Machine Learning Final Project Report

Chang Yan (cyan13) and Jingguo Liang (jliang35)

Goal

The goal of this project is to develop deep neural networks that are able to detect human faces in an image and make predictions about this person's age, sex, and race. There are lots of applications for this in real-time cameras.

Deliverables

We successfully finished all of our deliverables. They are listed below:

Must accomplish

1. A program with a trained deep convolutional network with optimized structure and best tuned parameters, that can predict the age of the face in the input image. ***(Fully implemented)***
2. The prediction of the program should be distinctly better than simple benchmark run, like a logistic regression. ***(Fully implemented)***
3. The program should use appropriate data preprocessing mothods to improve the performance. ***(Fully implemented)***

Expect to accomplish

1. The program should be able to predict not only age but also sex and ethnicity of the face. ***(Fully implemented)***
2. The program should use multiple different network structures, ensembles or other methods to further improve the performance. ***(Fully implemented, used stack ensemble of ResNet18, DenseNet121, SqueezeNet1.1)***
3. The prediction of the program should reach an accuracy that is usable in real-life, like over 90%,. ***(Fully achieved for facial detection Faster-RCNN and age prediction in Ensemble, not achieved by age and race prediction.)***
4. Adding feature extraction methods before the network (like edge detection) to create more features from the image. ***(Fully implemented)***

Would like to accomplish

1. The program should be able to first identify the location of the face in the image, and then predict using the cropped face area. The position of the face should also be an output. ***(Fully implemented, very accurate)***
2. The program should be able to take in different sizes of image files. ***(Fully implemented)***
3. The program should be able to identify an image with no face, instead of giving random output. ***(Fully implemented, the model will output a score for each detection. If too low it will notify user)***

4. The program should work with both RGB and grayscale images, and uses the RGB channels to achieve better performance than grayscale. ***(Fully implemented)***
5. The speed of the prediction should be fast enough to be done in real-time. ***(Fully implemented, inference of one picture takes less than 1 second)***

Dataset

The dataset used in the project is UTKFace, which can be found on the following website: https://susanqq.github.io/UTKFace/ (https://susanqq.github.io/UTKFace/)

The raw data consists of four parts:

(1) an "original" RGB image in .jpg format with arbitrary size that contains one human face.

(2) a "cropped" RGB image in .jpg format with size (200, 200) that is part of the image in (1) where the face locates in.

(3) labels for the image, including four parts: age, sex, race of the people in the image, and the date on which the image is collected. The labels are contained in the name of the images, formatted as [age]*[gender]*[race]*[date&time].jpg For cropped images, they are named [age]*[gender]*[race]* [date&time].jpg.chip.jpg. Thus, we can find the corresponding original/cropped images using their labels. [age] is an integer from 0 to 116. [gender] is either 0 (indicating a male) or 1 (indicating a female). [race] is an integer from 0 to 4. 0 denotes White, 1 denotes Black, 2 denotes (East) Asian, 3 denotes Indian, 4 denotes others (e.g. Hispanic, Latino, Middle Eastern) [date&time] is in the format of yyyymmddHHMMSSFFF, indicating the date on which the image is collected to the dataset.

(4) Facial landmarks located in the cropped image

In the project, we will only use the first three parts of the data (original image, cropped image, labels). For the labels, we will only use age, gender and race labels, as data&time is collection time which it irrelevant.

Data Processing

The full data preprocessing scripts can be found in the repo under the folder "DataPreprocessing".

Data cleansing

The first thing we do is to go through all cropped images and remove those which are obvious wrong (clearly not a face). We also removed all black and white images because we want RGB images as input.

Crop Box Generation

Then, we want to locate the cropbox in the original image. We can do that through SIFT and then finding the homography that maps the cropped image to the original image.

In [ ]:
```
# Code is only for demonstration purpose and not executable in Jupyter Notebook.
# Complete, executable script can be found in the deliverables (.py files)

import numpy as np
import cv2 as cv

def extractKeypoints(img1, img2, N):
    sift = cv.SIFT_create()
    keypoint1, descriptor1 = sift.detectAndCompute(img1, None)
    keypoint2, descriptor2 = sift.detectAndCompute(img2, None)

    if keypoint1 is None or keypoint2 is None or len(keypoint1) < N or len(keypoint2) <
        return np.array([]), np.array([])

    bf = cv.BFMatcher_create()
    matches = bf.knnMatch(descriptor1, descriptor2, k = 2)

    good = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            good.append(m)

    matches = sorted(good, key = lambda x:x.distance)[:N]

    points1 = np.float32([keypoint1[m.queryIdx].pt for m in matches])
    points2 = np.float32([keypoint2[m.trainIdx].pt for m in matches])

    return points1, points2

[pts1, pts2] = ExtractKeypoints.extractKeypoints(imgOriginal, imgCropped, 25)
[H, status] = cv.findHomography(pts2, pts1, method = cv.RANSAC)
```

