Pneumonia is a very common disease. It can be either: 1) Bacterial pneumonia 2) Viral Pneumonia 3) Mycoplasma pneumonia and 4) Fungal pneumonia. This dataset consists pneumonia samples belonging to the first two classes. The dataset consists of only very few samples and that too unbalanced. The aim of this kernel is to develop a robust deep learning model from scratch on this limited amount of data. We all know that deep learning models are data hungry but if you know how things work, you can build good models even with a limited amount of data.

In [1]:

linkcode

```
# This Python 3 environment comes with many helpful analytics libraries install
ed
# It is defined by the kaggle/python docker image: https://github.com/kaggle/do
cker-python
# For example, here's several helpful packages to load in

import os
import glob
import h5py
import shutil
import imgaug as aug
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.image as mimg
import imgaug.augmenters as iaa
from os import listdir, makedirs, getcwd, remove
from os.path import isfile, join, abspath, exists, isdir, expanduser
from PIL import Image
from pathlib import Path
from skimage.io import imread
from skimage.transform import resize
from keras.models import Sequential, Model
from keras.applications.vgg16 import VGG16, preprocess_input
from keras.preprocessing.image import ImageDataGenerator,load_img, img_to_arr
ay
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Dropout, Input, Flatten
, SeparableConv2D
from keras.layers import GlobalMaxPooling2D
from keras.layers.normalization import BatchNormalization
from keras.layers.merge import Concatenate
from keras.models import Model
from keras.optimizers import Adam, SGD, RMSprop
from keras.callbacks import ModelCheckpoint, Callback, EarlyStopping
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from mlxtend.plotting import plot_confusion_matrix
```

```
from sklearn.metrics import confusion_matrix
import cv2
from keras import backend as K
color = sns.color_palette()
%matplotlib inline

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list
the files in the input directory
print(os.listdir("../input"))

# Any results you write to the current directory are saved as output.
```

```
/opt/conda/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning:
Conversion of the second argument of issubdtype from `float` to `np.floati
ng` is deprecated. In future, it will be treated as `np.float64 == np.dtyp
e(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
['chest-xray-pneumonia', 'vgg16', 'xray-best-model']
```

Reproducibility is a great concern when doing deep learning. There was a good discussion on KaggleNoobs slack regarding this. We will set a numer of things in order to make sure that the results are almost reproducible(if not fully).

In [2]:

```
import tensorflow as tf

# Set the seed for hash based operations in python
os.environ['PYTHONHASHSEED'] = '0'

# Set the numpy seed
np.random.seed(111)

# Disable multi-threading in tensorflow ops
session_conf = tf.ConfigProto(intra_op_parallelism_threads=1, inter_op_parall
elism_threads=1)

# Set the random seed in tensorflow at graph level
tf.set_random_seed(111)

# Define a tensorflow session with above session configs
sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)

# Set the session in keras
K.set_session(sess)

# Make the augmentation sequence deterministic
aug.seed(111)
```

The dataset is divided into three sets: 1) train set 2) validation set and 3) test set. Let's grab the dataset

In [3]:

```
# Define path to the data directory
data_dir = Path('../input/chest-xray-pneumonia/chest_xray/chest_xray')

# Path to train directory (Fancy pathlib...no more os.path!!)
train_dir = data_dir / 'train'

# Path to validation directory
val_dir = data_dir / 'val'

# Path to test directory
test_dir = data_dir / 'test'
```

We will first go through the training dataset. We will do some analysis on that, look at some of the samples, check the number of samples for each class, etc. Lets' do it.

Each of the above directory contains two sub-directories:

- NORMAL: These are the samples that describe the normal (no pneumonia) case.
- PNEUMONIA: This directory contains those samples that are the pneumonia cases.

In [4]:

```
# Get the path to the normal and pneumonia sub-directories
normal_cases_dir = train_dir / 'NORMAL'
pneumonia_cases_dir = train_dir / 'PNEUMONIA'

# Get the list of all the images
normal_cases = normal_cases_dir.glob('*.jpeg')
pneumonia_cases = pneumonia_cases_dir.glob('*.jpeg')

# An empty list. We will insert the data into this list in (img_path, label) fo
rmat
train_data = []

# Go through all the normal cases. The label for these cases will be 0
for img in normal_cases:
    train_data.append((img,0))

# Go through all the pneumonia cases. The label for these cases will be 1
for img in pneumonia_cases:
    train_data.append((img, 1))

# Get a pandas dataframe from the data we have in our list
train_data = pd.DataFrame(train_data, columns=['image', 'label'],index=None)

# Shuffle the data
train_data = train_data.sample(frac=1.).reset_index(drop=True)
```

```
# How the dataframe looks like?
train_data.head()
```

