CNN Cancer Detection Kaggle Mini-Project

Brief description of the problem and data

In this mini-project, a CNN model is used to identify metastatic cancer in small image patches taken from larger digital pathology scans (this problem is taken from a past Kaggle completition). PCam packs the clinically-relevant task of metastasis detection into a straight-forward **binary image classification** task, which the CNN model performs. The CNN Model is trained on Colab GPU on a given set of images and then it predicts / detects tumors in a set of unseen test images.

The dataset provides a large number of small pathology images to classify. Files are named with an image `id`. The `train_labels.csv` file provides the ground truth for the images in the train folder. The goal is to predict the labels for the images in the test folder. A positive label indicates that the center 32×32px region of a patch contains at least one pixel of tumor tissue. Tumor tissue in the outer region of the patch does not influence the label.

Exploratory Data Analysis (EDA)

First we need to import all `python` packages / functions that are required for building the CNN model. We shall use `tensorflow / keras` to train the deep learning model.

In [4]:

```
import numpy as np
import pandas as pd
import matplotlib.pylab as plt
import seaborn as sns
import os

import tensorflow as tf
import tensorflow.keras as keras
from keras.layers import Conv2D, MaxPool2D, BatchNormalization, Flatten, Dropout, Dens
e, Activation
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from tensorflow.keras.optimizers import Adam

import visualkeras
from tifffile import imread
```

As can be seen, the number of images in training folder (ones with ground-truth labels) and test folders (without ground-truth labels) are around 220k and 57k, respectively.

In [2]:

```
base_dir = 'histopathologic-cancer-detection'
train_dir, test_dir = f'{base_dir}/train/', f'{base_dir}/test/'
ntrain, ntest = len(os.listdir(train_dir)), len(os.listdir(test_dir))
print(f'#training images = {ntrain}, #test inages={ntest}')
```

```
#training images = 220025, #test inages=57458
```

The `train_labels.csv` file is loaded as a `pandas` `DataFrame` (first few rows of the dataframe are shown below). It contains the `id` of the `tif` image files from the training folder, along with their ground-truth label. Let's have the file names in the `id` column instead, by concatenating the `.tif` extension to the `id` values. It will turn out to be useful later while reading the files from the folder automatically.

In [3]:

```
train_df = pd.read_csv(f'{base_dir}/train_labels.csv')
train_df['label'] = train_df['label'].astype(str)
train_df['id'] = train_df['id'] + '.tif'
train_df.head()
```

Out[3]:

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

The images are RGB color images of shape 96×96, as seen from the next code snippet, the images are loaded using `tif` `imread()` function.

In [12]:

```
imread(f'{train_dir}/{train_df.id[0]}').shape
```

Out[12]:

```
(96, 96, 3)
```

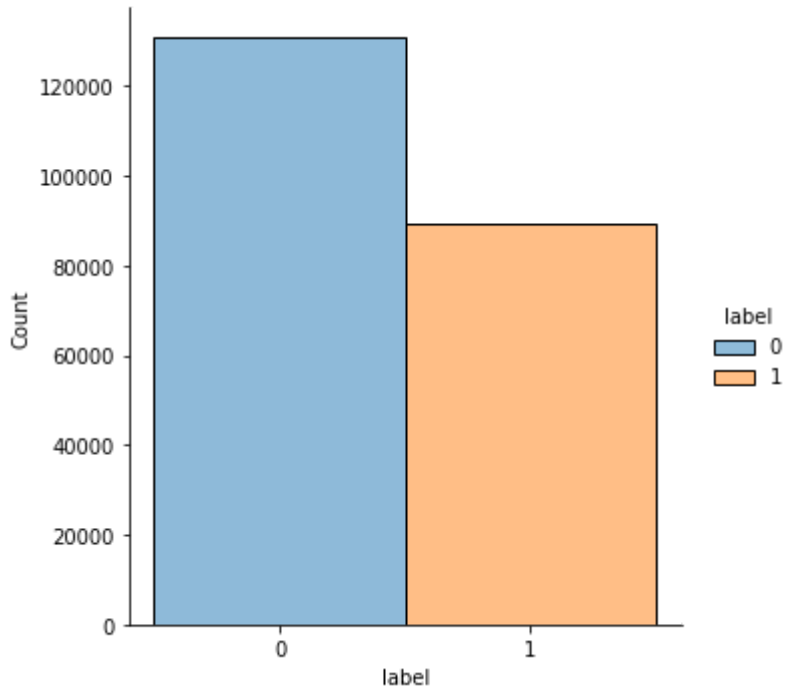
Also, as can be seen from below plot, there are around 130k benign and 89k cancerous images in the trainign dataset, hence the dataset is not very imbalanced, also the data size is large. Hence we are not using augmentation like preprocessing steps.

