

# Section 1: Cancer Detection Project

**Background:** Kaggle hosted a competition from November 16, 2018 to March 30, 2019. The challenge asked participants to create an algorithm for binary classification to identify images as either: having no cancerous tumor or as having a cancerous tumor.

**Description of Dataset:** The PatchCamelyon (PCam) dataset contains color images which are 96 by 96 pixels of lymph node biopsies. The images include a label identifying whether the tissue contained tumor tissue or not. A label of one denotes the presence of a cancerous tumor -- whereas zero denotes no cancerous tissue. The original dataset consists of 327680 instances. The dataset posted on Kaggle was modified from the original to remove duplicates. The training dataset posted on Kaggle contained over 220000 images in .tif format with associated labels file in .csv format. The test dataset contains over 57500 images in .tif format.

**Date:** 2018

**Citation:** Will Cukierski. Histopathologic Cancer Detection. <https://kaggle.com/competitions/histopathologic-cancer-detection>, 2018. Kaggle.

**Link:** <https://www.kaggle.com/competitions/histopathologic-cancer-detection/overview>

**Project Approach:** For this assignment, we were asked to utilize the PCam dataset summarized above to conduct analysis of the images and build CNN models for binary classification. We plan to split the data into training and validation sets, in order to monitor the performance of these models on the validation data -- which will not be used in the model training. The predictions from our best performing model can then be submitted to the Kaggle competition page, where we will screenshot the submission as asked for this project. We will plot the AUC curve for this model, as asked by the Kaggle competition.

Since we have validation data, we can compare the results across models. In particular, we will look at comparisons of the Accuracy and Loss on the validation data across the models. The best performing model(s) will be identified, as well as the associated hyperparameters.

**Github Link:** [https://github.com/chiffr3/CNN\\_project](https://github.com/chiffr3/CNN_project)

## 1.1 About the Data:

Table 1

Explanation of Data Fields with Description of each Variable

Variable	Description	Data Type
<b>Id</b>	The identification number associated with the image	Alphanumeric
<b>Label</b>	The classification of the image. Options: 0 (no tumor), 1 (tumor)	Integer

Note: As shown in Table 1 above, the labels dataset contains 2 variables of which 1 is alphanumeric and the other is numeric. There are 220025 rows of training data.

## 1.2 Evaluation Metrics

Our primary task is binary image classification. Therefore, we will use two main tools:

- **Classification Report:** This report shows Accuracy, Precision, Recall and F1 scores for the classes. In particular, we will analyze the Accuracy and F1 scores. Accuracy indicates the overall percentage of correct predictions, but can be skewed by imbalanced data. The F1 score blends the Precision and Recall scores. In the case of "tumor", Precision indicates the percentage of our predictions that were actually "cancerous" -- whereas Recall indicates the percentage of "tumors" that we correctly identified. Ideally, we want a model that predicts well across either "tumor" or "no tumor" categories.
- **Confusion Matrix:** This matrix shows the numbers of correctly identified and misidentified classes. These values are also referred to as: True Positive, True Negative, False Positive and False Negative. A Type I error is a False Positive and Type II error is a False Negative. For example, a False Positive value indicates how many positive cancerous tumors were incorrectly predicted as negative according to the model. Again, ideally, we want a model that predicts well across either "tumor" or "no tumor" categories.

## Summary of Results

Here is a summary of the top Accuracy scores from models on the Validation Data. We will walk through a number of these models below. The results from the top model is shown in blue.

Table 2

Summary of Results for top Models on Validation Data

Model	Accuracy %	Filter Layers	Kernel Initializer	Normalization	Dropout	Final Activation	Optimizer
Model 1	90.56	5	Glorot Uniform	None	Multiple	Softmax	Adam
Model 2	92.75	5	He Normal	Batch	Single	Softmax	Adam
Model 3	93.62	6	He Normal	Batch	None	Softmax	AdamW
Model 6	88.79	4	He Normal	Batch	Multiple	Sigmoid	Adam
Model 7	88.97	4	He Normal	Batch	Multiple	Softmax	Adam
Model 8	89.56	4	He Normal	Batch	Multiple	Sigmoid	RMS Prop
Model 9	89.33	4	He Normal	Batch	Multiple	Softmax	Adam
Model 11*	93.69	5	Glorot Uniform	None	Multiple	Softmax	Adam

Note: Model 11 contained the same parameters as Model 1, but was analyzed with a smaller set of Validation Data

### Highlights from Table 2:

- Similar Accuracy scores were attained from different models utilizing various: initializers, with or without normalization, dropout, max pooling, final activation methods, as well as optimizers.
- The highest Accuracy scores noted in blue above were from "Model 3", which also scored best on the Kaggle test data.
- As noted in Table 2, Model 11 actually contained the same parameters as Model 1. However, Model 11 ran on a smaller set of Validation Data, which is why the Accuracy score is in red. Models 6 through 9 were also executed on smaller sets of Validation Data.

- The Filter Layers refers to the number of layers of filters in the model. These filters progressively increased from 32 to 64 to 128, etc.
- The execution time varied across these models and hyperparameters.

In summary, the size of the Validation Data set impacted the results. Therefore, a number of key models were re-run with a larger Validation Data size to investigate the results. Those models performed better on the Kaggle test scores and, as a result, are shown below.

## Section 2: Import Packages and Load Data

```
In [51]: # Load packages
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report, roc_curve, auc, roc_auc_score
import os
import shutil
import cv2
import re
```

```
In [61]: # Import Tensorflow/Keras packages
import tensorflow as tf
from tensorflow import keras
from keras.layers import Dense, Dropout, Activation, Flatten, Conv2D, MaxPooling2D, BatchNormalization
from keras.models import Sequential
from keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint
from keras.regularizers import l2
from keras.optimizers import Adam, RMSprop, AdamW
from keras.models import load_model
from keras.src.legacy.preprocessing.image import ImageDataGenerator
```

### Load Training Labels

```
In [49]: # Unzip files
import zipfile

#with zipfile.ZipFile('cancer_images/train.zip', 'r') as zip_ref:
#    zip_ref.extractall('cancer_images/train')

#with zipfile.ZipFile('cancer_images/test.zip', 'r') as zip_ref:
#    zip_ref.extractall('cancer_images/test')
```

```
In [49]: #train_dir = 'cancer_images/train'
#test_dir = 'cancer_images/test'
```

```
In [71]: # Load the training labels file
label_df = pd.read_csv('train_labels.csv')
```

## Section 3: Exploratory Data Analysis

To start, let's check the labels data to see what it contains and view the id format.

```
In [8]: # View head of labels
label_df.head()
```

```
Out[8]:
```

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

Here we see the id is alphanumeric and the label is simply zero or one. Next, let's check the number of entries, columns, data types and non-null counts.

```
In [11]: # Check the info
label_df.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
```

Here we see over 220 thousand rows, no null values and the data types. Let's check the describe view to see the range of values.

```
In [9]: # Check the describe view
label_df.describe()
```

```
Out[9]:
```

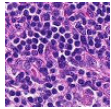
	label
count	220025.000000
mean	0.405031
std	0.490899
min	0.000000
25%	0.000000
50%	0.000000
75%	1.000000
max	1.000000

**Note:** Here we see the 50% quantile of data is labeled 0 and the mean is less than 0.50; therefore, it appears the data is not evenly split. We want to check the counts of our data below.

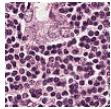
First, let's view some images and see what this data entails.

## View Images

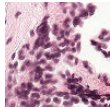
Let's view some images for a visualization of the data involved in this project. First, we can randomly sample an image from the training data.



The image below was labeled as tumor.



Meanwhile, the following image was labeled as no tumor.



While it is highly unlikely we would discern the cancerous tumor, given our lack of expertise reading the images, it does provide a sense of the data and the difficulty in detection. There was an issue flagged with one image which is black. Let's see if that image is viewable or needs to be removed from the data.



That image does not appear readable since it is entirely black. Let's remove that image from the dataset below.

**Please note:** In Section 4: Image Generators and Normalization, Image Data Generator is utilized for Normalization as well as shifting or rotating images.

Next, let's explore balancing options since it appears we have more data labeled non-cancerous than cancerous.

## Check for balanced data

Next, let's check the labels to see if we have a balanced dataset with similar numbers of cancerous and non-cancerous labels. To start, we will look at the counts of each label as 0 (no tumor) or 1 (tumor). Then, we can view a visualization of the comparison.

```
In [12]: #get count of each label
label_df.label.value_counts()
```

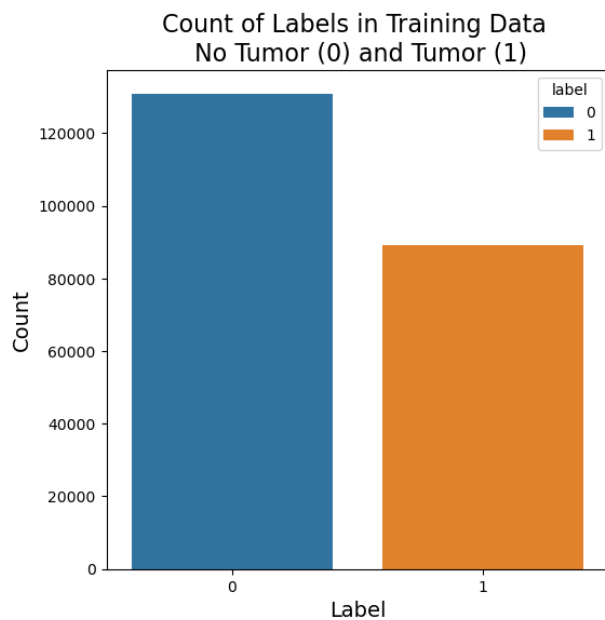
```
Out[12]:
```

label	
0	130908
1	89117
Name: count, dtype: int64	

For ease of comparison, let's view a visualization of these counts below.

**Figure 1: Count of Labels in Training Data**

```
In [13]: # View split of labels
plt.figure(figsize = (6,6))
sns.countplot(x = 'label', data = label_df, hue = 'label')
plt.title("Count of Labels in Training Data \n No Tumor (0) and Tumor (1)", size = 16)
plt.xlabel("Label", size = 14)
plt.ylabel("Count", size = 14)
plt.show()
```



**Summary: Consider balancing data, given difference in size of labeled data (0) versus (1)**

Since this data is not balanced, a model using all of these labels would likely predict better on the 0 class which is most represented in this dataset. However, our goal is to also predict the cancerous 1 class. Therefore, we will explore balancing options below to ensure we can predict both with a fairly high degree of accuracy.

## Section 4: Data Preprocessing

### Balance the Data

Since we have fewer images labeled as tumor than not labeled as tumor, we will use all of the data containing the cancerous tumor labels and sample the non-cancerous images in equal size to the cancerous. The intent is to have a dataset with equal numbers of cancerous and non-cancerous images so we can predict well on both.

