Machine learning Engineer Nanodegree.

Capstone Report

Sudha Parvangada

April 26,2019

Domain background:

Musculoskeletal conditions affect more than 1.7 billion people worldwide, and are the most common cause of severe, long-term pain and disability, with 30 million emergency department visits annually and increasing. So, Stanford ML group has come up with a dataset MURA (**mu**sculoskeletal **ra**diographs), which is a large dataset of bone X-rays of finger, wrist, elbow, forearm, hand, humerus, and shoulder studies. (Ref0: https://arxiv.org/pdf/1712.06957.pdf).

My personal motivation is to build models to help in healthcare sector, which would identify most of false negatives which would remove untimely treatment and negligence out of the system. Accurately being able to predict the false positives would keep cost of health care down, by not subjecting the patients to a plethora of tests, which might not have been necessary to begin with and alleviate unnecessary stress on the patients.

Problem statement:

MURA, a large dataset of musculoskeletal radiographs containing 40,561 images from 14,863 studies, where each study is manually labeled by radiologists as either normal or abnormal. To evaluate models robustly and to get an estimate of radiologist performance, the Stanford group collected additional labels from six board-certified Stanford radiologists on the test set, consisting of 207 musculoskeletal studies. These 207 radiographs are considered as labeled data and will help in evaluating the performance of radiologist in labeling the rest of the radiographs. To keep the dataset manageable, we will consider the data for one study group, in my case -the wrist study.

Solution plan:

According to MURA paper:

"The model should take as input one or more views for a study of an upper extremity. On each view, the model should predict the probability of abnormality. Then the overall probability of abnormality for the study is computed by taking the arithmetic mean of the abnormality probabilities output by the network for each image. The model makes the binary prediction of abnormal if the probability of abnormality for the study is greater than 0.5."

The model used here is a 169 layered dense convolutional network architecture, which connects each layer to every other layer in a feed-forward fashion. The final fully connected is layer is replaced with a

layer that has a single output, which gives the arithmetic mean of all the probabilities for views under a study group. Then a sigmoid function is applied to the resulting probability to determine if its abnormal or not.

A study_data collection is built, by reading in the paths of the study group per patient, the count of number of study/views per study group and a label whether the study/view was abnormal or not.

A pytorch dataset is created which the reads the images from the study group, using the path for the views. It then creates a stack tensor for the list of images/views and returns a label (abnormal or normal) for the set of views under a study group.

The images are of varying sizes, hence needs to be normalized to imageNet's mean and standard deviation. The images are also resized to 224X224, with a horizontal flip and random rotation.

A dataloader (iterator) is then used to iterate through the dataset and apply transformations to it.

Evaluation Metrics:

According to the MURA paper:

"We compare radiologists and our model on the Cohen's kappa statistic, which expresses the agreement of each radiologist/model with the gold standard, defined as the majority vote of a disjoint group of radiologists. "

"On finger studies and wrist studies, model performance is comparable to the best radiologist performance."

	Radiologist 1	*Radiologist 2*	*Radiologist 3*	*MURA Model*
Wrist	0.791 (0.766, 0.817)	0.931 (0.922, 0.940)	0.931 (0.922, 0.940)	0.931 (0.922, 0.940)

Implementation:

I could not train my module with all the training data, which was 3000+ images. Due to the limitations on my personal laptop, the GPU could handle around 400 studys(views/radiographs). Beyond that I would get Out of Memory errors. Since there is no back propagation in the validation phase, I also made sure that we are not tracking the tensors for computing the gradients, as that was using up a lot of memory.

In the Mura paper, they had evaluated the model with Cohen-kappa statistic, as the imbalance in classes might be skewed with plain accuracy alone.
***"Observed Accuracy** is simply the number of instances that were classified correctly throughout the entire confusion matrix. To calculate **Observed Accuracy**, we simply add the number of instances that the* machine learning classifier *agreed with the* ground truth *label and divide by the total number of instances*.**"** – Ref1: https://stats.stackexchange.com/questions/82162/cohens-kappa-in-plain-english

Implementation:
I have derived the Observed accuracy by computing the number of predictions that were correct (running_corrects in train.py), both with abnormal and normal radiographs and then dividing it by the total data size.

*"The **Expected Accuracy** is directly related to the number of instances of each class (**class 1** and **class 2**), along with the number of instances that the* machine learning classifier *agreed with the* ground truth *label. To calculate **Expected Accuracy** for our confusion matrix, first multiply the* marginal frequency *of **class 1** for one "rater" by the* marginal frequency *of **class 1** for the second "rater" and divide by the total number of instances. The* marginal frequency *for a certain class by a certain "rater" is just the sum of all instances the "rater" indicated were that class**."*** – Ref1: above.

