Chest X-ray Abnormalities Detection- VINAI Kaggle Competition

1. Introduction and related work

The chest X-ray abnormalities detection is a machine learning competition featured on Kaggle by VINAI, which ended in March 2021 [1]. In this competition, participants will have to automatically classify and localize 14 types of thoracic abnormalities from chest radiographs. The dataset consists of 18,000 chest scans (DICOM format) that have been annotated by experience radiologists. The dataset has been divided into 15,000 independently labeled training images and 3,000 test images. These annotations were collected via a web-based platform, Vinlab. Details on building this dataset can be found in this paper [2]. The 14 critical radiographic findings are: 0 – Aortic enlargement, 1 – Atelectasis, 2 – Calcification, 3 – Cardiomegaly, 4 – Consolidation, 5 – ILD, 6 – Infiltration, 7 – Lung Opacity, 8 – Nodule/Mass, 9 – Other lesion, 10 – Pleural effusion, 11 - Pleural thickening, 12 – Pneumothorax, 13 – Pulmonary fibrosis. There is also another class 14 for "No finding". The training files also have a train.csv file that contains trainset metadata, with one row for each object with columns such as unique image identifier: image_id, the name of the class of detected object: class_name, the ID of the class of detected object: class_id, rad_id, x_min, y_min, x_max, y_max.

This is an object detection and classification problem. Many models have been developed in recent years for this type of task: a series model based on region proposal with RCNN (by Ross Girshick et al [3]), Fast R-CNN (also by Ross Girshick [4]) and Faster R-CNN (by Shaoqing Ren et al [5]). These models only look at a region of the image which has high probabilities of containing the object. In contrast, another type of model and its series called YOLO (You only look once [6]) that look at the entire image and split it into grid lines and predict the bounding boxes and the class probabilities of these boxes. YOLO models offer faster processing magnitude with 45 frames per second but have limitation detecting small objects within image. Another object detection model is SSD [7] that offered a good balance between accuracy and speed. Other inferior models are HOG (Histogram of Oriented Gradients by Navneet Dalal and Bill Triggs [8] and Spatial Pyramid Pooling (SPP-net [9]).

2. Methods

For this problem, I will be mainly experimenting with YOLO model.

2.1 Data preprocessing

In this phase, dataset will be pre-processed so that I will be ready for training. Class 14 for "No finding" was dropped from the train.csv file since we do not predict such class. The predicted results appear as is in the image with bounding boxes and predicted class or none. Next, the image dataset in DICOM format was converted into JPG format since the YOLO model only accept some common image formats. The model has a built-in min-max normalization that accepts straightaway image dataset but I still have to normalize bounding box coordinates which are x_min, y_min, x_max, y_max , height, width and area in train.csv file.

The model also required a list of bounding box features (x_min, y_min, x_max, y_max, w, h, area) incorporated as data frame in X, and class_id in Y. Also, class_id and class_name is concatenated into a class list used for displaying respective class ID, class name during prediction.

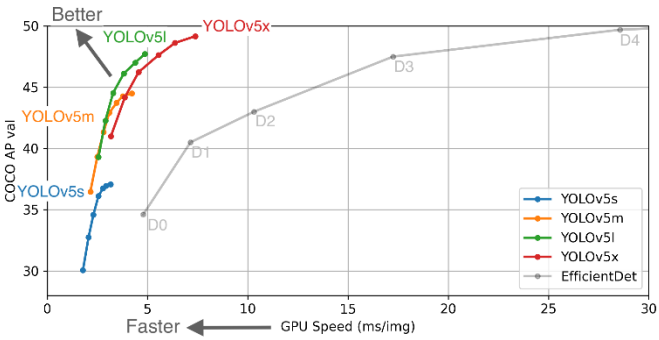
To validate the model performance on the unseen parts of the dataset, it is useful to use K fold cross-validation [10] to split the training set into K folds without repetition and cross-validate the model performance across the K folds. Here I am using GroupKfold [11] with K = 5 (number of splits)

The model also incorporated mosaic augmentation [12] which combines four images into four tiles of random ratio.

