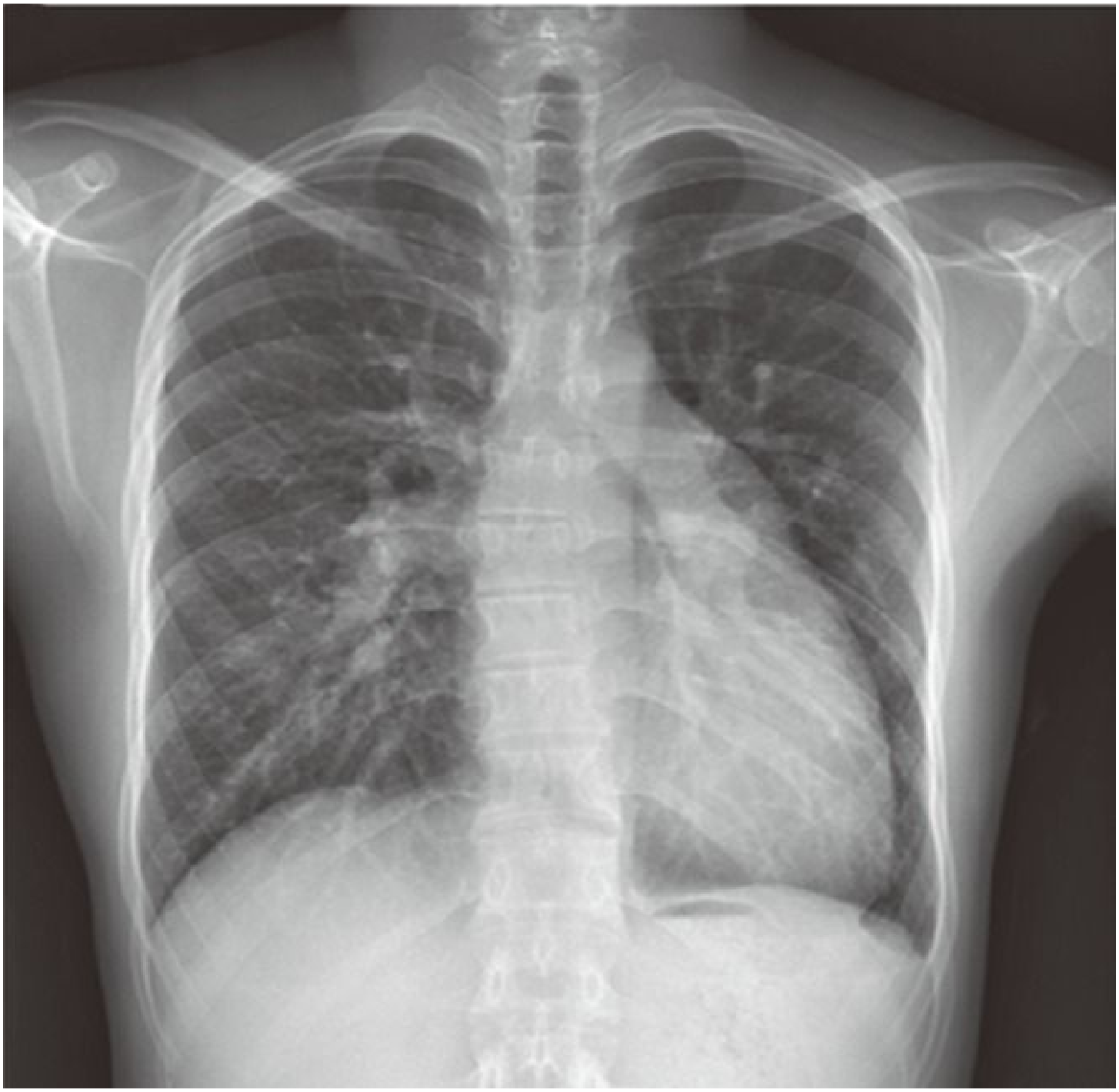


Classification of cardiomegaly using CNN

01.22.2018

Anoop Singh

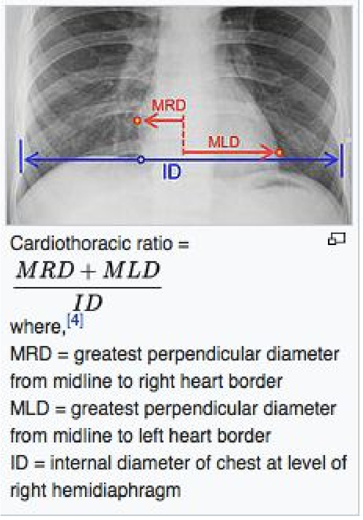
I. Definition

Project Overview (copied verbatim from proposal)

From Wikipedia:

Cardiomegaly is a medical condition in which the heart is enlarged. It is more commonly referred to as an enlarged heart. The causes of cardiomegaly may vary. Many times this condition results from high blood pressure (hypertension) or coronary artery disease. An enlarged heart may not pump blood effectively, resulting in congestive heart failure.

