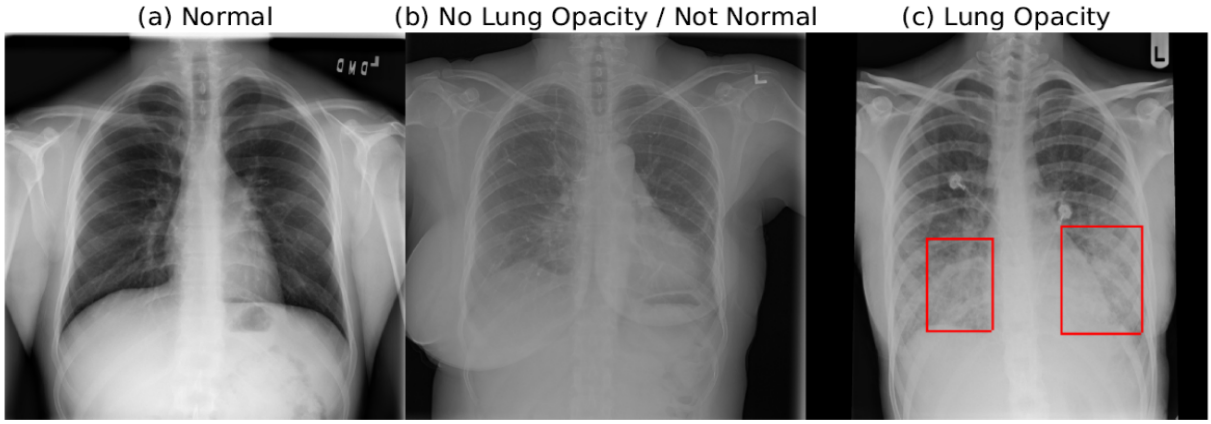
Final Project: Intelligent Analysis of Biomedical Images

Mahtab Bigverdi (96105604)

February 17, 2021

Goal

- Pneumonia accounts for over 15% of all deaths of children under 5 years old internationally. In 2015, 920,000 children under the age of 5 died from the disease. In the United States, pneumonia accounts for over 500,000 visits to emergency departments and over 50,000 deaths in 2015, keeping the ailment on the list of top 10 causes of death in the country.
- While common, accurately diagnosing pneumonia is a tall order. It requires review of a chest radiograph (CXR) by highly trained specialists and confirmation through clinical history, vital signs and laboratory exams. Pneumonia usually manifests as an area or areas of increased opacity on CXR. However, the diagnosis of pneumonia on CXR is complicated because of a number of other conditions in the lungs such as fluid overload (pulmonary edema), bleeding, volume loss (atelectasis or collapse), lung cancer, or post-radiation or surgical changes. Outside of the lungs, fluid in the pleural space (pleural effusion) also appears as increased opacity on CXR. When available, comparison of CXRs of the patient taken at different time points and correlation with clinical symptoms and history are helpful in making the diagnosis.
- CXRs are the most commonly performed diagnostic imaging study. A number of factors such as positioning of the patient and depth of inspiration can alter the appearance of the CXR, complicating interpretation further. In addition, clinicians are faced with reading high volumes of images every shift.
- **In this task we want to detect pneumonia from a kaggle dataset. (Object Detection Task)**



1 Data

- Downloading using cookies from kaggle and wget command!

```
[1] from google.colab import files
    files.upload()
```

```
Choose Files  cookies.txt
• cookies.txt(text/plain) - 2854 bytes, last modified: 2/11/2021 - 100% done
Saving cookies.txt to cookies.txt
{'cookies.txt': b'# Netscape HTTP Cookie File\n# http://curl.haxx.se/rfc/cookie_spec.html\n# This is a generated file!  Do not edit.\n\nwww.kaggle.com\tFALSE
```

```
!wget -x --load-cookies cookies.txt "https://www.kaggle.com/c/10338/download-all" -O data.zip
```

```
--2021-02-11 15:38:24--  https://www.kaggle.com/c/10338/download-all
Resolving www.kaggle.com (www.kaggle.com)... 35.244.233.98
Connecting to www.kaggle.com (www.kaggle.com)|35.244.233.98|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://storage.googleapis.com/kaggle-competitions-data/kaggle-v2/10338/862042/bundle/archive.zip?GoogleAccessId=web-data@kaggle-161607.iam.gservic
--2021-02-11 15:38:24--  https://storage.googleapis.com/kaggle-competitions-data/kaggle-v2/10338/862042/bundle/archive.zip?GoogleAccessId=web-data@kaggle-161
Resolving storage.googleapis.com (storage.googleapis.com)... 108.177.126.128, 108.177.127.128, 108.177.119.128, ...
Connecting to storage.googleapis.com (storage.googleapis.com)|108.177.126.128|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3932287530 (3.7G) [application/zip]
Saving to: 'data.zip'

data.zip            100%[===================>]   3.66G  60.0MB/s    in 73s

2021-02-11 15:39:38 (51.4 MB/s) - 'data.zip' saved [3932287530/3932287530]
```