In [ ]:

```
sns.displot(data=train_df, x='label', hue='label')
train_df['label'] = train_df['label'].astype(int)
train_df['label'].value_counts()
```

Out[ ]:

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

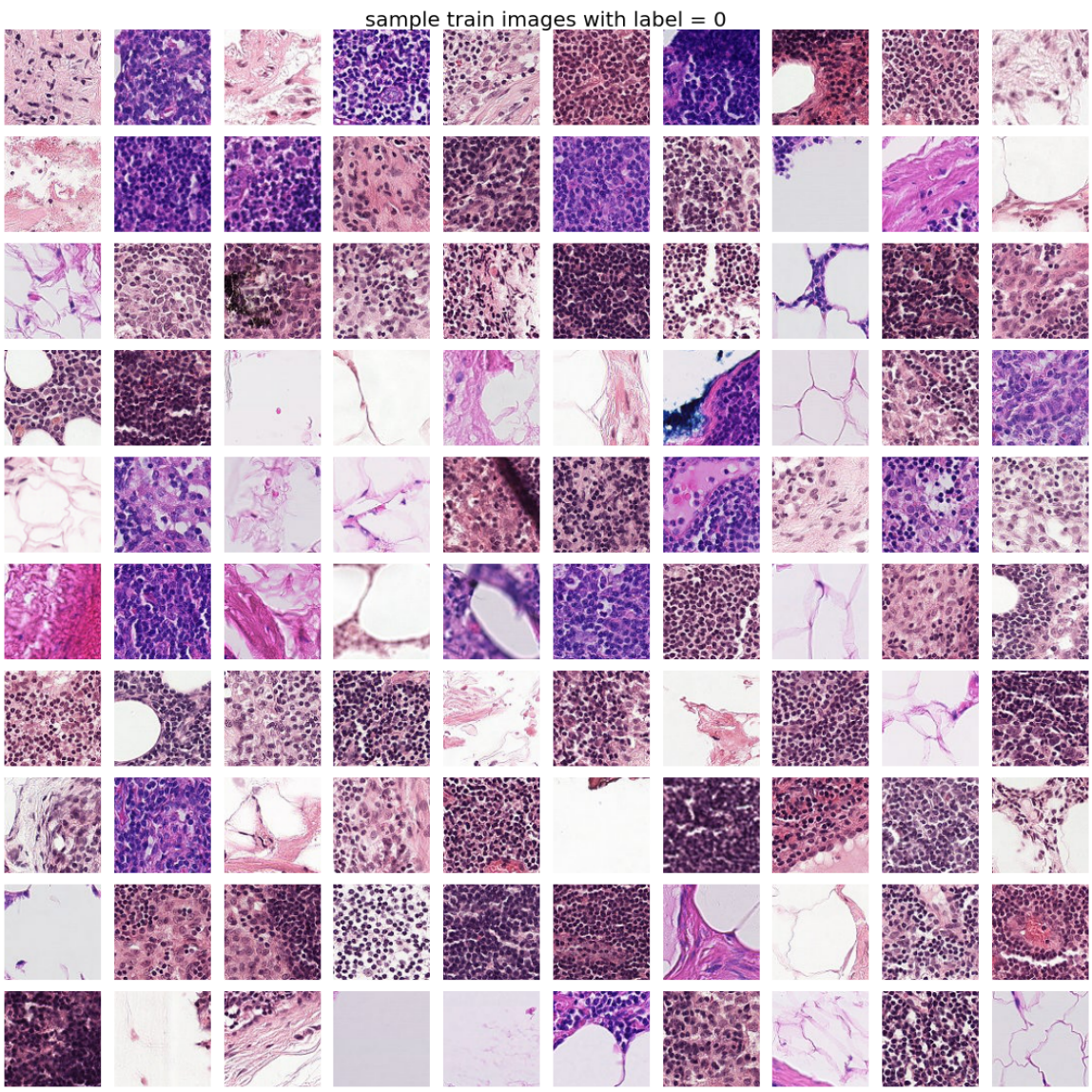


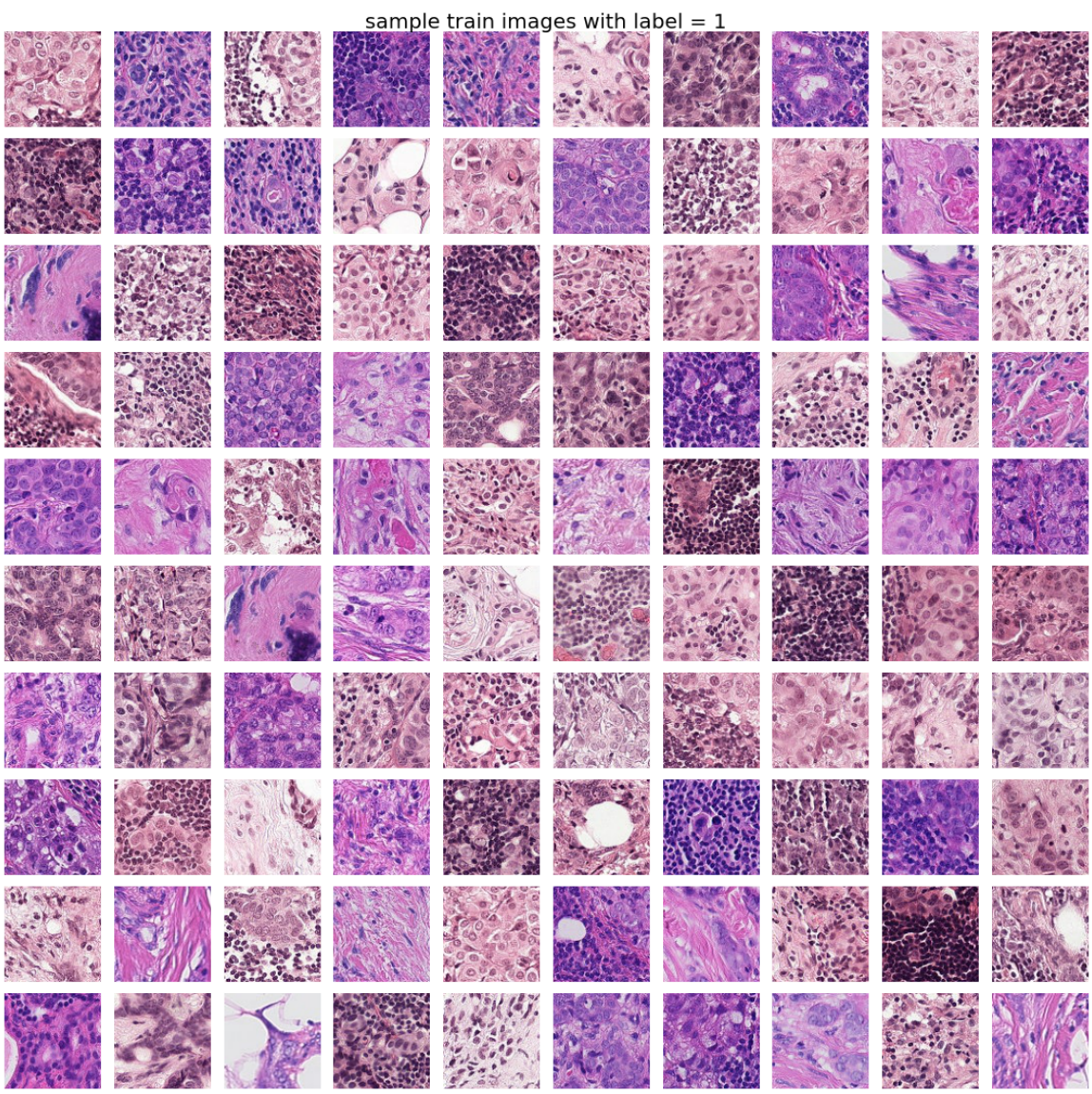
Next let's plot 100 randomly selected bening and cancerous images (with label 0 and 1, respectively) from training dataset to visuallhy inspect the difference, if any.

In [ ]:

```
def plot_images(df, label, n=100):
  df_sub = df.loc[df.label == label].sample(n)
  imfiles = df_sub['id'].values
  plt.figure(figsize=(15,15))
  for i in range(n):
    im = imread(f'{train_dir}/{imfiles[i]}')
    plt.subplot(10,10,i+1)
    plt.imshow(im)
    plt.axis('off')
  plt.suptitle(f'sample train images with label = {label}', size=20)
  plt.tight_layout()
  plt.show()

for label in train_df.label.unique():
  plot_images(train_df, label)
```


