Chest X-Ray Medical Diagnosis with Deep Learning

Bem-vindo à aplicação de IA para diagnóstico médico!

Nesta aplicação, você explorará o diagnóstico de imagens médicas construindo um classificador de raios-X de tórax de última geração usando o Keras.

A tarefa percorrerá algumas das etapas de construção e avaliação desse modelo de classificador de aprendizado profundo. Em particular:

- Pré-processe e preparo de um conjunto de dados de raios X do mundo real.
- Aprendizado de transferência para treinar novamente um modelo DenseNet para classificação de imagens de raios-X.
- Técnica para lidar com o desequilíbrio de classe
- Mensurar o desempenho do diagnóstico calculando a AUC (área sob a curva) para a curva ROC (característica operacional do receptor).
- Visualize a atividade do modelo usando GradCAMs.

Tópicos:

- Preparação de dados
  - Preparação de dados.
  - Prevenção de vazamento de dados.
- Desenvolvimento de modelo
  - Lidando com o desequilíbrio de classe.
  - Aproveitando modelos pré-treinados usando o aprendizado de transferência.
- Avaliação
  - Curvas AUC e ROC.

1. Importar pacotes e funções

Faremos uso dos seguintes pacotes:

- `numpy` e `pandas` é o que usaremos para manipular nossos dados
- `matplotlib.pyplot` e `seaborn` será usado para produzir gráficos para visualização
- `util` fornecerá as funções de utilidade definidas localmente que foram fornecidas para esta atribuição

Também usaremos vários módulos do framework `keras` para construir modelos de aprendizado profundo.

Execute a próxima célula para importar todos os pacotes necessários.

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
from public_tests import *
from test_utils import *

import tensorflow as tf
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)
```

```
Using TensorFlow backend.
```

2. Load the Datasets

Para esta tarefa, usaremos oChestX-ray8 dataset que contém 108.948 imagens de raios X de visão frontal de 32.717 pacientes únicos.

- Cada imagem no conjunto de dados contém vários rótulos extraídos de texto que identificam 14 condições patológicas diferentes.
- Estes, por sua vez, podem ser usados pelos médicos para diagnosticar 8 doenças diferentes.
- Usaremos esses dados para desenvolver um único modelo que fornecerá previsões de classificação binária para cada uma das 14 patologias rotuladas.
- Em outras palavras, irá prever 'positivo' ou 'negativo' para cada uma das patologias.

Você pode baixar todo o conjunto de dados gratuitamente here.

- Fornecemos um subconjunto de ~1000 imagens das imagens para você.
- Estes podem ser acessados no caminho da pasta armazenado na variável `IMAGE_DIR`.

O conjunto de dados inclui um arquivo CSV que fornece os rótulos para cada raio-X.

Para tornar seu trabalho um pouco mais fácil, processamos os rótulos de nossa pequena amostra e geramos três novos arquivos para você começar. Esses três arquivos são:

1. `nih/train-small.csv`: 875 imagens do nosso conjunto de dados a serem usadas para treinamento.
2. `nih/valid-small.csv`: 109 imagens do nosso conjunto de dados a serem usadas para validação.
3. `nih/test.csv`: 420 imagens do nosso conjunto de dados a serem usadas para teste.

Este conjunto de dados foi anotado por consenso entre quatro radiologistas diferentes para 5 de nossas 14 patologias:

- `Consolidação`
- `Edema`
- `Efusão`

- `Cardiomegalia`
- `Atelectasia`

Barra lateral sobre o significado de 'classe'

Vale a pena notar que a palavra **'classe'** é usada de várias maneiras nessas discussões.

- Às vezes nos referimos a cada uma das 14 condições patológicas rotuladas em nosso conjunto de dados como uma classe.
- Mas, para cada uma dessas patologias, estamos tentando prever se uma determinada condição está presente (ou seja, resultado positivo) ou ausente (ou seja, resultado negativo).
  - Esses dois rótulos possíveis de 'positivo' ou 'negativo' (ou o equivalente numérico de 1 ou 0) também são normalmente referidos como classes.
- Além disso, também usamos o termo em referência a 'classes' de código de software, como `ImageDataGenerator`.

Desde que você esteja ciente de tudo isso, porém, não deve causar nenhuma confusão, pois o termo 'classe' geralmente é claro no contexto em que é usado.

2.1 Loading the Data

Let's open these files using the pandas library

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

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

train_df.head()
```

Out[2]:

	Image	Atelectasis	Cardiomegaly	Consolidation	Edema	Effusion	Emphysema	Fibr
0	00008270_015.png	0	0	0	0	0	0	
1	00029855_001.png	1	0	0	0	1	0	
2	00001297_000.png	0	0	0	0	0	0	
3	00012359_002.png	0	0	0	0	0	0	
4	00017951_001.png	0	0	0	0	0	0	

In [3]:
```
labels = ['Cardiomegaly',
          'Emphysema',
          'Effusion',
          'Hernia',
          'Infiltration',
          'Mass',
          'Nodule',
          'Atelectasis',
          'Pneumothorax',
          'Pleural_Thickening',
          'Pneumonia',
          'Fibrosis',
```

```
          'Edema',
          'Consolidation']
```

2.2 Prevenção de vazamento de dados

Vale a pena notar que nosso conjunto de dados contém várias imagens para cada paciente. Este pode ser o caso, por exemplo, quando um paciente fez várias imagens de raios-X em momentos diferentes durante suas visitas ao hospital. Em nossa divisão de dados, garantimos que a divisão seja feita no nível do paciente, para que não haja "vazamento" de dados entre os conjuntos de dados de treinamento, validação e teste.

