Analysis & Data Presentation Laboratory- 094295

Project Report- HW2

COVID19: face-mask object detection

Submitters: Almog Gueta 204783351, Inbal Croitoru 312234396

Code is available via this GitHub repository:
https://github.com/almog-gueta/Lab-hw2-face-mask-detection

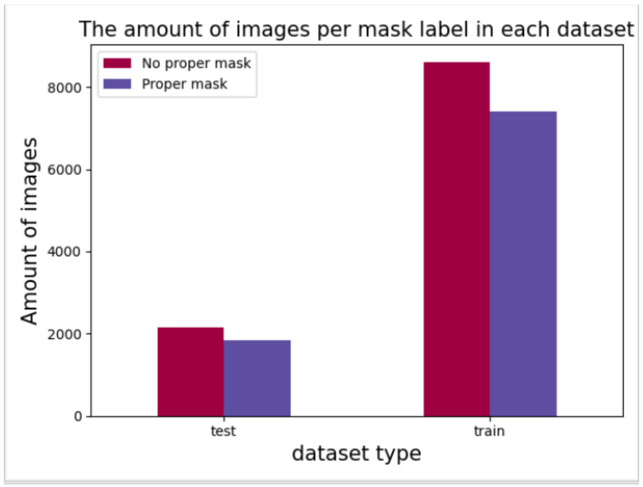
EDA

Raw Data:

The dataset consists of jpg images of faces of people from a variety of genders, ages, and ethnics, and from different angles. The images are in different sizes, where the maximum size is 224x224. Each image contains a label (proper mask, no mask) and a bbox (bounding box) from the format: [x,y,w,h]. If the image contains more than one person, the prediction is for the most left one with his face covered.

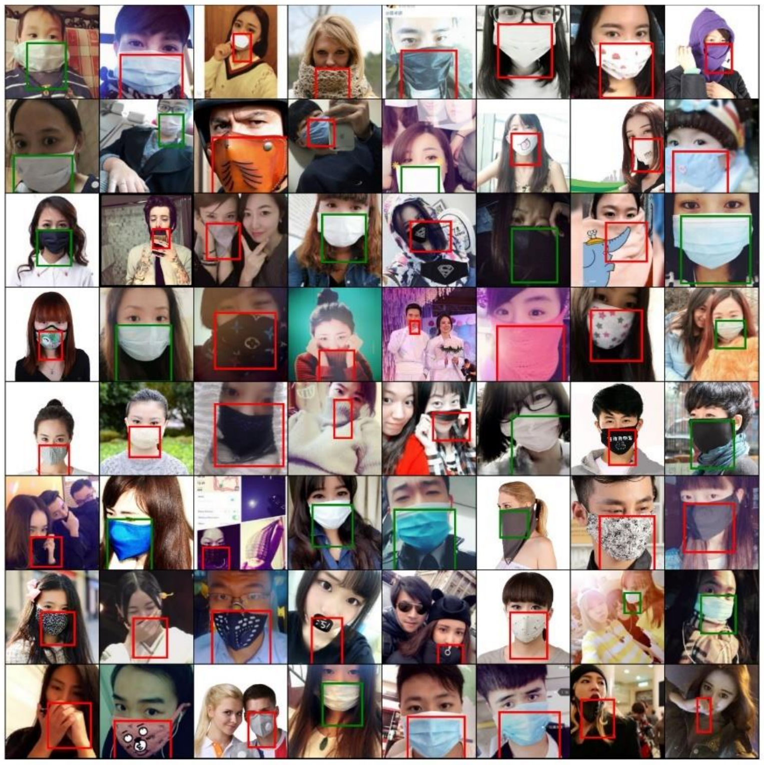
- Distribution of images per dataset, with separation to classes:

Dataset	**Total**	**Proper mask**	**No mask**
Train	16000	7401	8599
Test	4000	1849	2151