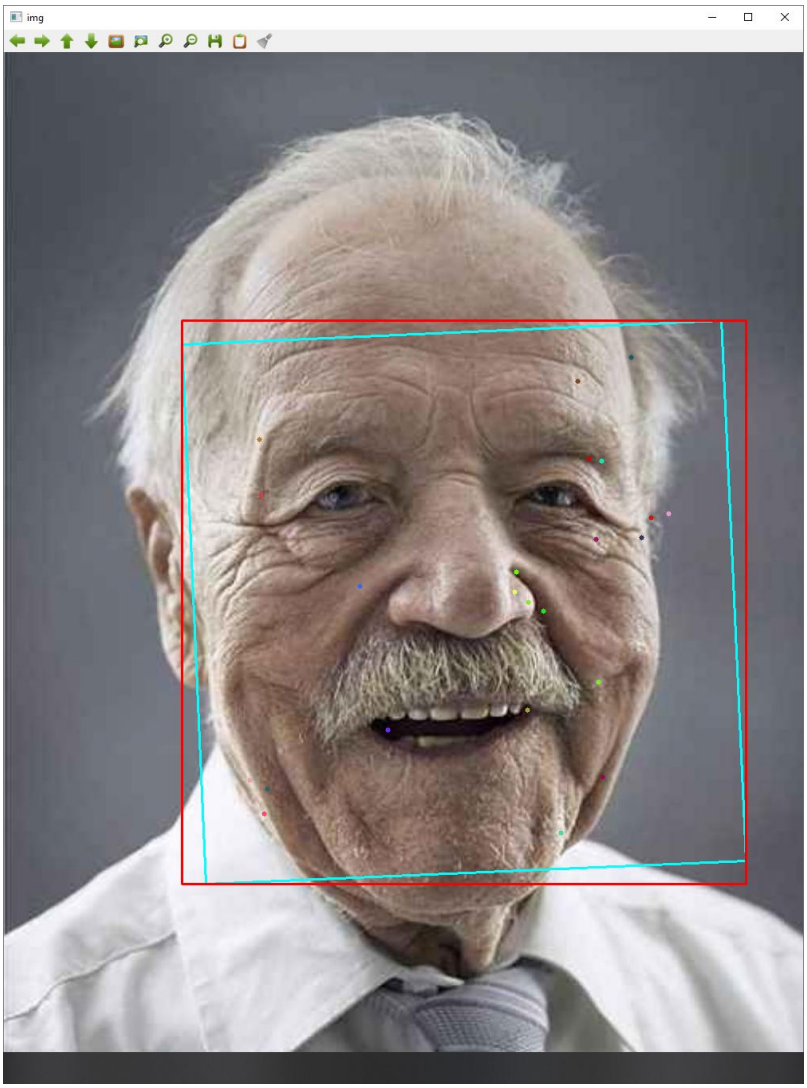
We notice that some images does not generate any feature through SIFT, or the number of features generated is lower than we want (<25). Those images are discarded.

After finding the homography, we can map the four corners of cropbox ([[0, 0], [0, 200], [200, 200], [200, 0]) back to the original image. We notice that after mapping the cropbox back, the sides of the box is not parallel to the side of the image. Thus, we also calculate the circumscribed box of the mapped cropbox.

We also discard the boxes which has a significant angle between the original image, because in that case the circumstribed box will be significantly larger than the cropbox. We can do this by extracting the rotation angle from the homography.

In [ ]:
```
# Code is only for demonstration purpose and not executable in Jupyter Notebook.
# Complete, executable script can be found in the deliverables (.py files)

angle = abs(math.atan2(H[1,0], H[0,0])) * 180 / math.pi
if angle < 5:
    originalShape = imgOriginal.shape
    croppedShape = imgCropped.shape
    croppedPts = np.array([[0, 0, 1], [0, croppedShape[1], 1], [croppedShape[0], croppedSh
    originalPts = np.matmul(H, croppedPts.transpose()).transpose()
    originalPts = np.round(originalPts[:, 0:2] / originalPts[:, 2:3]).astype(np.int32)
    x1 = np.min(originalPts[:, 0]).clip(0, originalShape[1])
    x2 = np.max(originalPts[:, 0]).clip(0, originalShape[1])
    y1 = np.min(originalPts[:, 1]).clip(0, originalShape[0])
    y2 = np.max(originalPts[:, 1]).clip(0, originalShape[0])
```

SIFT



Cropbox

Feature Engineering

We would also like to use the intensity image and the edge map in training. Intensity map can be obtained by a weighted combination of RGB values, and the edge map can be obtained through canny edge detection. We can add the intensity image and the edge map to the RGB image, getting a 5-channel image.

The input to the neural model should be in [N,C,H,W] format, while currently our image is in [H,W,C] format. We can simply swap the axes to get the nchw format that we want.