Exercício 1 - Verifique se há vazamento

Na célula abaixo, escreva uma função para verificar se há vazamento entre dois conjuntos de dados. Usaremos isso para garantir que não haja pacientes no conjunto de teste que também estejam presentes nos conjuntos de treinamento ou validação.

▶ Hints

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])
    df2_patients_unique = set(df2[patient_col])

    patients_in_both_groups = df1_patients_unique.intersection(df2_patients_unique

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

        leakage = True
    else:
        leakage = False
    # boolean (true if there is at least 1 patient in both groups)

    ### END CODE HERE ###

    return leakage
```

In [5]:
```
### do not edit this code cell
check_for_leakage_test(check_for_leakage)
```

```
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

 All tests passed.
```

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
```

```
All tests passed.
```

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 valid: {}".format(check_for_leakage(train_df, val
print("leakage between train and test: {}".format(check_for_leakage(train_df, test_
print("leakage between valid and test: {}".format(check_for_leakage(valid_df, test_
```

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

Expected output

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

2.3 Preparando Imagens

Com nossas divisões de conjunto de dados prontas, agora podemos prosseguir com a configuração de nosso modelo para consumi-las.

- Para isso usaremos a classe ImageDataGenerator pronta para uso do framework Keras, que nos permite construir um "gerador" para imagens especificadas em um dataframe .
- Esta classe também fornece suporte para aumento básico de dados, como inversão horizontal aleatória de imagens.
- Também usamos o gerador para transformar os valores em cada lote para que sua média seja 0 e seu desvio padrão seja 1.
  - Isso facilitará o treinamento do modelo padronizando a distribuição de entrada.
- O gerador também converte nossas imagens de raios X de canal único (escala de cinza) em um formato de três canais, repetindo os valores da imagem em todos os canais.
  - Queremos isso porque o modelo pré-treinado que usaremos requer entradas de três canais.

Como é principalmente uma questão de ler e entender a documentação do Keras, implementamos o gerador para você. Há algumas coisas a serem observadas:

1. Normalizamos a média e o desvio padrão dos dados
2. Embaralhamos a entrada após cada época.
3. Definimos o tamanho da imagem como 320px por 320pxWith our dataset splits ready, we can now proceed with setting up our model to consume them.

In [7]:
```
def get_train_generator(df, image_dir, x_col, y_cols, shuffle=True, batch_size=8, 
    """
    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
```

Crie um gerador separado para conjuntos válidos e de teste

Agora precisamos construir um novo gerador para validação e teste de dados.

Por que não podemos usar o mesmo gerador para os dados de treinamento?

Olhe novamente para o gerador que escrevemos para os dados de treinamento.

- Normaliza cada imagem **por lote**, ou seja, usa estatísticas de lote.
- Não devemos fazer isso com os dados de teste e validação, pois em um cenário da vida real não processamos as imagens recebidas um lote por vez (processamos uma imagem por vez).
- Conhecer a média por lote de dados de teste efetivamente daria uma vantagem ao nosso modelo.
    - O modelo não deve ter nenhuma informação sobre os dados de teste.

O que precisamos fazer é normalizar os dados de teste recebidos usando as estatísticas **computadas do conjunto de treinamento**.

- Implementamos isso na função abaixo.
- Existe uma nota técnica. Idealmente, gostaríamos de calcular nossa média amostral e desvio padrão usando todo o conjunto de treinamento.
- No entanto, como isso é extremamente grande, isso consumiria muito tempo.
- No interesse do tempo, pegaremos uma amostra aleatória do conjunto de dados e calcularemos a média da amostra e o desvio padrão da amostra.

In [8]:
```
def get_test_and_valid_generator(valid_df, test_df, train_df, image_dir, x_col, y_
    """
```

