Homework 4 due: 5/18 11:59 pm

Submission Instructions

You will need to submit the following materials:

1. **Questions and Answers** in PDF format. Remember to put your name in your report.
2. A folder containing all ipynb files with your results for the the problems. Please remember to keep all results and logs in the submitted ipynb files to get full credit.

All submitted files should be put into one single zip file named as HW#_xxx.zip, e.g. HW4_George_Clooney.zip**,** including all ipynb files with answers (both results and discussions), and the **Questions and Answers** report.

Problem 1: Faster RCNN (20%)

1) Getting Started

In this homework assignment, we will use Detectron2 to help us to do the tasks of detection and segmentation.

Detectron2 is Facebook AI Research's software system that implements object detection algorithms. Here, we will go through some basic usage of detectron2, and finish the following tasks:

- Run inference on images, with existing pre-trained detectron2 models
- Train your own models on two custom datasets: traffic sign & balloon

2) Run a pretrained model

We first download some images from the given URLs. We can see there exists multiple objects in these images: bottles, microwaves, tables, refrigerators, people, etc. Let us see if we can detect them all by using a pre-trained model given by Detectron2.

Fig.1 downloaded images (input.jpg, test1.jpg, test2.jpg)

Let's take the first image (input.jpg) as an example. After applying the `COCO-Detection/faster_rcnn_R_50_FPN_1x.yaml`, a Faster R-CNN model which has a ResNet-50 backbone with the feature pyramid for dealing with different scales of objects and was pre-

trained on COCO dataset with 80 different object categories, the model outputs are shown as the following, Let's take a look at the model output:

```
tensor([72, 39, 39, 39, 68, 45, 75, 55, 71, 41, 39, 75, 41, 45, 71, 41, 39],
       device='cuda:0')
Boxes(tensor([[  0.6764, 158.7422, 105.0817, 397.2280],
        [480.6438, 298.5876, 509.7858, 378.3471],
        [293.3246, 287.8804, 344.7385, 388.2297],
        [501.2999, 306.3395, 533.4362, 387.4037],
        [300.8816, 148.8771, 366.4546, 227.1448],
        [104.8806,  73.4484, 148.7521,  88.7021],
        [205.1707,  55.5022, 231.9424,  93.1075],
        [357.5101, 394.1846, 393.9808, 425.1449],
        [ 38.0390, 355.4640, 315.9145, 442.3582],
        [357.1301, 286.4151, 387.7237, 365.4046],
        [197.1212, 267.4070, 208.2965, 292.6432],
        [241.4115,  57.8996, 267.1059,  93.9473],
        [454.1900, 310.0651, 484.8028, 360.4231],
        [243.9083, 382.6956, 275.9801, 407.7662],
        [185.0177, 355.2176, 304.0785, 409.8493],
        [441.3940, 307.2240, 470.4242, 347.5514],
        [330.0779, 267.5388, 355.4027, 303.1052]], device='cuda:0'))
```

In **inference** mode, the model outputs a `list[dict]`, one dict for each image. For the object detection task, the dict contains the following fields:

* "instances": Instances object with the following fields:
  * "pred_boxes": Storing N boxes, one for each detected instance.
  * "scores": a vector of N scores.
  * "pred_classes": a vector of N labels in range [0, num_categories].

For more details, please see https://detectron2.readthedocs.io/tutorials/models.html#model output-format for specification. We can use "Visualizer" to draw the predictions on the image.

Fig.2 Visualization of model outputs

The model we just used is a two-stage Faster R-CNN detector, `COCO-Detection/faster_rcnn_R_50_FPN_1x.yaml`. Actually, the Detectron2 provides us more than that, you may find great amounts of models for different tasks in the given [MODEL_ZOO](https://github.com/facebookresearch/detectron2/tree/master/configs). Let's try to use some of these models and answer the following questions.

