C1M1_Assignment

August 14, 2020

1 Chest X-Ray Medical Diagnosis with Deep Learning

Welcome to the first assignment of course 1!

In this assignment! You will explore medical image diagnosis by building a state-of-the-art chest X-ray classifier using Keras.

The assignment will walk through some of the steps of building and evaluating this deep learning classifier model. In particular, you will: - Pre-process and prepare a real-world X-ray dataset - Use transfer learning to retrain a DenseNet model for X-ray image classification - Learn a technique to handle class imbalance - Measure diagnostic performance by computing the AUC (Area Under the Curve) for the ROC (Receiver Operating Characteristic) curve - Visualize model activity using GradCAMs

In completing this assignment you will learn about the following topics:

- Data preparation
  - Visualizing data
  - Preventing data leakage
- Model Development
  - Addressing class imbalance
  - Leveraging pre-trained models using transfer learning
- Evaluation
  - AUC and ROC curves

1.1 Outline

Use these links to jump to specific sections of this assignment!

        * Section **??**
        * Section **??**
    - Section **??**
- Section **??**
    - Section **??**
- Section **??**
    - Section **??**
    - Section **??**

1. Import Packages and Functionsű

We'll make use of the following packages: - `numpy` and `pandas` is what we'll use to manipulate our data - `matplotlib.pyplot` and `seaborn` will be used to produce plots for visualization - `util` will provide the locally defined utility functions that have been provided for this assignment

We will also use several modules from the `keras` framework for building deep learning models.

Run the next cell to import all the necessary packages.

```
In [1]: import numpy as np
        import pandas as pd
        import seaborn as sns
        import matplotlib.pyplot as plt

        from keras.preprocessing.image import ImageDataGenerator
        from keras.applications.densenet import DenseNet121
        from keras.layers import Dense, GlobalAveragePooling2D
        from keras.models import Model
        from keras import backend as K

        from keras.models import load_model

        import util
```

```
Using TensorFlow backend.
```

2 Load the Datasets

For this assignment, we will be using the ChestX-ray8 dataset which contains 108,948 frontal-view X-ray images of 32,717 unique patients. - Each image in the data set contains multiple text-mined labels identifying 14 different pathological conditions. - These in turn can be used by physicians to diagnose 8 different diseases. - We will use this data to develop a single model that will provide binary classification predictions for each of the 14 labeled pathologies. - In other words it will predict 'positive' or 'negative' for each of the pathologies.

You can download the entire dataset for free here. - We have provided a ~1000 image subset of the images for you. - These can be accessed in the folder path stored in the `IMAGE_DIR` variable.

The dataset includes a CSV file that provides the labels for each X-ray.

To make your job a bit easier, we have processed the labels for our small sample and generated three new files to get you started. These three files are:

1. `nih/train-small.csv`: 875 images from our dataset to be used for training.
2. `nih/valid-small.csv`: 109 images from our dataset to be used for validation.
3. `nih/test.csv`: 420 images from our dataset to be used for testing.

This dataset has been annotated by consensus among four different radiologists for 5 of our 14 pathologies: - `Consolidation` - `Edema` - `Effusion` - `Cardiomegaly` - `Atelectasis`

Sidebar on meaning of 'class' It is worth noting that the word **'class'** is used in multiple ways is these discussions. - We sometimes refer to each of the 14 pathological conditions that are labeled in our dataset as a class. - But for each of those pathologies we are attempting to predict whether a certain condition is present (i.e. positive result) or absent (i.e. negative result). - These two possible labels of 'positive' or 'negative' (or the numerical equivalent of 1 or 0) are also typically referred to as classes. - Moreover, we also use the term in reference to software code 'classes' such as `ImageDataGenerator`.

As long as you are aware of all this though, it should not cause you any confusion as the term 'class' is usually clear from the context in which it is used.

Read in the data Let's open these files using the pandas library

```
In [2]: train_df = pd.read_csv("nih/train-small.csv")
        valid_df = pd.read_csv("nih/valid-small.csv")

        test_df = pd.read_csv("nih/test.csv")

        train_df.head()
```

Out[2]:

	Image	Atelectasis	Cardiomegaly	Consolidation	Edema	Effusion	Emphysema	Fibrosis	Hernia	Infiltration	Mass	Nodule	PatientId	Pleural_Thickening	Pneumonia	Pneumothorax
0	00008270_015.png	0	0	0	0	0	0	0	0	0	0	0	8270	0	0	0
1	00029855_001.png	1	0	0	0	1	0	0	0	1	0	0	29855	0	0	0
2	00001297_000.png	0	0	0	0	0	0	0	0	0	0	0	1297	1	0	0
3	00012359_002.png	0	0	0	0	0	0	0	0	0	0	0	12359	0	0	0
4	00017951_001.png	0	0	0	0	0	0	0	0	1	0	0	17951	0	0	0

```
In [3]: labels = ['Cardiomegaly',
                 'Emphysema',
```

```
                'Effusion',
                'Hernia',
                'Infiltration',
                'Mass',
                'Nodule',
                'Atelectasis',
                'Pneumothorax',
                'Pleural_Thickening',
                'Pneumonia',
                'Fibrosis',
                'Edema',
                'Consolidation']
```

2.1 Preventing Data Leakage It is worth noting that our dataset contains multiple images for each patient. This could be the case, for example, when a patient has taken multiple X-ray images at different times during their hospital visits. In our data splitting, we have ensured that the split is done on the patient level so that there is no data "leakage" between the train, validation, and test datasets.

Exercise 1 - Checking Data Leakage In the cell below, write a function to check whether there is leakage between two datasets. We'll use this to make sure there are no patients in the test set that are also present in either the train or validation sets.

Hints

Make use of python's set.intersection() function.