X-ray images help see the condition of the lungs and heart. If the heart is enlarged on an X-ray, other tests will usually be needed to find the cause. A useful measurement on X-ray is the cardio-thoracic ratio, which is the transverse diameter of the heart, compared with that of the thoracic cage." These diameters are taken from PA chest x-rays using the widest point of the chest and measuring as far as the lung pleura, not the lateral skin margins. If the cardiac thoracic ratio is greater than 50%, pathology is suspected, assuming the x-ray has been taken correctly. The measurement was first proposed in 1919 to screen military recruits. A newer approach to using these x-rays for evaluating heart health, takes the ratio of heart area to chest area and has been called the two-dimensional cardiothoracic ratio.



Cardiothoracic ratio =

$$\frac{MRD + MLD}{ID}$$

where,[4]
MRD = greatest perpendicular diameter from midline to right heart border
MLD = greatest perpendicular diameter from midline to left heart border
ID = internal diameter of chest at level of right hemidiaphragm

X-ray exams are the first step in diagnosing cardiomegaly in a patient. Once the x-ray's available, a radiologist looks at it and tries to diagnose the disease.

From GlobalDiagnostiX - Context: *According to WHO figures, **more than two thirds of the world's population does not have access** to this essential x-ray imaging equipment. Too often in developing countries, patients die of trivial problems, which, due to a lack of access to diagnosis, take dramatic proportions.*

From Most of the World Doesn't Have Access to X-Rays: *After the 2010 earthquake in Haiti, Mendel and Partners in Health stocked the University Hospital in Mirebalais (UHM) with a CT scanner—the first in a public hospital in Haiti and the first to cost their patients nothing. But the hospital still doesn't have enough money to hire a radiologist to run the machine, Mendel says.*

One solution is telemedicine. UHM uses a picture-archiving and communication system that sends CT scans to a server in Boston, which stores the images and creates an electronic medical record for volunteer radiologists in the U.S. and Canada to read. The volunteers log on twice a week to look over scans, which each take about ten minutes. In 2014, 40 volunteers read approximately 4,000 CT scans.

But telemedicine has limits, especially in an emergency. "Bus accidents happen every day in Nepal," Schwarz explains. "You have literally 25 patients all at once, who are all bleeding. That's challenging enough. You're certainly not waiting for someone in a different country or time zone to tell you what an x-ray shows."

With initiatives like GlobalDiagnostiX, there's hope that low cost x-ray systems will be more readily available in underdeveloped countries as time progresses. Radiologist availability still remains limited and lives are lost while patients wait for diagnosis.

The ImageNet challenge has led to successful advances in the field of computer vision and the ability to use convolutional neural networks for image recognition tasks. Using transfer learning, there have been several instances where a successful ImageNet architecture's used and modified/re-trained to recognize images in a particular field with high accuracy.

Due to the availability of large datasets, progress in computer hardware and an active interest in using machine learning for medical diagnosis, researchers have demonstrated at-par or better performance when compared to medical professionals.

Radiologists will be 'obsolete' in five years led me to gain interest in this particular problem. If we can solve this problem, it can drastically bring down diagnosis time (particularly in developing countries) and potentially make diagnosis cheaper.

When compared to medical professionals, we may get better diagnosis as well due to:

- the ability to train on tens and hundreds of thousands of images, to pick up intricate details
- no overworked radiologists who have to occasionally screen over 100 x-rays a day, potentially leading to errors

Research Citations (copied verbatim from proposal)

- ChestX-ray8: Hospital-scale Chest X-ray Database and Benchmarks on Weakly-Supervised Classification and Localization of Common Thorax Diseases
- CheXNet: Radiologist-Level Pneumonia Detection on Chest X-Rays with Deep Learning

Problem Statement (copied verbatim from proposal)

Develop an algorithm that can detect cardiomegaly from chest X-rays.

This is a binary classification problem: does the image exhibit Cardiomegaly or not?

Inputs for the problem: Images labeled as *Cardiomegaly* or *No Finding*, we'll ignore all the other features.