- Object Detection. Use the same configuration `COCO-Detection/faster_rcnn_R_50_FPN_1x.yaml`, with IoU threshold of 0.5 (`SCORE_THRESH_TEST=0.5`), to also run inference on the **rest two images** (test1.jpg & test2.jpg) and **view** the outputs with bounding boxes.
- Object Detection. Use the `COCO-Detection/faster_rcnn_R_101_FPN_3x.yaml`, which has a ResNet-101 as the backbone, with IoU threshold of 0.5 and **view** the outputs of all three images with bounding boxes. By looking at the outputs, can you find the **difference** with the one `COCO-Detection/faster_rcnn_R_50_FPN_1x.yaml` we used in Q1? (e.g., numbers of objects, confidence scores, ...)
- Object Detection. Use the `COCO-Detection/faster_rcnn_R_101_FPN_3x.yaml` with an IoU threshold of 0.9 and **view** the outputs of all three images with bounding boxes.

3) Train on a custom dataset - Faster R-CNN

You have already tried the pre-trained model on MS COCO datasets. Why not train your own model to perform customized detections? Here, let's **fine-tune** an existing Faster R-CNN model on a custom dataset in a new format.

We use the [traffic sign dataset] (https://www.dropbox.com/s/d8y6uc06027fpqo/traffic_sign_data.zip?dl=1). We'll train a traffic

sign detection model from an existing model pre-trained on COCO dataset, available in detectron2's model zoo. Note that the COCO dataset does not have the "traffic sign" category, but we'll be able to recognize this new class in a few minutes.

a. Prepare the dataset.
Follow the instructions in the notebook to download and unzip the data. You will find the file structures like this: traffic_sign_data/ ├── JPEGImages ├── labels ├── traffic_sign_test.txt └── traffic_sign_train.txt

- JPEGImages: .jpg format images
- labels: corresponding .txt files that contain the bounding box label information
- traffic_sign_train.txt: a file that contains the path to the training images
- traffic_sign_test.txt: a file that contains the path to the testing images

Here, the traffic sign dataset is in its custom dataset, therefore we write a function to parse it and prepare it into detectron2's standard format. See `get_traffic_sign_dicts` function in the notebook for more details. To verify the data loading is correct, let's visualize the annotations of randomly selected samples in the training set:

Fig.3 Random visualization of *bbox* annotations of training samples

b. Train our own model.
Now, let's fine-tune a COCO-pretrained R50-FPN Faster-RCNN model on the traffic sign dataset. We use Tensorboard to visualize the training progress. The `total_loss` is the sum of 4 losses, RPN regression loss: loss_rpn_loc, RPN objectness loss: loss_rpn_cls, ROI regression loss: loss_box_loc and ROI classification loss: loss_box_cls. The x-axis represents the number of iterations, and y-axis represents the loss value. With the increase of iterations, you will notice the total_loss decreases.

