Readme

August 1, 2020

1 Overview

This is a CNN project for a dataset consisting of labeled chest x-ray images. There are over 200k x-rays in the dataset, each labeled with 14 findings. The dataset in its raw form is both multi-label and multi-class in that each x-ray can have more than one label, and each of these labels can be one of 4 values (non mentuned, negative, uncertain and positive).

This project will reduce this dataset to about 137k images with 12 binary labels making this project just a multi-label CNN problem.

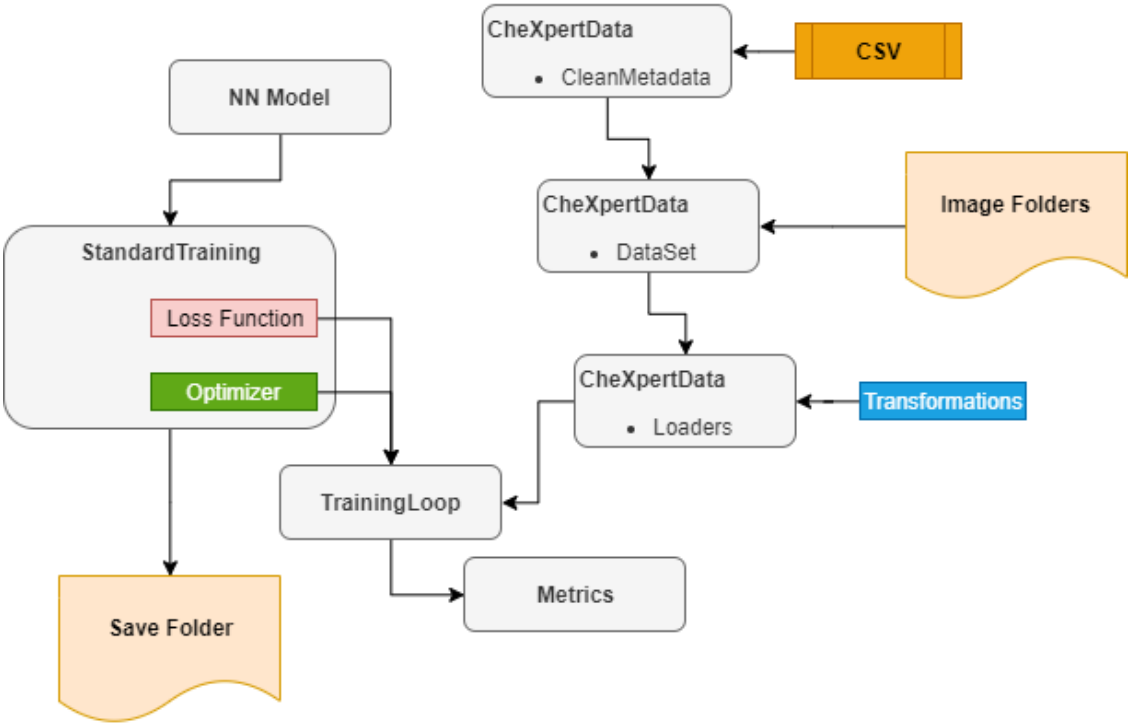
The dataset comes from the Stanford Machine Learning group and can be accessed via this link: https://stanfordmlgroup.github.io/competitions/chexpert/

2 Purpose

The purpose of this project is to build a framework that will easily allow repeatable runs of the dataset with variable parameters, images counts, models, etc. Though it is always desired to produce the best possible accuracy, recall and precision in our predictions, the main goal of this project is to build the framework to more easily allow finetuning of model runs and not specifically finding the best configuration.

3 Framework

Below if a high level diagram of how the framework was built:



4 Goals

4.0.1 Build a Framework for better Understanding

The primary goal of this project is to gain a deeper understanding of CNN and PyTorch. There are already a lot of frameworks out there that do a lot more than what is built into this project, Keras and TensorBoard are some good examples.

But it is very difficult to get a good understanding of what is happening behind the scenes if the work is already done for you. So in this project, preference is given to manual coding over pre-built packages.

The coding mistakes and wrong assumptions made in this journey are invaluable. In fact, these missteps are the key to better understanding. Things like: - Why can't I build the ROC curve from just my actual and predicted labels? - Why is my accuracy over 90% when all my predictions are negative? - Why can't I use pre-trained models with my grayscale images? - How do I get my model's output to match my loss function?

There is no intention of making a framework that is better than ones already available. Chances are for future CNN projects, I will use a lot more pre-built packages. But the hope is the understanding gained with this project will allow me to leverage features and parameters in these packages much more accurately and efficiently.

4.0.2 Code Reuse / Repeatable Model Runs

Jupyter Notebooks are a great way of combining code, visualizations and descriptive text in a single location. But using notebooks for repeatable runs is not the most efficient approach. You

can copy/paste code or entire notebooks, but if you need to make a change, you have to change every notebook.

Building Python modules to house the repeatable code is a much better approach and the route taken with this project.

Notebooks are used for specific runs, i.e. DenseNet vs. ResNet, Learning Rate 1e-4 vs. Learning Rate 1e-6. But since most of the code is in modules, these notebooks (other than a couple of bootstrap blocks) contain only the model and training loop configurations.

4.0.3 Model Goals

Though this project is more focused on understanding and building a framework, it is still important to have a target model goals to shoot for.

The order of importance for the model results are: - Recall - ROC Area Under the Curve - Precision

This is somewhat arbitrary, but the thought is it would be better to suggest a false positive finding to a Radiologist then to miss a possible diagnosis. Radiologists tend to make their own judgments about x-rays, but having a suggested finding might influence the doctor to spend more time analyzing these suggestions.

ROC is always important, but this is more of a true metric of how well the model performed since it is based on probabilities and not predictions. But this might not be something the Radiologist would see.

We can easily make our recall values 100% by always predicting a positive finding. So Recall is of no value if we don't keep the precision under control.

5 Dataset

The raw dataset comes as a set of images along with a csv index file. There is one row in the csv file for each x-ray. There are both train and validation dataset with 223,415 and 234 images respectively. Since the validation set was so small, only the train dataset was used. These 223,415 images were randomly split into train and val during the model runs.

The csv contains some tabular features such as sex and age, aspectes of the image such as orientation and view, and a path for the image file.

The csv also contains the 14 labels: - Enlarged_Cardiomediastinum - Cardiomegaly - Lung_Opacity - Lung_Lesion - Edema - Consolidation - Pneumonia - Atelectasis - Pneumothorax - Pleural_Effusion - Pleural_Other - Fracture - No Finding - Support Devices (i.e. pacemakers)

Any image can have any number of labels making this a multi-label dataset. i.e. An x-ray can show both Pleural Effusion and Cardiomegaly.

Also, each of these labels can have one of 4 values: - Not Mentioned = Null - Negative = -1 - Uncertain = 0 - Positive = 1

How the x-ray was taken is stored in 2 columns, "Frontal/Lateral" and "AP/PA". Lateral x-rays are taken from the patient's side. AP/PA stands for Anterior and Posterior. AP means the x-rays entered the patient's from and exited out of the patient's back and PA is the opposite.

From the image path, we can parse out a PatientID and StudyID. A study is a set of x-rays taken at the same time. A study can have just a single frontal x-ray, a single lateral x-ray or have both a frontal and lateral set of x-rays. Patients who only had one set of x-rays will have only a StudyID of 1. Patients with multiple studies will have StudyIDs 1,2,3,4,…. The max number of studies for a single patient in this dataset was 72.

Competition CheXpert was set up as a competition, but only 5 of the 12 targets are included: ['Atelectasis', 'Cardiomegaly', 'Consolidation', 'Edema', 'Pleural_Effusion']

So many of model runs will only focus and/or train only on these 5 labels.

Note: This project is NOT intended to enter this competition

6 CheXpert User Agreement

The images in this dataset cannot be made publically available. Because of this, the images are not included in this project. But you can obtain the dataset yourself and run this code. You may need to adjust the working directories in the top bootstrap code block at the start of every notebook.

