**CS 2770: Homework 2**

**Due:** 3/18/2021, 11:59pm  
  
This homework assignment consists of four parts, and is worth 100 points. First, you will extract features from a pretrained classification network and train a support vector machine which discriminates between 20 object categories, using these features. Second, you will train a network (with weights initialized from the same pre-trained network) and train it on this task. You will compare the performance of the pre-trained network to the network you trained. Third, you will train two object detection networks, and compute intersection over union scores to evaluate their performance. Fourth, you will go through a tutorial for Facebook AI Research's (FAIR's) Detectron framework and use it to output results on images of your choice.  
  
You will use PyTorch package, an open source machine learning library based on the Torch library, primarily developed by FAIR. You will do the assignment using Google's Colab service ([introduction](https://colab.research.google.com/notebooks/intro.ipynb)). Your code should be included in a Jupyter notebook and will be run on the cloud, where you have access to GPUs for free with your Google account. To make a new notebook, go to File -> New notebook in the above link. Then add both code and text snippets; use text snippets to explain what your code does, and state which part this snippet is implementing. Submit a zip file with your notebook and any requested outputs (images).  
  
Turn on the GPU mode in Edit -> Notebook settings and set Hardware accelerator to GPU. You can print("GPU Model: %s" % torch.cuda.get\_device\_name(0)) to see what type of GPU you are assigned; e.g. it may be a Tesla T4. Training the CNN in this assignment may take a long time, so be sure to start this assignment early.  
  
The datasets and files you need can be found at the following links:  
PASCAL classification: [link](http://people.cs.pitt.edu/~nhonarvar/TA_Spring_2020/hw2_data.zip)  
PASCAL detection: [link](http://people.cs.pitt.edu/~nhonarvar/TA_Spring_2020/PASCAL.zip)  
Pedestrian detection: [link](http://people.cs.pitt.edu/~nhonarvar/TA_Spring_2020/PennFudanPed_hw3.zip)  
Related files: [link](http://people.cs.pitt.edu/~nhonarvar/TA_Spring_2020/Requied_Files.zip)  
  
Please post on Piazza if you find issues with this assignment.

Part A: Loading and Using a Pretrained Network as a Feature Extractor (30 points)

1. [2 pts] Import required modules and libraries:  
   import torch  
   import torch.nn as nn  
   import torch.optim as optim  
   from torch.optim import lr\_scheduler  
   import numpy as np  
   import torchvision  
   from torchvision import datasets, models, transforms  
   import time  
   import os  
   import copy  
   from sklearn import svm  
   from sklearn.metrics import accuracy\_score
2. [2 pts] Download the data to your home directory and preprocess the data:  
   data\_transforms = {  
    'train': transforms.Compose([  
     transforms.Resize((224,224)),  
     transforms.ToTensor(),  
     transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])  
     ]),  
    'val': transforms.Compose([  
     transforms.Resize((224,224)),  
     transforms.ToTensor(),  
     transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])  
     ]),  
    'test': transforms.Compose([  
     transforms.Resize((224,224)),  
     transforms.ToTensor(),  
     transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])  
     ]),  
   }  
     
   where transforms.Resize((224,224)) is for resizing all images to the same size, transforms.ToTensor() converts the input to tensor with values in the range [0,1], and transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]) normalizes the tensor values based on the mean and standard deviation for RGB values. [0.485, 0.456, 0.406] contains the mean values for Red, Green and Blue channels, respectively. [0.229, 0.224, 0.225] contains the standard deviation for Red, Green and Blue channels, respectively.
3. [2 pts] Create a data loader as follows:  
   data\_dir = 'hw2\_data'  
   image\_datasets = {x: datasets.ImageFolder(os.path.join(data\_dir, x), data\_transforms[x])  
     for x in ['train', 'val', 'test']}  
   dataloaders = {x: torch.utils.data.DataLoader(image\_datasets[x], batch\_size=8, shuffle=True, num\_workers=4)  
     for x in ['train', 'val' , 'test']}  
   dataset\_sizes = {x: len(image\_datasets[x]) for x in ['train', 'val', 'test']}  
   class\_names = image\_datasets['train'].classes  
     
