CNN Cancer Detection

A deep learning project 💪

Overview

In today's world we are fortunate enough to have advanced technology which allows the medical field to provide better patient care. Cancer is a highly researched area because there are so many people who suffer and/or die of cancerous disease. This study aims to improve cancer detection in lymph nodes by using computer vision machine learning technqiues. We will examine the data given in this competition to get a better understanding of it. Then, we will run multiple convolutional neural network models with the intent to be able to classify cancerous (1) and non-cancerous cells (0). With improved and faster cancer detection, patients will be able to recieve life-saving treatments faster. This the first step in those peoples cancer survival story. This notebook covers the thought process as to how to create simple CNN models. We will create two models, one without hyperparameter tuning, and one with tuning. Finally, we will suggest ways to improve the models in future studies.

Layout for this notebook was given in assignment brief and is as follows:

1. Brief Description of the Problem and Data
2. Exploratory Data Analysis (EDA) - Inspect, Visualize and Clean the Data
3. Describe Model Architecture
4. Results and Analysis
5. Conclusion

In [1]:
```
#import libraries

#general libraries
import numpy as np
import pandas as pd
import os
import random
from sklearn.utils import shuffle
import shutil

#visualizations
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import matplotlib.patches as patches

# work with images
from skimage.transform import rotate
from skimage import io
import cv2 as cv

# model development
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import RandomFlip, RandomZoom, RandomRotation
from tensorflow.keras.layers import Conv2D, MaxPooling2D, AveragePooling2D
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.optimizers import Adam

import warnings
warnings.simplefilter("ignore", category=DeprecationWarning)
```

In [2]:
```
tf.__version__
```

Out[2]:
```
'2.6.4'
```

In [3]:
```
#get files
test_path = '../input/histopathologic-cancer-detection/test/'
train_path = '../input/histopathologic-cancer-detection/train/'
sample_submission = pd.read_csv('../input/histopathologic-cancer-detection/sample_submission.csv')
train_data = pd.read_csv('../input/histopathologic-cancer-detection/train_labels.csv')
```

In [4]:
```
# declare constants for reproduciblity
RANDOM_STATE = 49
```

1. Brief Description of the Problem and Data

This data contains thousands of small images where the 96x96 pixel images with 3 channels, each with an identifying label and id. We have two datasets, a training and testing set already split for us. The training set contains 220,025 unique images and the test set contains about 57,500. To use these images in a machine learning model, we are also given an identifying dataframe with two columns: 'id' which is the unique image ID correpsonding to the training directory, and 'label' which tells us the classification category. Each label is either a 0 or 1, depending whether the image is non-cancerous (0) or cancerous (1). In the competition description, we find that if at least one pixel of an image is identified as cancerous then the whole image is therefore marked with a 1, otherwise it is 0. It is important to note that we do not have any missing values in this data which will make preprocessing more efficient.

In [5]:

```
# have a look at the format of the data
train_data.head()
```

Out[5]:

	id	label
0	f38a6374c348f90b587e046aac6079959adf3835	0
1	c18f2d887b7ae4f6742ee445113fa1aef383ed77	1
2	755db6279dae599ebb4d39a9123cce439965282d	0
3	bc3f0c64fb968ff4a8bd33af6971ecae77c75e08	0
4	068aba587a4950175d04c680d38943fd488d6a9d	0

In [6]:

```
# take a look at the data further
train_data.describe()
```

Out[6]:

	label
count	220025.000000
mean	0.405031
std	0.490899
min	0.000000
25%	0.000000
50%	0.000000
75%	1.000000
max	1.000000

In [7]:

```
# check information, data types, and for missing data
train_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 220025 entries, 0 to 220024
Data columns (total 2 columns):
 #   Column  Non-Null Count   Dtype 
---  ------  --------------   ----- 
 0   id      220025 non-null  object
 1   label   220025 non-null  int64 
dtypes: int64(1), object(1)
memory usage: 3.4+ MB
```

2. Exploratory Data Analysis (EDA) - Inspect, Visualize and Clean the Data

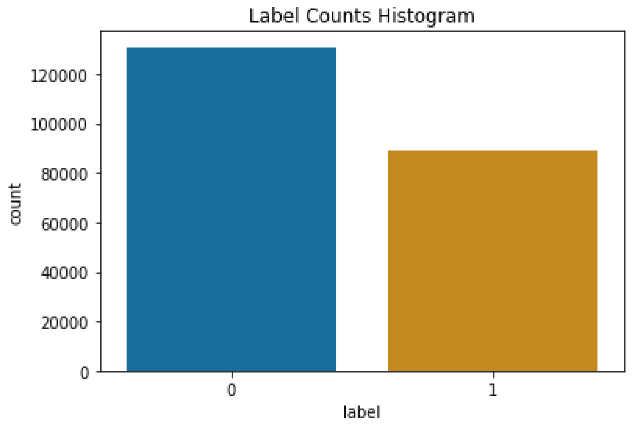
First, we will visualize the data. Then we will clean/preprocess the data.

We can see in the histogram and pie chart below that we have 59.5% of the labels are 0 (non-cancerous images) and 40.5% are labeled 1 (cancerous images). We were told in the competition description that the data is 50/50 split between cancerous and non-cancerous images however, from what we are finding here, we have a split which is closer to 40/60. This means that our data is unbalanced, however it is not severely unbalanced either (compared to a split such as 30/70 or even 10/90). We also have thousands of images to train with. For this reason, we can assume we will be able to create a sufficiently performing model which identifies cancerous images.