Note: See Hierarchical Path below for special directory structure

7

8 Project Directory Structure

8.1 data

- **Raw**
  - Holds metadata csv files from CheXpert
  - Holds download x-ray images
- **intermediate** - Holds Clean DF
- **d0 through d49** - Holds same x-ray images, but in a hierarchical directory structure for CoLab

8.2 modules

- **lib**
  - CheXpertData
    - Clean Meta Data (DataFrame)
    - PyTorch DataSet (Subclass of torch.utils.data.Dataset)
    - PyTorch DataLoaders
  - Metrics
    - Holds results passed to it via training
    - Displays metrics of the run along with things like ROC curves
  - StandardTraining
    - Runs the training loop with overridable default parameters
    - Allow consistency between different run

        * ModelLoop class - Run multiple combinations of parameters in same notebook
    - TrainingLoop
        * Runs the training
        * Pass in, NN, cuda device, optimizer, loss function and Metrics class described above
- **models**
    - CustomPneumonia
        * Custom build CNN model
        * Intended to be more educational rather than effective
        * Goal was to make sure:
            · Shapes match between each layer and between convolution and fully connected layers
            · Ability to calculate the number of trainable parameters
    - DenseNet
        * A fairly standard structure of DenseNet (see below for more details)
    - ResNet
        * ResNet_GrayScale
            · Uses torchvision.models.resnetXX models
            · Overrides conv1 to take in 1 input channel
            · Add fully connected layer at end of forward to reduce the 1000 changes to the desired output length
            · Registers a forward hook to add dropout2d is drop_out_precent is passed in
        * ResNet_Pretrained
            · Uses torchvision.models.resnetXX models with pretrained=True
            · Keeps input channels as 3
            · Add fully connected layer at end of forward to reduce the 1000 changes to the desired output length
            · Registers a forward hook to add dropout2d is drop_out_precent is passed in
            · Uses torch.repeat_interleave to duplicate the 1 grayscale channel to 3 channels

8.3 notebooks

- **Educational** - Two notebooks exploring PyTorch functionality
    - Pytorch Linear Regression from Scratch
    - Pytorch Automatic Differentiation
- **Kaggle Pneumonia**
    - Very early work with CNN on Kaggle Pneumonia chest x-rays
    - Basis of CustomPneumonia model
- **ModelLoop**
    - Notebooks that run combinations of different modes and/or parameters with a reduced dataset
- **ModelRuns**
    - Notebooks that train a single model and parameters
    - **saved**
        * Holds the pickle serialization of the model runs
- **Support Notebooks for Modules**
    - The workspace used to help build the modules
    - CheXpertData_NB
    - EDA

- HierarchicalPath
- Metrics and Training Loop
- Metrics_NB
- ModelLoop_NB
- Oversampling
- StandardTraining_NB
- TrainingLoop_NB

9

10 Environments

This project is intended to run both locally with GPU and on Google CoLab. Locally (GEFORCE RTX 2080 - 8GB), most models run very well, but might need to adjust the batch size down for some. The same notebooks can also run on Google CoLab without any changes. The bootstrap code block at the top of the notebooks determines the working directory on both environments and sets the PyTorch device to cuda.

Most if not all of the notebooks in this project were completed locally. Though CoLab has the advantage from a computational power perspective, it is very slow accessing the image files.

11 Hierarchical Directory

To see how to implement the hierarchical path, see: Hierarchical Path Notebook

In Google CoLab, the images need to be accessible. Since the dataset is not public and there is no API to retrieve the images, you need to mount a drive in order for CoLab to get access to these images. In the CoLab environment, the images are stored on Google Drive. There are plenty of posts on how to mount a drive in this way.

But the directory structure on Google Drive is not the same as your local file system. Since Google Drive is primarily a web based application, it cannot navigate directly to a file via a path. So every time you give it a path, a call must be made to obtain an ID for the folder and or path before it can access it. That means that every image read during the training loop must do a lookup of the item ID first.

With over 200k files, the training loop would have to find the ID out of 200,000 IDs. But it has to do this for every image, so the number of lookups in a single epoch would be 200k X 200k which is over 40 billion lookups.

In fact, just uploading the images to Google Drive was extremely slow. After several days of upload, the process was still not done. A silent timeout error was occuring most of the time so many of the uploaded images didn't get to the drive. See top of HierarchicalPath.ipynb (link above).

After a fair amount of research, the approach taken to resolve this was to copy the files into a hierarchical folder structure where no more than 50 items existed in any one folder. To get the proper ID, there are only 3 ID lookups, each with only 50 items. So instead of 200k lookup, we only needed to do 150.

Though this did resolve the timeout issue, the training loops were still much slower than running locally. So all models were run locally as long as there was enough memory to support it.

Note: The hierarchical directories greatly helped the upload process!

12 EDA

Bootstrap code to set working directory and mount drive in CoLab

[1]:
```
import sys
import os, os.path

sys.path.append(os.path.join(os.getcwd() ,'/modules'))
root_path = "C:/git/Springboard-Public/Capstone Project 2/"
IN_COLAB = 'google.colab' in sys.modules
if IN_COLAB:
    from google.colab import drive
    drive.mount('/content/drive')
    root_path = "/content/drive/My Drive/Capstone Project 2/"

print('Current Working Dir: ', os.getcwd())
print('Root Path: ', root_path)

# We need to set the working directory since we are using relative paths from␣
 ↪various locations
if os.getcwd() != root_path:
  os.chdir(root_path)
```

```
Current Working Dir:  C:\git\Springboard-Public\Capstone Project 2
Root Path:  C:/git/Springboard-Public/Capstone Project 2/
```

[2]:
```
import sys
import os, os.path
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

from modules.lib.CheXpertData import CleanMetaData
from modules.lib.TrainingLoop import *
from modules.lib.Metrics import *
from modules.lib.StandardTraining import *

%matplotlib inline
```

13 Data Cleansing

To see the process of data cleaning, please go to the supporing notebook: EDA Support Notebook

To see the module, please go to: ChextXRayImages

13.0.1 Frontal only

To reduce the complexity of learning and to reduce the overall number of images, only frontal views were included.

13.0.2 Multiple Studies

To prevent a patients with serial studies biasing the results, only the first 4 studies were included in the clean dataset.

These 2 filters reduced the image count from 223,415 to 131,748.

```
df_reduced = df[
             (df.View == 'Frontal') & # Don't show x-rays from the side
             (df.StudyID < 5) & # Don't include more than 4 studies for a single patient
             ((df.Orientation == 'AP') | (df.Orientation == 'PA')) # Don't show Left or Right La
             ]
```

13.0.3 12 Targets

Two targets were removed from the original 14: - No Finding - Support Devices

The first was removed for the same reason we do with one-hot encoding. Support Devices (i.e. Pacemakers, LVAD, etc.) was removed since these findings could be considered more of a feature than a diagnosis.

13.0.4 Binary Labels

To reduce the complexity of combining multi-label and multi-class classifications, the labels were converted into Boolean values.