Out[4]:

	image	label
0	../input/chest-xray-pneumonia/chest_xray/chest...	0
1	../input/chest-xray-pneumonia/chest_xray/chest...	0
2	../input/chest-xray-pneumonia/chest_xray/chest...	1
3	../input/chest-xray-pneumonia/chest_xray/chest...	1
4	../input/chest-xray-pneumonia/chest_xray/chest...	1

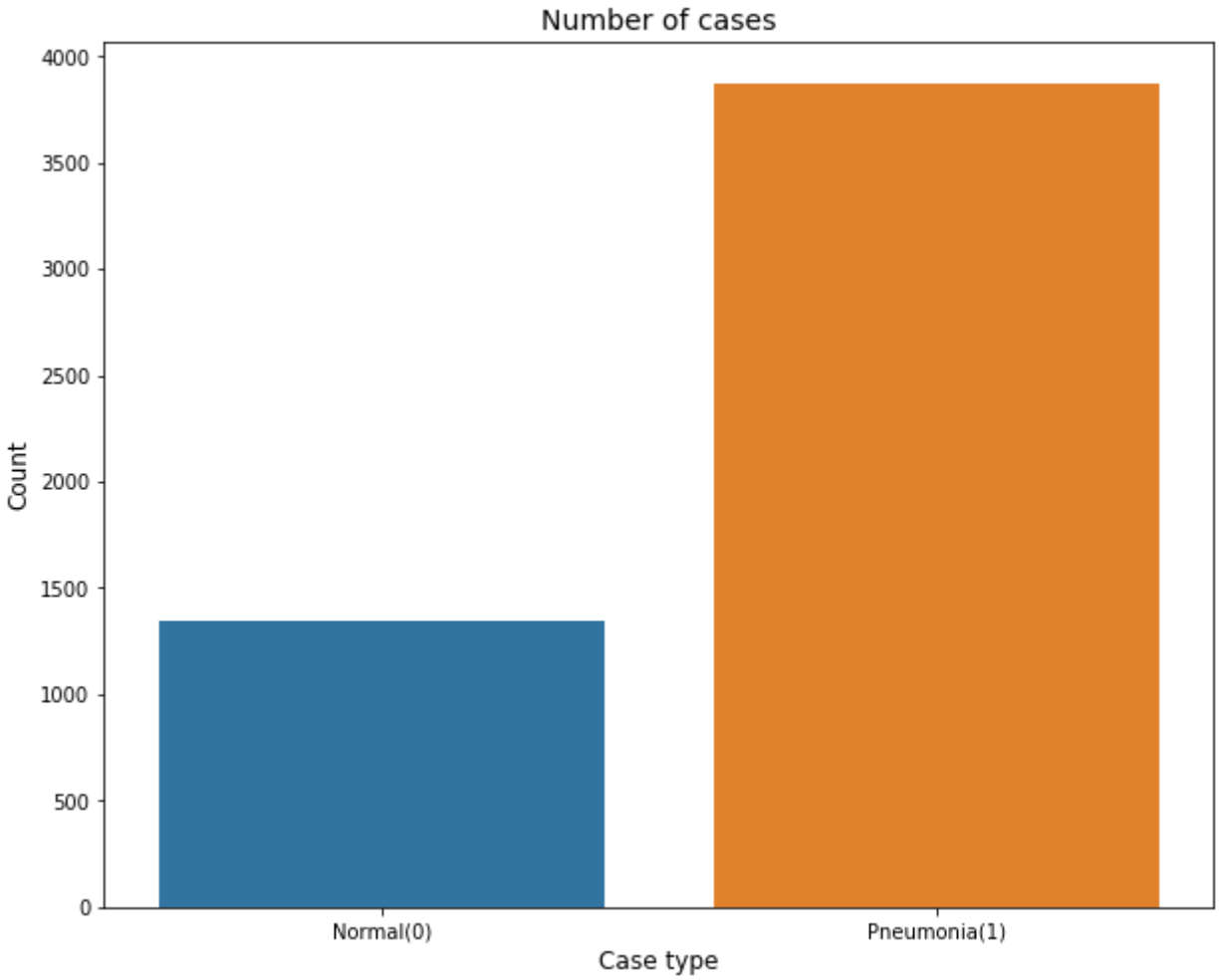
How many samples for each class are there in the dataset?

In [5]:

```
# Get the counts for each class
cases_count = train_data['label'].value_counts()
print(cases_count)

# Plot the results
plt.figure(figsize=(10,8))
sns.barplot(x=cases_count.index, y= cases_count.values)
plt.title('Number of cases', fontsize=14)
plt.xlabel('Case type', fontsize=12)
plt.ylabel('Count', fontsize=12)
plt.xticks(range(len(cases_count.index)), ['Normal(0)', 'Pneumonia(1)'])
plt.show()
```

```
1    3875
0    1341
Name: label, dtype: int64
```



As you can see the data is highly imbalanced. We have almost with thrice pneumonia cases here as compared to the normal cases. This situation is very normal when it comes to medical data. The data will always be imbalanced. either there will be too many normal cases or there will be too many cases with the disease.