```
    Return generator for validation set and 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
    """
    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
```

Com nossa função de gerador pronta, vamos criar um gerador para nossos dados de treinamento e um para cada um de nossos conjuntos de dados de teste e validação.

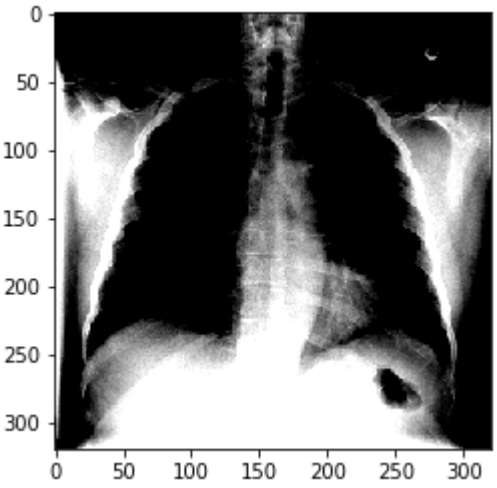
In [10]:
```
IMAGE_DIR = "data/nih/images-small/"
train_generator = get_train_generator(train_df, IMAGE_DIR, "Image", labels)
valid_generator, test_generator= get_test_and_valid_generator(valid_df, test_df, t
```

```
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.
```

Vamos dar uma olhada no que o gerador fornece ao nosso modelo durante o treinamento e a validação. Podemos fazer isso chamando a função `__get_item__(index)` :

In [13]:
```
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] for integers).
```



3. Desenvolvimento de modelo

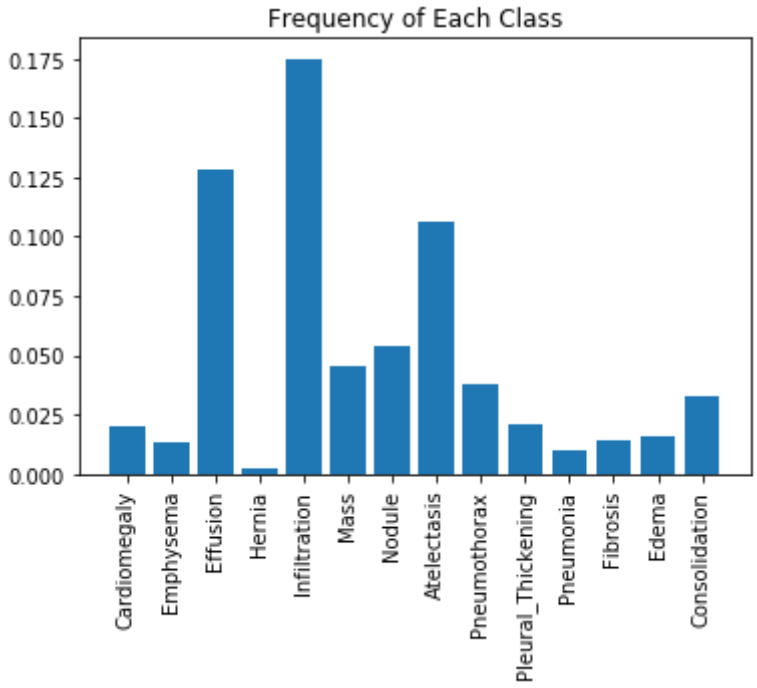
Agora passaremos para o treinamento e desenvolvimento do modelo. No entanto, temos alguns desafios práticos para lidar antes de realmente treinar uma rede neural. O primeiro é o desequilíbrio de classe.

3.1 Lidando com o Desequilíbrio de Classe

Um dos desafios de trabalhar com conjuntos de dados de diagnóstico médico é o grande desequilíbrio de classes presente nesses conjuntos de dados. Vamos plotar a frequência de cada um dos rótulos em nosso conjunto de dados:

In [32]:
```
plt.xticks(rotation=90)
plt.bar(x=labels, height=np.mean(train_generator.labels, axis=0))
```

```
plt.title("Frequency of Each Class")
plt.show()
```



Podemos ver neste gráfico que a prevalência de casos positivos varia significativamente entre as diferentes patologias. (Essas tendências também refletem as do conjunto de dados completo.)

- A patologia 'Hérnia' tem o maior desequilíbrio com a proporção de casos de treinamento positivo sendo de cerca de 0,2%.
- Mas mesmo a patologia 'Infiltração', que tem a menor quantidade de desequilíbrio, tem apenas 17,5% dos casos de treinamento rotulados como positivos.

Idealmente, treinaríamos nosso modelo usando um conjunto de dados uniformemente balanceado para que os casos de treinamento positivo e negativo contribuíssem igualmente para a perda.

Se usarmos uma função de perda de entropia cruzada normal com um conjunto de dados altamente desbalanceado, como estamos vendo aqui, o algoritmo será incentivado a priorizar a classe majoritária (ou seja, negativa em nosso caso), pois contribui mais para a perda.

Impacto do desequilíbrio de classe na função de perda

Vamos dar uma olhada nisso. Suponha que teríamos usado uma perda de entropia cruzada normal para cada patologia. Lembramos que a contribuição da perda de entropia cruzada do caso de dados de treinamento i^{th} é:

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

onde x_i e y_i são os recursos de entrada e o rótulo, e $f(x_i)$ é a saída do modelo, ou seja, a probabilidade de ser positivo.

Observe que, para qualquer caso de treinamento, $y_i = 0$ ou então $(1 - y_i) = 0$, portanto, apenas um desses termos contribui para a perda (o outro termo é multiplicado por zero e

se torna zero).

Podemos reescrever a média geral da perda de entropia cruzada em todo o conjunto de treinamento $\mathcal{D}$ de tamanho N como segue:

$$\mathcal{L}_{entropiacruzada}(\mathcal{D}) = -\frac{1}{N}\Big(\sum_{\text{exemplos positivos}} \log(f(x_i)) + \sum_{\text{exemplos negativos}} \log(1-f(x_i))\Big).$$

Usando essa formulação, podemos ver que se houver um grande desequilíbrio com pouquíssimos casos de treinamento positivo, por exemplo, então a perda será dominada pela classe negativa. Somando a contribuição de todos os casos de treinamento para cada classe (ou seja, condição patológica), vemos que a contribuição de cada classe (ou seja, positiva ou negativa) é:

$$freq_p = \frac{\text{número de exemplos positivos}}{N}$$

$$\textoe$$

$$freq_n = \frac{\text{número de exemplos negativos}}{N}.$$

Exercício 2 - Calcular frequências de classe

Complete a função abaixo para calcular as frequências para cada rótulo em nosso conjunto de dados.

▶ Hints

In [14]:
```
# 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) / N
    negative_frequencies = 1 - positive_frequencies