First, let's remove the picture that was black from the dataset. It was labeled as non-cancerous and we have ample non-cancerous images in the dataset; therefore, removing a single image will not impact the results.

```
In [33]: # Check for picture that is all black
label_df[label_df['id'] == '9369c7278ec8bcc6c880d99194de09fc2bd4efbe']
```

```
Out[33]:
```

	id	label
138245	9369c7278ec8bcc6c880d99194de09fc2bd4efbe	0

```
In [33]: # Remove from the label df0
label_df = label_df.drop(label_df[label_df['id'] == '9369c7278ec8bcc6c880d99194de09fc2bd4efbe'].index)
```

```
In [33]: # Confirm it was dropped
label_df[label_df['id'] == '9369c7278ec8bcc6c880d99194de09fc2bd4efbe']
```

```
Out[33]:
```

	id	label
--	----	-------

Now, we can proceed with creating a balanced dataset. We will use a sample size of 88000 and get equal quantities of images from each category.

```
In [33]: # Need sample size to be even number so we have equal numbers of data with tumor/no tumor
sample_size = 88000
```

```
# Use most of label 1, sample label 0 in equal size to label 1
label_df1 = label_df[label_df['label'] == 1].sample(sample_size, random_state = 42)

label_df0 = label_df[label_df['label'] == 0].sample(sample_size, random_state = 42)
```

```
In [33]: # check the size of each
print("Size of Label 0:", len(label_df0))
print("Size of Label 1:", len(label_df1))
```

```
Size of Label 0: 88000
Size of Label 1: 88000
```

```
In [34]: # Concatenate to a balanced dataframe and shuffle
label_df_bal = pd.concat([label_df0, label_df1], axis = 0).reset_index(drop = True).sample(frac = 1)

# check
label_df_bal.label.value_counts()
```

```
Out[34]:
```

label	count
1	88000
0	88000

Name: count, dtype: int64

```
In [41]: # Check that df is shuffled
label_df_bal.head(5)
```

Out[41]		id	label
	77111	26c47d79e2328715beac8cb6751f9fa8c11f5444	0
	103552	5faab99d1a12b6ea294fbe858fc5bd45c6f0b792	1
	101728	df4ebd0cb25c950801a9938bbe49aed058c3af5d	1
	28427	3c2e25c33947dd6bc2a89dff70949045db2e8e9e	0
	67862	1e0aaf20f82264f2cd122c7f2db2f3bc32ce503f	0

```
In [42]: # Create train and validation sets
y = label_df_bal['label'].copy()

train_df, valid_df = train_test_split(label_df_bal, test_size = 0.15, random_state = 42, stratify = y)
```

```
In [43]: # confirm counts
print("Train label counts:", train_df.label.value_counts())
print("Validation label counts:", valid_df.label.value_counts())
```

```
Train label counts: label
1    74800
0    74800
Name: count, dtype: int64
Validation label counts: label
1    13200
0    13200
Name: count, dtype: int64
```

Figure 2: Count of Labels in Balanced Training Data

```
In [44]: # View split of labels
plt.figure(figsize = (6,6))
tr = sns.countplot(x = 'label', data = train_df, hue = 'label')
sns.move_legend(tr, "upper right", bbox_to_anchor = (1.2, 1.0), frameon = True)
plt.title("Count of Labels in Balanced Training Data \n No Tumor (0) and Tumor (1)", size = 16)
plt.xlabel("Label", size = 14)
plt.ylabel("Count", size = 14)
plt.show()
```

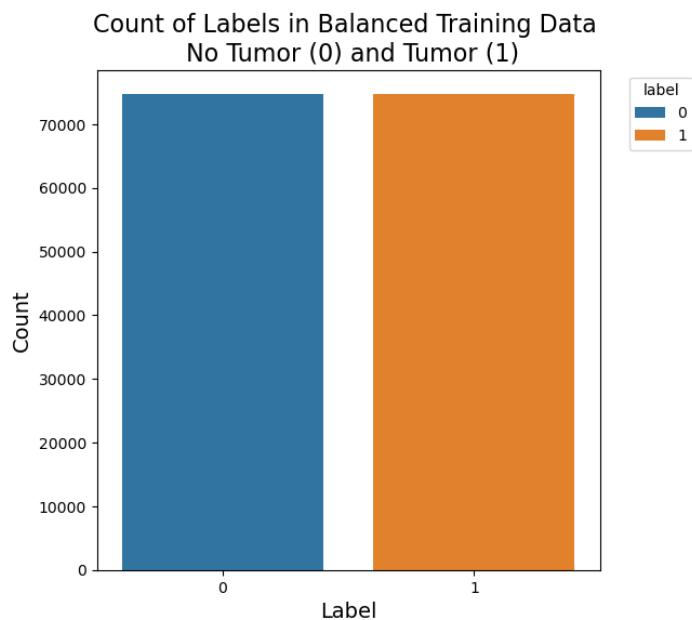
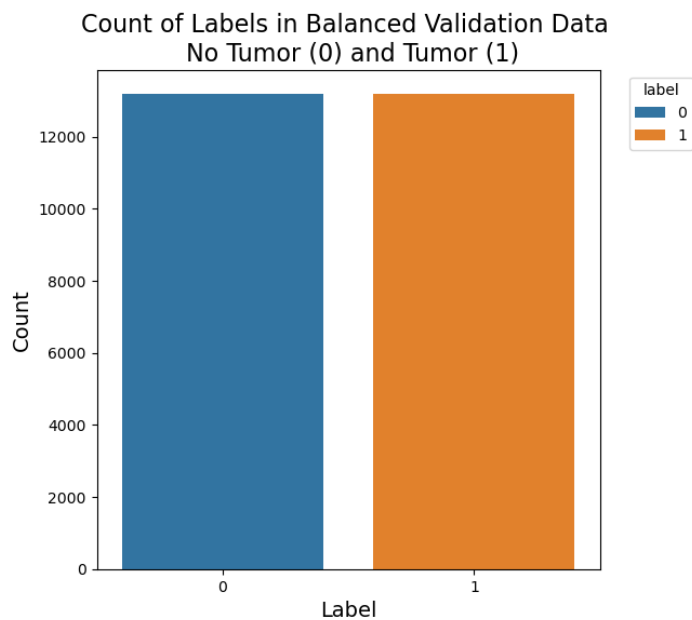


Figure 3: Count of Labels in Balanced Validation Data

```
In [45]: # View split of labels
plt.figure(figsize = (6,6))
v = sns.countplot(x = 'label', data = valid_df, hue = 'label')
sns.move_legend(v, "upper right", bbox_to_anchor = (1.2, 1.0), frameon = True)
plt.title("Count of Labels in Balanced Validation Data \n No Tumor (0) and Tumor (1)", size = 16)
plt.xlabel("Label", size = 14)
plt.ylabel("Count", size = 14)
plt.show()
```



## Create directories

```
In [46]: # Create project directory
proj_dir = 'proj_dir'
os.mkdir(proj_dir)

In [47]: # Create new folders for training and validation

train_dir = os.path.join(proj_dir, 'train_dir')
os.mkdir(train_dir)

valid_dir = os.path.join(proj_dir, 'valid_dir')
os.mkdir(valid_dir)

# Create separate folders inside train for different labels
no_tumor = os.path.join(train_dir, 'no_tumor')
os.mkdir(no_tumor)

tumor = os.path.join(train_dir, 'tumor')
os.mkdir(tumor)

# Create separate folders inside validation for different labels
no_tumor = os.path.join(valid_dir, 'no_tumor')
os.mkdir(no_tumor)

tumor = os.path.join(valid_dir, 'tumor')
os.mkdir(tumor)

In [48]: # Check folders exist
print("Train_dir folders:", os.listdir('proj_dir/train_dir'))
print("Valid_dir folders:", os.listdir('proj_dir/valid_dir'))

Train_dir folders: ['no_tumor', 'tumor']
Valid_dir folders: ['no_tumor', 'tumor']
```

## Test Directory

```
In [49]: # Check test folder
test_sz = len(os.listdir('cancer_images/test'))
print("Test folder size:", test_sz)

Test folder size: 57459

In [50]: # Create test directory
test_dir = 'test_dir'
os.mkdir(test_dir)

# prepare to put images in 'test_imgs' in test_dir
test_imgs = os.path.join(test_dir, 'test_imgs')
os.mkdir(test_imgs)

In [51]: # Check test directory
os.listdir('test_dir')

Out[51]: ['test_imgs']

In [52]: # Check folders exist
print("Test_dir folders:", os.listdir('test_dir'))

Test_dir folders: ['test_imgs']
```

## Transfer images into new folders

```

In [53]: # Check column names and count
label_df_bal.info()

<class 'pandas.core.frame.DataFrame'>
Index: 176000 entries, 77111 to 26151
Data columns (total 2 columns):
 #   Column   Non-Null Count  Dtype
---  --
 0    id     176000 non-null  object
 1   label   176000 non-null  int64
dtypes: int64(1), object(1)
memory usage: 4.0+ MB

In [54]: # Index is set as 'id' in label df
label_df_bal.set_index('id', inplace = True)

In [55]: # Get list of image numbers
train_img = list(train_df['id'])
valid_img = list(valid_df['id'])

In [56]: # Check files listdir
os.listdir('cancer_images')

Out[56]: ['.DS_Store', 'test', 'test.zip', 'train', 'train.zip']

In [57]: # Transfer images for training
for image in train_img:
    # add .tif to filename
    fname = image + '.tif'
    # get label for image
    target = label_df_bal.loc[image, 'label']
    # set target for each folder no_tumor and tumor
    if target == 0:
        label = 'no_tumor'
    # else, since df only has 0 & 1
    else:
        label = 'tumor'
    # source path
    source_pth = os.path.join('cancer_images/train', fname)
    # destination path to image
    dest_pth = os.path.join(train_dir, label, fname)
    # transfer images
    shutil.copyfile(source_pth, dest_pth)

In [58]: # Transfer images for validation
for image in valid_img:
    # add .tif to filename
    fname = image + '.tif'
    # get label for image
    target = label_df_bal.loc[image, 'label']
    # set target for each folder no_tumor and tumor
    if target == 0:
        label = 'no_tumor'
    # else, since df only has 0 & 1
    else:
        label = 'tumor'
    # source path
    source_pth = os.path.join('cancer_images/train', fname)
    # destination path to image
    dest_pth = os.path.join(valid_dir, label, fname)
    # transfer images
    shutil.copyfile(source_pth, dest_pth)

In [59]: # Check number of images in each folder
print("Train folder, no tumor:", len(os.listdir('proj_dir/train_dir/no_tumor')))
print("Train folder, tumor:", len(os.listdir('proj_dir/train_dir/tumor')))

Train folder, no tumor: 74800
Train folder, tumor: 74800

In [60]: # Check number of images in each folder
print("Validation folder, no tumor:", len(os.listdir('proj_dir/valid_dir/no_tumor')))
print("Validation folder, tumor:", len(os.listdir('proj_dir/valid_dir/tumor')))

Validation folder, no tumor: 13200
Validation folder, tumor: 13200

In [61]: # Check number of images in test folder
test_sz = len(os.listdir('cancer_images/test'))
print("Test folder size:", test_sz)

Test folder size: 57459

```

## Image Generators and Normalization

Here we can use the Image Data Generator to prepare the images, since the Tensorflow `image_dataset_from_directory` function does not support .tif files.

**Normalization:** Using the `rescale` argument, we can multiply the pixel by `1.0/255` to normalize to a range of 0 to 1.

**Data Augmentation:** We can also use `width shift range`, `height shift range` to randomly shift the images. While `horizontal flip` and `vertical flip` set to true will randomly flip the images.

```

In [62]: # Create paths to each
train_pth = 'proj_dir/train_dir'
valid_pth = 'proj_dir/valid_dir'
test_pth = 'cancer_images/test'

In [63]: # Set samples and batch sizes

```

```
img_sz = 96

train_sample_sz = len(train_df)
valid_sample_sz = len(valid_df)
print("Train sample size:", train_sample_sz, "\nValid sample size:", valid_sample_sz)

# Set batch size
train_batch_sz = 10
valid_batch_sz = 10

Train sample size: 149600
Valid sample size: 26400
```

```
In [44]: # Create instance of image data generator, rescale/normalize and randomly shift + flip images
idg = ImageDataGenerator(rescale = 1.0/255,
                          width_shift_range = 8,
                          height_shift_range = 8,
                          horizontal_flip = True,
                          vertical_flip = True)
```

```
In [45]: # Prepare train and validation datasets
train_gen = idg.flow_from_directory(train_pth,
                                    target_size = (img_sz, img_sz),
                                    batch_size = train_batch_sz,
                                    class_mode = 'categorical')

valid_gen = idg.flow_from_directory(valid_pth,
                                    target_size = (img_sz, img_sz),
                                    batch_size = valid_batch_sz,
                                    class_mode = 'categorical')
```

Found 149600 images belonging to 2 classes.  
Found 26400 images belonging to 2 classes.

**Note:** Below we create the test generator with the validation set so we can view Evaluation Metrics for the models.

Later, for the actual predictions on the test data, we will reset the path in order to get our predictions for the Kaggle submission.

```
In [46]: # Prepare test data - do not shuffle
test_gen = idg.flow_from_directory(valid_pth,
                                   target_size = (img_sz, img_sz),
                                   batch_size = 1,
                                   class_mode = 'categorical',
                                   shuffle = False)
```

Found 26400 images belonging to 2 classes.

## Section 5: Model Architecture and Preparation

The Visual Geometry Group (VGG) at Oxford University published multiple research papers on Convolutional Neural Networks (CNN). One of their CNN architectures for image recognition is referred to as VGG-16, since it contains 16 layers -- whereas another architecture is referred to as VGG-19, since it contains 19 layers.

