# -\*- coding: utf-8 -\*-

"""train.ipynb

Automatically generated by Colaboratory.

Original file is located at

https://colab.research.google.com/drive/1c0uoKEuvj5r4CQuDkdbnCPAaIgpW9ZzY

# Chest X-Ray Medical Diagnosis with Deep Learning

"""

# Commented out IPython magic to ensure Python compatibility.

from google.colab import drive

drive.mount('/content/drive')

"""<img src="xray-header-image.png" style="padding-top: 50px;width: 87%;left: 0px;margin-left: 0px;margin-right: 0px;">

We will explore medical image diagnosis by building a state-of-the-art chest X-ray classifier using Keras.

In particular, we 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

We will discuss in details 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

## Outline

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

- [1. Import Packages and Function](#1)

- [2. Load the Datasets](#2)

- [2.1 Preventing Data Leakage](#2-1)

- [Exercise 1 - Checking Data Leakage](#Ex-1)

- [2.2 Preparing Images](#2-2)

- [3. Model Development](#3)

- [3.1 Addressing Class Imbalance](#3-1)

- [Exercise 2 - Computing Class Frequencies](#Ex-2)

- [Exercise 3 - Weighted Loss](#Ex-3)

- [3.3 DenseNet121](#3-3)

- [4. Training [optional]](#4)

- [4.1 Training on the Larger Dataset](#4-1)

- [5. Prediction and Evaluation](#5)

- [5.1 ROC Curve and AUROC](#5-1)

- [5.2 Visualizing Learning with GradCAM](#5-2)

<a name='1'></a>

## 1. Import Packages and Functions¶

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.

"""

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

"""<a name='2'></a>

## 2 Load the Datasets

We will be using the [ChestX-ray8 dataset](https://arxiv.org/abs/1705.02315) 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.

We can download the entire dataset for free [here](https://nihcc.app.box.com/v/ChestXray-NIHCC).

- 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.

This dataset has been annotated by consensus among four different radiologists for 5 of our 14 pathologies:

- `Consolidation`

- `Edema`

- `Effusion`

- `Cardiomegaly`

- `Atelectasis`

#### Read in the data

Let's open these files using the [pandas](https://pandas.pydata.org/) library

"""

train\_df = pd.read\_csv("/content/drive/My Drive/Colab Notebooks/preprocessed\_dataset/train-small.csv")

valid\_df = pd.read\_csv("/content/drive/My Drive/Colab Notebooks/preprocessed\_dataset/valid-small.csv")

test\_df = pd.read\_csv("/content/drive/My Drive/Colab Notebooks/preprocessed\_dataset/test.csv")

train\_df.head()

labels = ['Cardiomegaly',

'Emphysema',

'Effusion',

'Hernia',

'Infiltration',

'Mass',

'Nodule',

'Atelectasis',

'Pneumothorax',

'Pleural\_Thickening',

'Pneumonia',

'Fibrosis',

'Edema',

'Consolidation']

"""<a name='2-1'></a>

### 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.

<a name='Ex-1'></a>

### Task 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.

"""

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

"""

df1\_patients\_unique = set(df1[patient\_col].values)

df2\_patients\_unique = set(df2[patient\_col].values)

patients\_in\_both\_groups = list(df1\_patients\_unique.intersection(df2\_patients\_unique))

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

leakage = len( patients\_in\_both\_groups)>0 # boolean (true if there is at least 1 patient in both groups)

return leakage

# 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')}")

"""##### Expected output

```Python

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.

"""

print("leakage between train and test: {}".format(check\_for\_leakage(train\_df, test\_df, 'PatientId')))

print("leakage between valid and test: {}".format(check\_for\_leakage(valid\_df, test\_df, 'PatientId')))

"""If we get `False` for both, then we're ready to start preparing the datasets for training.

<a name='2-2'></a>

### 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](https://keras.io/preprocessing/image/) 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

"""

def get\_train\_generator(df, image\_dir, x\_col, y\_cols, shuffle=True, batch\_size=16, seed=1, target\_w = 320, target\_h = 320):

"""

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.

\*\* We use the same generator as 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.

"""

def get\_test\_and\_valid\_generator(valid\_df, test\_df, train\_df, image\_dir, x\_col, y\_cols, sample\_size=100, batch\_size=16, seed=1, target\_w = 320, target\_h = 320):

"""

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 set and validation set respectively

"""

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

test\_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))

valid\_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 test\_generator, valid\_generator

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

IMAGE\_DIR = "/content/drive/My Drive/Colab Notebooks/preprocessed\_dataset/images-small"

train\_generator = get\_train\_generator(train\_df, IMAGE\_DIR, "Image", labels)

test\_generator, valid\_generator= get\_test\_and\_valid\_generator(valid\_df, test\_df, train\_df, IMAGE\_DIR, "Image", labels)

"""Let's check what the generator gives our model during training and validation. We can do this by calling the `\_\_get\_item\_\_(index)` function:"""

x, y = train\_generator.\_\_getitem\_\_(0)

plt.imshow(x[0]);

"""<a name='3'></a>

## 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.

<a name='3-1'></a>

### 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:

"""

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:

$$freq\_{p} = \frac{\text{number of positive examples}}{N} $$

$$\text{and}$$

$$freq\_{n} = \frac{\text{number of negative examples}}{N}.$$

<a name='Ex-2'></a>

### Task 2 - Computing Class Frequencies

Calculation of the frequences for each label in our dataset.

"""

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)

"""

# total number of patients (rows)

N = len(labels)

positive\_frequencies = np.sum(labels,axis=0)/N

negative\_frequencies = 1-positive\_frequencies

return positive\_frequencies, negative\_frequencies

# 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}")

"""##### Expected output

```Python

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.