sample train images with label = 0

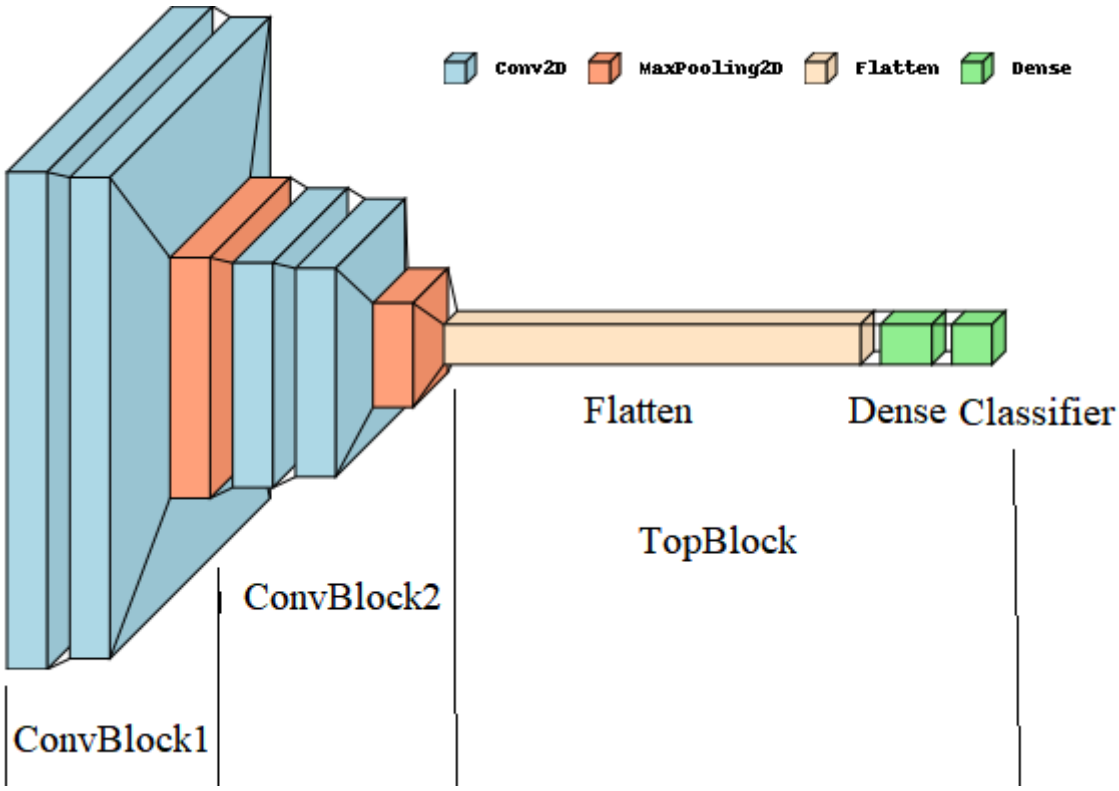

sample train images with label = 1

Model Architecture

We shall implement the model using `keras` *model subclassing* with *reusable blocks* and *functional APIs*.

- There will be a Convolution Block implemented by the class `ConvBlock` which will contain 2 `Conv2D` layers of same size followed by a `MaxPool2D` layer. There can be an optional `BatchNormalization` layer immediately following each fo the the `Conv2D` layers. The filter size, kernel size, activation, pooling size and whether batch norm will be present or not will be configurable (with default values) and can be specified during instantiation of the block.
- The next reusable block will be the `TopBlock` that will contain a Flatten layer followed by two dense layers, the last of which will be classification layer, which will have a single output neuron. The size of the dense layer will be configurable.
- By default `ReLU` activation will be used in the layers, except in the last classifier layer, which will use `sigmoid` activation.

The model is shown below:



The `batch_size` and `im_size` will also be hyperparameters that can be tuned / changed. We shall use batch size of 256 and all the images will be resized to 64×64, in order to save memory.

In [5]:

```
batch_size, im_size = 256, 64
```

In [6]:

```
class ConvBlock(tf.keras.layers.Layer):
    def __init__(self, n_filter, kernel_sz=(3,3), activation='relu', pool_sz=(2,2), bat
ch_norm=False):
        super(ConvBlock, self).__init__()
        self.batch_norm = batch_norm
        self.conv_1 = Conv2D(n_filter, kernel_sz, activation=activation)
        self.bn_1 = BatchNormalization()
        self.conv_2 = Conv2D(n_filter, kernel_sz, activation=activation)
        self.bn_2 = BatchNormalization()
        self.pool = MaxPool2D(pool_size=pool_sz)

    def call(self, x):
        x = self.conv_1(x)
        if self.batch_norm:
          x = self.bn_1(x)
          x = tf.keras.layers.ReLU()(x)
        x = self.conv_2(x)
        if self.batch_norm:
          x = self.bn_2(x)
          x = tf.keras.layers.ReLU()(x)
        return self.pool(x)

class TopBlock(tf.keras.layers.Layer):
    def __init__(self, n_units=256, n_class=1, activation='relu', drop_out=False, drop_
rate=0.5):
        super(TopBlock, self).__init__()
        self.drop_out = drop_out
        self.flat  = tf.keras.layers.Flatten()
        self.dropout = Dropout(drop_rate)
        self.dense = tf.keras.layers.Dense(n_units, activation=activation)
        self.classifier = tf.keras.layers.Dense(1, activation='sigmoid')

    def call(self, x, training=False):
        x = self.flat(x)
        #if training:
        #  x = self.dropout(x)
        x = self.dense(x)
        return self.classifier(x)
```

We shall create couple of models by tuning the hyperparameters `Conv2D` filtersize and `Dense` layersize, with / without normalization:

- `CNNModel1` : the first one without `BatchNormalization` and ConvBlock sizes 16, 32, respectively and dense layer size 256, as shown in the next code snippet.