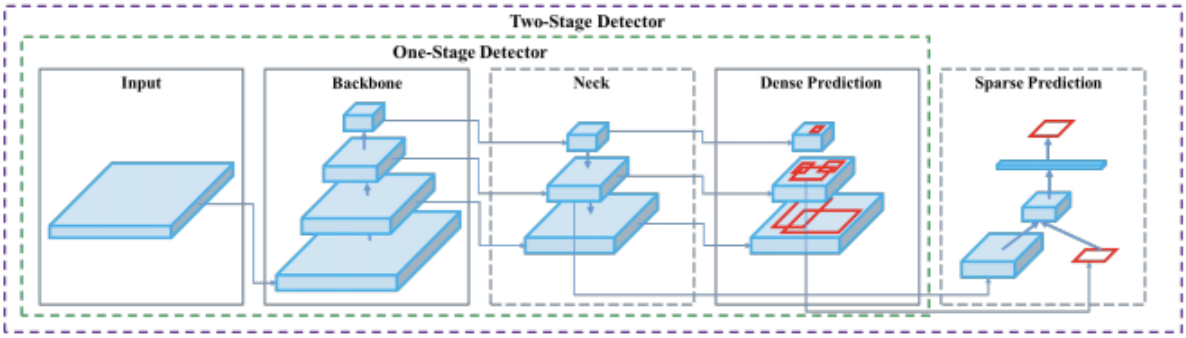
2.2 Model Architecture

For this project, I will be experimenting 2 models of YOLO version 5 developed by Ultralytics [13], one uses transfer learning with previous trained weights on COCO Dataset [14] and one does not use pretrained weights. YOLO version 5 developed by Ultralytics has 5 models: YOLOv5s, YOLOv5m, YOLOv5l and YOLOv5x that have been previously trained on COCO dataset and offer difference Average Precision and GPU speed.

Model	AP^{val}	AP^{test}	AP_{50}	$Speed_{GPU}$	FPS_{GPU}		params	FLOPS
YOLOv5s	37.0	37.0	56.2	**2.4ms**	**476**		7.5M	13.2B
YOLOv5m	44.3	44.3	63.2	3.4ms	333		21.8M	39.4B
YOLOv5l	47.7	47.7	66.5	4.4ms	256		47.8M	88.1B
YOLOv5x	49.2	49.2	67.7	6.9ms	164		89.0M	166.4B
YOLOv5x + TTA	**50.8**	**50.8**	**68.9**	25.5ms	39		89.0M	354.3B



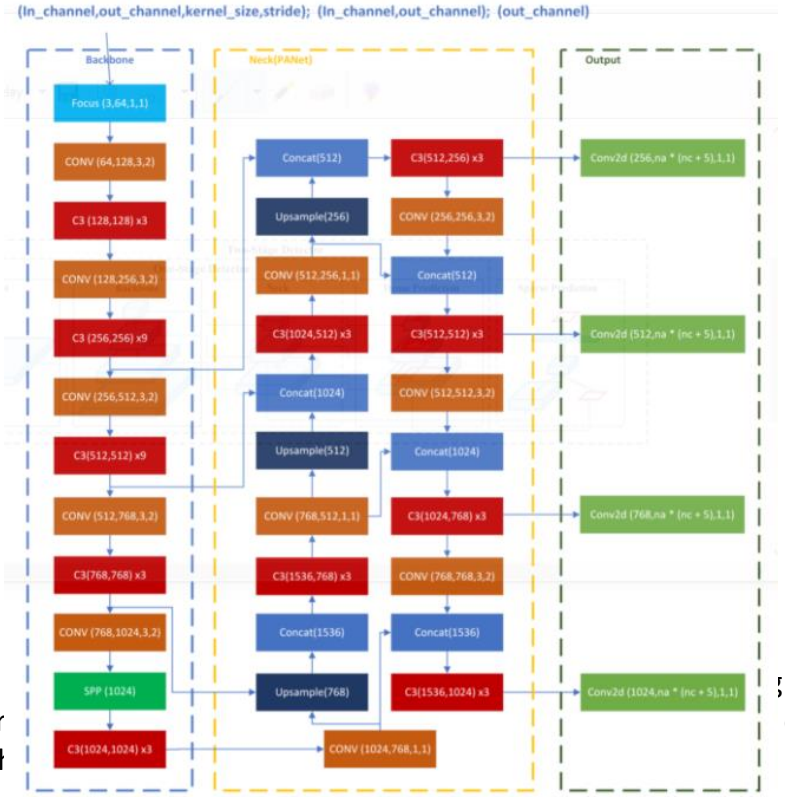
Number of overall parameters increase in YOLOv5s to YOLOv5x. However, the structures of these models stay the same and the only differences are the depth level of CNN layer in each backbone and neck part of the YOLO model.