   where data\_dir is a directory which has been created after downloading and unzipping hw2\_data.zip, and contains the data for test, train and validation sets,  
   image\_datasets keeps the path to all images in train, val and test directories,  
   dataloaders receives the image\_datasets,   batch\_size, shuffle and num\_workers as input and returns the data loader for train, validation and test sets,  
   batch\_size specifies the size of mini-batch in every forward pass to the model,  
   num\_workers specifies how many subprocesses to use for data loading, and  
   shuffle specifies whether you want to shuffle the original order of images or not.
4. [4 pts] Load a pretrained CNN model. The model that we are loading has been trained on 14M images from the ImageNet dataset, to classify them into 1000 classes (which aren't the same as the categories we aim to classify). To use the pretrained model as feature extractor, you need to create the following class:  
   class VGG16\_Feature\_Extraction(torch.nn.Module):  
    def \_\_init\_\_(self):  
     super(VGG16\_Feature\_Extraction, self).\_\_init\_\_()  
     VGG16\_Pretrained = models.vgg16(pretrained=True)  
     self.features = VGG16\_Pretrained.features  
     self.avgpool = VGG16\_Pretrained.avgpool  
     self.feature\_extractor = nn.Sequential(\*[VGG16\_Pretrained.classifier[i] for i in range(6)])  
    def forward(self, x):  
     x = self.features(x)  
     x = self.avgpool(x)  
     x = torch.flatten(x, 1)  
     x = self.feature\_extractor(x)  
     return x  
     
   This class is of type torch.nn.Module. In the initialization, first we load the pretrained VGG16 model and then copy the features and avgpool modules. The features contains the convolutional and pooling layers. (You can find PyTorch implementation of VGG16 in [this](https://github.com/pytorch/vision/blob/master/torchvision/models/vgg.py) link). For our feature\_extractor, we copy all the layers except the last fully connected layer from classifier of VGG16. (Note: the last fully connected layer from VGG16 is for classification on 1000 images and we do not need to have it as a part of our feature extractor). In the forward section of model, we first use the features module and then we apply avgpool. Before sending the result to the feature\_extractor we need to flatten the data. Next, we use the feature\_extractor to extract features.  
     
   Finally, you must use the class of VGG16\_Feature\_Extraction(torch.nn.Module) to extract the features for all images. You need to create an instance from the VGG16\_Feature\_Extraction and transfer it to the cuda device:  
   model = VGG16\_Feature\_Extraction()  
   device = 'cuda:0'  
   model = model.to(device)
5. [4 pts] Use the model to extract features of images. You can extract and save the features in different ways and here is the code of one way to do it:  
   image\_features = {}  
   image\_labels = {}  
   for phase in ['train', 'test']:  
    for inputs, labels in dataloaders[phase]:  
     inputs = inputs.to(device)  
     model\_prediction = model(inputs)  
     model\_prediction\_numpy = model\_prediction.cpu().detach().numpy()  
     if (phase not in image\_features):  
      image\_features[phase] = model\_prediction\_numpy  
      image\_labels[phase] = labels.numpy()  
     else:  
      image\_features[phase] = np.concatenate((image\_features[phase], model\_prediction\_numpy), axis=0)  
      image\_labels[phase] = np.concatenate((image\_labels[phase], labels.numpy()), axis=0)  
     
   In this code, first we create dictionaries for image features and image labels for both test and train sets.  
   Then we need to go through both train and test set, use the dataloaders which we have prepared before and extract features for every mini batch by this model\_prediction = model(inputs) command. Since we want to use these features to train a SVM classifier, our features and labels must be numpy arrays. The output of model predictions are tensor on CUDA device and we need to transfer them to numpy array. The code model\_prediction\_numpy = model\_prediction.cpu().detach().numpy() converts the tensors to NumPy arrays.  
   In the last step, we save the predictions. There are two main methods for saving the images: 1) you can concatenate features and labels in every step, or 2) create a 2d array for features and labels in both test and train set. The size of array for features representation is n\*4096 in which n is number of images and 4096 is size of extracted feature. (Note: The second approach is more efficient bacause it does not need concatenation in every step)
6. [8 pts] After retrieving features from the pre-trained VGG16 network, train a linear SVM using SKLearn's [LinearSVC](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) function on the train set but do not train on the withheld validation set or test set. You need to standardize the train set and test set before training and testing your SVM. You can use the [sklearn.preprocessing.StandardScaler](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler) to do this.
7. [8 pts] Test your SVM on the test set (remember to standardize test features using the train mean and standard deviation first) and report the accuracy of the SVM at predicting the class (i.e. the folder that the image was in). Also include a confusion matrix of the predictions using the [sklearn.metrics.confusion\_matrix](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html) function and include it in your submission. Include a brief description of what you observe about the types of errors the network makes, as a text snippet in your notebook.