- **Cross-Validation**
  I used **sklearn model_selection StratifiedKFold** for spliting train data to folds. This cross-validation object is a variation of KFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

```
def create_folds(df: pd.DataFrame, X: pd.DataFrame, y: pd.DataFrame, nb_folds: int,
if_save: bool) -> pd.DataFrame:
df["fold"] = -1
skf = StratifiedKFold(n_splits=nb_folds, shuffle=True, random_state=42)
for fold, (train_index, test_index) in enumerate(skf.split(X, y)):
    df.loc[test_index, "fold"] = fold
if if_save:
    df.to_csv(os.path.join(DATA_DIR, "folds.csv"), index=False)
return df
```

1.1 Augmentation

In the article heavy augmentation has following attributes:(two libraries were very useful in this section **1.skimage.transform 2.imgaug.augmenters**)

Examples: Basics

A standard use case

The following example shows a standard use case. An augmentation sequence (crop + horizontal flips + gaussian blur) is defined once at the start of the script. Then many batches are loaded and augmented before being used for training.

```
from imgaug import augmenters as iaa

seq = iaa.Sequential([
    iaa.Crop(px=(0, 16)), # crop images from each side by 0 to 16px (randomly chosen)
    iaa.Fliplr(0.5), # horizontally flip 50% of the images
    iaa.GaussianBlur(sigma=(0, 3.0)) # blur images with a sigma of 0 to 3.0
])

for batch_idx in range(1000):
    # 'images' should be either a 4D numpy array of shape (N, height, width, channels)
    # or a list of 3D numpy arrays, each having shape (height, width, channels).
    # Grayscale images must have shape (height, width, 1) each.
    # All images must have numpy's dtype uint8. Values are expected to be in
    # range 0-255.
    images = load_batch(batch_idx)
    images_aug = seq(images=images)
    train_on_images(images_aug)
```

AffineTransform

class skimage.transform. **AffineTransform**(*matrix=None, scale=None, rotation=None, shear=None, translation=None*) [source]

Bases: skimage.transform._geometric.ProjectiveTransform

2D affine transformation.

Has the following form:

```
X = a0*x + a1*y + a2 =
  = sx*x*cos(rotation) - sy*y*sin(rotation + shear) + a2

Y = b0*x + b1*y + b2 =
  = sx*x*sin(rotation) + sy*y*cos(rotation + shear) + b2
```

where `sx` and `sy` are scale factors in the x and y directions, and the homogeneous transformation matrix is:

```
[[a0  a1  a2]
 [b0  b1  b2]
 [0   0    1]]
```

Parameters

matrix : (3, 3) array, optional
Homogeneous transformation matrix.

scale : (s as float or (sx, sy) as array, list or tuple), optional
Scale factor(s). If a single value, it will be assigned to both sx and sy.
New in version 0.17: Added support for supplying a single scalar value.

rotation : float, optional
Rotation angle in counter-clockwise direction as radians.

shear : float, optional
Shear angle in counter-clockwise direction as radians.

translation : (tx, ty) as array, list or tuple, optional
Translation parameters.

1. mild rotations (up to 6 degrees) and shear

```
skimage.transform.AffineTransform(rotation=self.angle * math.pi / 180,
shear=self.shear * math.pi / 180)
```

2. shift, scale

```
skimage.transform.AffineTransform(translation=(self.src_center_x, self.src_center_y)
skimage.transform.AffineTransform(scale=(1.0 / self.scale_x, 1.0 / self.scale_y))
skimage.transform.AffineTransform(translation=(-self.crop_size / 2, -self.crop_size
```

3. horizontal flip

```
if self.hflip:
    scale_x *= -1
```

4. for some images random level of blur, noise and gamma changes

5. a limited the amount of brightness / gamma augmentations

```
iaa.Sometimes(0.1, iaa.CoarseSaltAndPepper(p=(0.01, 0.01), size_percent=(0.1, 0.2)))
iaa.Sometimes(0.5, iaa.GaussianBlur(sigma=(0.0, 2.0))),
iaa.Sometimes(0.5, iaa.AdditiveGaussianNoise(scale=(0, 0.04 * 255))),
```