The VGG-16 Architecture is depicted below. These layers are a combination of Convolution 2D, followed by MaxPooling, and Fully Connected (FC) Dense layers. Let's summarize the key components:

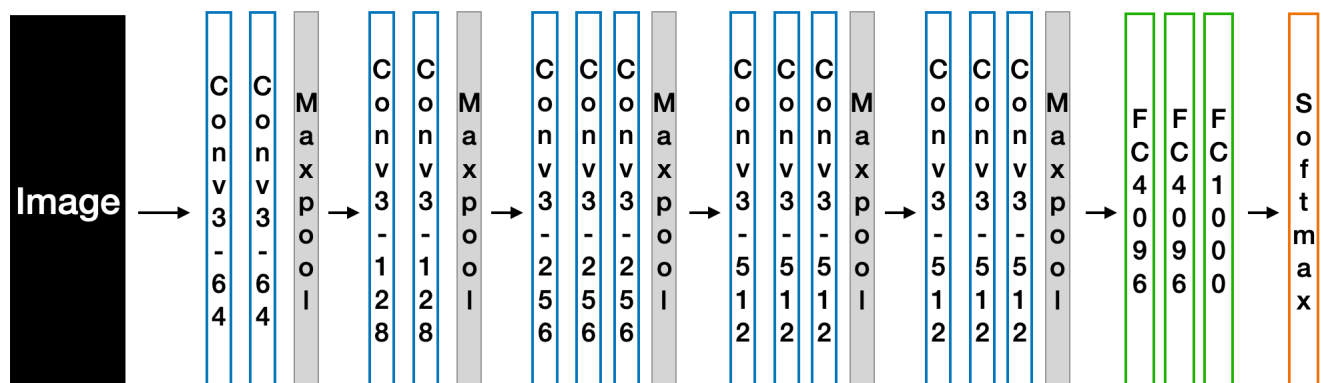
**Convolution 2D:** The number 3 indicates the 3x3 filter size and 64, 128, 256, 512 are the number of filters.

**MaxPooling:** After each block of Convolution 2D, there is Max Pooling.

**Fully Connected (FC):** The Fully Connected Dense Layers occur after the last block of Convolution 2D and Max Pooling.

**Activation:** While not shown due to space, the ReLU (Rectified Linear Unit) activation function is used with the Convolution layers. Meanwhile, the final activation is Softmax which is shown below.

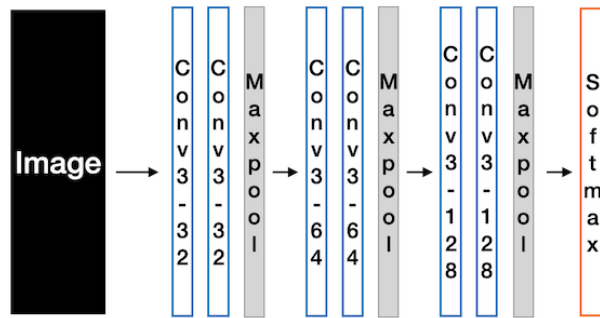
# VGG-16 Architecture



For the initial model, we started with a modified version of the VGG-16 Architecture. For simplicity, this was narrowed down into 3 blocks of Convolution 2D layers with MaxPooling, a single Dense layer and Softmax for the final activation. ReLu was used for our other activation layers. A diagram of this model is shown below.



# Initial Model



## Initial Model Details:

- **Kernel Size:** Here we start with a kernel size of 3x3 which is the size of each filter.
- **Pool Size:** Since MaxPooling is utilized, we set a pool size of 2x2. This operation will downsample or downscale along the height and width dimensions.
- **Number of Filters with each layer:** For images, the recommendation was to increase the number of filters per layer. Therefore, we start with three layers with the number of filters proceeding from 32 to 64 to 128.
- **Convolution 2D:** Since we have images, we use the Convolution 2D package with the above kernel size and filters. The default kernel initializer is glorot uniform.
- **Activation:** For each layer, we used the Rectified Linear Unit or 'relu' activation. Then, for the final activation, we use Softmax with 2 units of output which will provide a probability of two classes -- which are zero and one in this project.
- **Optimizer:** When we compile the model, we select the Adaptive Moment estimation or 'Adam' optimizer which tends to perform well and; therefore, is a good starting point. The default learning rate for Adam is 0.001.

While the results were good, there were some challenges with the number of epochs, learning rate, early stopping, and Validation Accuracy dropping as the model trained. Therefore, we experimented with different hyperparameters and methods to improve our results. Also, when those results were submitted to Kaggle, there was a significant decline in performance on the Kaggle test scores. As a result, we started over with a larger Validation Data set to improve our results.

## Building New Models with larger Validation Data set:

One key item noticed in our modeling was additional layers of increasing numbers of filters led to better performance. Rather than having multiple Convolution 2D layers together, these models used a single Convolution layer while increasing the filters throughout. Also, the multiple Convolution Layers plus Max Pooling significantly slowed the execution times of our models while not performing better on the Kaggle test scores. Instead, strides, Batch Normalization and Dropouts were used to try and improve our results while executing more quickly.

Now, let's prepare to build our new models. After prior experimentation, this section will be shortened to display the top models with explanation of the parameters and learnings from other modeling efforts. Some of the experimentation has included:

- Activation functions
- Batch Normalization
- Layer Normalization
- Optimizers
- Regularizers
- Number of layers

During this experimentation, Layer Normalization and L2 regularizer did not yield good results. Therefore, those methods are not shown below.

## Model 1: 5 Filters with Drop Out, but no normalization

**Kernel Size:** Here we start with a kernel size of 3x3 which is the size of each filter.

**Number of Filters with each layer:** For images, the recommendation was to increase the number of filters per layer. Therefore, we start with five layers with the number of filters proceeding from 32 -> 64 -> 128 -> 256 -> 512.

**Convolution 2D:** Since we have images, we use the Convolution 2D package with the above kernel size and filters. The default kernel initializer is glorot uniform. We also set padding to same, which keeps the same size from input to output.

**Dropout:** For each layer, after the Convolution 2D, we use a dropout of 25%. This Dropout randomly drops a percentage of the training dataset, which can help prevent overfitting.

**Activation:** For each layer, we will use the Rectified Linear Unit or 'relu' activation. Then, for the final activation, we use Softmax with 2 units of output which will provide a probability of two classes -- which are zero and one in this project.

**Optimizer:** When we compile the model, we select the Adaptive Moment estimation or 'Adam' optimizer which tends to perform well and; therefore, is a good starting point. Instead of the default learning rate, we use a smaller learning rate of 0.0001.

```
In [73]: # Set initial kernel size, and filters
kernel_sz = (3,3)

# Number of filters with each layer
filters_1 = 32
filters_2 = 64
filters_3 = 128
filters_4 = 256
filters_5 = 512
```

```
# Set the learning rate
optimizer = keras.optimizers.Adam(learning_rate = 0.0001)
```

```
In [74]: # Initiate model
model1 = Sequential()
```

```
In [75]: # Use strides instead of maxpooling, with Dropout but no normalization

model1.add(Conv2D(filters_1, kernel_sz, strides = (1,1), activation = 'relu', padding = 'same',
                  kernel_initializer = 'glorot_uniform'))
model1.add(Dropout(0.25))

model1.add(Conv2D(filters_2, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'glorot_uniform'))
model1.add(Dropout(0.25))

model1.add(Conv2D(filters_3, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'glorot_uniform'))
model1.add(Dropout(0.25))

model1.add(Conv2D(filters_4, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'glorot_uniform'))
model1.add(Dropout(0.25))

model1.add(Conv2D(filters_5, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'glorot_uniform'))
model1.add(Dropout(0.25))

model1.add(Flatten())
model1.add(Dense(units = 2, activation = 'softmax'))
```

```
In [76]: # Compile model
model1.compile(loss = 'binary_crossentropy', optimizer = optimizer, metrics = ['accuracy'])
```

```
In [77]: # Set a reduced learning rate and early stopping
drop_lr = ReduceLROnPlateau(monitor = 'val_accuracy', factor = 0.5, patience = 2, verbose = 1, mode = 'max',
                             min_lr = 0.00001)

early_stop = EarlyStopping(monitor = 'val_accuracy', mode = 'max', verbose = 1, patience = 2, restore_best_weights = True)

cb_lst = [drop_lr, early_stop]
```

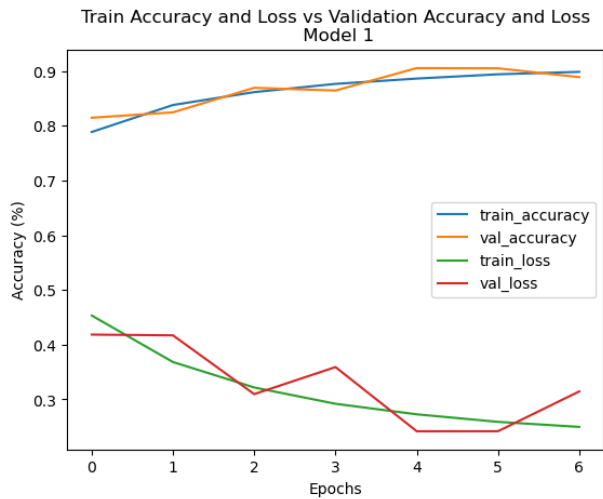
```
In [78]: history1 = model1.fit(train_gen,
                              validation_data = valid_gen,
                              epochs = 10,
                              verbose = 1,
                              callbacks = cb_lst)
```

```
Epoch 1/10
14960/14960 ————— 1350s 90ms/step - accuracy: 0.7445 - loss: 0.5046 - val_accuracy: 0.8148 - val_loss: 0.4184 - learning_rate: 1.0000
e-04
Epoch 2/10
14960/14960 ————— 1449s 97ms/step - accuracy: 0.8301 - loss: 0.3850 - val_accuracy: 0.8248 - val_loss: 0.4170 - learning_rate: 1.0000
e-04
Epoch 3/10
14960/14960 ————— 1237s 83ms/step - accuracy: 0.8553 - loss: 0.3321 - val_accuracy: 0.8696 - val_loss: 0.3095 - learning_rate: 1.0000
e-04
Epoch 4/10
14960/14960 ————— 1268s 85ms/step - accuracy: 0.8743 - loss: 0.2969 - val_accuracy: 0.8646 - val_loss: 0.3590 - learning_rate: 1.0000
e-04
Epoch 5/10
14960/14960 ————— 1527s 102ms/step - accuracy: 0.8843 - loss: 0.2783 - val_accuracy: 0.9056 - val_loss: 0.2417 - learning_rate: 1.0000
e-04
Epoch 6/10
14960/14960 ————— 1553s 104ms/step - accuracy: 0.8915 - loss: 0.2628 - val_accuracy: 0.9052 - val_loss: 0.2417 - learning_rate: 1.0000
e-04
Epoch 7/10
14960/14960 ————— 0s 100ms/step - accuracy: 0.8970 - loss: 0.2531
Epoch 7: ReduceLROnPlateau reducing learning rate to 4.999999873689376e-05.
14960/14960 ————— 1554s 104ms/step - accuracy: 0.8970 - loss: 0.2531 - val_accuracy: 0.8891 - val_loss: 0.3145 - learning_rate: 1.0000
e-04
Epoch 7: early stopping
Restoring model weights from the end of the best epoch: 5.
```

**Note:** After multiple prior experiments, this model was our best yet. Let's view the scores below.

```
In [79]: # View plot
model1_loss = pd.DataFrame(model1.history.history)
model1_loss = model1_loss.rename(columns = {'accuracy': 'train_accuracy', 'loss': 'train_loss'})
model1_loss[['train_accuracy', 'val_accuracy', 'train_loss', 'val_loss']].plot()
plt.title("Train Accuracy and Loss vs Validation Accuracy and Loss\n Model 1")
plt.xlabel("Epochs")
plt.ylabel("Accuracy (%)")
```

```
Out[79]: Text(0, 0.5, 'Accuracy (%)')
```



```
In [80]: max_val_acc1 = model1_loss.val_accuracy.max()
print("Maximum Validation Accuracy for Model:", round(max_val_acc1 * 100,2), "%")
```

Maximum Validation Accuracy for Model: 90.56 %

```
In [81]: min_val_loss1 = model1_loss.val_loss.min()
print("Minimum Validation Loss for Model:", round(min_val_loss1 * 100,2), "%")
```

Minimum Validation Loss for Model: 24.17 %

#### Summary:

- These are the best results yet with a maximum Validation Accuracy over 90%. Let's save this model.
- Next, we can view the Evaluation Metrics for this model.
- Let's see what happens with Batch Normalization and a new kernel initializer.
- We can also investigate the results with a new optimizer.
- After which, we can compare the results across models.

```
In [83]: # Save model
model1.save('cancer_project_model1.keras')
```

#### Evaluation Metrics for Model 1

```
In [85]: # Get predictions
yp_1 = model1.predict(test_gen, steps = len(valid_df), verbose = 1)
```

26400/26400 ————— 95s 4ms/step

```
In [86]: # Check predictions
yp_1
```

```
Out[86]: array([[9.9954408e-01, 4.5592090e-04],
 [9.5443135e-01, 4.5568623e-02],
 [9.9914896e-01, 8.5101317e-04],
 ...,
 [3.2082519e-01, 6.7917484e-01],
 [5.9385253e-05, 9.994063e-01],
 [1.3870365e-03, 9.9861300e-01]], dtype=float32)
```

```
In [87]: # Need to convert predictions to 0 or 1
yp1 = yp_1.argmax(axis = 1)
yp1
```

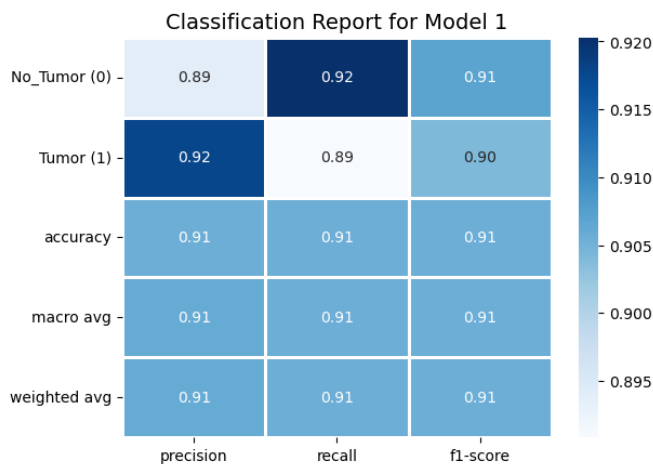
```
Out[87]: array([0, 0, 0, ..., 1, 1, 1])
```

Now that we have the predictions in the necessary format, let's view the Classification Report and Confusion Matrix below.