In [7]:

```
class CNNModel1(tf.keras.Model):
    def __init__(self, input_shape=(im_size,im_size,3), n_class=1):
        super(CNNModel1, self).__init__()
        # the first conv module
        self.conv_block_1 = ConvBlock(16)
        # the second conv module
        self.conv_block_2 = ConvBlock(32, batch_norm=True)
        # model top
        self.top_block = TopBlock(n_units=256)

    def call(self, inputs, training=False, **kwargs):
        # forward pass
        x = self.conv_block_1(inputs)
        x = self.conv_block_2(x)
        return self.top_block(x)

model1 = CNNModel1()
model1.build(input_shape=(batch_size,im_size,im_size,3))
model1.summary()
```

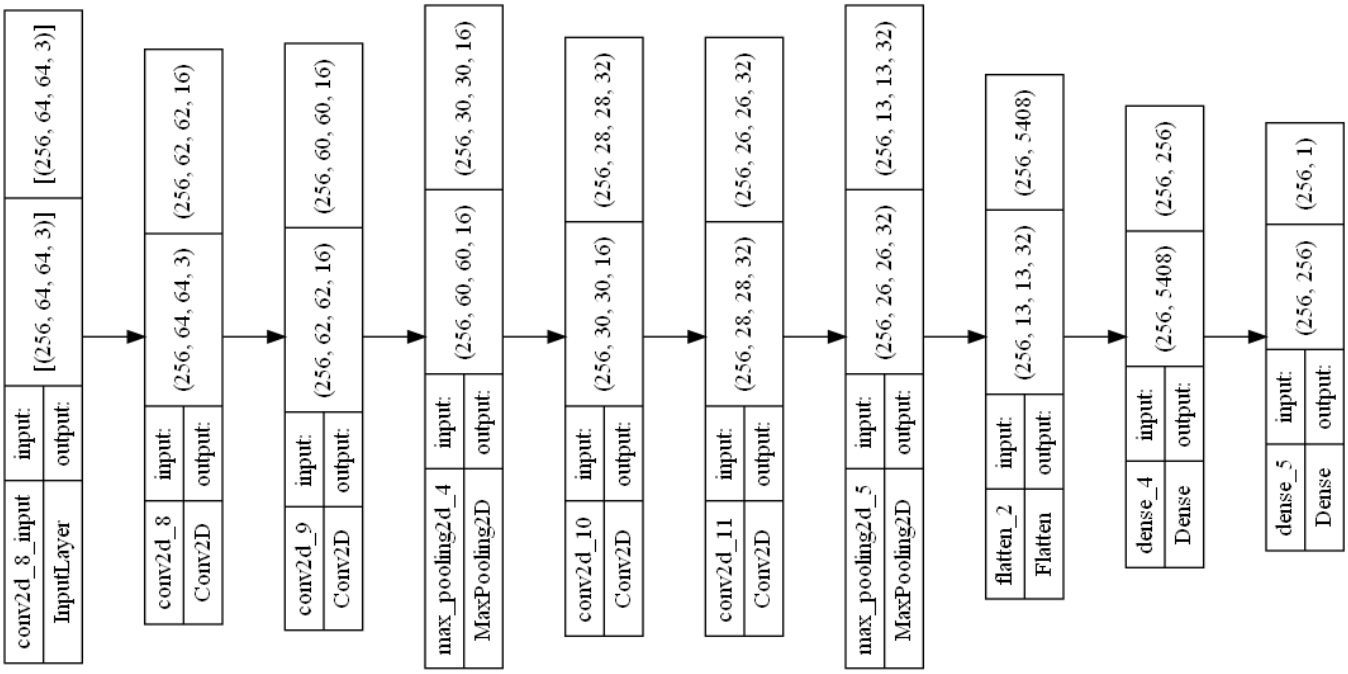
```
Model: "base_model1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_block (ConvBlock)      multiple                  2768

 conv_block_1 (ConvBlock)    multiple                  14144

 top_block (TopBlock)        multiple                  1384961

=================================================================
Total params: 1,401,873
Trainable params: 1,401,745
Non-trainable params: 128
_________________________________________________________________
```

The model architecutre looks like the following (the model can be defined using `keras Sequential` too):



- `CNNModel2` : the second one with `BatchNormalization` and ConvBlock sizes 32, 32, respectively and dense layer size 512, defined / instantiated using the next code snippet.