- Not Mentioned = 0
- Negative = 0
- Uncertain = 0
- Positive = 1

14 CleanMetaData

[3]:
```
metaData = CleanMetaData()
target_columns = metaData.target_columns
df_clean= metaData.getCleanDF()
display(df_clean)
```

```
         PatientID  StudyID  Age  Sex_Male  Sex_Unknown  Orientation_PA  \
ImageID
0                1        1   68         0            0               0
1                2        2   87         0            0               0
2                2        1   83         0            0               0
4                3        1   41         1            0               0
```

```
5               4        1   20         0            0              1
...           ...      ...  ...       ...          ...            ...
223409      64537        2   59         1            0              0
223410      64537        1   59         1            0              0
223411      64538        1    0         0            0              0
223412      64539        1    0         0            0              0
223413      64540        1    0         0            0              0

         Support Devices                                    Image_Path  \
ImageID
0                    1.0  data/raw/train/patient00001/study1/view1_front...
1                    0.0  data/raw/train/patient00002/study2/view1_front...
2                    0.0  data/raw/train/patient00002/study1/view1_front...
4                    0.0  data/raw/train/patient00003/study1/view1_front...
5                    0.0  data/raw/train/patient00004/study1/view1_front...
...                  ...                                                ...
223409               0.0  data/raw/train/patient64537/study2/view1_front...
223410               0.0  data/raw/train/patient64537/study1/view1_front...
223411               0.0  data/raw/train/patient64538/study1/view1_front...
223412               0.0  data/raw/train/patient64539/study1/view1_front...
223413               0.0  data/raw/train/patient64540/study1/view1_front...

                Hierarchical_Path  Enlarged_Cardiomediastinum  ...  \
ImageID                                                        ...
0                data/d0/d1/i0.jpg                           0  ...
1                data/d1/d2/i1.jpg                           0  ...
2                data/d2/d2/i2.jpg                           0  ...
4                data/d4/d3/i4.jpg                           0  ...
5                data/d5/d4/i5.jpg                           0  ...
...                            ...                         ...  ...
223409    data/d9/d37/i223409.jpg                            0  ...
223410   data/d10/d37/i223410.jpg                            0  ...
223411   data/d11/d38/i223411.jpg                            0  ...
223412   data/d12/d39/i223412.jpg                            0  ...
223413   data/d13/d40/i223413.jpg                            0  ...

         Lung_Opacity  Lung_Lesion  Edema  Consolidation  Pneumonia  \
ImageID
0                   0            0      0              0          0
1                   1            0      0              0          0
2                   1            0      0              0          0
4                   0            0      1              0          0
5                   0            0      0              0          0
...               ...          ...    ...            ...        ...
223409              0            0      0              0          0
223410              0            0      0              0          0
223411              0            0      0              0          0
223412              1            0      0              0          0
```

```
223413              0            0      0              0          0

         Atelectasis  Pneumothorax  Pleural_Effusion  Pleural_Other  Fracture
ImageID
0                  0             0                 0              0         0
1                  0             0                 0              0         1
2                  0             0                 0              0         1
4                  0             0                 0              0         0
5                  0             0                 0              0         0
...              ...           ...               ...            ...       ...
223409             0             0                 1              0         0
223410             0             0                 0              0         0
223411             0             0                 0              0         0
223412             1             0                 0              0         0
223413             0             0                 0              0         0

[131748 rows x 21 columns]
```

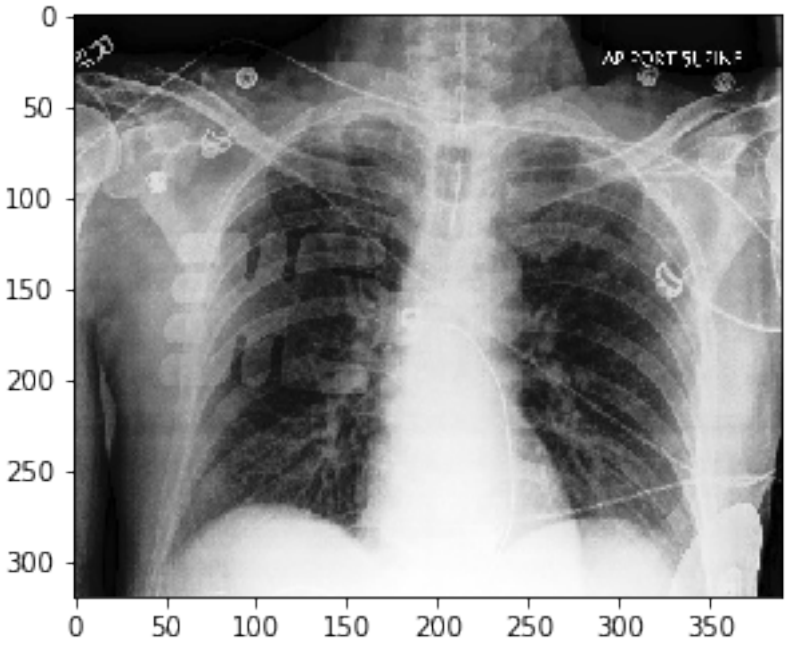
15 Intrinsic Leaks

Though there is nothing we can do about this, there are intrinsic leaks in the images that we should be aware of.

In-patients are much more likely to have positive findings then out-patients, especially out-patient routine screening x-rays. In-patient x-rays oftentimes can have leaks in the images. ECG leads, IVs and gown snaps are often present. Higher acuity patients will oftentimes have their x-rays taken in bed with AP orientation. PA is preferred since it does not tend to exaggerate the cardiac silhouette. Also, it might be harder to properly align very sick patients. All of these can leak features more common in positive diagnoses.

15.0.1 X-Ray with ECG Leads indicating an In-patient

[4]:
```
img = metaData.displayImage(123)
imgplot = plt.imshow(img, cmap=plt.get_cmap('gray'))
plt.show()
```

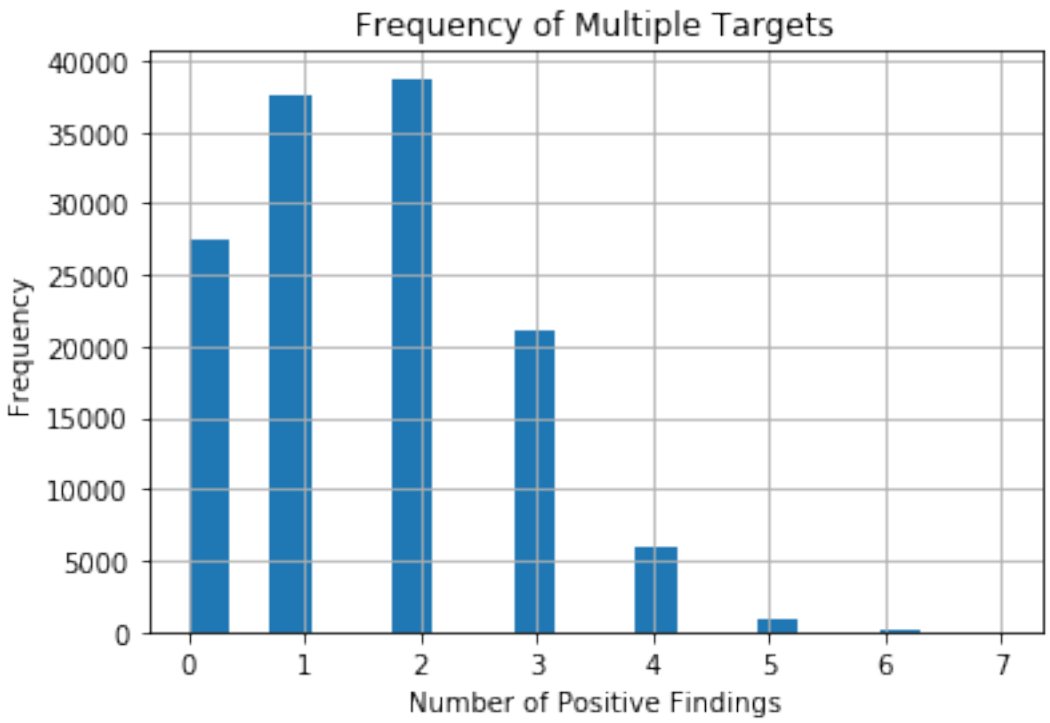


16 Target Interdependence

Though each of the 12 labels can be found independently of each other, it is very common to have findings that correlate with other findings. For example, it would be more common to find atelectasis in patients with pneumonia or pleural effusions. Rib fractures may make it painful to ambulate potentially leading to other lung diseases such as atelectasis or pneumonia.

Patients with high comorbidities may have several of these findings. Diagnoses like Cardiomeglia tend to not be transient and can persist for years. These longer term diseases may or may not correlate to other pathologies.

[5]:
```
ax = df_clean.iloc[:, -12:-1].sum(axis=1).hist(bins=20)
ax.set_title('Frequency of Multiple Targets')
ax.set_xlabel('Number of Positive Findings')
ax.set_ylabel('Frequency')
plt.show()
```

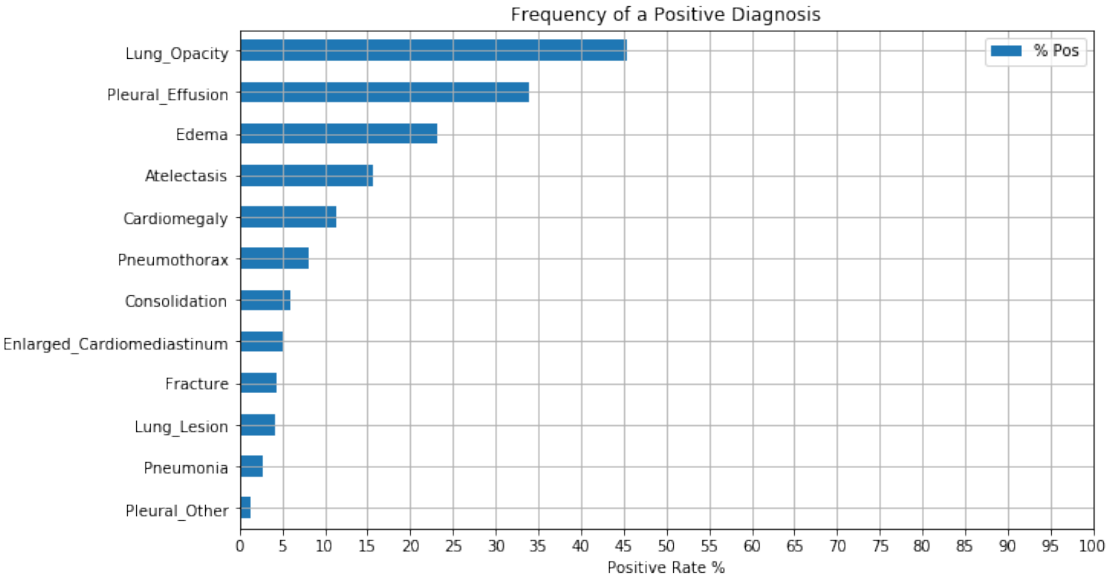


17 Imbalance

One of the biggest challenges is the imbalance of the 12 targets. The most frequent target, Lung Opacity, occurs in about 45% of the x-rays. But as you can see below, the majority of the labels have less than a 15% positive occurrence.

This imbalance along with non-independent multi-label classification will comprise the majority of the obstacles for this project.

[6]:
```
ax =pd.DataFrame(df_clean.iloc[:, -12:].sum() / len(df_clean) * 100,
             columns=['% Pos']).sort_values('% Pos').plot.barh(figsize=(10,6))
ax.set_title("Frequency of a Positive Diagnosis")
ax.set_xlabel("Positive Rate %")
ax.set_xlim(0, 100)
ax.set_xticks(range(0, 101, 5))
ax.grid()
```



18 CheXpertData module (DataFrame, Dataset and DataLoaders)

Please go to the supporting notebook for these classes: CheXpertData Notebook

To see the module, please go to: ChextXRayImages

18.0.1 CleanMetaData (DataFrame)

In addition to the data cleansing, the CleanMetaData class performs 2 critical functions: - Randomly select rows from metadata * This allow us to do model runs with reduced size for debugging and parameter evaluation - Splits the rows randomly into train and validation * This is be done after the the dataset has been reduced from the above step * The balance is checked between train and val - For each feature, the % positives is found - If the difference in this percentage between train and val exceeds the default 2%, a warning is displayed - The warning lists all features that have an imbalance - This is often seen with very low row counts - See bottom of EDA notebook: EDA Support Notebook

18.0.2 Dataset

This is a subclass of torch.utils.data.Dataset. The main purpose of the derived class to use the clean metadata to get the image path, the ground truth for the 12 targets and the ImageID index value of the image.

One the init method, lists are created to hold the above data. The clean dataframe is walked to load these lists.

In the getitem method, 3 tasks are performed: - Get the ImageID - Pull the PIL image from the image path and apply any transformations - Build a vector of the target values

The output of the getitem is a dictionary with 3 keys, 'id', 'img' and 'labels'. During training, this dictionary has to be parsed manually before feeing the images into the model.

18.0.3 Loaders

This class is used to obtain instances of torch.utils.data.DataLoader. A single loader for all images in the clean dataframe can be returned, but most of the time 2 loaders are returned for train and validation.

This class uses the other 2 classes in the CheXpertData module, **CleanMetaData** and **Dataset**. Because of these, these 2 classes are usually never called directly.

On instantiation of this class, you pass in the following parameters:

```
def __init__(self,
             image_width=320,
             image_height=320,
             affineDegrees=5,
             translatePrecent=0.05,
             shearDegrees=4,
             brightnessJitter=0.2,
             contrastJitter=0.1,
             augPercent=0.2,
             observation_min_count = None,
             target_columns=None):
```

The image shape parameters are self explanatory. The next 5 parameters are used to add image augmentations to the train transformation sequence (RandomAffine and ColorJitter). The augPercent is the percentage that these 2 augmentation types are applied to the dataset.

The observation_min_count parameter is for oversampling (see below).

The target_columns let you reduce the labels of the dataset.
If you pass in: - **training_columns = ['Pleural_Effusion', 'Edema']**-

only these 2 labels will be used for training and metrics.

In addition, the pixel mean and standard deviation are hard coded with values found in the support notebook:

```
    self.pixel_mean = 0.5064167
    self.pixel_sd = 0.16673872
```

The most common method used in this class is:

```
def getDataTrainValidateLoaders(self, batch_size=64,
                                val_percent = 0.2,
                                n_random_rows = 0):
```

This method returns a tuple of (train, val) data loaders with the total size and split percent passed in.

Oversampling The Loaders class also does the oversampling. We cannot use tools like SMOTE for this. Our features are embedded in the images and are derived via the convolutions during training. Because of this, we have nothing metrizable to do a nearest neighbors concept.

But we do have the combination of targets. There are 802 unique ground truth vectors in our clean dataframe. If you look at the frequencies, you see the typical exponential decal pattern. The all negative vector [0,0,0,0,0,0,0,0,0,0,0,0,0] is the most common followed by "Lung Opacity only" and "Lung Opacity, Pleural_Effusion only". The counts very quickly drop to very low numbers after the first few combinations.

So the approach to oversampling is to group rows by unique target vectors. A parameter **observation_min_count** can be passed into the class. This parameter is used to find any target vector who has less observations than this threshold. Then the observations in this group are sampled with replacement to add duplicate rows to the dataframe to reach the min count required.

Please go to the supporting notebook that help derived this method: Oversampling Notebook

[19]:
```
total_rows = df_clean.shape[0]
df_combined = pd.DataFrame(df_clean.PatientID)
df_combined['Combined_Targets'] = ''
for c in target_columns:
    df_combined['Combined_Targets'] = df_combined['Combined_Targets'] +␣
 ↪df_clean[c].map({-1:'N', 0:'0', 1:'P'})

df_combinations = pd.DataFrame(df_combined.groupby(['Combined_Targets']).
 ↪count())
df_combinations.columns = ['Frequency']
df_combinations['Percent'] = df_combinations.Frequency / total_rows
print(f'Total Unique Target Combinations {df_combinations.shape[0]:,}\n')
df_combinations = df_combinations.sort_values(['Frequency'], ascending=False)
print('\n'.join(target_columns))
display(df_combinations.head(15))
display(df_combinations.tail(5))
```

```
Total Unique Target Combinations 802

Enlarged_Cardiomediastinum
Cardiomegaly
Lung_Opacity
Lung_Lesion
Edema
Consolidation
Pneumonia
Atelectasis
Pneumothorax
Pleural_Effusion
Pleural_Other
Fracture
```

	Frequency	Percent

```
Combined_Targets
000000000000          25805  0.195866
00P000000000          12816  0.097277
00P000000P00          10586  0.080350
0000P0000000           4834  0.036691
000000000P00           4770  0.036205
00P0P0000P00           4584  0.034794
00P0P0000000           4116  0.031241
0000000P0000           3409  0.025875
00000000P000           2900  0.022012
0000P0000P00           2479  0.018816
00P0000P0000           2476  0.018793
0P0000000000           2293  0.017404
0000000P0P00           2206  0.016744
00P0000P0P00           1766  0.013404
00000000000P           1716  0.013025


                  Frequency   Percent
Combined_Targets
0P0000P000P0              1  0.000008
0P0000P0000P              1  0.000008
P00P0P000000              1  0.000008
0P000000PP0P              1  0.000008
PPPPP0P00P00              1  0.000008
```

19 TrainingLoop and Metrics modules

Both of the modules hold a single class with the same name.

To see the modules, please go to:

TrainingLoop

Metrics

Please go to the supporting notebooks, please go to:

Metrics Notebook

TrainingLoop Notebook

Metrics and Training Loop Notebook

The TrainingLoop performs the training epics and passes the results to an instance of the Metrics object. The Metrics object holds the probabilities and predictions for each epoch run. It also displays and saves this data. The Metrics instance can be used in the training loop to display selected results after each epoch. It also allows you to display all metrics including epoch by epoch plots after training is completed.

19.0.1 TrainingLoop

This is a relatively simple class. On instantiation, you pass in the device (CPU or CUDA), NN, optimizer and loss functions and an instance of the Metrics class.

```
def __init__(self, device, net, optimizer, criterion, metrics):
```

The **train** method gets passed in the number of epochs and the train and validation data loaders.

```
def train(self, num_epochs, train_loader, val_loader):
```

For each epoch, both the training and validation loaders are enumerated. The size of the batches is a parameter in the 2 data loaders, so the train code is unaware of this value. The train() and eval() methods are called on the model for train and val respectively. Only the train gets back propagation while val calls torch.no_grad().

The main calls in the epoch loops are below:

```
self.net.train()
for i, data in enumerate(train_loader, 0):
    ids, inputs, labels, outputs = self.processBatch(data)
    self.metrics.appendEpochBatchData(ids, outputs)
    self.epoch_loss += self.backProp(outputs, labels)

...

self.net.eval()
with torch.no_grad():
    for i, data in enumerate(val_loader, 0):
        ids, inputs, labels, outputs = self.processBatch(data, is_validation=True)
        self.metrics.appendEpochBatchData(ids, outputs, is_validation=True)
```

The processBatch method is:

```
def processBatch(self, data, is_validation=False):

    # Convert output from loader
    ids, inputs, labels = self.parseLoaderData(data)

    if not is_validation:
        # zero the parameter gradients
        self.optimizer.zero_grad()

    # Get outputs from Model
    outputs = self.net(inputs)

    return ids, inputs, labels, outputs
```

Since the dataset subclass returns a dictionary with the getitem method, the values are parsed with:

```
ids, inputs, labels = data['id'], data['img'], data['labels']
# move data to device GPU OR CPU
inputs, labels = inputs.to(self.device), labels.to(self.device)
```

The inputs variable is the transformed images for the batch.
The shape of this tensor is **[batch_size, channels, height, width**]

The outputs tensor from the net call has a shape of **[batch_size, target_count]**. There is no activation function on the last **nn.Linear** layer in the models, so the outputs are raw numbers between $-\infty$ and ∞. So the assumption is that the loss function will do the squashing of this output (see Criterion below).

The metrics.appendEpochBatchData(ids, outputs) hands off the batch results to the Metrics object.

For train, backprop is done with this method:

```
def backProp(self, outputs, labels):
    loss = self.criterion(outputs, labels)
    loss.backward()
    self.optimizer.step()
    return loss.item()
```

19.0.2 Metrics

Some Metric Background This class is key to understanding how our model is doing, both on completion and for each epoch.

Looking at multiple values after each epoch is critical in this dataset. Since most targets are close to 90% negative, chances are you are going to get very high accuracy scores. The model is trying to predict the positive labels. A bad model that can't do this will still get ~90% of the predictions right.

There are also multiple ways to look at accuracy for multi-label data. You can say the prediction is accurate only if all 12 targets are correct. This is a very high bar and doesn't reflect the effectiveness of the model. If every image correctly identifies 11 out of 12 labels, the accuracy would still be 0% since no image got all 12 correct. This type of accuracy score is returned with **sklearn.metrics.accuracy_score**.

You can also use the number of correct predictions divided by the total number of predictions (1-Hamming score).
This type of accuracy score is returned with **1 - sklearn.metrics.hamming_loss**.

Butthe Hamming approach gets us back to the 90% accuracy problem we just described above. The bottom line is that accuracy is probably not the best way to access the model effectiveness.

Model probability vs. predictions We need to be sure of the differences between these two before we discuss the next metric scores.

The CNN model outputs produce one value for each target. These values can be anything from from $-\infty$ to ∞. So take these outputs and squash them using the sigmoid function to produce probabilities between 0 and 1.

To get a prediction of either 1 or 0, we take this probability and check it against a threshold (usually 0.50). If the prob is >= 0.5, we predict positive or 1. If below the threshold, we predict negative or 0.

The end result is predictions make a single assertion about an image, i.e. either it is a dog or it is not. But the predictions don't show how close we got. i.e. it is not a dog since the prob was 0.49999.

Probabilities show how close we got. Scores based on probabilities more accurately reflect effectiveness of our model. But if your desired output is a single assertion, prediction based scores are probably better.

Combined Metrics vs. Itemized Metrics Like we tried to do above with accuracy, we can combine the probabilities and/or predictions of the 12 labels into a single score, so simply show the score 12 times, one for each target. We tend to prefer a single score since it is easier to make a statement about the effectiveness of the entire model. But taking some kind of average of these 12 scores can be very misleading. A bad prediction for a few targets can suggest the model is a poor performer.

For this reason, the itemized scores will be the primary approach for accessing model performance.

Four Primary Sores

- Recall (Sensitivity)
- Precision
- ROC AUC
- Average Precision

The first 2 are based only on the predictions and do not take probabilities into account.
- $Recall = \frac{TP}{TP+FN}$ - What % of the true positives were found - $Precision = \frac{TP}{TP+FP}$ - What % of the true predictions were correct

The last 2 scores are based on probability. Because the scores are probabilities, this gives us "many" values as opposed to only the 4 prediction values (TP. TN, FP, FN). i.e. a TP can have many different probabilities (.12, .45, .78, .98, .87. …). These floats allow us to plot both the Receiver Operator Curves (ROC) and the Precision/Recall curves. To get a value from these curves, we take the area under the curve.

If our recall is 50%, half the actual positives would be true positives and half would be false negatives. If all the false negatives have a probability of 0.4999, and all the true post positives have a probability of 0.9999, then the ROC AUD and Average Precision would be much greater than 50%.

Since our stated model goal was to try to get the Recall scores, this is the primary score used to determine the model's success. But we can't just look at a single value. A good recall with a bad precision means we won't miss any actual positives, but we won't be able to trust these predictions very much. We can simply make the model always predict Positive for all observations. This would give us a 100% Recall, but a very low Precision.

19.0.3 Metrics Class

To see the Metrics modules, please go to:

Metrics

Please go to the supporting notebook, please go to:

Metrics Notebook

Metrics and Training Loop Notebook

On instantiation, this class is given the target columns, the actual label values, and optionally the target thresholds.

```
def __init__(self, target_columns, train_actual, val_actual, target_thresholds=None, ...
```

The training loop passes the model outputs after each batch iteration. It also passes the set of ImageIDs so that these results can be matched back to the original metadata.

```
def appendEpochBatchData(self, ids, outputs, is_validation=False):
```

This batch data is stored temporarily in a dictionary of ID:Output. Each batch appends to this dictionary until the epoch is done. Then the dictionaries for train and val are converted to a dataframe and appended to the prediction and probability histories:

```
def closeEpoch(self, epochNumber, is_validation=False):
    if is_validation:
        self.df_val_prediction = self.getPredictionDataFrame(self.epoch_val_predictions)
        self.val_prediction_hx[epochNumber] = self.df_val_prediction.copy()
        self.epoch_val_predictions = {}

        self.df_val_probability = self.getProbilityDataFrame(self.epoch_val_probabilities)
        self.val_probability_hx[epochNumber] = self.df_val_probability.copy()
        self.epoch_val_probabilities = {}
    else:
        self.df_train_prediction = self.getPredictionDataFrame(self.epoch_train_predictions)
        self.train_prediction_hx[epochNumber] = self.df_train_prediction.copy()
        self.epoch_train_predictions = {}

        self.df_train_probability = self.getProbilityDataFrame(self.epoch_train_probabilities)
        self.train_probability_hx[epochNumber] = self.df_train_probability.copy()
        self.epoch_train_probabilities = {}
```

The Metrics class assumes the final output of the model has not been passed through any activation function (see Criterion below). So the Metrics class need to generate probabilities and predictions from these raw outputs:

```
def getPredictionsFromOutput(self, outputs):
    """
    We are using BCEWithLogitsLoss for our loss
    In this loss function, each label gets the sigmoid (inverse of Logit) before the CE loss
    So our model outputs the raw values on the last FC layer
    This means we have to apply sigmoid to our outputs to squash them between 0 and 1
    We then take values >= .5 as Positive and < .5 as Negative (or optional target thresholds)
    """
```

```
    probabilities = torch.sigmoid(outputs.data)
    predictions = probabilities.clone()

    #We need to make sure all tensors are in the same memory space
    if self.target_thresholds.device != predictions.device:
        self.target_thresholds = self.target_thresholds.to(predictions.device)

    predictions[predictions >= self.target_thresholds] = 1 # assign 1 label to those with gt or
    predictions[predictions < self.target_thresholds] = 0 # assign 0 label to those with less t

    return probabilities, predictions
```

The target thresholds are optional with default values of 0.5:

```
    if target_thresholds is None:
        #Default thresholds for all targets is .5
        self.target_thresholds = np.ones(len(self.target_columns)) * .5
    else:
        #Custom Thresholds
        #i.e. set Edema threshold to .2.  If Edema probability >= .2, prediction = 1
        self.target_thresholds = target_thresholds
```

These thresholds are a trick. This is done entirely in the Metrics class and has nothing to do with the loss function. So the predictions displayed in this class are affected, but training and weight adjustments are completely unaware of these thresholds.

At the end of the epoch, the current dataframes are used to build the scores and optionally displayed in the training loop display. This display method has many different parameters to allow you to pick which items to display:

```
def displayMetrics(self,
                   metricDataSource = MetricDataSource.Both, #train, val, both
                   showCombinedMetrics=True,
                   showMetricDataFrame=True,
                   showROCCurves=True,
                   showPrecisionRecallCurves=True,
                   include_targets=None,
                   combinedAverageMethod='samples',
                   gridSpecColumnCount=4,
                   gridSpecHeight=3,
                   gridSpecWidth=20):
```

The include_targets parameter lest you display only a subset of the targets. For example, instead of showing all 12 label scores after each epoch, you can just display the 5 targets identified in the CheXpert competation.

At the end of training, the last epoch results can be displayed. The display for the epoch and the final results are the same method (displayMetrics), but with different display options passed in.

The Metrics class also has a method (displayEpochProgression) to display plots by epoch. i.e. display the progression of the recall score plotted against the epoch number.

```
def displayEpochProgression(self,
                            showResultDataFrames=False,
                            showAccuracyProgression=True,
                            showRecallProgression=True,
                            showPrecisionProgression=True,
                            showF1Progression=True,
                            showROCAUCProgression=True,
                            showAvgPrecisionProgression=True,
                            include_targets=None):
```

19.1 Standard Training Module

This module is the glue for all other modules.

To see the module, please go to:

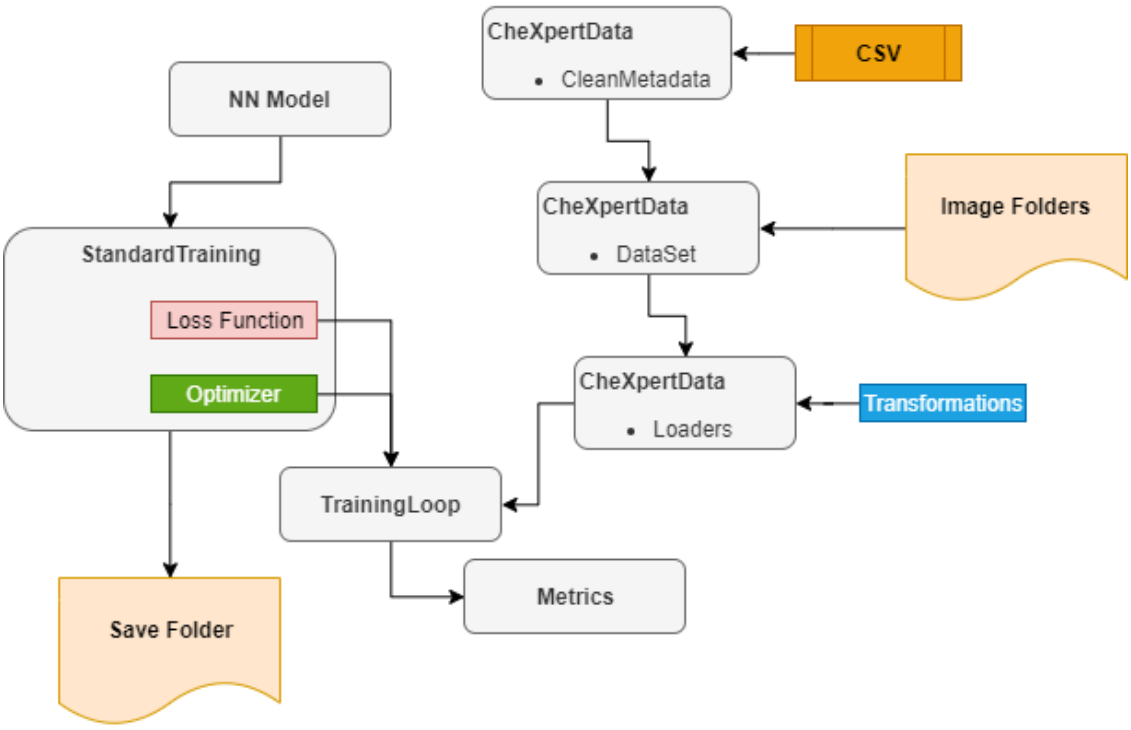
StandardTraining

Please go to the supporting notebook, please go to:

Standard Training Notebook

There are 2 classes in this module, StandardTraining and ModelLoop.

19.1.1 StandardTraining Class



Since this is the gateway class, the constructor takes in a lot of optional parameters:

```
    def __init__(self,   number_images,
```

```
batch_size,
learning_rate,
num_epochs,
device,
net,
epoch_args='standard',
use_positivity_weights=False,
image_width = 320,
image_height = 320,
affineDegrees=5,
translatePrecent=0.05,
shearDegrees=5,
brightnessJitter=0.2,
contrastJitter=0.1,
augPercent=0.2,
observation_min_count=None,
l2_reg=0,
loss_reduction='mean',
target_columns=None,
target_thresholds=None,
save_path=None,
net_name=None,
net_kwargs=None):
```

These parameters include options for the data loaders, metrics and training loop classes as well as the model itself. There are also options needed for the loss function, optimizer and persistence.

The main purpose of the StandardTraining class is to allow repeatable model runs using the exact same default values. So if you want to compare 2 runs with and without L2 regularization, you only need to override one parameter. All the other parameters will be the same since the default values are used.

There are also some training values that are hard coded into the class. For example, the loss function is always BCEWithLogitsLoss and the optimizer is always Adam. You modify some parameters of these two functions, but you cannot change their type.

19.2 Criterion (Loss Function)

19.3 Optimizer

19.3.1 ModelLoop

This class is simular to the StandardTraining class, but you pass in a collection of different models and parameter and all items are run in the same notebook. This turned out to be of limited value due to the inability to fully do gargage collection between runs. But there some a few loop notebooks in the following folder:

Model Loop Folder

There is also a support notebook:

20 Models Used

20.0.1 Custom

20.0.2 ResNet

20.0.3 Dropout

- Pre-Trained
- DenseNet # Training Features ### Criterion ### BCEWiogitsLoss
- pos_weight
- Reduction ### Optimizer
- Adam
- L2 Regularization ### Image Augmentations
- Affine
- Translation
- Shear
- Brightness
- Contract ### Oversampling
- Min observation count ### Training Targets # Metrics ### Epoch (Itemized)
- Recall
- Precision
- F1
- ROC AUC
- Avg Precision ### Training Completion
- Accuracy
- All Labels
- Hamming Loss
- Combined Recall/Precision
- ROC / Precision-Recall Curves ### Epoch Progression
- Accuracy
- Recall
- Precision
- F1
- ROC AUC
- Avg Precision
- Persistence Combination Tried A B C D Results Educational Notebooks Todo

[8]:
```
save_name = 'ResNet34_Oversample_L2_Sum_PosWeight_5_Targets'
path= f'notebooks/ModelRuns/saved/{save_name}/'
StandardTraining.displayRunParameters(path)
metrics = StandardTraining.loadMetrics(path)
```

```
Network Name:  ResNet_GrayScale
Nework Arguments:  layers:34,drop_out_precent:0.5,out_channels:5
```

	Paramter	Value
0	number_images	0

```
1               batch_size        64
2            learning_rate     1e-05
3               num_epochs         4
4               epoch_args  standard
5   use_positivity_weights      True
6              image_width       320
7             image_height       320
8            affineDegrees         5
9         translatePrecent      0.05
10            shearDegrees         5
11        brightnessJitter       0.2
12          contrastJitter       0.1
13              augPercent       0.4
14   observation_min_count       150
15                  l2_reg       0.1
16          loss_reduction       sum
```

```
Targets:  Atelectasis,Cardiomegaly,Consolidation,Edema,Pleural_Effusion
```

[9]:
```
save_name = 'DenseNet_Oversample_L2_Sum_PosWeight_5_Targets'
path= f'notebooks/ModelRuns/saved/{save_name}/'
StandardTraining.displayRunParameters(path)
metrics = StandardTraining.loadMetrics(path)
```

```
Network Name:  DenseNet
Nework Arguments:  nr_classes:5
```

```
                  Paramter     Value
0            number_images         0
1               batch_size        16
2            learning_rate     1e-05
3               num_epochs         4
4               epoch_args  standard
5   use_positivity_weights      True
6              image_width       224
7             image_height       224
8            affineDegrees         5
9         translatePrecent      0.05
10            shearDegrees         5
11        brightnessJitter       0.2
12          contrastJitter       0.1
13              augPercent       0.4
14   observation_min_count       150
15                  l2_reg       0.1
16          loss_reduction       sum
```

```
Targets:  Atelectasis,Cardiomegaly,Consolidation,Edema,Pleural_Effusion
```

[10]:
```
metrics.displayMetrics()
```

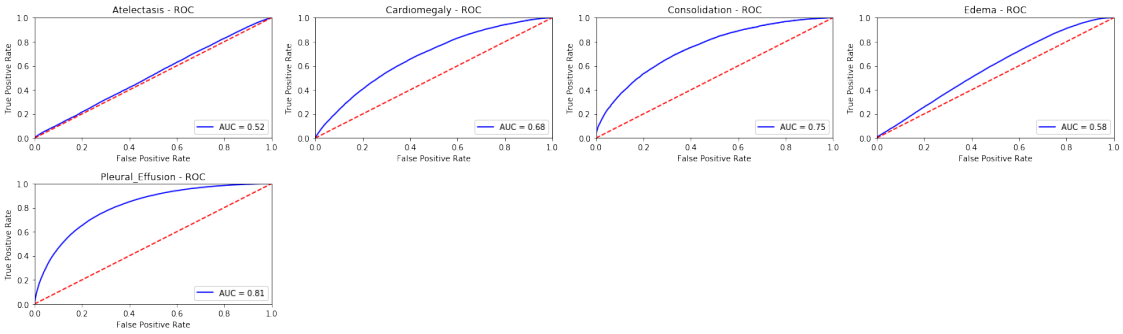
```
TRAINING
```

	Score for all Targets
Accuracy Score	0.201868
Hamming Loss	0.355581
Combined Recall	0.307234
Combined Precision	0.183334
Combined F1	0.215095

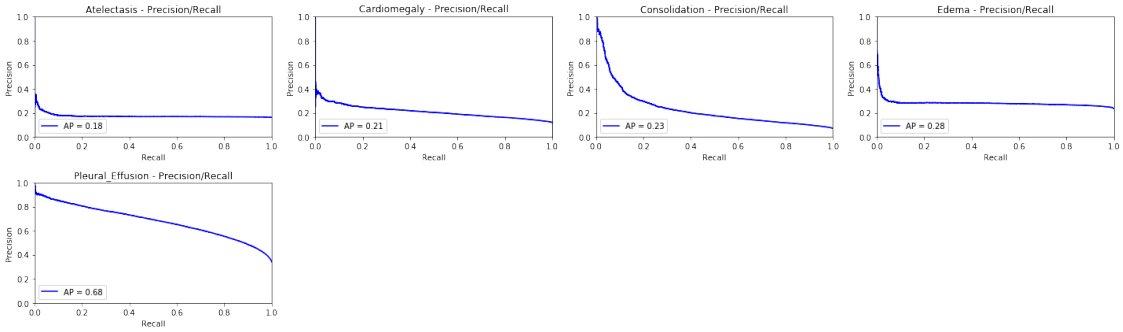
	Target	True Positives	Predicted Positives	Recall	Precision	\
0	Atelectasis	17421	25858	0.256415	0.172751	
1	Cardiomegaly	12832	65322	0.817488	0.160589	
2	Consolidation	7614	37507	0.684660	0.138987	
3	Edema	25220	22878	0.258366	0.284815	
4	Pleural_Effusion	36416	46646	0.747117	0.583265	

	F1	ROC AUC	Avg Precision
0	0.206428	0.521008	0.175118
1	0.268444	0.675674	0.209964
2	0.231068	0.747371	0.230013
3	0.270947	0.583052	0.281555
4	0.655101	0.810452	0.678997

```
***** ROC *****
```




```
***** Precision / Recall *****
```



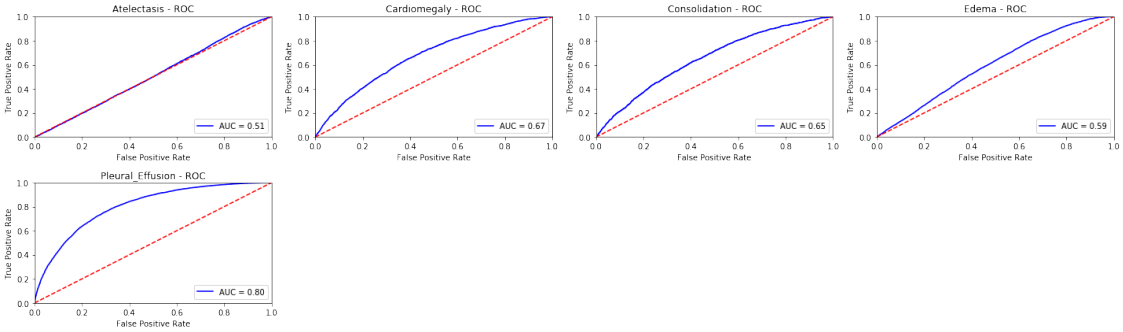
VALIDATION

	Score for all Targets
Accuracy Score	0.227460
Hamming Loss	0.334851
Combined Recall	0.289649
Combined Precision	0.179136
Combined F1	0.206391