In [ ]:
```
# Code is only for demonstration purpose and not executable in Jupyter Notebook.
# Complete, executable script can be found in the deliverables (.py files)

class PreprocessImage(object):

    @classmethod
    def preprocess(cls, img, normalize):
        shape = img.shape
        intensity = (0.11 * img[:,:,0] + 0.59 * img[:,:,1] + 0.30 * img[:,:,2])
        intensity = intensity.reshape((shape[0], shape[1], 1))
        edge = cv.Canny(img, 100, 100, apertureSize = 3)
        # cv.imshow('img', edge)
        # cv.waitKey(0)
        edge = edge
        edge = edge.reshape((shape[0], shape[1], 1))
        if normalize is True:
            intensity = intensity / 255.0
            edge = edge / 255.0
            img = img / 255.0
            result = np.append(img, intensity, axis = -1)
            result = np.append(result, edge, axis = -1)
            result = np.moveaxis(result, -1, 0)
            result[[0,2]] = result[[2,0]]
            return result.astype(float)
        else:
            result = np.append(img, intensity, axis = -1)
            result = np.append(result, edge, axis = -1)
            result = np.moveaxis(result, -1, 0)
            result[[0,2]] = result[[2,0]]
            return result.astype(np.uint8)
```

Label Encoding

The original dataset gives ages directly as a number. However, in our model we would like to divide all ages into several categories:
0-3: label 0
4-6: label 1
7-10: label 2
11-15: label 3
16-20: label 4
21-25: label 5
26-30: label 6
31-40: label 7
41-50: label 8
51-60: label 9
61-80: label 10
81+: label 11

Besides age, we also have sex and race in the labels. For sex:
0: male
1: female

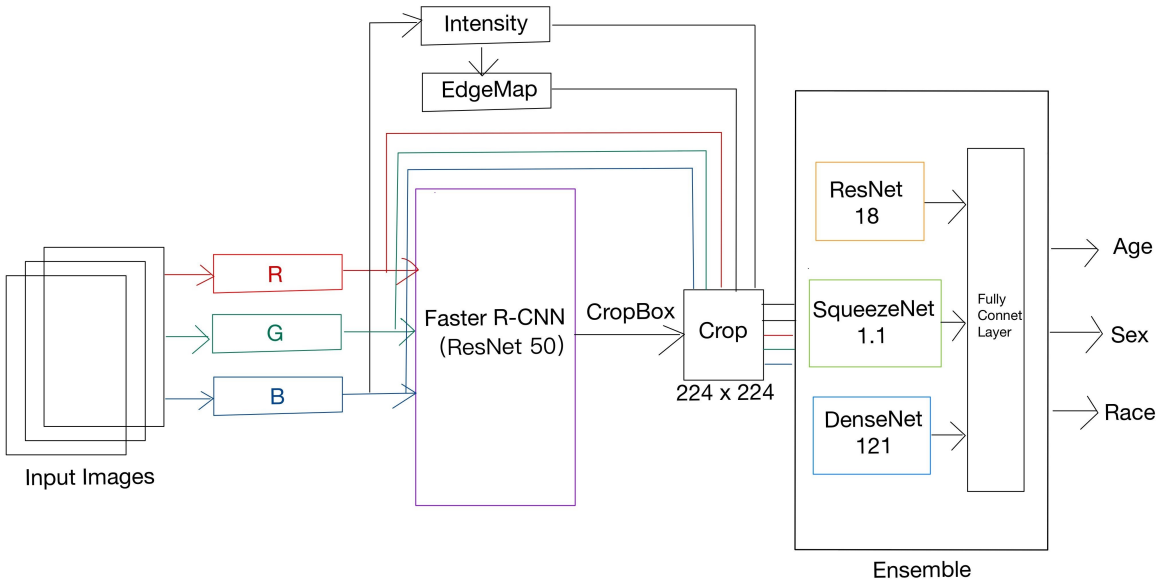
For race:
0: White
1: Black
2: (East) Asian
3: Indian
4: Other

5 fold cross validation

The last thing to do is to divide the data into 5 parts. We will use them for 5-fold cross validation in the training. After the division, we can save the data as .npz files, which will be read back and directly used in training. Each fold contains 1000 cases. In each fold, there will be one single .npz for all cropboxes, one single .npz for all labels and one .npz file for each of the origina l images.

Model and pipeline design

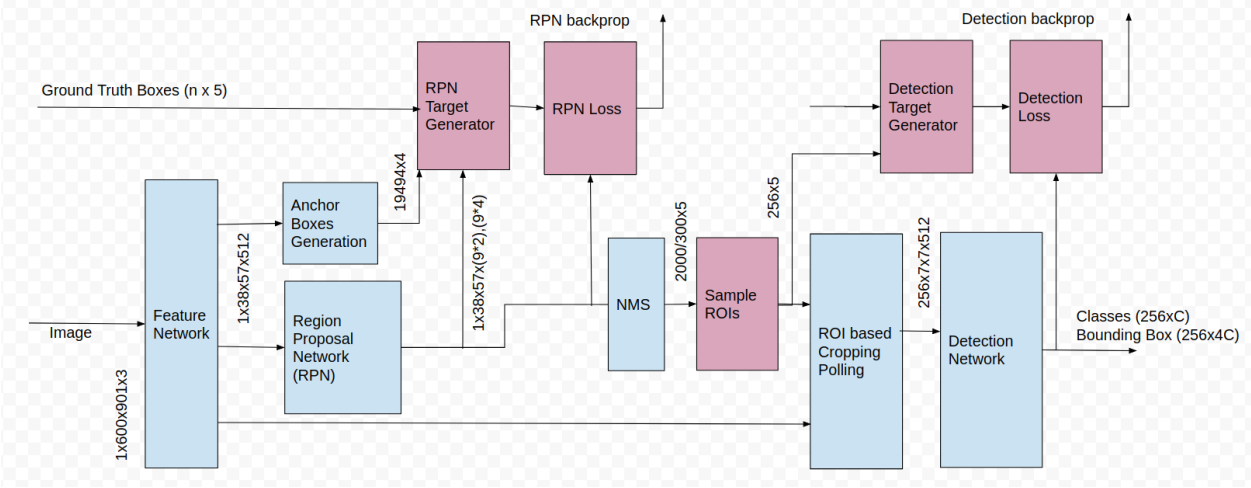
Here is an overview of the design of our program:



The program takes the input of a number of images with different sizes. The image can be either RGB or grayscale, but we will always extract 3 RGB channels from the program. Then, the 3 channels will be sent to feature engineering function and create channel 4 intensity and channel 5 edgemap. Also, the RGB channel images with different sizes will be sent to the Faster R-CNN model, which is a region detection convolution network with ResNet50 as its feature extraction

backbone. We trained the RCNN to predict face region. Then, the network will predict a cropbox of each image, which can be used to crop the face region out of the image. After that, we added the intensity and edgemap channels to it, and resize the 5-channel image to 224 x 224 as it is the sized required by those classification networks. We stacked three image classification networks as our classification ensemble. The ResNet 18, SqueezeNet 1.1 and DenseNet 121. They are all the smallest version of the corresponding networks, and we used ensemble to boost their performance. All of their outputs are connected to the fully connected final layer, and the output will be sent in to a softmax activation layer to predict label probabilities. We trained 3 sets of ensembles to predict age, sex and race as they have different number of classes.

Model Training Part I: Facial Detection: Faster-RCNN



The Faster RCNN is consisted of 3 major networks. The first network is the backbone feature network, where we used a pre-trained ResNet50 model. The second one is the RPN network which is used to generate a sequence of possible bounding boxes, and the Detection network is the actual network to detect the facial region. Both the RPN and the Detection network produce a loss, and they are summed and backpropagated together.

Our model and trainer are the "rcnn.py" and "train_rcnn.py". As we mentioned before, we did 5 fold cross validation on this model, and we recorded the detection score (which is produced by the detection network as a detection confidence score), validation loss and the root mean square error (RMSE) of the predicted boxes. Here is part of our code for model initialization for demonstration ONLY:

In [ ]:
```
# Code is only for demonstration purpose and not executable in Jupyter Notebook.
# Complete, executable script can be found in the deliverables (.py files)

class Rcnn:

    def __init__(self, path, name, num_epochs, batch, pre=True):
        self.model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=pre)
        num_classes = 2
        in_features = self.model.roi_heads.box_predictor.cls_score.in_features
        self.model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
        self.path = os.path.join(path, name)
        self.device = torch.device("cuda:1" if torch.cuda.is_available() else "cpu")
        self.model.to(self.device)
        self.model.float()
        params = [p for p in self.model.parameters() if p.requires_grad]
        self.optimizer = optim.SGD(params, lr=0.005, momentum=0.9, weight_decay=0.0005)
        self.cur_epoch = 0
        self.epochs = num_epochs
        self.loss = np.infty
        self.best_loss = np.infty
        self.start_time = int(time.time())
        self.freq_for_save = 5
        self.batch = batch
```

As we can see from the code above, we replaced the FastRCNNPredictor in the model (which is the detection network) with a new one because the original model was trained in 91 labels but we are only using 2 labels here (Face and Not face). Here is the result status of our 5 fold cross validation: (The evaluation metrics are generated by "eva_rcnn.py")

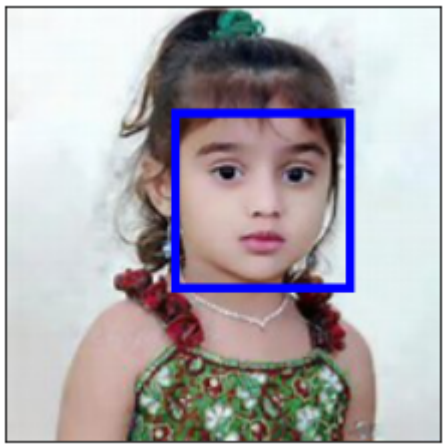
Statistics	Fold 0	Fold 1	Fold 2	Fold 3	Fold 4
Match Score	99.4%	99.4%	99.7%	99.8%	99.1%
Validation loss	0.049	0.047	0.059	0.041	0.048
RMSE	9.76	4.93	5.01	3.52	4.98

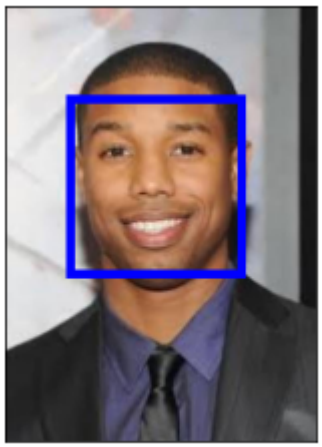
We can see that our match score is extremely high, indicating a good detection by the detection network. The validation loss is also quite slow, so we do not have a over fit problem (it is very close to train loss actually). Also, the RMSE is only around 5 pixels, which is quite small given that our images can be as large as 2000 x 2000 pixels, indicate a relatively accurate detection.