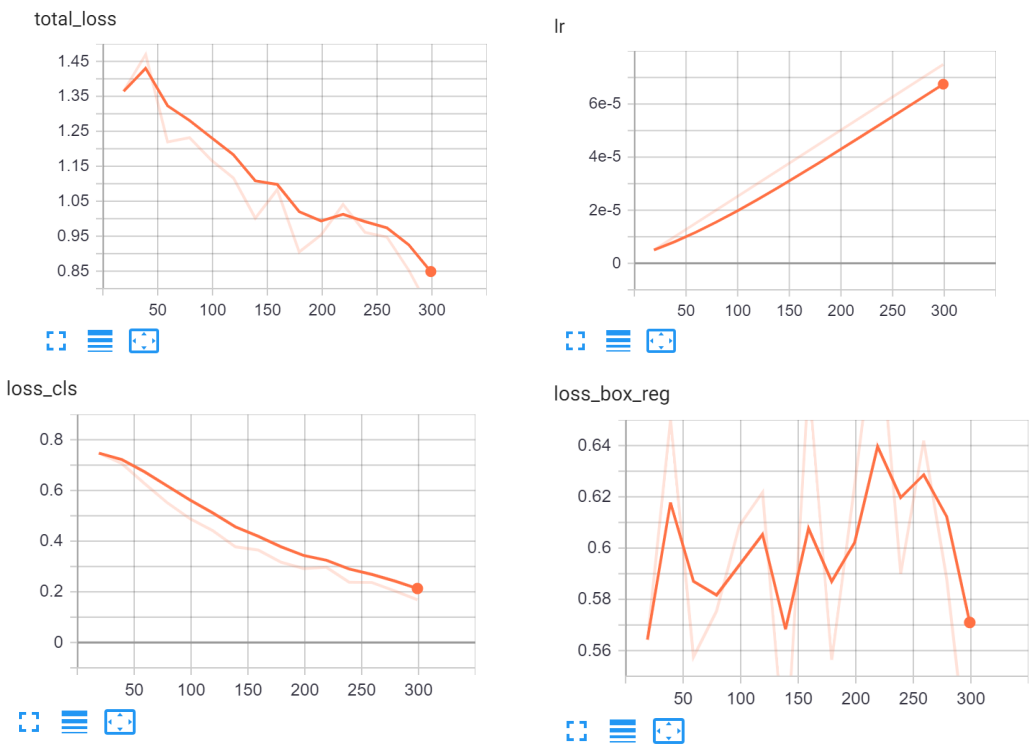


Fig.4 Training Curves shown in the Tensorboard

c. Inference & Evaluation state-of-the-art

Inference: Now let's run inference containing everything we've set previously. First, let's create a predictor using the model we just trained. Then, we randomly select several samples to visualize the prediction results.

Fig.5 Visualization of Faster R-CNN outputs

Evaluation: We can also evaluate its performance using the Average Precision (AP) metric.

1. Loading and preparing results...
2. DONE (t=0.01s)
3. creating index...

```
index created!
Running per image evaluation...
Evaluate annotation type *bbox*
COCOeval_opt.evaluate() finished in 0.08 seconds.
Accumulating evaluation results...
COCOeval_opt.accumulate() finished in 0.02 seconds.
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.353
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.570
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.409
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.120
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.309
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.496
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.253
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.517
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.573
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.311
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.585
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.686
[10/28 05:09:22 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   | AP50  | AP75  | APs   | APm   | APl   |
|:------:|:-----:|:-----:|:-----:|:-----:|:-----:|
| 35.349 | 57.048 | 40.929 | 12.027 | 30.909 | 49.640 |
OrderedDict([('bbox', {'AP': 35.34874923307184, 'AP50': 57.04849039231301, 'AP75': 40.92856309409448, 'APs': 12.026670579981444, 'APm': 30.909472073584805, 'APl': 49.639910640259735})])
```

The AP is ~35.3%. You may also see the detailed metrics for small, medium and large objects as well. Not bad! Here are something that I want you to try by yourself:

- **Change** the initial learning rate (`BASE_LR`) from `0.0001` to `0.00025` and show the 4 training curves from the TensorBoard. By viewing the results (You may keep the rest of configurations fixed), does it improve the AP or not? Explain why.

- **Change** the number of iterations (`MAX_ITERS`) from `300` to `500` and show the 4 training curves from the Tensorboard. By viewing the results (You may keep the rest of configurations fixed), does it improve the AP or not? What about `1000`? Explain why.

- **Apply** the data augmentation techniques mentioned in HW2 to the training set and show the 4 training curves from the Tensorboard and view the AP performance. Does it improve the AP or not? Explain why.