Let's look at how a normal case is different from that of a pneumonia case. We will look at somes samples from our training data itself.

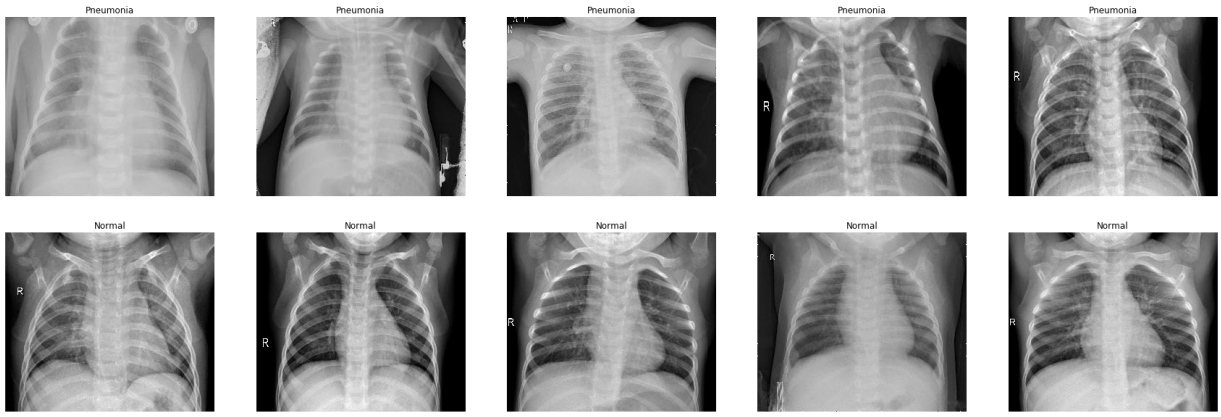
In [6]:

```
# Get few samples for both the classes
pneumonia_samples = (train_data[train_data['label']==1]['image'].iloc[:5]).to
list()
normal_samples = (train_data[train_data['label']==0]['image'].iloc[:5]).tolis
t()

# Concat the data in a single list and del the above two list
samples = pneumonia_samples + normal_samples
del pneumonia_samples, normal_samples

# Plot the data
f, ax = plt.subplots(2,5, figsize=(30,10))
```

```
for i in range(10):
    img = imread(samples[i])
    ax[i//5, i%5].imshow(img, cmap='gray')
    if i<5:
        ax[i//5, i%5].set_title("Pneumonia")
    else:
        ax[i//5, i%5].set_title("Normal")
    ax[i//5, i%5].axis('off')
    ax[i//5, i%5].set_aspect('auto')
plt.show()
```



If you look carefully, then there are some cases where you won't be able to differentiate between a normal case and a pneumonia case with the naked eye. There is one case in the above plot, at least for me ,which is too much confusing. If we can build a robust classifier, it would be a great assist to the doctor too.

Preparing validation data

We will be defining a generator for the training dataset later in the notebook but as the validation data is small, so I can read the images and can load the data without the need of a generator. This is exactly what the code block given below is doing.

In [7]:

```
# Get the path to the sub-directories
normal_cases_dir = val_dir / 'NORMAL'
pneumonia_cases_dir = val_dir / 'PNEUMONIA'

# Get the list of all the images
normal_cases = normal_cases_dir.glob('*.jpeg')
pneumonia_cases = pneumonia_cases_dir.glob('*.jpeg')

# List that are going to contain validation images data and the corresponding l
abels
valid_data = []
valid_labels = []
```

```
# Some images are in grayscale while majority of them contains 3 channels. So,
if the image is grayscale, we will convert into a image with 3 channels.
# We will normalize the pixel values and resizing all the images to 224x224

# Normal cases
for img in normal_cases:
    img = cv2.imread(str(img))
    img = cv2.resize(img, (224,224))
    if img.shape[2] ==1:
        img = np.dstack([img, img, img])
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = img.astype(np.float32)/255.
    label = to_categorical(0, num_classes=2)
    valid_data.append(img)
    valid_labels.append(label)

# Pneumonia cases
for img in pneumonia_cases:
    img = cv2.imread(str(img))
    img = cv2.resize(img, (224,224))
    if img.shape[2] ==1:
        img = np.dstack([img, img, img])
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = img.astype(np.float32)/255.
    label = to_categorical(1, num_classes=2)
    valid_data.append(img)
    valid_labels.append(label)

# Convert the list into numpy arrays
valid_data = np.array(valid_data)
valid_labels = np.array(valid_labels)

print("Total number of validation examples: ", valid_data.shape)
print("Total number of labels:", valid_labels.shape)
```

```
Total number of validation examples:  (16, 224, 224, 3)
Total number of labels: (16, 2)
```

Augmentation

Data augmentation is a powerful technique which helps in almost every case for improving the robustness of a model. But augmentation can be much more helpful where the dataset is imbalanced. You can generate different samples of undersampled class in order to try to balance the overall distribution.