In [8]:

```
#create histogram
print(pd.DataFrame(data={'Label Counts': train_data['label'].value_counts()}))
sns.countplot(x=train_data['label'], palette='colorblind').set(title='Label Counts Histogram');
```

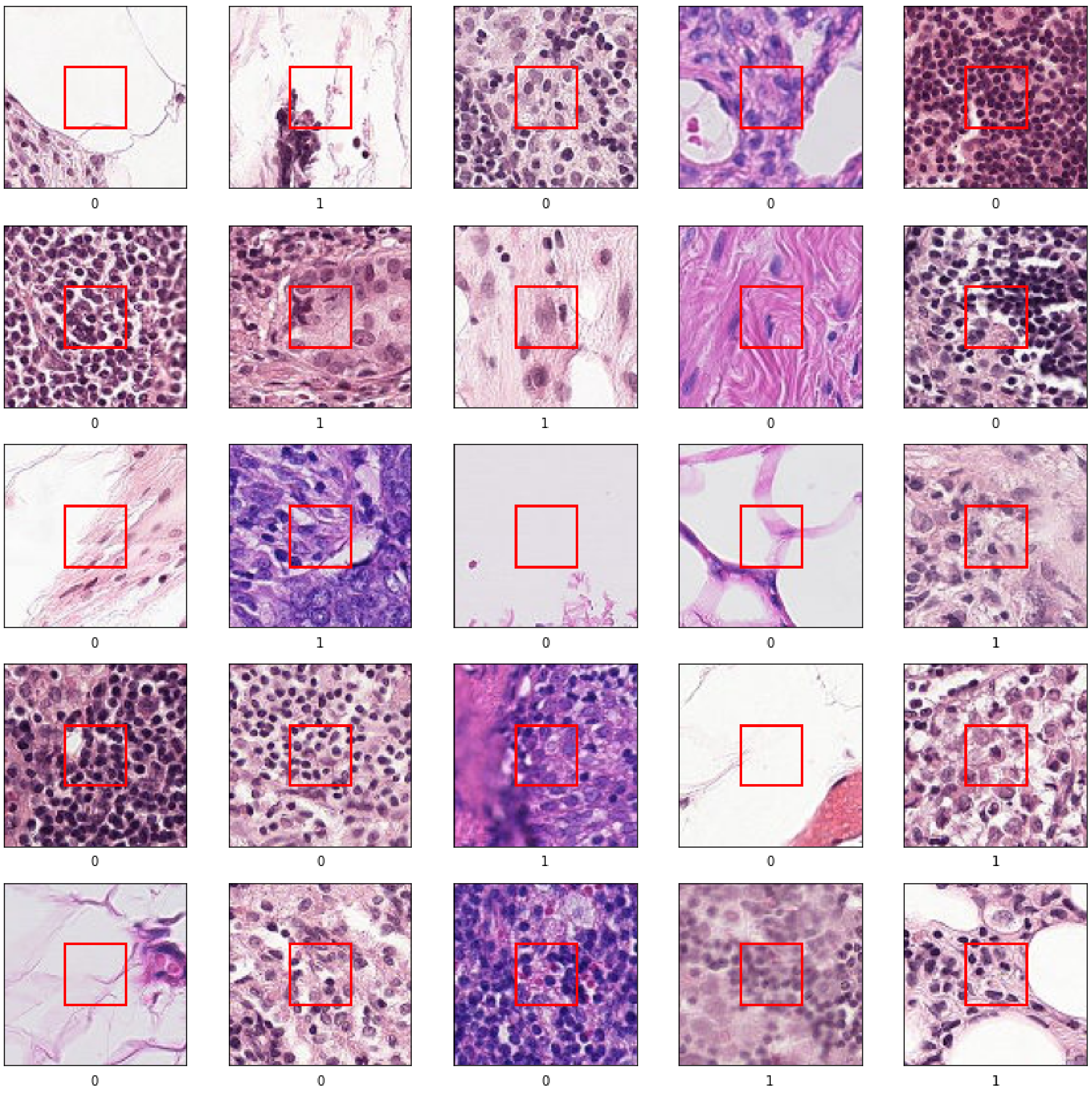
```
   Label Counts
0        130908
1         89117
```



In [9]:
```
#create pie chart
fig = px.pie(train_data,
             values = train_data['label'].value_counts().values,
             names = train_data['label'].unique())
fig.update_layout(
    title={
        'text': "Label Percentage Pie Chart",
        'y':.99,
        'x':0.5,
        'xanchor': 'center',
        'yanchor': 'top'})
fig.show()
```

Below we can see some example images from the training data. As someone who is not familiar at looking at cancer cells and images, it would be very challenging for me to classify these. However, we mitigate this by putting the correct label for each image is below it. Additionally, we were told that each image has the (potentially) cancerous cells centered in each 32x32 pixel image, so we have drawn a box around this area as a focal point.