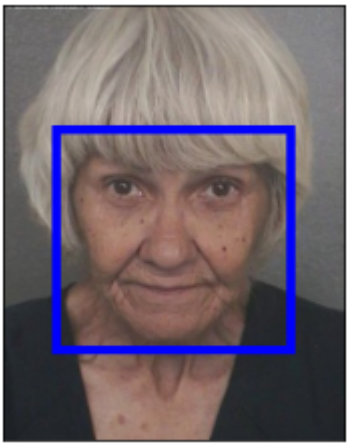
Model Result Visualization: Facial Detection: Faster-RCNN

Here are some examples of the predictions made by the facial detection Faster-RCNN network. The visualization is done using the file "visualization.py". Note that this is a fully working prediction file that can use the trained networks to detect face and prediction labels. We also provide a part of the code here for demonstration ONLY. Full code can be found on visualization.py.

In [ ]:
```
def show(imgs):
    if not isinstance(imgs, list):
        imgs = [imgs]
    fix, axs = plt.subplots(ncols=len(imgs), squeeze=False)
    for i, img in enumerate(imgs):
        img = img.detach()
        img = F.to_pil_image(img)
        axs[0, i].imshow(np.asarray(img))
        axs[0, i].set(xticklabels=[], yticklabels=[], xticks=[], yticks=[])
    plt.savefig("image.jpg")

def main(args):
    image = cv.imread("./Test/9.jpg")
    imageT = read_image("./Test/9.jpg")
    image = PreprocessImage.preprocess(image)
    shape = image.shape

    imageT = F.resize(imageT, [shape[1], shape[2]])
    test_image = image[0:3].astype(float) / 255
    model = Rcnn(args.svpath, "Rcnn_full_feature_fold_4.pt", 50, 1)
    model.load()
    boxes, scores = model.predict([test_image])
    if (boxes[0] is not None):
        result = draw_bounding_boxes(imageT, torch.tensor([boxes[0]], dtype=torch.float),
        show(result)
```





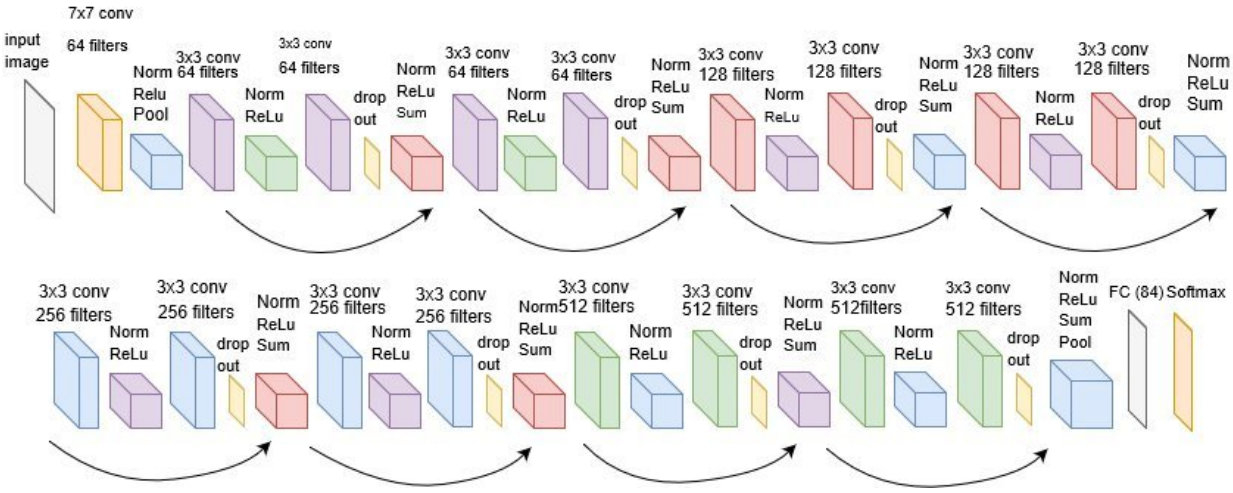


We can see that our predication is very accurate. The networks in each fold capture every face in each test set. We only show 3 random cases here, but all other results look as accurate as those, and the accuracy is quite consistent across the 5 folds. This demonstrates that the facial detection of our network is very reliable.