I like imgaug a lot. It comes with a very clean api and you can do hell of augmentations with it. It's worth exploring!! In the next code block, I will define a augmentation sequence. You will notice `Oneof` and it does exactly that. At each iteration, it will take one augmentation technique out of the three and will apply that on the samples

In [8]:

```
# Augmentation sequence
seq = iaa.OneOf([
    iaa.Fliplr(), # horizontal flips
    iaa.Affine(rotate=20), # roatation
    iaa.Multiply((1.2, 1.5))]) #random brightness
```

Training data generator

Here I will define a very simple data generator. You can do more than this if you want but I think at this point, this is more than enough I need.

In [9]:

```
def data_gen(data, batch_size):
    # Get total number of samples in the data
    n = len(data)
    steps = n//batch_size

    # Define two numpy arrays for containing batch data and labels
    batch_data = np.zeros((batch_size, 224, 224, 3), dtype=np.float32)
    batch_labels = np.zeros((batch_size,2), dtype=np.float32)

    # Get a numpy array of all the indices of the input data
    indices = np.arange(n)

    # Initialize a counter
    i =0
    while True:
        np.random.shuffle(indices)
        # Get the next batch
        count = 0
        next_batch = indices[(i*batch_size):(i+1)*batch_size]
        for j, idx in enumerate(next_batch):
            img_name = data.iloc[idx]['image']
            label = data.iloc[idx]['label']

            # one hot encoding
            encoded_label = to_categorical(label, num_classes=2)
            # read the image and resize
            img = cv2.imread(str(img_name))
            img = cv2.resize(img, (224,224))

            # check if it's grayscale
            if img.shape[2]==1:
                img = np.dstack([img, img, img])

            # cv2 reads in BGR mode by default
            orig_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            # normalize the image pixels
            orig_img = img.astype(np.float32)/255.
```

```
            batch_data[count] = orig_img
            batch_labels[count] = encoded_label

            # generating more samples of the undersampled class
            if label==0 and count < batch_size-2:
                aug_img1 = seq.augment_image(img)
                aug_img2 = seq.augment_image(img)
                aug_img1 = cv2.cvtColor(aug_img1, cv2.COLOR_BGR2RGB)
                aug_img2 = cv2.cvtColor(aug_img2, cv2.COLOR_BGR2RGB)
                aug_img1 = aug_img1.astype(np.float32)/255.
                aug_img2 = aug_img2.astype(np.float32)/255.

                batch_data[count+1] = aug_img1
                batch_labels[count+1] = encoded_label
                batch_data[count+2] = aug_img2
                batch_labels[count+2] = encoded_label
                count +=2

            else:
                count+=1

            if count==batch_size-1:
                break

        i+=1
        yield batch_data, batch_labels

        if i>=steps:
            i=0
```

Model

This is the best part. If you look at other kernels on this dataset, everyone is busy doing transfer learning and fine-tuning. **You should transfer learn but wisely**. We will be doing partial transfer learning and rest of the model will be trained from scratch. I will explain this in detail but before that, I would love to share one of the best practices when it comes to building deep learning models from scratch on limited data.

1. Choose a simple architecture.
2. Initialize the first few layers from a network that is pretrained on imagenet. This is because first few layers capture general details like color blobs, patches, edges, etc. Instead of randomly initialized weights for these layers, it would be much better if you fine tune them.
3. Choose layers that introduce a lesser number of parameters. For example, `Depthwise SeparableConv` is a good replacement for `Conv` layer. It introduces lesser number of parameters as compared to normal convolution and as different filters are applied to each channel, it captures more information. `Xception` a powerful network, is built on top of such layers only. You can read about `Xception` and `Depthwise Separable Convolutions` in this paper.

4. Use batch norm with convolutions. As the network becomes deeper, batch norm start to play an important role.
5. Add dense layers with reasonable amount of neurons. Train with a higher learning rate and experiment with the number of neurons in the dense layers. Do it for the depth of your network too.
6. Once you know a good depth, start training your network with a lower learning rate along with decay.

This is all that I have done in the next code block.