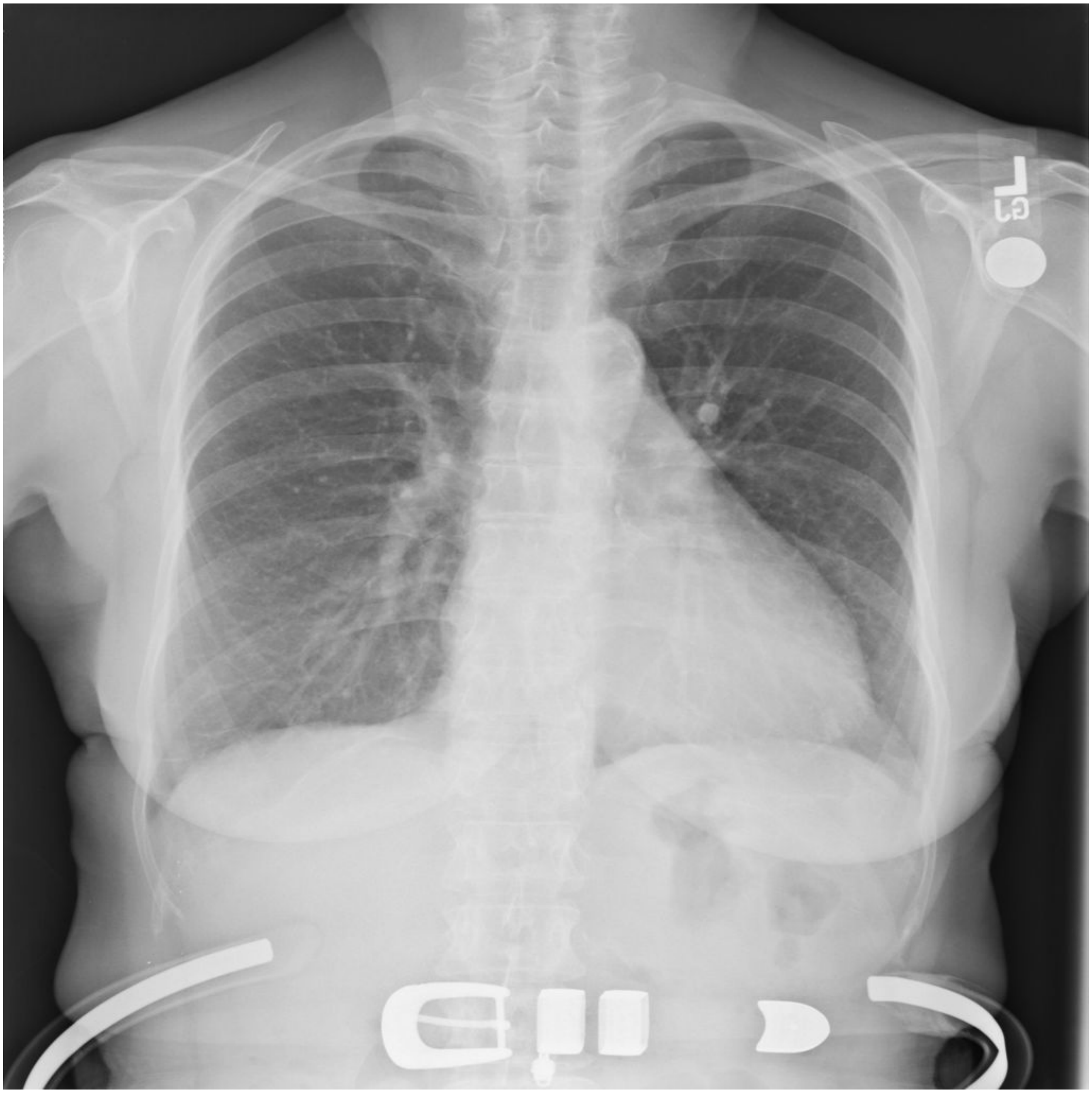
Outputs for the problem: label for the image (*Cardiomegaly* or *No Finding*)

Datasets and Inputs (copied verbatim from proposal)

CXR8 dataset provided by the National institute of Health Clinical Center

It has labels that diagnose a chest x-ray into the following 8 thoracic pathologies: *Atelectasis, Cardiomegaly, Effusion, Infiltration, Mass, Nodule, Pneumonia* and *Pneumathorax*. An image may be labeled as one or more of these pathologies. A *No Finding* label indicates that the image was not labeled as any of the 8 pathologies.

Sample image (PNG, 1024 x 1024, 414 KB):



Data:

Image Index,Finding Labels,Follow-up #,Patient ID,Patient Age,Patient Gender,View Position,OriginalImage[Width,Height],OriginalImagePixelSpacing[x,y],

00000001_000.png,Cardiomegaly,0,1,58,M,PA,2682,2749,0.143,0.143

00000001_001.png,Cardiomegaly|Emphysema,1,1,58,M,PA,2894,2729,0.143,0.143

00000002_000.png,No Finding,0,2,81,M,PA,2500,2048,0.171,0.171

..

..

00000008_000.png,Cardiomegaly,0,8,69,F,PA,2048,2500,0.171,0.171 (for sample image)

Intended solution

1. As part of data pre-processing, we separate the images into two folders: *cardiomegaly* and *no finding* using the metadata in the dataset
2. This is further separated into *train*, *valid* and test. (Eg. *train/cardiomegaly*, *valid/cardiomegaly* and *test/cardiomegaly* ...)
3. Images are reduced to 224x224 from 1024x1024 to bring the dataset size down for faster and more economical processing
4. Using Keras and TensorFlow, we create our own CNN model and assess its performance
5. Using transfer learning, we train on ImageNet trained models like Inception, ResNet50, VGG19 and Xception, assessing their performance

We hope to settle on a model, that can do well on the task of binary classifying a chest X-ray image into *Cardiomegaly* or *No Finding*.