As shown above [15], YOLO model will have 3 main parts:

1) Backbone: a convolutional neural network model consists of fully convolutional layers, filters, batch normalization and ReLU, used to learn and form feature map at different scales. Usually, a very deep VGG, ResNet, EfficientNet etc is used.
2) Neck: a series of layer to incorporate localization information to learning parameters, typically using FPN (Feature Pyramid Network) or PANet(Path Aggregation Network) etc.
3) Head: dense prediction and sparse prediction are for predicting bounding boxes with highest class probability and choose the best fit based on Non-Max Suppression

The detailed visualization of YOLOv5 layers is as following [16]:



For ; rate 0.01 with a decaying
lear dataset for transfer learning,
witł

For Yolov5l, I also used default hyper-parameters, SGD learning rate 0.01 with decaying learning rate scheduler, without pre-trained weights (training from scratch), batch size 25 and 80 total epoch number.

2.3 Main Challenges

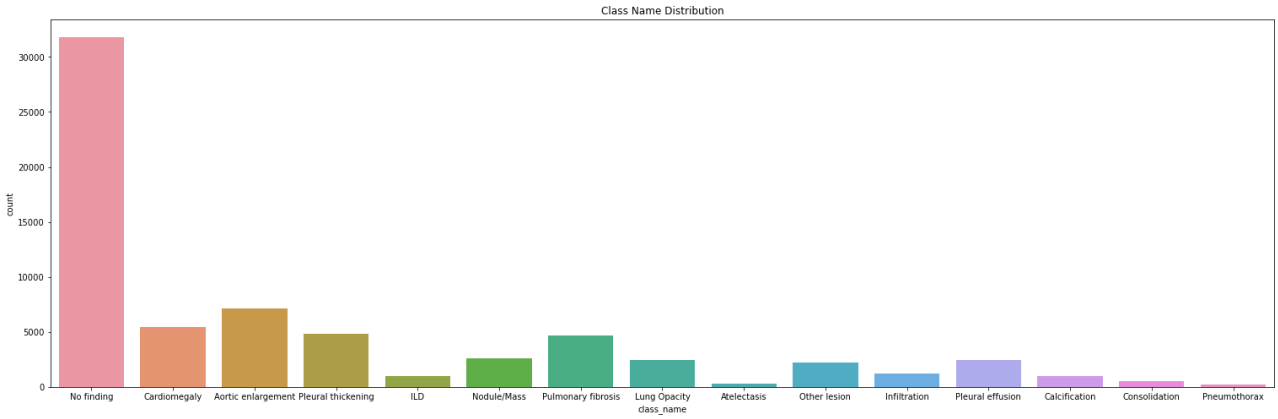
Some of the challenges encountered in this project are:

- Preprocessing the dataset in DICOM format
- Incorporating class names, number of classes, bounding boxes information, train and validation files to meet requirement since the model used yaml [18] built in.
- Limited CPU and GPU computation capability

Some of resolution that I have performed:

- Manually converted dataset into jpg
- Incorporating all class names, number of classes, bounding boxes information, feature lists into train_df and write this train_df into file and split it into train.txt and val.txt and then combined them into a data dict with train, val, number of class, class names, then dumb all information into yaml
- Use models that have less params so that the available resources (CPU and GPU) can accommodate.

The main challenge was how to achieve the highest accuracy as possible after training with limited computational power and with heavily biased dataset as below:



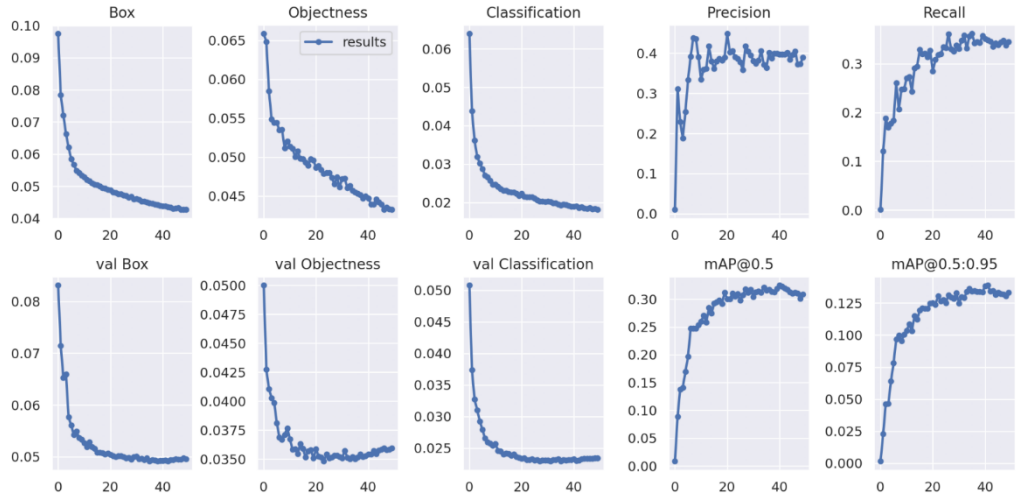
The majority of the distribution was with no-finding and there are some classes that have extremely low number of samples such as Atelectasis and Pneumothorax. Also, it is common sense that some of the diseases only appear in certain location of the image, particularly the lung area. The remedy that I used was using the built-in mosaic data augmentation, which does not required implementation and specifying.

2.4 Results and Discussions

Training took a few hours to complete. The results obtained as below:

- YOLOv5s with pretrained weights:

```
wandb: Run summary:
wandb:                  _step 654
wandb:               _runtime 4405
wandb:             _timestamp 1619278089
wandb:       train/box_loss 0.04278
wandb:       train/obj_loss 0.04328
wandb:       train/cls_loss 0.01812
wandb:    metrics/precision 0.38964
wandb:       metrics/recall 0.34456
wandb:      metrics/mAP_0.5 0.30844
wandb: metrics/mAP_0.5:0.95 0.13331
wandb:         val/box_loss 0.0496
wandb:         val/obj_loss 0.03592
wandb:         val/cls_loss 0.02338
wandb:                x/lr0 0.00203
wandb:                x/lr1 0.00203
wandb:                x/lr2 0.00203
```

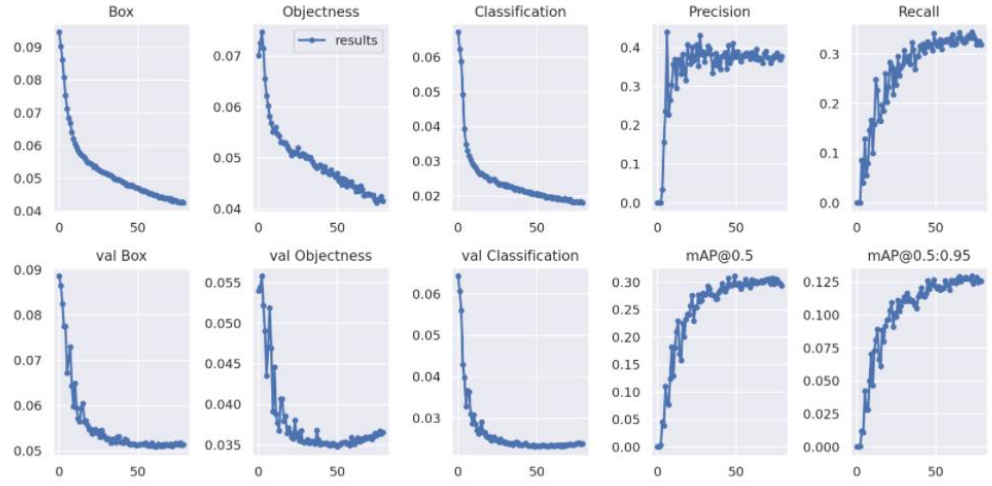



```
               Class      Images     Targets           P           R      mAP@.5  mAP@.5:.95: 100%|██████████| 55/55 [00:13<0
0:00,  3.96it/s]
                 all         879    7.22e+03       0.449       0.284         0.3       0.125
     21/49     4.51G    0.0481   0.04889   0.02158    0.1186         157         640: 100%|██████████| 220/220 [01:05<00:00,  3.3
3it/s]
```

The mAP (mean average precision) after training is 0.3. The best precision obtained is 0.449 at epoch 21/49, also with recall 0.284.