Random train images with their bboxes

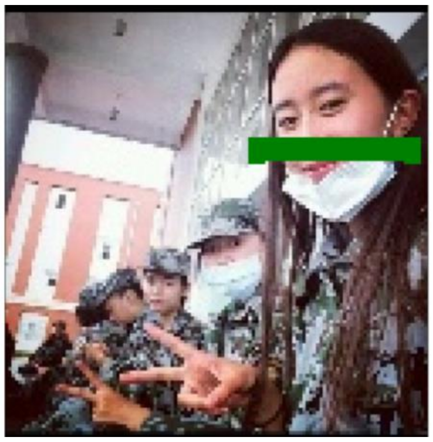
Random test images with their bboxes



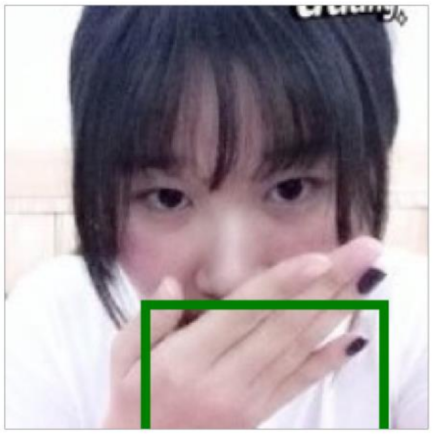
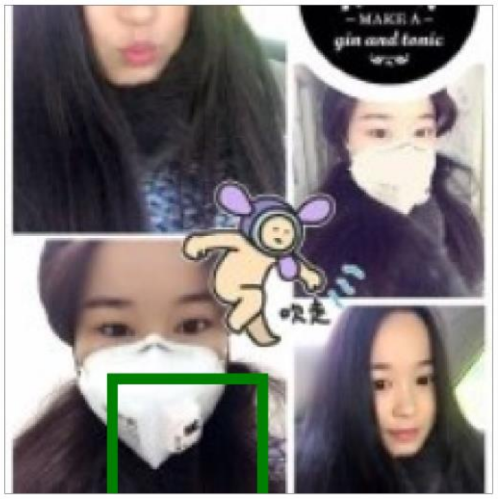
From the example images we can see that there are different shooting angels, different masks textures, different sizes of masks, and different amount of people in the image. Importantly, we can see that the variety in the images we presented occur in both the training set and the test set.

While exploring the data, we came across with unusual images:

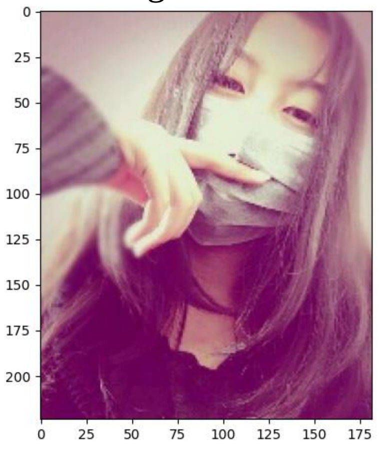
- One image has a negative "h" value and was deleted from the dataset: 004828__[72, 85, 50, -2]__False.jpg



- Some images have bbox that is outside of the image boundaries. In these cases, we changed the bbox to be at the corresponding boundary of the image.

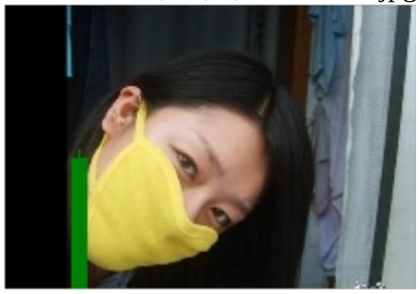



- Some images have a smaller size than 224x224:

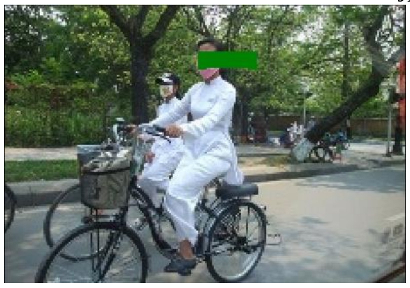


- Some images have bbox that is a "line" and not a box, or a box of size 0 and were deleted from training set and 'fixed' on test set by expanding the bbox in one pixel in the problematic coordinate:

008710__[40, 78, 0, 86]__True.jpg



016148__[108, 30, 32, 0]__False.jpg



We can see in green that the bboxes do not have any area.