Part B: Train and Test the CNN on Our Dataset (30 points)

**Preparing the network:** [2 pts]

1. In this step instead of using the VGG16 as a feature extractor, you will train it on our dataset (PASCAL, for a classification task). To do so, first you need to load the VGG16 with pretrained weight from ImageNet.  
   model = models.vgg16(pretrained=True)
2. Then you need to extract the number of input features for the last fully connected layer of model:  
   num\_ftrs = model.classifier[6].in\_features
3. At the end, you need to replace the last fully connected layer with a new layer. This new layer has the same number of input features as the original network but the number of outputs are the same as the number of classes in our dataset:  
   model.classifier[6] = nn.Linear(num\_ftrs, len(class\_names))

**Steps before starting training:** [2 pts]

1. Set the number of epochs to 25.  
   num\_epochs = 25
2. Send the model to CUDA device:  
   model = model.to(device)
3. Specify the criterion for evaluating the trained model:  
   criterion = nn.CrossEntropyLoss()
4. Set the optimizer, learning rate and momentum:  
   optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
5. At the end, create a scheduler to control the way that learning rate changes during the training process:  
   scheduler = lr\_scheduler.StepLR(optimizer, step\_size=7, gamma=0.1)

**Training:** [10 pts]

1. Before starting to iterate over epochs, we need to save the initial model weight as the best model weight and set the best accuracy as zero.  
   best\_model\_wts = copy.deepcopy(model.state\_dict())  
   best\_acc = 0.0
2. Now we can start to iterate over the epochs.  
   for epoch in range(num\_epochs):
3. In the next step, we need to iterate over the train and validation sets. (Note: In every epoch once you need to go through the train set for training the model parameters and then you need to go through the validation set to evaluate the trained model.)  
   for phase in ['train', 'val']:  
    if phase == 'train':  
     model.train()  
    else:  
     model.eval()
4. In the next step we need to go through the data by using the dataloader which we have prepared in previous steps. In every iteraion, we get a minibatch of images and their corresponding labels.

for inputs, labels in dataloaders[phase]:  
 inputs = inputs.to(device)  
 labels = labels.to(device)

1. Before staring to use the model for predicting a mini batch, we need to initialize the gradient vector to all zeros.  
   optimizer.zero\_grad()
2. Now we need to use the current model weight for prediction and backpropagating the prediction loss.  
   with torch.set\_grad\_enabled(phase == 'train'):  
    outputs = model(inputs)  
    \_, preds = torch.max(outputs, 1)  
    loss = criterion(outputs, labels)  
    if phase == 'train':  
     loss.backward()  
     optimizer.step()  
   all\_batchs\_loss += loss.item() \* inputs.size(0)  
   all\_batchs\_corrects += torch.sum(preds == labels.data)  
   In the first line we use this code with torch.set\_grad\_enabled(phase == 'train')to enable gradient calculation in train phase.  
   Then we use the model to predict the classes of every minibatch and compute the loss.  
   If we are in training, we need to send the loss backward to network and update the optimizer.

At the end we need to sum the loss and number of correctly predicted values over all batchs.

1. After iteraring over all minibatchs and if we are in training phase, we need to run scheduler.step() to update the scheduler status as follows:  
   if phase == 'train':  
    scheduler.step()
2. In the next step we compute the loss and accuracy of the epoch.  
   epoch\_loss = all\_batchs\_loss / dataset\_sizes[phase]  
   epoch\_acc = all\_batchs\_corrects.double() / dataset\_sizes[phase]
3. At the end if we are in validation set, we check whether the accuracy of classification is better than the best accuracy so far to save the best model parameters.  
   if phase == 'val' and epoch\_acc > best\_acc:  
    best\_acc = epoch\_acc  
    best\_model\_wts = copy.deepcopy(model.state\_dict())  
    torch.save(best\_model\_wts , 'best\_model\_weight.pth')