Implementation:
In my case I computed the abnormal_expected accuracy(class 1) by multiplying the abnormal label counts with abnormal predicted counts and dividing it by the total data size. Similarly computed the normal expected accuracy (class 2) by multiplying the normal label counts with normal predicted counts and dividing it by the total data size.
The expected accuracy for both classes was computed by adding the abnormal expected accuracy with normal expected accuracy and dividing it by the total size.
I then computed the Cohen's kappa statistic score, given by Kappa = (observed accuracy - expected accuracy)/(1 - expected accuracy).

Interpretation

*"There is not a standardized interpretation of the kappa statistic. According to Wikipedia (citing their paper), Landis and Koch considers 0-0.20 as slight, 0.21-0.40 as fair, 0.41-0.60 as moderate, 0.61-0.80 as substantial, and 0.81-1 as almost perfect. Fleiss considers kappas > 0.75 as excellent, 0.40-0.75 as fair to good, and < 0.40 as poor**."*** – *Ref1: above*

Implementation:

Since I had run the code for 5 epochs only with a subset of training data, I got a kappa score of `kappa: 0.3249,` which is considered fair or poor(Interpretation above).

Analysis:

Datasets and inputs:

As defined above, MURA is a dataset of musculoskeletal radiographs consisting of 14,982 `studies` from 12,251 `patients`, with a total of 40,895 `multi-view radiographic images`. Each `study` belongs to one of seven standard upper extremity radiographic `study types`: elbow, finger, forearm, hand, humerus, shoulder and wrist.

MURA dataset comes with `train` and `valid` folders containing corresponding datasets, `train.csv` and `valid.csv` containing paths of `radiographic images` and their labels. Each image is labeled as 1 (abnormal) or 0 (normal) based on whether its corresponding study is negative or positive, respectively, by the radiologists. Sometimes, these radiographic images are also referred as `views`.

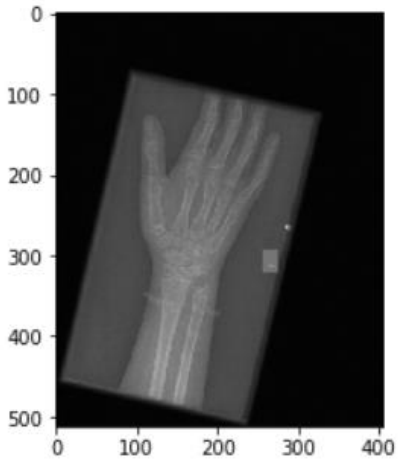
- The link to the data can be found at https://stanfordmlgroup.github.io/competitions/mura/
- There are 9751 views in training set of wrist images, with more than one image per patient. There are 3459 labeled data for patients in the training set. Off the 3459, there are 1325 abnormal (1) and 2134(normal).

The validation set include 659 views, with more than one image per patient.
There are 237 labeled data for patients in the validation set. Off the labeled data 97 are abnormal (1) and 140 normal (0).
The test data set needs to be formulated using samples from the training data set.

- The images are gray scale (as it's a radiograph), but is of varying dimensions, with bounding Boxes also of varying dimensions. There is more than one view per patient.

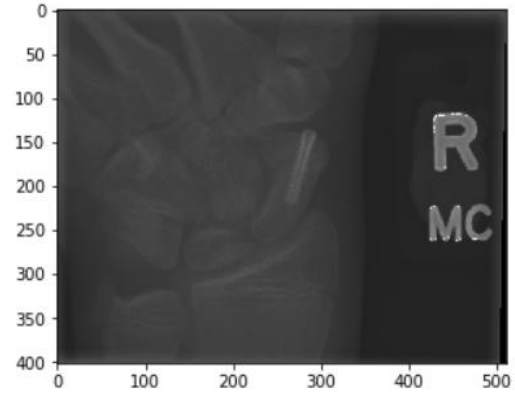
Here are some of the sample images, to show the varying size, shape and rotation and why it was important to resize, horizontal flip, apply rotation and normalize it to the mean and standard deviation of imageNet images in the preprocessing phase.

Size: (512, 406, 3)



Size: (512, 406, 3)

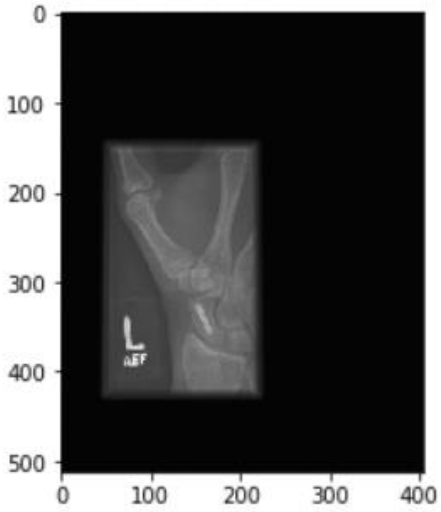
se.