Experiments

We have conducted 3 different experiments, each one with a different model: YOLO, FasterRCNN, and a not-pretrained Resnet followed by linear layers (this is our basic model). The YOLO experiment did not gave good results when not using a pretrained network, and the available code from pytorch hub was not easy to manipulate to our scenario (different measurements, only one box per image, etc.). Therefore, we did not continue with it and will not present it in this report.

In both experiments (FasterRCNN, Resnet) we had a similar loading, pre-processing and cleaning procedures:

1. Parsing the image info (label_bbox_imageid) from the image file name.
2. Loading the image using PIL.
3. Transformation on the image: converting toTensor (also scale to [0,1]).
4. Fixing bbox coordinates that are out of boundaries or not valid (for example a 'line' instead of a 'box').
5. For FasterRCNN, converting the bbox from format [x,y,w,h] to [x_min, y_min, x_max, y_max].
6. For Resnet, we resized the images to a fixed size of 224*224 and normalized the images using the train dataset's mean and std. Since we resized the images, we also resized the bounding boxes so they will fit the new images' sizes.

Just to clarify, we did NOT use any pretrained model nor external data.

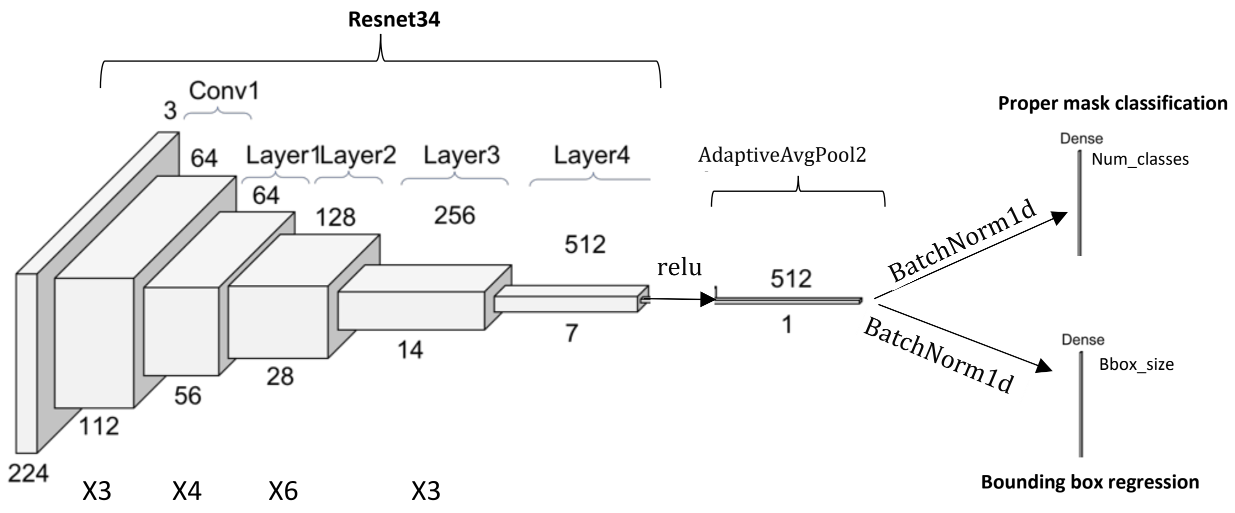
Our final model is FasterRCNN with a Resnext50 backbone.