Problem 2: Tracktor for Pedestrian Multi-Object Tracking (20%)

Modern multiple object tracking (MOT) systems usually follow the **tracking-by-detection** paradigm. It has (1) a detection model for target localization and (2) an appearance embedding model for data association (3) frame-by-frame association rules. As discussed in the class, Tracktor (https://arxiv.org/pdf/1903.05625.pdf) is a recent MOT method that accomplishes tracking without specifically targeting any of these tasks, in particular, this method performs no training or optimization by exploiting the bounding box regression of an object detector to predict the position of an object in the next frame, thereby converting a detector into a Tracktor.

As shown in Fig. 6, first, the regression of the object detector aligns already existing track bounding boxes b_{t-1}^{k} of frame $t-1$ to the object's new position at frame t. The corresponding object classification scores s_t^k of the new bounding boxes positions are then used to kill potentially occluded tracks. Second, the object detector (or a given set of public detections) provides a set of detections D_t of frame t. Finally, a new track is initialized if a detection has no substantial IoU with any bounding box of the set of active tracks $B_t = \{b_t^{k1}, b_t^{k2}, \ldots\}$.

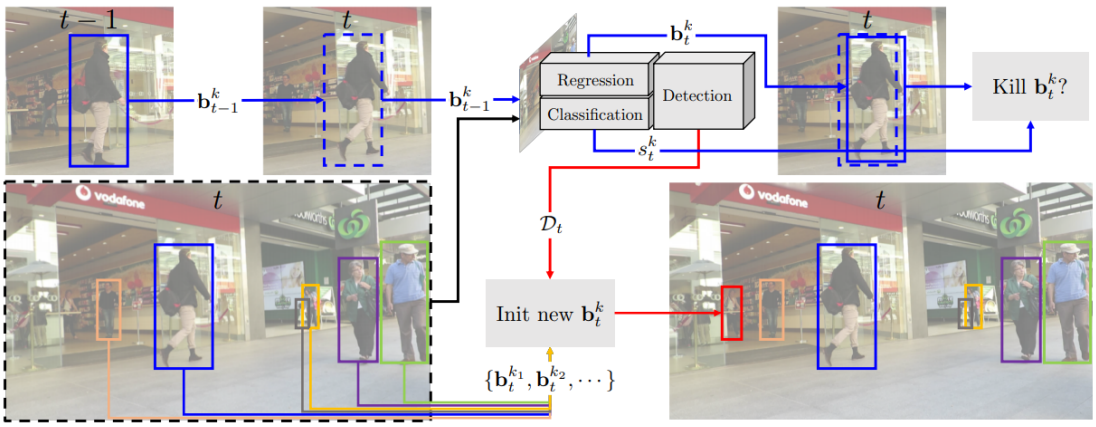


Fig 6: The presented Tracktor accomplishes multi-object tracking only with an object detector and consists of two primary processing steps, indicated in blue and red.

(a) Prepare Codes and MOT-17 Datasets

In this problem, we will use the MOT-17 dataset to train an object detector. The multi-object tracking benchmark MOT-17 (https://motchallenge.net/) consists of several challenging pedestrian tracking sequences, with frequent occlusions and crowded scenes. Sequences vary in their angle of view, size of objects, camera motion and frame rate.

In order to use the MOT-17 dataset to train both models, we first download and prepare this dataset:

1) Download the source codes from GitHub using the provided sample codes.
2) Download the MOT-17 dataset (~1.8G) and unzip the data to the directory.
3) Visualize some images in the MOT-17 dataset.

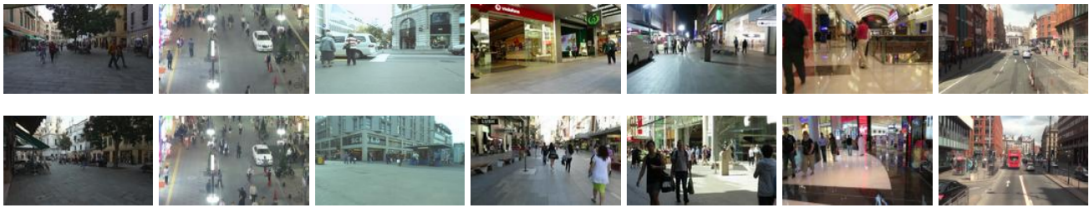


Fig. 7: Examples in the MOT-17 dataset

(b) Train and Test the Object Detector (10%)