Size: (402, 512, 3)

Size: (512, 343, 3)

Size: (512, 379, 3)



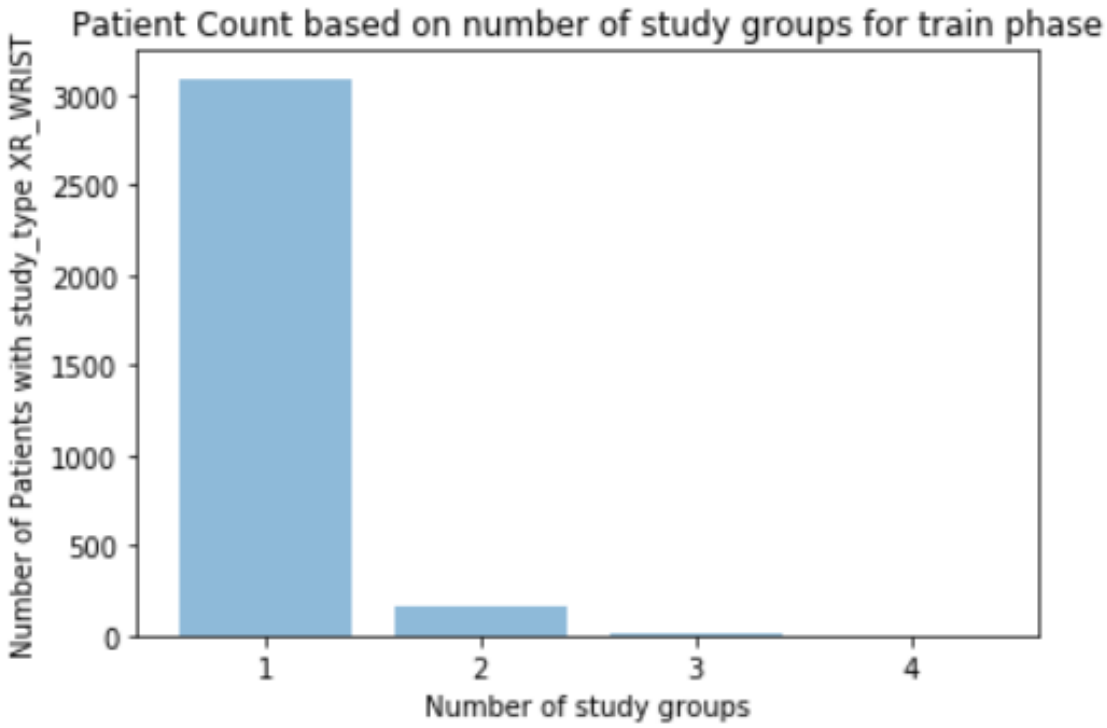
Size: (512, 405, 3)

Implementation:

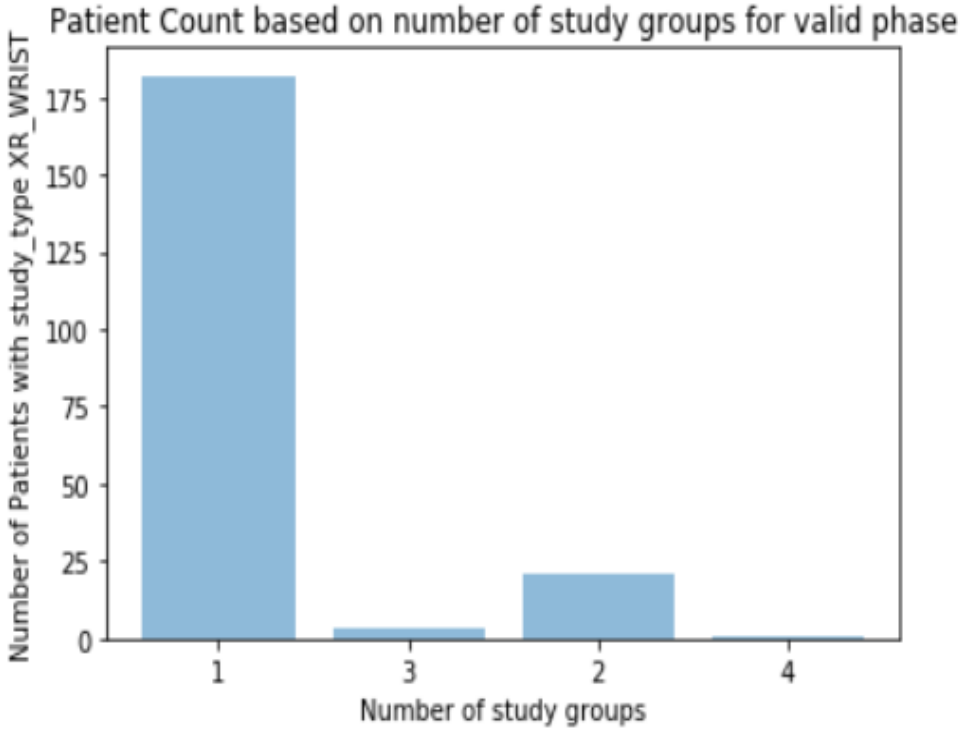
I used pytroch for reading the data pipeline, as it seemed fairly simple to read the data in the format I needed. get_path_imgCount_label function in mura.ipynb, reads the mura data set and dataframes them into paths ,counts and labels for each study. From the EDA I realized there could be more than one study group (positive or negative) per patient. Each study group could have more than one view or radiograph. So, the get_path_imgCount, looks into each study group for a patient and lists the path for each study group, with the total count of images for the study group and the label whether the prediction was abnormal (1) or normal (0).