Metrics

Our dataset has 2,776 *Cardiomegaly* images and 60,361 *No Finding* images. This means, that a naive algorithm classifying everything as *No Finding* would give us an accuracy of 95.6%.

Since our data set will have a strong bias towards No Finding vs Cardiomegaly, F-1 score is a better metric than accuracy.

We'll compare our own model's results with observations from the benchmark (ImageNet trained models). We'll be looking at:

- F-1 score (we specify *average as micro* to take the label imbalance into account)
- Precision score
- Recall score

II. Analysis

Data Exploration (copied verbatim from proposal)

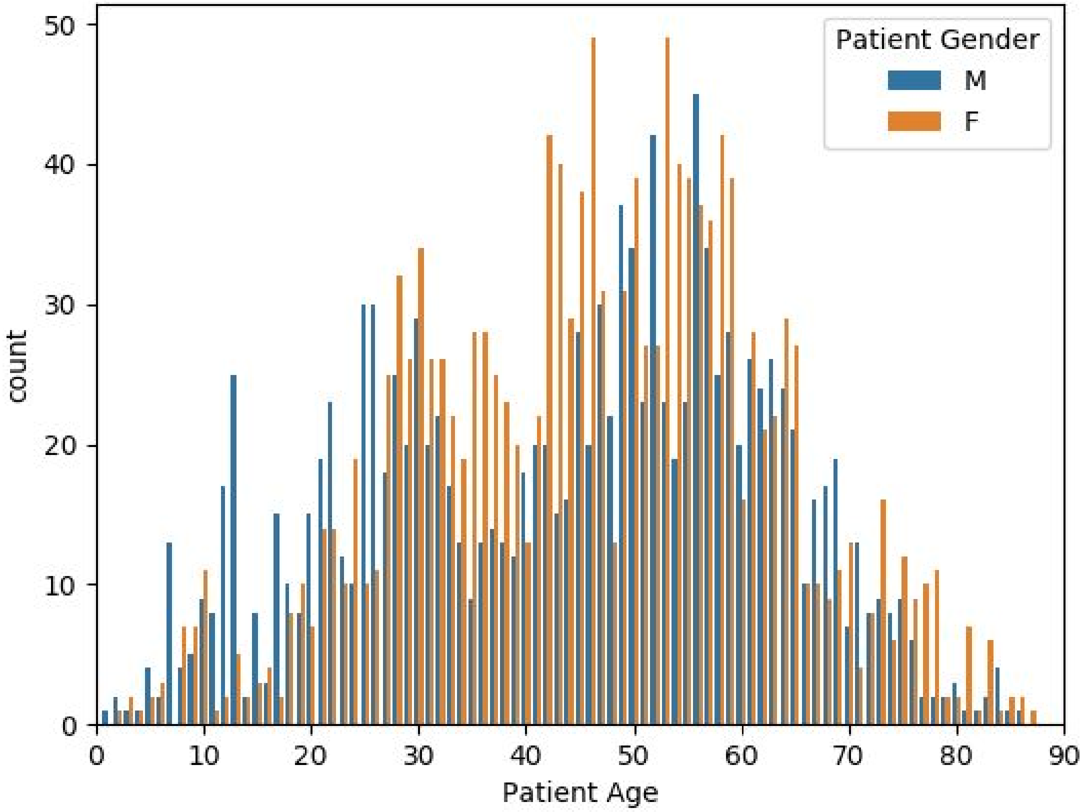
(See ***Datasets and Inputs*** section too)

- How are the images structured?
    - PNG, 1024 x 1024, 400 - 500 KB
- Are there color layers?
    - The images are in grayscale and should not need any color transformation before they can be used
- What are the dimensions (or ranges of dimensions)?
    - 1024 x 1024
- How many examples are there in the dataset that you'll be using?
    - The dataset contains over 100,000 anonymized chest x-ray from more than 30,000 patients and takes 45.14 GB of space when compressed
    - We'll be using every image
- How many classes will there be (just two? or more?) and are they balanced?
    - Just two - *Cardiomegaly* and *No Finding*
    - are they balanced? - No, the dataset will be biased towards more images with the *No Finding* label
- How will you split the data into training/validation/testing sets?
    - May start off with a 70 - 10 - 20 split and see how it goes
- Will you do anything to maintain class balances across each subset?
    - Yes, will aim for a consistent ratio of images labeled *Cardiomegaly* to images labeled *No Finding* within each subset

Exploratory Visualization

Let's look at the breakdown of cardiomegaly patients by gender and age:

(To reproduce - *python visualize.py*)



Although we ignore the age and gender labels for our solution, this visualization is still worth discussing. As we notice, under some age buckets, male and female patients exhibit different patterns.