**Testing:** [4 pts]

1. The testing process is very similar to train process except that we do not need to backpropagate the loss. For testing the model, first we need to prepare the model in the same way that we prepared for training process and load the best model weight that we saved in training process.  
   model = models.vgg16()  
   num\_ftrs = model.classifier[6].in\_features  
   model.fc = nn.Linear(num\_ftrs, 20)  
   model = model.to(device)  
   model.load\_state\_dict(torch.load('best\_model\_weight.pth'))
2. After loading the model weight, we need to set the model to eval and the value of phase to 'test'.

model.eval()  
phase = 'test'

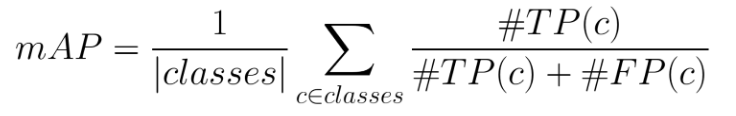
1. In the next step, we need to go through test set, predict the category of images, and compute number of correctly classified images.  
   for inputs, labels in dataloaders[phase]:  
    inputs = inputs.to(device)  
    labels = labels.to(device)  
    outputs = model(inputs)  
    \_, preds = torch.max(outputs, 1)  
    all\_batchs\_corrects += torch.sum(preds == labels.data)
2. At the end, we compute the accuracy over all data.  
   epoch\_acc = all\_batchs\_corrects.double() / dataset\_sizes[phase]

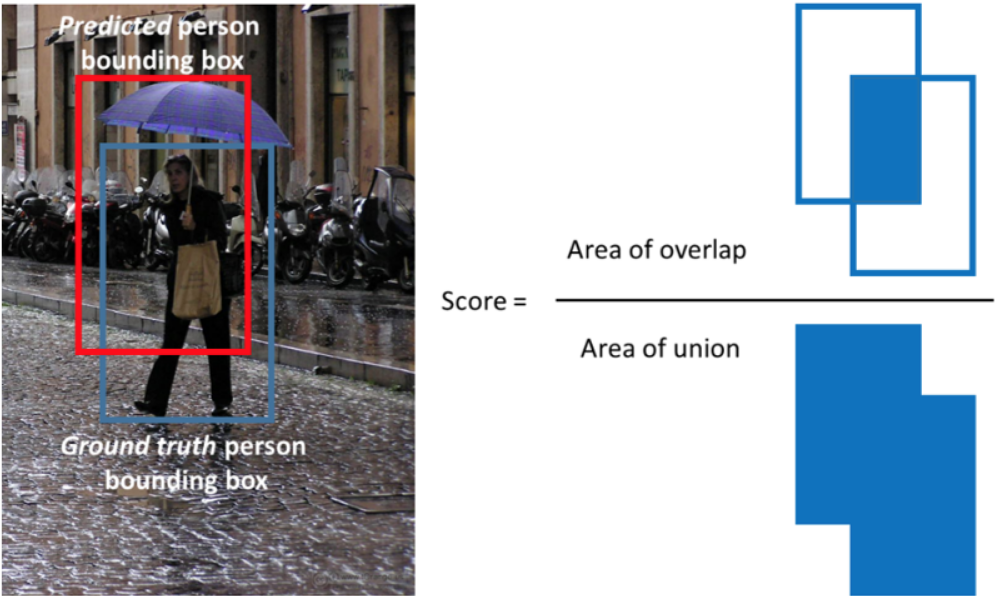
**Repeating with different hyperparameters:** [12 pts]

1. Retrain your model with different hyperparameters / implementation details and include the confusion matrix and accuracy of the model on the test data. For example, you can play with learning rate, batch size, choice of optimizer, regularization, etc. You need to experiment with at least three different hyperparameters and two settings for each.

Part C: Object Detection (Faster RCNN) Training and Evaluation (30 points)

The goal of this part of the assignment is to make you familiar with a recent object detection methods. You will work on two datasets to explore the factors affecting performance. You will be using a Faster RCNN model pretrained on the COCO dataset. Since the datasets in this homework (PASCAL and pedestrians) are different from the COCO, you need to further train the model to fine-tune it on our datasets. One of the datasets contains 5 categories of objects and the other one is a dataset of pedestrians. You will separately train networks for each of these datasets, and evaluate their performance.