Remember we've learned multiple object detectors, which are mainly sorted into (1) two-stage detectors, i.e., Faster R-CNN (2) one-stage detectors, i.e, RetinaNet. Successful application of multi-object tracking requires not only a reliable but a real-time detector as well. Here, we adopt the two-stage Faster R-CNN with the backbone of ResNet-50 and Feature Pyramid Network (FPN), which can detect objects of different scales, with some modifications.

1) **(5 points)** Use the provided pretrained Faster R-CNN to further train the model for 27 epochs on MOT-17 dataset. Use the sample codes to evaluate and report the accuracy of Average Precision (AP) on both train set (**val test**).
2) **(5 points)** Randomly select some images and visualize their detection results.

(c) Inference and Visualization (10%)

Based on the (b), the model for object detection is saved in outputs/ directory. Now we can run the Tracktor to perform the multi-person tracking. The Tracktor can be configured by changing the corresponding experiments/cfgs/tracktor.yaml config file. The default configurations runs Tracktor with the FPN object detector are almost same as described in the paper except the Re-identification model is turned off (do_reid=False, **load_results=True**).

1) Run the inference experiments/scripts/test_tracktor.py using the MOT-17 train set input. The tracking results are logged in the corresponding outputs/ folder. Open one of the generated results, explain what are the first six values generated in each line? (i.e., frame_id, bounding box (xywh/xyxy?), confidence, track_id, etc.). Plot the values on the corresponding images and show the video results.

Fig. 8: Visualization of Tracktor tracking results

2) Evaluate the performance using test_tracktor.py and report the following metrics: MOTA, MOTP, IDF1, FP. (**Hints: These metrics have already been logged in Colab outputs from the previous problems. You can just copy down here.**)
3) Run the inference test_tracktor.py with the changed configurations in experiments/cfgs/tracktor.yaml and evaluate the performance:
   tracktor/tracker:
   - detection_person_thresh (FRCNN score threshold for detections): 0.5
   - detection_nms_thresh (NMS threshold for detection): 0.3
   - number_of_iterations (maximal number of iterations): 100
   - max_features_num (How much last appearance features are to keep): 10
   - motion_model (motion model settings, mentioned in 2.3): disabled

   Feel free to change at least **three** hyperparameters (can be from detection or tracking). **Discuss** how these changes may affect the tracking performance based on MOTA.

Problem 3: Mask R-CNN on a custom dataset (10%)

In Problem 1, We use Faster R-CNN to train on the traffic sign datasets to perform **object detection**. With few line modifications, we can train an **instance segmentation** model using Mask R-CNN as well. Notice that the traffic sign dataset only contains the bounding box labeling information, with no segmentation mask labeling, which is not enough to train a Mask R-CNN model. Due to this reason, we switch to another dataset: [balloon segmentation dataset](https://github.com/matterport/Mask_RCNN/tree/master/samples/balloon), which only has one class: balloon. Follow the instructions in the notebook to download the balloon dataset and answer the following questions.

Write codes to load and visualize the balloon dataset in the similar manner. You need to take a careful look at the label files and construct your `get_balloon_dicts` functions to load extra poly mask information. If you load the dataset correctly, you will see training samples like the following.

Fig.9 Random visualization of *segm* annotations of training samples

- **Fine-tune** the pre-trained model `COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_1x.yaml` on the balloon dataset with the following configurations and **show** the TensorBoard Visualization.
    - IMS_BATCH_SIZE = 2
    - BASE_LR = 0.00025
    - MAX_ITER = 300
    - ROI_HEADS.BATCH_SIZE_PER_IMG = 128
    - ROI_HEADS.NUM_CLASSES = 1

- Use your own trained model to do the **inference** on testing datasets, at least **plot 3** prediction results. Then, use the COCO API to **report** your testing Average Precision (AP). If your model is trained correctly, you will see the prediction results like the following figures.