- YOLOv5l without pretrained weights:

```
wandb: Run summary:
wandb:                  _step 1044
wandb:               _runtime 16054
wandb:             _timestamp 1619309025
wandb:       train/box_loss 0.04245
wandb:       train/obj_loss 0.04148
wandb:       train/cls_loss 0.01799
wandb:    metrics/precision 0.37665
wandb:       metrics/recall 0.31839
wandb:      metrics/mAP_0.5 0.29363
wandb: metrics/mAP_0.5:0.95 0.12567
wandb:         val/box_loss 0.05143
wandb:         val/obj_loss 0.03654
wandb:         val/cls_loss 0.02383
wandb:                x/lr0 0.00201
wandb:                x/lr1 0.00201
wandb:                x/lr2 0.00201
```




```
               Class      Images     Targets           P           R      mAP@.5  mAP@.5:.95: 100%|██████████| 36/36 [00:22<0
0:00,  1.63it/s]
                 all         879    7.22e+03        0.43       0.256       0.265       0.103
     28/79     14.1G   0.05174   0.05081   0.02319    0.1257         183         640: 100%|██████████| 141/141 [02:51<00:00,  1.2
1s/it]
```

The best mAP obtained after training is 0.29. Best precision is 0.43 and recall 0.256 at epoch 28/79.

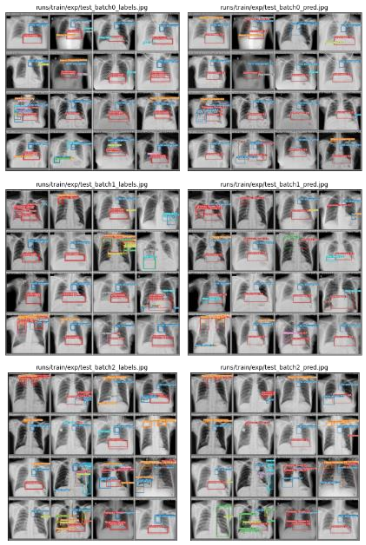
The model trainings achieved low accuracy mainly due two 2 reasons:

- Imbalanced data set whereas there is class with overwhelming number of samples and some classes have relatively small number of samples.
- YOLO may have trouble with detecting tiny objects.
- The model may not have been trained long enough due to limited computational resources.

Comparing to other competitor performances [19]:

#	△pub	Team Name	Notebook	Team Members	Score	Entries	Last
1	▲1	$\mathcal{MS}^2$F			0.314	366	1mo
2	▼1	SZI			0.307	381	1mo
3	—	scumed			0.305	448	1mo
4	▲39	fantastic_hirarin			0.303	68	1mo
5	▲40	Watercooled			0.299	128	1mo
6	▲68	Guanshuo Xu			0.297	8	1mo
7	▲16	CSM			0.296	313	1mo
8	▼1	lung poem			0.295	212	1mo
9	▲5	Azat Davletshin			0.294	54	2mo
10	▼4	Kiet Chu			0.293	140	1mo

Most other competitors achieved a mAP of around 0.3. So, my experimentation results are still relatively acceptable, although it is not at par with radiologists. Some prediction samples are as followings:



2.5 Conclusion and Future Work

In this project, I have experimented the lung abnormalities object detection problem with two models of YOLOv5 and have trained the two models. The models achieved an average

performance comparing to other competitor performances, however, still not outperform radiologists. As previously mentioned, one of the main problems is the underbalanced dataset and the other problem is limitation of YOLO model in detecting small objects, although it offered fast computation with less resource. Nevertheless, there are still room to improve performance, perhaps, by experimenting with different backbone CNN networks in YOLO, or experimenting with Faster-RCNN but may consumer more computation resource and take longer to train, as well as trying other data augmentation technique.

APPENDIX

Notebook can be viewed online on [20]

A) Installing Dependencies and Data Preprocessing

```
!pip install --upgrade seaborn
```

+ Code + Markdown

```
import numpy as np, pandas as pd
from glob import glob
import shutil, os
import matplotlib.pyplot as plt
from sklearn.model_selection import GroupKFold
from tqdm.notebook import tqdm
import seaborn as sns
```