In [10]:
```
#visualize a few images
fig, ax = plt.subplots(5, 5, figsize=(15, 15))
for i, axis in enumerate(ax.flat):
    file = str(train_path + train_data.id[i] + '.tif')
    image = io.imread(file)
    axis.imshow(image)
    box = patches.Rectangle((32,32),32,32, linewidth=2, edgecolor='r',facecolor='none', linestyle='-')
    axis.add_patch(box)
    axis.set(xticks=[], yticks=[], xlabel = train_data.label[i]);
    #cv2.waitKey(0)
```



Preprocessing Techniques

When working with training data, first we want to handle any missing data. From the source description and our findings above, we can see that we have no missing data in the training set. Next, we are told that the part of the image which is going to be of interest is the center pixels (red box above). Right now the images are 96x96x3 and we know that the "3" is the RGB channel. For now, we will leave this as well. Lastly, there are many augmentations you can perform on images to produce better results and facilitate model learning, but to start we will not touch these either for simplicity.

What we will do with the image data is **shuffle** the data so that the model doesn't learn based on the image ordering/pattern of input, which could potentially have consequences in the model training. We will also **split** the data into training and validation set to improve model development. During training we will also **normalize** the pixels by dividing by 255.0, which should help data processing and model training.

3. Describe Model Architecture

For this model we will be using Keras' library to run a convolutional neural network (CNN). The first model we will run without tuning any hyperparameters within the model and use that as our baseline. Then, we will run a second model, tuning hyperparameters such as learning rate, batch normalization, regularization, filter size, stride, activation layers, etc.

Our CNN model will have a network such that there are two convolutional layers then a MaxPool layer, and we repeat this *n* number of times. Specifically, we will create a fairly simple model with two (n=2) of these clusters. In other words, our model will be input --> Conv2D --> Conv2D --> MaxPool --> Conv2D --> Conv2D --> MaxPool --> Flatten --> Output with sigmoid activation.

First model:

1. Normalize images pre-training (image/255)
2. Output layer activation (sigmoid)

Second model contains all the first model parameters, but we also add:

1. Dropout (0.1)

2. Batch Normalization
3. Optimization (Adam)
4. Learning rate (0.0001)
5. Hidden layer activations (ReLU)

Before we start with the models, let's describe what each of the parameters will do to the model. In both models we will use 1 and 2, numbers 3 to 5 will be used in the second and final models.

1. **Normalize images** : this will take the pixels and divide each pixel by 255 to normalize the data and have values between 0-1.
2. **Output layer activation** : we will use a sigmoid activation function on the output layer since we are working with binary data
3. **Dropout** : we will set dropout at 0.1 which will randonly select some weights and set them to equal 0 which regularizes the model because it is using a smaller number of weights for each training run.
4. **Optimization** : we will use adaptive moment estimation (Adam) for optimizing the model which essentially mimics momentum for gradient adn gradient-squared.
5. **Learning rate**: we will set our learning rate to 0.0001 which will assist in the gradient descent such that as the model learns, the speed of learning decreases so that it is less likely to overstep the (hopefully) global minimum.
6. **Hidden layer activations**: we will use rectified linear regression (ReLU) as our hidden layer activation function which will help the model to converge better, prevent saturation, and provide less need for computation power.

In addition, we will use fairly large **batch sizes**, set at 256 to help reduce variance. Finally we wil train our two models with 10 **epochs**. We will use accuracy and the ROC-AUC curve to measure model performance, as well as binary cross-entropy as our loss function.

In [11]:
```
# set model constants
BATCH_SIZE = 256
```

In [12]:
```
# prepare data for training
def append_tif(string):
    return string+".tif"

train_data["id"] = train_data["id"].apply(append_tif)
train_data['label'] = train_data['label'].astype(str)

# randomly shuffle training data
train_data = shuffle(train_data, random_state=RANDOM_STATE)
```

In [13]:
```
# modify training data by normalizing it
# and split data into training and validation sets
datagen = ImageDataGenerator(rescale=1./255.,
                             validation_split=0.15)
```

In [14]:
```
# generate training data
train_generator = datagen.flow_from_dataframe(
    dataframe=train_data,
    directory=train_path,
    x_col="id",
    y_col="label",
    subset="training",
    batch_size=BATCH_SIZE,
    seed=RANDOM_STATE,
    class_mode="binary",
    target_size=(64,64))        # original image = (96, 96)
```

```
Found 187022 validated image filenames belonging to 2 classes.
```

In [15]:
```
# generate validation data
valid_generator = datagen.flow_from_dataframe(
    dataframe=train_data,
    directory=train_path,
    x_col="id",
    y_col="label",
    subset="validation",
    batch_size=BATCH_SIZE,
    seed=RANDOM_STATE,
    class_mode="binary",
    target_size=(64,64))        # original image = (96, 96)
```

```
Found 33003 validated image filenames belonging to 2 classes.
```

In [16]:
```
# Setup GPU accelerator - configure Strategy. Assume TPU...if not set default for GPU/CPU
tpu = None
try:
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.TPUStrategy(tpu)
except ValueError:
    strategy = tf.distribute.get_strategy()
```

Start First model building and training

Here we build our first model. This model is fairly simple with a handful of layers, void of activations except the last layer. The last layer we use a sigmoid activation function since our data is binary (0-1). We can see below how the model is built and how many parameters we will need to train for this model. The next cell below we can see the model training per epoch.