Model Train Rart II: The Ensemble

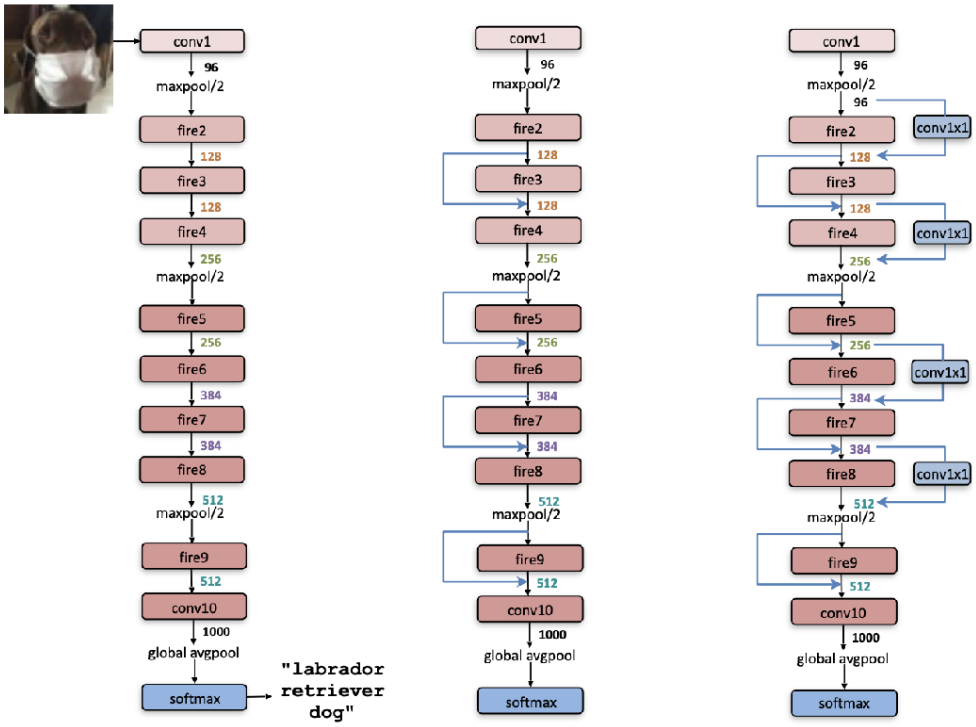
We will discuss the three networks used in the ensemble: ResNet18, SqueezeNet1.1 and DenseNet121. Note that we modified all the first layers of the models to change input channels from 3 to 5, and we also modified the output layer to set the correct number of output feature according to the number of classes we are predicting.

ResNet 18



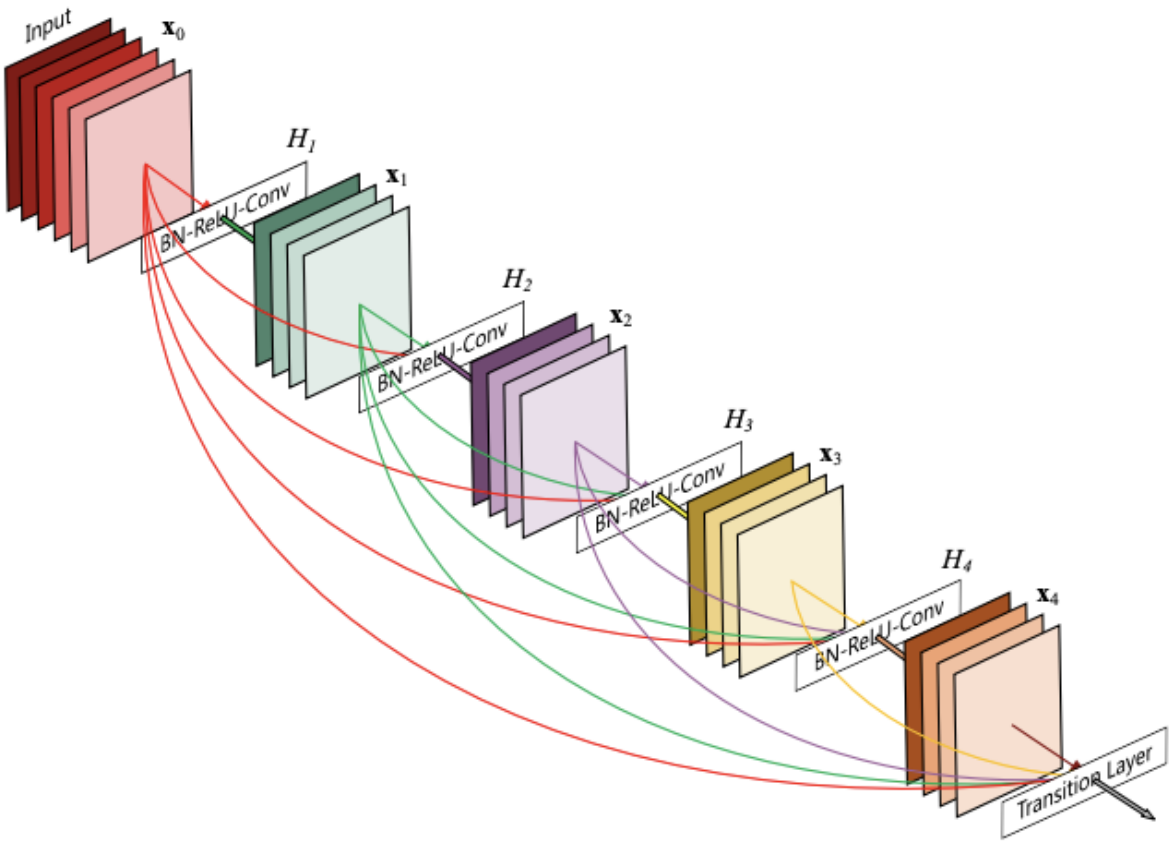
Here is the diagram of the ResNet18 network we were using. It is a very common image classification network consisted of a long series of convolution layers, dropout layers and ReLu activations. ResNet18 is the smallest one of this set of networks.