+ Code + Markdown

```
dim = 'original' #512, 256, 'original'
fold = 4
```

```
train_df = pd.read_csv(f'../input/vinbigdata-{dim}-image-dataset/vinbigdata/train.csv')
train_df.head()
```

```
train_df['image_path'] = f'/kaggle/input/vinbigdata-{dim}-image-dataset/vinbigdata/train/'+train_df.image_id+('.png' if dim!='original' else '.jpg')
train_df.head()
```

```
train_df = train_df[train_df.class_id!=14].reset_index(drop = True)
```

```
train_df['x_min'] = train_df.apply(lambda row: (row.x_min)/row.width, axis =1)
train_df['y_min'] = train_df.apply(lambda row: (row.y_min)/row.height, axis =1)

train_df['x_max'] = train_df.apply(lambda row: (row.x_max)/row.width, axis =1)
train_df['y_max'] = train_df.apply(lambda row: (row.y_max)/row.height, axis =1)

train_df['x_mid'] = train_df.apply(lambda row: (row.x_max+row.x_min)/2, axis =1)
train_df['y_mid'] = train_df.apply(lambda row: (row.y_max+row.y_min)/2, axis =1)

train_df['w'] = train_df.apply(lambda row: (row.x_max-row.x_min), axis =1)
train_df['h'] = train_df.apply(lambda row: (row.y_max-row.y_min), axis =1)

train_df['area'] = train_df['w']*train_df['h']
train_df.head()
```

```
features = ['x_min', 'y_min', 'x_max', 'y_max', 'x_mid', 'y_mid', 'w', 'h', 'area']
X = train_df[features]
y = train_df['class_id']
X.shape, y.shape
```

```
class_ids, class_names = list(zip(*set(zip(train_df.class_id, train_df.class_name))))
classes = list(np.array(class_names)[np.argsort(class_ids)])
classes = list(map(lambda x: str(x), classes))
classes
```

B) K Fold Cross Validation and preparing file

```
gkf  = GroupKFold(n_splits = 5)
train_df['fold'] = -1
for fold, (train_idx, val_idx) in enumerate(gkf.split(train_df, groups = train_df.image_id.tolist())):
    train_df.loc[val_idx, 'fold'] = fold
train_df.head()
```

+ Code + Markdown

```
train_files = []
val_files   = []
val_files += list(train_df[train_df.fold==fold].image_path.unique())
train_files += list(train_df[train_df.fold!=fold].image_path.unique())
len(train_files), len(val_files)
```

```
class_ids, class_names = list(zip(*set(zip(train_df.class_id, train_df.class_name))))
classes = list(np.array(class_names)[np.argsort(class_ids)])
classes = list(map(lambda x: str(x), classes))
classes
```

C) YOLOv5 preparation and training

```
from os import listdir
from os.path import isfile, join
import yaml

cwd = '/kaggle/working/'

with open(join( cwd , 'train.txt'), 'w') as f:
    for path in glob('/kaggle/working/vinbigdata/images/train/*'):
        f.write(path+'\n')

with open(join( cwd , 'val.txt'), 'w') as f:
    for path in glob('/kaggle/working/vinbigdata/images/val/*'):
        f.write(path+'\n')

data = dict(
    train =  join( cwd , 'train.txt') ,
    val   =  join( cwd , 'val.txt' ),
    nc    = 14,
    names = classes
    )

with open(join( cwd , 'vinbigdata.yaml'), 'w') as outfile:
    yaml.dump(data, outfile, default_flow_style=False)

f = open(join( cwd , 'vinbigdata.yaml'), 'r')
print('\nyaml:')
print(f.read())
```

```
# https://www.kaggle.com/ultralytics/yolov5
# !git clone https://github.com/ultralytics/yolov5  # clone repo
# %cd yolov5
shutil.copytree('/kaggle/input/yolov5-official-v31-dataset/yolov5', '/kaggle/working/yolov5')
os.chdir('/kaggle/working/yolov5')
# %pip install -qr requirements.txt # install dependencies

import torch
from IPython.display import Image, clear_output  # to display images

clear_output()
print('Setup complete. Using torch %s %s' % (torch.__version__, torch.cuda.get_device_properties(0) if torch.cuda.is_available() else 'CPU'))
```

```
# !WANDB_MODE="dryrun" python train.py --img 640 --batch 16 --epochs 3 --data coco128.yaml --weights yolov5s.pt --nosave --cache
!WANDB_MODE="dryrun" python train.py --img 640 --batch 25 --epochs 80 --data /kaggle/working/vinbigdata.yaml --cfg models/yolov5l.yaml --cache

#!WANDB_MODE="dryrun" python train.py --img 640 --batch 25 --epochs 80 --data /kaggle/working/vinbigdata.yaml --weights yolov5l.pt --cache
#!WANDB_MODE="dryrun" python train.py --img 640 --batch 18 --epochs 50 --data /kaggle/working/vinbigdata.yaml --weights yolov5x.pt --cache
```