In [9]:

```
class CNNModel2(tf.keras.Model):
    def __init__(self, input_shape=(im_size,im_size,3), n_class=1):
        super(CNNModel2, self).__init__()
        # the first conv module
        self.conv_block_1 = ConvBlock(32, batch_norm=True)
        # the second conv module
        self.conv_block_2 = ConvBlock(32, batch_norm=True)
        # model top
        self.top_block = TopBlock(n_units=512)

    def call(self, inputs, training=False, **kwargs):
        # forward pass
        x = self.conv_block_1(inputs)
        x = self.conv_block_2(x)
        return self.top_block(x)

model2 = CNNModel2()
model2.build(input_shape=(batch_size,im_size,im_size,3))
model2.summary()
```

```
Model: "base_model2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_block_2 (ConvBlock)    multiple                  10400

 conv_block_3 (ConvBlock)    multiple                  18752

 top_block_1 (TopBlock)      multiple                  2769921

=================================================================
Total params: 2,799,073
Trainable params: 2,798,817
Non-trainable params: 256
_________________________________________________________________
```

Now, we need to read the images, to do this automatically during the training phase, we shall use `ImageDataGenerator` with the `flow_from_dataframe()` method and hold-out 25% of the training images for validation performance evaluation.

In [7]:

```
generator = ImageDataGenerator(rescale=1./255, validation_split=0.25)

train_data = generator.flow_from_dataframe(
    dataframe = train_df,
    x_col='id', # filenames
    y_col='label', # labels
    directory=train_dir,
    subset='training',
    class_mode='binary',
    batch_size=batch_size,
    target_size=im_size)

val_data = generator.flow_from_dataframe(
    dataframe=train_df,
    x_col='id', # filenames
    y_col='label', # labels
    directory=train_dir,
    subset="validation",
    class_mode='binary',
    batch_size=batch_size,
    target_size=im_size)
```

```
Found 165019 validated image filenames belonging to 2 classes.
Found 55006 validated image filenames belonging to 2 classes.
```

Results and Analysis

The model without the `BatchNormalization` layers is first trained. `Adam` optimizer is used with `learning_rate=0.001` (higher learning rates seems to diverge), with loss function as *BCE* (`binary_crossentropy`) and trained for 10 epochs.

Results with Model1

In [ ]:

```
opt = Adam(learning_rate=0.0001)
model1.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
hist = model.fit(train_data, validation_data=val_data, epochs=10)
```

```
Epoch 1/10
645/645 [==============================] - 465s 702ms/step - loss: 0.4813
- accuracy: 0.7729 - val_loss: 0.4514 - val_accuracy: 0.7940
Epoch 2/10
645/645 [==============================] - 313s 486ms/step - loss: 0.4457
- accuracy: 0.7974 - val_loss: 0.4334 - val_accuracy: 0.8033
Epoch 3/10
645/645 [==============================] - 295s 458ms/step - loss: 0.4254
- accuracy: 0.8090 - val_loss: 0.4130 - val_accuracy: 0.8156
Epoch 4/10
645/645 [==============================] - 304s 472ms/step - loss: 0.4085
- accuracy: 0.8181 - val_loss: 0.4145 - val_accuracy: 0.8127
Epoch 5/10
645/645 [==============================] - 293s 454ms/step - loss: 0.3970
- accuracy: 0.8232 - val_loss: 0.4027 - val_accuracy: 0.8197
Epoch 6/10
645/645 [==============================] - 307s 476ms/step - loss: 0.3859
- accuracy: 0.8291 - val_loss: 0.3780 - val_accuracy: 0.8340
Epoch 7/10
645/645 [==============================] - 298s 462ms/step - loss: 0.3756
- accuracy: 0.8343 - val_loss: 0.3716 - val_accuracy: 0.8381
Epoch 8/10
645/645 [==============================] - 299s 464ms/step - loss: 0.3689
- accuracy: 0.8372 - val_loss: 0.3652 - val_accuracy: 0.8383
Epoch 9/10
645/645 [==============================] - 303s 470ms/step - loss: 0.3609
- accuracy: 0.8416 - val_loss: 0.3569 - val_accuracy: 0.8441
Epoch 10/10
645/645 [==============================] - 297s 461ms/step - loss: 0.3544
- accuracy: 0.8449 - val_loss: 0.3611 - val_accuracy: 0.8432
```