Figure 4: Classification Report for Model 1

```
In [19]: # Classification Report
test_labels = valid_gen.classes
names = ['No_Tumor (0)', 'Tumor (1)']
clf_test = classification_report(test_labels, yp1, target_names = names, output_dict = True)

# Heatmap view
sns.heatmap(pd.DataFrame(clf_test).iloc[:-1, :].T, annot = True, cmap = "Blues", linewidth = 1, fmt = '.2f')
plt.title("Classification Report for Model 1", fontsize = 14)
#plt.yticks(rotation = 0)
plt.show()
```



#### Summary of Classification Report for Model 1:

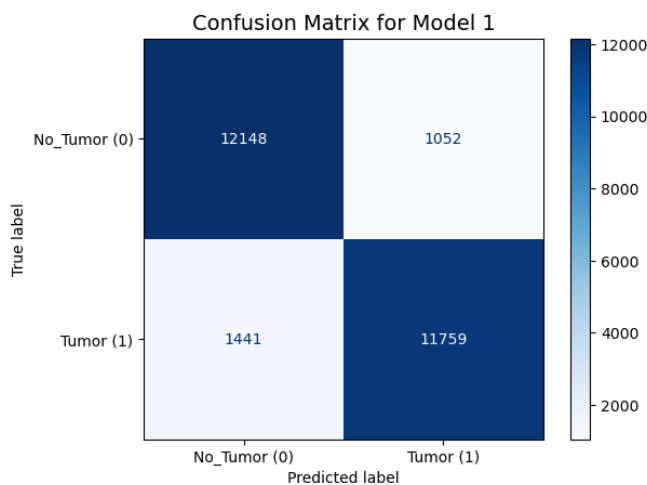
Here we see reasonably good results for the No Tumor (0) and Tumor (1) classes. The scores ranged from 89% to 92%. Although, we do note some difference between the Precision and Recall scores for the No Tumor versus Tumor classes.

Next, let's view the Confusion Matrix.

Figure 5: Confusion Matrix for Model 1

```
In [19]: # Get labels and store confusion
test_labels = valid_gen.classes
labels = ['No_Tumor (0)', 'Tumor (1)']

# Display confusion matrix
cm = confusion_matrix(test_labels, yp1)
cmd = ConfusionMatrixDisplay(cm, display_labels=labels)
cmd.plot(cmap = plt.cm.Blues)
plt.title('Confusion Matrix for Model 1', fontsize = 14)
plt.grid(False)
plt.show()
```



#### Summary of Confusion Matrix for Model 1:

Here we see the model did a better job predicting the "No Tumor" class than the "Tumor" class. There were hundreds more predicted to be "No Tumor (0)" which should have been "Tumor (1)". While 90.6% Accuracy score on the Validation Data is certainly good results, missing over 1400 tumor classifications is concerning. Therefore, we would like to find an even better performing model.

Let's see what we can do to improve upon these results.

## Model 2: 5 Filters with Batch Normalization

Instead of using Dropout following each Convolution 2D, let's see the results with Batch Normalization. We will use a single Dropout after the last Convolution 2D layer and between the Dense layers.

**Kernel Size:** Here we start with a kernel size of 3x3 which is the size of each filter.

**Number of Filters with each layer:** For images, the recommendation was to increase the number of filters per layer. Therefore, we have five layers with the number of filters proceeding from 32 -> 64 -> 128 -> 256 -> 512.

**Conv2D:** Since we have images, we use the Convolution 2D package with the above kernel size and filters. Here we use a different kernel initializer for the convolution kernel, he\_uniform. From prior investigation, it appeared the model may run faster with this initializer. We also set padding to same, which keeps the same size from input to output.

**Batch Normalization:** For each layer, after the Convolution 2D, we use Batch Normalization, which can help prevent overfitting.

**Dropout:** After the final Convolution 2D, we use a dropout of 50%. This Dropout randomly drops a percentage of the training dataset, which can help prevent overfitting.

**Activation:** For each layer, we will use the Rectified Linear Unit or 'relu' activation. Then, for the final activation, we use Softmax with 2 units of output which will provide a probability of two classes -- which are zero and one in this project.

**Optimizer:** When we compile the model, we select the Adaptive Moment estimation or 'Adam' optimizer which tends to perform well and; therefore, is a good starting point. Instead of the default learning rate, we use a smaller learning rate of 0.0001.

```
In [11]: # Set initial kernel size, and filters
kernel_sz = (3,3)

# Number of filters with each layer
filters_1 = 32
filters_2 = 64
filters_3 = 128
filters_4 = 256
filters_5 = 512

# Set the learning rate
optimizer = keras.optimizers.Adam(learning_rate = 0.0001)

In [12]: # Initiate model
model2 = Sequential()

In [13]: # Use strides instead of maxpooling, with Batch Normalization and final dropout

#Batch 1
model2.add(Conv2D(filters_1, kernel_sz, strides = (1,1), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model2.add(BatchNormalization())

#Batch 2
model2.add(Conv2D(filters_2, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model2.add(BatchNormalization())

#Batch 3
model2.add(Conv2D(filters_3, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model2.add(BatchNormalization())

#Batch 4
model2.add(Conv2D(filters_4, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model2.add(BatchNormalization())

#Batch 5
model2.add(Conv2D(filters_5, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model2.add(BatchNormalization())

# Flatten and convert to Dense layer
model2.add(Flatten())
model2.add(Dense(256, activation = 'relu'))
model2.add(Dropout(0.50))
model2.add(Dense(units = 2, activation = 'softmax'))

In [14]: # Compile model
model2.compile(loss = 'binary_crossentropy', optimizer = optimizer, metrics = ['accuracy'])

In [15]: # Set a reduced learning rate and early stopping
drop_lr = ReduceLROnPlateau(monitor = 'val_accuracy', factor = 0.5, patience = 2, verbose = 1, mode = 'max',
                             min_lr = 0.00001)

early_stop = EarlyStopping(monitor = 'val_accuracy', mode = 'max', verbose = 1, patience = 2, restore_best_weights = True)

cb_lst = [drop_lr, early_stop]

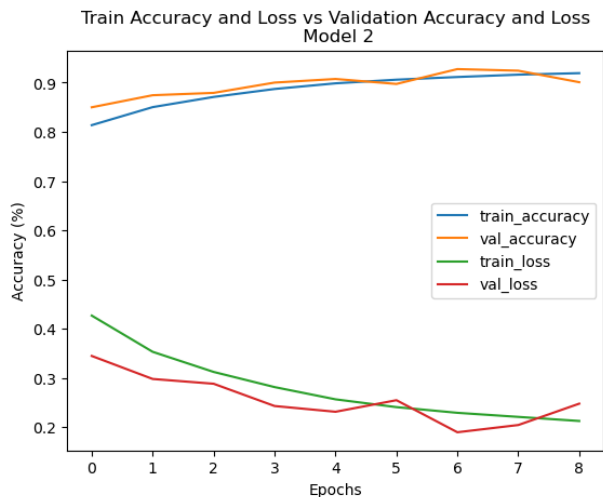
In [16]: history2 = model2.fit(train_gen,
                              validation_data = valid_gen,
                              epochs = 10,
                              verbose = 1,
                              callbacks = cb_lst)
```

```
Epoch 1/10
14960/14960 — 1626s 109ms/step — accuracy: 0.7911 — loss: 0.4742 — val_accuracy: 0.8500 — val_loss: 0.3449 — learning_rate: 1.0000
e-04
Epoch 2/10
14960/14960 — 1806s 121ms/step — accuracy: 0.8457 — loss: 0.3646 — val_accuracy: 0.8745 — val_loss: 0.2981 — learning_rate: 1.0000
e-04
Epoch 3/10
14960/14960 — 1534s 103ms/step — accuracy: 0.8646 — loss: 0.3227 — val_accuracy: 0.8792 — val_loss: 0.2884 — learning_rate: 1.0000
e-04
Epoch 4/10
14960/14960 — 1475s 99ms/step — accuracy: 0.8845 — loss: 0.2879 — val_accuracy: 0.9001 — val_loss: 0.2432 — learning_rate: 1.0000
e-04
Epoch 5/10
14960/14960 — 1716s 115ms/step — accuracy: 0.8954 — loss: 0.2644 — val_accuracy: 0.9075 — val_loss: 0.2315 — learning_rate: 1.0000
e-04
Epoch 6/10
14960/14960 — 1817s 121ms/step — accuracy: 0.9043 — loss: 0.2454 — val_accuracy: 0.8974 — val_loss: 0.2550 — learning_rate: 1.0000
e-04
Epoch 7/10
14960/14960 — 1824s 122ms/step — accuracy: 0.9094 — loss: 0.2337 — val_accuracy: 0.9275 — val_loss: 0.1899 — learning_rate: 1.0000
e-04
Epoch 8/10
14960/14960 — 1434s 96ms/step — accuracy: 0.9155 — loss: 0.2238 — val_accuracy: 0.9243 — val_loss: 0.2046 — learning_rate: 1.0000
e-04
Epoch 9/10
14960/14960 — 0s 114ms/step — accuracy: 0.9174 — loss: 0.2146
Epoch 9: ReduceLROnPlateau reducing learning rate to 4.99999873689376e-05.
14960/14960 — 1764s 118ms/step — accuracy: 0.9174 — loss: 0.2146 — val_accuracy: 0.9009 — val_loss: 0.2480 — learning_rate: 1.0000
e-04
Epoch 9: early stopping
Restoring model weights from the end of the best epoch: 7.
```

**Note:** After multiple prior experiments, this model was our best yet. Let's view the scores below.

```
In [12]: # View plot
model2_loss = pd.DataFrame(model2.history.history)
model2_loss = model2_loss.rename(columns = {'accuracy': 'train_accuracy', 'loss': 'train_loss'})
model2_loss[['train_accuracy', 'val_accuracy', 'train_loss', 'val_loss']].plot()
plt.title("Train Accuracy and Loss vs Validation Accuracy and Loss\n Model 2")
plt.xlabel("Epochs")
plt.ylabel("Accuracy (%)")
```

```
Out[12]: Text(0, 0.5, 'Accuracy (%)')
```



```
In [12]: max_val_acc2 = model2_loss.val_accuracy.max()
print("Maximum Validation Accuracy for Model:", round(max_val_acc2 * 100,2), "%")
```

Maximum Validation Accuracy for Model: 92.75 %

```
In [12]: min_val_loss2 = model2_loss.val_loss.min()
print("Minimum Validation Loss for Model:", round(min_val_loss2 * 100,2), "%")
```

Minimum Validation Loss for Model: 18.99 %

**Note:** The Validation Accuracy was better with this model. Let's view the Evaluation Metrics.

Next, we will view the Classification Report and Confusion Matrix to see how this model performed predicting each of the classes, No Tumor and Tumor.