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

In [15]:
```
### do not edit this code cell
compute_class_freqs_test(compute_class_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]

 All tests passed.
```

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]

     All tests passed.
```

Agora vamos calcular as frequências para nossos dados de treinamento.

In [16]:
```
freq_pos, freq_neg = compute_class_freqs(train_generator.labels)
freq_pos
```

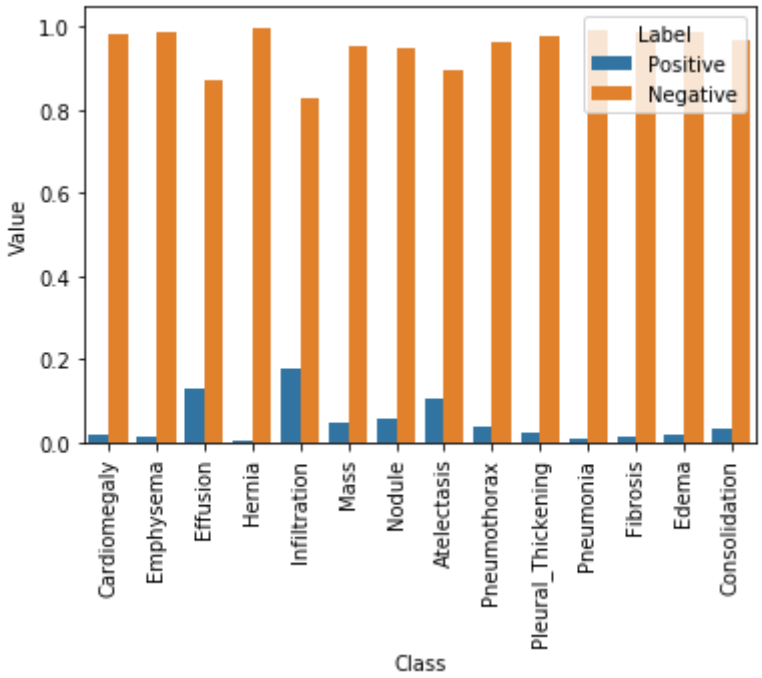
Out[16]:
```
array([0.02 , 0.013, 0.128, 0.002, 0.175, 0.045, 0.054, 0.106, 0.038,
       0.021, 0.01 , 0.014, 0.016, 0.033])
```

Expected output