1. Our Basic Net:

Results:

	Accuracy [0,100]	**IoU [0,1]**	**Average score [0,1]**
Train	75.6	0.44	0.598
Test	74.8	0.44	0.594

Architecture: Since Resnet is a well-known and robust architecture for vision tasks, and since it is easy to use it with Torchvision implementation, we decided to represent the images with layers of resnet34 as described in the net architecture figure. In order to receive the bbox and class predictions we added two separate linear layers:



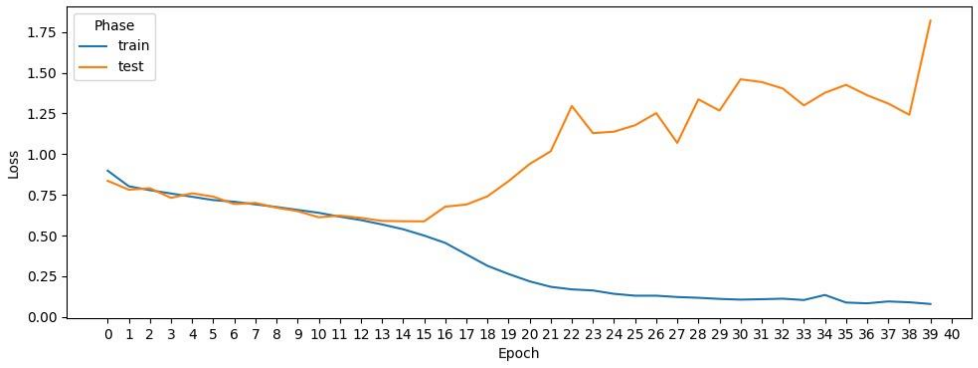
Loss functions: For each task we used a suited loss function and for the backpropagation process we summed both losses. For the regression task, since we predict coordinates, we chose the L1 loss, and for the classification task we used Cross Entropy loss.

Optimizers: For this model we used Adam optimizer.

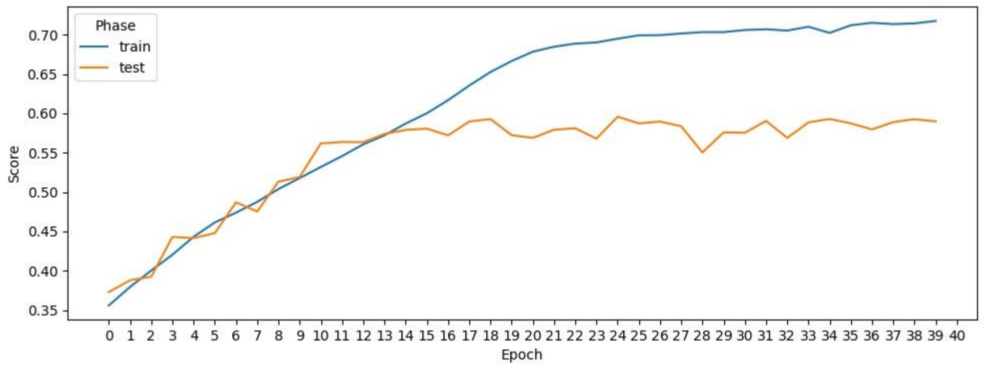
Hyper-parameters: The best results were obtained for the following values: batch size 16, learning rate 0.006, epoch 18 (chosen by observing the graphs below).

Graphs:

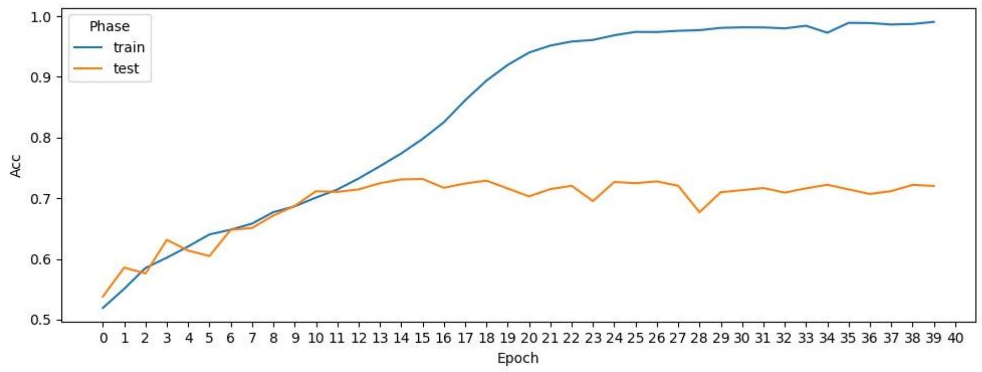
Loss as a function of epoch



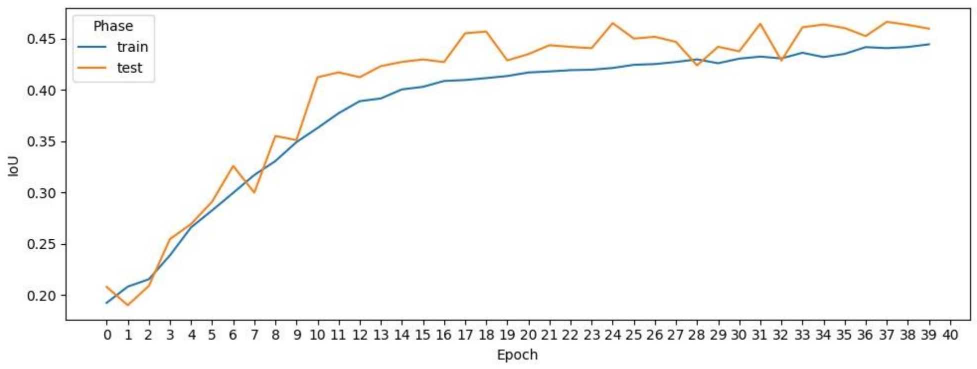
Averaged IOU and accuracy as a function of epoch



Accuracy as a function of epoch



IOU as a function of epoch



1. FasterRCNN – experimented 3 backbones, chose Resnext50:

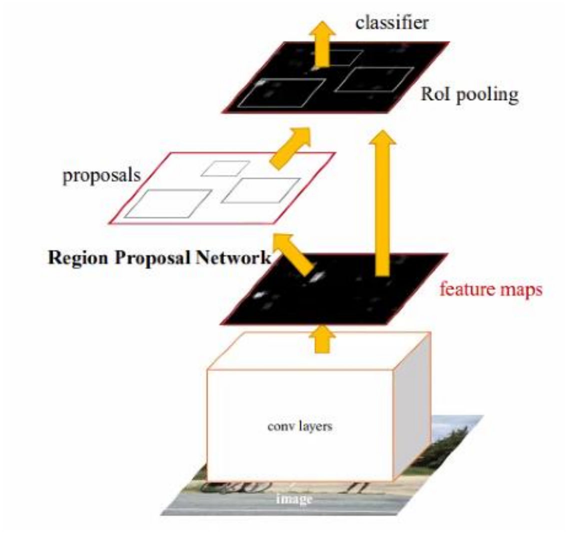
Results:

	Accuracy [0,100]	**IoU [0,1]**	**Average score [0,1]**
Train	80.504	0.709	0.75702
Test	77.875	0.658	0.718375

Architecture: We have used the known FasterRCNN model implemented by Torchvision, used for object detection (bbox and classification).

- The model was proposed in the following paper: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks
- The source code is here: https://pytorch.org/vision/stable/_modules/torchvision/models/detection/faster_rcnn.html
- The relevant tutorial is here: https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

This model is an end-to-end, fast model for OD, composed of two modules: a deep fully convolutional network that proposes regions, and a Fast R-CNN detector that uses the proposed regions. The model returns both the bbox predictions and the corresponding classifications.
To fit our task, we limited the model to return at most one bbox per image.