First, we save the model so we can load it later if needed.

**Save the model**

```
In [12]: # Save model
model2.save('cancer_project_model2.keras')
```

## Evaluation Metrics for Model 2

First, we get the predictions on the validation data and see how these scored across classes.

```
In [12]: # Get predictions
yp_2 = model2.predict(test_gen, steps = len(valid_df), verbose = 1)
```

26400/26400 ————— 99s 4ms/step

```
In [13]: # Check predictions
yp_2
```

```
Out[13]: array([[9.9998367e-01, 1.6323285e-05],
 [9.9858022e-01, 1.4198310e-03],
 [9.9999821e-01, 1.7836467e-06],
 ...,
 [3.2442885e-03, 9.9675566e-01],
 [8.4448438e-03, 9.9155515e-01],
 [2.3404529e-05, 9.9997663e-01]], dtype=float32)
```

```
In [13]: # Need to convert predictions to 0 or 1
yp2 = yp_2.argmax(axis = 1)
yp2
```

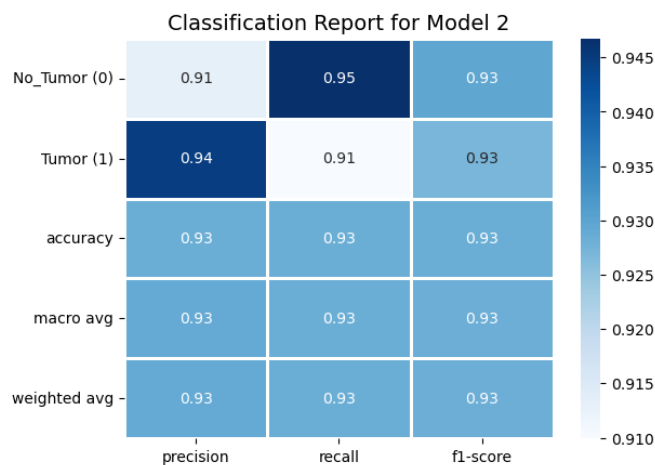
```
Out[13]: array([0, 0, 0, ..., 1, 1, 1])
```

Now that we have the predictions in the necessary format, let's view the Classification Report and Confusion Matrix below.

**Figure 6: Classification Report for Model 2**

```
In [13]: # Classification Report
test_labels = test_gen.classes
names = ['No_Tumor (0)', 'Tumor (1)']
clf_test = classification_report(test_labels, yp2, target_names = names, output_dict = True)

# Heatmap view
sns.heatmap(pd.DataFrame(clf_test).iloc[:-1, :].T, annot = True, cmap = "Blues", linewidth = 1, fmt = '.2f')
plt.title("Classification Report for Model 2", fontsize = 14)
#plt.yticks(rotation = 0)
plt.show()
```



**Summary of Classification Report for Model 2:**

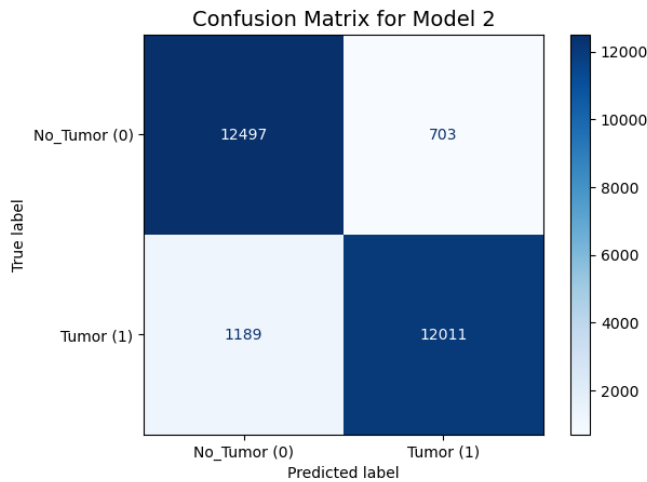
While we see a good overall Accuracy score, we notice differences between the No Tumor (0) and Tumor (1) classes. The scores ranged from 91% to 95%. In particular, there is some difference between the Precision and Recall scores for the No Tumor versus Tumor classes.

Next, let's view the Confusion Matrix.

**Figure 7: Confusion Matrix for Model 2**

```
In [13]: # Get labels and store confusion
test_labels = test_gen.classes
labels = ['No_Tumor (0)', 'Tumor (1)']

# Display confusion matrix
cm = confusion_matrix(test_labels, yp2)
cmd = ConfusionMatrixDisplay(cm, display_labels=labels)
cmd.plot(cmap = plt.cm.Blues)
plt.title('Confusion Matrix for Model 2', fontsize = 14)
plt.grid(False)
plt.show()
```



#### Summary of Confusion Matrix for Model 2:

While 93% Accuracy is a good score, the Confusion Matrix shows more than 1100 images were predicted as No Tumor (0) when, in fact, those images contained a Tumor (1). That error would mean the model missed identifying people whose images actually contained a cancerous tumor. The alarming part of those inaccuracies would be people who may not be treated. Thus, we would hope to continue to improve on our modeling so we are able to better identify those cancerous tumors.

Meanwhile, 700+ images were predicted as a Tumor (1) when, in fact, those images contained No Tumor (0). In this case, we have people who may be alarmed to hear the image contains a cancerous tumor and require further imaging to determine there is no tumor.

All in all, we are happy the model performed well. However, ideally, the goal would be to improve upon this model. Therefore, let's explore the results with a sixth layer of filters and a new optimizer, AdamW, below.

## Model 3: 6 Filters with Batch Normalization and AdamW Optimizer

**Kernel Size:** Here we start with a kernel size of 3x3 which is the size of each filter.

**Number of Filters with each layer:** For images, the recommendation was to increase the number of filters per layer. Therefore, we increase the layers to six with the number of filters proceeding from 32 -> 64 -> 96 -> 128 -> 256 -> 512.

**Conv2D:** Since we have images, we use the Convolution 2D package with the above kernel size and filters. Here we use a different kernel initializer for the convolution kernel, he\_uniform. We also set padding to same, which keeps the same size from input to output.

**Dropout:** For each layer, after the Convolution 2D, we use a dropout of 25%. This Dropout randomly drops a percentage of the training dataset, which can help prevent overfitting.

**Activation:** For each layer, we will use the Rectified Linear Unit or 'relu' activation. Then, for the final activation, we use Softmax with 2 units of output which will provide a probability of two classes -- which are zero and one in this project.

**Optimizer:** When we compile the model, we select 'AdamW' optimizer which is similar to the 'Adam' optimizer but adds a method to decay the weights. We also increase the learning rate to 0.0002.

```
In [94]: # Set initial kernel size, and filters
kernel_sz = (3,3)

# Number of filters with each layer
filters_1 = 32
filters_2 = 64
filters_3 = 96
filters_4 = 128
filters_5 = 256
filters_6 = 512

# Set the learning rate
optimizer = keras.optimizers.AdamW(learning_rate = 0.0002)

In [95]: # Initiate model
model3 = Sequential()

In [96]: # Use strides instead of maxpooling, with Batch Normalization

#Batch 1
model3.add(Conv2D(filters_1, kernel_sz, strides = (1,1), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model3.add(BatchNormalization())

#Batch 2
model3.add(Conv2D(filters_2, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model3.add(BatchNormalization())

#Batch 3
model3.add(Conv2D(filters_3, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model3.add(BatchNormalization())

#Batch 4
model3.add(Conv2D(filters_4, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model3.add(BatchNormalization())
```



```
#Batch 5
model3.add(Conv2D(filters_5, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model3.add(BatchNormalization())

#Batch 6
model3.add(Conv2D(filters_6, kernel_sz, strides = (2,2), activation = 'relu', padding = 'same',
                  kernel_initializer = 'he_uniform'))
model3.add(BatchNormalization())

# Flatten and convert to Dense layer
model3.add(Flatten())
model3.add(Dense(units = 2, activation = 'softmax'))
```

```
In [97_ # Compile model
model3.compile(loss = 'binary_crossentropy', optimizer = optimizer, metrics = ['accuracy'])
```

```
In [98_ # Set a reduced learning rate and early stopping
drop_lr = ReduceLROnPlateau(monitor = 'val_accuracy', factor = 0.5, patience = 2, verbose = 1, mode = 'max',
                             min_lr = 0.00001)

early_stop = EarlyStopping(monitor = 'val_accuracy', mode = 'max', verbose = 1, patience = 2, restore_best_weights = True)

cb_lst = [drop_lr, early_stop]
```

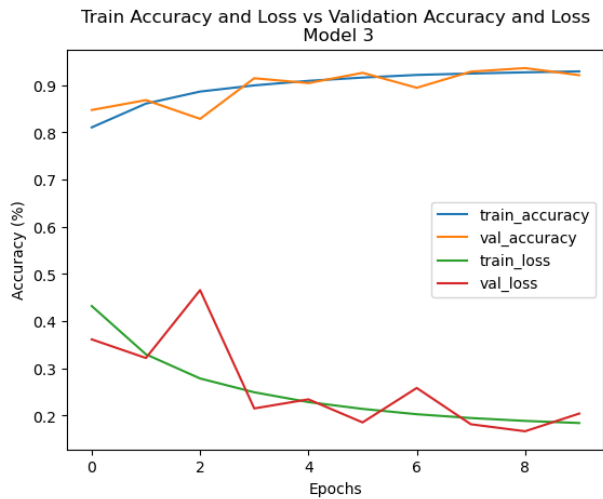
```
In [99_ history3 = model3.fit(train_gen,
                             validation_data = valid_gen,
                             epochs = 10,
                             verbose = 1,
                             callbacks = cb_lst)
```

```
Epoch 1/10
14960/14960 ————— 1168s 78ms/step - accuracy: 0.7826 - loss: 0.4933 - val_accuracy: 0.8475 - val_loss: 0.3616 - learning_rate: 2.0000
e-04
Epoch 2/10
14960/14960 ————— 1076s 72ms/step - accuracy: 0.8530 - loss: 0.3463 - val_accuracy: 0.8683 - val_loss: 0.3221 - learning_rate: 2.0000
e-04
Epoch 3/10
14960/14960 ————— 1270s 85ms/step - accuracy: 0.8819 - loss: 0.2873 - val_accuracy: 0.8286 - val_loss: 0.4659 - learning_rate: 2.0000
e-04
Epoch 4/10
14960/14960 ————— 1309s 88ms/step - accuracy: 0.8963 - loss: 0.2569 - val_accuracy: 0.9147 - val_loss: 0.2153 - learning_rate: 2.0000
e-04
Epoch 5/10
14960/14960 ————— 1265s 85ms/step - accuracy: 0.9097 - loss: 0.2299 - val_accuracy: 0.9044 - val_loss: 0.2345 - learning_rate: 2.0000
e-04
Epoch 6/10
14960/14960 ————— 1309s 87ms/step - accuracy: 0.9133 - loss: 0.2194 - val_accuracy: 0.9262 - val_loss: 0.1857 - learning_rate: 2.0000
e-04
Epoch 7/10
14960/14960 ————— 1253s 84ms/step - accuracy: 0.9209 - loss: 0.2050 - val_accuracy: 0.8945 - val_loss: 0.2587 - learning_rate: 2.0000
e-04
Epoch 8/10
14960/14960 ————— 1334s 89ms/step - accuracy: 0.9229 - loss: 0.1990 - val_accuracy: 0.9287 - val_loss: 0.1818 - learning_rate: 2.0000
e-04
Epoch 9/10
14960/14960 ————— 1400s 94ms/step - accuracy: 0.9274 - loss: 0.1897 - val_accuracy: 0.9362 - val_loss: 0.1670 - learning_rate: 2.0000
e-04
Epoch 10/10
14960/14960 ————— 1413s 94ms/step - accuracy: 0.9285 - loss: 0.1857 - val_accuracy: 0.9212 - val_loss: 0.2044 - learning_rate: 2.0000
e-04
Restoring model weights from the end of the best epoch: 9.
```

**Note:** After multiple prior experiments, this model was our best yet. Let's view the scores below.

```
In [10_ # View plot
model3_loss = pd.DataFrame(model3.history.history)
model3_loss = model3_loss.rename(columns = {'accuracy': 'train_accuracy', 'loss': 'train_loss'})
model3_loss[['train_accuracy', 'val_accuracy', 'train_loss', 'val_loss']].plot()
plt.title("Train Accuracy and Loss vs Validation Accuracy and Loss\nModel 3")
plt.xlabel("Epochs")
plt.ylabel("Accuracy (%)")
```

```
Out[10_ Text(0, 0.5, 'Accuracy (%)')
```



```
In [10]: max_val_acc3 = model3_loss.val_accuracy.max()
print("Maximum Validation Accuracy for Model:", round(max_val_acc3 * 100,2), "%")
```

Maximum Validation Accuracy for Model: 93.62 %

```
In [10]: min_val_loss3 = model3_loss.val_loss.min()
print("Minimum Validation Loss for Model:", round(min_val_loss3 * 100,2), "%")
```

Minimum Validation Loss for Model: 16.7 %

Those are good results. Let's save the model.