	Path to study group/patient	Count	Label
0	./MURA-v1.1/train/XR_WRIST/patient00006/study1...	3	1
1	./MURA-v1.1/train/XR_WRIST/patient00012/study1...	4	0
2	./MURA-v1.1/train/XR_WRIST/patient00021/study1...	3	1
3	./MURA-v1.1/train/XR_WRIST/patient00021/study2...	3	1

Exploratory Data Analysis:



Train: 1: 3094, 2: 157, 3: 12, 4: 4

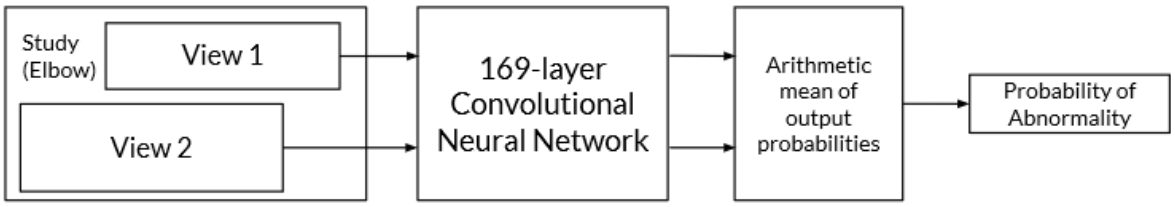


Valid: 1: 182, 2: 21, 3: 3, 4: 1

The above graphs give us the insight that the patient with the study type of XR_WRIST, could have more than one study group, meaning more than one set of x-rays taken. Each study group comes with the folder name like “study1_postive”, which I derive to get the label of normal or abnormal. Each of the study folder can contain more than one radiographs. So, in the graphs above we see that for training data, there were 157 patients with 2 views or radiographs and there was one patient in valid set that had 4 views.

Algorithms and techniques:

As the Mura paper (Ref0:) states that they have used the 169-layer convolutional neural network.



“*We used a 169-layer convolutional neural network to predict the probability of abnormality for each image in a study. The network uses a Dense Convolutional Network architecture – detailed in Huang et al. (2016) – which connects each layer to every other layer in a feed-forward fashion to make the optimization of deep networks tractable. We replaced the final fully connected layer with one that has a single output, after which we applied a sigmoid nonlinearity*” - Ref0:

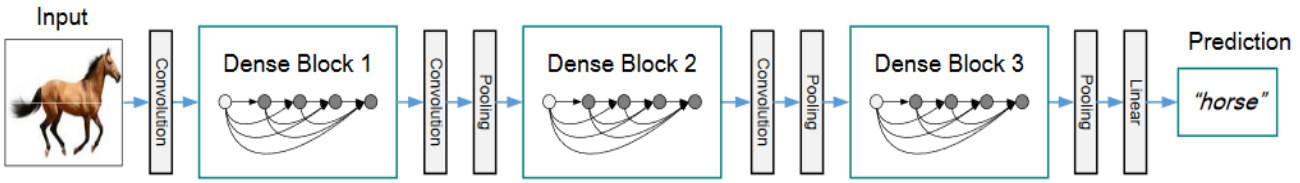


Figure 2. A deep DenseNet with three dense blocks. The layers between two adjacent blocks are referred to as transition layers and change feature map sizes via convolution and pooling.