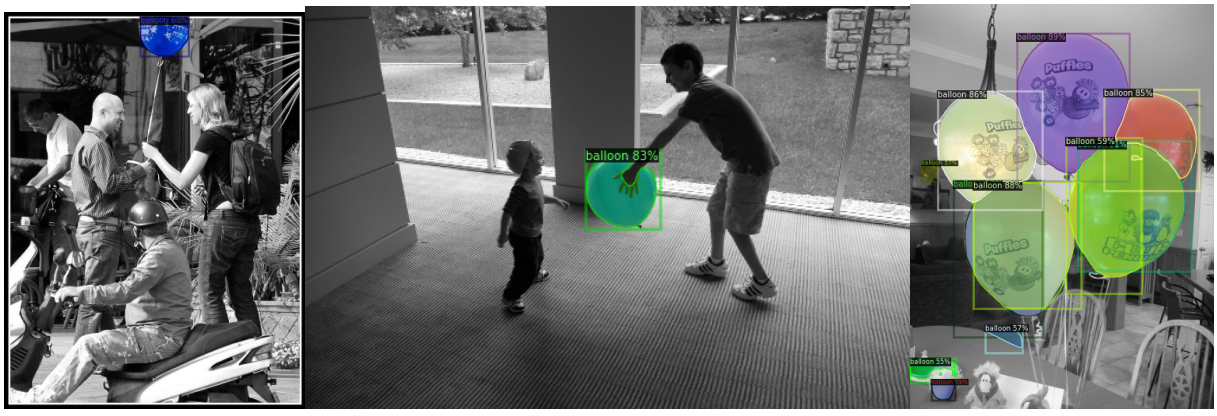


Fig.10 Visualization of Mask R-CNN outputs

Problem 4: 2D Human Pose Estimation (25%)

Human pose estimation (HPE) is an important task in the computer vision community. For the deep learning based HPE methods, they are usually divided into two categories, i.e., **top-down**

and **bottom-up**. The top-down approach starts by identifying and localizing individual person instances using a bounding box object detector, such as a Faster R-CNN. This is then followed by estimating the pose of a single person. The bottom-up approach starts by localizing identity-free semantic entities, then grouping them into person instances. OpenPose introduced in the lecture is a bottom-up method that predicts the keypoints directly from the global image and then groups the keypoints from the same identity.

In this problem, we would like to discuss a top-down HPE method, named Stacked Hourglass Network and work with an MPII human pose dataset (http://human-pose.mpi-inf.mpg.de/).

(a) MPII Dataset Introduction

MPII Human Pose dataset, provided by the Max Planck Institute for Informatics, is a comprehensive benchmark for evaluation of articulated human pose estimation. The dataset includes around 25K images containing over 40K people with annotated body joints. The images are systematically collected using an established taxonomy of everyday human activities.

Overall the dataset covers 410 human activities and each image is provided with an activity label. Each image is extracted from a YouTube video and provided with preceding and following un-annotated frames. In addition, for the test set richer annotations are available, including body part occlusions and 3D torso and head orientations.

Fig. 11: Some examples of human poses in MPII dataset.

(b) Stacked Hourglass Network

The stacked hourglass network (HG) for human pose estimation was proposed in ECCV'16 (The paper can be found at http://arxiv.org/abs/1603.06937). Unlike OpenPose, HG is a top-down method for human pose estimation, which predicts a single human pose from each input image.