For example, in the 10 to 30 age group, there seem to be more male patients whereas in 30 to 40 age group, the opposite is true with more female patients. However, we must not read too much into this as it maybe a limitation of the dataset at hand.

There's a possibility that including patient gender and age in the training process leads to better results. This is left as a future improvement and excluded from the current solution.

Algorithms and Techniques

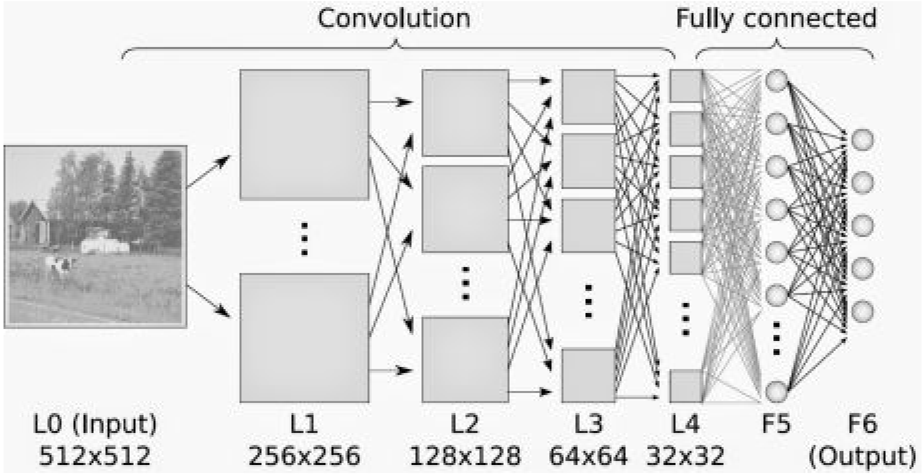
In this section, we'll go over the architecture for our own model and talk briefly about transfer learning using ImageNet trained models.

To solve the problem at hand, we employ a technique called Convolutional Neural Networks (CNN).

(reference link)

In a standard neural network, where neurons are connected with each other, signals are passed in a single direction only. This is known as forward-feed. Although successful, this required all the neurons to be connected to each other driving up complexity, especially for large datasets.

CNN was developed as a better alternative and involves four main steps: convolution, subsampling, activation and full connectedness.



Convolution

The first layers that receive an input signal are called convolution filters. Convolution is a process where the network tries to label the input signal by referring to what it has learned in the past. The resulting output signal is then passed on to the next layer.

Each convolution filter represents a feature of interest (e.g whiskers, fur), and the CNN algorithm learns which features comprise the resulting reference (i.e. cat). The output signal strength is not dependent on where the features are located, but simply whether the features are present.

Subsampling

Inputs from the convolution layer can be “smoothened” to reduce the sensitivity of the filters to noise and variations. This smoothing process is called subsampling, and can be achieved by taking averages or taking the maximum over a sample of the signal.

Activation

The activation layer controls how the signal flows from one layer to the next, emulating how neurons are fired in our brain. Output signals which are strongly associated with past references would activate more neurons, enabling signals to be propagated more efficiently for identification. CNN is compatible with a wide variety of complex activation functions to model signal propagation, the most common function being the Rectified Linear Unit (ReLU), which is favored for its faster training speed.

Fully Connected

The last layers in the network are fully connected, meaning that neurons of preceding layers are connected to every neuron in subsequent layers. This mimics high level reasoning where all possible pathways from the input to output are considered.

Learning algorithms require feedback. This is done using a validation set where the CNN would make predictions and compare them with the true labels or ground truth. The predictions which errors are made are then fed backwards to the CNN to refine the weights learned, in a so called backwards pass. Formally, this algorithm is called backpropagation of errors, and it requires functions in the CNN to be differentiable (almost).

Own model

```
model = Sequential()

model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu',
            input_shape=(IMAGE_DIMENSION, IMAGE_DIMENSION, 3)))

model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Conv2D(filters=128, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Conv2D(filters=256, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Conv2D(filters=512, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Conv2D(filters=1024, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(GlobalAveragePooling2D())
model.add(Dense(2, activation='softmax'))
```

Justification for choices:

- We start with 16 filters and keep on doubling it until 1024 - this allows us to pick up finer features as we go along. Eg. imagine picking up an edge in the initial layers and picking up facial characteristics from an human face in the later ones.
- MaxPooling2D after each layer ensures that we down-sample the input representation, reducing its dimensionality, which will also help lower down the number of parameters and bring down computation cost (source)
- Dropout after each layer to prevent overfitting
- GlobalAveragePooling2D at the end for similar reasons (MaxPooling2D)
- Densely connected 2 layer at the end with softmax due to the two categories(*Cardiomegaly*, *No Finding*), and to get probability of prediction

Transfer Learning on ImageNet trained models

ImageNet trained models have been trained on a dataset (objects, animals, scenery, etc.) which's drastically different from the our dataset (chest X-rays). This means that we'll have to make some efforts to transfer learn sensibly.