```
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])
```

Vamos visualizar essas duas taxas de contribuição uma ao lado da outra para cada uma das patologias:

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



Como vemos no gráfico acima, a contribuição dos casos positivos é significativamente menor do que a dos casos negativos. No entanto, queremos que as contribuições sejam iguais. Uma maneira de fazer isso é multiplicar cada exemplo de cada classe por um fator de peso específico da classe, w_{pos} e w_{neg}, para que a contribuição geral de cada classe seja a mesma.

Para ter isso, queremos

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

o que podemos fazer simplesmente tomando

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

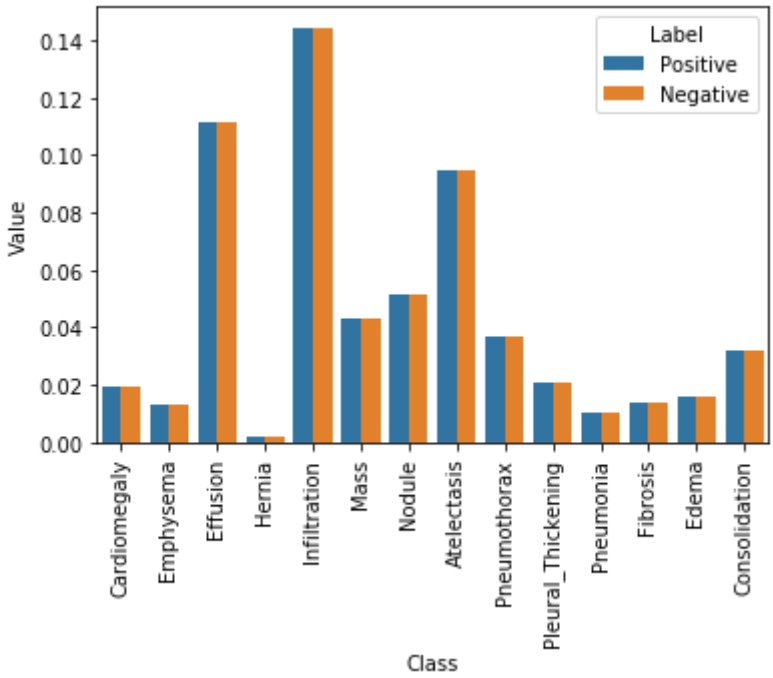
$$w_{neg} = freq_{pos}$$

Dessa forma, estaremos equilibrando a contribuição dos rótulos positivos e negativos.

In [18]:
```
pos_weights = freq_neg
neg_weights = freq_pos
pos_contribution = freq_pos * pos_weights
neg_contribution = freq_neg * neg_weights
```

Vamos verificar isso representando graficamente as duas contribuições uma ao lado da outra novamente:

In [19]:
```
data = pd.DataFrame({"Class": labels, "Label": "Positive", "Value": pos_contributi
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);
```



Como mostra a figura acima, ao aplicar essas ponderações, os rótulos positivos e negativos dentro de cada classe teriam a mesma contribuição agregada para a função de perda. Agora vamos implementar essa função de perda.

Depois de calcular os pesos, nossa perda ponderada final para cada caso de treinamento será

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

Exercício 3 - Obtenha Perda de Peso

Preencha a função `weighted_loss` abaixo para retornar uma função de perda que calcula a perda ponderada para cada lote. Lembre-se de que, para a perda multiclasse, somamos a perda média para cada classe individual. Observe que também queremos adicionar um pequeno valor, ϵ, aos valores previstos antes de obter seus logs. Isso é simplesmente para evitar um erro numérico que ocorreria caso o valor previsto fosse zero.

Observação

Use as funções de Keras para calcular a média e o log.

- Keras.mean
- Keras.log

In [22]:
```
# 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_c
      neg_weights (np.array): array of negative weights for each class, size (num_c

    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_clas
            y_pred (Tensor): Tensor of predicted labels, size is (num_examples, nur
        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] + eps
                             + neg_weights[i]* (1 - y_true[:,i]) * K.log( 1 - y_pr
        return loss

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

In [23]:

```
# test with a large epsilon in order to catch errors.
# In order to pass the tests, set epsilon = 1
epsilon = 1

### do not edit anything below
sess = K.get_session()
get_weighted_loss_test(get_weighted_loss, epsilon, sess)
```

```
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 you weighted them correctly, you'd expect the two losses to be the same.
With epsilon = 1, your losses should be, L(y_pred_1) = -0.4956203 and L(y_pred_2)
= -0.4956203

Your outputs:

L(y_pred_1) =  -0.4956203
L(y_pred_2) =  -0.4956203
Difference: L(y_pred_1) - L(y_pred_2) =  0.0

 All tests passed.
```

Expected output

with epsilon = 1

```
 Your outputs:

 L(y_pred_1) =  -0.4956203
 L(y_pred_2) =  -0.4956203
 Difference: L(y_pred_1) - L(y_pred_2) =  0.0

     All tests passed.
```

If you are missing something in your implementation, you will see a different set of losses for `L(y_pred_1)` and `L(y_pred_2)` (even though `L(y_pred_1)` and `L(y_pred_2)` will be the same).

3.2 DenseNet121

Em seguida, usaremos um modelo DenseNet121 pré-treinado que podemos carregar diretamente do Keras e adicionar duas camadas sobre ele:

1. Uma camada `GlobalAveragePooling2D` para obter a média das últimas camadas de convolução de DenseNet121.
2. Uma camada `Dense` com ativação `sigmoid` para obter os logits de predição para cada uma de nossas classes.

Podemos definir nossa função de perda personalizada para o modelo especificando o parâmetro `loss` na função `compile()`.

In [24]:
```
# create the base pre-trained model
base_model = DenseNet121(weights='models/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))
```

4. Treinamento (opcional)

Com nosso modelo pronto para treinamento, usaremos a função `model.fit()` no Keras para treinar nosso modelo.

- Estamos treinando em um pequeno subconjunto do conjunto de dados (~1%).
- Portanto, o que nos preocupa neste momento é garantir que a perda no conjunto de treinamento esteja diminuindo.

Como o treinamento pode levar um tempo considerável, para fins pedagógicos, optamos por não treinar o modelo aqui, mas sim carregar um conjunto de pesos pré-treinados na próxima seção. No entanto, você pode usar o código mostrado abaixo para praticar o treinamento do modelo localmente em sua máquina ou no Colab.

NOTA: Não execute o código abaixo na plataforma Coursera, pois ele excederá as limitações de memória da plataforma.