6. custom rotations
Instead of rotating the corners we rotated two points at each edge, at 1/3 and 2/3 edge length from the corner (8 points in total), and calculated the new bounding box as min/max of the rotated points.
Generating 8 points for each bounding box of each image:

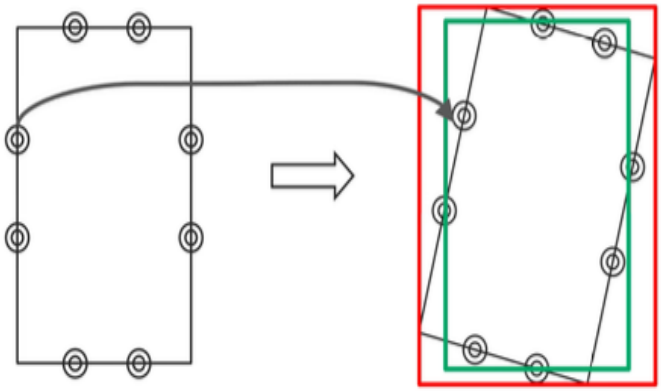


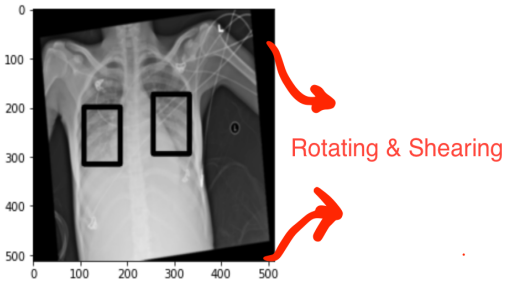
Figure 1: custom rotation (Green bounding box)

```
points = np.array(
    [
        [x, y + h / 3],
        [x, y + h * 2 / 3],
        [x + w, y + h / 3],
        [x + w, y + h * 2 / 3],
        [x + w / 3, y],
        [x + w * 2 / 3, y],
        [x + w / 3, y + h],
        [x + w * 2 / 3, y + h],
    ]
    )
    self.annotations[patient_id].append(points)
```

After this we will use **Inverse function of skimage transform** and apply it on our 8 coordinates and get new and transformed points.

```
points = cfg.transform().inverse(annotation)
res = np.zeros((1, 5))
p0 = np.min(points, axis=0)
p1 = np.max(points, axis=0)
res[0, 0:2] = p0
res[0, 2:4] = p1
```

Some examples of augmentation which were not in the paper:



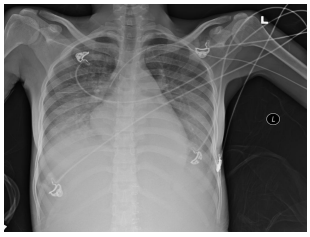


Figure 2: Original Image

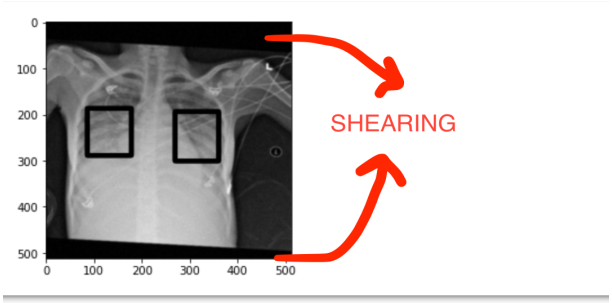


Figure 3: After Augmentation

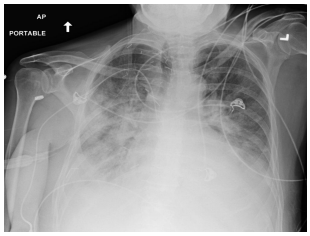


Figure 4: Original Image

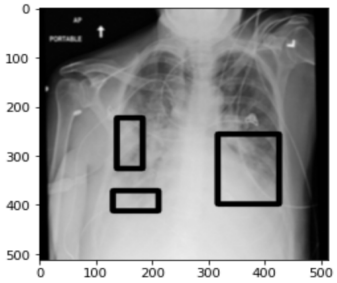


Figure 5: After Augmentation

1.2 Dataset

Images are dicom files, so we used pydicom library.

```
dcm_data = pydicom.read_file(os.path.join(TRAIN_DIR, f"{patient_id}.dcm"))
img = dcm_data.pixel_array
```