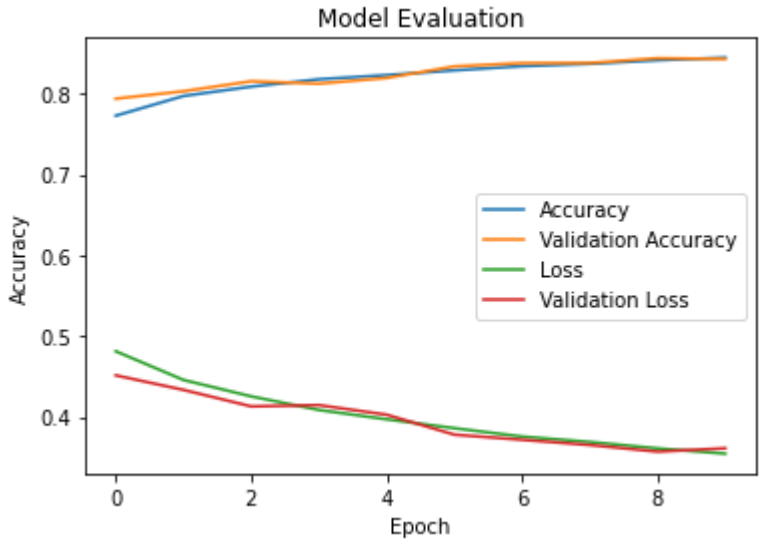
As can be seen, the validation loss went as low as 0.3611 and validation accuracy went upto 84%. The next figure shows how both the training and validation loss decreased over epochs, whereas both the training and validation accuracy increased steadily over epochs during training, with the model.

In [8]:

```
def plot_hist(hist):
  plt.plot(hist.history["accuracy"])
  plt.plot(hist.history['val_accuracy'])
  plt.plot(hist.history['loss'])
  plt.plot(hist.history['val_loss'])
  plt.title("Model Evaluation")
  plt.ylabel("Accuracy")
  plt.xlabel("Epoch")
  plt.legend(["Accuracy","Validation Accuracy","Loss","Validation Loss"])
  #plt.grid()
  plt.show()
```

In [ ]:

```
plot_hist(hist)
```



Results with Model2

Next we tried the model with batch normalization enabled and the other hyperparameters tune, as described above. The optimizer and number of epochs used were same as above.

In [12]:

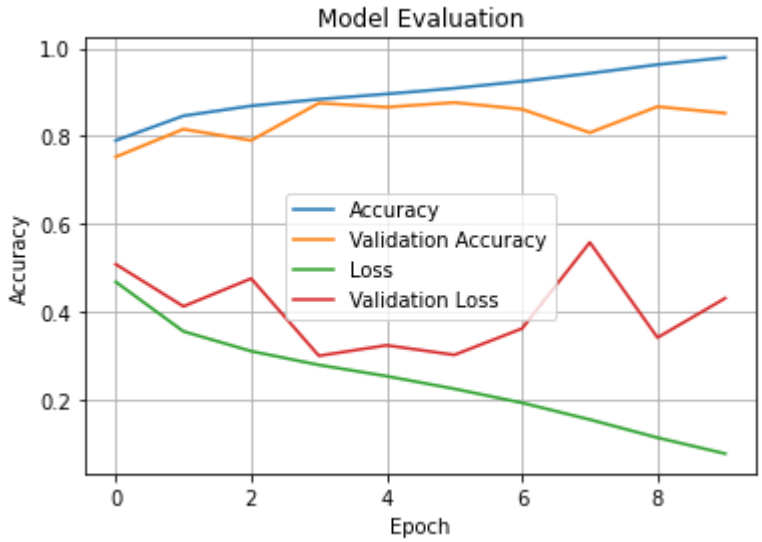
```
opt = Adam(learning_rate=0.0001)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
hist = model.fit(data_train, validation_data=data_validate, epochs=10)
```

```
Epoch 1/10
645/645 [==============================] - 339s 524ms/step - loss: 0.4674
- accuracy: 0.7898 - val_loss: 0.5076 - val_accuracy: 0.7524
Epoch 2/10
645/645 [==============================] - 295s 457ms/step - loss: 0.3545
- accuracy: 0.8459 - val_loss: 0.4115 - val_accuracy: 0.8156
Epoch 3/10
645/645 [==============================] - 290s 450ms/step - loss: 0.3094
- accuracy: 0.8687 - val_loss: 0.4750 - val_accuracy: 0.7899
Epoch 4/10
645/645 [==============================] - 296s 458ms/step - loss: 0.2780
- accuracy: 0.8837 - val_loss: 0.2989 - val_accuracy: 0.8750
Epoch 5/10
645/645 [==============================] - 295s 457ms/step - loss: 0.2524
- accuracy: 0.8958 - val_loss: 0.3228 - val_accuracy: 0.8658
Epoch 6/10
645/645 [==============================] - 295s 457ms/step - loss: 0.2236
- accuracy: 0.9088 - val_loss: 0.3009 - val_accuracy: 0.8763
Epoch 7/10
645/645 [==============================] - 292s 453ms/step - loss: 0.1917
- accuracy: 0.9246 - val_loss: 0.3610 - val_accuracy: 0.8610
Epoch 8/10
645/645 [==============================] - 293s 454ms/step - loss: 0.1535
- accuracy: 0.9428 - val_loss: 0.5574 - val_accuracy: 0.8076
Epoch 9/10
645/645 [==============================] - 289s 448ms/step - loss: 0.1121
- accuracy: 0.9628 - val_loss: 0.3404 - val_accuracy: 0.8670
Epoch 10/10
645/645 [==============================] - 296s 459ms/step - loss: 0.0758
- accuracy: 0.9787 - val_loss: 0.4302 - val_accuracy: 0.8520
```