SqueezeNet 1.1



Here is the diagram of the SqueezeNet 1.1 we were using. It basically added those "squeeze" channels to the model, as we can see in the right most line with those conv1x1 layers. SqueezeNet 1.1 is an updated version for SqueezeNet 1.0 with smaller size and same accuracy. Generally it works not as good as ResNet but it is small and fast to train.

DenseNet 121



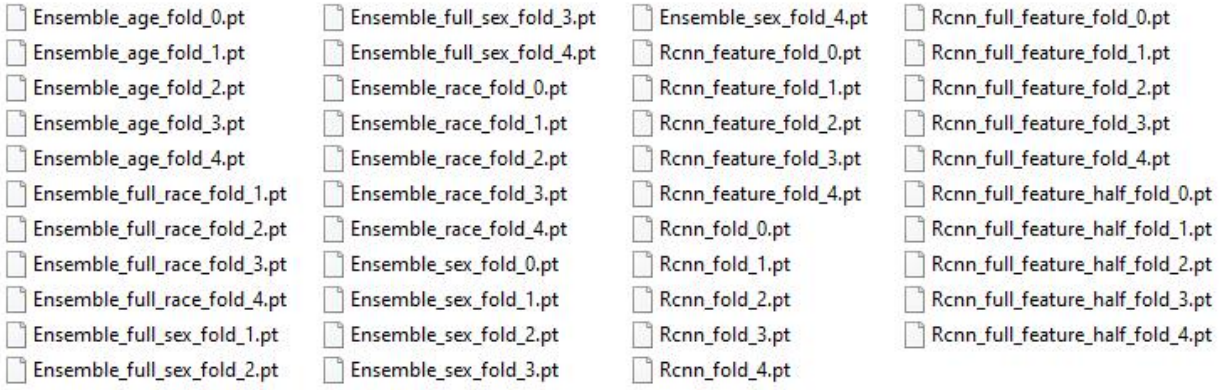
Here is the diagram of the DenseNet 121 model we were using. It is called dense because of those connections between each layers, meaning that every layer will see the predication from other layers, so it is called "DenseNet". We used the smallest version of it. Generally DenseNet works better but needs more time to train.

The file to train the ensemble mode is in "train_ensemble.py", and the actual model is in "ensemble.py". We provide the construction code of the ensemble here for reference ONLY. The full code is in the git repo.

In [ ]:
```
# Code is only for demonstration purpose and not executable in Jupyter Notebook.
# Complete, executable script can be found in the deliverables (.py files)

class Ensemble(nn.Module):
    def __init__(self, num_classes, pre=True):
        super(Ensemble, self).__init__()
        self.model1 = models.resnet18(pretrained=pre)
        weight = self.model1.conv1.weight.clone()
        self.model1.conv1 = nn.Conv2d(5, 64, kernel_size=7, stride=2, padding=3,
                                      bias=False)
        with torch.no_grad():
            self.model1.conv1.weight[:, :3] = weight
            self.model1.conv1.weight[:, 3] = (weight[:,0] + weight[:,1] + weight[:,2]) / 3
            self.model1.conv1.weight[:, 4] = (weight[:,0] + weight[:,1] + weight[:,2]) / 3
        num_ftrs = self.model1.fc.in_features
        self.model1.fc = nn.Linear(num_ftrs, num_classes)

        self.model2 = models.squeezenet1_1(pretrained=pre)
        weight = self.model2.features[0].weight.clone()
        self.model2.features[0] = nn.Conv2d(5, 64, kernel_size=3, stride=2)
        with torch.no_grad():
            self.model2.features[0].weight[:, :3] = weight
            self.model2.features[0].weight[:, 3] = (weight[:,0] + weight[:,1] + weight[:,2
            self.model2.features[0].weight[:, 4] = (weight[:,0] + weight[:,1] + weight[:,2
        self.model2.classifier[1] = nn.Conv2d(512, num_classes, kernel_size=(1,1), stride=
        self.model2.num_classes = num_classes

        self.model3 = models.densenet121(pretrained=pre)
        weight = self.model3.features.conv0.weight.clone()
        self.model3.features.conv0 = nn.Conv2d(5, 64, kernel_size=7, stride=2,
                                               padding=3, bias=False)
        with torch.no_grad():
            self.model3.features.conv0.weight[:, :3] = weight
            self.model3.features.conv0.weight[:, 3] = (weight[:,0] + weight[:,1] + weight[
            self.model3.features.conv0.weight[:, 4] = (weight[:,0] + weight[:,1] + weight[
        num_ftrs =  self.model3.classifier.in_features
        self.model3.classifier = nn.Linear(num_ftrs, num_classes)

        self.output = nn.Sequential(
            nn.Linear(num_classes * 3, num_classes),
            nn.Softmax(dim = 1)
        )

    def forward(self, x):
        x1 = self.model1(x)
        x2 = self.model2(x)
        x3 = self.model3(x)
        x = torch.cat((torch.cat((x1,x2),1),x3),1)
        x = self.output(x)
        return x
```

