Machine Learning 2
Final Project Report
Team 8

Relabeling Chest X-Rays

Joshua Bernstein, Kshitij Bhat, Zachary Buckley

Introduction

The project initially aimed at training a Neural Network in PyTorch to solve a multi-label classification problem based on the National Institute of Health's (NIH's) Chest X-Ray dataset, released in 2017[1]. To achieve that initially a 2 GB random sample was used from the larger dataset NIH released, available to download from Kaggle[2]. Further literature survey led to a change in plans, as an article[3] published by Luke Oakden-Rayner, a radiologist and Ph.D candidate doing research for the University of Adelaide in Australia,[4] raised concerns regarding the labels present in the data sample from NIH.

Oakden-Rayner's paper brings up a number of concerns regarding the NIH Chest X-Ray dataset, particularly related to the labeling they applied to the X-Ray Images. According to the paper, an NLP model was used to classify the labels based on radiologists' written reports. However, after visually examining a sample of the images, it was found that the extracted labels frequently do not actually match the images. His concerns inspired us to look around for better labeled Chest X-Ray data. We eventually stumbled across CheXpert[5], a larger Chest X-Ray dataset released by Stanford University. This dataset also used an NLP algorithm to label the images, but with very different techniques, and more validation of the labeling being applied, by qualified individuals[6].

This path led us to our ultimate goal for this project. By training a model on CheXpert data, we would hopefully come up with a model which could accurately label chest X-ray with correct labels. After producing such a model, we could then run it on the Kaggle dataset, and produce a more accurate labeling for the images than the current labelling that NIH provided.

---

[1] http://openaccess.thecvf.com/content_cvpr_2017/papers/Wang_ChestX-ray8_Hospital-Scale_Chest_CVPR_2017_paper.pdf

[2] https://www.kaggle.com/nih-chest-xrays/sample

[3] https://lukeoakdenrayner.wordpress.com/2017/12/18/the-chestxray14-dataset-problems/

[4] https://lukeoakdenrayner.wordpress.com/about-me/

[5] https://stanfordmlgroup.github.io/competitions/chexpert/

[6] https://arxiv.org/abs/1901.07031

Datasets

NIH

The National Institute of Health's Chest X-Ray dataset consists of 112,120 lung X-rays from 30,805 different patients. Images were labeled as to whether or not the lungs contained each of the following diseases: Hernia, Pneumonia, Fibrosis, Edema, Emphysema, Cardiomegaly, Pleural Thickening, Consolidation, Pneumothorax, Mass, Nodule, Atelectasis, Effusion, and Infiltration. Any image which did not contain any of these labels was labeled as "No Finding."

For this project, we downloaded a random sample constructed by Kaggle, which contained 5,606 images which had been resized to 1024x1024 pixels.

In order to determine labels for the images, the authors used NLP techniques to extract this information from radiology reports. However, in his article, Luke Oakden-Rayner gives reasons why this may not produce accurate labels. The intention of radiology reports is to provide specific information to a doctor, often regarding only a specific disease. Report content is based on the patient's past history and the clinical context of the report. Furthermore, Oakden-Rayner noticed that many of the images tagged as positive for "pneumothorax" actually were lungs which had formerly contained pneumothoraces, but which had since been drained. This means that the images do not correctly indicate what a lung with a pneumothorax looks like.

Chexpert

Stanford University's CheXpert Dataset consists of 224,316 chest X-rays from 65,240 different patients. For some of the patients, lateral images were included in addition to frontal ones. Like the NIH dataset, images were labeled based on NLP extractions from radiology reports. Information regarding the following conditions was extracted: Enlarged Cardiomediastinum, Cardiomegaly, Lung Opacity, Lung Lesion, Edema, Consolidation, Pneumonia, Atelectasis, Pneumothorax, Pleural Effusion, Pleural Other, and Fracture, as well as the presence of a Support Device.

Each image's report was examined for mentions of each of the above conditions. For each condition, the image could receive a label of Positive (1), Negative (0), Uncertain (u), or receive no label (blank). Positive indicates mention of a condition being present, and negative indicates mention of a condition being absent. An Uncertain label indicates that explicit uncertainty is present in the report (e.g. "diffuse reticular pattern *may* represent mild interstitial pulmonary edema"). If a condition is not mentioned at all, the label is left blank.

Each image also had a "No Finding" value, which was set to Positive if all the pathology labels (excluding "Support Device") were Negative or blank.

The CheXpert authors made several improvements on the NLP used by the NIH authors. Among other changes, they found ways to better distinguish between mentions of absence and uncertainty, and made efforts to deal with double negatives (e.g. “cannot exclude pneumothorax”).

Intersection of Labels

Both the NIH and CheXpert datasets contained the following labels in common: Pneumonia, Edema, Cardiomegaly, Consolidation, Pneumothorax, Atelectasis, and No Finding. Therefore, when evaluating our models on CheXpert test data, as well as the NIH/Kaggle data, we only evaluate these seven labels. This means we are dropping 8 NIH labels and 7 CheXpert labels from our comparison.

Preprocessing

Labels

Before training a model on the CheXpert data, we had to decide how to handle Uncertain labels and blanks. The CheXpert authors had tried a variety of methods on how to handle Uncertainty. Two of the methods that performed best were called U-Zeros and U-Ones. U-Zeros involves replacing all Uncertain labels with 0 (negative) before training; U-Ones involved replacing them all with 1 (positive).

The paper did not make clear how they handled blanks in their training. One option we considered was to treat them as Uncertain, thus changing them to 0s or 1s when using U-Zeros or U-Ones respectively. Our other option was to always convert blanks to zeros, thereby making the assumption that if an image was not classified as definitely or possibly having a condition (1 or uncertain), then we assume the condition is absent. In the end, we decided on the second approach, as it seemed too risky to ever treat the lack of a mention as a positive.

Images

Our initial image processing, used for 15 epoch model below (U-Zeros only) consisted of reading the images by using keras's read_img function. We used the various options in the Keras API to convert the image to RGB, and resize the image to 600x600. In addition we used torchvision transforms to normalize the image using a mean of 0 for each channel, and a standard deviation of 1 for each channel. After examining a plot of training and validation loss (see Figure 1), and seeing how overfitted the model became after just a few epochs of training, we looked for better options.

We came across a repository by jfhealthcare which at some point previously had been used to train the 1st place model for the CheXpert competition, though today it appears to have fallen to 3rd place[7][8]. Our next set of models used very different preprocessing steps, inspired by those we saw in jfhealthcare's GitHub repository. The jfhealthcare codebase used cv2 for their image processing pretty heavily, so to avoid recreating the wheel, we decided to adopt that library and drop the Keras API's image preprocessing routines. The following transforms are now being applied to all input images being fed into our models[9]:

[7] https://stanfordmlgroup.github.io/competitions/chexpert/
[8] https://github.com/jfhealthcare/Chexpert
[9] The individual transforms can be found in Code/ImageTransforms.py from our GitHub. They are used in Code/training_common_utils.py.

1) Histogram Equalization: Applied first, while the image is still in grayscale. This is a common image preprocessing step, used to improve contrast in images, which ultimately should give the Neural Network a better chance of picking out relevant features[10].
2) Grayscale to RGB: This transform is applied by necessity so that we can use the torchvision densenet implementation (it assumes a 3 channel initial input). After quite a bit of digging, we determined that while possible, converting the torchvision densenet over to grayscale likely wouldn't be worth the invested time.
3) Fixed Ratio Resize: This transform was one of the more interesting things we came across in jfhealthcare's implementation. It determines the aspect ratio of the image based on the size, chooses the correct 'long' side, and then resizes the image to a width and height that maintains the aspect ratio with the longest edge being 512 pixels. It then pads the resulting image with the mean value (128). It ends with a 512 by 512 version of the original image, which maintains the aspect ratio of image components as much as possible.
4) Gaussian Blur: This Transform will effectively smooth out features, most likely in an attempt to avoid the model paying too much attention to noise in the files[11].
5) Normalization: The only thing worth mentioning here is that we changed our mean and standard deviation values to match those used by jfhealthcare, since they seem to offer more room for the model to fine-tune than our initial choice (which seems like it could be helpful for avoiding vanishing gradient issues) (mean of 128, standard deviation of 64).

Specifically for the training dataset, we also added random affine transforms, using torchvision's RandomAffine transform (also inspired by jfhealthcare's code). The RandomAffine transform provided by torchvision will randomly choose to rotate, translate, scale, or shear the images as we iterate through them via the dataloader.

[10] https://en.wikipedia.org/wiki/Histogram_equalization
[11] https://en.wikipedia.org/wiki/Gaussian_blur

Model

Network Design

We chose to use densenet121 for our model based on the findings from paper associated with the CheXpert dataset[12]. The DenseNet-BC implementation we're using is available both pre-trained on the ImageNet dataset, and untrained from torchvision[13]. DenseNet builds on the "regular" Convolutional Neural Network (CNN) concept by allowing each CNN layer in the network to connect to every subsequent CNN layer. It does this to simultaneously ensure that each layer has a more direct connection to feature maps provided by the first CNN layer, and a more direct connection to the final outputs. Having a full range of short to long paths between the CNN layers in the network makes it much quicker to train than existing very deep CNN architectures, as vanishing gradient problems are reduced, without sacrificing the network's ability to fit to complicated functions that only a very deep CNN can handle. The outcome is a very large network that can train effectively given a relatively small number of inputs compared to other similarly very deep neural networks[14].

DenseNet-BC takes the DenseNet concept and mixes in Bottleneck Layers and Compression. Bottleneck Layers are a technique which improves computational efficiency by reducing the number of input feature-maps per layer. Compression reduces the number of output feature-maps by some factor; as described, it seems fairly analogous to Dropout, except applied to feature-maps instead of features. Practically, this means that in addition to the vanishing gradient reducing nature of DenseNet's design, we also have further reduction of overfitting, while simultaneously reducing the memory footprint of the network[15].

We looked at potentially adapting the densenet components provided by torchvision to use grayscale as an input directly, but were unable to get that working in a reasonable amount of time. Further work towards that front could lead to improvements of the model's classifications, as we're effectively forced to multiply the number of input features by 3 to accommodate torchvision's assumption of RGB input images.

Training Setup

We setup 2 very similar python files (training_team8_U-Zeros.py, and training_team8_U-Ones.py) for training our models. One used the U-Zeros label transformation, and one used the U-Ones label transformation described by the paper that originally released

[12] https://arxiv.org/abs/1901.07031
[13] https://pytorch.org/docs/stable/torchvision/models.html?highlight=models
[14] https://arxiv.org/pdf/1608.06993.pdf
[15] https://arxiv.org/pdf/1608.06993.pdf

the CheXpert dataset.[16] CheXpert provided training data, which we downsampled to 30% in order to be able to run it more efficiently. Our final models were created using the 45 epoch training method described in the last part of this section, while the “15 Epoch Training Attempt”, and the “4 Epoch Training Attempt” sections describe intermediate steps we took along the way.

15 Epoch Training Attempt

The first model we trained used input data that was prepared by a relatively minimal preprocessing pipeline and the U-Zeros label transformation strategy, leading to a very loose connection between our training and validation loss almost immediately. We then started to dig more heavily into how we needed to improve our preprocessing, and into any optimizations we could apply to speed up the training loop (as training this model took approx. 35 minutes per epoch or approx. 9 hours). Due to time required for training it, and the missing data augmentation/preprocessing steps, we didn’t attempt to train a model for the U-Ones label transformation strategy this time around. We also had only partially implemented our checkpointing mechanism at the time, and were saving the model anytime the loss decreased on either the training or validation set.

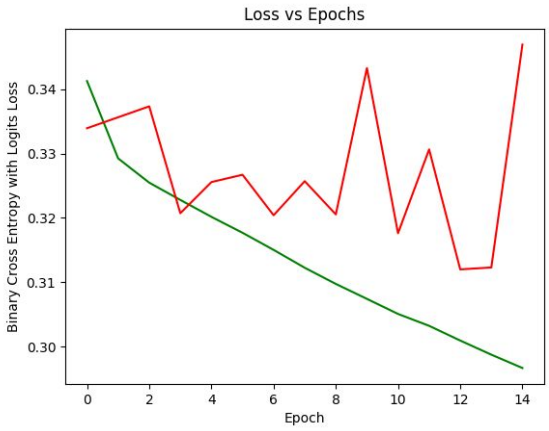

Figure 1: Loss vs Epoch for 15 epochs

[16] https://arxiv.org/abs/1901.07031

4 Epoch Training Attempt

The 4 Epoch training attempts gave us 2 models (model_team8_U-Ones_v1.pt, and model_team8_U-Zeros_v1.pt). Based on our experience in training the 15 epoch model above, we decided to apply early stopping and restrict the number of epochs to 4. Going into this we had applied some additional preprocessing transformations to the images (described above in Preprocessing section), including:

- Histogram Equalization
- Fixed Ratio Resizing
- Gaussian Blurring

For Data Augmentation, we also added in random transformations to account for slight variances in the input images. All the data augmentation transformations were inspired by a GitHub repository, containing code for building models on the CheXpert domain.[17] The repository was used to get first place in Stanford's CheXpert competition at some point, though it is now in third place.[18]

In addition, to help speed up the data loading, we set pinned_memory to True for the DataLoader class provided by the pytorch API. This removes an in-memory copy operation[19] [20] and ultimately sped up our training by approximately 10 minutes per epoch, thus leaving us with 25-minute epochs. Both of the 4-epoch models were trained using a batch size of 16, on a 16 GB graphics card.

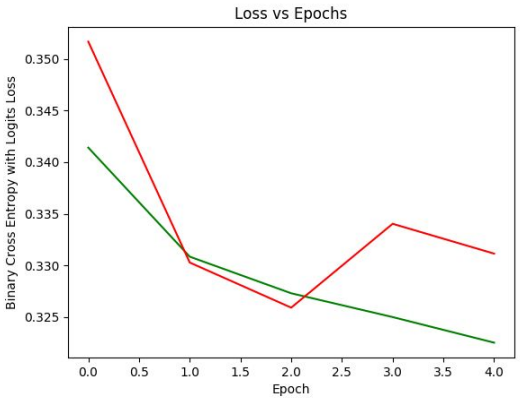

Figure 2. Loss vs Epoch for U-Zeros (4 epochs)

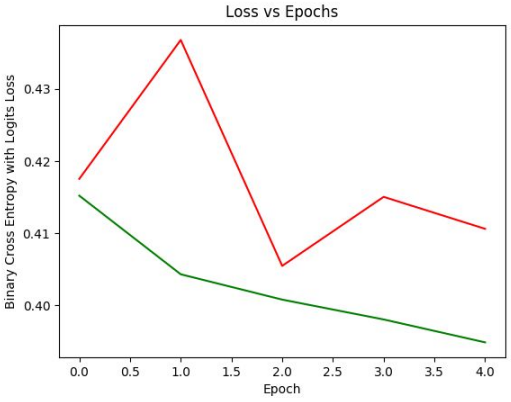

Figure 3. Loss vs Epoch for U-Ones (4 epochs)

[17] https://github.com/jfhealthcare/Chexpert
[18] https://stanfordmlgroup.github.io/competitions/chexpert/
[19] https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/
[20] https://discuss.pytorch.org/t/when-to-set-pin-memory-to-true/19723

45 Epoch Training

Our final models were trained for a total of 45 epochs. Inspired by the same GitHub repository we referenced when putting our preprocessing chain together, we setup a pytorch learning rate scheduler (LRStep) to reduce our initial learning rate by 1/10th every 2 epochs. We started with a batch size of 16 for the initial 15 epochs of training, and then reduced the batch size to 8 for the remaining epochs. (This was actually driven by a desire to have more than one of us be able to train additional models and the lengthy time these training operations ended up taking.) In previous model we had also enabled Shuffle with both our training and validation data loaders; however, it appears that turning that off drastically decreased the variability between our training and validation losses, as shown below.

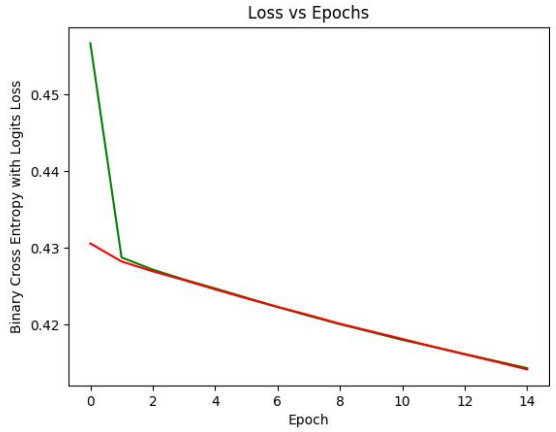

Figure 4. Loss vs Epoch for U-Zeros (first 15 epochs)

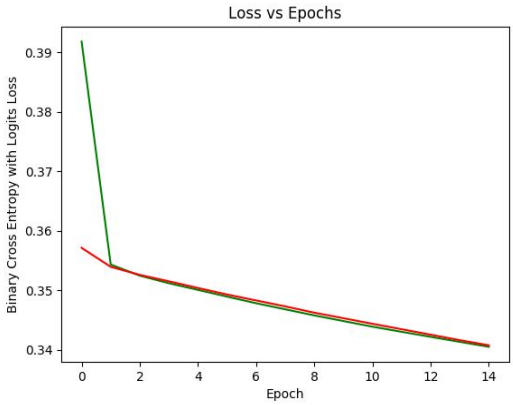

Figure 5. Loss vs Epoch for U-Ones (first 15 epochs)

Inspecting these plots led us to believe that further training would only further improve the models as we saw nice agreement between the training (red) and validation (green) loss functions each epoch. For our next 15 epochs, the only change we made was to reduce the batch size to 8, as mentioned above.

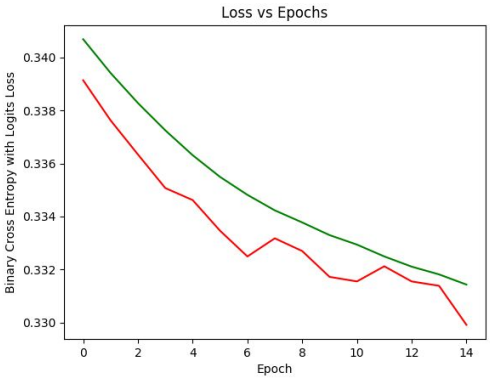

Figure 6. Loss vs Epoch for U-Zeros (epochs 15-29)

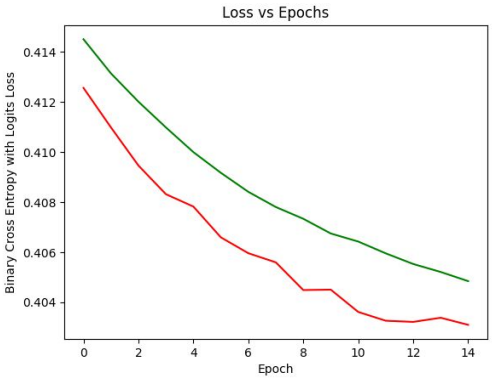

Figure 7. Loss vs Epoch for U-Ones (epochs 15-29)

For our last training iteration we removed the StepLR scheduler from the mix, and kept a constant learning rate. We came to this decision because graphs above indicated we were getting less training returns for each epoch. The consistency of that pattern made us wonder if the training rate reduction is actually stopping us from getting to a minimum as quickly as we otherwise would. Unfortunately, we experienced technical difficulties in training the final models, and despite managing to get the model trained, were unable to generate the Loss vs Epoch plots.