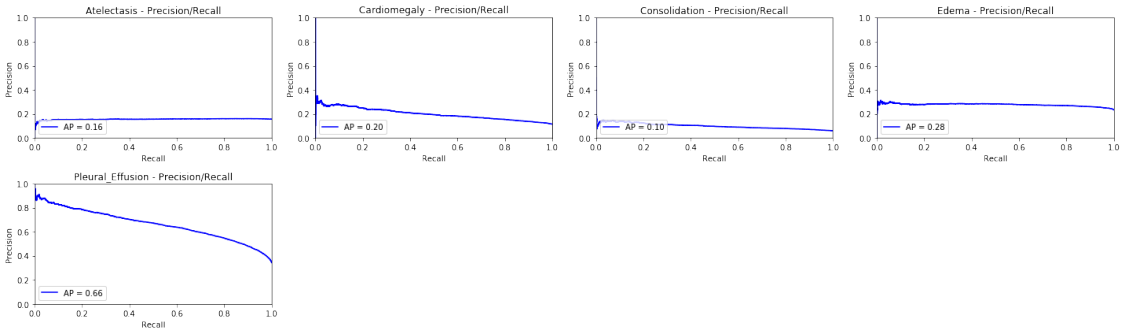
	Target	True Positives	Predicted Positives	Recall	Precision	\
0	Atelectasis	4104	5471	0.205653	0.154268	
1	Cardiomegaly	2995	15220	0.791987	0.155848	
2	Consolidation	1571	6830	0.446213	0.102635	
3	Edema	6067	5483	0.256799	0.284151	
4	Pleural_Effusion	8958	11182	0.727395	0.582722	

	F1	ROC AUC	Avg Precision
0	0.176292	0.506564	0.155841
1	0.260445	0.674204	0.200342
2	0.166885	0.651252	0.100495
3	0.269784	0.594591	0.276674
4	0.647071	0.801780	0.662120

***** ROC *****

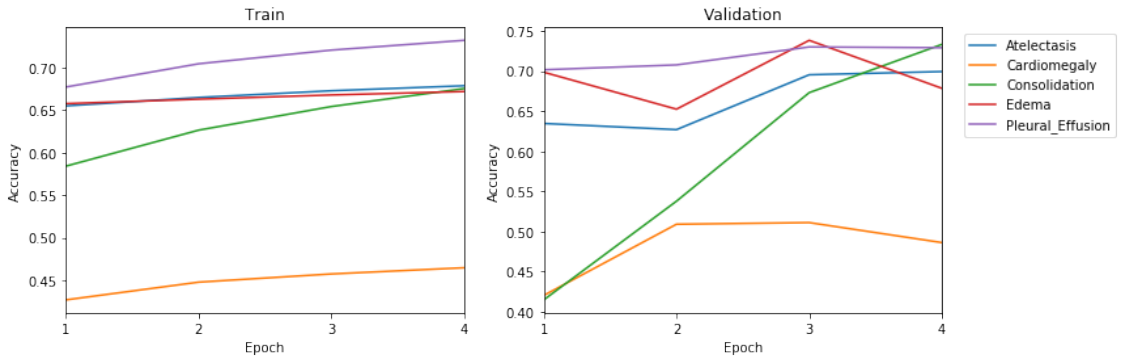


***** Precision / Recall *****

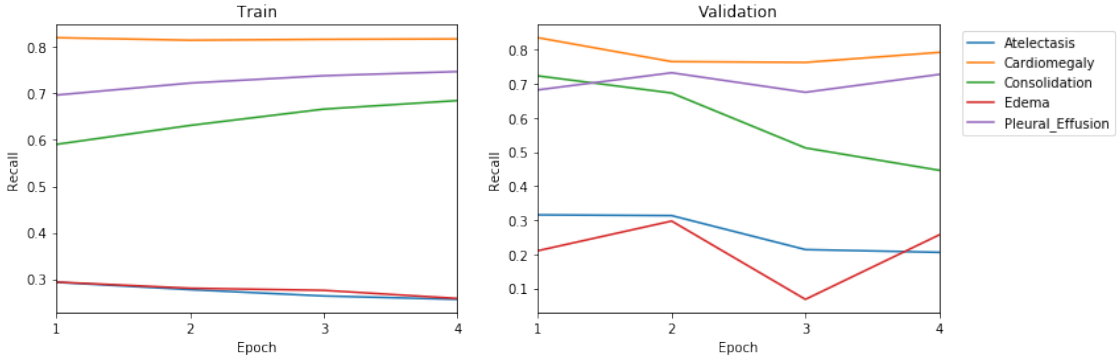


[11]:
```
metrics.displayEpochProgression()
```

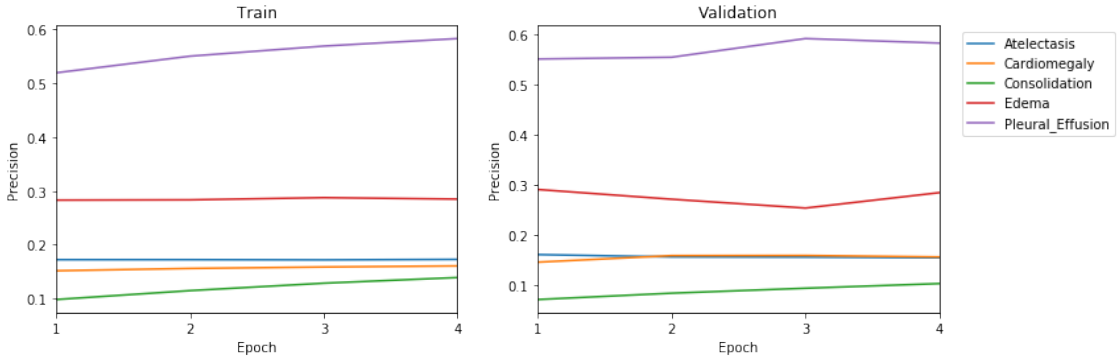
ACCURACY



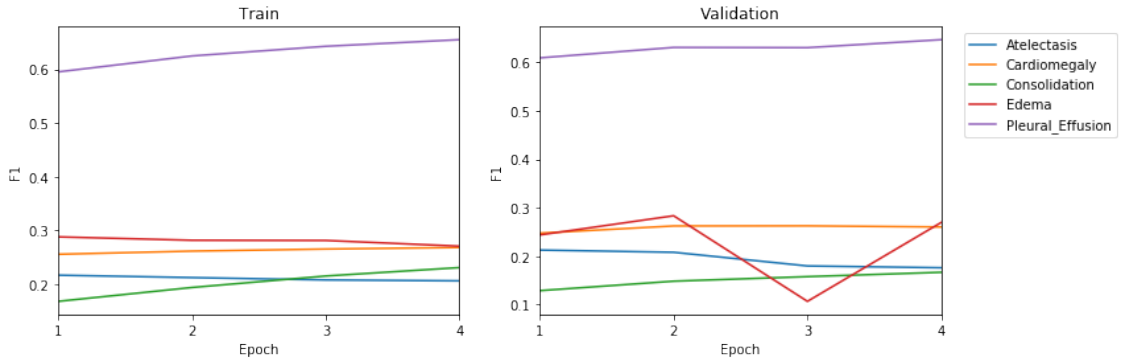
RECALL



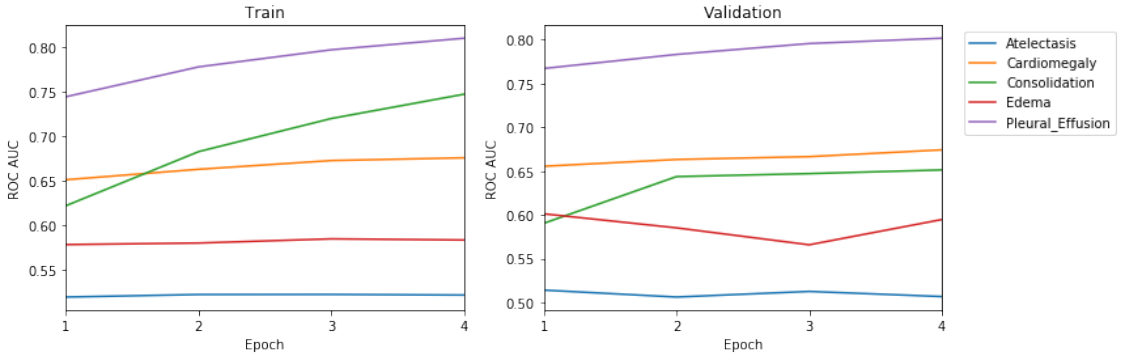
PRECISION



F1



ROC AUC



AVERAGE PRECISION