In [10]:

```
def build_model():
    input_img = Input(shape=(224,224,3), name='ImageInput')
    x = Conv2D(64, (3,3), activation='relu', padding='same', name='Conv1_1')(
input_img)
    x = Conv2D(64, (3,3), activation='relu', padding='same', name='Conv1_2')(
x)
    x = MaxPooling2D((2,2), name='pool1')(x)

    x = SeparableConv2D(128, (3,3), activation='relu', padding='same', name='
Conv2_1')(x)
    x = SeparableConv2D(128, (3,3), activation='relu', padding='same', name='
Conv2_2')(x)
    x = MaxPooling2D((2,2), name='pool2')(x)

    x = SeparableConv2D(256, (3,3), activation='relu', padding='same', name='
Conv3_1')(x)
    x = BatchNormalization(name='bn1')(x)
    x = SeparableConv2D(256, (3,3), activation='relu', padding='same', name='
Conv3_2')(x)
    x = BatchNormalization(name='bn2')(x)
    x = SeparableConv2D(256, (3,3), activation='relu', padding='same', name='
Conv3_3')(x)
    x = MaxPooling2D((2,2), name='pool3')(x)

    x = SeparableConv2D(512, (3,3), activation='relu', padding='same', name='
Conv4_1')(x)
    x = BatchNormalization(name='bn3')(x)
    x = SeparableConv2D(512, (3,3), activation='relu', padding='same', name='
Conv4_2')(x)
    x = BatchNormalization(name='bn4')(x)
    x = SeparableConv2D(512, (3,3), activation='relu', padding='same', name='
Conv4_3')(x)
    x = MaxPooling2D((2,2), name='pool4')(x)

    x = Flatten(name='flatten')(x)
    x = Dense(1024, activation='relu', name='fc1')(x)
    x = Dropout(0.7, name='dropout1')(x)
    x = Dense(512, activation='relu', name='fc2')(x)
    x = Dropout(0.5, name='dropout2')(x)
```

```
    x = Dense(2, activation='softmax', name='fc3')(x)

    model = Model(inputs=input_img, outputs=x)
    return model
```

In [11]:

```
model =  build_model()
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
ImageInput (InputLayer)      (None, 224, 224, 3)       0
_________________________________________________________________
Conv1_1 (Conv2D)             (None, 224, 224, 64)      1792
_________________________________________________________________
Conv1_2 (Conv2D)             (None, 224, 224, 64)      36928
_________________________________________________________________
pool1 (MaxPooling2D)         (None, 112, 112, 64)      0
_________________________________________________________________
Conv2_1 (SeparableConv2D)    (None, 112, 112, 128)     8896
_________________________________________________________________
Conv2_2 (SeparableConv2D)    (None, 112, 112, 128)     17664
_________________________________________________________________
pool2 (MaxPooling2D)         (None, 56, 56, 128)       0
_________________________________________________________________
Conv3_1 (SeparableConv2D)    (None, 56, 56, 256)       34176
_________________________________________________________________
bn1 (BatchNormalization)     (None, 56, 56, 256)       1024
_________________________________________________________________
Conv3_2 (SeparableConv2D)    (None, 56, 56, 256)       68096
_________________________________________________________________
bn2 (BatchNormalization)     (None, 56, 56, 256)       1024
_________________________________________________________________
Conv3_3 (SeparableConv2D)    (None, 56, 56, 256)       68096
_________________________________________________________________
pool3 (MaxPooling2D)         (None, 28, 28, 256)       0
_________________________________________________________________
Conv4_1 (SeparableConv2D)    (None, 28, 28, 512)       133888
_________________________________________________________________
bn3 (BatchNormalization)     (None, 28, 28, 512)       2048
_________________________________________________________________
Conv4_2 (SeparableConv2D)    (None, 28, 28, 512)       267264
_________________________________________________________________
bn4 (BatchNormalization)     (None, 28, 28, 512)       2048
_________________________________________________________________
Conv4_3 (SeparableConv2D)    (None, 28, 28, 512)       267264
_________________________________________________________________
```

```
pool4 (MaxPooling2D)         (None, 14, 14, 512)       0
_________________________________________________________________
flatten (Flatten)            (None, 100352)            0
_________________________________________________________________
fc1 (Dense)                  (None, 1024)              102761472
_________________________________________________________________
dropout1 (Dropout)           (None, 1024)              0
_________________________________________________________________
fc2 (Dense)                  (None, 512)               524800
_________________________________________________________________
dropout2 (Dropout)           (None, 512)               0
_________________________________________________________________
fc3 (Dense)                  (None, 2)                 1026
=================================================================
Total params: 104,197,506
Trainable params: 104,194,434
Non-trainable params: 3,072
_________________________________________________________________
```