Train Augmentation Attributes:

```
scale=0.15, angle=6.0, shear=4.0, gamma=0.25, hflip=np.random.choice([True, False])
```

get_item function in object detection tasks must return annotations too with image and its label. In this task we have just **one class** object

```
target = {}

if len(annotations):
    annotations = np.row_stack(annotations)
else:
    annotations = np.zeros((0, 4))

target["boxes"] = annotations
target["labels"] = [1] * len(annotations)
return crop, target
```

Augmentations for train and validation set are different so I add is_training boolean to handle this.

```
dataset_train = DetectionDataset(
        fold=fold,
        img_size= IMG_SIZE,
        is_training=True,

    )
```

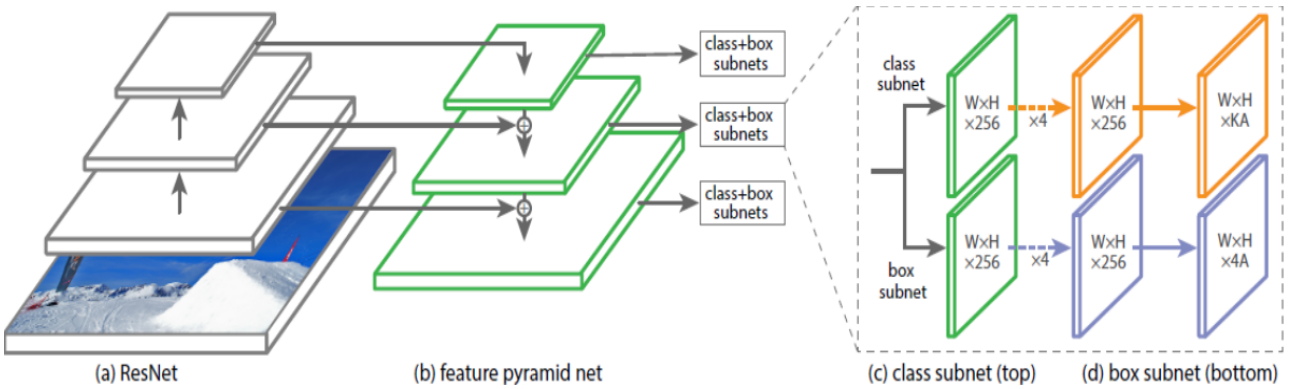
1.3 DataLoader

Each item from dataset is an image and targets (boxes and labels). For this kind of item we should change our dataloader with this collate_fn function.

```
def collate_fn(batch):
    return tuple(zip(*batch))

dataloader_train = DataLoader(
        dataset_train,
        batch_size= 6,
        shuffle=True,
        collate_fn=collate_fn,
        drop_last=True,
    )
```

2 Model

Due to the lack of resources and time, I reduced the problem from both classification and object detection to just **Object detection with RETINANET**.



Details of Model:

- **Backbone : Resnet50 Feature Pyramid Network - pretrained on ImageNet**
- **Whole Model pretrained on COCO dataset**
- **number of classes = 2 (background and main object)**
- **First convolution layer input channels is 1 because images are grayscale.) number of classes = 2 (background and main object)**

RetinaNet

`torchvision.models.detection.retinanet_resnet50_fpn(pretrained=False, progress=True, num_classes=91, pretrained_backbone=True, **kwargs)` [SOURCE]

Constructs a RetinaNet model with a ResNet-50-FPN backbone.

The input to the model is expected to be a list of tensors, each of shape `[C, H, W]`, one for each image, and should be in `0-1` range. Different images can have different sizes.

The behavior of the model changes depending if it is in training or evaluation mode.

During training, the model expects both the input tensors, as well as a targets (list of dictionary), containing:

- boxes (`FloatTensor[N, 4]`): the ground-truth boxes in `[x1, y1, x2, y2]` format, with values between `0` and `H` and `0` and `W`
- labels (`Int64Tensor[N]`): the class label for each ground-truth box

```
import torchvision
from torchvision.models.detection.retinanet import RetinaNetHead

def define_model():
    model = torchvision.models.detection.retinanet_resnet50_fpn
    (pretrained= True, pretrained_backbone= True)

    num_classes = 2  # 1 class  + background

    model.head = RetinaNetHead(in_channels=256, num_anchors=9, num_classes=2)
```

```
    ### input is grayscale with one channel
    model.backbone.body.conv1 = nn.Conv2d(1, 64,
    kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)


    model.transform = torchvision.models.detection.transform
    .GeneralizedRCNNTransform(min_size=(800,),
    max_size=1333, image_mean=[0.485], image_std= [0.229])
    return model
```

```
RetinaNet(
  (backbone): BackboneWithFPN(
    (body): IntermediateLayerGetter(
      (conv1): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
      (bn1): FrozenBatchNorm2d(64)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (bn2): FrozenBatchNorm2d(64)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(256)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): FrozenBatchNorm2d(256)
          )
```