We want to benefit from the pre-trained model's ability to know about elementary features (edges, shapes, etc). However, instead of the model's ability to know about dogs and cats, we'd like it to learn about chest X-rays or maybe organ outlines.

Here's how we go about achieving that:

1. Add the following layers at the end:
    a. GlobalAveragePooling2D (for same reasons as in *own model*)
    b. Densely connected 1024 layer
    c. Densely connected 2 layer with softmax (for same reasons as in *own model*)
2. We freeze all the base layers of the pre-trained model
3. Train the model for 1 epoch. This trains the layers added by us to have apt weights
4. Freeze the first three fourth of the model's layers and train the rest for 3 epochs
    a. The hope is to use pre-trained model's ability to pick up basic features (edges, shapes, etc) while training to learn higher level features

Benchmark

(See ***Metrics*** section too)

For evaluation, we'll test on the *test* data, comparing our own model against the ImageNet trained models.

III. Methodology

Data Preprocessing

1. find . -maxdepth 1 -iname "*.png" | xargs -L1 -I{} convert -resize 21.875% "{}" "{}"
    a. This converts all images from 1024x1024 to 224x224 lowering down the dataset size from 45.29 GB to 1.37 GB
    b. The code implementation uses 197x197 to further lower down training time. 197x197 was selected as that's the lowest resolution that ResNet50 accepts.
2. (look at prune_dataset.py)
    a. This script reads the Data_Entry_2017.csv file and moves images into either of the two newly created folders (cardiomegaly and no finding)
3. (look at split_dataset.py)
    a. This script moves the images from above into *train*, *valid* and *test* folders with a 70/10/20 split

Implementation

(See ***Algorithms and Techniques*** section too)

(look at train_multiple.py - some code and ideas borrowed from Keras documentation)

To repeat, the actual implementation is:

1. Load images after data pre-processing
2. Train and evaluate own model
3. Train and evaluate ImageNet trained models

Challenges faced during implementation

- Dataset size + hardware limitation
    - Initially, perhaps somewhat naively, I believed that by reducing the image resolution and/or working with a subset of data, I would be able to train the model on my laptop (MacBook Air - 8 GB RAM, no Nvidia GPU, etc).
    - I quickly realized that this would not be possible. Just loading the images to memory failed with a Python *MemoryError*. Keras's *ImageDataGenerator flow_from_directory()* was useful in circumventing this but it was still terribly slow.
    - In the end, I developed/tested the code locally using 6 images (3 per category, 1 each for train, valid and test), followed by running it in AWS for the entire dataset
- Choosing a cloud provider
    - Google Cloud Platform vs Amazon Web Services
    - GCP although free (first time user), made it terribly inconvenient to set things up. I'd have to pick up an instance, install python, needed libraries, TensorFlow with GPU support, Nvidia drivers and CUDA support before doing anything productive
    - AWS in contrast, provides:
        - Deep Learning AMI, which comes pre-installed with ML libraries, CUDA support and Nvidia drivers
        - p2.xlarge EC2 instance which comes with a Nvidia Tesla K80 GPU, which made the task at hand considerably easy

As a result, I settled on using AWS.

Setup on AWS EC2 machine:

1. SSH to machine
    a. *ssh -i aws.pem ec2-user@a.b.c.d*
2. Upload the code and dataset
    a. *scp -i /Users/asingh/Downloads/aws.pem Archive.zip ec2-user@a.b.c.d:*
3. Update packages

  a. *sudo yum update*

4. Activate Conda environment for TensorFlow
    a. *source activate tensorflow_p36*
5. Manage dependencies
    a. *conda uninstall graphviz jasper* (Needed to facilitate pillow install)
    b. *conda install pillow scikit-learn tqdm*
6. Update pip packages
    a. *pip freeze --local | grep -v '^\-e' | cut -d = -f 1 | xargs -n1 pip install -U*
7. Start a screen session
8. Start implementation script
    a. *python train_multiple.py > output.txt*
9. For monitoring:
    a. Memory use *watch -n0 free -m*
    b. GPU use *nvidia-smi -l 1*

output.txt and the weights from the trained models were downloaded from the EC2 machine.