HG is a stack of hourglass modules. It got this name because the shape of each hourglass module closely resembles an hourglass (shown in Fig. 12), which is similar to the fully convolution network (FCN) discussed in object segmentation. The idea behind stacking multiple HG modules instead of forming a giant encoder and decoder network is that each HG stack module will produce a full heat-map for joint prediction. Intermediate supervision is applied to the

predictions of each hourglass stage, i.e, the predictions of each hourglass in the stack are supervised, and not only the final hourglass predictions. Thus, the latter HG module can learn from the joint predictions of the previous HG module.

Similar with the definition of heatmaps to represent joint locations mentioned in OpenPose, the authors of this work also adopt the idea and find the peaks of the heatmaps and use that as the joint locations.

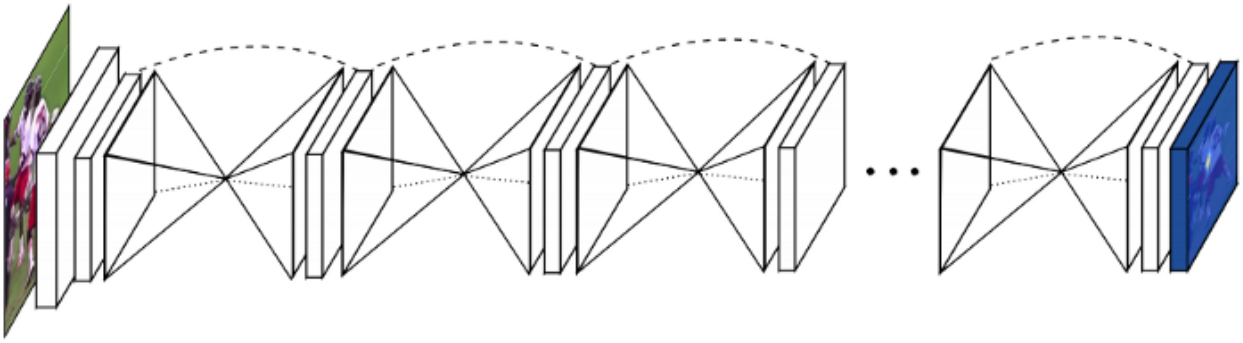


Fig. 12: Diagram of Stacked Hourglass Networks for Human Pose Estimation.

As for each hourglass module inside the diagram, the architecture is shown in Fig. 13. In the figure, each box is a residual block, proposed in the ResNet paper (https://arxiv.org/abs/1512.03385). In general, an HG module is an encoder and decoder architecture, where we downsample the features first, and then upsample the features to recover the info and form a heat-map. Each encoder layer would have a connection to its decoder counterpart, and we could stack as many as layers we want.

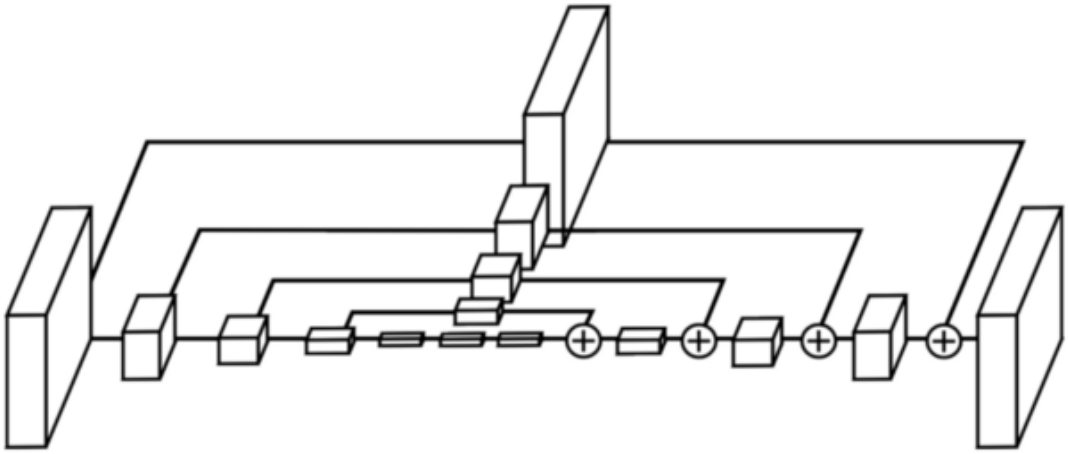


Fig. 13: Architecture of the hourglass module.