1. Computing the performance of object detection is more complicated compared to object classification. The output of object detection are bounding boxes and to compute the performance we use mAP (mean Average Precision). The definition of mAP is as follows:  
          where
   * True Positive - TP(c): a predicted bounding box (pred\_bb) was made for class c, there is a ground truth bounding box (gt\_bb) of class c, and IoU(pred\_bb, gt\_bb) >= threshold.
   * False Positive - FP(c): a pred\_bb was made for class c, and there is no gt\_bb of class c. Or there is a gt\_bb of class c, but IoU(pred\_bb, gt\_bb) < threshold.

For a given class c, to compute the Intersection over Union metric (IoU) (see image below) between any individual predicted bounding box and the ground truth bounding boxes, take the best overlap (i.e. the highest overlap between the predicted and any ground truth box) as your final score for that predicted bounding box. If there is no ground truth bounding box, but you predict a positive window, your score for that box is 0.  
  
  
  
[10 pts] You need to write your own function to compute mAP scores, given predicted and ground-truth bounding boxes (with associated labels) as input. Needless to say, do not look up or copy solutions for this part from the web. Include your code in an appropriately named function, and use it below when needing to report mAP scores.

1. There are two datasets that you need to download, PASCAL.zip and PennFudanPed.zip, both available at the links above. For the PASCAL dataset, inside each of the train, test, and val directories, there exist three directories: 1) Images, 2) BBox and 3) Labels. The images folder contains the images from PASCAL VOC dataset, the BBox folder contains the ground truth bounding boxes of objects in every image and the Labels contains the object category for bounding boxes in every image. For PennFudanPed, there exist two directories: 1) Images and 2) Masks. The Images folder contains images from Penn-Fudan dataset and Masks folder contains the segmentation mask of objects in every image. The number of categories which are in **PASCAL** dataset is 5. The categories are person, bicycle, car, motorcycle, airplane and their corresponding labels are 1, 2, 3, 4, 5, respectively. In addition to these labels, label 0 belongs to category of background and as the result **the total number of classes** which you need to use for training process is **6**.  
   **PennFudanPed** just contains pedestrian (person) category. As the result **the total number of classes** for the object detection task is **2**. You should also download and copy files from this zip file into your working directory: Required\_Files.zip. Finally, the assignment also relies on [this API](https://github.com/philferriere/cocoapi).
2. You need to import required modules and libraries:  
   import torch  
   import torchvision  
   from torchvision.models.detection.faster\_rcnn import FastRCNNPredictor  
   from pascal\_dataset import PASCALDataset  
   import utils  
   from coco\_utils import get\_coco\_api\_from\_dataset  
   from coco\_eval import CocoEvaluator  
   import copy  
   import torch.optim as optim  
   from torch.optim import lr\_scheduler  
   from PennFudanDataset import PennFudanDataset
3. To represent our datasets, we have prepared the PASCALDataset class in pascal\_dataset.py and PennFudanDataset class in PennFudanDataset.py. You can use these classes in your code as follows:  
   dataset = PASCALDataset('path\_to\_data')  
   dataset = PennFudanDataset('path\_to\_data')  
   Note: The path to data is the path to train, test, or val sets **NOT** the directory which includes the whole dataset. As the result, you need to create a separate dataset for each of train, test and validation sets.
4. Next, create the data loader, for example by:  
   data\_loader = torch.utils.data.DataLoader(dataset, batch\_size=4, shuffle=True, num\_workers=4, collate\_fn=utils.collate\_fn)  
   where collate\_fn=utils.collate\_fn. collate\_fn=utils.collate\_fn is used to return the tuples of images and image annotations in every iteration.
5. Load the pre-trained detection model:  
   model = torchvision.models.detection.fasterrcnn\_resnet50\_fpn(pretrained=True)  
   The number of classes in pre-trained model is different from the number of classes in our datasets. So similar to what you did in homework 2, you need to replace the box\_predictor of model with a new FastRCNNPredictor layer to predict 6 classes when you are training on PASCAL dataset and 2 classes when you are training on PennFudanPed dataset.  
   Before starting the training process you need to set the optimizer, scheduler and number of epochs as before.
6. [6 pts] Now you can start to train the network and in every epoch you have two phases of train and validation. In every epoch, if the mAP of the validation set is the largest mAP so far, you need to save the model weight. Iterate over train set to perform the training process and then iterate over the validation set to evaluate the performance of trained model. Here is the set of commands to iterate over data, prepare the images and labels, and use them as input to model for object detection task:  