3 Training

3.1 Optimizer

```
optimizer = optim.Adam(model.parameters(), lr=1e-5)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, patience=4, verbose=True, factor=0.2
    )
######## metric is regression loss of validation
if scheduler is not None:
    scheduler.step(np.mean(reg_loss_val_tmp))
```

For learning rate scheduler we used available in Pytorch ReduceLROnPlateau with the patience of 4 and learning rate decrease factor of 0.2.

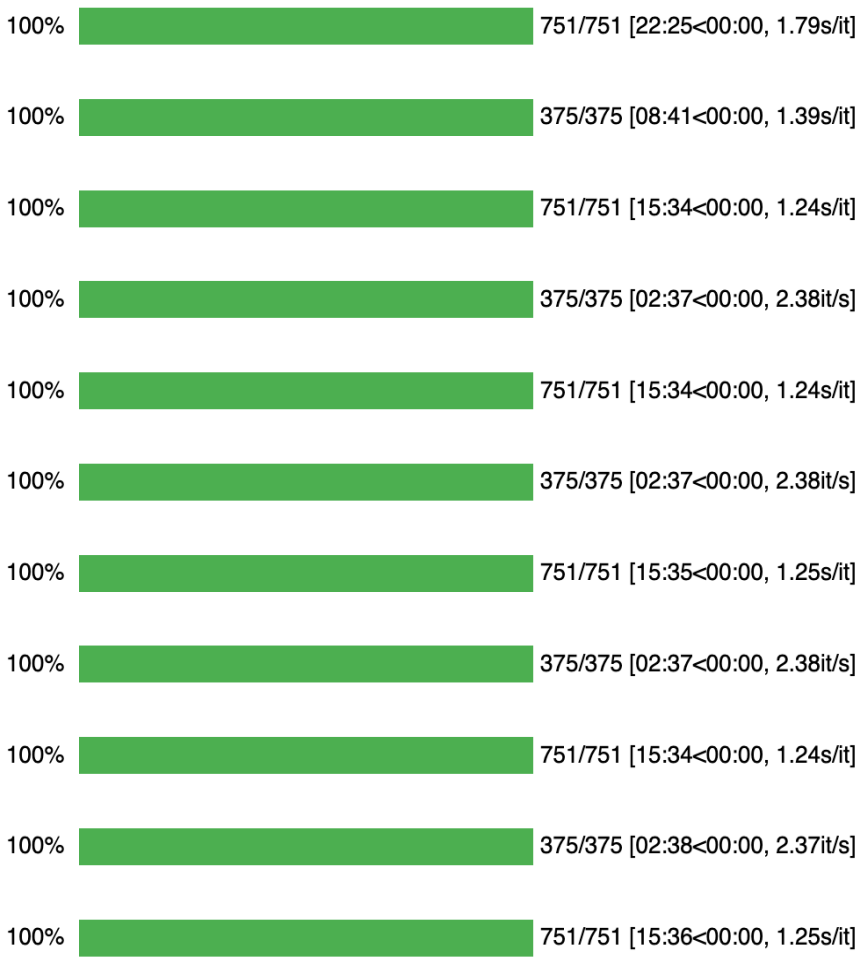
CLASS `torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=10, threshold=0.0001, threshold_mode='rel', cooldown=0, min_lr=0, eps=1e-08, verbose=False)` [SOURCE]

Reduce learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This scheduler reads a metrics quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

Parameter	Description
Optimizer	Adam
Initial learning rate	1e-5
Learning rate scheduler	ReduceLROnPlateau
Patience	4
Image size	512×512