In order to match the automatic grader's expectations, please start the line of code with df1_patients_unique... [continue your code here]

```
In [4]: # UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
        def check_for_leakage(df1, df2, patient_col):
            """
            Return True if there any patients are in both df1 and df2.

            Args:
                df1 (dataframe): dataframe describing first dataset
                df2 (dataframe): dataframe describing second dataset
                patient_col (str): string name of column with patient IDs

            Returns:
                leakage (bool): True if there is leakage, otherwise False
            """

            ### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###

            df1_patients_unique = set(df1[patient_col].unique().tolist())
            df2_patients_unique = set(df2[patient_col].unique().tolist())

            patients_in_both_groups = df1_patients_unique.intersection(df2_patients_unique)

            # leakage contains true if there is patient overlap, otherwise false.
```

```
            leakage = len(patients_in_both_groups) >= 1  # boolean (true if there is at least
            ### END CODE HERE ###

            return leakage
```

```
In [5]: # test
        print("test case 1")
        df1 = pd.DataFrame({'patient_id': [0, 1, 2]})
        df2 = pd.DataFrame({'patient_id': [2, 3, 4]})
        print("df1")
        print(df1)
        print("df2")
        print(df2)
        print(f"leakage output: {check_for_leakage(df1, df2, 'patient_id')}")
        print("-------------------------------------")
        print("test case 2")
        df1 = pd.DataFrame({'patient_id': [0, 1, 2]})
        df2 = pd.DataFrame({'patient_id': [3, 4, 5]})
        print("df1:")
        print(df1)
        print("df2:")
        print(df2)

        print(f"leakage output: {check_for_leakage(df1, df2, 'patient_id')}")
```

```
test case 1
df1
   patient_id
0           0
1           1
2           2
df2
   patient_id
0           2
1           3
2           4
leakage output: True
-------------------------------------
test case 2
df1:
   patient_id
0           0
1           1
2           2
df2:
   patient_id
0           3
```

```
1           4
2           5
leakage output: False
```

Expected output

```
test case 1
df1
   patient_id
0           0
1           1
2           2
df2
   patient_id
0           2
1           3
2           4
leakage output: True
--------------------------------------
test case 2
df1:
   patient_id
0           0
1           1
2           2
df2:
   patient_id
0           3
1           4
2           5
leakage output: False
```

Run the next cell to check if there are patients in both train and test or in both valid and test.

```
In [6]: print("leakage between train and test: {}".format(check_for_leakage(train_df, test_df,
        print("leakage between valid and test: {}".format(check_for_leakage(valid_df, test_df,
```

```
leakage between train and test: False
leakage between valid and test: False
```

If we get `False` for both, then we're ready to start preparing the datasets for training. Remember to always check for data leakage!

2.2 Preparing Images

With our dataset splits ready, we can now proceed with setting up our model to consume them. - For this we will use the off-the-shelf ImageDataGenerator class from the Keras framework, which allows us to build a "generator" for images specified in a dataframe. - This class also provides support for basic data augmentation such as random horizontal flipping of images. - We also

use the generator to transform the values in each batch so that their mean is 0 and their standard deviation is 1. - This will facilitate model training by standardizing the input distribution. - The generator also converts our single channel X-ray images (gray-scale) to a three-channel format by repeating the values in the image across all channels. - We will want this because the pre-trained model that we'll use requires three-channel inputs.

Since it is mainly a matter of reading and understanding Keras documentation, we have implemented the generator for you. There are a few things to note: 1. We normalize the mean and standard deviation of the data 3. We shuffle the input after each epoch. 4. We set the image size to be 320px by 320px

```
In [7]: def get_train_generator(df, image_dir, x_col, y_cols, shuffle=True, batch_size=8, seed=
            """
            Return generator for training set, normalizing using batch
            statistics.

            Args:
              train_df (dataframe): dataframe specifying training data.
              image_dir (str): directory where image files are held.
              x_col (str): name of column in df that holds filenames.
              y_cols (list): list of strings that hold y labels for images.
              batch_size (int): images per batch to be fed into model during training.
              seed (int): random seed.
              target_w (int): final width of input images.
              target_h (int): final height of input images.

            Returns:
                train_generator (DataFrameIterator): iterator over training set
            """
            print("getting train generator...")
            # normalize images
            image_generator = ImageDataGenerator(
                samplewise_center=True,
                samplewise_std_normalization= True)

            # flow from directory with specified batch size
            # and target image size
            generator = image_generator.flow_from_dataframe(
                    dataframe=df,
                    directory=image_dir,
                    x_col=x_col,
                    y_col=y_cols,
                    class_mode="raw",
                    batch_size=batch_size,
                    shuffle=shuffle,
                    seed=seed,
                    target_size=(target_w,target_h))

            return generator
```

Build a separate generator for valid and test sets Now we need to build a new generator for validation and testing data.

Why can't we use the same generator as for the training data?

Look back at the generator we wrote for the training data. - It normalizes each image **per batch**, meaning that it uses batch statistics. - We should not do this with the test and validation data, since in a real life scenario we don't process incoming images a batch at a time (we process one image at a time). - Knowing the average per batch of test data would effectively give our model an advantage.
- The model should not have any information about the test data.

What we need to do is normalize incoming test data using the statistics **computed from the training set**. * We implement this in the function below. * There is one technical note. Ideally, we would want to compute our sample mean and standard deviation using the entire training set. * However, since this is extremely large, that would be very time consuming. * In the interest of time, we'll take a random sample of the dataset and calcualte the sample mean and sample standard deviation.

```
In [8]: def get_test_and_valid_generator(valid_df, test_df, train_df, image_dir, x_col, y_cols
            """
            Return generator for validation set and test test set using
            normalization statistics from training set.

            Args:
              valid_df (dataframe): dataframe specifying validation data.
              test_df (dataframe): dataframe specifying test data.
              train_df (dataframe): dataframe specifying training data.
              image_dir (str): directory where image files are held.
              x_col (str): name of column in df that holds filenames.
              y_cols (list): list of strings that hold y labels for images.
              sample_size (int): size of sample to use for normalization statistics.
              batch_size (int): images per batch to be fed into model during training.
              seed (int): random seed.
              target_w (int): final width of input images.
              target_h (int): final height of input images.

            Returns:
                test_generator (DataFrameIterator) and valid_generator: iterators over test se
            """
            print("getting train and valid generators...")
            # get generator to sample dataset
            raw_train_generator = ImageDataGenerator().flow_from_dataframe(
                dataframe=train_df,
                directory=IMAGE_DIR,
                x_col="Image",
                y_col=labels,
                class_mode="raw",
                batch_size=sample_size,
                shuffle=True,
                target_size=(target_w, target_h))
```

```
    # get data sample
    batch = raw_train_generator.next()
    data_sample = batch[0]

    # use sample to fit mean and std for test set generator
    image_generator = ImageDataGenerator(
        featurewise_center=True,
        featurewise_std_normalization= True)

    # fit generator to sample from training data
    image_generator.fit(data_sample)

    # get test generator
    valid_generator = image_generator.flow_from_dataframe(
            dataframe=valid_df,
            directory=image_dir,
            x_col=x_col,
            y_col=y_cols,
            class_mode="raw",
            batch_size=batch_size,
            shuffle=False,
            seed=seed,
            target_size=(target_w,target_h))

    test_generator = image_generator.flow_from_dataframe(
            dataframe=test_df,
            directory=image_dir,
            x_col=x_col,
            y_col=y_cols,
            class_mode="raw",
            batch_size=batch_size,
            shuffle=False,
            seed=seed,
            target_size=(target_w,target_h))
    return valid_generator, test_generator
```