#### Save the model

```
In [10]: # Save the model
model3.save('cancer_project_model3.keras')
```

Next, let's view the evaluation metrics for this model.

#### Evaluation Metrics for Model 3

Let's get the predictions so we can view the Classification Report and Confusion Matrix to better evaluate the results.

```
In [10]: # Get predictions
yp_3 = model3.predict(test_gen, steps = len(valid_df), verbose = 1)
```

26400/26400 ————— 74s 3ms/step

```
In [10]: # Check predictions
yp_3
```

```
Out[10]: array([[9.99378085e-01, 6.21891057e-04],
 [9.95705545e-01, 4.29441407e-03],
 [9.99983668e-01, 1.62810611e-05],
 ...,
 [1.38202775e-02, 9.86179650e-01],
 [1.09782917e-02, 9.89021659e-01],
 [1.93003530e-06, 9.9998093e-01]], dtype=float32)
```

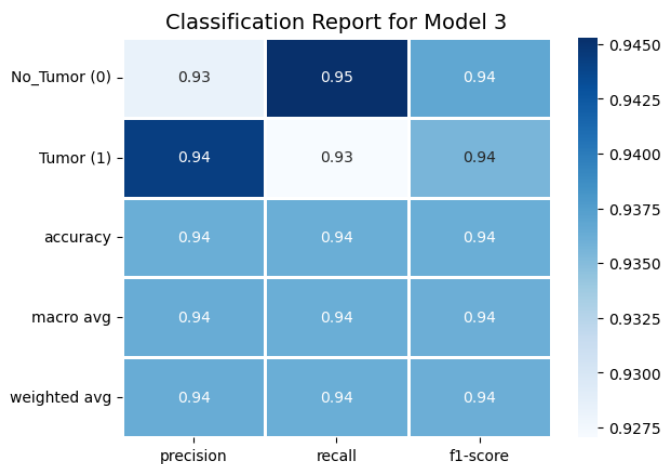
```
In [10]: # Need to convert predictions to 0 or 1
yp3 = yp_3.argmax(axis = 1)
yp3
```

```
Out[10]: array([0, 0, 0, ..., 1, 1, 1])
```

Figure 8: Classification Report for Model 3

```
In [11]: # Classification Report
test_labels = test_gen.classes
names = ['No_Tumor (0)', 'Tumor (1)']
clf_test = classification_report(test_labels, yp3, target_names = names, output_dict = True)

# Heatmap view
sns.heatmap(pd.DataFrame(clf_test).iloc[:-1, :].T, annot = True, cmap = "Blues", linewidth = 1, fmt = '.2f')
plt.title("Classification Report for Model 3", fontsize = 14)
#plt.yticks(rotation = 0)
plt.show()
```



#### Summary of Classification Report for Model 3:

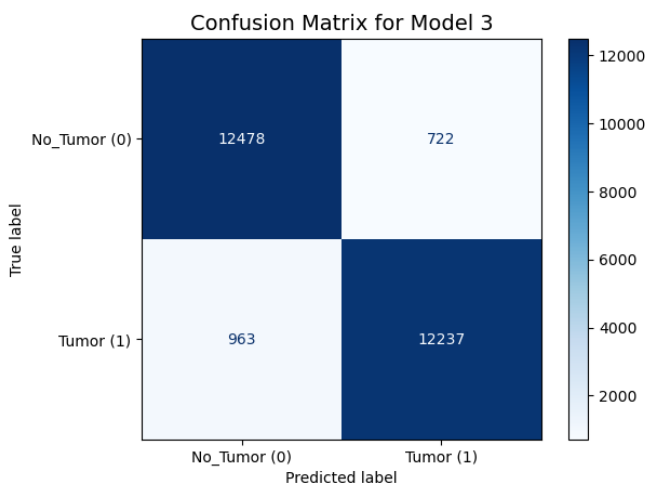
Here we see fairly balanced results for the No Tumor (0) and Tumor (1) classes. The scores ranged from 93% to 95%. Although, we do note some difference between the Precision and Recall scores for the No Tumor versus Tumor classes.

Next, let's view the Confusion Matrix.

**Figure 9: Confusion Matrix for Model 3**

```
In [11]: # Get labels and store confusion
test_labels = test_gen.classes
labels = ['No_Tumor (0)', 'Tumor (1)']

# Display confusion matrix
cm = confusion_matrix(test_labels, yp3)
cmd = ConfusionMatrixDisplay(cm, display_labels=labels)
cmd.plot(cmap = plt.cm.Blues)
plt.title('Confusion Matrix for Model 3', fontsize = 14)
plt.grid(False)
plt.show()
```



#### Summary of Confusion Matrix for Model 3:

While 94% Accuracy is a good score, the Confusion Matrix shows more than 900 images were predicted as No Tumor (0) when, in fact, those images contained a Tumor (1). That error indicates the model missed identifying people whose images actually contained a cancerous tumor. The alarming part of a 6% inaccuracy rate would be those people with a cancerous tumor who may not be treated. Thus, we would aim to improve our modeling so we better identify those cancerous tumors.

Meanwhile, 700+ images were predicted as a Tumor (1) when, in fact, those images contained No Tumor (0). In this case, we have people who may be alarmed to hear the image contains a cancerous tumor and further imaging is necessary to verify there is no tumor.

All in all, we are happy that our best model performed well. However, ideally, the goal would be to improve upon this model.

## Section 6: Test Directory and Images for Kaggle submissions

Now we can switch to our Test Directory, create predictions on the test data and submit to Kaggle to check our models.

```
In [14]: # Create test directory
test_dir = 'test_dir'
os.mkdir(test_dir)

# prepare to put images in 'test_imgs' in test_dir
test_imgs = os.path.join(test_dir, 'test_imgs')
os.mkdir(test_imgs)
```

```
In [14]: os.listdir('test_dir')

Out[14]: ['test_imgs']

In [14]: # Check folders exist
print("Test_dir folders:", os.listdir('test_dir'))

Test_dir folders: ['test_imgs']
```

#### Images were transferred

This code is shown for display purposes but the code is not run, since we do not want to re-transfer the images.

```
In [14]: # Transfer images for test
#test_img_lst = os.listdir('cancer_images/test')

#for image in test_img_lst:
#    # filename is the image
#    fname = image
#    # source path
#    sourcepth_tst = os.path.join('cancer_images/test', fname)
#    # destination path to image
#    destpth_tst = os.path.join(test_imgs, fname)
#    # transfer images
#    shutil.copyfile(sourcepth_tst, destpth_tst)
```

Here we change the path for the test data to generate predictions for Kaggle.

```
In [14]: # Change Directory to Test Data
test_pth = 'test_dir'

# Prepare the Test Data
test_gen = idg.flow_from_directory(test_pth,
                                   target_size=(img_sz, img_sz),
                                   batch_size=1,
                                   class_mode='categorical',
                                   shuffle=False)
```

Found 57458 images belonging to 1 classes.

#### Predictions for Model 1

```
In [15]: # Get predictions on test data for Model 1
test_sz = 57458
yp_test1 = model1.predict(test_gen, steps = test_sz, verbose = 1)

57458/57458 ————— 157s 3ms/step
```

```
In [15]: # Need to convert predictions to 0 or 1
ypred_test1 = yp_test1.argmax(axis = 1)
ypred_test1
```

```
Out[15]: array([1, 1, 1, ..., 0, 1, 0])
```

```
In [15]: # Create dataframe with label column
yp_test_df1 = pd.DataFrame(ypred_test1, columns = ['label'])
yp_test_df1.head()
```

```
Out[15]:
```

	label
0	1
1	1
2	1
3	0
4	0

#### Test image names need to be converted into an 'id' column

For the Kaggle submittal, we need to include the test image names as simply an 'id' column which does not include the .tif extension nor the 'test\_imgs/' at the start of the name.

Therefore, we will get the names then remove the .tif extension and remove the 'test\_imgs/' at the front. Once we have done so, we can use these image names in our future model submittals.

```
In [14]: # get test image names
test_img_names = test_gen.filenames
```

```
In [15]: # Check test image names
test_img_names[0:5]
```

```
Out[15]: ['test_imgs/00006537328c33e284c973d7b39d340809f7271b.tif',
'test_imgs/0000ec92553fda4ce39889f9226ace43cae3364e.tif',
'test_imgs/00024a6dee61f12f7856b0fc6be20bc7a48ba3d2.tif',
'test_imgs/000253dfaa0be9d0d100283b22284ab2f6b643f6.tif',
'test_imgs/000270442cc15af719583a8172c87cd2bd9c7746.tif']
```

```
In [15]: # Add column to dataframe with image names
yp_test_df1['img_names'] = test_img_names
```

```
In [15]: yp_test_df1.head()
```

```
Out[15]:
label
0 1 test_imgs/00006537328c33e284c973d7b39d340809f7...
1 1 test_imgs/0000ec92553fda4ce39889f9226ace43cae3...
2 1 test_imgs/00024a6dee61f12f7856b0fc6be20bc7a48b...
3 0 test_imgs/000253dfaa0be9d0d100283b22284ab2f6b6...
4 0 test_imgs/000270442cc15af719583a8172c87cd2bd9c...

In [16]:
# Change image name column to id
yp_test_df1['id'] = yp_test_df1['img_names']

In [16]:
# Drop the .tif extension
yp_test_df1['id'] = yp_test_df1['id'].str[:-4].replace({'\.tif':''}, regex = True)

In [16]:
# Check df to confirm .tif removed
pd.set_option('display.max_colwidth', 1000)
print(yp_test_df1.id.tail(5))

57453 test_imgs/ffffdd1cbb1ac0800f65309f344dd15e9331e1c53
57454 test_imgs/ffffdf4b82ba01f9cae88b9fa45be103344d9f6e3
57455 test_imgs/fffec7da56b54258038b0d382b3d55010eceb9d7
57456 test_imgs/ffff276d06a9e3fffc456f2a5a7a3fd1a2d322c6
57457 test_imgs/ffffeb4c0756098c7f589b7beec08ef1899093b5
Name: id, dtype: object

In [16]:
# Drop the test_imgs/ at front of id
yp_test_df1.id = yp_test_df1.id.str.slice(10)

In [16]:
# Confirm that it was dropped
pd.set_option('display.max_colwidth', 1000)
print(yp_test_df1.id.tail(5))

57453 fffffdd1cbb1ac0800f65309f344dd15e9331e1c53
57454 fffffdf4b82ba01f9cae88b9fa45be103344d9f6e3
57455 fffec7da56b54258038b0d382b3d55010eceb9d7
57456 ffff276d06a9e3fffc456f2a5a7a3fd1a2d322c6
57457 fffffeb4c0756098c7f589b7beec08ef1899093b5
Name: id, dtype: object

In [17]:
# View current dataframe to check these are correct
pd.set_option('display.max_colwidth', 1200)
yp_test_df1.head()
```

	label	img_names	id
0	1	test_imgs/00006537328c33e284c973d7b39d340809f7271b.tif	00006537328c33e284c973d7b39d340809f7271b
1	1	test_imgs/0000ec92553fda4ce39889f9226ace43cae3364e.tif	0000ec92553fda4ce39889f9226ace43cae3364e
2	1	test_imgs/00024a6dee61f12f7856b0fc6be20bc7a48ba3d2.tif	00024a6dee61f12f7856b0fc6be20bc7a48ba3d2
3	0	test_imgs/000253dfaa0be9d0d100283b22284ab2f6b643f6.tif	000253dfaa0be9d0d100283b22284ab2f6b643f6
4	0	test_imgs/000270442cc15af719583a8172c87cd2bd9c7746.tif	000270442cc15af719583a8172c87cd2bd9c7746

Create dataframe in submittal format with id and label columns

This data will be put into a dataframe to prepare for our Kaggle submittal, as well as for future usage below.