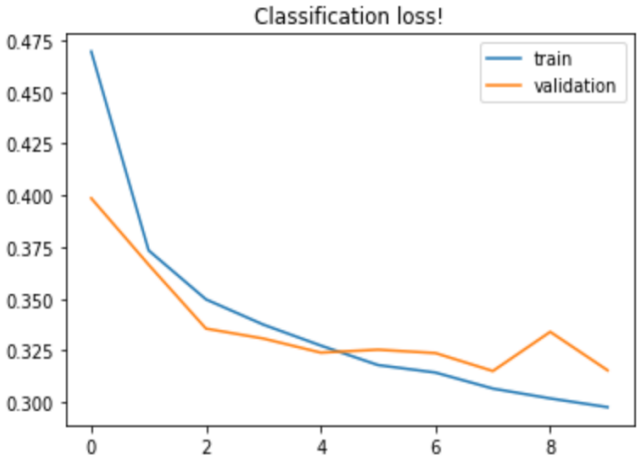
3.2 Details

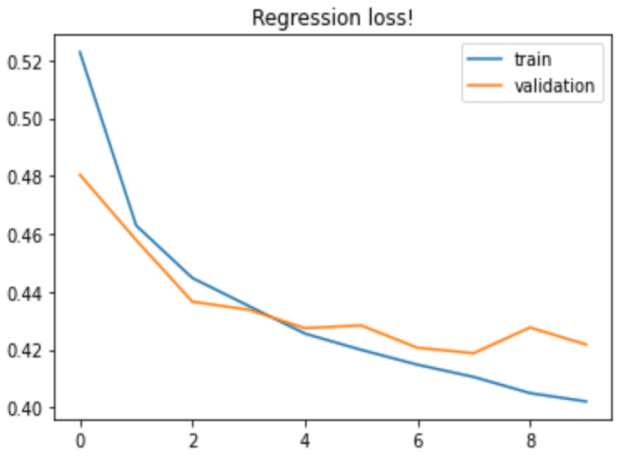
- number of train datapoints in each fold about 7000
- number of validation datapoints in each fold about 2000
- train data loader : batch_size = 6
- test data loader : batch_size = 4
- time of each epoch about 30 minutes
- time of each run for each fold about 3 hours



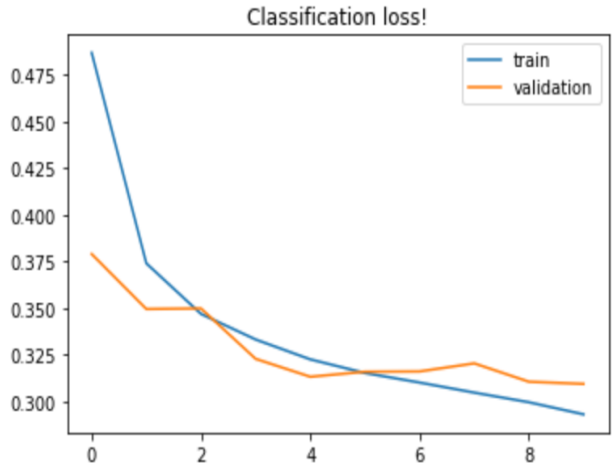
3.3 Results of Each Fold (4 folds)