With our generator function ready, let's make one generator for our training data and one each of our test and validation datasets.

```
In [9]: IMAGE_DIR = "nih/images-small/"
        train_generator = get_train_generator(train_df, IMAGE_DIR, "Image", labels)
        valid_generator, test_generator= get_test_and_valid_generator(valid_df, test_df, train_
```

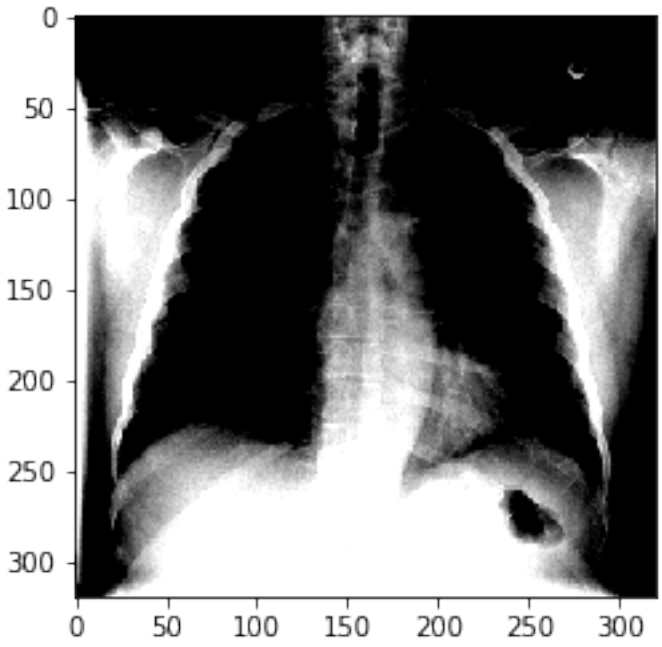
```
getting train generator...
Found 1000 validated image filenames.
getting train and valid generators...
Found 1000 validated image filenames.
```

```
Found 200 validated image filenames.
Found 420 validated image filenames.
```

Let's peek into what the generator gives our model during training and validation. We can do this by calling the `__get_item__(index)` function:

```
In [10]: x, y = train_generator.__getitem__(0)
         plt.imshow(x[0]);
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
```

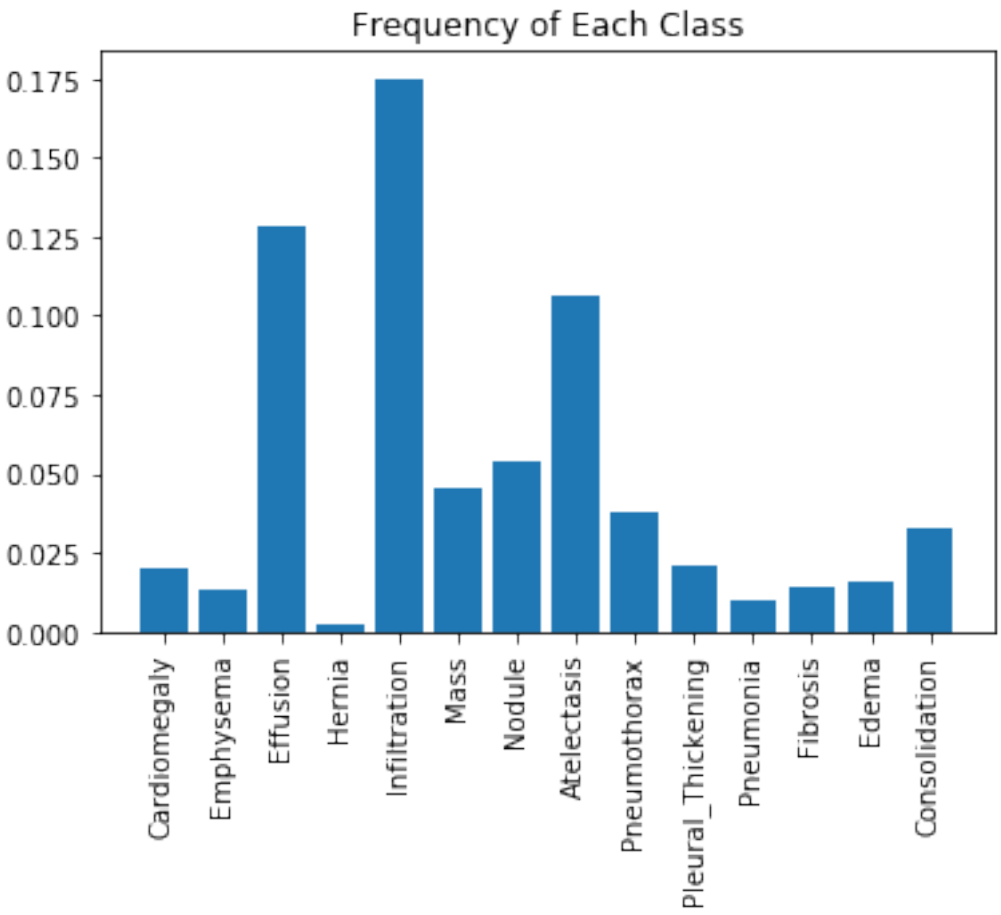


3 Model Development

Now we'll move on to model training and development. We have a few practical challenges to deal with before actually training a neural network, though. The first is class imbalance.