Refinement

- Picking the image size:
    - Dataset's images - sized at 1024x1024 and around 400 KB per image, resulted in a huge size
    - Several resolutions from 128x128 to 224x224 were tested to find the right balance between training time/memory usage and presenting our CNN with enough information to work with
    - Finally settled on 197x197 since that's the lowest resolution that ResNet50 accepts
- Architecture for own model:
    - Replicated the same architecture as the one used in 'classifying dog breeds' Udacity assignment with minor changes
    - Added another *Conv2D* layer at the end with 1024 filters, in the hope that higher level features are better picked up
    - Hyperparameters were not tuned since the performance was satisfactory
- Transfer Learning - deciding how many layers to freeze vs how many to train:
    - Based solely on intuition, I decided to freeze the first three fourth of layers and unfreeze the rest, making them trainable
    - Since it was reasonably fast to train with these settings on the Nvidia Tesla K80 GPU and the performance was satisfactory, I didn't experiment with other ratios

- Picking number of epochs for training:
    - Settled on 3 epochs as I noticed that metrics started converging after 2 epochs and there was minimal improvement beyond that. As an additional benefit, training finished sooner.
- Hyperparameter tuning:
    - Relied on the defaults from the Keras documentation

IV. Results

Model Evaluation and Validation

(look at output.txt)

Model	Training time (s)	loss	categorical_accuracy	val_loss	val_categorical_accuracy
Own model	521	0.7062	0.9562	0.3127	0.9564
InceptionV3	1657.1888	0.1518	0.9562	0.2409	0.9564
ResNet50	2358.5821	0.7062	0.9562	0.7030	0.9564
Xception	2156.5169	0.1519	0.9563	0.1780	0.9562
VGG19	2479.0062	0.1322	0.9562	0.1339	0.9564

Test results were same across all the 5 models:

- F1 score: 0.9553143951484201
- Precision score: 0.9553143951484201
- Recall score: 1.0
- Accuracy score: 0.9553143951484201

The results are comparable.

We do notice that our own model takes a (statistically significant) shorter amount to train. However, it’s not a like to like comparison when matching this against the ImageNet trained models. Recall that we trained the ImageNet trained models for 4 epochs (as opposed to 3 for own model). In addition, the ImageNet trained models were trained with first three fourth layers frozen and the remainder layers as trainable. Perhaps training fewer layers would have given a similar performance with shorter training time.

With similar test scores and nothing else to separate on, I'd pick VGG19 since it has the lowest training and validation losses.

With a F1 score of 0.9553, I'd be comfortable evaluating VGG19 for real world applications.

Robustness

Does the model generalize well to unseen data? Is it sensitive to small changes in the data, or to outliers? Essentially, can we trust this model, and why or why not?

Before any kind of real world use, I'd do the following:

- Use image augmentation for training/testing to better handle real world images:
  - horizontal flip
  - vertical flip
  - noise
  - rotation
  - shift pixels
  - blur
- In the dataset, a chest x-ray can indicate upto 8 pathologies, with cardiomegaly being one of them. It's unclear without testing, how well our classifier can detect cardiomegaly when multiple pathologies are present. We'd benefit from training and testing on these kind of images
- Test the classifier with images outside of this dataset

Once we do this and the results are acceptable, I'd feel more comfortable with using the classifier for real world uses.

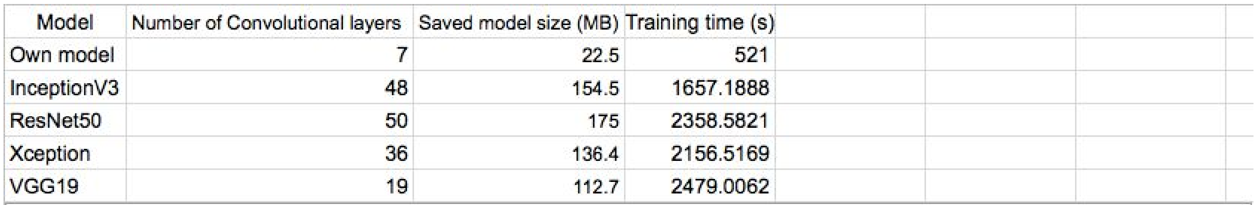
Justification

(See ***Model Evaluation and Validation*** section too)