We've summarized the training settings for each "Phase" in the table below, the settings were kept consistent for both the U-Zeros, and U-Ones models:

Model	**Training Epochs**	**Batch Size**	**Scheduler**
v2.pt	0-14	16	LRStep, 1/10 every 2 epochs
v3.pt	15-29	8	LRStep, 1/10 ever 2 epochs
v4.pt	30-44	8	None (constant LR)

Performance Analysis

In order to analyze the performance of our models against the CheXpert test set, we decided to use the Receiver Operating Characteristic (ROC) and Area under the Curve (AUC) for each of the seven labels shared by CheXpert and NIH discussed above.

We get the following performance chart for the 4 epoch models:

Labels	U-Zeros AUC	U-Ones AUC
Pneumonia	0.559735	0.748341
Edema	0.784362	0.789183
Cardiomegaly	0.701984	0.729802
Consolidation	0.893110	0.867632
Pneumothorax	0.654314	0.681416
Atelectasis	0.788393	0.796104
No Finding	0.845999	0.835929

For our final models, we get the following charts for the v2 models:

Labels	U-Zeros AUC	U-Ones AUC
Pneumonia	0.705199	0.678650
Edema	0.762728	0.763316
Cardiomegaly	0.742293	0.738749
Consolidation	0.733152	0.793909
Pneumothorax	0.759956	0.733407
Atelectasis	0.716234	0.736607
No Finding	0.813776	0.813776

The following is the performance chart for the v3 models:

Labels	U-Zeros AUC	U-Ones AUC
Pneumonia	0.560288	0.736726
Edema	0.762375	0.768136
Cardiomegaly	0.718728	0.715804
Consolidation	0.812604	0.853912
Pneumothorax	0.732854	0.697456
Atelectasis	0.755276	0.777273
No Finding	0.853652	0.861305

Finally, the following is the performance chart for the v4 models:

Labels	U-Zeros AUC	U-Ones AUC
Pneumonia	0.552544	0.759956
Edema	0.758730	0.761552
Cardiomegaly	0.719437	0.714918
Consolidation	0.835519	0.861149
Pneumothorax	0.722345	0.709071
Atelectasis	0.754545	0.779870
No Finding	0.853786	0.863856

We could not see much improvement in AUC scores from **v3** to **v4**. Hence we sensed that increasing the number of epochs would not make any significant difference in the model performance.

PPV (Positive Predictive Value or Precision) metrics for the v4 models is as below:

Labels	U-Zeros PPV	U-Ones PPV
Pneumonia	0.932793	0.932793
Edema	0.652367	0.652367
Cardiomegaly	0.503251	0.503251
Consolidation	0.737837	0.737837

Pneumothorax	0.932793	0.932793
Atelectasis	0.433121	0.433121
No Finding	0.759484	0.787700

The above Precision values are mostly the same between the two models. After some digging we discovered that this was likely due to the size of the test set being insufficient to adequately test the model. We were using the validation set provided by CheXpert, which we realized was heavily unbalanced and small (under 250 images). We didn't discover this until late in the project, and weren't able to take steps to correct it in time. In order to test on a larger set of data, we sampled 30% of the data provided by CheXpert for training. Since we only trained on 30% of the data before, our new sample would not be the same as our training data, but disappointingly there is some risk of overlap. This risk of overlap is a flaw, but we decided it was worth the risk in order to test on more data. The results of this test are provided in the following table (the modified code to achieve this is in a branch called `test_on_training_subset`).

Labels	**U-Zeros PPV**	**U-Ones PPV**
Pneumonia	0.945906	0.786100
Edema	0.730872	0.715448
Cardiomegaly	0.774423	0.712859
Consolidation	0.873153	0.655775
Pneumothorax	0.832754	0.809125
Atelectasis	0.723606	0.488606
No Finding	0.861171	0.863509

NIH Dataset Results

Loading and applying the same preprocessing transforms to both the labels and images from the NIH Chest X-Ray Dataset was reasonably straight forward. We get the following results from running our evaluation scripts on the kaggle sample of the NIH Dataset for each version of our models:

4 Epoch Models:

Labels	**U-Zeros AUC**	**U-Ones AUC**
Pneumonia	0.548218	0.681347

Edema	0.804757	0.799956
Pneumothorax	0.683456	0.635095
Cardiomegaly	0.636777	0.637737
Consolidation	0.746635	0.738144
Atelectasis	0.686915	0.692372
No Finding	0.680838	0.677493

V2:

Labels	**U-Zeros AUC**	**U-Ones AUC**
Pneumonia	0.622675	0.656906
Edema	0.740333	0.750593
Cardiomegaly	0.613022	0.613280
Consolidation	0.681655	0.699247
Pneumothorax	0.628742	0.623899
Atelectasis	0.637792	0.658794
No Finding	0.625170	0.626849

V3:

Labels	**U-Zeros AUC**	**U-Ones AUC**
Pneumonia	0.546246	0.690366
Edema	0.804130	0.807935
Cardiomegaly	0.632323	0.636616
Consolidation	0.742122	0.739008
Pneumothorax	0.655864	0.636634
Atelectasis	0.669308	0.678329
No Finding	0.671345	0.673807

V4:

Labels	U-Zeros AUC	U-Ones AUC
Pneumonia	0.536398	0.686517
Edema	0.801280	0.811856
Cardiomegaly	0.630075	0.634492
Consolidation	0.743061	0.741096
Pneumothorax	0.661201	0.642607
Atelectasis	0.671364	0.679177
No Finding	0.673435	0.674797

The following table shows V4 performance evaluation in terms of Positive Predictive Value (PPV) or Precision, as we did with the CheXpert dataset before:

Labels	U-Zeros PPV	U-Ones PPV
Pneumonia	0.978003	0.978003
Edema	0.960727	0.960901
Cardiomegaly	0.950329	0.950329
Consolidation	0.920997	0.920997
Pneumothorax	0.905655	0.905655
Atelectasis	0.826977	0.826977
No Finding	0.636713	0.636242

Conclusion

Overall, our approach of trying to improve the NIH dataset by using a model trained on CheXpert still seems valid. However, we have concerns about the strength of our specific models. On the one hand, we had high AUC values when using the models on the CheXpert test set, which indicates predictive power. On the other hand, our high precision values are likely driven by relatively high null accuracy rates for some labels. In our test set, for any given label, most of the images have a value of zero. We realized too late that our CheXpert test set was small and imbalanced. This means that the model would usually predict a zero for any label. This would automatically lead to higher precision, as there would unlikely be a false

positive. Another suspicious factor was that U-Zeros and U-Ones produced almost the exact same precision values as each other when running them on the NIH data. Therefore, we are not confident enough of our models predictive capability to believe that we have achieved a better labelling of the NIH data.

Future Improvements

First and foremost, the best way to improve our understanding of problem set would be to go back and separate training and testing subsets out of the data provided by CheXpert for training, to ensure enough data is available for testing.

Digging a bit deeper into the model itself, we believe we could see vast improvements by adapting a Decision Tree based ensemble constructed based on various backbone models. This concept is loosely inspired by what we understood of the paper describing the current first place entry in the Chexpert competition[21]. Further, based on the codebase provided for what is currently the third-best entry in the competition, performance could be improved by including a "Feature Pyramid Attention Network" which was apparently being used by that model.

Perhaps the most reasonable approach would be to build on what others have done entirely, instead of training a separate model incorporating only snippets of the ideas they have had success with. It may be best to incorporate their code into a vast ensemble model, using either a simple voting scheme, or even a Logistic Regression to perform the ensembling. Such an ensemble based on perhaps the top 5 models, or even a cluster of models that agree the least despite all having reasonably high scores would almost have to get some sort of improvement.

[21] https://arxiv.org/abs/1911.06475