3.1 Addressing Class Imbalance One of the challenges with working with medical diagnostic datasets is the large class imbalance present in such datasets. Let's plot the frequency of each of the labels in our dataset:

```
In [11]: plt.xticks(rotation=90)
         plt.bar(x=labels, height=np.mean(train_generator.labels, axis=0))
         plt.title("Frequency of Each Class")
         plt.show()
```



We can see from this plot that the prevalance of positive cases varies significantly across the different pathologies. (These trends mirror the ones in the full dataset as well.) * The `Hernia` pathology has the greatest imbalance with the proportion of positive training cases being about 0.2%. * But even the `Infiltration` pathology, which has the least amount of imbalance, has only 17.5% of the training cases labelled positive.

Ideally, we would train our model using an evenly balanced dataset so that the positive and negative training cases would contribute equally to the loss.

If we use a normal cross-entropy loss function with a highly unbalanced dataset, as we are seeing here, then the algorithm will be incentivized to prioritize the majority class (i.e negative in our case), since it contributes more to the loss.

Impact of class imbalance on loss function Let's take a closer look at this. Assume we would have used a normal cross-entropy loss for each pathology. We recall that the cross-entropy loss contribution from the i^{th} training data case is:

$$\mathcal{L}_{cross-entropy}(x_i) = -(y_i \log(f(x_i)) + (1 - y_i) \log(1 - f(x_i))),$$

where x_i and y_i are the input features and the label, and $f(x_i)$ is the output of the model, i.e. the probability that it is positive.

Note that for any training case, either $y_i = 0$ or else $(1 - y_i) = 0$, so only one of these terms contributes to the loss (the other term is multiplied by zero, and becomes zero).

We can rewrite the overall average cross-entropy loss over the entire training set $\mathcal{D}$ of size N as follows:

$$\mathcal{L}_{cross-entropy}(\mathcal{D}) = -\frac{1}{N}\Big(\sum_{\text{positive examples}} \log(f(x_i)) + \sum_{\text{negative examples}} \log(1 - f(x_i))\Big).$$

Using this formulation, we can see that if there is a large imbalance with very few positive training cases, for example, then the loss will be dominated by the negative class. Summing the contribution over all the training cases for each class (i.e. pathological condition), we see that the contribution of each class (i.e. positive or negative) is:

$$freq_p = \frac{\text{number of positive examples}}{N}$$

and

$$freq_n = \frac{\text{number of negative examples}}{N}.$$

Exercise 2 - Computing Class Frequencies Complete the function below to calculate these frequences for each label in our dataset.

Hints

Use numpy.sum(a, axis=), and choose the axis (0 or 1)

```
In [12]: # UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
         def compute_class_freqs(labels):
             """
             Compute positive and negative frequences for each class.

             Args:
                 labels (np.array): matrix of labels, size (num_examples, num_classes)
             Returns:
                 positive_frequencies (np.array): array of positive frequences for each
                                                  class, size (num_classes)
                 negative_frequencies (np.array): array of negative frequences for each
                                                  class, size (num_classes)
             """
             ### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###

             # total number of patients (rows)
             N = labels.shape[0]

             positive_frequencies = np.sum(labels, axis=0) / labels.shape[0]
             negative_frequencies = 1 - positive_frequencies

             ### END CODE HERE ###
             return positive_frequencies, negative_frequencies
```

```
In [13]: # Test
         labels_matrix = np.array(
             [[1, 0, 0],
              [0, 1, 1],
              [1, 0, 1],
              [1, 1, 1],
              [1, 0, 1]]
         )
         print("labels:")
         print(labels_matrix)

         test_pos_freqs, test_neg_freqs = compute_class_freqs(labels_matrix)

         print(f"pos freqs: {test_pos_freqs}")

         print(f"neg freqs: {test_neg_freqs}")
```

```
labels:
[[1 0 0]
 [0 1 1]
 [1 0 1]
 [1 1 1]
 [1 0 1]]
pos freqs: [0.8 0.4 0.8]
neg freqs: [0.2 0.6 0.2]
```

Expected output

```
labels:
[[1 0 0]
 [0 1 1]
 [1 0 1]
 [1 1 1]
 [1 0 1]]
pos freqs: [0.8 0.4 0.8]
neg freqs: [0.2 0.6 0.2]
```

Now we'll compute frequencies for our training data.

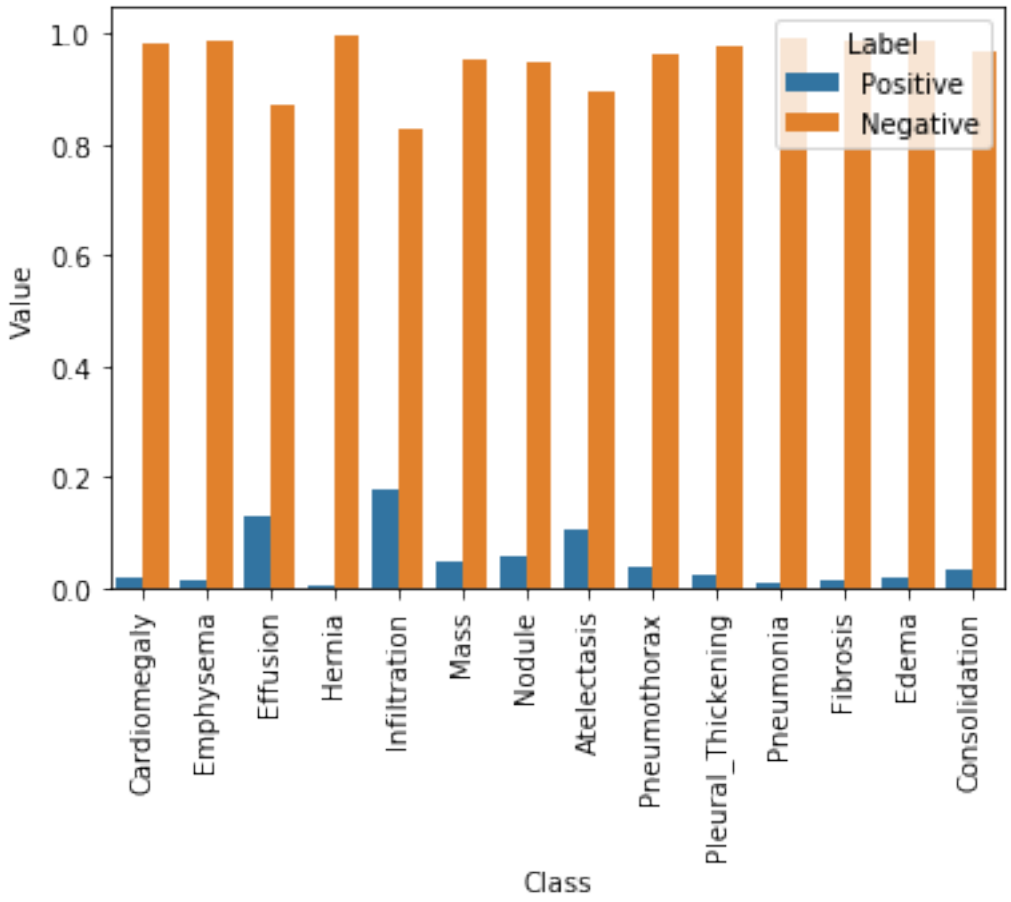
```
In [14]: freq_pos, freq_neg = compute_class_freqs(train_generator.labels)
         freq_pos
```

```
Out[14]: array([0.02 , 0.013, 0.128, 0.002, 0.175, 0.045, 0.054, 0.106, 0.038,
                0.021, 0.01 , 0.014, 0.016, 0.033])
```