DenseNet is an architecture that has dense blocks, interspersed with transition layers (Figure 2 - Ref2). DenseNet-169 has an initial feature size of 64, with a growth rate of 32(num of features that get added to each layer) and a block config of (6,12,32,32). It also has a bottle neck layer of 4 and drop rate of 0 by default.

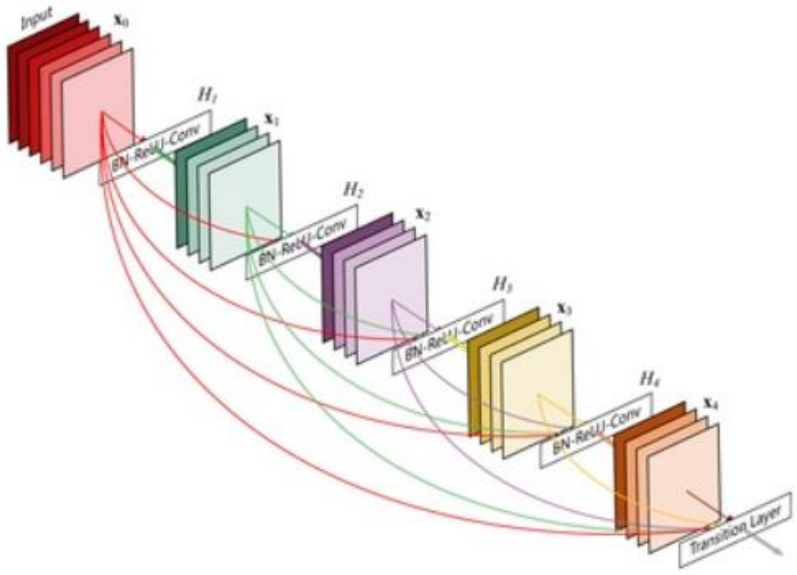


Figure 1: A dense block with 5 layers and growth rate 4.

Above Figure 1(Ref2:) depicts a dense block with 5 layers and a growth rate or feature set of 4.

The first Convolution in DenseNet-169 (in between the input and the first dense block in Figure:2 above) , has a convolution layer, with a BatchNormalization ,a ReLu activation ,a maxPool layer and a dropout layer if dropout is defined.

First Convolution: BN→ReLu→MaxPool→dropOut (if specified).

Each Dense block in DenseNet-169 contains Dense layers. Each Dense layer has a BatchNormalization layer, with a ReLu activation, a Convolution layer, a second Batch Normalization, second ReLu and second Convolution layer. Any drop out layer is added, if drop rate is specified to overcome any overfitting.

Dense Layer: BN→ReLu→Conv→BN→ReLu→Conv→dropOut (if specified).

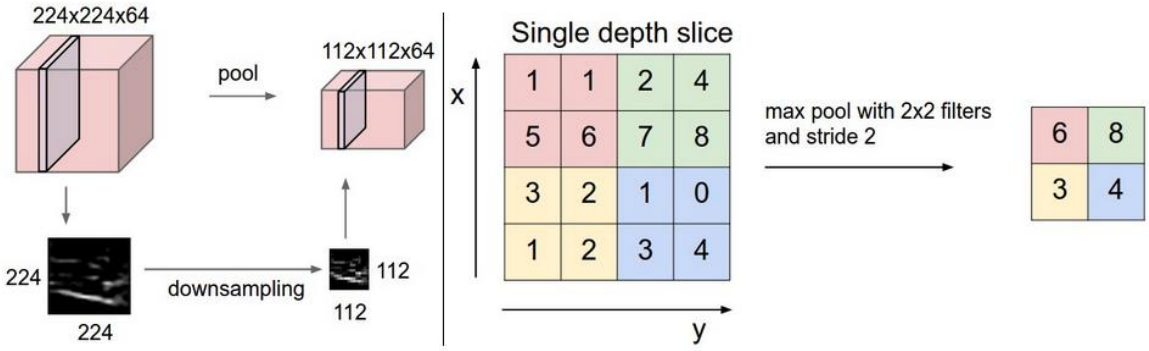
Dense blocks are interspersed with transition layer. Each Transition layer contains a BatchNormalization layer, a ReLu activation layer, a Conv layer and an Average Pooling layer.

Transition Layer: BN→ReLu→Conv→AvgPool

Final layers are a ReLu activation layer, an average Pooling layer with a sigmoid layer at the very end.

Final layers: ReLu→AvgPool→Sigmoid

Pooling Layer:(Ref: http://cs231n.github.io/convolutional-networks/)



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

Convolution Layer:

This layer will compute the outputs that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume.

click here to see the demo: http://cs231n.github.io/assets/conv-demo/index.html

RELU Layer:

This layer will apply an elementwise activation function, such as the *max*(0,*x*).