Some resources above credit to: https://towardsdatascience.com/human-pose-estimation-with-stacked-hourglass-network-and-tensorflow-c4e9f84fd3ce.

(c) Train the Network using MPII Dataset

Here, we use an open source package of the stacked HG: https://github.com/princeton-vl/pytorch_stacked_hourglass, for this problem.

1) Download the source code from GitHub using the provided sample code.
2) Download the MPII dataset and unzip the data to the directory (may take ~20 min).
3) Visualize some images in the MPII dataset.
4) Change the following parameters in the configurations:
    - 'nstack': 2 (number of stacks)
    - 'train_iters': 100 (number of iterations in each epoch)

   Train the network for 4 epochs. Draw the plot of the loss values from your training log file. (No need to be well-trained if you do not have enough time.)
5) Evaluate your trained models on the MPII validation set using the tool provided. Print out your evaluation scores. (Your scores are not necessarily good since it requires a very long time for the training)

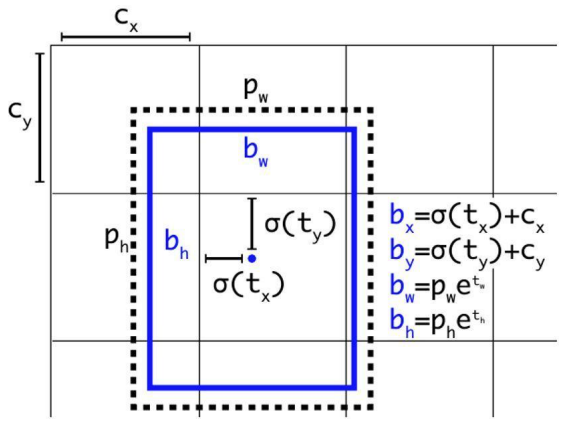
(d) Inference and Visualization

1) Download 2HG and 8HG pretrained model from https://github.com/princeton-vl/pytorch_stacked_hourglass#pretrained-models (8HG means the number of stacks is 8).
2) Infer the HPE results on MPII validation set and evaluate the performance for both 2HG and 8HG models.
3) Write visualization code and visualize some human pose inference results.
4) Find some images from the Internet (or your own images) with humans and try to infer human pose using the pretrained model. Visualize the results.

Problem 4: Questions and Answers (25%)

1. (8%) In a Faster RCNN, how many different classifiers and regressors are used, what are their corresponding loss functions used? Also given a 800x1000, what are the total number of anchors can be created (assume we sample proposals from the feature maps after 16 times of size reduction through maximum pooling).

2. (8%) Given a YOLO2 regressor output of a detected object bounding box value of (t_x, t_y, t_w, t_h)=(1.6, -2.8, -0.36, 0.27), of an image of 480x480 size, with respect to a cell of top-left corner offset (c_x, c_y)=(128, 192) and the bounding box prior of size (p_w, p_h)=(46, 72) , what is the corresponding (b_x, b_y, b_w, b_h) of the detected object bounding box when displayed in the original image of size 416x416?



3. (9%) A Hungarian assignment

	Detection 1	Detection 2	Detection 3
Tracklet 1	482	437	512
Tracklet 2	421	399	432
Tracklet 3	502	510	518
Tracklet 4	414	402	411

There will be one of the tracklet cannot find the corresponding detection at the current frame, it might be due to the tracklet without assignment has exited the field of view (FOV) of the camera, or the detector misses the object detection, i.e., a false negative. What is your thought on determining which is the scenario?