We will initialize the weights of first two convolutions with imagenet weights,

In [12]:

```
# Open the VGG16 weight file
f = h5py.File('../input/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5', 'r')

# Select the layers for which you want to set weight.

w,b = f['block1_conv1']['block1_conv1_W_1:0'], f['block1_conv1']['block1_conv1_b_1:0']
model.layers[1].set_weights = [w,b]

w,b = f['block1_conv2']['block1_conv2_W_1:0'], f['block1_conv2']['block1_conv2_b_1:0']
model.layers[2].set_weights = [w,b]

w,b = f['block2_conv1']['block2_conv1_W_1:0'], f['block2_conv1']['block2_conv1_b_1:0']
model.layers[4].set_weights = [w,b]

w,b = f['block2_conv2']['block2_conv2_W_1:0'], f['block2_conv2']['block2_conv2_b_1:0']
model.layers[5].set_weights = [w,b]

f.close()
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
```

ImageInput (InputLayer)	(None, 224, 224, 3)	0
Conv1_1 (Conv2D)	(None, 224, 224, 64)	1792
Conv1_2 (Conv2D)	(None, 224, 224, 64)	36928
pool1 (MaxPooling2D)	(None, 112, 112, 64)	0
Conv2_1 (SeparableConv2D)	(None, 112, 112, 128)	8896
Conv2_2 (SeparableConv2D)	(None, 112, 112, 128)	17664
pool2 (MaxPooling2D)	(None, 56, 56, 128)	0
Conv3_1 (SeparableConv2D)	(None, 56, 56, 256)	34176
bn1 (BatchNormalization)	(None, 56, 56, 256)	1024
Conv3_2 (SeparableConv2D)	(None, 56, 56, 256)	68096
bn2 (BatchNormalization)	(None, 56, 56, 256)	1024
Conv3_3 (SeparableConv2D)	(None, 56, 56, 256)	68096
pool3 (MaxPooling2D)	(None, 28, 28, 256)	0
Conv4_1 (SeparableConv2D)	(None, 28, 28, 512)	133888
bn3 (BatchNormalization)	(None, 28, 28, 512)	2048
Conv4_2 (SeparableConv2D)	(None, 28, 28, 512)	267264
bn4 (BatchNormalization)	(None, 28, 28, 512)	2048
Conv4_3 (SeparableConv2D)	(None, 28, 28, 512)	267264
pool4 (MaxPooling2D)	(None, 14, 14, 512)	0
flatten (Flatten)	(None, 100352)	0
fc1 (Dense)	(None, 1024)	102761472
dropout1 (Dropout)	(None, 1024)	0
fc2 (Dense)	(None, 512)	524800

```
_________________________________________________________________
dropout2 (Dropout)           (None, 512)               0         
_________________________________________________________________
fc3 (Dense)                  (None, 2)                 1026      
=================================================================
Total params: 104,197,506
Trainable params: 104,194,434
Non-trainable params: 3,072
_________________________________________________________________
```

In [13]:

```
# opt = RMSprop(lr=0.0001, decay=1e-6)
opt = Adam(lr=0.0001, decay=1e-5)
es = EarlyStopping(patience=5)
chkpt = ModelCheckpoint(filepath='best_model_todate', save_best_only=True, sa
ve_weights_only=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'],optimizer=opt)
```

In [14]:

```
batch_size = 16
nb_epochs = 20

# Get a train data generator
train_data_gen = data_gen(data=train_data, batch_size=batch_size)

# Define the number of training steps
nb_train_steps = train_data.shape[0]//batch_size

print("Number of training and validation steps: {} and {}".format(nb_train_st
eps, len(valid_data)))
```

```
Number of training and validation steps: 326 and 16
```

I have commented out the training step as of now as it will train the network again while rendering the notebook and I would have to wait for an hour or so which I don't want to. I have uploaded the weights of the best model I achieved so far. Feel free to use it for further fine-tuning of the network. We will load those weights and will run the inference on the test set using those weights only. But...but for your reference, I will attach the screenshot of the training steps here.