Once we have it put into a dataframe, we can simply put the submittal into .csv format. At which point, we will load the .csv file into the Kaggle submission tab and screenshot the results.

```
In [19]:
# For submittal, keep just the id and label columns
yp_df1 = yp_test_df1[['id', 'label']]
yp_df1.head()
```

	id	label
0	00006537328c33e284c973d7b39d340809f7271b	1
1	0000ec92553fda4ce39889f9226ace43cae3364e	1
2	00024a6dee61f12f7856b0fc6be20bc7a48ba3d2	1
3	000253dfaa0be9d0d100283b22284ab2f6b643f6	0
4	000270442cc15af719583a8172c87cd2bd9c7746	0


```
In [17]:
# Save predictions to pandas .csv
yp_df1.to_csv('model1_submission.csv', index=False, header = True)
```

Kaggle screenshot of model submission

Submission and Description

Private Score ⓘ

Public Score ⓘ



**model1\_submission.csv**  
Complete (after deadline) · now · Model 1 submission

0.8138

0.8563

Summary:.

Here we see a Private Score, as well as Public Score. When one views the information icon, Kaggle explains the Public Score is calculated rapidly on 20% of the data. After which, a Private Score is calculated on 80% of the data. As a result, there is often a different score reported for each.

For Model 1, we had a 90.6% Validation Accuracy score then an 85.6% Public Score and 81.4% Private Score.

Interestingly enough, some models have higher Public Scores than other models -- yet, have a lower Private Score. Thus, we will submit all 3 of these models to see where each model scores.

Next, let's get our predictions on the test data for Model 2.

Predictions for Model 2

```
In [17]: # Get predictions on test data
test_sz = 57458
yp_test2 = model2.predict(test_gen, steps = test_sz, verbose = 1)
57458/57458 ----- 198s 3ms/step

In [17]: # Convert predictions to 0 or 1
ypred_test2 = yp_test2.argmax(axis = 1)
ypred_test2

Out[17]: array([1, 1, 1, ..., 0, 0, 0])

In [17]: # Create dataframe with new label column
yp_test_df2 = pd.DataFrame(ypred_test2, columns = ['new_label'])
yp_test_df2.head()

Out[17]:
```

	new_label
0	1
1	1
2	1
3	0
4	0

Concatenate the test dataframe with model predictions

Since we already put the test image names in a dataframe for Model 1 submittal, we can simply concatenate the appropriate columns for our next submittal.

```
In [17]: # Concatenate the test image names with predictions
new_df2 = pd.concat([yp_df1, yp_test_df2], join = 'outer', axis = 1)
new_df2.head()

Out[17]:
```

	id	label	new_label
0	00006537328c33e284c973d7b39d340809f7271b	1	1
1	0000ec92553fda4ce39889f9226ace43cae3364e	1	1
2	00024a6dee61f12f7856b0fc6be20bc7a48ba3d2	1	1
3	000253dfaa0be9d0d100283b22284ab2f6b643f6	0	0
4	000270442cc15af719583a8172c87cd2bd9c7746	0	0

```
In [18]: # Keep the id and new label
test_df2 = new_df2[['id', 'new_label']].copy()

In [18]: # Rename the new label for submission
test_df2.rename(columns = {'new_label': 'label'}, inplace = True)
test_df2.head()


Out[18]:
```

	id	label
0	00006537328c33e284c973d7b39d340809f7271b	1
1	0000ec92553fda4ce39889f9226ace43cae3364e	1
2	00024a6dee61f12f7856b0fc6be20bc7a48ba3d2	1
3	000253dfaa0be9d0d100283b22284ab2f6b643f6	0
4	000270442cc15af719583a8172c87cd2bd9c7746	0

```
In [18]: # Save predictions to pandas.csv
test_df2.to_csv('model2_submission.csv', index = False, header = True)
```

Next, we will submit to Kaggle and include the screenshot below.

Kaggle Screenshot of model submission

Submission and Description	Private Score ⓘ	Public Score ⓘ
<div> <b>model2_submission.csv</b> Complete (after deadline) · now · Model 2 submission</div>	<b>0.8115</b>	<b>0.8711</b>

Model 2 performed better on the Kaggle Public Score than Model 1, but worse on Private Score

Here we see the Kaggle Public Score for Model 2 was 87.1% which is higher than Model 1's Public Score of 85.6% -- yet, the Private Score for Model 2 of 81.2% is less than the Private Score for Model 1 of 81.4%.

Therefore, while Model 2 had a higher Validation Accuracy and higher Kaggle Public Score, it actually performed slightly worse on the test data.

Next, let's get our predictions for Model 3 and see how this model performs on the Kaggle scoring.

#### Predictions for Model 3

```
In [18]: # Get predictions on test data
test_sz = 57458
yp_test3 = model3.predict(test_gen, steps = test_sz, verbose = 1)
```

57458/57458 ————— 140s 2ms/step

```
In [18]: # Convert predictions to 0 or 1
ypred_test3 = yp_test3.argmax(axis = 1)
ypred_test3
```

```
Out[18]: array([1, 1, 1, ..., 0, 1, 0])
```

```
In [18]: # Create new label column
yp_test_df3 = pd.DataFrame(ypred_test3, columns = ['new_label'])
yp_test_df3.head()
```

```
Out[18]:
```

	new_label
0	1
1	1
2	1
3	0
4	0

#### Concatenate the test dataframe with model predictions

Since we already put the test image names in a dataframe for Model 1 submittal, we can simply concatenate the appropriate columns for our next submittal.

```
In [18]: # Concatenate the test image names with predictions
new_df3 = pd.concat([yp_df1, yp_test_df3], join = 'outer', axis = 1)
new_df3.head()
```

```
Out[18]:
```

	id	label	new_label
0	00006537328c33e284c973d7b39d340809f7271b	1	1
1	0000ec92553fda4ce39889f9226ace43cae3364e	1	1
2	00024a6dee61f12f7856b0fc6be20bc7a48ba3d2	1	1
3	000253dfaa0be9d0d100283b22284ab2f6b643f6	0	0
4	000270442cc15af719583a8172c87cd2bd9c7746	0	0

```
In [18]: # Keep the id and new label
test_df3 = new_df3[['id', 'new_label']].copy()
```

```
In [18]: # Rename the new label for submission
test_df3.rename(columns = {'new_label': 'label'}, inplace = True)
test_df3.head()
```

```
Out[18]:
```

	id	label
0	00006537328c33e284c973d7b39d340809f7271b	1
1	0000ec92553fda4ce39889f9226ace43cae3364e	1
2	00024a6dee61f12f7856b0fc6be20bc7a48ba3d2	1
3	000253dfaa0be9d0d100283b22284ab2f6b643f6	0
4	000270442cc15af719583a8172c87cd2bd9c7746	0

```
In [19]: # Save predictions to pandas.csv
test_df3.to_csv('model3_submission.csv', index = False, header = True)
```

#### Screenshot of Kaggle submission

### Submission and Description

Private Score ⓘ

Public Score ⓘ



**model3\_submission.csv**

Complete (after deadline) · now · Model 3 submission

**0.8263**

**0.8792**

#### Model 3 delivers the best Kaggle Private and Public Scores of these models

Here we see the Kaggle submission for Model 3 yielded higher Public and Private Scores than either Model 1 or Model 2.

Therefore, of these models, Model 3 delivered the best results on the test data.

However, if we had more time and additional GPUs, we would keep trying to improve our results.

### Summary of Model Layers and Parameters

Below is a summary of the layers, shape and parameters for Model 3.

```
In [31]: model3.summary()
```

Model: "sequential\_2"

Layer (type)	Output Shape	Param #
conv2d_10 (Conv2D)	(None, 96, 96, 32)	896
batch_normalization (BatchNormalization)	(None, 96, 96, 32)	128
conv2d_11 (Conv2D)	(None, 48, 48, 64)	18,496
batch_normalization_1 (BatchNormalization)	(None, 48, 48, 64)	256
conv2d_12 (Conv2D)	(None, 24, 24, 96)	55,392
batch_normalization_2 (BatchNormalization)	(None, 24, 24, 96)	384
conv2d_13 (Conv2D)	(None, 12, 12, 128)	110,720
batch_normalization_3 (BatchNormalization)	(None, 12, 12, 128)	512
conv2d_14 (Conv2D)	(None, 6, 6, 256)	295,168
batch_normalization_4 (BatchNormalization)	(None, 6, 6, 256)	1,024
conv2d_15 (Conv2D)	(None, 3, 3, 512)	1,180,160
batch_normalization_5 (BatchNormalization)	(None, 3, 3, 512)	2,048
flatten_2 (Flatten)	(None, 4608)	0
dense_2 (Dense)	(None, 2)	9,218

Total params: 5,018,856 (19.15 MB)

Trainable params: 1,672,226 (6.38 MB)

Non-trainable params: 2,176 (8.50 KB)

Optimizer params: 3,344,454 (12.76 MB)

### Receiver Operating Characteristic (ROC) Curve

One ask of the Kaggle competition was to evaluate submissions on the Area Under the ROC Curve (AUC). Please see that plot below.

```
In [31]: # Get probabilities of model 3
ypp = yp_3[:,1]
```

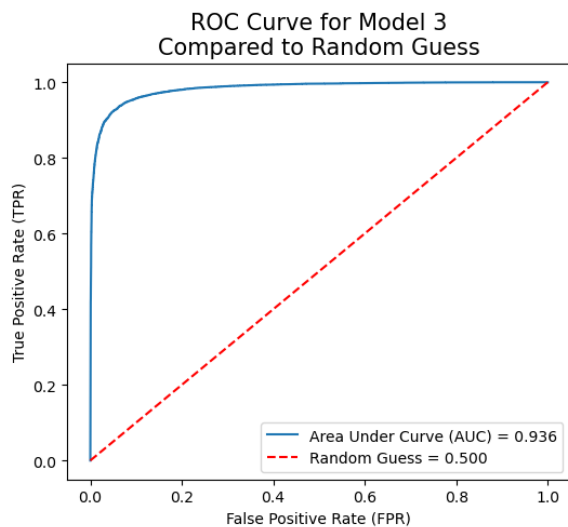
```
# Get FPR and TPR values for ROC Curve plot
valid_class = valid_gen.classes
fpr_3, tpr_3, thresholds_3 = roc_curve(valid_class, ypp)
```

```
# get auc score for plot
auc_3 = roc_auc_score(valid_class, yp3)
print("AUC for Model:", round(auc_3, 5))
```

AUC for Model: 0.93617

```
In [36]: # Create plot of ROC curve and random guess
plt.figure(figsize = (6,5))
plt.plot(fpr_3, tpr_3, label = 'Area Under Curve (AUC) = {:.3f}'.format(auc_3))
plt.plot([0, 1], [0, 1], 'r--', label = 'Random Guess = {:.3f}'.format(0.5))
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve for Model 3\nCompared to Random Guess', fontsize = 15)
plt.legend(loc = 'best')
plt.show()
```





#### Summary:

Here we see the Area Under the Curve (AUC) is 93.6% which is significantly better than the 50% chance of a Random Guess in predicting between the two classes.

Next, let's view a comparison of the results across models.

## Section 7: Compare Results Across Models

Below we have visualizations of the Validation Accuracy and Validation Loss scores across models.

There is also a table comparing the hyperparameters for the models, along with their score.