We trained over 40 different models. They are for different folds and with different configurations. The model including "full" words means that the full model is trainable, while for others only the ensemble layers are trainable. The model including "feature" means that we used pre-trained

weights for the feature extraction network. This is called transfer learning. The word "half" means that we tried to only use half the data to see the performance of the network. We found that transfer learning increase training speed but sometime also cause the loss to increase, and setting the full model to be trainable greatly improves the performance of Faster-RCNN, but only training the ensemble layer is best for the ensemble model. We also found that reducing the training data for Faster-RCNN by half do not change its performance on test case, and it still reaches over 99% accuracy even with only 50 pictures in one fold. As each of the models are larger than 500M, it is impossible to put them in git, but I provided the screen shot here to prove our work.



Benckmark result

Traget, Benchmark	NetsResNet 18	SqueezeNet 1.1	Densenet-121
Sex	0.71	0.62	0.72
Race	0.47	0.44	0.48
Age	0.21	0.12	0.19

This is a simple benchmark run using the there individual models. The values are absolute accuracy (Acc@1), meaning that it is 1 when the label classification is exactly right, and 0 when any other predictions. Absolute accuracy is almost always much lower than accuracy calculated using predicted possibilities, but it indicates the True prediction of the net, so we are using it here. We can see Sex prediction is much better than Race, and Race prediction is much better than Age. Also, we can se DenseNet and ResNet perform better than SqueezeNet.

Training result and 5 fold cross validation

Traget, Fold	0	1	2	3	4
Sex	0.908	0.878	0.863	0.881	0.881
Race	0.750	0.638	0.626	0.630	0.638
Age	0.284	0.261	0.266	0.343	0.307

Again, we are using absolute accuracy here. We can see that the Sex prediction reaches really good

accuracy, and Race is a little worse but a absolution accuracy of 0.75 is acceptable for a network prediction given that some race (Asia and Indian) maybe hard to distinguish. (Our investigation also shows that the network better separates races that are far away.) For age, the prediction is expected to be low as our dataset is not balanced in age, and the age predication is really a hard task where most models cannot reach high absolute accuracy. Our dataset has more 20-30 images than other classes, so the prediction is a little biased.

Also, aside from the accuracy themselves, we can see that in all folds, our ensemble predicts much better than the benchmark models, meaning that our ensemble method does increase the accuracy greatly.

Model Result Visualization: Label Prediction: The Ensemble

Here are some examples of the predictions made by the ensemble classification network. The visualization is done using the file "visualization.py". Note that this is a fully working prediction file that can use the trained networks to detect face and prediction labels. We also provide a part of the code here for demonstration ONLY. Full code can be found on visualization.py.

In [ ]:
```
# Code is only for demonstration purpose and not executable in Jupyter Notebook.
# Complete, executable script can be found in the deliverables (.py files)

def main(args)
    imageCropped = cv.imread('./Test/cropped3.jpg')
    imageCropped = PreprocessImage.preprocess2(imageCropped).reshape((1, 5, 224, 224))
    model = EnsembleWrapper(args.svpath, "Ensemble_age_fold_0.pt", 12, 50, 1)
    model.load()
    age = model.predict(imageCropped)
    print(age)
    model = EnsembleWrapper(args.svpath, "Ensemble_full_sex_fold_4.pt", 2, 50, 1)
    model.load()
    sex = model.predict(imageCropped)
    print(sex)
    model = EnsembleWrapper(args.svpath, "Ensemble_full_race_fold_4.pt", 5, 50, 1)
    model.load()
    race = model.predict(imageCropped)
    print(race)
```

Age: [6] (26-30)
Sex: [0] (Male)

Race: [1] (Black)

Age: [10] (61-80)
Sex: [0] (Male)
Race: [0] (White)

Age: [6] (26-30)
Sex: [1] (Female)
Race: [3] (Indian)

We can see that both the prediction of sex and race are quite accurate, but for age the third one is a bit higher than it should be. We believe that there are three reasons for this: 1. The original labels of age may are not accurate, as we actually found some wrongly labeled ones and removed them, but there should still be some missing. 2. Age is very hard to predict even by human eyes, causing difficulties for the network (We also did not use the largest version of each model because of training time). 3. The labels are imbalanced. 26-30 actually is the most common label and the network is a bit biased to it.

Conlusion and Furture Steps

In this project, we have successfully trained a facial detection network with over 99% accuracy, which is very high. We also have very low validation loss meaning that there is no over fit problem. We also successfully trained an ensemble classification network which predicts age with about 88% accuracy, race with about 70% accuracy and age with about 30% accuracy. In the future we can possibly try a better structure to try to classify age accurately.

Our model inference speed is really fast (less than 1 second), which is useable in real-life applications. Our model also takes images with any sizes with out the need of cropping by the user, and the normalization is also automatic.

We trained our models on two 24GB GPUs, and spent over 100 hours total in training.

In [ ]: