CheXpert: AI and Healthcare

Utilizing Deep Learning to Detect Cardiomegaly from Medical Images

By Joseph Earnshaw

Capstone Project 2: Project Proposal

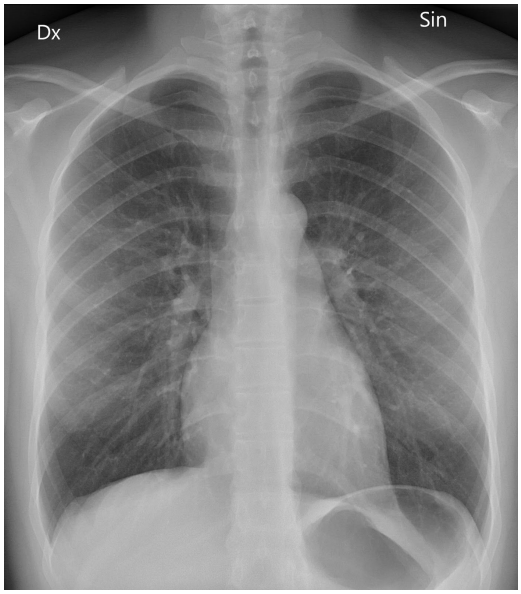


Image By Mikael Häggström

Background

For the past two years, I have been a CrossFit coach. I've spent thousands of hours helping others create healthier and happier lives. I don't say that lightly. It has allowed me to see people go from having difficulties doing simple activities like getting around the house to being able to play with their kids (or grandkids), which has been an incredible experience!

While my last project was undoubtedly fascinating, and in a sense helped me rediscover the passion I had for the game of basketball, the impact it has outside of the NBA is next to nothing. For my next project, I wanted to get back to my roots and revisit my original motivation for pursuing a career in data science which was to figure out how to help people live healthier and happier lives.

Recently, machine learning researchers at Stanford developed CheXNeXt, a deep learning algorithm that can concurrently detect a range of diseases in chest radiographs. Its performance in detecting specific pathologies was comparable to a group of practicing radiologists. Perhaps more impressively, it was able to do this in significantly less time (~two minutes vs. ~4 hours for radiologists).

Why is this so important? To begin, two-thirds of the world's population has no access to 'basic' X-ray or ultrasound examinations according to the World Health Organization. Even most developed countries around the world are experiencing a shortage of qualified radiologists, with up to 97% of radiology departments unable to meet diagnostic reporting requirements. With an increased workload brings the increased chance of error due to stress or fatigue.

In summary: there are not enough doctors to meet the ever-increasing demand for medical image analysis.

However, the primary issue when it comes to deep learning and medical image analysis has been that it requires a lot of data, which hadn't been accessible to train a reliable model. Then in 2017, the NIH Clinical Center released its ChestX-ray14 data set, which contained over 112,000 frontal-view chest radiographs from nearly 31,000 unique patients.

Not to be outdone, Stanford released CheXpert, a public dataset consisting of 224,315 chest radiographs from 65,240 patients with MIT co-releasing its MIMIC-CXR with nearly 372,000 chest x-rays! Additionally, the Radiological Society of North America released a data set on Kaggle for a competition to see who could build the best algorithm to detect potential pneumonia cases. So with more and more data becoming available, the opportunity to create a reliable deep learning model that can accurately identify pathologies is there, as evidenced by CheXNeXt.

In my opinion, this is an enormous opportunity to not only become familiar with a growing technology but to also become a part of the current revolution in healthcare. Additionally, it presents a chance to become familiar with working on a cloud computing platform as Google has the NIH data set available through its Cloud Healthcare API.

Summary

What is the problem you want to solve?

- Create a model that can detect pathologies from medical images at the same level as practicing radiologists.

Who is your client and why do they care about this problem? In other words, what will your client do or decide based on your analysis that they wouldn't have done otherwise?

- My clients would be healthcare systems, and more specifically radiologists, who need to review an ever-growing number of medical images. A model that shows evidence it can perform at a high-level could be used as a tool to assist in examining images, allowing for a more efficient workflow. This has a direct impact on patients as well, who are able to receive a diagnosis (and potentially treatment) much sooner.

What data are you using? How will you acquire the data?

- There are a few data sets available, all of which are publicly available.
    - CheXpert
    - MIMIC-CXR
    - RSNA's Kaggle data set
    - NIH's ChestX-ray14
- I'm currently in the process of examing the data and have not determined which one would be the most appropriate for this project
- Want to quickly note as well that deep learning will be utilized for this project and NIH's data is easily available via Google Cloud's Cloud Healthcare API.

Briefly outline how you'll solve this problem. Your approach may change later, but this is a good first step to get you thinking about a method and solution.

- I'll begin this project by looking further into convolutional neural networks, which was the basis for CheXNext to see if I can replicate their performance. While I hope to develop something that can at

least attempt to predict all the pathologies laid out by the team at Stanford, the complexity may be too much for me at this point. To keep things simple, the algorithm that I develop may only predict for a specific disease, like pneumonia, for example. However, only time and further exploration will tell.

What are your deliverables?

- In the end, I plan to deliver my code (via Jupyter Notebooks), a comprehensive paper that details the entire process and a slide deck to present my findings. More importantly, I hope to take a step towards helping others live healthier lives.

Head In The Clouds?

Data Acquisition and Cloud Computing Set-up for Capstone #2

For my second capstone project, the primary objective for my second capstone project is to create a deep learning model that can detect a pathology[1] from medical images at approximately the same level that Stanford's Machine Learning group achieved.

Achieving an AUC of 0.907 is not an easy feat, but I'm very excited about this project, primarily because it'll allow me to begin to experiment with both cloud computing and deep learning, two of the 'hottest' subjects in technology right now. Also, the two mostly go hand-in-hand as deep learning requires a significant amount of computing power and hardware via GPU's[2] that most home computers like my iMac don't even compare. Even though I was slightly intimidated at first to jump into what I perceived was the deep end of the technological swimming pool, I closed my eyes and took that leap of faith.

Before I go further in regards to setting up my data and compute instance in the cloud, I first want to discuss why I chose CheXpert over the other data sets I mentioned previously in my project proposal.

Firstly, gaining access to the data was relatively simple. All one has to do is go to this webpage, scroll to the bottom, fill out the necessary information and you'll be emailed a link to download the zip file that contains all the images from the study. In contrast, acquiring the MIMIC-CXR data set from MIT was a multi-step process that involved passing multiple ethics quizzes and an approval process that could take weeks.

The ChestXray14 dataset was also relatively easy to access[3], but upon further investigation, it looks like this particular data set doesn't seem quite up to snub. Luke Oakden-Rayner, a Ph.D. candidate and Radiologist, pointed out multiple problems with CXR14, one of them being a problem with the labeler that extracted the pathologies for each image from the associated medical report.

---

[1] Or potentially pathologies; with an additional number of potential classes the model building process does get more complicated and I am working with limited resources

[2] Or TPUs

[3] It can be download via Kaggle or accessed via GCP's Cloud Healthcare API

Upon further visual inspection of the images, Oakden-Rayner also found many inaccurate labels from randomly chosen sets of images for the pathologies indicating the potential of a high error rate in the labeling process[4].

Instantly, my 'garbage in, garbage out' senses were tingling.

Upon some further investigation, I came upon a few other complaints[5] about CSXR14's quality, which ultimately led me to decide not to use it for my project despite it being readily available.

However, I want to stress that CheXpert is not perfect. While a significant improvement on CXR14's labeler, CheXpert labels were also extracted via NLP[6], which has an irreducible error due to inaccurate/nondescriptive medical reports. Additionally, there are several repeat scans which reduce the effective size of the data set along with issues related to the accurate visual interpretation of the images due to downsampling[7].

However, Oakden-Rayner comments that this dataset is a significant improvement upon CXR14 and that CheXpert is quite possibly the best medical imaging data set that is publically available. In summary, this data set may not be quite ready for training full-fledged AI models -- after all, we are talking about life and death -- but it does present an excellent opportunity for continuing research in utilizing deep learning for medical image analysis.

Now, with data in hand, the next question is: what cloud service to use? Amazon Web Services is by and large the leader when it comes to cloud computing, with other options like Microsoft Azure, GCP[8], IBM, and Oracle taking up the rest of the market.

For this project, I've decided to utilize Google Cloud Platform mainly due to the $300 credit that Google gives new users to try out their platform. Another influence was Google's focus on AI as a company and perhaps more so than the other providers, has optimized its cloud platform with this approach in mind. I'd also been utilizing the GCP platform for fast.ai, a MOOC created by Jeremy Howard in hopes of democratizing deep learning. The combination of ease-of-use, previous hands-on experience, and the $300 credit made picking GCP a no-brainer.

With data downloaded and cloud account setup, the next step was uploading the data set (i.e., CheXpert) to 'the cloud.' As a beginner, I found this first task to be quite daunting; however, after reading plenty of documentation combined with a little playing around, the process was relatively straight forward.

The first step is to create a bucket, which are the primary containers that hold your data in GCP. A user can create a bucket via the GCP console, command line using gsutil, the REST API, or even with Python code!

Additionally, I would like to point out that I utilized a multi-regional storage class. What does this mean and what even is a storage class? It is a property of your bucket that applies to how objects (i.e., data) added to the said bucket are stored.

---

[4] See the section titled ‘Part 1: Visual label accuracy in ChestXray14’ in Oakden-Rayner’s post for in-depth analysis
[5] See here and here for a few examples
[6] Natural language processing
[7] Images are 8-bit png images with 256 grey levels
[8] Google Cloud Platform

For this project, I utilized multi-regional storage, which is best for data that is frequently accessed, otherwise known as "hot" objects[9]. The benefit of utilizing multi-regional storage is that it provides the highest data availability, which also means that it comes at the highest cost. The storage cost was moot though; the cost was only $0.026[10] versus $0.020 for the regional storage[11]. Since my data was only ~11GB, storage cost was going to be minimal no matter which option I selected. However, if I were to have stored hundreds of GBs or even multiple TBs of data, then I would have had to take storage classes into further consideration.

After my bucket was created -- with the original name of 'joes-capstone2-data' -- I could then upload my CheXpert data as a zip file. By drag and dropping the file from my local desktop process and the upload started. After approximately 1-2 hours - due to a somewhat limited personal internet bandwidth - my data was ready to be accessed.

At this point, we have our data in the 'cloud' but you may be wondering: how do we, you know, access it and begin to analyze it via a tool like Jupyter Notebooks? Answer: Google's Compute Engine!

Compute Engine allows us to create high-performance virtual machines, which you can think of, in a sense, as your personal computer in the cloud. The benefit of this setup is that it allows you to personalize the hardware you're using, which, in our case, allows us to attach GPU's to our virtual machine. GPUs, or graphics processing units, are beneficial when it comes to deep learning as they are more efficient, in both processing and cost, than regular CPUs[12].

The setup was more complicated than the previous steps and did require more patience. There are two primary routes you can take when creating a VM instance: command line or the GCP console. The command line route requires the user to install Google's CLI (command line interface) and configure the Google Cloud SDK on your local machine. After this, using some command line magic, you can create an instance.

The other option is via the GCP console and setup is similar to when we created the bucket, albeit with different options we need to select. After you click on 'Create Instance,' the user interface takes you to a screen that gives you options such as Region, Zone, and Boot disk, amongst others. The most important feature though is the Machine type, which is where we pick the hardware of our virtual computer.

In the end, this could be a frustrating part of the project, especially for individuals that are not too familiar with cloud computing (of which I belong to!). Luckily though fast.ai made this process more straightforward and was instrumental in helping me get my compute instance up and running.

Once you have the instance setup, it's (almost) as easy as hitting the play button! After navigating to the Compute Engine/VM instances interface in GCP, the user sees a list of their available VMs. Check the box next to the one you wish to use and hit the play button on the top of the page. Your instance is ready once there is a green check mark next to it, after which you enter the following command:

```
gcloud compute ssh --zone=$ZONE jupyter@$INSTANCE_NAME -- -L 8080:localhost:8080
```

[9] More information on storage classes can be found here
[10] Per GB per month
[11] Data is stored in a narrow geographic region
[12] More information can be found here

After this, a Jupyter Lab/ Notebook is at your disposable! You can then access your bucket, download and unzip the CheXpert data and begin analysis. This entire process can be seen in more detail in the following Jupyter Notebook

‘Chex’ This Out!

Exploratory Data Analysis on CheXpert Data Set

For my second capstone, I’ve utilized the CheXpert data set via Stanford’s Machine Learning Group. This particular data set represents one of the most significant steps in recent years to advance research in regards to deep learning within the healthcare space. Before we jump into creating our model, let’s take a more in-depth look at the patients themselves. After downloading CheXpert, you see that it includes a train.csv and valid.csv file (see below).

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 223414 entries, 0 to 223413
Data columns (total 19 columns):
Path                            223414 non-null object
Sex                             223414 non-null object
Age                             223414 non-null int64
Frontal/Lateral                 223414 non-null object
AP/PA                           191027 non-null object
No Finding                      22381 non-null float64
Enlarged Cardiomediastinum      44839 non-null float64
Cardiomegaly                    46203 non-null float64
Lung Opacity                    117778 non-null float64
Lung Lesion                     11944 non-null float64
Edema                           85956 non-null float64
Consolidation                   70622 non-null float64
Pneumonia                       27608 non-null float64
Atelectasis                     68443 non-null float64
Pneumothorax                    78934 non-null float64
Pleural Effusion                133211 non-null float64
Pleural Other                   6492 non-null float64
Fracture                        12194 non-null float64
Support Devices                 123217 non-null float64
dtypes: float64(14), int64(1), object(4)
memory usage: 32.4+ MB
None
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 234 entries, 0 to 233
Data columns (total 19 columns):
Path                            234 non-null object
Sex                             234 non-null object
Age                             234 non-null int64
Frontal/Lateral                 234 non-null object
AP/PA                           202 non-null object
No Finding                      234 non-null float64
Enlarged Cardiomediastinum      234 non-null float64
Cardiomegaly                    234 non-null float64
Lung Opacity                    234 non-null float64
Lung Lesion                     234 non-null float64
Edema                           234 non-null float64
Consolidation                   234 non-null float64
Pneumonia                       234 non-null float64
Atelectasis                     234 non-null float64
Pneumothorax                    234 non-null float64
Pleural Effusion                234 non-null float64
Pleural Other                   234 non-null float64
Fracture                        234 non-null float64
Support Devices                 234 non-null float64
dtypes: float64(14), int64(1), object(4)
memory usage: 34.8+ KB
None
```

Above are info() outputs from EDA notebook (see here)

Above you can see the high-level information from the provided training and validation CSV files. The training set, which we'll use to train our deep learning model, has nearly 223,414 entries with 19 columns. Most of these columns are devoted to the specific pathologies like enlarged cardiomediastinum and atelectasis to support devices (i.e., this is technically not a pathology but was included by the team at Stanford based on its prevalence within the images).

The `Path` column represents the 'path' to the image, which will come in handy once we get to the model training portion of the capstone. The information in this column assists the learner to connect an observation from the CSV file to the associated image file within our data folder.

There are two somewhat distinct columns - sex and age - that represents whether the patient was male or female and the patient's age, respectively.

The next two columns indicate the positioning of the medical image. In this specific data set, x-rays were done from either the front (i.e., frontal) or the side (i.e., lateral), which is indicated by the Frontal/Lateral column. Lastly, the AP/PA column shows whether or not the patient was standing or lying down when the imaging was done (which we'll dive further into a little later).

Another critical thing to make a note of is that the training set had quite a few NaNs, meaning that there was missing information. However, upon further investigation of Stanford's Machine Learning GitHub, we can clarify

that missing values mean that no mention of that particular pathology was extracted via the labeler. Stanford treated those cases as negative (i.e. `0`) when they created their model so we did the same and relabelled them as `0` for these pathological columns.

What does AP/PA mean?

Within the data set there was one column -- AP/PA -- where ~14% of the observations were missing values, so we decided to dig deeper. While 14% may not sound like a lot when you consider there are approximately 223k observations that 14% adds up to a little over 32,000 missing values. For most data sets, there are specific techniques you can use to address missing values - replace with the mean or median, use linear regression to predict the value, etc. - but with this data set, we're somewhat stuck because we're dealing with images and can't merely input a number.

As a reminder AP/PA has to do with the positioning of the medical image so if we wanted to maintain the integrity of our data (at least within the CSV file), we'd have to go through each image and manually label them one by one, preferably with the assistance of a domain expert (i.e., board-certified radiologist(s)). Now, what exactly do PA and AP mean?

PA stands for posterior-anterior, which is obtained when the patient is in a standing position, facing the cassette and the x-ray tube is approximately 72 inches away.

AP stands for anterior-posterior, which is when the patient is lying down, and the x-ray tube is only 40 inches from the patient.[13]

In summary, it indicates whether or not a patient was sitting or standing when the image(s) were taken. In a traditional setting (i.e., a hospital or healthcare facility), radiologists have access to top of the line medical imaging equipment, which allows them to analyze images at a very high pixel rate. Unfortunately, we do not have access to these high-resolution images because that would require a significant amount of storage. The small downsampled version is 11 gigabytes, compared to 439 gigabytes for the 'original' version of the images, which are not downscaled but reduced to 8-bit png images. What does this mean? These 8-bit images have 256 grey levels; for comparison, clinical x-rays are stored at 16-bit or 65,536 grey levels!

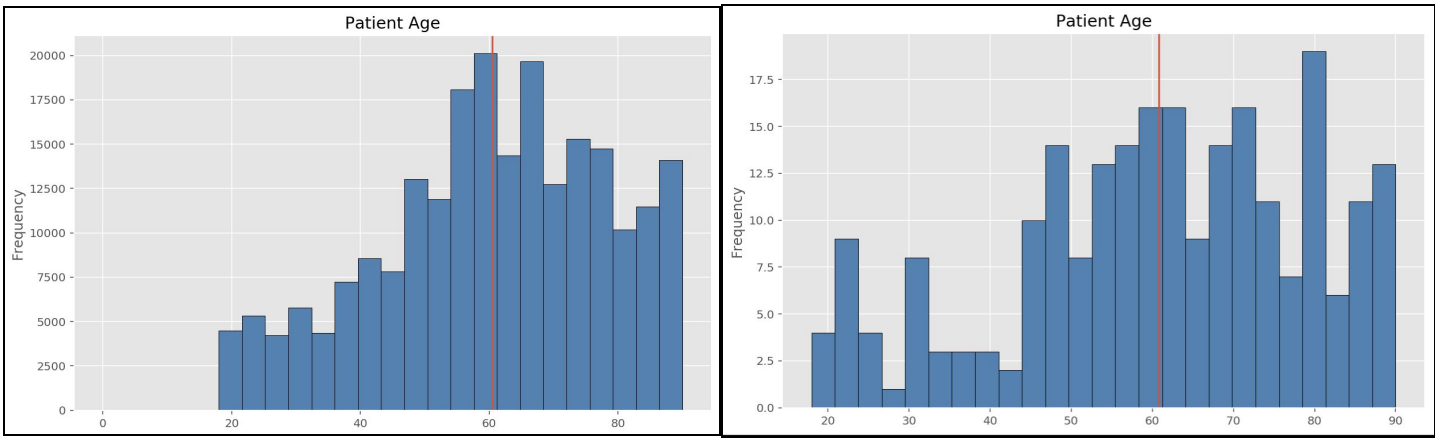
Additionally, a tiny number of observations were labeled as either ‘LL’ or ‘RL’. What did these mean? From what I could gather from my investigation is that these particular observations were done laterally in that the central ray passes from one side of the body to the other through the axial plane.[14]

However, I could not find the following patients in the below data frame upon further inspection. Within the downloaded ‘train’ folder, there were no patients with the corresponding patient ID numbers. It looks as if these images were not included which is odd, but due to the minimal number of observations (~17) that were labeled either ‘LL’ or ‘RL’, I'm not too worried about it. Even if there were some information, its effect on the model would be negligible at best.

---

[13] Source
[14] Source

Observations: Patient Age



Histograms showing age distribution of patients within training set (left) and validation set (right)

The first aspect that sticks out about the histograms above is that around age 50, there is a significant uptick in the number of cases as evidenced by the higher bar heights. The bins climb steadily until the age 60 bin (which contains a range of ages around 60), which serves as the peak in the training set (top histogram) and represents the second highest peak in the validation set (bottom histogram).

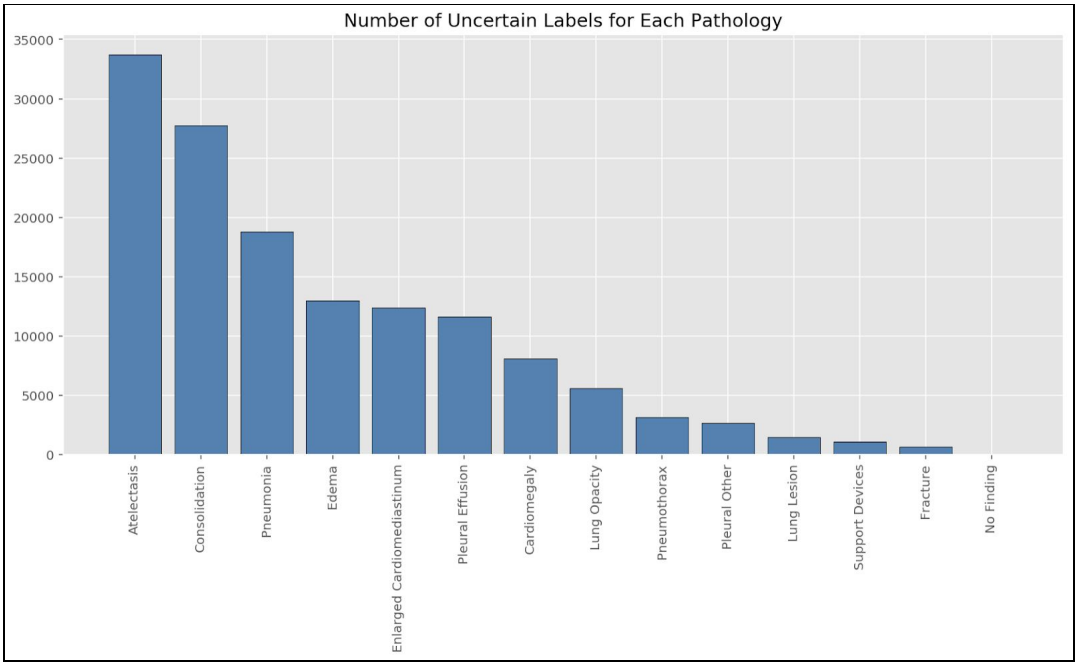
The red line that vertically cuts through both the histograms represents the mean age of that particular cohort. They both look to be around the same at slightly over 60 years of age. When we take a look at the training sets (i.e., top) histogram, we can see that the values on the x-axis start at 0. Why was this? An initial hypothesis was that there were observations that may have an age of '0.' It turns out this wasn't entirely the case, but there were three observations below the age of 18, which was skewing the histogram.

When we compare the two histograms, we can see some slight differences in their distribution. While both appear to peak at approximately the same point (around age 60) the distribution to either side is different. For ages greater than 60, the training set has a downward trend with a spike in the number of individuals whose ages are in the late-80s. For the validation set's histogram, there are spikes around every decade, i.e., age 70, 80, and 90.

Now let's take a look at the lower end, the observations whose age is less than 60. From the ages of 20 to 40, the number of observations in the training set remains pretty consistent with a slight uptick from approximately ages 40 to 50. This is slightly different however from the validation set which again shows an uptick around the decade marks of age 20 and 30, then a sudden spike around the late-40s.

There is one thing that we have to keep in mind though when comparing the training and validation set: the training set is ~223,000 observations while the validation set is only 234, which is approximately ~0.1% of the training data. Overall, the two histograms are not the same but are relatively close in shape, which is pretty spectacular, considering the gap in the number of observations between the two data sets.

Observations: Uncertainty Amongst Pathologies



Graph indicating the total number of uncertain labels for each pathology

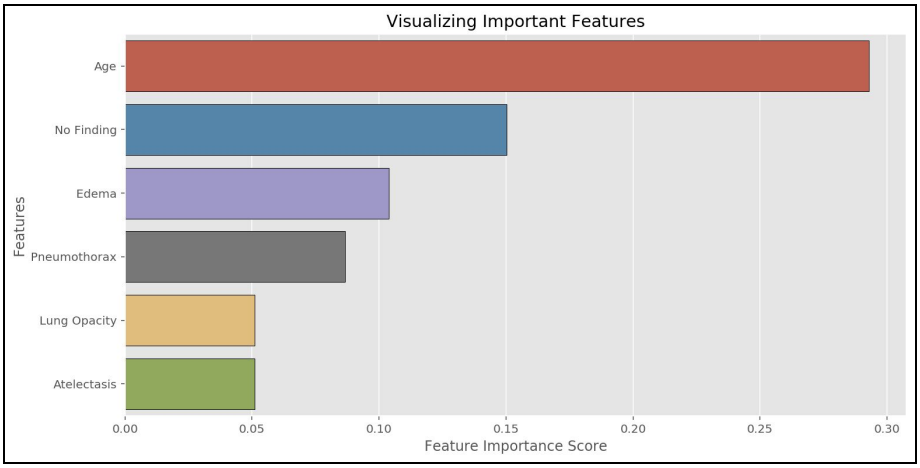
The above graph shows in descending order the pathologies with the most uncertain labels. Now is a good chance to mention how the team at Stanford went about labeling the chest x-rays. In the accompanying radiology reports, they developed and applied an automated, rule-based labeler to extract the labels. Observations mentioned at least once were positively classified (i.e., 1) while the presence of at least one negatively classified mention resulted in a negative classification (i.e., 0). In the case that a pathology was mentioned but could not be confirmed as either positive or negative, it was assigned an uncertain label (i.e., -1).

As we can see from the graph above, atelectasis, consolidation, and pneumonia were the top three pathologies in terms of uncertain labels. Because they are technically neither positive nor negative, this begs the question: what do we do with these uncertain labels?

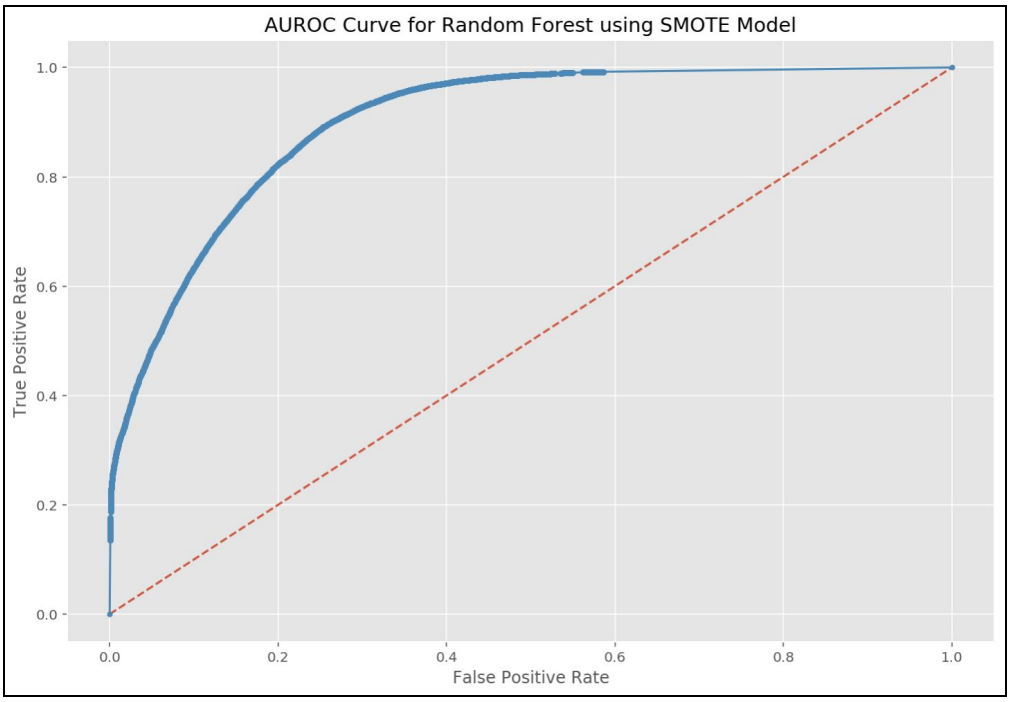
There were a few different strategies that the Stanford team suggested:
Ignoring: simply drop the uncertain labels during training
Binary Mapping: map all uncertain instances to either negative (U-Zeroes) or positive (U-Positive)
Self-training: build a baseline model with the ignoring approach, then relabel uncertain labels according to predictions made by the model
3-Class Classification: treats uncertainty as a class, turns original binary classification problem into a multi-class classification problem

Motivated by the self-training technique, we asked the question: what if we could generate a machine learning model from the information provided so far (i.e., from just the CSV file)?

Can we use ML to label uncertain Pathologies?



Feature importance scores from Random Forest using SMOTE model



ROC curve of Random Forest model that utilized SMOTE due to highly unbalanced label distribution of cardiomegaly

For this portion, we decided to use the random forest classifier due primarily to is versatility in handling different types of features, it requires minimal input preparation and is routinely one of the best performing machine learning algorithms. However, we remembered the adage ‘garbage in, garbage out’ and decided to clean up certain parts of the data to hopefully help generate a more accurate model. Below are a few of the modifications we made to the data set:

- Removed all observations that contained any uncertain pathological labels
- Converted all the pathology columns to categorical types
- Dropped Path, Frontal/Lateral, AP/PA, uncertain_features columns
- Converted categorical variable within Sex column into dummy/indicator variables

After this, we created the variable X, which contained all the features of the data set, and the variable y, which included our target variable -- the cardiomegaly column.

We created three models: a baseline random forest, a random forest utilizing an oversampled data set, and another random forest that utilized an oversampled data set using SMOTE. The reason for the oversampling was due to the relatively poor performance of the baseline model. The baseline random forest had a mean ROC-AUC of 0.743 on the test set but a mean score of 0.363 and 0.109, respectively, for the precision and recall metrics. Due to the highly uneven nature of the cardiomegaly labels -- approximately 88% of the observations were negative -- we decided to utilize oversampling, which is a technique that adds more examples from the minority class (i.e., the positive label). The hypothesis was that it would help the algorithm better detect that patterns that led to an observation being labeled as either positive or negative.

Using the sample function, which randomly sampled with replacement from our current data, we were able to generate a data set that had the same number of positive cases of cardiomegaly as negative by using the observations already available to us. Utilizing this strategy, we were able to significantly improve on the baseline, with a ROC-AUC score of 0.848, 0.727 for precision, and 0.88 for recall.

For the last model, we utilized SMOTE, which stands for synthetic minority oversampling technique. In contrast to the sample function, SMOTE takes it a step further by synthesizing points for the minority class from those that already exist. It does this by randomly picking a point from the minority class and then computing the k-nearest neighbors for that point. After that, it creates synthetic points between the chosen point and its neighbors. Essentially, we are creating ‘new’ patients!

With this technique, we were able to achieve the best performing model: a random forest that had a mean ROC-AUC of 0.896, mean precision of 0.776, and a mean recall of 0.886. Additionally, the hyperparameters of the random forest were not further tuned, so there is potential for achieving even better performance.

While this is showing promise, I have yet to determine if I want to utilize it to re-label the uncertain labels. I believe the best course would be to start with one of the uncertainty methods that the Stanford team utilized (i.e., ignoring, binary mapping, etc.) and depending on the results, then experiment with random forest mentioned above (after hyperparameter tuning).

Developing A Deep Learning Model: The Beginning

If you venture over to the repository for my second capstone, you'll notice a folder called `playground_nbs`, which contains all of the experiments I conducted during the process of developing the deep learning model. In summary, some ideas worked, and some didn't; those that did, I took to the next notebook and tinkered with another tactic to try and increase the model's performance.

It was a fascinating process, and I enjoyed the process of experimenting with things like different architectures, learning rates, data augmentation, over/undersampling, and numerous other processes that I described in more detail later. Sure, it was hard at times to wait for the training to finish and could be slightly demoralizing when the performance didn't improve or even got worse...

That is what being a scientist is all about, testing hypotheses and seeing if they work because what may sound good in theory may not always turn out so great in practice!

Before I go any further though, I want to give a shout out a few sources that were significant sources of help getting this project off the ground.

- Simon Grest, walking through your notebook and seeing the results was amazing! Additionally, the tactics you used to create the data set, sample it, and utilize the validation data set to test during training, were a significant help! If interested, click this link to check out his work.
- Kerem Turgutlu, your setup for the ImageDataBunch helped me get a better understanding of the parameters involved with `get_transforms` and allowed me to tinker when it came to data augmentation. Additionally, your question regarding what to do with the missing values helped me address right away the correct approach. If interested, click this link to check out his work.

Lastly, to the team at fast.ai, thank you so much for all the work you've done and continue to do! After completing the Practical Deep Learning for Coders course, I was able to put together this project and generate a model with relatively high performance. There is much more to learn, but you've been instrumental in setting a solid foundation for my future growth. I cannot thank you enough!

Learning To Fly

How many times do you think it took the Wright brothers to get an airplane airborne? How many years did it take Einstein to develop the Theory of Relativity? Perhaps better yet, how many Jupyter Notebooks did it take Joe Earnshaw to create a good deep learning model?

Like the Wright brothers and Einstein, it wasn't an overnight process that's for sure!

There were a few fundamental aspects we needed to address, including:

- *Data wrangling*: We needed to make sure the accompanying CSVs for the training and validation sets wouldn't hinder the learner. In our case, since this a deep learning computer vision task, the CSVs are not the primary data source; however, it is always best practice to ensure data is as clean as possible.
- *Reproducibility*: This is a crucial aspect considering that we're going to be randomly sampling from the data. Why is this so important? Well, if we want to accurately compare the performance for two different

model architectures, for example, we need to input the same data for both. If we skip this, comparing performance is null and void because you're comparing the models based on two different data sets!
- *Sampling*: 224,000 images is a lot, and this presented the potential for long training times. We want to be able to iterate quickly and test new techniques without having to wait too long for training.
- *Uncertainty*: There were uncertain labels in the training data set, which means the labeler couldn't confidently tell from the medical documentation where there was a particular pathology present or not. We need to determine a strategy to address these observations with the hope of maintaining as much information as possible.
- *Performance Metrics*: The team at Stanford utilized AUROC (Area Under the Receiver Operating Characteristics) which we'll also use. However, are there any other metrics we can use to help us get a better idea of the model's performance?

Cowboy Up! - Wrangling the CheXpert Data Set

This was arguably the most challenging task of the entire project, and most of the experimental Jupyter Notebooks were devoted to determining different strategies to get the data in a format suitable for training the deep learning model. All this experimentation though paid off though and culminated in the creation of the `sample.py` Python script.

By merely calling `from capstone import sample` we can access all the custom functions created to handle wrangling the CheXpert CSV data. However, there is a prerequisite step before we can utilize these handy functions, and that is to set the `path` variable.

What this variable essentially does is take note of the location of our data, which in this specific case is `'/home/jupyter/springboard-capstone-2/data'`. This was configured using fast.ai's `Config.data_path()`, which allows us to create a path both so we can access the location where our data (i.e., the X-rays) is stored and to save our model weights during training. By default, it creates a temporary, and hidden, folder called `.fastai` but after a little bit of playing around, we set the path to the one mentioned above. The main benefit of this is that we were able to access a permanent folder where we initially stored the data. This version of the CheXpert data set is approximately 11GB, so it would've been incredibly time-consuming to have to re-upload it every day to the default temporary folder to work on it.

At this point, I also want to mention though that playing around with the path is a somewhat tricky (and potentially lengthy) ordeal, especially for someone not used to dealing with folders and working directories. Another strategy is to set the path variable using the `os` Python library. There is more documentation for it which makes it more approachable as opposed to the `Config.data_path()` technique.

At this point we now have our `path` variable which is set to `'/home/jupyter/springboard-capstone-2/data'` and this gives us the ability to use one of our custom functions finally! The following code is all we need to address the vast majority of our data wrangling needs:

```
train_df, valid_df = sample.prep_data(path);
```

Yes, one line is all we need! How is this possible you may ask? Well, do you remember the call we made above, `from capstone import sample`? This command allows us to access the custom functions in the `sample.py` Python script. Pretty cool, huh?

The only required input for the `prep_data()` function is the path variable, and once it has that it can go to work. Below are the steps it takes to return a cleaned up training and validation pandas DataFrame:

1. Reads in the `train.csv` and `valid.csv` files using `pd.read_csv()` and stores them in `train_df` and `valid_df`, respectively.
2. Adds a `valid` column to both data sets, with the value of `False` assigned to the training set and `True` assigned to the validation set.
3. Extracts the patient id from the `Path` column and assigns to a new column called `patient`.
4. Creates a list of all the pathologies and fills any `NaN` values with 0 (which is the same approach that the team at Stanford used).
5. Converts all the pathology columns to integer types (were originally floats).
6. Gathers the counts of the values (i.e., 0, 1, or -1) in the `Cardiomegaly` column.
    - As a reminder, this is our target variable, as we're trying to develop a model that detects this particular pathology within the X-rays.
7. Replaces the -1 (i.e., uncertain) labels within the training set with 0 (i.e., negative) label.
    - This particular strategy for addressing the uncertain labels was known as `U-Zero` in the Stanford paper. I'll go into further details in a little bit, but for now, know this strategy produced some of the best results for Stanford's model.
8. Asserts that the replacement of uncertain labels to negative was correct (i.e., returns a True/False statement).
9. Presents further detailed information related to the reclassification of uncertain labels, from both pre and post-relabeling.
10. Returns `train_df` and `valid_df`

Are you sure about that? - Approaches to Uncertain Labels

Before we go any further, I want to take a deep dive into what we did with the uncertain labels within the training set. As mentioned above, we replaced all the Cardiomegaly observations that had a value of -1 with 0. Where did this idea come from? In their paper, the team at Stanford mentions a few different approaches that could also be used for the uncertain labels, which include:

1. Ignoring
2. Binary Mapping
3. Self-Training
4. 3-Class Classification

The first approach, ignoring, is simplest: drop all uncertain labels during training. However, its simplicity is overcome by its potential to induce significant information loss. The second approach, binary mapping, is the one we used. It says to map all uncertain labels to either 0 (*U-Zeroes*, which we used) or 1 (*U-Ones*). This is still relatively simple to implement and allows us to keep all our images. However, it could potentially distort the learner due to images being mislabeled. As a result of the misclassification, the model's performance could be degraded.

While not used for this specific project, two more advanced approaches could be utilized to address the uncertainty: self-training and 3-class classification. With the self-training approach, you train an initial model using the ignoring method until it reaches some performance threshold and using this model, relabel the uncertain labels with either 0 or 1. 3-class classification takes the next step and treats the uncertain labels as

their own class, turning what was a binary classification problem into a multi-class one (as indicated by its name).

In the end, why did I choose to utilize the *U-Zero* uncertainty approach? Well, it had the second-highest performance in terms of AUROC amongst the various approaches, plus its simplicity made it a logical first choice. However, there is one other route that would be interesting to explore in further analysis. During the exploratory data analysis portion of this project, we're able to create a random forest model utilizing the data from the training data set that could predict cardiomegaly with an AUROC of 0.896, which is very strong. In theory, we could use this model to relabel the uncertain cardiomegaly observations, which is similar to the self-training approach. It would be interesting to see if this could provide any significant bumps in performance.

Fool Me Once - The Issue of Reproducibility

Since we are using samples of the data set, we need to set the randomization seed for the environment. By setting the seed, it helps not only with the stability of whatever particular environment we're working in but keeps our results consistent across multiple trials.

To put it more clearly, I'll use a hypothetical situation where we're trying to compare the difference between model X and model Y. Each uses a different architecture with model X being a ResNet and model Y being a DenseNet. We train the models and then get the results, with model Y performing slightly better than model X. There's one caveat though: during training, each model utilized a different sample from the data set. Now the question becomes, can you compare accurately assess the performance of the two models?

The answer is 'not really' because they were trained on different samples. Now, model Y could very well be the better performer, but you can't say that with too much confidence because the data set it used may have contained better information than model X's data. To fix this situation, you would want to train both model X and model Y on the same sample of the data. If model Y's performance is again higher than you can more confidently say it is the better of the two.

So how do we ensure we generate the same random sample? We can set the randomization seed through our imported `sample.py` script with the following line of code:

```
sample.set_seed(58)
```

This sets the seed for all of the following modules within our environment:

- `random.seed`
- `os.environ['PYTHONHASHSEED']`
- `np.random.seed`
- Multple components of `torch`

In summary, by using this function, we're maximizing our reproducibility, which gives us more confidence in determining what might be helping or hurting a respective model's performance.

To Over or Under-sample? Dealing with Unbalanced Data

To be blunt, the models weren't good during the early stages of this project. At first, I was a little confused by this, but then I decided to test the hypothesis that the unbalanced data might be negatively impacting the model's performance. The reason I did not address this imbalance upfront was due to the assumption that the

deep learning model would account for this. However, with ~88% of the observations being negative, I realize how misguided my initial approach was.

Realizing the fallacy of my original thought process, I decided to look into resampling strategies, namely over and under-sampling. Both involve randomly sampling from the data set, but approach it from different perspectives: under-sampling involves removing samples from the majority class to match the minority class while over-sampling is the opposite, adding more examples from the minority class until it matches the majority class.

After initial exploration with both techniques, oversampling tended to produce slightly better results, so it was the approach we used primarily. This process was accomplished using the following line of code: `sample.oversample_and_prep(train_df, valid_df, 0.3);`.

Notice this is another function from the `sample.py` Python script. Below is a little more detail on what `oversample_and_prep()` does on the backend. Quick note: notice this function takes in three arguments, `train_df`(the training data set), `valid_df` (the validation set), and `0.3` which represents the fraction of the oversampled data set that's returned (0.3 representing 30%).

1. Starts by gathering the number of observations labeled with negative (0) and positive (1) labels and stores them in the variables `count_class_0` and `count_class_1`, respectively.
2. Divides `train_df` into two separate DataFrames - `df_class_0` & `df_class_1` - containing just the negative observations and the other just the positive observations, respectively.
    - It also outputs the shape of each new DataFrame.
3. Using the `sample()` function, samples with replacement from `df_class_1` according to the count of negative observations `count_class_0`.
4. Concats the `df_class_0`, containing only the negative observations, with the oversampled DataFrame generated in step 3.
    - This returns a data set of nearly 400,000 observations! Remember we want to iterate quickly, which means that we then...
5. Take a 30% sample, as represented by our original argument of `0.3`, of the combined data set and then reset the index.
6. Gather counts of class 0 and 1 observations and print out their values to ensure the split is approximately 50/50.
7. Combine oversampled DataFrame - `df_test_over_sample` - with the validation data set.
8. Return the combined data set, `full_df`.

The resulting `full_df` has approximately 59,000 observations for both class 0 and class 1, for nearly 118,000 total observations. It looks like our data set is finally ready for the next step: deep learning!

The Alley-Oop - How to Prepare Data for Deep Learning

If you take a look at the `trial30` Jupyter Notebook and navigate to the section titled *Deep Learning Fast.ai Set Up*, you'll see three cells.

Wait, how did you prep the data with so little code?

Answer: more Python scripts! Similar to what we did during the data wrangling stage with `sample.py`, we're going to import another Python script - `replicate.py` - which contains functions to help us prep our images utilizing the `full_df` we just created. You can see the full command below:

```
from capstone import replicate
```

This command goes into the capstone folder in the main directory and loads in the `replicate.py` script, which has two particularly useful functions - `get_src()` and `get_data()` - which get our data in the proper format for the deep learning model.

Let's turn our attention first to `get_src()` and the following line:

```
src = replicate.get_src(full_df, path, feature_col='Cardiomegaly')
```

What exactly is going on in the backend? First, there are three arguments that we need to input; `full_df` which contains all the information on the observations we're going to feed into the model; `path` which, if you remember from earlier, contains information on where the images are stored; and `feature_col='Cardiomegaly'`, which is telling us which column to use as our target feature.

With these arguments, the function can then go to work! This is how we convert our `full_df` pandas DataFrame into a fast.ai `ImageList`

1. Use `ImageList.from_df()` to generate an ImageList class from our `full_df` with the path set to our `path`variable we created at the beginning of the project, `'/home/jupyter/springboard-capstone-2/data'`.
2. Split the data set into a training and validation set using the `split_from_df()` that takes 'valid' as an argument.
    - Remember when we created the 'valid' column earlier? Told you it would come in handy sooner or later!
    - Observations with a value of `True` in the 'valid' column are put into validation set.
3. Label the images using the `label_from_df()` function, that takes the value from `feature_col` as input.
    - To put it in more layman terms, for each observation and its corresponding image, it assigns a value of 0 or 1 to that image depending on the value present within the `Cardiomegaly` column.
4. Returns the information as the variable `src`.

We now have a proper `ImageList` however, we still have one more step which is actually to get the data (i.e., images), and that is what `get_data()` helps us accomplish. Let's take a look at the full line below:

```
data = replicate.get_data(320, src, default_trans = False)
```

The `get_data()` function takes three arguments: image size, an ImageList, and whether or not we want the default fast.ai image transformations. For the image size, we utilized `320`, which is the same size used by the team at Stanford; next, we pass in our `src` ImageList we created in the previous step to locate the images.

The last aspect - image transformations - is perhaps better known as *data augmentation*, which applies small random transformations to the images by doing such things like flipping them or making them brighter without changing the pixel values. By utilizing data augmentation, we can potentially help our model to generalize better to new images.

Now that we're familiar with the inputs, we can see what `get_data()` does with them on the backend.

1. Assigns batch size to either 32 or 16, based on available GPU memory.
    - Memory issues were pretty common during this project, which is the primary motivation for including this 'check'.
    - Batch size 'is a hyperparameter that defines the number of samples to work through before updating the internal model parameters'. (Source)
    - Essentially it is the number of images we're going to feed into the model at a time before it updates its weights.
    - Most of the time it set batch size equal to 16, which is technically called a 'mini-batch'.
    - This article offers a great introduction to how the batch size works within deep learning.
2. With `default_trans` equal to `False`, we use a minimal amount of data augmentation. The only random transformations applied are a slight increase in lighting.
    - I believed this to be the best approach because medical images to my knowledge are not altered.
    - If this had been set to `True`, there would have been more data augmentation involved.
3. Sets image size equal to 320 and padding equal to `zeros`.
    - Padding has to do with how we treat missing pixels within the images (which can result from applying transformations like a rotation).
    - By setting it equal to `zeros`, we're telling it to input any missing pixels as black.
4. Convert the original `ImageList` into a `DataBunch` which can then be passed into the PyTorch `Dataloader`.
    - Binds training, validation and test sets into a single data object.
5. Normalizes the data using `imagenet_stats`.
6. Returns the `ImageDataBunch` stored in the variable `data`.

In cell block below the `get_data()` call, we can see the output of `data`. We can see that it is an `ImageDataBunch` that contains a training set of 117,848 items and a validation set of 234 items. Now it's time to create our deep learning model!

It's Alive - Bringing the DenseNet To Life

Now let's take a look at the code block underneath the *Create DenseNet Model* in the `trial30` notebook.