In [17]:
```
# set ROC AUC as metric
ROC_1 = tf.keras.metrics.AUC()

# use GPU
with strategy.scope():

    #create model
    model_one = Sequential()

    model_one.add(Conv2D(filters=16, kernel_size=(3,3)))
    model_one.add(Conv2D(filters=16, kernel_size=(3,3)))
    model_one.add(MaxPooling2D(pool_size=(2,2)))

    model_one.add(Conv2D(filters=32, kernel_size=(3,3)))
    model_one.add(Conv2D(filters=32, kernel_size=(3,3)))
    model_one.add(AveragePooling2D(pool_size=(2,2)))

    model_one.add(Flatten())
    model_one.add(Dense(1, activation='sigmoid'))

    #build model by input size
    model_one.build(input_shape=(BATCH_SIZE, 64, 64, 3))          # original image = (96, 96, 3)

    #compile
    model_one.compile(loss='binary_crossentropy', metrics=['accuracy', ROC_1])

#quick look at model
model_one.summary()
```

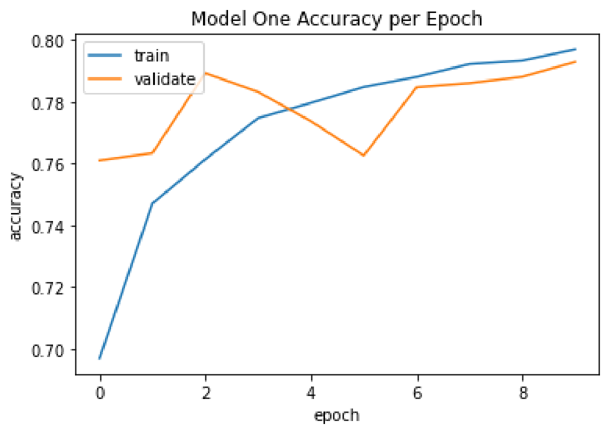
```
2022-09-14 11:37:43.151821: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-09-14 11:37:43.157362: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-09-14 11:37:43.158125: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-09-14 11:37:43.159304: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2022-09-14 11:37:43.159596: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (256, 62, 62, 16)         448       
_________________________________________________________________
conv2d_1 (Conv2D)            (256, 60, 60, 16)         2320      
_________________________________________________________________
max_pooling2d (MaxPooling2D) (256, 30, 30, 16)         0         
_________________________________________________________________
conv2d_2 (Conv2D)            (256, 28, 28, 32)         4640      
_________________________________________________________________
conv2d_3 (Conv2D)            (256, 26, 26, 32)         9248      
_________________________________________________________________
average_pooling2d (AveragePo (256, 13, 13, 32)         0         
_________________________________________________________________
flatten (Flatten)            (256, 5408)               0         
_________________________________________________________________
dense (Dense)                (256, 1)                  5409      
=================================================================
Total params: 22,065
Trainable params: 22,065
Non-trainable params: 0
_________________________________________________________________
node zero
2022-09-14 11:37:43.160318: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-09-14 11:37:43.160950: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-09-14 11:37:45.576116: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-09-14 11:37:45.576905: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-09-14 11:37:45.577603: I tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero
2022-09-14 11:37:45.578276: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 15381 MB memory:  -> device: 0, name: Tesla P100-PCIE-16GB, pci bus id: 0000:00:04.0, compute capability: 6.0
```

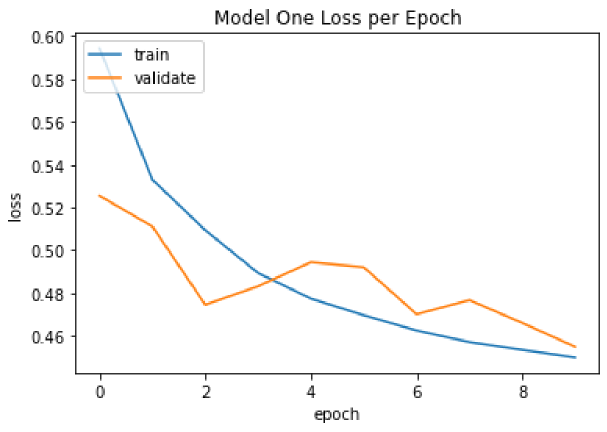
In [18]:
```
EPOCHS = 10

# train the model
history_model_one = model_one.fit_generator(
                        train_generator,
                        epochs = EPOCHS,
                        validation_data = valid_generator)
```