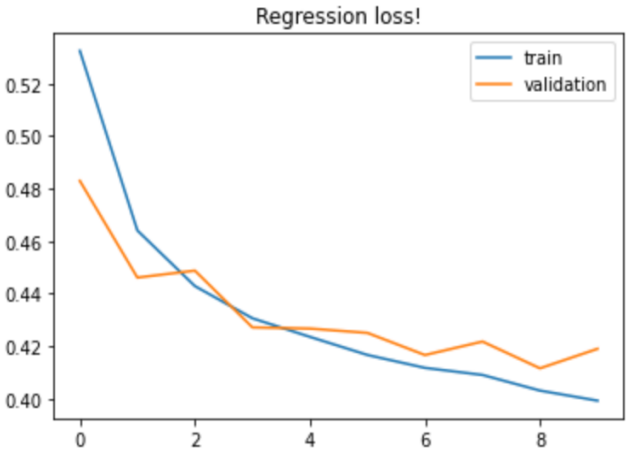
- **Fold 1**



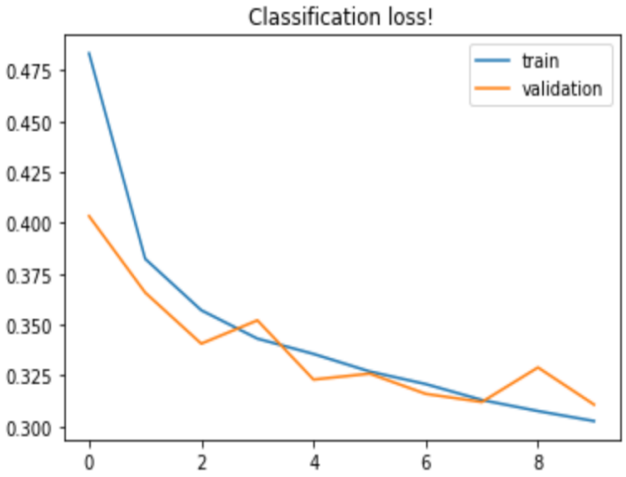


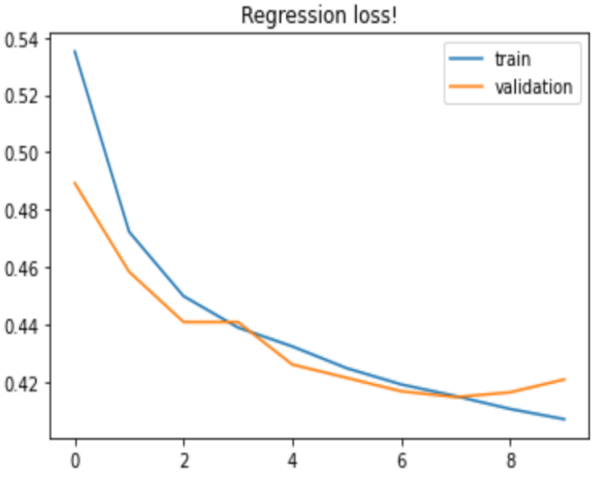
- **Fold 2**





- **Fold 3**





3.4 Comparing Folds with final losses

- **Fold 1**

```
classification loss train:  0.3396131204761138
classification loss validation:  0.3369380010286967
Regression loss train:  0.43430764434261426
Regression loss validation:  0.43528179406325024
```

- **Fold 2**

```
classification loss train:  0.33849684891703924
classification loss validation:  0.3285527082403501
Regression loss train:  0.43329178012322167
Regression loss validation:  0.43252485931714374
```

- **Fold 3**

```
classification loss train:  0.3472403307292814
classification loss validation:  0.3378429267485937
Regression loss train:  0.440568991257411
Regression loss validation:  0.43460278208255765
```

4 Evaluation Metric

- This competition is evaluated on the mean average precision at different intersection over union IoU thresholds. The IoU of a set of predicted bounding boxes and ground truth bounding boxes is calculated as:

$$IoU(A,B) = \frac{A \cap B}{A \cup B}.$$

```
def iou(box1: List[int], box2: List[int]) -> float:
    """
    Args:
        box1, box2: x, y, w, h of the boxes
    Output:
        Intersection over union
    """
    x11, y11, w1, h1 = box1
    x21, y21, w2, h2 = box2
    if w1 * h1 <= 0 or w2 * h2 <= 0:
        return 0
    x12, y12 = x11 + w1, y11 + h1
    x22, y22 = x21 + w2, y21 + h2

    area1, area2 = w1 * h1, w2 * h2
    xi1, yi1, xi2, yi2 = max([x11, x21]), max([y11, y21]), min([x12, x22]), min([y12, y22
    if xi2 <= xi1 or yi2 <= yi1:
        return 0
    else:
        intersect = (xi2 - xi1) * (yi2 - yi1)
        union = area1 + area2 - intersect
        return intersect / union
```

- The metric sweeps over a range of IoU thresholds, at each point calculating an average precision value. The threshold values range from 0.4 to 0.75 with a step size of 0.05: (0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75). In other words, at a threshold of 0.5, a predicted object is considered a "hit" if its intersection over union with a ground truth object is greater than 0.5.

  At each threshold value t, a precision value is calculated based on the number of true positives TP, false negatives FN, and false positives FP resulting from comparing the predicted object to all ground truth objects:

$$\frac{TP(t)}{TP(t) + FP(t) + FN(t)}.$$

- A true positive is counted when a single predicted object matches a ground truth object with an IoU above the threshold. A false positive indicates a predicted object had no associated ground truth object. A false negative indicates a ground truth object had no associated predicted object.