As can be seen, the validation loss went to 4302 and validation accuracy went upto 85%. The next figure shows how both the training loss / accuracy steadily decreased / increased over epochs, resepectively, but the validation loss / accuracy became unstable.

In [14]:

```
plot_hist(hist)
```



Predictions on test images

Again a dataframe with test image ids were stored in a dataframe and the test images were automatically read during rediction phase with the function `flow_from_dataframe()` , as before. The predictions are submitted to *kaggle* to obtain the score (although leaderboard selection was disabled).

In [15]:

```
import os
images_test = pd.DataFrame({'id':os.listdir(test_dir)})
generator_test = ImageDataGenerator(rescale=1./255)

test_data = generator_test.flow_from_dataframe(
    dataframe = images_test,
    x_col='id', # filenames
    directory=test_dir,
    class_mode=None,
    batch_size=1,
    target_size=im_size,
    shuffle=False)

predictions = model.predict(test_data, verbose=1)
```

```
Found 57458 validated image filenames.
57458/57458 [==============================] - 209s 4ms/step
```

In [16]:

```
predictions = predictions.squeeze()
predictions.shape
```

Out[16]:

```
(57458,)
```

In [17]:

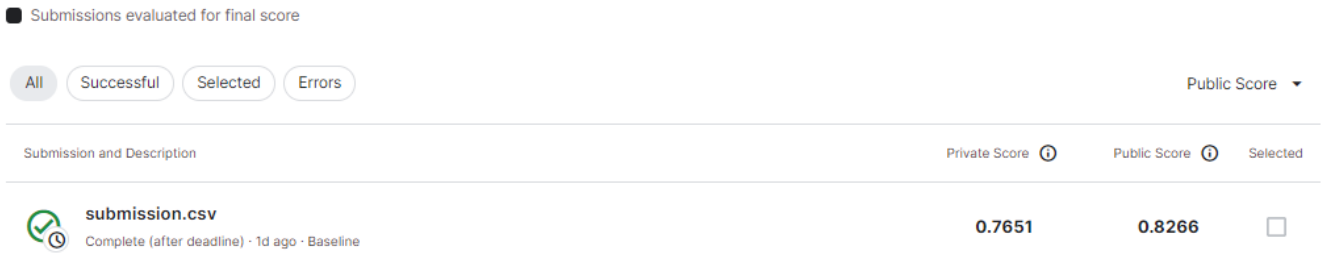
```
submission_df = pd.DataFrame()
submission_df['id'] = images_test['id'].apply(lambda x: x.split('.')[0])
submission_df['label'] = list(map(lambda x: 0 if x < 0.5 else 1, predictions))

submission_df['label'].value_counts()
submission_df.to_csv('submission2.csv', index=False)

print(submission_df.head())
```

```
                                     id  label
0  86cbac8eef45d436a8b1c7469ada0894f0b684cc      0
1  cb452f428031d335eadb8dc8eb4c7744b0cab276      0
2  6ff0a28ac41715a0646473c78b4130c64929a21d      1
3  b56127fff86222a42457749884daca4df8fef050      0
4  787a40cb598ad5f2afe00937af2488e68df2a4fc      1
```

With the first model (`CNNModel1`) ~82.7% accuracy score was obtained on the unseen test dataset in *kaggle*, as shown below, which was more than the one obtained with the second model (`CNNModel2`).



Conclusion

As we could see, the CNN model without `BatchNormalization` (`CNNModel1`) outputperformed the other model (`CNNModel2`) with one, given a small number of epochs (namely 10) were used to train both the models. It's likely that the `CNNModel2` could improve its generalizability on the unseen test images, if it were trained for more epochs (e.g. 30 epochs). We could also use popular CNN architectures such as VGG1/19, ResNet50/101, InceptionV3 or EfficientNet (either using *pre-trained* weights from *imagenet*, with *transfer learning* / *fine tuning* or training them from scratch), to get better accuracy