Let's visualize these two contribution ratios next to each other for each of the pathologies:

```
In [15]: data = pd.DataFrame({"Class": labels, "Label": "Positive", "Value": freq_pos})
         data = data.append([{"Class": labels[l], "Label": "Negative", "Value": v} for l,v in
```

```
plt.xticks(rotation=90)
f = sns.barplot(x="Class", y="Value", hue="Label" ,data=data)
```



As we see in the above plot, the contributions of positive cases is significantly lower than that of the negative ones. However, we want the contributions to be equal. One way of doing this is by multiplying each example from each class by a class-specific weight factor, w_{pos} and w_{neg}, so that the overall contribution of each class is the same.

To have this, we want

$$w_{pos} \times freq_p = w_{neg} \times freq_n,$$

which we can do simply by taking

$$w_{pos} = freq_{neg}$$
$$w_{neg} = freq_{pos}$$

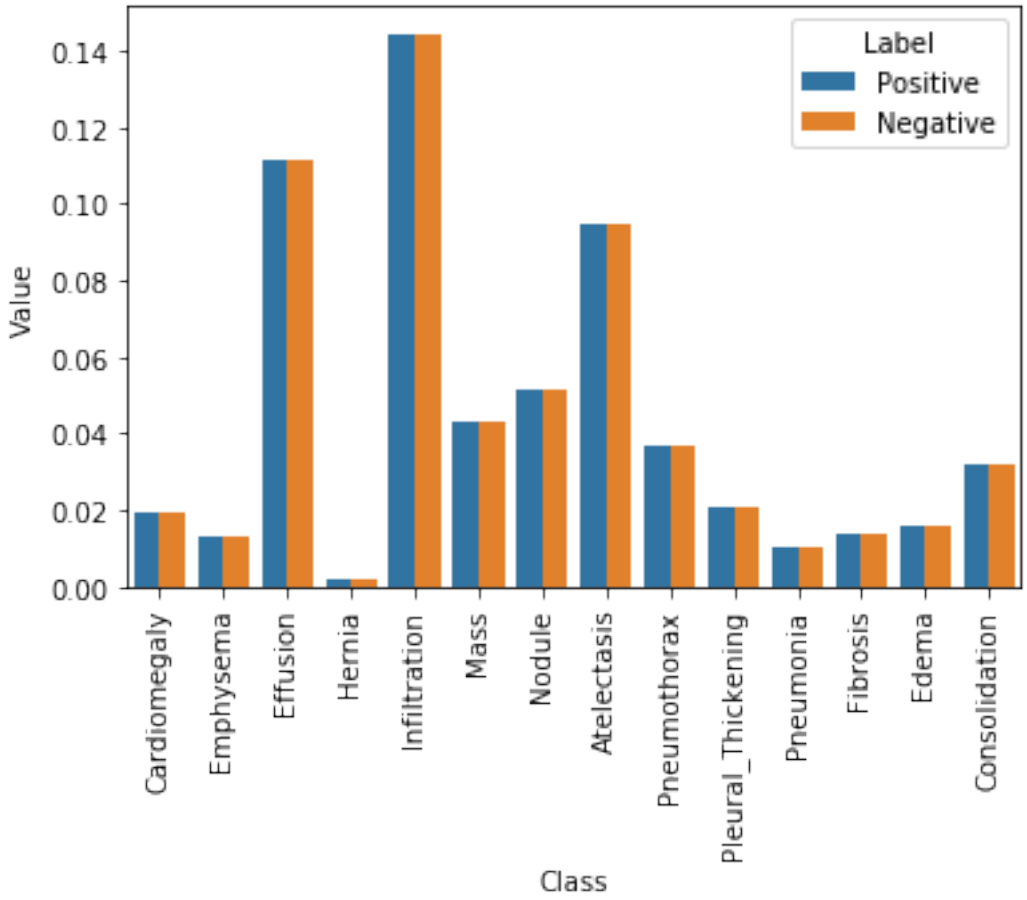
This way, we will be balancing the contribution of positive and negative labels.

```
In [16]: pos_weights = freq_neg
         neg_weights = freq_pos
```

```
pos_contribution = freq_pos * pos_weights
neg_contribution = freq_neg * neg_weights
```

Let's verify this by graphing the two contributions next to each other again:

```
In [17]: data = pd.DataFrame({"Class": labels, "Label": "Positive", "Value": pos_contribution}
         data = data.append([{"Class": labels[l], "Label": "Negative", "Value": v}
                             for l,v in enumerate(neg_contribution)], ignore_index=True)
         plt.xticks(rotation=90)
         sns.barplot(x="Class", y="Value", hue="Label" ,data=data);
```



As the above figure shows, by applying these weightings the positive and negative labels within each class would have the same aggregate contribution to the loss function. Now let's implement such a loss function.

After computing the weights, our final weighted loss for each training case will be

$$\mathcal{L}^{w}_{cross-entropy}(x) = -(w_p y \log(f(x)) + w_n(1-y)\log(1-f(x))).$$

Exercise 3 - Weighted Loss Fill out the `weighted_loss` function below to return a loss function that calculates the weighted loss for each batch. Recall that for the multi-class loss, we

add up the average loss for each individual class. Note that we also want to add a small value, ϵ, to the predicted values before taking their logs. This is simply to avoid a numerical error that would otherwise occur if the predicted value happens to be zero.