```
learn = cnn_learner(data=data, base_arch=models.densenet121, metrics=[AUROC(),
Recall(), Precision(), error_rate], pretrained=True, ps=0.5, bn_final=True)
```

By using fast.ai's `cnn_learner` method, we can create the learner for our deep learning model! However, as we can see, there are several things within the learner that we need to address. Let's walk through them one by one.

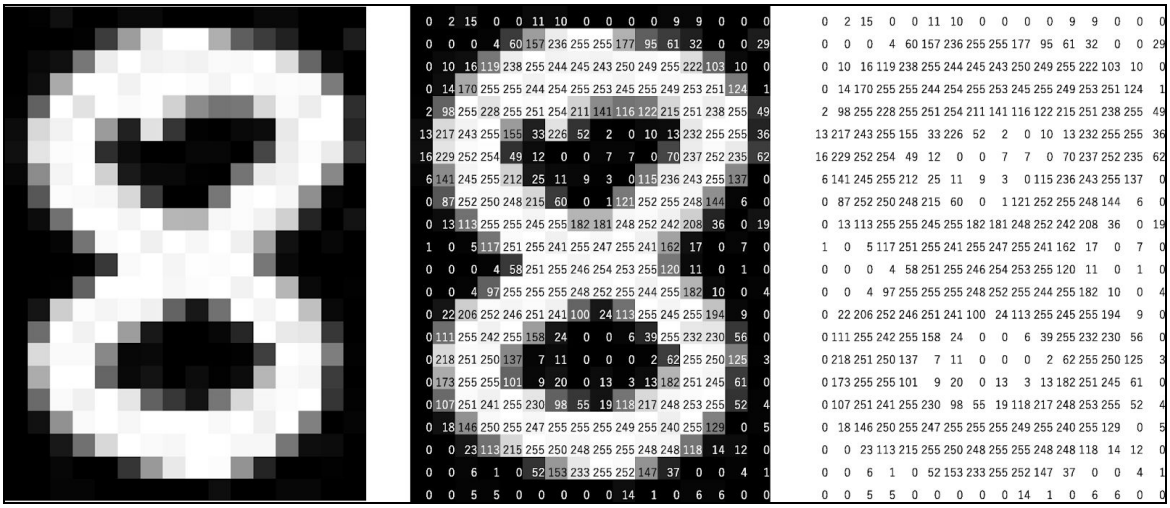
The first argument, `data` takes in a `DataBunch`. This is where we input the `ImageDataBunch` we just created, which is conveniently stored in the variable `data`.

The second argument, `base_arch`, is the architecture (i.e., backbone) of the deep learning model. For this project, we used DenseNet, or a densely connected convolutional (neural) network, with 121 layers. Before we go any further, let's get a high-level overview of what is going on with this particular architecture.

CNN's and How DenseNet is (Slightly) Different

Before we dive into what a convolutional neural network is, I want to ask the following question: what makes up the image below?

One word: pixels! Or more specifically a *matrix* (which is essentially a square-like object) of pixels. Below is a little better example of what is going on underneath the image.

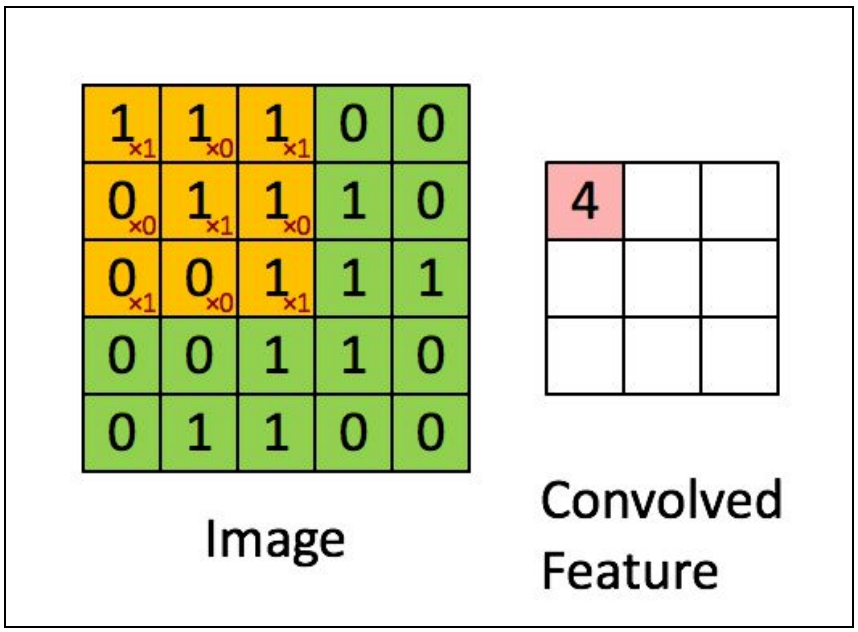


Source

As you can see, a black pixel is assigned a value of 0, and a white pixel is assigned a value of 255, with a variety of numbers in-between indicating different shades.

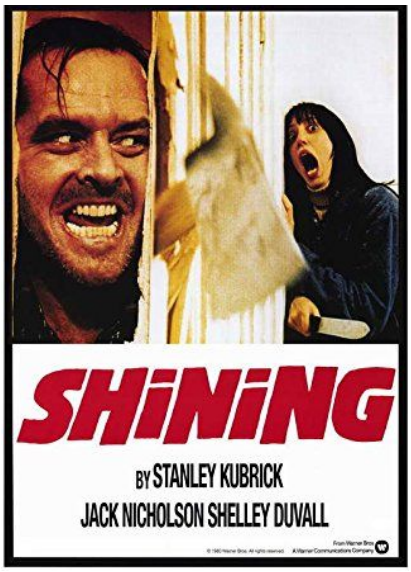
As you can see, a black pixel is assigned a value of 0, and a white pixel is assigned a value of 255, with a variety of numbers in-between indicating different shades. Now, what does this have to do with a CNN? Glad you asked! The first part of the CNN is called the *Convolutional Layer*. This layer uses a kernel to iterate over a

pixel space like the one with the number 8 above. In an attempt to be as user-friendly as possible, a visualization my help to better understand this process.



Source

The green box is the image with each of its pixels, the yellow box is our kernel, and the pink box is our convolved feature (which I'll get to in a second). Now, the kernel can be viewed kind of like a flashlight, where it steadily works its way across the entire width and height of the image until it has shined its light on every part of it. Guess you could call it a deep learning version of the shining[15].



What this kernel (or 'shining') is doing each step of the way is performing matrix multiplication between its weights/parameters (i.e., the red numbers in the bottom right corner) and the pixel values in the image.

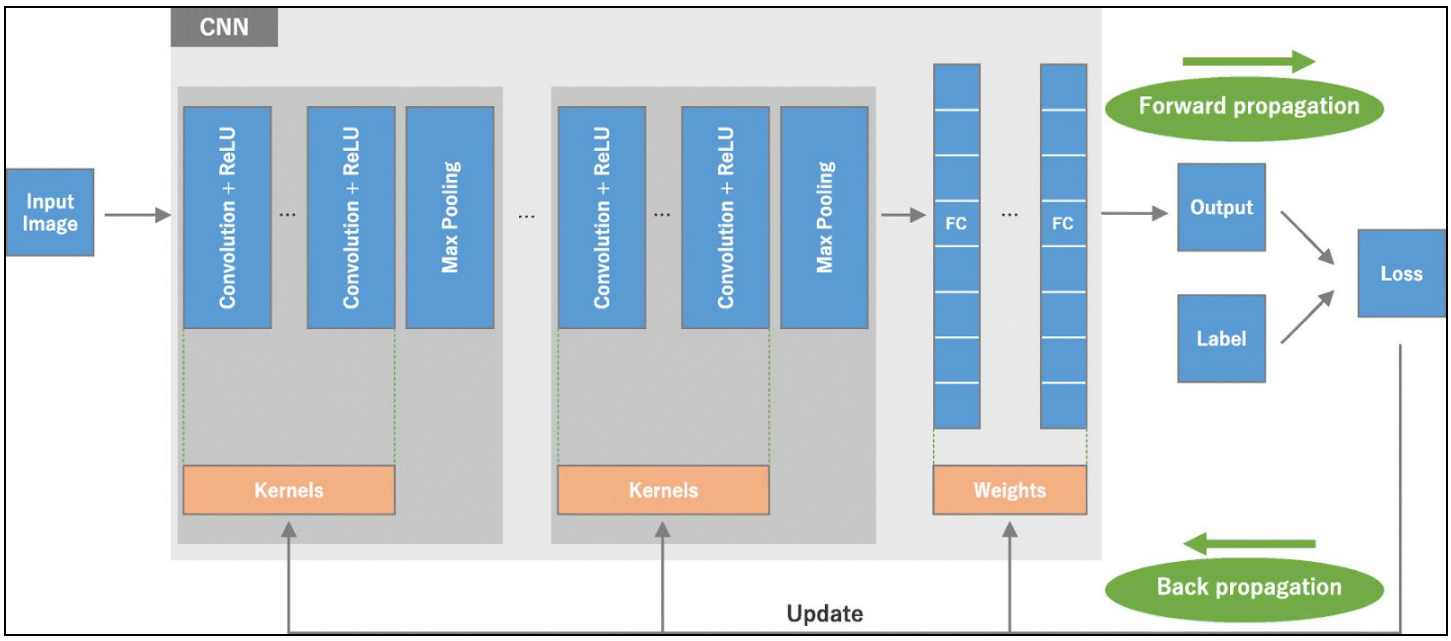
If you are unfamiliar with matrix multiplication, I highly suggest checking out this website. It provides a great visualization of how it works.

At this point, we can turn our attention to the pink box. These matrix multiplications are then all summed up and output to the convolved feature. For example, the kernel starts in the upper left corner, performs matrix multiplication after which it sums these numbers together and outputs that number (i.e., 4) to the convolved feature (i.e., pink box).

The convolutional layer's main goal is to filter, as it checks for patterns in each section of the image. This brings us to the next layer, which is called the pooling layer. It's similar to the convolution in that it too utilizes a kernel. However, we're using this kernel to reduce the spatial size of our convolved feature further. There are two primary approaches for pooling - max and average - which returns either the max value or the average of all the values from a particular section covered by the pooling kernel. The last main layer is the activation layer, which takes these linear outputs and squashes them into a range using a nonlinear function like ReLu (rectified linear unit). A ReLu sounds a lot scarier than it is. Mathematically, it can be defined as $y = max(0,x)$, meaning

[15] Source for *Shining* poster

that if the number is negative, it returns 0; otherwise, it takes whatever value x is. Below is a visual representation of what is going on with convolutions, pooling, and activation layers.



Source

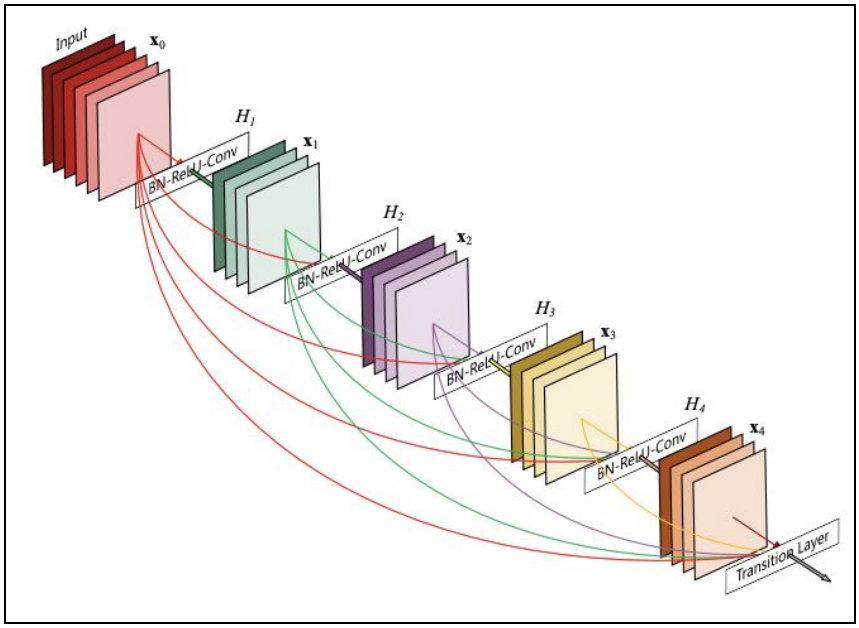
Now take note: there are multiple convolutions, ReLUs, and pooling layers above. Most convolutional neural networks have multiple layers, with 100+ layers not being uncommon for more complicated input images (like medical images). Additionally, you may be wondering what is going on at the end of the picture with the 'FC' columns and backpropagation. The FC's are the fully connected layers, which take input from the previous layer and outputs a vector with N dimensions, with N representing the number of classes the model has to choose from. For example, if you were trying to determine whether an image was of a cat or a dog, N would be equal to 2 (i.e., cat represents one potential class, and dog represents another potential class). What is output to this N-dimensional vector is not a 'yes' or a 'no'; instead, it outputs the probability of being in that particular class. To continue with our cat/dog example, it may output 0.50 for the dog class and 0.50 for the cat class meaning that the image has a 50% probability of being a dog or a cat (this is known as the softmax function, which you can find more information about here).

This brings us to *back propagation*, which is essential to how neural networks 'learn.' Our process began when we input the image, and it worked its way through the multiple layers. This beginning stage is called the forward pass and results in us getting the N-dimensional vector of probabilities as previously mentioned. Let's go back to our dog/cat example and say our input image was, in fact, a dog. However, since the model wasn't very confident that it was a dog, it's going to go back and do what is called a backward pass through the network and update the weights (i.e., the red numbers in the kernel) to minimize its loss function. Essentially, this process allows the model to become more 'confident' that that image (and those similar to it) are of dogs. If everything goes right, with each additional round of training, the model becomes more confident that our original input image is a dog by minimizing its loss function. This higher confidence is reflected in a higher probability for that class.

So we have a general understanding of what a CNN is, so now you may be wondering...

What does DenseNet have to do with any of this?

Well, DenseNet takes our convolutional neural network and adds what is called a *dense block*. These blocks are interconnected with each other, which gives higher layers the ability to reuse features from lower layers. In essence, the network is better able to communicate what is essential and then leverage it at any particular layer since each dense block is connected.



Source

Image - Each color is a block of layers; notice the red and green lines that show how each block is interconnected.

While this interconnectedness between layers may at first seem to increase complexity, it was found to *reduce* the number of parameters and increase performance (compared to ResNet). (Source)

So a model architecture that's less complex and offers better performance? Sign me up!

(PS - It was also the same architecture the team at Stanford found to have the best performance during their CheXpert research...)

Last Remarks on CNN/DenseNet

There are two features - dropout and batch normalization - that I want to touch upon really quickly before we look into the performance metrics, model training process, and the results. Both are relatively new techniques that help prevent a deep learning model from overfitting the data. Dropout accomplishes this by randomly removing a percentage of the activations to prevent the model from relying to heavily on any one feature from an image.

Batch normalization 'normalizes' the output data from the previous activation layer. What exactly does this mean? Essentially it "allows each layer of the network to learn by itself a little bit more independently of other layers." (Source) Since no any one feature can have an overwhelming impact, resulting in its weights cascading down through the network, the model is more stable during training. Additionally, it gives us the

added benefit of being able to use higher learning rates, which allows us to train our model faster as well. (Source) Below are the commands within `cnn_learner()` for dropout and batch normalization:

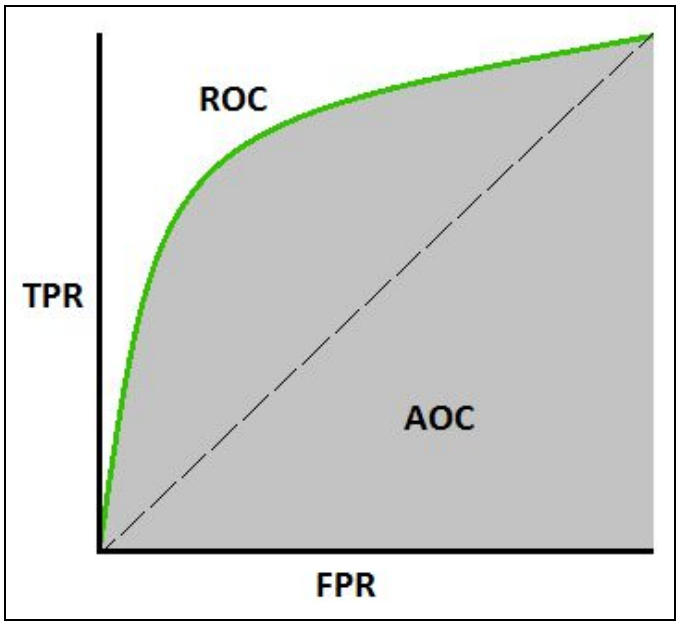
- `ps=0.5`: dropout, 0.5 is the probability that an activation is dropped
- `bn_final=True`: batch normalization

The Metrics: AUROC, Recall, and Precision

When it comes to classification problems, there are quite a few metrics one could use. The most obvious is accuracy, which takes the number of correct predictions and the total number of predictions (i.e., the number of data points).

For a model that classifier that labels images as either a cat or a dog, accuracy may be a suitable metric. However, from the CheXpert data, we are trying to build something that has the potential to assist medical professionals detects diseases and quite literally save lives. To say the stakes are a little higher may be an understatement.

Due to this fact, we are going to use AUROC as the primary metric. This metric is an extension of the ROC curve (i.e., the green line below), which is plotted using the true positive rate on the y-axis and false positive rate on the x-axis (see below). In our case, a true positive is represented by positive prediction on an observation that did indeed have cardiomegaly; a false positive would be a positive prediction on an observation that did not have cardiomegaly.



Source

Now AUROC takes the ROC curve a step further and assesses the area underneath that curve, which it is otherwise known as Area Under the Receiver Operating Characteristics.

A perfect classifier would have an area of 1 (i.e., the graph would go straight up and over, resembling a flipped L). What does the dashed line that cuts through the middle of the graph represent? This line represents an AUROC of 0.5, which would theoretically represent a classifier that's no better than random at detecting cardiomegaly.

Now what are recall and precision and what do they have to do with this analysis?

Recall represents the number of true positives divided by the sum of true positives and false negatives. What recall can tell us is how good the model is at finding the data points of interest (i.e., observations that have cardiomegaly). Precision, on the other hand, represents the value of true positives divided by the sum of true positives and false positives which tells us the proportion of observations that our model said was relevant that actually were.

Recall & Precision In Context

- *RECALL* = Cases of cardiomegaly correctly identified divided by (cardiomegaly cases correctly identified +cardiomegaly cases incorrectly labeled as no cardiomegaly)
- *PRECISION* = Cases of cardiomegaly correctly identified divided by (cardiomegaly cases correctly identified + no cardiomegaly cases incorrectly labeled as having cardiomegaly)

Ideally, we want a high recall and precision as this indicates that the model can identify not just only relevant instances but all of them as well.

The Strategy: Training the Model

In the `trial30` Jupyter notebook, you can see there are five 'rounds' where we trained the model. Let's begin by getting a general overview of what is going on each round.