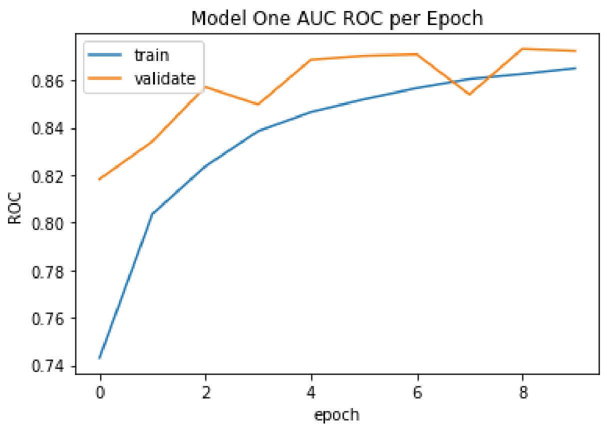
```
/opt/conda/lib/python3.7/site-packages/keras/engine/training.py:1972: UserWarning:

`Model.fit_generator` is deprecated and will be removed in a future version. Please use `Model.fit`, which supports generators.

2022-09-14 11:37:47.841212: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are e
nabled (registered 2)
Epoch 1/10
2022-09-14 11:37:51.316815: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369] Loaded cuDNN version 8005
731/731 [==============================] - 1218s 2s/step - loss: 0.5943 - accuracy: 0.6969 - auc: 0.7431 - val_loss: 0.5255 - val
_accuracy: 0.7610 - val_auc: 0.8183
Epoch 2/10
731/731 [==============================] - 292s 399ms/step - loss: 0.5330 - accuracy: 0.7471 - auc: 0.8036 - val_loss: 0.5112 -
val_accuracy: 0.7634 - val_auc: 0.8342
Epoch 3/10
731/731 [==============================] - 280s 384ms/step - loss: 0.5094 - accuracy: 0.7614 - auc: 0.8237 - val_loss: 0.4746 -
val_accuracy: 0.7893 - val_auc: 0.8571
Epoch 4/10
731/731 [==============================] - 291s 398ms/step - loss: 0.4894 - accuracy: 0.7747 - auc: 0.8385 - val_loss: 0.4832 -
val_accuracy: 0.7832 - val_auc: 0.8498
Epoch 5/10
731/731 [==============================] - 301s 412ms/step - loss: 0.4775 - accuracy: 0.7797 - auc: 0.8465 - val_loss: 0.4945 -
val_accuracy: 0.7737 - val_auc: 0.8686
Epoch 6/10
731/731 [==============================] - 284s 389ms/step - loss: 0.4697 - accuracy: 0.7848 - auc: 0.8519 - val_loss: 0.4921 -
val_accuracy: 0.7626 - val_auc: 0.8703
Epoch 7/10
731/731 [==============================] - 291s 398ms/step - loss: 0.4626 - accuracy: 0.7881 - auc: 0.8566 - val_loss: 0.4702 -
val_accuracy: 0.7847 - val_auc: 0.8709
Epoch 8/10
731/731 [==============================] - 285s 390ms/step - loss: 0.4572 - accuracy: 0.7922 - auc: 0.8605 - val_loss: 0.4768 -
val_accuracy: 0.7860 - val_auc: 0.8540
Epoch 9/10
731/731 [==============================] - 279s 382ms/step - loss: 0.4535 - accuracy: 0.7934 - auc: 0.8626 - val_loss: 0.4661 -
val_accuracy: 0.7881 - val_auc: 0.8732
Epoch 10/10
731/731 [==============================] - 279s 382ms/step - loss: 0.4500 - accuracy: 0.7970 - auc: 0.8650 - val_loss: 0.4550 -
val_accuracy: 0.7929 - val_auc: 0.8723
```

Now that we have trained the model, we can take at look at how it did graphically with the training data. Below we can see the accuracy and loss with regards to the validation and training set. We can also see how it did with the ROC AUC per epoch.

In [19]:
```
# plot model accuracy per epoch
plt.plot(history_model_one.history['accuracy'])
plt.plot(history_model_one.history['val_accuracy'])
plt.title('Model One Accuracy per Epoch')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
plt.show();

# plot model loss per epoch
plt.plot(history_model_one.history['loss'])
plt.plot(history_model_one.history['val_loss'])
plt.title('Model One Loss per Epoch')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
plt.show();

# plot model ROC per epoch
plt.plot(history_model_one.history['auc'])
plt.plot(history_model_one.history['val_auc'])
plt.title('Model One AUC ROC per Epoch')
plt.ylabel('ROC')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
plt.show();
```







Start second model building and training

Our second model incorporates hyperparameter tuning. We can see we are using ReLU activation functions for hidden layers, dropout between clusters, and batch normalization. In addition, we have added another hidden layer before the flatten and output layers. Lastly, we are using Adam optimizer with a low learning rate.

In [20]:
```
# build second model like first but with hyperparameters and optimizer(s)
ROC_2 = tf.keras.metrics.AUC()

with strategy.scope():

    #create model
    model_two = Sequential()

    model_two.add(Conv2D(filters=16, kernel_size=(3,3), activation='relu', ))
    model_two.add(Conv2D(filters=16, kernel_size=(3,3), activation='relu'))
    model_two.add(MaxPooling2D(pool_size=(2,2)))
    model_two.add(Dropout(0.1))

    model_two.add(BatchNormalization())
    model_two.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
    model_two.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
    model_two.add(AveragePooling2D(pool_size=(2,2)))
    model_two.add(Dropout(0.1))

    model_two.add(BatchNormalization())
    model_two.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
    model_two.add(Flatten())
    model_two.add(Dense(1, activation='sigmoid'))
    #build model by input size
    model_two.build(input_shape=(BATCH_SIZE, 64, 64, 3))        # original image = (96, 96, 3)

    #compile
    adam_optimizer = Adam(learning_rate=0.0001)
    model_two.compile(loss='binary_crossentropy', metrics=['accuracy', ROC_2], optimizer=adam_optimizer)

#quick look at model
model_two.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (256, 62, 62, 16)         448
_________________________________________________________________
conv2d_5 (Conv2D)            (256, 60, 60, 16)         2320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (256, 30, 30, 16)         0
_________________________________________________________________
dropout (Dropout)            (256, 30, 30, 16)         0
_________________________________________________________________
batch_normalization (BatchNo (256, 30, 30, 16)         64
_________________________________________________________________
conv2d_6 (Conv2D)            (256, 28, 28, 32)         4640
_________________________________________________________________
conv2d_7 (Conv2D)            (256, 26, 26, 32)         9248
_________________________________________________________________
average_pooling2d_1 (Average (256, 13, 13, 32)         0
_________________________________________________________________
dropout_1 (Dropout)          (256, 13, 13, 32)         0
_________________________________________________________________
batch_normalization_1 (Batch (256, 13, 13, 32)         128
_________________________________________________________________
conv2d_8 (Conv2D)            (256, 11, 11, 32)         9248
_________________________________________________________________
flatten_1 (Flatten)          (256, 3872)               0
_________________________________________________________________
dense_1 (Dense)              (256, 1)                  3873
=================================================================
Total params: 29,969
Trainable params: 29,873
Non-trainable params: 96
_________________________________________________________________
```