Batch Normalization Layer:

Batch Norm is the normalization of the output in each hidden layer. The figure below shows how to implement the Batch Norm in the neural nets.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

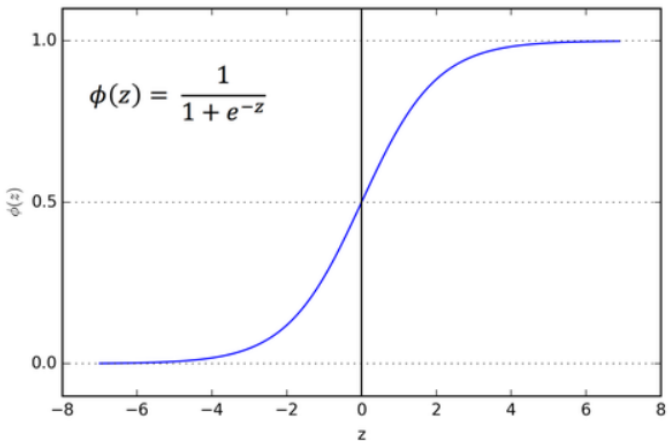
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Sigmoid Layer:

1. Sigmoid or Logistic Activation Function

The Sigmoid Function curve looks like a S-shape.



The main reason why we use sigmoid function is because it exists between **(0 to 1).** Therefore, it is especially used for models where we have to **predict the probability** as an output. Since probability of anything exists only between the range of **0 and 1,** sigmoid is the right choice.(Ref: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6)

Benchmark :

The one advantage of this data set is, it is baselined against a 169-layer convolutional neural network to detect and localize abnormalities. MURA paper states that:

"Our baseline uses a 169-layer convolutional neural network to detect and localize abnormalities. The model takes as input one or more views for a study of an upper extremity. On each view, our 169-layer convolutional neural network predicts the probability of abnormality."

Though for the XR_WRIST study, the paper states a very high accuracy of prediction, mine was fair at best, because I used a very small data set to train the model and do the validations.

Data Preprocessing:

MURA paper states that:

"Before feeding images into the network, we normalized each image to have the same mean and standard deviation of images in the ImageNet training set. We then scaled the variable-sized images to 224×224. We augmented the data during training by applying random lateral inversions and rotations." - Ref0:

Implementation:

The data or the images/views/radiographs came in different sizes and needs to be normalized. I used pytorch dataloaders(https://www.sanyamkapoor.com/machine-learning/pytorch-data-loaders/) to load data. With dataloaders, it's fairly simple to implement transforms. I used the transforms.Resize, to size the data to 224X224. I used the RandomFilp, RandomRotation and Normalize the data to the imageNet mean and standard deviation for the training data set.

On the validation set Resized to (224X224) and normalized to the imageNet's mean and standard deviation.

Implementation of metrics, algorithms and techniques:

DenseNet-169 is defined by the paper, Ref2: https://arxiv.org/abs/1608.06993

"*The weights of the network were initialized with weights from a model pretrained on ImageNet****."*** Ref0***:***

I used the available densenet-169 implementation
Ref3: https://pytorch.org/docs/master/_modules/torchvision/models/densenet.html, with a pretrained value of true, which initialized the weights from a model pretrained on imageNet.

"*For each image X of study type T in the training set, we optimized the weighted binary cross entropy loss***."-** Ref0:
I used the Weighted Binary cross entropy loss function, given by the formula - (wt1*y*log(p)+wt0*(1−y)log(1−p)). Where wt1 is the weight of abnormal class and wt0, is the weight of the normal class, y is the target labels and p is the input predictions. I had to normalize the weights for any skewing.

"*The network was trained end-to-end using Adam with default parameters*
*We trained the model using minibatches of size 8. We used an initial learning rate of 0.0001that is decayed by a factor of 10 each time the validation loss plateaus after an epoch****." –*** Ref0:
I used Adam for my optimizer with a learning rate of 0.0001. I also used the scheduler, lr_scheduler.reducelronplateau, which allows dynamic learning rate reducing based on validation measurements.
I ran it on one device (cuda0:) meaning one GPU, as my computer could not really use the parallelization given by using multiple GPUs at the same.

Complications and issues:

I primarily used pytorch, because the data pipelines with transformation could be easily created and had examples and tutorials everywhere. But then I did not know how to implement the denseNet model. I did not have an architectural blue print like proposed in the dog-project. So, thought of turning back to keras and OpenCV. But then the data manipulation seemed a lot harder and I needed to come up with my own model for denseNet as well. So, reverted back to pytorch. My Python knowledge is a "java to python translation", which does not work too well with Data manipulations and python classes. Just got to know how much I do not know☺

The one of Problems I faced was running Out of Memory every so often. I then realized by trial and error that it died on 423 items of my training data. So, I limited my training to a dataset of 400. Then on

validation set it would die randomly. I then used the no_grad option which would not cache all the tensor Variables(parameters) as back propagation was not needed.