D) Getting results

```python
plt.figure(figsize = (15, 15))
plt.imshow(plt.imread('runs/train/exp/train_batch0.jpg'))

plt.figure(figsize = (15, 15))
plt.imshow(plt.imread('runs/train/exp/train_batch1.jpg'))

plt.figure(figsize = (15, 15))
plt.imshow(plt.imread('runs/train/exp/train_batch2.jpg'))
```

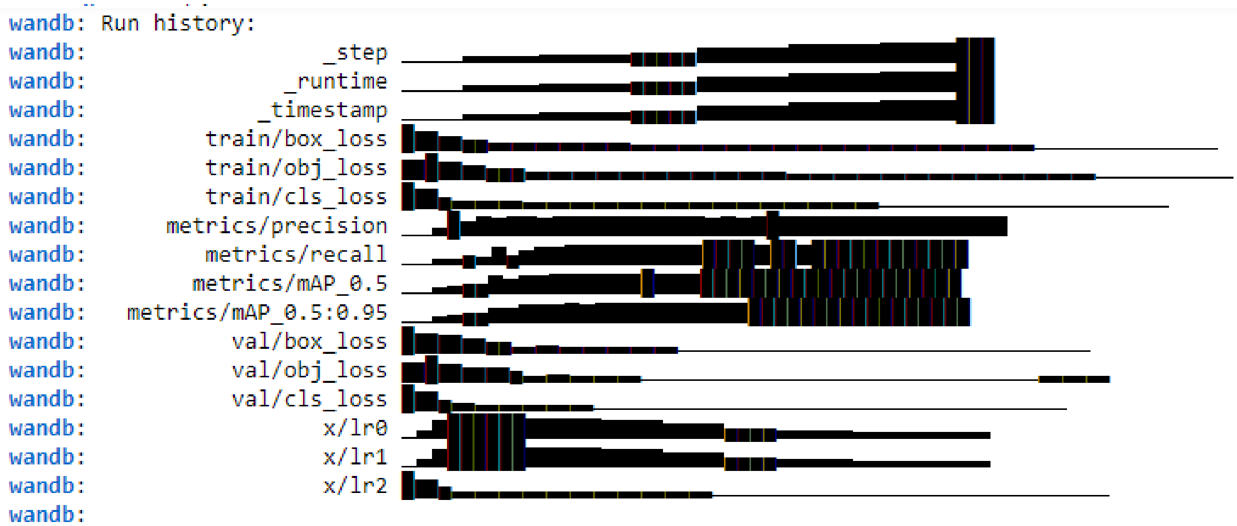
+ Code + Markdown

```python
plt.figure(figsize=(30,15))
plt.axis('off')
plt.imshow(plt.imread('runs/train/exp/results.png'));
```

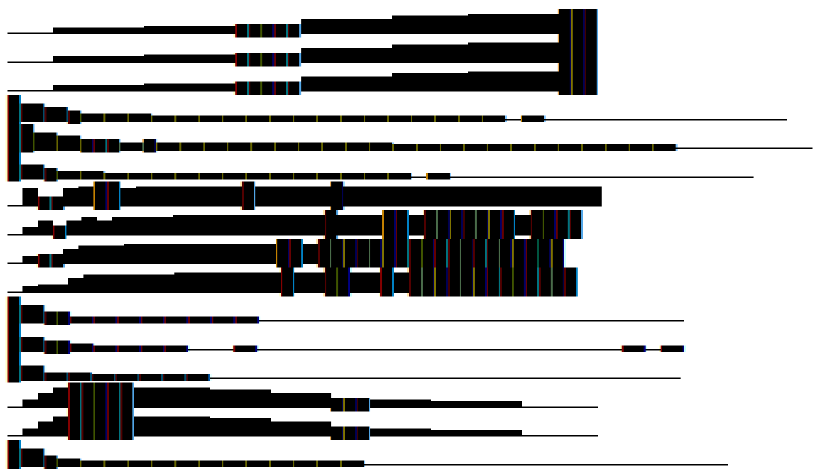
```python
plt.figure(figsize=(30,15))
plt.axis('off')
plt.imshow(plt.imread('runs/train/exp/confusion_matrix.png'));
```