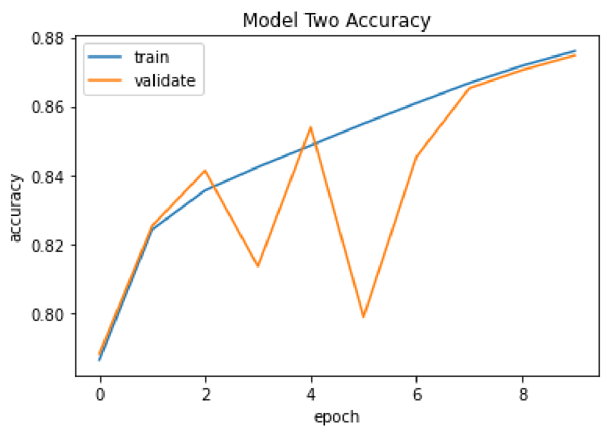
In [21]:
```
EPOCHS = 10

# train model
history_model_two = model_two.fit_generator(
                            train_generator,
                            epochs = EPOCHS,
                            validation_data = valid_generator)
```

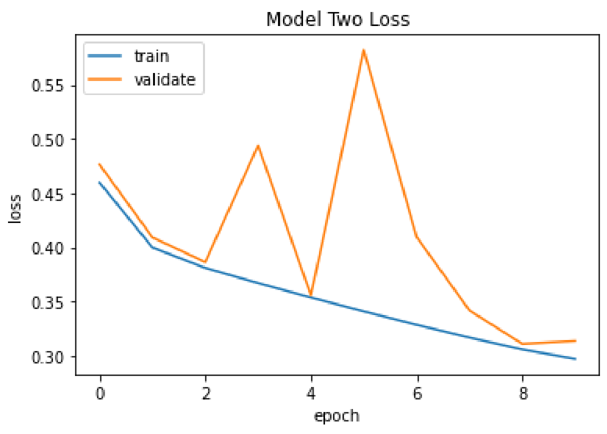
```
Epoch 1/10
731/731 [==============================] - 285s 388ms/step - loss: 0.4597 - accuracy: 0.7864 - auc_1: 0.8575 - val_loss: 0.4764
- val_accuracy: 0.7882 - val_auc_1: 0.8932
Epoch 2/10
731/731 [==============================] - 286s 391ms/step - loss: 0.3998 - accuracy: 0.8242 - auc_1: 0.8942 - val_loss: 0.4092
- val_accuracy: 0.8253 - val_auc_1: 0.9065
Epoch 3/10
731/731 [==============================] - 292s 400ms/step - loss: 0.3809 - accuracy: 0.8356 - auc_1: 0.9052 - val_loss: 0.3864
- val_accuracy: 0.8413 - val_auc_1: 0.9178
Epoch 4/10
731/731 [==============================] - 331s 453ms/step - loss: 0.3670 - accuracy: 0.8423 - auc_1: 0.9131 - val_loss: 0.4940
- val_accuracy: 0.8135 - val_auc_1: 0.9170
Epoch 5/10
731/731 [==============================] - 290s 396ms/step - loss: 0.3537 - accuracy: 0.8486 - auc_1: 0.9196 - val_loss: 0.3557
- val_accuracy: 0.8539 - val_auc_1: 0.9280
Epoch 6/10
731/731 [==============================] - 285s 390ms/step - loss: 0.3409 - accuracy: 0.8549 - auc_1: 0.9257 - val_loss: 0.5822
- val_accuracy: 0.7988 - val_auc_1: 0.9163
Epoch 7/10
731/731 [==============================] - 286s 391ms/step - loss: 0.3285 - accuracy: 0.8609 - auc_1: 0.9312 - val_loss: 0.4095
- val_accuracy: 0.8453 - val_auc_1: 0.9251
Epoch 8/10
731/731 [==============================] - 287s 392ms/step - loss: 0.3167 - accuracy: 0.8667 - auc_1: 0.9359 - val_loss: 0.3417
- val_accuracy: 0.8652 - val_auc_1: 0.9325
Epoch 9/10
731/731 [==============================] - 284s 388ms/step - loss: 0.3057 - accuracy: 0.8717 - auc_1: 0.9400 - val_loss: 0.3109
- val_accuracy: 0.8705 - val_auc_1: 0.9406
Epoch 10/10
```