Note Please use Keras functions to calculate the mean and the log.

- Keras.mean
- Keras.log

```
In [18]: # UNQ_C3 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
         def get_weighted_loss(pos_weights, neg_weights, epsilon=1e-7):
             """
             Return weighted loss function given negative weights and positive weights.

             Args:
               pos_weights (np.array): array of positive weights for each class, size (num_cla
               neg_weights (np.array): array of negative weights for each class, size (num_cla

             Returns:
               weighted_loss (function): weighted loss function
             """
             def weighted_loss(y_true, y_pred):
                 """
                 Return weighted loss value.

                 Args:
                     y_true (Tensor): Tensor of true labels, size is (num_examples, num_classe
                     y_pred (Tensor): Tensor of predicted labels, size is (num_examples, num_c
                 Returns:
                     loss (Float): overall scalar loss summed across all classes
                 """
                 # initialize loss to zero
                 loss = 0.0

                 ### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###

                 for i in range(len(pos_weights)):
                     # for each class, add average weighted loss for that class
                     loss += -(K.mean( pos_weights[i] * y_true[:,i] * K.log(y_pred[:,i] + epsi
                                       neg_weights[i] * (1 - y_true[:,i]) * K.log(1 - y_pred
                 return loss

                 ### END CODE HERE ###
             return weighted_loss
```

Now let's test our function with some simple cases.

```
In [19]: # Test
         sess = K.get_session()
```

```
        with sess.as_default() as sess:
            print("Test example:\n")
            y_true = K.constant(np.array(
                [[1, 1, 1],
                 [1, 1, 0],
                 [0, 1, 0],
                 [1, 0, 1]]
            ))
            print("y_true:\n")
            print(y_true.eval())

            w_p = np.array([0.25, 0.25, 0.5])
            w_n = np.array([0.75, 0.75, 0.5])
            print("\nw_p:\n")
            print(w_p)

            print("\nw_n:\n")
            print(w_n)

            y_pred_1 = K.constant(0.7*np.ones(y_true.shape))
            print("\ny_pred_1:\n")
            print(y_pred_1.eval())

            y_pred_2 = K.constant(0.3*np.ones(y_true.shape))
            print("\ny_pred_2:\n")
            print(y_pred_2.eval())

            # test with a large epsilon in order to catch errors
            L = get_weighted_loss(w_p, w_n, epsilon=1)

            print("\nIf we weighted them correctly, we expect the two losses to be the same.")
            L1 = L(y_true, y_pred_1).eval()
            L2 = L(y_true, y_pred_2).eval()
            print(f"\nL(y_pred_1)= {L1:.4f}, L(y_pred_2)= {L2:.4f}")
            print(f"Difference is L1 - L2 = {L1 - L2:.4f}")
```

```
Test example:

y_true:

[[1. 1. 1.]
 [1. 1. 0.]
 [0. 1. 0.]
 [1. 0. 1.]]

w_p:

[0.25 0.25 0.5 ]
```

```
w_n:

[0.75 0.75 0.5 ]

y_pred_1:

[[0.7 0.7 0.7]
 [0.7 0.7 0.7]
 [0.7 0.7 0.7]
 [0.7 0.7 0.7]]

y_pred_2:

[[0.3 0.3 0.3]
 [0.3 0.3 0.3]
 [0.3 0.3 0.3]
 [0.3 0.3 0.3]]

If we weighted them correctly, we expect the two losses to be the same.

L(y_pred_1)= -0.4956, L(y_pred_2)= -0.4956
Difference is L1 - L2 = 0.0000
```

Additional check If you implemented the function correctly, then if the epsilon for the `get_weighted_loss` is set to 1, the weighted losses will be as follows:

```
L(y_pred_1)= -0.4956, L(y_pred_2)= -0.4956
```

If you are missing something in your implementation, you will see a different set of losses for L1 and L2 (even though L1 and L2 will be the same).

3.3 DenseNet121

Next, we will use a pre-trained DenseNet121 model which we can load directly from Keras and then add two layers on top of it: 1. A `GlobalAveragePooling2D` layer to get the average of the last convolution layers from DenseNet121. 2. A `Dense` layer with `sigmoid` activation to get the prediction logits for each of our classes.

We can set our custom loss function for the model by specifying the `loss` parameter in the `compile()` function.

```
In [20]: # create the base pre-trained model
         base_model = DenseNet121(weights='./nih/densenet.hdf5', include_top=False)

         x = base_model.output

         # add a global spatial average pooling layer
         x = GlobalAveragePooling2D()(x)
```

```
        # and a logistic layer
        predictions = Dense(len(labels), activation="sigmoid")(x)

        model = Model(inputs=base_model.input, outputs=predictions)
        model.compile(optimizer='adam', loss=get_weighted_loss(pos_weights, neg_weights))
```

```
WARNING:tensorflow:From /opt/conda/lib/python3.6/site-packages/tensorflow_core/python/ops/resou
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
WARNING:tensorflow:From /opt/conda/lib/python3.6/site-packages/keras/backend/tensorflow_backend
```

```
WARNING:tensorflow:From /opt/conda/lib/python3.6/site-packages/keras/backend/tensorflow_backend
```

4 Training [optional]

With our model ready for training, we will use the `model.fit()` function in Keras to train our model. - We are training on a small subset of the dataset (~1%).
- So what we care about at this point is to make sure that the loss on the training set is decreasing.

Since training can take a considerable time, for pedagogical purposes we have chosen not to train the model here but rather to load a set of pre-trained weights in the next section. However, you can use the code shown below to practice training the model locally on your machine or in Colab.

NOTE: Do not run the code below on the Coursera platform as it will exceed the platform's memory limitations.

Python Code for training the model:

```
history = model.fit_generator(train_generator,
                              validation_data=valid_generator,
                              steps_per_epoch=100,
                              validation_steps=25,
                              epochs = 3)
```

```
plt.plot(history.history['loss'])
plt.ylabel("loss")
plt.xlabel("epoch")
plt.title("Training Loss Curve")
plt.show()
```

4.1 Training on the Larger Dataset

Given that the original dataset is 40GB+ in size and the training process on the full dataset takes a few hours, we have trained the model on a GPU-equipped machine for you and provided the weights file from our model (with a batch size of 32 instead) to be used for the rest of this assignment.