In [15]:

```
# # Fit the model
# history = model.fit_generator(train_data_gen, epochs=nb_epochs, steps_per_epo
ch=nb_train_steps,
#                               validation_data=(valid_data, valid_labels),call
backs=[es, chkpt],
#                               class_weight={0:1.0, 1:0.4})
```

```
WARNING:tensorflow:Variable *= will be deprecated. Use variable.assign_mul if you want assignment t
o the variable value or 'x = x * y' if you want a new python Tensor object.
Epoch 1/20
326/326 [==============================] - 208s 639ms/step - loss: 0.2338 - acc: 0.7651 - val_loss:
1.5346 - val_acc: 0.5000
Epoch 2/20
326/326 [==============================] - 200s 615ms/step - loss: 0.0921 - acc: 0.9505 - val_loss:
0.6662 - val_acc: 0.7500
Epoch 3/20
326/326 [==============================] - 201s 615ms/step - loss: 0.0761 - acc: 0.9592 - val_loss:
0.2838 - val_acc: 0.9375
Epoch 4/20
326/326 [==============================] - 201s 615ms/step - loss: 0.0647 - acc: 0.9640 - val_loss:
0.2067 - val_acc: 0.8750
Epoch 5/20
326/326 [==============================] - 200s 614ms/step - loss: 0.0524 - acc: 0.9705 - val_loss:
0.2506 - val_acc: 0.9375
Epoch 6/20
326/326 [==============================] - 201s 616ms/step - loss: 0.0606 - acc: 0.9697 - val_loss:
0.3630 - val_acc: 0.8125
Epoch 7/20
326/326 [==============================] - 201s 616ms/step - loss: 0.0542 - acc: 0.9680 - val_loss:
0.1701 - val_acc: 0.9375
Epoch 8/20
326/326 [==============================] - 200s 615ms/step - loss: 0.0479 - acc: 0.9770 - val_loss:
0.2480 - val_acc: 0.8750
Epoch 9/20
326/326 [==============================] - 201s 615ms/step - loss: 0.0438 - acc: 0.9795 - val_loss:
0.5402 - val_acc: 0.7500
Epoch 10/20
326/326 [==============================] - 201s 615ms/step - loss: 0.0332 - acc: 0.9824 - val_loss:
1.6942 - val_acc: 0.5625
Epoch 11/20
326/326 [==============================] - 201s 617ms/step - loss: 0.0347 - acc: 0.9810 - val_loss:
0.1863 - val_acc: 0.9375
Epoch 12/20
326/326 [==============================] - 201s 616ms/step - loss: 0.0379 - acc: 0.9826 - val_loss:
0.0497 - val_acc: 1.0000
Epoch 13/20
326/326 [==============================] - 200s 613ms/step - loss: 0.0412 - acc: 0.9806 - val_loss:
0.5208 - val_acc: 0.8125
Epoch 14/20
326/326 [==============================] - 200s 615ms/step - loss: 0.0316 - acc: 0.9868 - val_loss:
0.2438 - val_acc: 0.9375
Epoch 15/20
326/326 [==============================] - 200s 615ms/step - loss: 0.0352 - acc: 0.9860 - val_loss:
0.4822 - val_acc: 0.8125
Epoch 16/20
326/326 [==============================] - 200s 614ms/step - loss: 0.0357 - acc: 0.9845 - val_loss:
0.6291 - val_acc: 0.7500
Epoch 17/20
326/326 [==============================] - 200s 615ms/step - loss: 0.0338 - acc: 0.9883 - val_loss:
0.9483 - val_acc: 0.7500
```

In [16]:

```
# Load the model weights
model.load_weights("../input/xray-best-model/best_model/best_model.hdf5")
```

In [17]:

```
# Preparing test data
normal_cases_dir = test_dir / 'NORMAL'
pneumonia_cases_dir = test_dir / 'PNEUMONIA'

normal_cases = normal_cases_dir.glob('*.jpeg')
pneumonia_cases = pneumonia_cases_dir.glob('*.jpeg')

test_data = []
test_labels = []

for img in normal_cases:
    img = cv2.imread(str(img))
    img = cv2.resize(img, (224,224))
    if img.shape[2] ==1:
        img = np.dstack([img, img, img])
    else:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = img.astype(np.float32)/255.
    label = to_categorical(0, num_classes=2)
    test_data.append(img)
    test_labels.append(label)

for img in pneumonia_cases:
    img = cv2.imread(str(img))
    img = cv2.resize(img, (224,224))
    if img.shape[2] ==1:
        img = np.dstack([img, img, img])
    else:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = img.astype(np.float32)/255.
    label = to_categorical(1, num_classes=2)
    test_data.append(img)
    test_labels.append(label)


test_data = np.array(test_data)
test_labels = np.array(test_labels)

print("Total number of test examples: ", test_data.shape)
print("Total number of labels:", test_labels.shape)
```

```
Total number of test examples:  (624, 224, 224, 3)
Total number of labels: (624, 2)
```

In [18]:

```
# Evaluation on test dataset
test_loss, test_score = model.evaluate(test_data, test_labels, batch_size=16)
print("Loss on test set: ", test_loss)
print("Accuracy on test set: ", test_score)
```

```
624/624 [==============================] - 7s 11ms/step
Loss on test set:  0.905697194154084
Accuracy on test set:  0.8269230769230769
```

In [19]:

```
# Get predictions
preds = model.predict(test_data, batch_size=16)
preds = np.argmax(preds, axis=-1)

# Original labels
orig_test_labels = np.argmax(test_labels, axis=-1)

print(orig_test_labels.shape)
print(preds.shape)
```

```
(624,)
(624,)
```