E) Training Logs

```
wandb: Run summary:
wandb:                  _step 1044
wandb:               _runtime 16054
wandb:             _timestamp 1619309025
wandb:         train/box_loss 0.04245
wandb:         train/obj_loss 0.04148
wandb:         train/cls_loss 0.01799
wandb:      metrics/precision 0.37665
wandb:         metrics/recall 0.31839
wandb:        metrics/mAP_0.5 0.29363
wandb:   metrics/mAP_0.5:0.95 0.12567
wandb:           val/box_loss 0.05143
wandb:           val/obj_loss 0.03654
wandb:           val/cls_loss 0.02383
wandb:                 x/lr0 0.00201
wandb:                 x/lr1 0.00201
wandb:                 x/lr2 0.00201
```



```
wandb: Run summary:
wandb:                  _step 654
wandb:               _runtime 4405
wandb:             _timestamp 1619278089
wandb:         train/box_loss 0.04278
wandb:         train/obj_loss 0.04328
wandb:         train/cls_loss 0.01812
wandb:      metrics/precision 0.38964
wandb:         metrics/recall 0.34456
wandb:        metrics/mAP_0.5 0.30844
wandb:   metrics/mAP_0.5:0.95 0.13331
wandb:           val/box_loss 0.0496
wandb:           val/obj_loss 0.03592
wandb:           val/cls_loss 0.02338
wandb:                  x/lr0 0.00203
wandb:                  x/lr1 0.00203
wandb:                  x/lr2 0.00203
wandb: Run history:
wandb:                  _step
wandb:               _runtime
wandb:             _timestamp
wandb:         train/box_loss
wandb:         train/obj_loss
wandb:         train/cls_loss
wandb:      metrics/precision
wandb:         metrics/recall
wandb:        metrics/mAP_0.5
wandb:   metrics/mAP_0.5:0.95
wandb:           val/box_loss
wandb:           val/obj_loss
wandb:           val/cls_loss
wandb:                  x/lr0
wandb:                  x/lr1
wandb:                  x/lr2
```



REFERENCE:

[1] https://www.kaggle.com/c/vinbigdata-chest-xray-abnormalities-detection/overview

[2] Ha Q Nguyen et al, VinDr-CXR: An open dataset of chest X-rays with radiologist's annotations https://arxiv.org/pdf/2012.15029.pdf

[3] Ross Girshick et al. Rich feature hierarchies for accurate object detection and semantic segmentation. https://arxiv.org/pdf/1311.2524.pdf

[4] Ross Girshick. Fast RCNN. https://arxiv.org/pdf/1504.08083.pdf

[5] Shaoqing Ren et al. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks

[6] Joseph Redmon et al. You Only Look Once: Unified, Real-Time Object Detection. https://arxiv.org/pdf/1506.02640.pdf

[7] Wei Liu et al. SSD: Single Shot MultiBox Detector. https://arxiv.org/abs/1512.02325

[8] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)

[9] Kaiming He et al. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. https://arxiv.org/abs/1406.4729

[10] Gareth James. An Introduction to Statistical Learning: with Applications in R (Springer Texts in Statistics) 1st ed. 2013, Corr. 7th printing 2017 Edition

[11] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GroupKFold.html

[12] Yukang Chen. Dynamic Scale Training for Object Detection. https://arxiv.org/pdf/2004.12432.pdf

[13] https://github.com/ultralytics/yolov5/releases/tag/v3.0

[14] Tsung-Yi Lin. Microsoft COCO: Common Objects in Context. https://arxiv.org/pdf/1405.0312.pdf

[15] Alexey Bochkovskiy et al. YOLOv4: Optimal Speed and Accuracy of Object Detection. https://arxiv.org/abs/2004.10934

[16] https://blog.csdn.net/Q1u1NG/article/details/107511465

[17] Sebastian Ruder. An overview of gradient descent optimization algorithms. https://arxiv.org/pdf/1609.04747.pdf

[18] https://yaml.org/

[19] https://www.kaggle.com/c/vinbigdata-chest-xray-abnormalities-detection/leaderboard

[20] https://www.kaggle.com/phamhoad/vinbigdata-project