The model architecture for our pre-trained model is exactly the same, but we used a few useful Keras "callbacks" for this training. Do spend time to read about these callbacks at your leisure as they will be very useful for managing long-running training sessions:

1. You can use `ModelCheckpoint` callback to monitor your model's `val_loss` metric and keep a snapshot of your model at the point.

2. You can use the `TensorBoard` to use the Tensorflow Tensorboard utility to monitor your runs in real-time.
3. You can use the `ReduceLROnPlateau` to slowly decay the learning rate for your model as it stops getting better on a metric such as `val_loss` to fine-tune the model in the final steps of training.
4. You can use the `EarlyStopping` callback to stop the training job when your model stops getting better in it's validation loss. You can set a `patience` value which is the number of epochs the model does not improve after which the training is terminated. This callback can also conveniently restore the weights for the best metric at the end of training to your model.

You can read about these callbacks and other useful Keras callbacks here.

Let's load our pre-trained weights into the model now:

```
In [21]: model.load_weights("./nih/pretrained_model.h5")
```

5 Prediction and Evaluation

Now that we have a model, let's evaluate it using our test set. We can conveniently use the `predict_generator` function to generate the predictions for the images in our test set.

Note: The following cell can take about 4 minutes to run.

```
In [22]: predicted_vals = model.predict_generator(test_generator, steps = len(test_generator))
```

```
WARNING:tensorflow:From /opt/conda/lib/python3.6/site-packages/keras/backend/tensorflow_backend
```

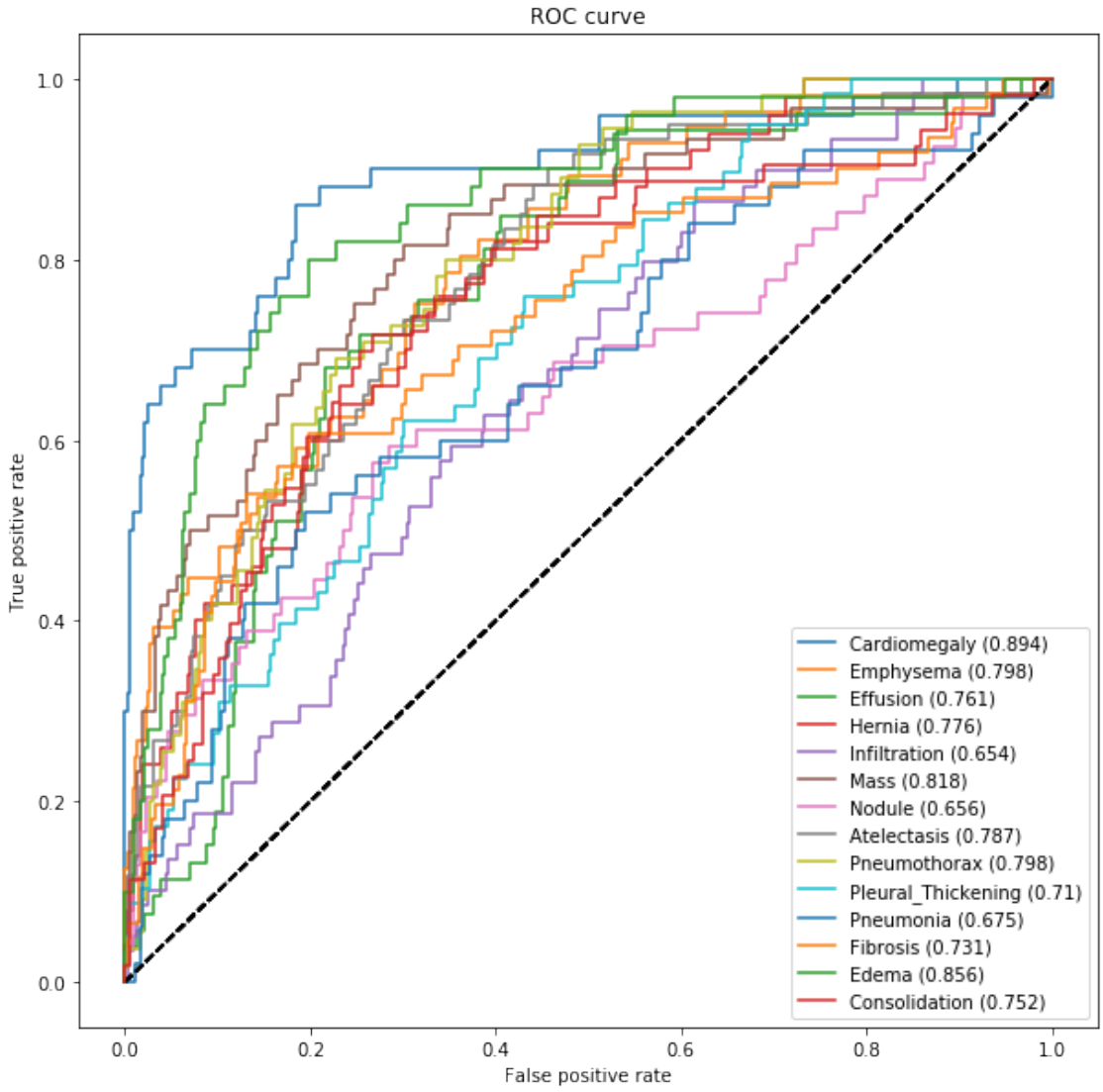
5.1 ROC Curve and AUROC We'll cover topic of model evaluation in much more detail in later weeks, but for now we'll walk through computing a metric called the AUC (Area Under the Curve) from the ROC (Receiver Operating Characteristic) curve. This is also referred to as the AUROC value, but you will see all three terms in reference to the technique, and often used almost interchangeably.

For now, what you need to know in order to interpret the plot is that a curve that is more to the left and the top has more "area" under it, and indicates that the model is performing better.

We will use the `util.get_roc_curve()` function which has been provided for you in `util.py`. Look through this function and note the use of the `sklearn` library functions to generate the ROC curves and AUROC values for our model.