  The average precision of a single image is calculated as the mean of the above precision values at each IoU threshold:

$$\frac{1}{|thresholds|}\sum_{t}\frac{TP(t)}{TP(t)+FP(t)+FN(t)}.$$

```python
def map_iou(boxes_true: np.ndarray, boxes_pred: np.ndarray, scores: np.ndarray, thresholds:
Tuple[float]=(0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75)) -> Optional[float]:
    """
    Mean average precision at differnet intersection over union (IoU) threshold
    Args:
        boxes_true: M x 4 numpy array of ground true bounding boxes of one image.
                    bbox format: (x1, y1, w, h)
        boxes_pred: N x 4 numpy array of predicted bounding boxes of one image.
                    bbox format: (x1, y1, w, h)
        scores:     length N numpy array of scores associated with predicted bboxes
        thresholds: IoU shresholds to evaluate the mean average precision on
    """
    # images with no ground truth bboxes are not included in the map score unless
    # there is a false positive detection
    # return None if both are empty, don't count the image in final evaluation
    if len(boxes_true) == 0 and len(boxes_pred) == 0:
        return None
    if len(boxes_pred):
        assert len(scores) == len(boxes_pred), "boxes_pred and scores should be same length"
        # sort boxes_pred by scores in decreasing order
        boxes_pred = boxes_pred[np.argsort(scores)[::-1], :]
    map_total = 0.0
    # loop over thresholds
    for t in thresholds:
        matched_bt = set()
        tp, fn = 0, 0
        for i, bt in enumerate(boxes_true):
            matched = False
            for j, bp in enumerate(boxes_pred):
                miou = iou(bt, bp)
                if miou >= t and not matched and j not in matched_bt:
                    matched = True
                    tp += 1  # bt is matched for the first time, count as TP
                    matched_bt.add(j)
            if not matched:
                fn += 1  # bt has no match, count as FN

        fp = len(boxes_pred) - len(matched_bt)  # FP is the bp that not matched to any bt
        m = tp / (tp + fn + fp)
        map_total += m
    return map_total / len(thresholds)
```

In the code above we first sort predictions by their scores after that we find their matches with ground truth and then we count tp,fp and fn.

5 Results

Mean Average Precision

- **Fold 0**

```
[21] map_train

     0.0335713092303844

                                    fold 0

[22] map_val

     0.03578645618915741
```

- **Fold 1**

```
[44] map_train

     0.03657988344625176

                                    fold1

[45] map_val

     0.03770988702431744
```

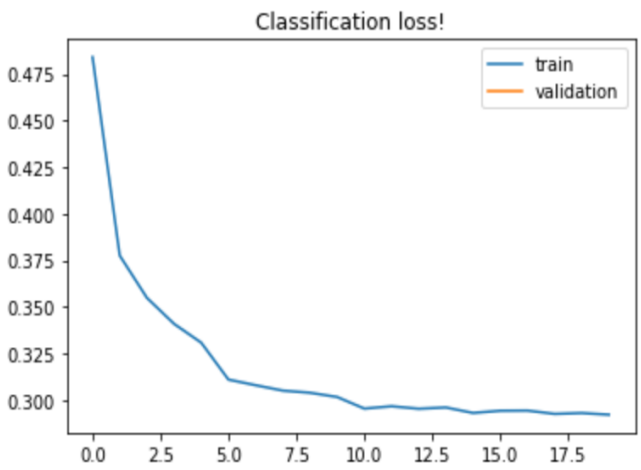
- **Fold 2**

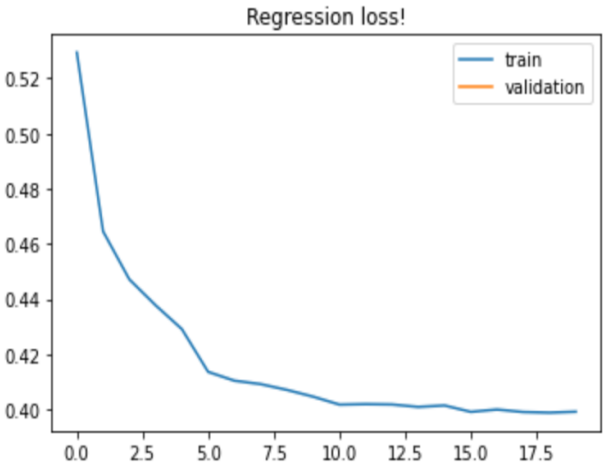
```
     map_train

     0.030923815890919557

                                    fold 2

[33] map_val

     0.03125163252982578
```

- **Fold 3**

```
[48] map_train

     0.03199808864755517

                                    fold3

     map_val

     0.032825864003013
```

As we see above results were not promising and ideas for improving results is changing encoder.





Finally, I tested one fold for 20 epoches, but results didn't change a lot.