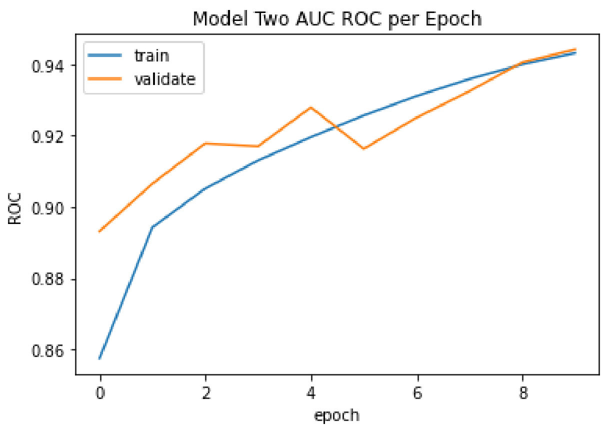
```
731/731 [==============================] - 285s 390ms/step - loss: 0.2970 - accuracy: 0.8760 - auc_1: 0.9432 - val_loss: 0.3136
- val_accuracy: 0.8748 - val_auc_1: 0.9442
```

In [22]:

```
# graph loss
plt.plot(history_model_two.history['accuracy'])
plt.plot(history_model_two.history['val_accuracy'])
plt.title('Model Two Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
plt.show();

plt.plot(history_model_two.history['loss'])
plt.plot(history_model_two.history['val_loss'])
plt.title('Model Two Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
plt.show();

# plot model ROC per epoch
plt.plot(history_model_two.history['auc_1'])
plt.plot(history_model_two.history['val_auc_1'])
plt.title('Model Two AUC ROC per Epoch')
plt.ylabel('ROC')
plt.xlabel('epoch')
plt.legend(['train', 'validate'], loc='upper left')
plt.show();
```







Test final model against the test set

Now that we have a trained model, we can test it on the unseen test data images. We must also normalize the test data like we did with the training data. Then we run the model to find its predictions. Let's hope it does well in the competition!

In [23]:
```
#double check what you're aiming the submission data set to look like
sample_submission.head()
```

Out[23]:

	id	label
0	0b2ea2a822ad23fdb1b5dd26653da899fbd2c0d5	0
1	95596b92e5066c5c52466c90b69ff089b39f2737	0
2	248e6738860e2ebcf6258cdc1f32f299e0c76914	0
3	2c35657e312966e9294eac6841726ff3a748febf	0
4	145782eb7caa1c516acbe2eda34d9a3f31c41fd6	0

In [24]:
```
#create a dataframe to run the predictions
test_df = pd.DataFrame({'id':os.listdir(test_path)})
test_df.head()
```

Out[24]:

	id
0	a7ea26360815d8492433b14cd8318607bcf99d9e.tif
1	59d21133c845dff1ebc7a0c7cf40c145ea9e9664.tif
2	5fde41ce8c6048a5c2f38eca12d6528fa312cdbb.tif
3	bd953a3b1db1f7041ee95ff482594c4f46c73ed0.tif
4	523fc2efd7aba53e597ab0f69cc2cbded7a6ce62.tif

In [25]:
```
# prepare test data (in same way as train data)
datagen_test = ImageDataGenerator(rescale=1./255.)

test_generator = datagen_test.flow_from_dataframe(
    dataframe=test_df,
    directory=test_path,
    x_col='id',
    y_col=None,
    target_size=(64,64),          # original image = (96, 96)
    batch_size=1,
    shuffle=False,
    class_mode=None)
```

```
Found 57458 validated image filenames.
```

In [26]:
```
#run model to find predictions

# predictions = model_one.predict(test_generator, verbose=1)
predictions = model_two.predict(test_generator, verbose=1)
```

```
57458/57458 [==============================] - 353s 6ms/step
```

In [27]:
```
#create submission dataframe
predictions = np.transpose(predictions)[0]
submission_df = pd.DataFrame()
submission_df['id'] = test_df['id'].apply(lambda x: x.split('.')[0])
submission_df['label'] = list(map(lambda x: 0 if x < 0.5 else 1, predictions))
submission_df.head()
```

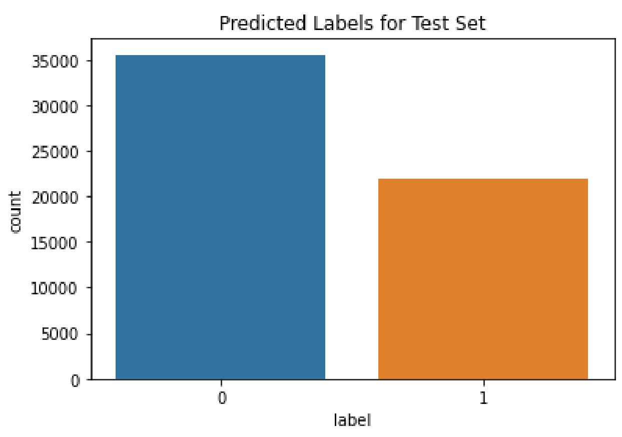
Out[27]:

	id	label
0	a7ea26360815d8492433b14cd8318607bcf99d9e	0
1	59d21133c845dff1ebc7a0c7cf40c145ea9e9664	0
2	5fde41ce8c6048a5c2f38eca12d6528fa312cdbb	0
3	bd953a3b1db1f7041ee95ff482594c4f46c73ed0	1
4	523fc2efd7aba53e597ab0f69cc2cbded7a6ce62	0