```
learn.fit_one_cycle(5, 9e-3,
                    callbacks=[SaveModelCallback(learn, every='improvement', monitor='auroc', name
='best-dn121-trial30-rd1')])
```

epoch	train_loss	valid_loss	auroc	recall	precision	error_rate	time
0	0.506237	0.421783	0.864103	0.647059	0.676923	0.192308	43:22
1	0.521744	0.473742	0.815291	0.367647	0.657895	0.239316	42:31
2	0.499431	0.435838	0.842576	0.529412	0.782609	0.179487	42:26
3	0.471259	0.460393	0.813076	0.514706	0.729167	0.196581	42:26
4	0.469090	0.447554	0.826541	0.397059	0.729730	0.217949	42:27

We utilized fast.ai's `fit_one_cycle()` method on our `learn` object, which contains our deep learning model. This particular method utilizes what is called the 1cycle policy, which can train complex models in significantly less time. It involves hyper-parameters such as learning rate, momentum, and weight decay, which are all subjects I do not feel quite qualified to discuss in detail. However, Sylvain Gugger, who is also a part of the fast.ai team, wrote a great blog post on the 1cycle policy that you should check out if you're interested in learning more.

The top cell includes the Python code we use to initiate training while the bottom cell corresponds to the output, where the `auroc`, `recall` and `precision` scores can be easily assessed. Also, `error_rate` is included to give an additional high-level perspective on how good our model is doing.

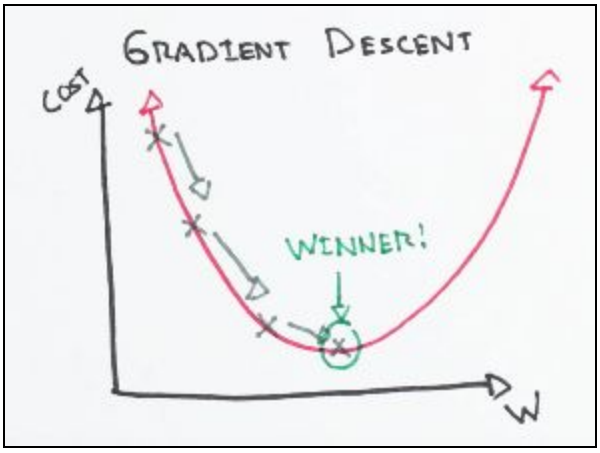
Within the coding cell, we can see that `fit_one_cycle()` includes the following: `5`, representing the number of epochs; `9e-3` which is the learning rate; and the `SaveModelCallback`, which allows us to save our best performing model (as judged by `auroc`).

Before we go any further, let's answer the following questions: what is an epoch, and what is the learning rate?

An epoch represents the case when an entire dataset is passed forward and backward through a neural network one time. Now, why did we train for five epochs? Good question. Unfortunately, to my knowledge, there is no right answer when it comes to the best number of epochs to use. Generally, things like training time and diversity of the data should be taken into consideration when considering epochs, but there is also quite a bit of experimentation required. I found 3-5 epochs to be the best range for this particular project, as it didn't require too much time, and the results tended to be good.

Now, onto learning rate; what is it?

Learning rate affects how much the weights of the network are adjusted with respect to the loss. This is otherwise known as gradient descent.



Source

Take a look at the picture above. Essentially the goal of gradient descent, given some higher dimensional space, is to find the minimum point (i.e., the bottom of the curve). Once this minimum (i.e., the 'winner') is reached, the corresponding model and its weights are optimal and return the minimum loss possible for that given data set. Now the learning rate comes into play in how fast we descend to this minimum (indicated by the arrows in above image). Notice how the dots are progressively getting closer to the minimum/'winner' point. A lower learning rate would take much longer to reach this 'winner' point if it even reaches it at all. On the flip side, if the learning rate is set too high, it runs the possibility of overshooting the minimum or even diverging out of the curve entirely.

With all the bases covered, let's take a look at the key characteristics from each round.

- *Rd. 1*: Original data set, 5 epochs, utilized `SaveModelCallback` to save best performing model based on AUROC
- *Rd. 2*: Loaded best performing weights from rd. 1 (`trial30-rd1`), unfroze entire model allowing every layer group to be trainable (`unfreeze()`), set learning rate to `~6.31e-07` then passed `slice()` to allow for differential learning rate, trained for 5 epochs
- *Rd. 3*: Set new seed, generated a new randomly sampled data set of images (~40,000), loaded best performing weights from rd. 2 (`best-dn121-trial30-rd2`), unfroze entire model to allow every layer to

be trainable, set learning rate to `~5.01e-07`) then passed to `slice()` to allow for differential learning rates, trained for 5 epochs
- *Rd. 4*: Set new seed, generated a new randomly sampled data set of images (~40,000), loaded best performing weights from rd. 3 (`best-dn121-trial30-rd3`), froze model up to last layer group `freeze()`, set learning rate equal to `3e-06` then passed to `slice()` to allow for differential learning rates, trained for 3 epochs
- *Rd. 5*: Loaded best performing weights from rd. 4(`best-dn121-trial30-rd4`), unfroze model, set learning rate to `~2.51e-07` then passed `slice(lr, 3e-4)` to allow for differential learning rates, trained for 3 epochs

Results & Future Improvements