- roc_curve
- roc_auc_score

```
In [23]: auc_rocs = util.get_roc_curve(labels, predicted_vals, test_generator)
```



You can compare the performance to the AUCs reported in the original ChexNeXt paper in the table below:

For reference, here's the AUC figure from the ChexNeXt paper which includes AUC values for their model as well as radiologists on this dataset:

This method does take advantage of a few other tricks such as self-training and ensembling as well, which can give a significant boost to the performance.

For details about the best performing methods and their performance on this dataset, we encourage you to read the following papers: - CheXNet - CheXpert - ChexNeXt

5.2 Visualizing Learning with GradCAM

One of the challenges of using deep learning in medicine is that the complex architecture used for neural networks makes them much harder to interpret compared to traditional machine learning models (e.g. linear models).

One of the most common approaches aimed at increasing the interpretability of models for computer vision tasks is to use Class Activation Maps (CAM). - Class activation maps are useful

for understanding where the model is "looking" when classifying an image.

In this section we will use a GradCAM's technique to produce a heatmap highlighting the important regions in the image for predicting the pathological condition. - This is done by extracting the gradients of each predicted class, flowing into our model's final convolutional layer. Look at the `util.compute_gradcam` which has been provided for you in `util.py` to see how this is done with the Keras framework.

It is worth mentioning that GradCAM does not provide a full explanation of the reasoning for each classification probability. - However, it is still a useful tool for "debugging" our model and augmenting our prediction so that an expert could validate that a prediction is indeed due to the model focusing on the right regions of the image.

First we will load the small training set and setup to look at the 4 classes with the highest performing AUC measures.

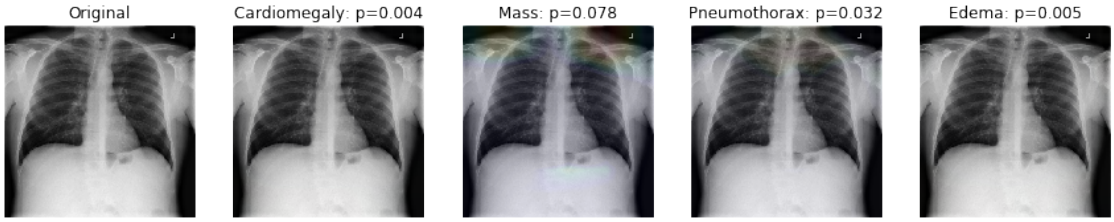
```
In [24]: df = pd.read_csv("nih/train-small.csv")
         IMAGE_DIR = "nih/images-small/"

         # only show the labels with top 4 AUC
         labels_to_show = np.take(labels, np.argsort(auc_rocs)[::-1])[:4]
```

Now let's look at a few specific images.

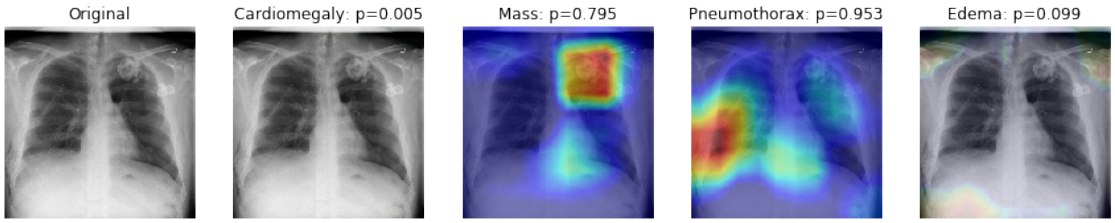
```
In [25]: util.compute_gradcam(model, '00008270_015.png', IMAGE_DIR, df, labels, labels_to_show)
```

```
Loading original image
Generating gradcam for class Cardiomegaly
Generating gradcam for class Mass
Generating gradcam for class Pneumothorax
Generating gradcam for class Edema
```



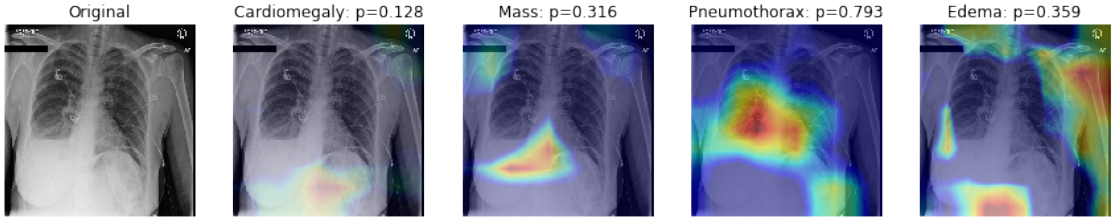

```
In [26]: util.compute_gradcam(model, '00011355_002.png', IMAGE_DIR, df, labels, labels_to_show)
```

```
Loading original image
Generating gradcam for class Cardiomegaly
Generating gradcam for class Mass
Generating gradcam for class Pneumothorax
Generating gradcam for class Edema
```



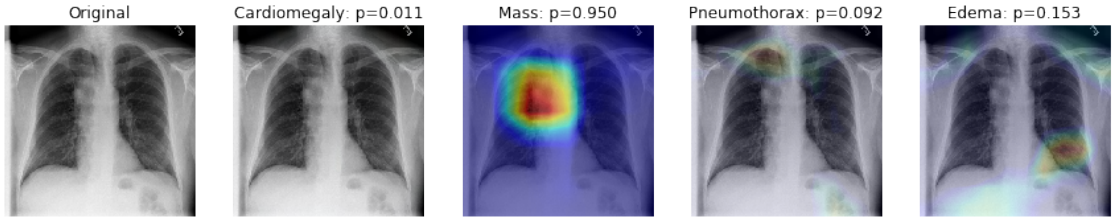

```
In [27]: util.compute_gradcam(model, '00029855_001.png', IMAGE_DIR, df, labels, labels_to_show)
```

```
Loading original image
Generating gradcam for class Cardiomegaly
Generating gradcam for class Mass
Generating gradcam for class Pneumothorax
Generating gradcam for class Edema
```




```
In [28]: util.compute_gradcam(model, '00005410_000.png', IMAGE_DIR, df, labels, labels_to_show)
```

```
Loading original image
Generating gradcam for class Cardiomegaly
Generating gradcam for class Mass
Generating gradcam for class Pneumothorax
Generating gradcam for class Edema
```



Congratulations, you've completed the first assignment of course one! You've learned how to preprocess data, check for data leakage, train a pre-trained model, and evaluate using the AUC. Great work!