   for images, targets in data\_loader:  
    images = list(image.to(device) for image in images)  
    targets = [{k: v.to(device) for k, v in t.items()} for t in targets]  
    loss\_dict = model(images, targets)  
   First line in for loop: Since the images and targets are of type tuple, we need to convert them to a list. In the first line of for loop, first all Images in the batch are transferred to GPU device and then a list is created from all images. The input to the model is the list of all images of the batch.  
   Second line in for loop: A target is a dictionary which contains the bounding box of objects, label of object, image ids, the areas of bounding boxes and whether or not the image is crowded. In the second line of for loop, all the values in dictionary of every image targets (annotations) are transferred to GPU device and finally a list is created from all targets in the batch.  
   Third line in for loop: We use the the images and targets as input to the model and get the loss values. Note that the inputs to the model in **train mode** are both the images and the targets (annotations). Since the task of object detection has more than one loss value, the output for every image is a dictionary of all loss values. The dictionary contains following loss values: 1) loss\_classifier: measures the performance of the object classification for detected bounding boxes, 2) loss\_box\_reg: measures the performance of network for retrieving the coordinates of the ground truth bounding boxes, 3) loss\_objectness: measures the performance of network for retrieving bounding boxes which contain an object and 4) loss\_rpn\_box\_reg: measures the performance of network for retrieving the region proposals.  
   You need to sum all losses and backpropagate the loss. As before, in every iteration of training phase you need to zero the gradient and apply step function of the optimizer. After iterating over the whole train set, you need to update the scheduler.
7. [6 pts] In the validation phase you need to create a coco evaluator to evaluate the performance of the network. To create an evaluator, first you need to create a coco API from our dataset:  
   coco = get\_coco\_api\_from\_dataset(data\_loader.dataset)  
   Then you need to specify the IoU type:  
   iou\_types = ["bbox"]  
   At the end, you can create a coco evaluator from coco API and IoU types:  
   coco\_evaluator = CocoEvaluator(coco, iou\_types)  
   At this point, you can start to iterate over validation set and compute the mAP. In every iteration first you need to transfer the images to GPU and then use them as input to the model. The input of model in **evaluation mode** is just images and as opposed to train phase, you do not need to transfer the target (annotations) to GPU. Here is the command to get the object detection for the images:  
   outputs = model(image)  
   For evaluation in coco\_evaluator, the outputs needs to be on CPU and you need to transfer them from GPU to CPU. Then you need to create the pair of target and output as follows:  
   res = {target["image\_id"].item(): output for target, output in zip(targets, outputs)}  
   Now the res is used to update the coco evaluator in every iteration: coco\_evaluator.update(res)  
   After iterating over all images, you need to run the following commands to get the final results for evaluation in every epoch: coco\_evaluator.synchronize\_between\_processes()  
   coco\_evaluator.accumulate()  
   coco\_evaluator.summarize()  
   At this point you can get the mAP over all validation set by the following command: coco\_evaluator.coco\_eval['bbox'].stats[0]  
   You need to save the model weight which has the highest mAP on the validation set.
8. [8 pts] Train networks for both PASCAL and PennFudanPed with a few different hyperparameters. Report the performance of the model which has the best test accuracy among all of your experiments (as a text snippet inside your notebook), and use it to visualize the object detection results. For your visulization, you need to write a code to draw the bounding boxes which have been detected by the network in the image. In addition to bounding boxes, the name of category and its score should be shown somewhere around the bounding box. Your code needs to find 20 images from the test set with highest mAP, draw the bounding boxes and save the outputs a directory; you will then submit these files.

Part D: Object Detection with Facebook's Detectron2 (10 points)

Go through [the following tutorial](https://colab.research.google.com/drive/16jcaJoc6bCFAQ96jDe2HwtXj7BMD_-m5) to determine how to apply the pretrained Detectron2 model on 10 images of your choice. Include the results in your submission.  
  
  
**Acknowledgements:** This assignment was prepared for you by Narges Honarvar Nazari, partly adapted from [PyTorch tutorial in transfer learning](https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html), and based on assignments developed by Chris Thomas and Nils Murrugarra-Llerena. The photos used for this assignment come from the [PASCAL VOC](http://host.robots.ox.ac.uk/pascal/VOC/) dataset and the [Penn-Fudan](https://www.cis.upenn.edu/~jshi/ped_html/) dataset.