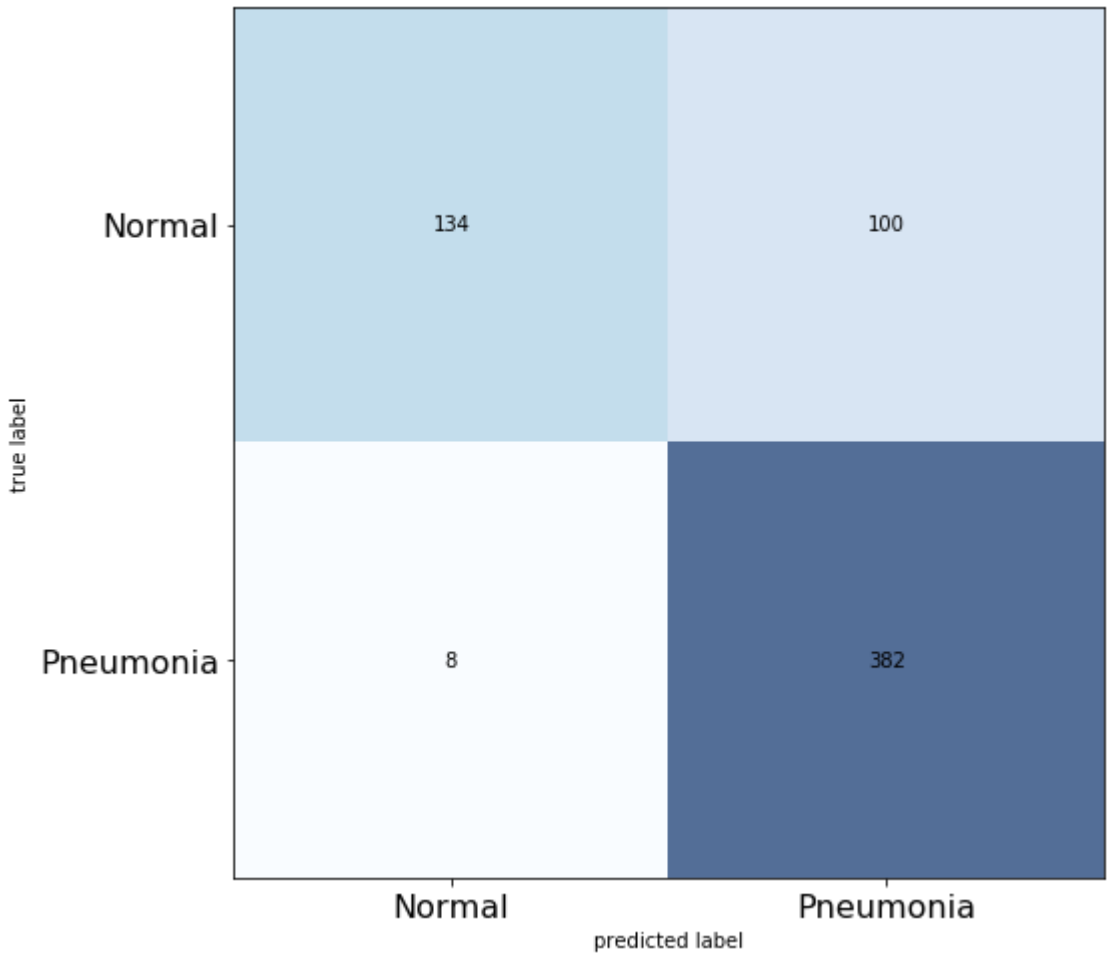
When a particular problem includes an imbalanced dataset, then accuracy isn't a good metric to look for. For example, if your dataset contains 95 negatives and 5 positives, having a model with 95% accuracy doesn't make sense at all. The classifier might label every example as negative and still achieve 95% accuracy. Hence, we need to look for alternative metrics. **Precision** and **Recall** are really good metrics for such kind of problems.

We will get the confusion matrix from our predictions and see what is the recall and precision of our model.

In [20]:

```
# Get the confusion matrix
cm  = confusion_matrix(orig_test_labels, preds)
plt.figure()
plot_confusion_matrix(cm,figsize=(12,8), hide_ticks=True, alpha=0.7,cmap=plt.
cm.Blues)
plt.xticks(range(2), ['Normal', 'Pneumonia'], fontsize=16)
plt.yticks(range(2), ['Normal', 'Pneumonia'], fontsize=16)
plt.show()
```

```
<matplotlib.figure.Figure at 0x7f41ec395fd0>
```



In [21]:

```
# Calculate Precision and Recall
tn, fp, fn, tp = cm.ravel()

precision = tp/(tp+fp)
recall = tp/(tp+fn)

print("Recall of the model is {:.2f}".format(recall))
print("Precision of the model is {:.2f}".format(precision))
```

```
Recall of the model is 0.98
Precision of the model is 0.79
```

Our model has a 98% recall. In such problems, a good recall value is expected. But if you notice, the precision is only 80%. This is one thing to notice. Precision and Recall follows a trade-off, and you need to find a point where your recall, as well as your precision, is more than good but both can't increase simultaneously.