"""

freq\_pos, freq\_neg = compute\_class\_freqs(train\_generator.labels)

freq\_pos

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

data = pd.DataFrame({"Class": labels, "Label": "Positive", "Value": freq\_pos})

data = data.append([{"Class": labels[l], "Label": "Negative", "Value": v} for l,v in enumerate(freq\_neg)], ignore\_index=True)

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.

"""

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:"""

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

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

<a name='Ex-3'></a>

### Task 3 - Weighted Loss

The `weighted\_loss` function below returns a loss function that calculates the weighted loss for each batch. For the multi-class loss, we add up the average loss for each individual class. We also add a small value, $\epsilon$, 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.

"""

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\_classes)

neg\_weights (np.array): array of negative weights for each class, size (num\_classes)

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\_classes)

y\_pred (Tensor): Tensor of predicted labels, size is (num\_examples, num\_classes)

-1 \* np.sum(w\_p[0] \*

y\_true[:, 0] \*

np.log(y\_pred[:, 0])

)

Returns:

loss (Float): overall scalar loss summed across all classes

"""

# initialize loss to zero

loss = 0.0

y\_true = K.cast(y\_true, dtype='float32')

y\_pred = K.cast(y\_pred, dtype='float32')

for i in range(len(pos\_weights)):

# for each class, add average weighted loss for that class

loss += -1 \* K.mean(pos\_weights[i]\*y\_true[:,i]\*K.log(y\_pred[:,i]+epsilon)) + -1 \* K.mean(neg\_weights[i]\*(1-y\_true[:,i])\*K.log(1-y\_pred[:,i]+epsilon)) #complete this line

return loss

return weighted\_loss

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

# 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}")

"""#### Additional check

If we implemented the function correctly, then if the epsilon for the `get\_weighted\_loss` is set to `1`, the weighted losses will be as follows:

```Python

L(y\_pred\_1)= -0.4956, L(y\_pred\_2)= -0.4956

```

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

<a name='3-3'></a>

### 3.3 DenseNet121

Next, we will use a pre-trained [DenseNet121](https://www.kaggle.com/pytorch/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.

"""

from keras.optimizers import SGD

from keras.metrics import \*

METRICS = [

TruePositives(name='tp'),

FalsePositives(name='fp'),

TrueNegatives(name='tn'),

FalseNegatives(name='fn'),

BinaryAccuracy(name='accuracy'),

Precision(name='precision'),

Recall(name='recall'),

AUC(name='auc'),

]

# create the base pre-trained model

base\_model = DenseNet121(weights='/content/drive/My Drive/Colab Notebooks/preprocessed\_dataset/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)

optimizer = SGD(learning\_rate=0.001,momentum=0.9,nesterov=False)

model = Model(inputs=base\_model.input, outputs=predictions)

model.compile(optimizer=optimizer, loss=get\_weighted\_loss(pos\_weights, neg\_weights),metrics=METRICS)

"""<a name='4'></a>

## 4 Training

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, we can use the code shown below to practice training the model locally on our machine or in Colab.

Python Code for training the model:

```python

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()

```

"""

!pip install pyyaml h5py

import os

checkpoint\_path = "training\_1/cp.ckpt"

checkpoint\_dir = os.path.dirname(checkpoint\_path)

# Create a callback that saves the model's weights

cp\_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint\_path,

save\_weights\_only=True,

verbose=1)

history = model.fit\_generator(train\_generator,

validation\_data=test\_generator,

steps\_per\_epoch=100,

validation\_steps=25,

epochs = 100)

plt.plot(history.history['loss'])

plt.ylabel("loss")

plt.xlabel("epoch")

plt.title("Training Loss Curve")

plt.show()

#predicted\_vals = model.predict\_generator(test\_generator, steps = len(test\_generator))

#auc\_rocs = util.get\_roc\_curve(labels, predicted\_vals, test\_generator)

#df = pd.read\_csv("preprocessed\_dataset/train-small.csv")

#IMAGE\_DIR = "preprocessed\_dataset/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."""

#util.compute\_gradcam(model, '00008270\_015.png', IMAGE\_DIR, df, labels, labels\_to\_show)

#util.compute\_gradcam(model, '00011355\_002.png', IMAGE\_DIR, df, labels, labels\_to\_show)

#util.compute\_gradcam(model, '00029855\_001.png', IMAGE\_DIR, df, labels, labels\_to\_show)

#util.compute\_gradcam(model, '00005410\_000.png', IMAGE\_DIR, df, labels, labels\_to\_show)