The model's default backbone is Resnet50. We have tried 3 different backbones: **Resnet50, Resnet34, and Resnext50_32x 4d**. In addition, we have tried both options- with and without a schedular. The rest of the parameters were the same. The best results were obtained using a schedular, and with Resnext50_32x 4d backbone, and this is the backbone we chose. However, the results were very similar.

The original implementation has two operating modes: training and evaluating. In the training phase, the model returns only the loss. In evaluation, it returns only the predictions. Since we wanted both predictions and loss while evaluating, we used a modification on the code that was proposed in this forum, and was implemented by our friends Omer and Nitzan. The modified implementation is available in this GitHub fork.

Loss functions: The Torchvision implementation includes both the model and the loss calculation. Each forward() step returns the loss itself. The model uses multiple losses, that we summed over each batch, and used the summed loss in the backward() step. The model includes two types of losses: classification loss measured by Binary Cross Entropy, and regression loss (for the bbox predictions) measured by L1 loss. Each loss type is calculated to each module of the model (out of the two), therefore we have 4 losses total.

Optimizers: We have tried both SGD and Adam, and chose Adam because it gave better results. We used an initial learning rate of 1e-3 and used a scheduler to control the lr during training. We will mention that we have tried a larger initial lr, although it resulted in exploding gradients, so we reduced it.
In addition, we have used gradient clipping in order to control better the gradients updating procedure.

Hyper parameters tuning: The tuning procedure was held by training the model on the train set, and evaluating on both train and test sets. We have tried different parameters' values, most of them are based on suggestions on tutorials and forums. For some parameters, we kept the default values. The remaining parameters that we have tuned are:

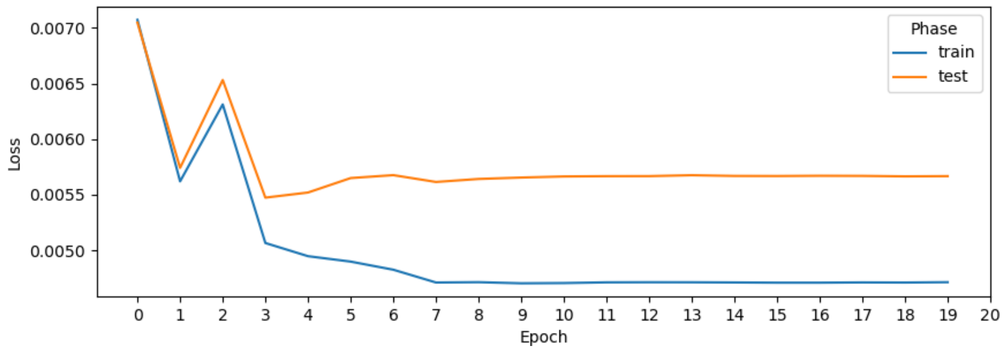
- box_detections_per_img = 1
- min_size = max_size = 224
- lr = 1e-3
- lr_step_size=3
- lr_gamma=0.1

Training configuration: batch size=32, epoch=9 (out of 20).

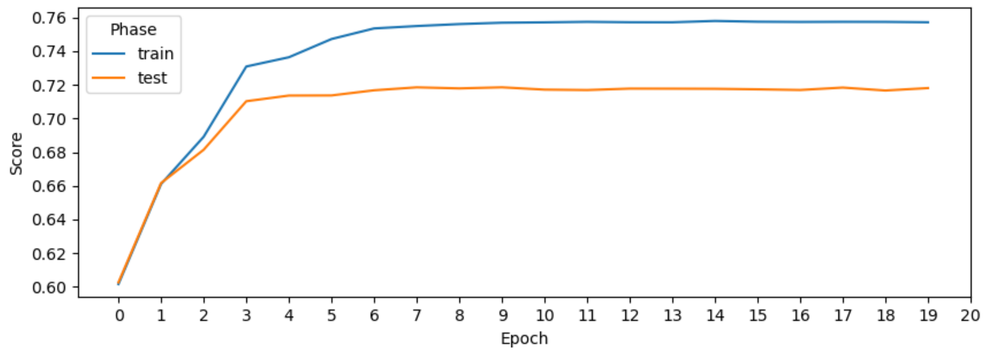
the model has 40761054 trainable parameters. The training time is 24 minutes per epoch on the VM.