Another problem I faced was there was dimensionality difference when one of the core modules was comparing labels with inputs. I knew they were of the same dimension. But then figured out after a while that the input was a tensor Variable of the format tensor ([1.0], device='cuda:0') and the label was a plain old int of value 1.

Refinement:

I used the weighted Binary cross entropy loss function instead of the plain Binary cross entropy function so if there were any class imbalances between the normal and abnormal radiographs could be addressed.

I also used a scheduler, lr_scheduler.reducelronplateau , which allows dynamic learning rate reducing based on validation measurements.

If I could use a cloud-based GPU, I probably could have trained the model with all of the 3400+ training data and could have had a better kappa score, which would have implied a better accuracy in prediction.

Results:

Model Evaluation and Validation:

The Test dataset had 200 images that was used to predict the normality or abnormality.

```
Number of abnormal labels=67, normal labels=133
Number of abnormal predictions =27, Number of normal predictions=80
Number of images predicted correctly =107 (both normal and abnormal)

Observed Accuracy =0.535
Expected Accuracy: 0.3112
Kappa Score (observed accuracy - expected accuracy)/(1 - expected accuracy) = 0.3249
Confusion Meter:
 [[0.60150373 0.39849624]
 [0.5970149  0.40298507]]
```

Since I had run the code for 5 epochs only with a subset of training data due to my GPU restrictions, I got a kappa score of **0.3249,** which is considered fair or poor (Interpretation above).

Justification:

For the XR_WRIST study type the model had a kappa of 0.931 (0.922, 0.940), where as I had a kappa of 0.3249(0.535, 0.3112) for a data set of 400 images from the training set of 3459 images. So, my justification is that had I more processing power I probably could have done lot better with more training data than the 400 images I trained with.

Conclusion:

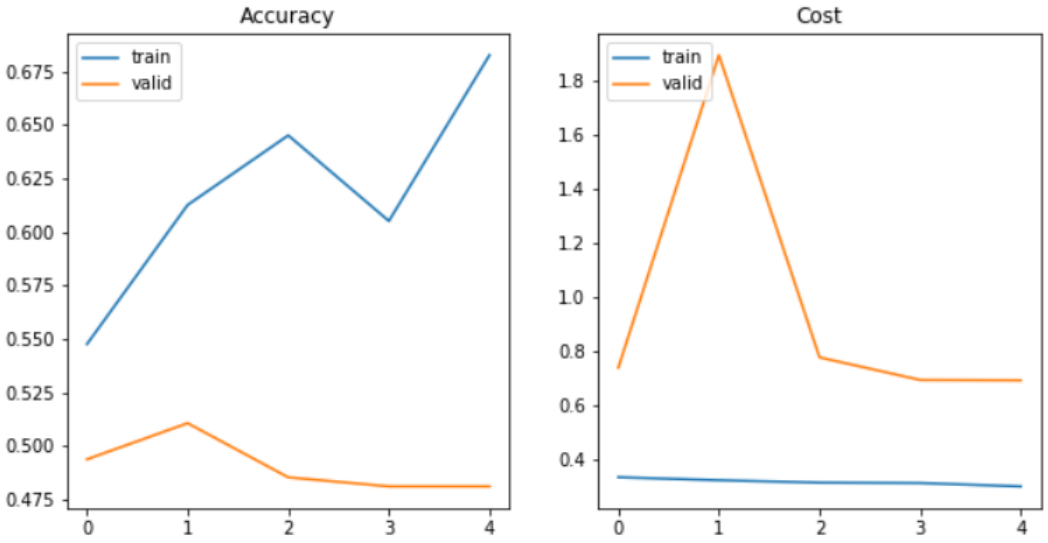
Free Form visualization:

I have evaluated the model for loss and absolute accuracy for the training and validation sets. The code is also printing the confusion matrix for the model for each epoch for both validation and training sets.

```
Train batches: 400
Valid batches: 237

Epoch 1/5
----------
Train:
train Loss: 0.3354 Acc: 0.5475
Confusion Meter:
 [[0.6487603  0.35123968]
 [0.6075949  0.39240506]]

Valid:
valid Loss: 0.7390 Acc: 0.4937
Confusion Meter:
 [[0.43571427 0.5642857 ]
 [0.4226804  0.57731956]]
Time elapsed: 4m 26s

Epoch 2/5
----------
Train:
train Loss: 0.3244 Acc: 0.6125
Confusion Meter:
 [[0.677686   0.32231405]
 [0.48734176 0.51265824]]
Valid:
valid Loss: 1.8916 Acc: 0.5105
Confusion Meter:
 [[0.6357143 0.3642857]
 [0.6701031 0.3298969]]
Time elapsed: 8m 45s

Epoch 3/5
----------
Train:
train Loss: 0.3156 Acc: 0.6450
Confusion Meter:
 [[0.7066116  0.29338843]
 [0.44936708 0.5506329 ]]
Valid:
valid Loss: 0.7773 Acc: 0.4852
Confusion Meter:
 [[0.54285717 0.45714286]
 [0.5979381  0.40206185]]
Epoch     2: reducing learning rate of group 0 to 1.0000e-05.
Time elapsed: 12m 48s

Epoch 4/5
----------
Train:
train Loss: 0.3140 Acc: 0.6050
Confusion Meter:
```

```
 [[0.45454547 0.54545456]
 [0.16455697 0.835443  ]]
Valid:
valid Loss: 0.6940 Acc: 0.4810
Confusion Meter:
 [[0.49285713 0.50714284]
 [0.53608245 0.46391752]]
Time elapsed: 16m 46s

Epoch 5/5
----------
Train:
train Loss: 0.3017 Acc: 0.6825
Confusion Meter:
 [[0.75206614 0.24793388]
 [0.42405063 0.5759494 ]]
Valid:
valid Loss: 0.6928 Acc: 0.4810
Confusion Meter:
 [[0.49285713 0.50714284]
 [0.53608245 0.46391752]]
```

The training loss seems to decrease over the epochs, but the validation loss jumps up in the second epoch and then goes down. The observed accuracy seems to vary from epoch to epoch. The confusion matrix is printed for every epoch.



Summary:

The goal of the model is to predict if a radiograph is abnormal (fractured) or normal for a particular study type(XR_WRIST). The input images were of varied shapes and sizes and had to be resized, normalized to imageNet's mean and standard deviation, horizontal flip and rotation transforms had to

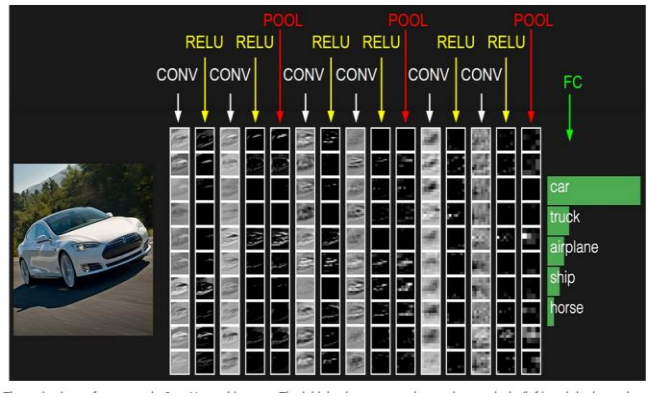
be applied. Used the same benchmark as the MURA paper – 169-layer convolutional neural network to detect and localize abnormalities.
Implemented a DenseNet-169 model, with varying dense blocks, interspersed with transition layers. The final layer was fed into a layer that computed the arithmetic mean of the probabilities for different views for a study group of a patient. The output of this layer was fed to a sigmoid function, that predicted if a view was abnormal > 0.5 or normal.
Used Adam as my optimizer and Weighted Binary cross entropy as my loss function.
Used the Cohen's kappa metrics and confusion matrix to evaluate the model.

Reflection:

Like I mentioned above, this project got to my attention there is a lot of things I don't know yet and this project is just the scratch on the surface. On the brighter side, I learnt Cohen's Kappa score calculation very well and why observed accuracy is not a good measure if the classes are imbalanced. Learnt about pytorch datasets and dataloaders. Using transforms with datasets was easy. Learnt about Tensor data and CUDA.
Got a real good idea of training a model, with optimizer and loss or error functions.
I probably could have done a better job with the matplotlib visualization had I known it well. Also wanted to really see the improvement in the model's learning, if I were to train the model with the complete training data set on a cloud-based computing environment. Would be nice to know, how much more my kappa score could have possibly risen.
Also, would have been good, to see the intermediate pictures after the image goes through different layers in the denseNet-169 model (not sure I know how).

Improvements:

- I should run the model on a better computing environment, so I could train the model with all the training data set. So, the problem here could be underfitting.
- I Could have provided the intermediate outputs of different layers when the model was training, like the picture of the car below.(Ref: http://cs231n.github.io/convolutional-networks/).



- Could have tried some images of non-radiographs like a tree or branch and see what the model would have done.