```
learn.fit_one_cycle(3, slice(lr, 3e-4),
                    callbacks=[SaveModelCallback(learn, every='improvement', monitor='auroc', name
='best-dn121-trial30-rd5')])
```

epoch	train_loss	valid_loss	auroc	recall	precision	error_rate	time
0	0.515831	0.468307	0.819454	0.485294	0.600000	0.243590	19:29
1	0.493614	0.457376	0.827693	0.470588	0.695652	0.213675	19:30
2	0.482237	0.451895	0.832920	0.441176	0.681818	0.222222	19:31

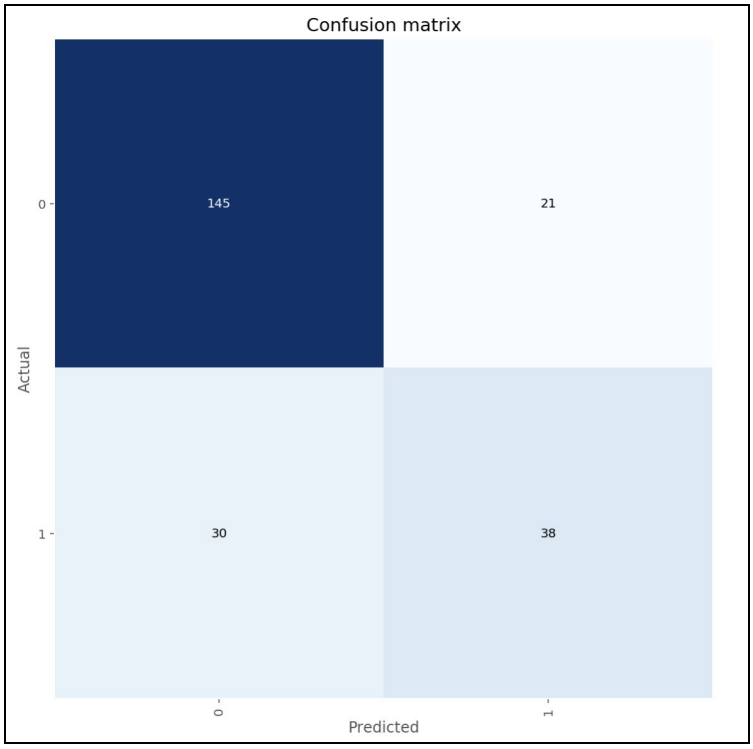
```
Better model found at epoch 0 with auroc value: 0.8194541931152344.
Better model found at epoch 1 with auroc value: 0.8276931047439575.
Better model found at epoch 2 with auroc value: 0.8329199552536011.
```

(*Output from last round of training*)

The image above is the output from our fifth and final round of training. When can see that the `auroc` steadily increases from ~0.81 to ~0.83, indicating a gradual increase in overall performance. As a reminder that when utilizing the `U-Zeros` approach, the team at Stanford achieved an AUROC of 0.84, meaning that, as is, this model is nearly at the same level!

However, there is still much room for improvement. Firstly, the `recall` score is pretty low, indicating a model that is perhaps a little too picky. The `precision` score is relatively high (although it too could be higher) meaning that for the most part, the images it thinks contain cardiomegaly, do have cardiomegaly. Yet, it misses a lot of other actual cases of cardiomegaly because it's so picky.

For a visual representation, let's take a look at a confusion matrix (see next page) from the fourth round of training. (Side note: the reason for taking confusion matrix from the fourth round as opposed to the fifth is because the model achieved a `recall` score in the mid-0.50's, slightly higher than that in the fifth.) If you take a look at the bottom left corner, we find 30 false-negative cases. This is nearly 13% of the validation data set and is cause for serious concern as these 30 observations represent cases where cardiomegaly was present, but the model predicted that there was no cardiomegaly. In the context of the situation, a false negative is very bad; it means that a patient who is sick doesn't get the treatment they need. So for future iterations of this model, we would need to figure out how to reduce the number of false negatives to ensure more cases of cardiomegaly are captured.



(Confusion Matrix from fourth round of training)

Closing Thoughts

Overall, this has been a very challenging but overall highly rewarding project. While the model we produced here is not ready to be used in any medical context, it highlights the enormous potential for artificial intelligence/deep learning within the healthcare space, especially with continued research.

This is just the first step of my deep learning journey, though, and I'll be looking to continue to grow my body of knowledge related to the subject. In particular, I hope to continue to combine the technical with the medical and hopefully contribute to healthcare in such a way that helps people live healthier and happier lives.

Thank you for taking the time to read this analysis and hope you got some valuable insights along the way. However, with that being said, it's time to get back to work!