Graphs:

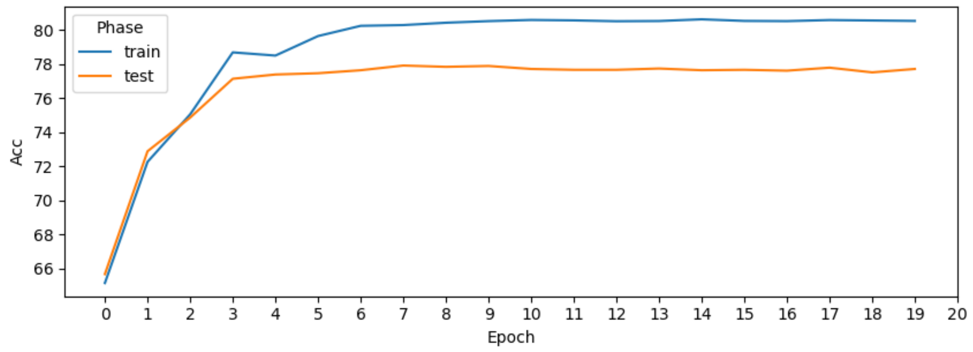
Loss as a function of epoch



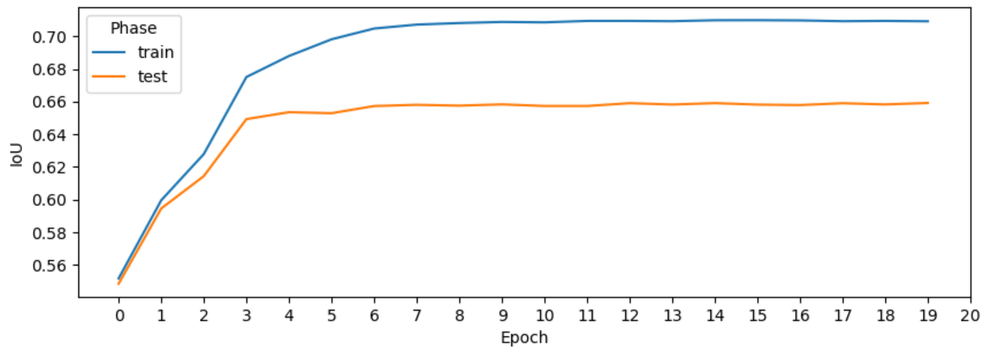
Averaged IOU and accuracy as a function of epoch



Accuracy as a function of epoch



IOU as a function of epoch



Conclusions

Considering both models' results, by observing the models' loss graphs, average scores, accuracies and IoUs, we can clearly see that FasterRCNN is a more advanced model which provided better results and captured the prediction task in a much better way.

As mentioned above, the FasterRCNN experiment includes 3 sub-experiments with different backbones. We chose to use the Resnext50 backbone since it provides the best combination of loss, accuracy, IoU. The model was trained for 20 epochs, and we chose the model's weights from epoch 9 after considering the model convergence and the model's behavior in a manner of overfit.

Other References

- Pytorch model code: https://pytorch.org/vision/stable/models.html#object-detection-instance-segmentation-and-person-keypoint-detection
- FasterRCNN tutorials:
    - https://blog.francium.tech/object-detection-with-faster-rcnn-bc2e4295bf49
    - https://johschmidt42.medium.com/train-your-own-object-detector-with-faster-rcnn-pytorch-8d3c759cfc70
    - https://fractaldle.medium.com/guide-to-build-faster-rcnn-in-pytorch-95b10c273439