Código Python para treinar o modelo:

```
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 Treinamento no conjunto de dados maior

Dado que o conjunto de dados original tem mais de 40 GB de tamanho e o processo de treinamento no conjunto de dados completo leva algumas horas, treinamos o modelo em uma máquina equipada com GPU para você e fornecemos o arquivo de pesos de nosso modelo (com um tamanho de lote de 32 em vez disso) para ser usado para o restante desta atribuição.

A arquitetura do modelo para o nosso modelo pré-treinado é exatamente a mesma, mas usamos alguns "callbacks" úteis do Keras para este treinamento. Reserve um tempo para ler sobre esses callbacks quando quiser, pois eles serão muito úteis para gerenciar sessões de treinamento de longa duração:

1. Você pode usar o retorno de chamada `ModelCheckpoint` para monitorar a métrica `val_loss` do seu modelo e manter um instantâneo do seu modelo no ponto.
2. Você pode usar o `TensorBoard` para usar o utilitário Tensorflow Tensorboard para monitorar suas execuções em tempo real.
3. Você pode usar `ReduceLROnPlateau` para decair lentamente a taxa de aprendizado do seu modelo à medida que ele para de melhorar em uma métrica como `val_loss` para ajustar o modelo nas etapas finais do treinamento.
4. Você pode usar o retorno de chamada `EarlyStopping` para interromper o trabalho de treinamento quando seu modelo parar de melhorar em sua perda de validação. Você pode definir um valor de `paciência` que é o número de épocas em que o modelo não melhora após o qual o treinamento é encerrado. Esse retorno de chamada também pode restaurar convenientemente os pesos para a melhor métrica no final do treinamento para seu modelo.

Você pode ler sobre esses retornos de chamada e outros retornos de chamada úteis do Keras aqui.

Vamos carregar nossos pesos pré-treinados no modelo agora:

In [25]:
```
model.load_weights("models/nih/pretrained_model.h5")
```

5. Previsão e Avaliação

Agora que temos um modelo, vamos avaliá-lo usando nosso conjunto de teste. Podemos usar convenientemente a função `predict_generator` para gerar as previsões para as imagens em nosso conjunto de teste.

Observação: A célula a seguir pode levar cerca de 4 minutos para ser executada.

In [26]:
```
predicted_vals = model.predict_generator(test_generator, steps = len(test_generato
```

5.1 Curva ROC e AUROC

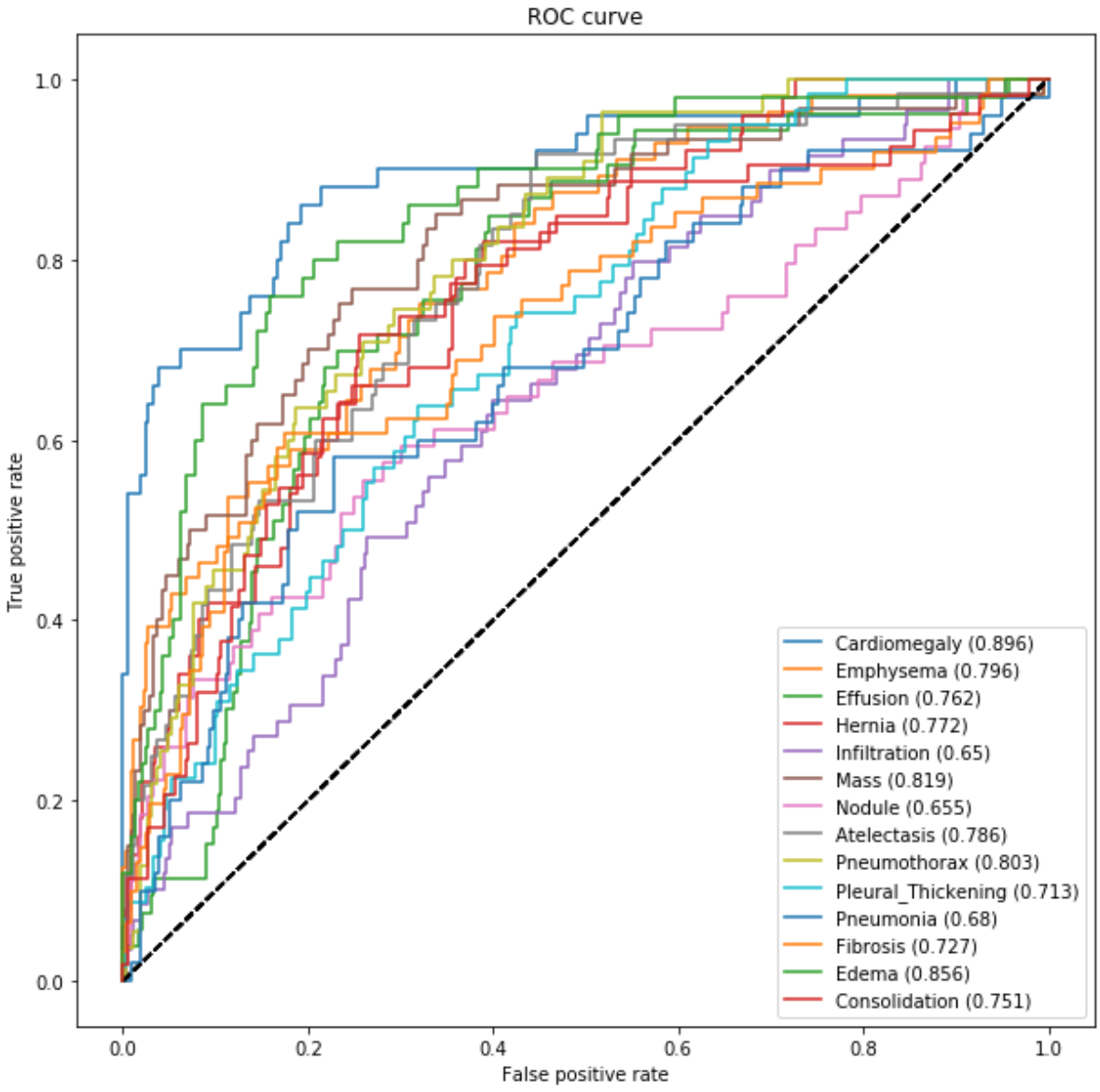
Abordaremos o tópico de avaliação do modelo com muito mais detalhes nas próximas semanas, mas, por enquanto, analisaremos a computação de uma métrica chamada AUC (Area Under the Curve) do ROC (Receiver Operating Characteristic) curva. Isso também é conhecido como o valor AUROC, mas você verá todos os três termos em referência à técnica e geralmente usados quase de forma intercambiável.

Por enquanto, o que você precisa saber para interpretar o gráfico é que uma curva mais à esquerda e no topo tem mais "área" abaixo dela e indica que o modelo está tendo um desempenho melhor.

Usaremos a função `util.get_roc_curve()` que foi fornecida para você em `util.py`. Observe esta função e observe o uso das funções da biblioteca `sklearn` para gerar as curvas ROC e os valores AUROC para nosso modelo.

- roc_curve
- roc_auc_score

In [27]:
```
auc_rocs = util.get_roc_curve(labels, predicted_vals, test_generator)
```



Você pode comparar o desempenho com as AUCs relatadas no documento ChexNeXt original na tabela abaixo:

Para referência, aqui está a figura AUC do artigo ChexNeXt, que inclui valores AUC para seu modelo, bem como radiologistas neste conjunto de dados:

Pathology	Radiologists (95% CI)	Algorithm (95% CI)	Algorithm – Radiologists Difference (99.6% CI)[a]	Advantage
Atelectasis	0.808 (0.777 to 0.838)	0.862 (0.825 to 0.895)	0.053 (0.003 to 0.101)	Algorithm
Cardiomegaly	0.888 (0.863 to 0.910)	0.831 (0.790 to 0.870)	−0.057 (−0.113 to −0.007)	Radiologists
