_all_faces.npy'))

    chars = ['andy', 'louie', 'ora', 'tommy']
    num_detections = len(detections)

    print('Start recognition process')
    for i in range(num_detections):
        image = cv.imread(os.path.join(self.params.dir_valid_images, file_names[i]))
        x_min, y_min = detections[i][0], detections[i][1]
        x_max, y_max = detections[i][2], detections[i][3]
        detection = image[y_min: y_max, x_min: x_max]
        detection = im.fromarray(detection)

        transformations = transforms.Compose([transforms.Resize((224, 224), interpolation=transforms.InterpolationMode.BICUBIC),
                                              transforms.ToTensor(),
                                              transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

        detection = transformations(detection).float().unsqueeze_(0)
        detection.to(device)

        with torch.no_grad():
            output = self.best_model(detection)

        output = output.to(device)
        output_prob = self.softmax(output)

        prob_max, prob_max_idx = torch.max(output_prob, 1)

        if prob_max > self.params.recognition_threshold:
            detections_chars[chars[prob_max_idx]].append(detections[i])
            scores_chars[chars[prob_max_idx]].append(scores[i])
            file_names_chars[chars[prob_max_idx]].append(file_names[i])

        print(f'Process detection {i}/{num_detections}')

    print('Saving solutions')
    for char in chars:
        np.save(os.path.join(self.params.sol_dir, f'task2/detections_{char}.npy'), detections_chars[char])
        np.save(os.path.join(self.params.sol_dir, f'task2/scores_{char}.npy'), scores_chars[char])
        np.save(os.path.join(self.params.sol_dir, f'task2/file_names_{char}.npy'), file_names_chars[char])

    print('End recognition process')
```

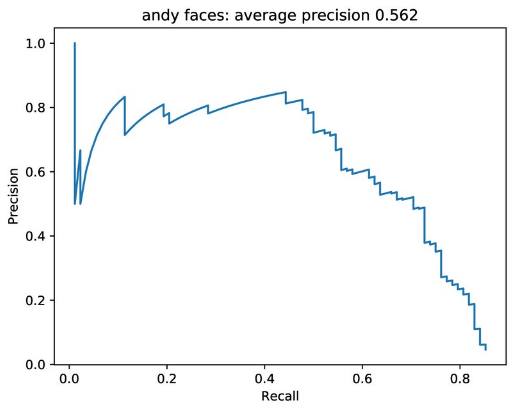
Figure 14: Inference

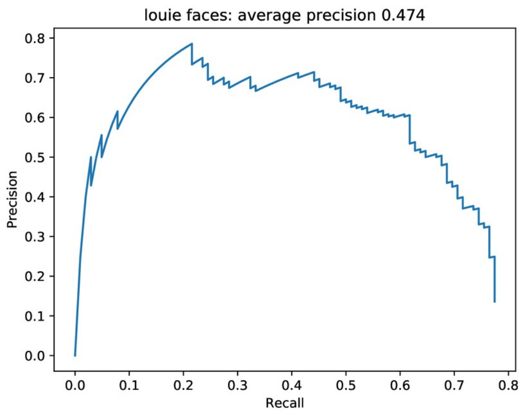
For the inference we load the detections from the task1 output and iterate through them. Apply the same transformations as we did on the training data and give them to the model. The model will ouput a 1x4 tensor with some numbers. Pass the output through softmax function and take the highest probability. If the probability is grater than a certain threshold the we have argmax(softmax(model_output)) class.

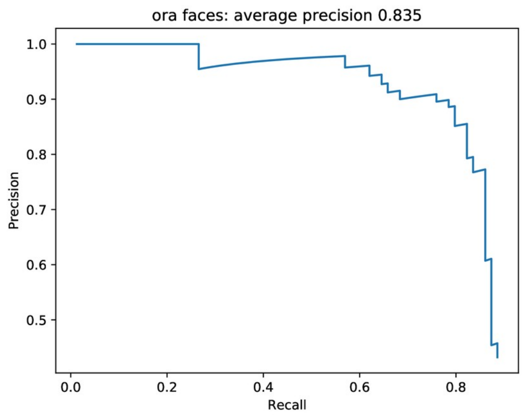
5. Results

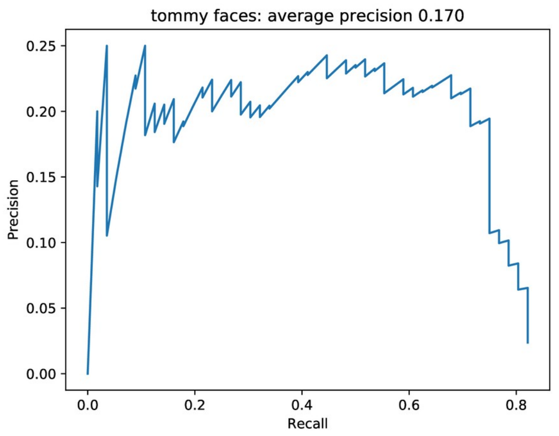
The performance of both models for each task on validation data is shown below.


All faces: average precision 0.692
Precision
Recall


andy faces: average precision 0.562
Precision
Recall


louie faces: average precision 0.474
Precision
Recall


ora faces: average precision 0.835
Precision
Recall


tommy faces: average precision 0.170
Precision
Recall