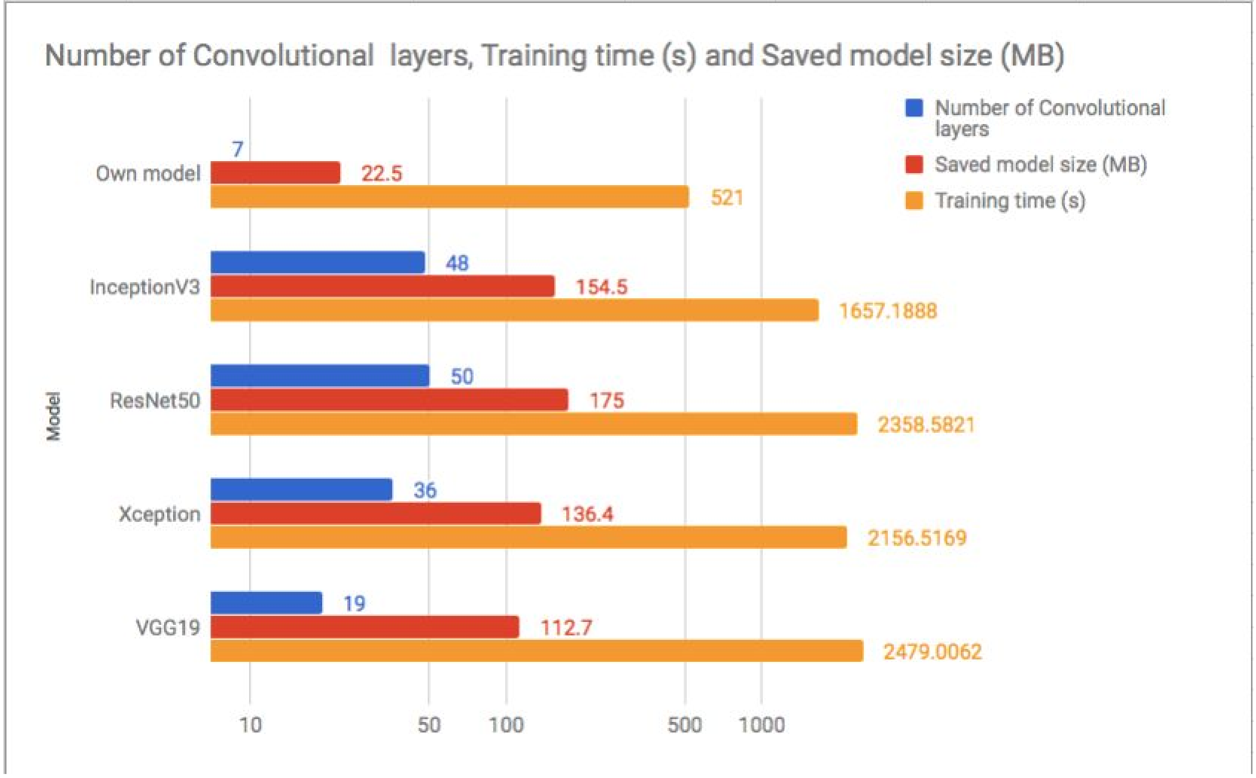
Using our own model's performance as a benchmark, we notice that the ImageNet trained models deliver at par, if not better results. InceptionV3, Xception and VGG19 do show lower training and validation losses but not by much. Perhaps we'll notice a difference when testing in the real world but based on the dataset test images, the performance is quite similar.

V. Conclusion

Free-Form Visualization



Model	Number of Convolutional layers	Saved model size (MB)	Training time (s)
Own model	7	22.5	521
InceptionV3	48	154.5	1657.1888
ResNet50	50	175	2358.5821
Xception	36	136.4	2156.5169
VGG19	19	112.7	2479.0062



This visualization compares the characteristics for our own model vs transfer learning on ImageNet trained models.

Our simple model with 7 conv layers gives similar performance to the other models, which have more (3x to 7x) layers. We'll need to test on real world data, perhaps images which are rotated, have background noise or are blurred. That may allow us to better understand, if one of these models generalizes better than others.

It’s also interesting that InceptionV3 was the fastest of the pre trained models, despite a high number of layers.

Saved model size seems to be correlated directly with the number of conv layers.

Reflection

Recap - here is the end-to-end solution:

- Obtain dataset, learn its characteristics and pick a subproblem (cardiomegaly)
- Image preprocessing to lower down training time/hardware requirements
- Setup AWS EC2 environment
- Architect own CNN model
- Transfer learn on ImageNet trained models
- Test classifier, compare performance and other characteristics

I’ve learnt quite a few lessons over the course of this project:

- Better understanding of resources needed for deep learning
  - Initially, I believed that by lowering the image’s resolution and/or picking a subset, I’d be able to train the models on my laptop and achieve good results. This proved impossible at the end without sacrificing performance and I ended up using AWS.
  - Using AWS spot instances, I was able to complete my work with a low bill of $2.45.
  - It was also amazing to see the Nvidia Tesla K80 GPU make light work of the training phase.
- Importance of going through the relevant research papers
  - After going through CheXNet and the other research papers, I had a good idea of the approach in general and the tweaks/optimizations involved for great results. This allowed me to have confidence in my approach and a few handy ideas for improving the results further, if needed.
- Keras ecosystem - still maturing
  - Although the scikit-learn APIs are quite mature and well documented, I didn’t have the same experience with Keras. It was a pain to figure out a way to add custom metrics or callbacks at the end of each training epoch.

Improvement

There’re several avenues of improvement:

- Use a higher resolution of images (we use 197x197) and see if we get better results
    - Dataset has images at 1024x1024 while the ChexNet research paper uses 224x224 images
    - I wonder if going up to maybe 256x256 delivers better results
- Augment the images with horizontal flipping
- As part of data pre-processing, standardize images to ImageNet’s mean and standard deviation (Was done in CheXNet but skipped in our implementation)
- More rigorous testing with real world X-ray images