Consolidation	0.841 (0.815 to 0.870)	0.893 (0.859 to 0.924)	0.052 (−0.001 to 0.101)	No difference
Edema	0.910 (0.886 to 0.930)	0.924 (0.886 to 0.955)	0.015 (−0.038 to 0.060)	No difference
Effusion	0.900 (0.876 to 0.921)	0.901 (0.868 to 0.930)	0.000 (−0.042 to 0.040)	No difference
Emphysema	0.911 (0.866 to 0.947)	0.704 (0.567 to 0.833)	−0.208 (−0.508 to −0.003)	Radiologists
Fibrosis	0.897 (0.840 to 0.936)	0.806 (0.719 to 0.884)	−0.091 (−0.198 to 0.016)	No difference
Hernia	0.985 (0.974 to 0.991)	0.851 (0.785 to 0.909)	−0.133 (−0.236 to −0.055)	Radiologists
Infiltration	0.734 (0.688 to 0.779)	0.721 (0.651 to 0.786)	−0.013 (−0.107 to 0.067)	No difference
Mass	0.886 (0.856 to 0.913)	0.909 (0.864 to 0.948)	0.024 (−0.041 to 0.080)	No difference
Nodule	0.899 (0.869 to 0.924)	0.894 (0.853 to 0.930)	−0.005 (−0.058 to 0.044)	No difference
Pleural thickening	0.779 (0.740 to 0.809)	0.798 (0.744 to 0.849)	0.019 (−0.056 to 0.094)	No difference
Pneumonia	0.823 (0.779 to 0.856)	0.851 (0.781 to 0.911)	0.028 (−0.087 to 0.125)	No difference
Pneumothorax	0.940 (0.912 to 0.962)	0.944 (0.915 to 0.969)	0.004 (−0.040 to 0.051)	No difference

[a]The AUC difference was calculated as the AUC of the algorithm minus the AUC of the radiologists. To account for multiple hypothesis testing, the Bonferroni-corrected CI (1 − 0.05/14; 99.6%) around the difference was computed.
The nonparametric bootstrap was used to estimate the variability around each of the performance measures; 10,000 bootstrap replicates from the validation set were drawn, and each performance measure was calculated for the algorithm and the radiologists on these same 10,000 bootstrap replicates. This produced a distribution for each estimate, and the 95% bootstrap percentile intervals (2.5th and 97.5th percentiles) are reported.
Abbreviations: AUC, area under the receiver operating characteristic curve; CI, confidence interval.

https://doi.org/10.1371/journal.pmed.1002686.t001

Esse método tira proveito de alguns outros truques, como autotreinamento e montagem, que podem dar um impulso significativo ao desempenho.

Para obter detalhes sobre os métodos de melhor desempenho e seu desempenho neste conjunto de dados, recomendamos a leitura dos seguintes documentos:

- CheXNet
- CheXpert
- ChexNeXt

5.2 Visualizando o aprendizado com o GradCAM

Um dos desafios do uso de aprendizado profundo na medicina é que a arquitetura complexa usada para redes neurais torna-as muito mais difíceis de interpretar em comparação com os modelos tradicionais de aprendizado de máquina (por exemplo, modelos lineares).

Uma das abordagens mais comuns destinadas a aumentar a interpretabilidade de modelos para tarefas de visão computacional é usar Mapas de Ativação de Classe (CAM).

- Os mapas de ativação de classe são úteis para entender onde o modelo está "olhando" ao classificar uma imagem.

Nesta seção, usaremos uma técnica GradCAM para produzir um mapa de calor destacando as regiões importantes na imagem para prever a condição patológica.

- Isso é feito extraindo os gradientes de cada classe prevista, fluindo para a camada convolucional final do nosso modelo. Veja `util.compute_gradcam` que foi fornecido para você em `util.py` para ver como isso é feito com a estrutura Keras.

Vale ressaltar que o GradCAM não fornece uma explicação completa do raciocínio para cada probabilidade de classificação.

- No entanto, ainda é uma ferramenta útil para "depurar" nosso modelo e aumentar nossa previsão para que um especialista possa validar que uma previsão é de fato devido ao modelo focar nas regiões corretas da imagem.

Primeiro, carregaremos o pequeno conjunto de treinamento e configuraremos para observar as 4 classes com as medidas AUC de maior desempenho.

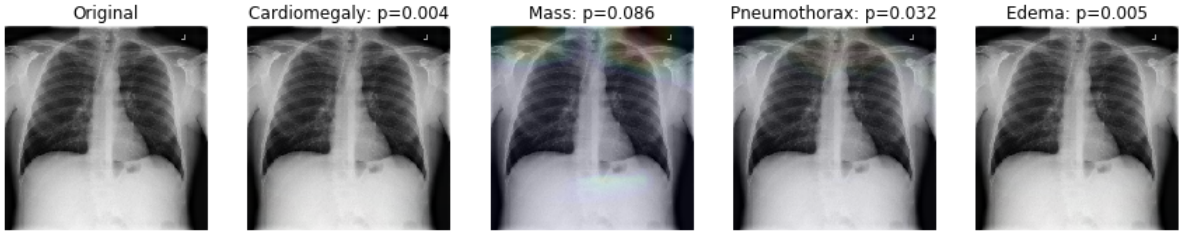
In [28]:
```
df = pd.read_csv("data/nih/train-small.csv")
IMAGE_DIR = "data/nih/images-small/"