In [28]:
```
#view test prediction counts
submission_df['label'].value_counts()
```

Out[28]:
```
0    35509
1    21949
Name: label, dtype: int64
```

In [29]:
```
#plot test predictions
sns.countplot(data=submission_df, x='label').set(title='Predicted Labels for Test Set');
```

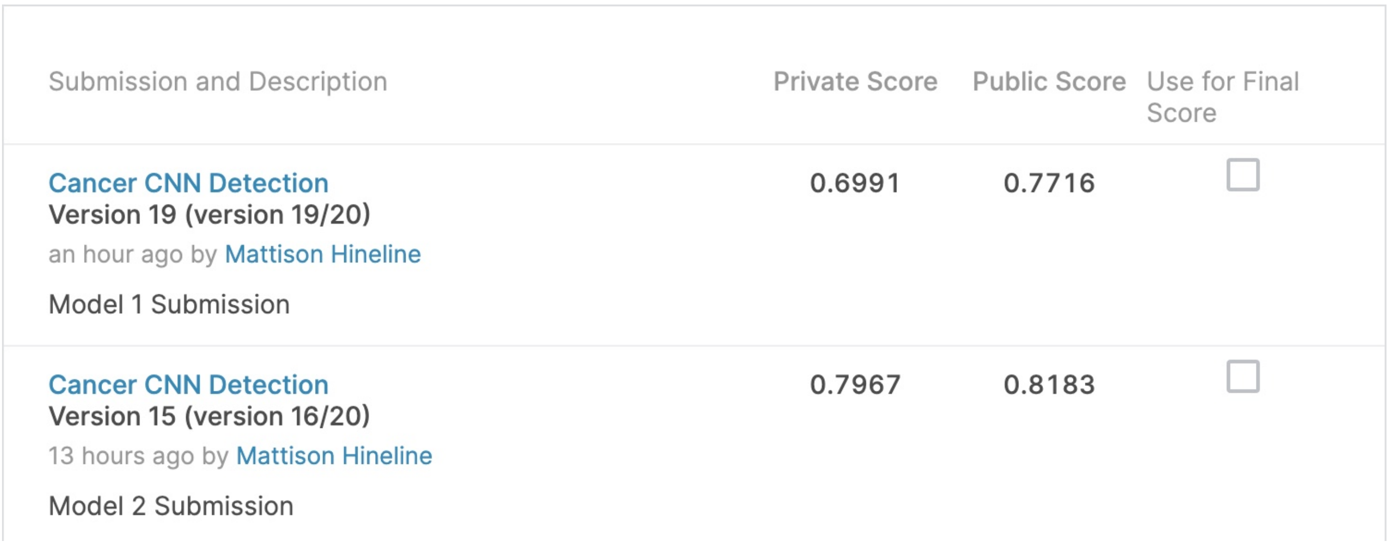


In [30]:
```
#convert to csv to submit to competition
submission_df.to_csv('submission.csv', index=False)
```

4. Results and Analysis

We can see from the above plots and diagrams for each model how well they performed with the training (and validation) sets. We see that model-one seemed to steady out a bit more than our more complex model (model-two) with regards to the ROC metric. We see in both models that the accuracy and loss does not steady, nor does the ROC in the second model. This could pertain towards the fact that we trained with very few epochs (10) and a simple CNN model with so many pictures may need more "time" to train to converge.

After submitting both trained models separately on the test set, we can see (below) how each model performed. As expected, we see that model-one (the model without hyperparameters) performed worse than model-two (with hyperparameters). Model-two performed with a score of about 0.80 which is fairly good considering the models we created.



Submission and Description	Private Score	Public Score	Use for Final Score
Cancer CNN Detection Version 19 (version 19/20) an hour ago by Mattison Hineline Model 1 Submission	0.6991	0.7716	☐
Cancer CNN Detection Version 15 (version 16/20) 13 hours ago by Mattison Hineline Model 2 Submission	0.7967	0.8183	☐

5. Conclusion

Our first model was simple with no hyperparameter tuning. The second model incorporated much more tuning and a few extra layers. Both models trained for 10 epochs and performed resonably well given that they both are fairly simple. As expected, the second model did better than the first. We can see that hyperparameter tuning does indeed contribute to the model performance and can improve the model if done correctly.

Some ways to make the models better could be a variety of factors. First, I would suggest to train the model with augmented images. In the preprocessing step, we only normalized the images. Instead, you could normalize, flip, zoom in/out, stretch, rotate, etc. with the images so that the model learns more instances of how images can be shaped. However, this was not done in this notebook due to time and memory constraints. In addition to image augmentation, using more epochs could allow the model to learn better. However, it is important that we don't overfit the data by training it too long. Since this project is for demonstration purposes, we did not use more epochs either. Some other ideas to make a stronger model would be to (1) transfer learning where part of the model is taken from another well-trained model, (2) tune hyperparameters in different ways such as strides, filter size, activation functions, learning rate, etc, and (3) have more complex model with more layers. It is hard to say which modification to the notebook and/or model would return in much better results without actually doing it and running the models however, we are confident that these are good steps to try in another notebook.