**Figure 10: Validation Accuracy Scores across Models**

```
In [31]: # Extract the accuracy scores for each model and create list of scores and models
val_acc6, val_acc7, val_acc8, val_acc9, val_acc11 = 0.8879, 0.8897, 0.8956, 0.8933, 0.9369

acc_scores = [max_val_acc1, max_val_acc2, max_val_acc3, val_acc6, val_acc7, val_acc8, val_acc9, val_acc11]
model_names = ["Model 1", "Model 2", "Model 3", "Model 6", "Model 7", "Model 8", "Model 9", "Model 11"]

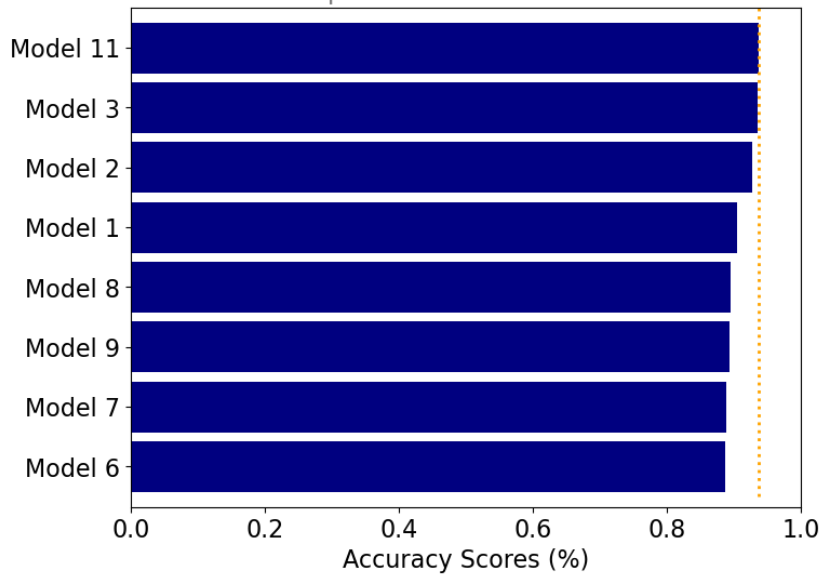
In [31]: # Create plot of Accuracy and model names
plt.figure()
plt.figure(figsize = (8,6))
plt.suptitle("Accuracy Scores across all Models", fontsize = 20, x = 0.51, y = 0.98)
plt.title("Scores reported for the Validation Data", fontsize = 18, color = "grey")

acc_plot = pd.Series(acc_scores,
                    index = model_names).sort_values(ascending = True)
ax = acc_plot.plot.barh(width = 0.85, color = 'navy')
plt.vlines(x = 0.937, ymin = -0.5, ymax = 100, color = "orange", ls = 'dotted', lw = 2)
plt.xticks(np.arange(0, 1.20, step= 0.2), fontsize = 16)
plt.yticks(fontsize = 16)
plt.xlabel("Accuracy Scores (%)", fontsize = 16)
plt.show()
```

<Figure size 640x480 with 0 Axes>

## Accuracy Scores across all Models

Scores reported for the Validation Data



### Summary of Accuracy Scores across Models:

Some items noted above:

- The best results were from Model 3 with the sixth layer of filters.
- Models 1, 2 and 3 generated similar Accuracy scores on the Validation Data with different hyperparameters.
- The first model, Model 1, generated decent results and better than a number of ensuing models. In particular, this model had a higher Learning Rate but did not have Early Stopping. Therefore, it ran for 10 epochs and that plot showed the variability as the Validation Accuracy fluctuated above and below the Training Accuracy.
- When Early Stopping was introduced, the models stopped too early. Once a value for Patience was added to ensure the epochs continued running, as well as reducing the Learning Rate, the models' performance improved.

### Figure 11: Validation Loss across Models

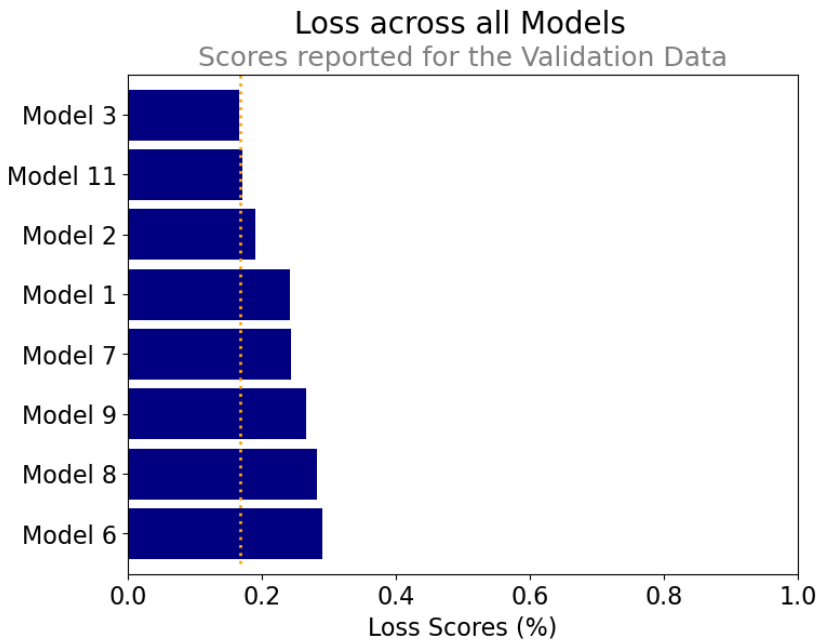
Next, let's view the loss for each model on the Validation Data.

```
In [31]: # Extract the validation loss for each model and create list of scores and models
val_loss6, val_loss7, val_loss8, val_loss9, val_loss11 = 0.2909, 0.2444, 0.2818, 0.2656, 0.1710
loss_scores = [min_val_loss1, min_val_loss2, min_val_loss3, val_loss6, val_loss7, val_loss8, val_loss9, val_loss11]
model_names = ["Model 1", "Model 2", "Model 3", "Model 6", "Model 7", "Model 8", "Model 9", "Model 11"]

In [31]: # Create plot of Loss and model names
plt.figure()
plt.figure(figsize = (8,6))
plt.suptitle("Loss across all Models", fontsize = 20, x = 0.51, y = 0.98)
plt.title("Scores reported for the Validation Data", fontsize = 18, color = "grey")

acc_plot = pd.Series(loss_scores,
                     index = model_names).sort_values(ascending = False)
ax = acc_plot.plot.barh(width = 0.85, color = 'navy')
plt.vlines(x = 0.168, ymin = -0.5, ymax = 100, color = "orange", ls = 'dotted', lw = 2)
plt.xticks(np.arange(0, 1.20, step= 0.2), fontsize = 16)
plt.yticks(fontsize = 16)
plt.xlabel("Loss Scores (%)", fontsize = 16)
plt.show()
```

<Figure size 640x480 with 0 Axes>



#### Summary:

- The top model, Model 3, had the lowest loss on the Validation Data.
- Model 11 had a higher Validation Loss than Model 3.
- Meanwhile, Model 8 which was in the middle of the rankings for Validation Accuracy was next to the last for Validation Loss.
- Therefore, the models with lowest Validation Loss were not exactly the same as the models with the highest Validation Accuracy.
- In summary, the selection of metrics can impact the model identified as top performer.

## Summary of Models and Hyperparameters

In conclusion, here is the summary of the top Accuracy scores from models on the Validation Data. We reviewed a number of these models above. The results from the top model is shown in blue, whereas, the score in red reflects a model tested on a smaller size of validation data. Hence, we re-ran a number of these models on a larger Validation Data set which yielded improved scores on the Kaggle test data.

We will walk through some of the Key Learnings and Takeaways for these iterations below, including certain functions and hyperparameters.

**Table 3**

Summary of Results for top Models on Validation Data

Model	Accuracy %	Loss %	Filter Layers	Kernel Initializer	Normalization	Dropout	Final Activation	Optimizer	LR
Model 1	90.56	24.2	5	Glorot Uniform	None	Multiple	Softmax	Adam	1e-4
Model 2	92.75	19.0	5	He Normal	Batch	Single	Softmax	Adam	1e-4
Model 3	93.62	16.7	6	He Normal	Batch	None	Softmax	AdamW	2e-4
Model 6	88.79	29.1	4	He Normal	Batch	Multiple	Sigmoid	Adam	1e-4
Model 7^	88.97	24.4	4	He Normal	Batch	Multiple	Softmax	Adam	1e-4
Model 8	89.56	28.2	4	He Normal	Batch	Multiple	Sigmoid	RMS Prop	1e-4
Model 9^	89.33	26.6	4	He Normal	Batch	Multiple	Softmax	Adam	1e-4
Model 11*	93.69	17.1	5	Glorot Uniform	None	Multiple	Softmax	Adam	1e-4

Note: Model 11\* contained the same parameters as Model 1, but was analyzed with a smaller set of Validation Data

#### Summary of Results:

- As noted previously, Models 1 and 11 contained the same hyperparameters but Model 11 was analyzed with a smaller set of Validation Data. The Kaggle score for Model 11 on the test data was lower than the Kaggle score for Model 1.
- For the original models, the default Learning Rate (LR) of 0.001 was used. However, the models were ending before attaining as high of Validation Accuracy scores as the other models. Therefore, a Learning Rate of 1e-4 or 0.0001 was used for many of these models and yielded better results. Model 3 actually used a Learning Rate of 2e-4. More experimentation with the Learning Rates could lead to a more optimal result.
- Models 6 through 9 were also run on a smaller set of Validation Data. Therefore, the performance of these models may change running them again on a larger set of Validation Data -- as evident with the differences in results attained by Model 1 versus Model 11.
- Models 7 and 9 were run with the same hyperparameters; however, the models ran for a different number of epochs and attained slightly different scores -- despite having a setting to restore the best weights.

Next, let's wrap up with the Conclusion and highlight the key learnings.

## Section 8: Conclusion

### Based on our metrics and results, which model would we select?

In summary, the top model we would select for analyzing and predicting with a high degree of accuracy on this data is Model 3 which used the six layers of filters, along with Batch Normalization and the AdamW optimizer.

However, a different specific metric could alter preferred modeling options. For example, in real world scenarios, a preference to detect cancer - at risk of misidentifying more non-cancerous cases as cancerous -- may lead one to choose a model with less balanced results across the two categories. In that case, models which are able to identify closer to 100% of cancerous tumors while possibly slipping to 90% of non-cancerous may be preferred.

### Key Learnings and Takeaways

- **Validation Data Size:** The percentage of data allocated to our Validation set impacted our results. The benefit was the ability to execute models more quickly to explore what hyperparameters were performing well or poorly in the various models. However, the downside was these models had high Validation Accuracy scores -- yet lower Kaggle Test Accuracy scores than other models.
- **Early Stopping:** Usage of Early Stopping helped save time when the model was not improving, while also allowing usage of the Patience hyperparameter to continue running the model if it simply had one epoch where the performance declined. This function was also utilized to restore the best weights for the model, if performance declined in subsequent epochs.
- **Learning Rates:** Modifying the default Learning Rate for the optimizer while setting the Patience equal to 2 in the Early Stopping helped the models persist through local minima and continue to improve across epochs. The Learning Rate was doubled for Model 3 and yielded our best results; therefore, further experimentation is possible.
- **Number of Layers:** Additional layers of filters tended to yield better results whether using Glorot Uniform or He Normal kernel initializers, Sigmoid or Softmax Activation, Batch Normalization, Adam, AdamW or RMS Prop optimizers.
- **Optimizers:** In the models above, Model 3 had the best results with AdamW optimizer. Although, in the plots of Validation Accuracy and Training Accuracy, we can see fluctuations across the epochs. Thus, the optimizer selection clearly impacts results as the model is trained. Furthermore, the Patience setting in our Early Stopping function helped ensure the model iterated through some of those fluctuations -- while stopping early when the results had not improved for more than 2 epochs.

## Other factors to consider

### Possible bias or factors that might impact results

Differences in the data cleaning, pre-processing and preparation could impact the results. Since this data contained more non-cancerous data than cancerous, the dataset was skewed to non-cancerous. However, we needed to balance the data to gain insight across both classes.

Another factor that could impact results is the system used for building these models. The iterations can be fairly time consuming and better hardware with more GPUs could process the same models in reduced time -- or, allow the execution of more complex models.

### Areas for consideration

Some areas that would be interesting to explore further might include:

- Modifying the Image Data Generator configuration to check results
- Experimentation with different Learning Rates and Learning Rate Schedules
- Using different customized Keras methods
- Evaluating how Pretrained Models from KerasHub performed on this data
- Employing other machine learning models

Overall, it was interesting to see how neural networks performed on this data.

**Thank you for reading!**

## Section 9: Resources

### Websites:

- [https://keras.io/api/layers/convolution\\_layers/convolution2d/](https://keras.io/api/layers/convolution_layers/convolution2d/)
- [https://keras.io/api/layers/normalization\\_layers/batch\\_normalization/](https://keras.io/api/layers/normalization_layers/batch_normalization/)
- [https://keras.io/api/layers/normalization\\_layers/layer\\_normalization/](https://keras.io/api/layers/normalization_layers/layer_normalization/)
- [https://keras.io/api/layers/pooling\\_layers/max\\_pooling2d/](https://keras.io/api/layers/pooling_layers/max_pooling2d/)
- [https://keras.io/api/layers/regularization\\_layers/dropout/](https://keras.io/api/layers/regularization_layers/dropout/)
- <https://keras.io/api/callbacks/>
- <https://keras.io/api/optimizers/>
- <https://arxiv.org/pdf/1409.1556>
- <https://www.kaggle.com/code/vbookshelf/cnn-how-to-use-160-000-images-without-crashing/notebook>
- <https://www.kaggle.com/code/blurredmachine/vggnet-16-architecture-a-complete-guide>
- <https://www.robots.ox.ac.uk/~vgg/>

**GitHub Link:** [https://github.com/chiffr3/CNN\\_project](https://github.com/chiffr3/CNN_project)