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

Agora vamos ver algumas imagens específicas.

In [35]:
```
util.compute_gradcam(model, '00008270_015.png', IMAGE_DIR, df, labels, labels_to_sl
```

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

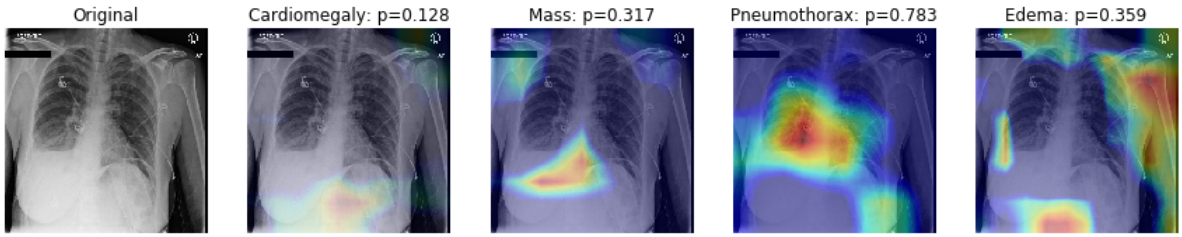


In [30]:
```
util.compute_gradcam(model, '00011355_002.png', IMAGE_DIR, df, labels, labels_to_sl
```

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

In [31]:
```
util.compute_gradcam(model, '00029855_001.png', IMAGE_DIR, df, labels, labels_to_sl
```

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



In [32]:
```
util.compute_gradcam(model, '00005410_000.png', IMAGE_DIR, df, labels, labels_to_sl
```

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