

CNN Cancer Detection Kaggle Mini-Project

Brief description of the problem and data

In this mini-project, we shall use **binary classification** to classify an image into cancerous (class label 1) or benign (class label 0), i.e., to identify metastatic cancer in small image patches taken from larger digital pathology scans (this problem is taken from a past Kaggle competition).

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 $32 \times 32\text{px}$ 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 [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot 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, Dense, 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]:

```
# count number of training and test images
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 images={ntest}')

#training images = 220025, #test images=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]:

```
# read the training images and ground truth labels into a dataframe
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)
```

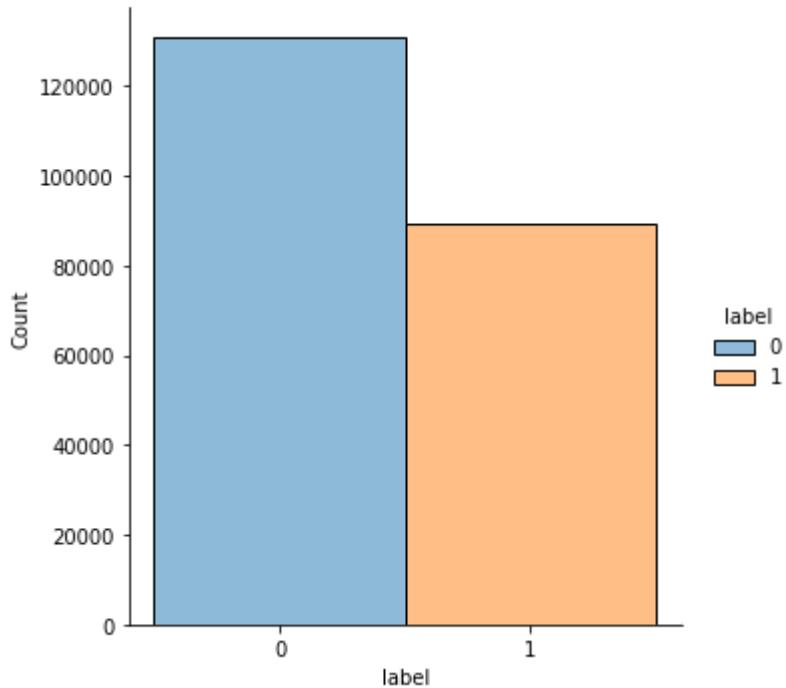
Also, as can be seen from below plot, there are around 130k benign and 89k cancerous images in the trainign dataset, hence the dataset is not very imbalanced, also the data size is large. Hence we are not using augmentation like preprocessing steps.

In []:

```
# histogram of class labels
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
```

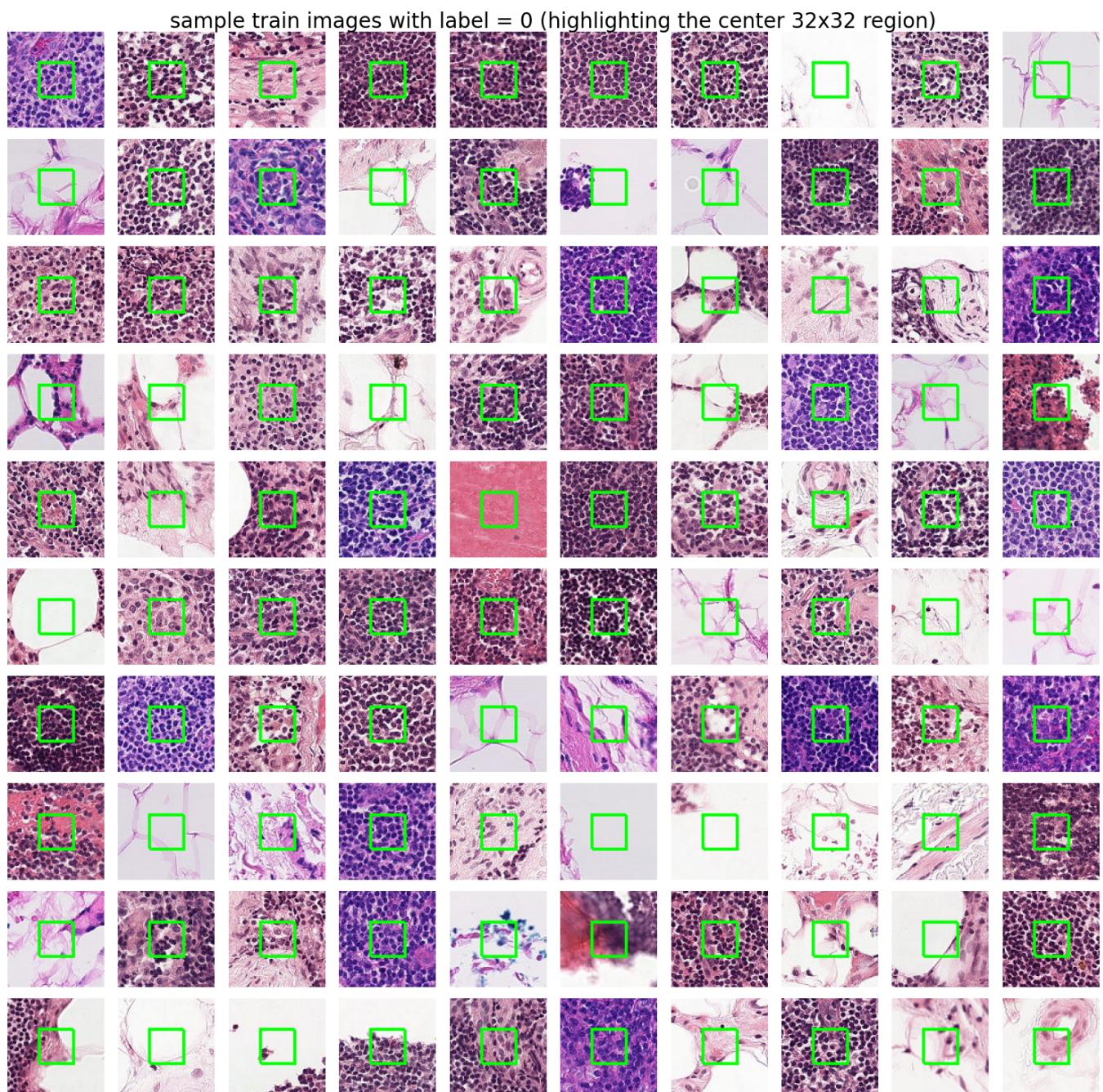


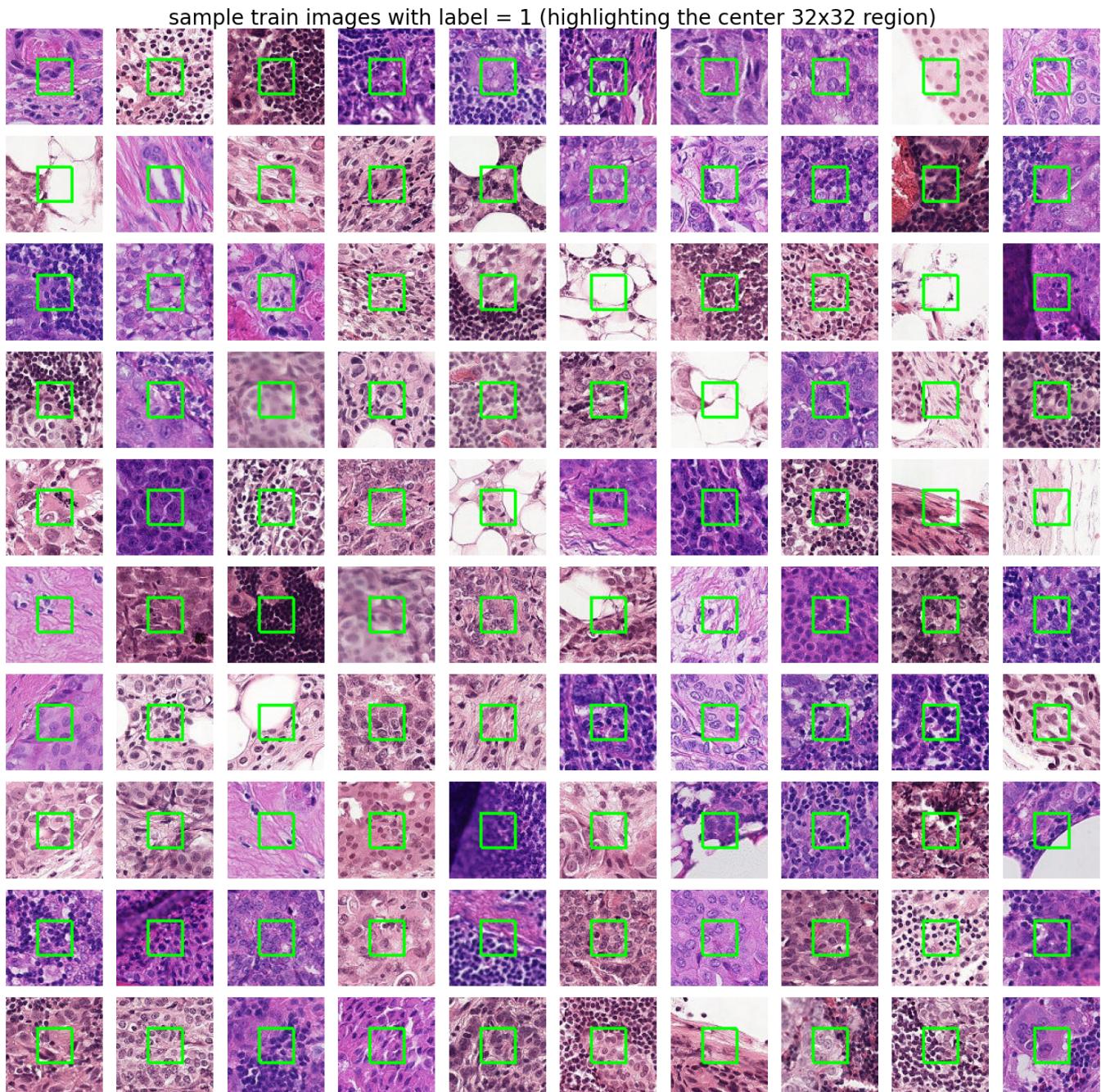
Next let's plot 100 randomly selected benign and cancerous images (with label 0 and 1, respectively, by highlighting the center 32x32 region) from training dataset to visually inspect the difference, if any.

In []:

```
def plot_images(df, label, n=100):
    row, col = draw.rectangle_perimeter(start=(96//2-32//2,96//2-32//2), end=(96//2+3
//2,96//2+32//2))
    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]}')
        for j in range(-1,2):
            im[row+j, col+j, :] = [0, 255, 0]
        plt.subplot(10,10,i+1)
        plt.imshow(im)
        plt.axis('off')
    plt.suptitle(f'sample train images with label = {label} (highlighting the center 32
x32 region)', size=20)
    plt.tight_layout()
    plt.show()

# sample and plot images from different classes
for label in train_df.label.unique():
    plot_images(train_df, label)
```





Preprocessing

- The image pixels will be scaled (normalized) in the range $[0 - 1]$, it will be done during the batch reading of the images.
- Resize the 96×96 images to size 64×64 , to reduce the memory.
- The dataset is not very imbalanced, also the data size is large. Hence we are not using augmentation like preprocessing steps.

Model Architecture

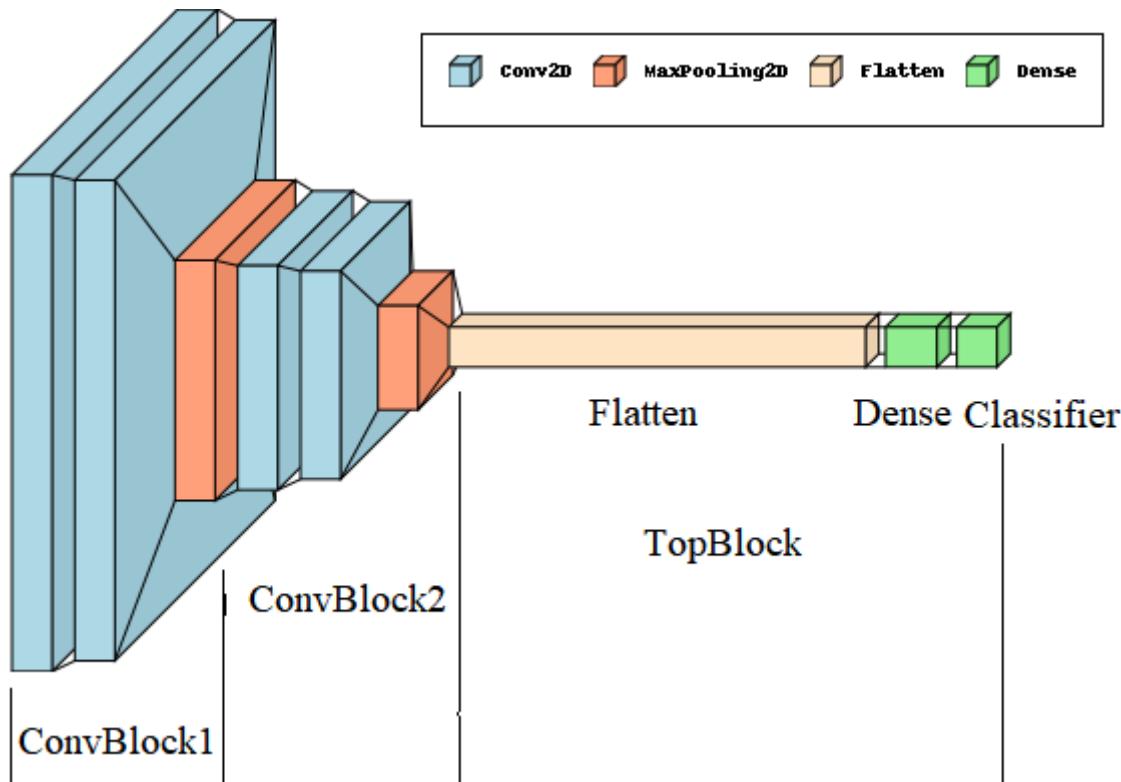
Quite a few models were trained to classify the images, starting from simpler baseline models to more complex models.

Baseline Models

We shall first implement a couple of baseline models using `keras model subclassing` with *configurable reusable blocks* (so that we can reuse them and change the *hyperparameters*) and *functional APIs*.

- The models will be traditional CNN models, which will comprise of a few Convolution/Pooling blocks (to obtain translation invariance, increase field of view, reduce number of parameters etc.) followed by flatten, Dense a binary classification layer.
- 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 (to reduce *internal covariate shift* between the batches) immediately following each of 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` non-linear activation will be used in the layers, except in the last classifier layer, which will use `sigmoid` activation (to keep the output in the range $[0 - 1]$, that can be interpreted as probability whether an image contains tumor cells or not).

The baseline model architecture 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 [2]:

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

In [3]:

```

class ConvBlock(tf.keras.layers.Layer):
    """
    implements CovBLock as
    Conv2D -> Conv2D -> MaxPool OR
    Conv2D -> BatchNorm -> Conv2D -> BatchNorm -> MaxPool
    as a reusable block in the CNN to be created
    """

    def __init__(self, n_filter, kernel_sz=(3,3), activation='relu', pool_sz=(2,2), batch_norm=False):
        # initialize with different hyperparamater values such as number of filters in Conv2D, kernel size, activation
        # BatchNorm present or not
        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):
        # forward pass
        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):
    """
    implements the top Layer of the CNN
    Flatten -> Dense -> Classification
    as a reusable block
    """

    def __init__(self, n_units=256, activation='relu', drop_out=False, drop_rate=0.5):
        # initialize the block with configurable hyperparameter values, e.g., number of neurons in Dense block, activation
        # Dropout present or not
        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):
        # forward pass
        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

- 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)
        # model top
        self.top_block = TopBlock(n_units=256)

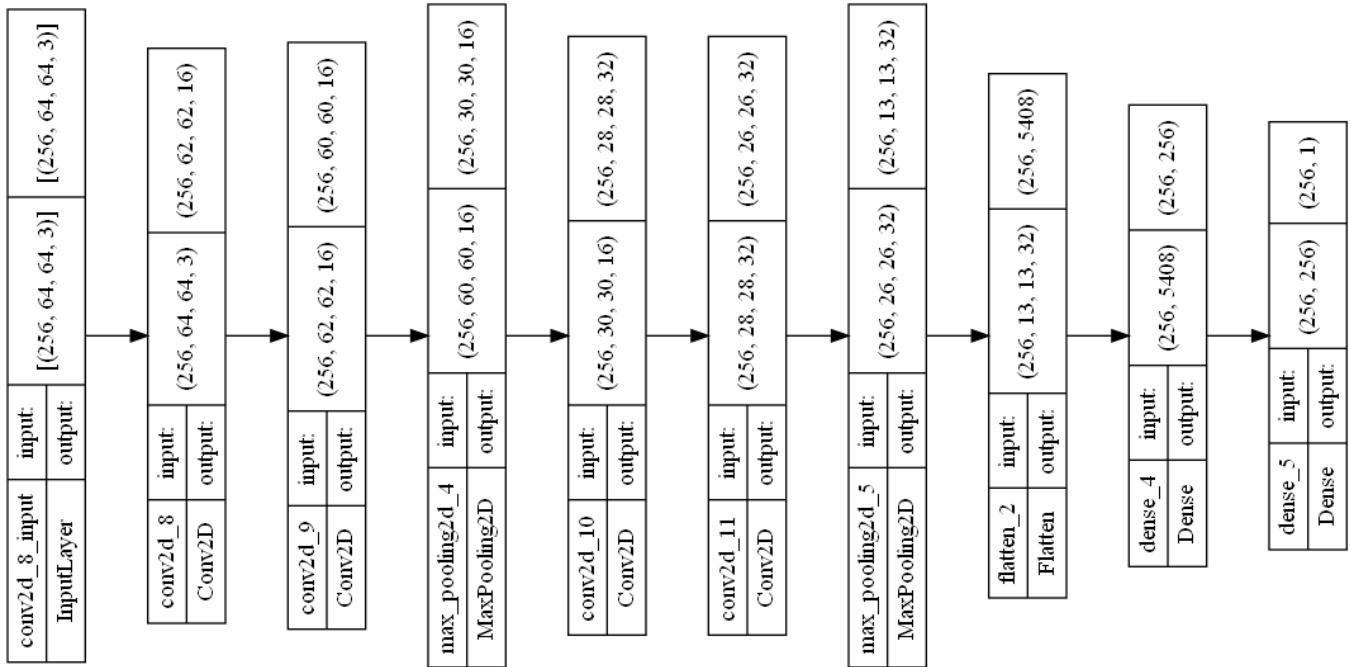
    def call(self, inputs, training=False, **kwargs):
        # forward pass
        x = self.conv_block_1(inputs)
        x = self.conv_block_2(x)
        return self.top_block(x)

# instantiate and build the model without BatchNorm
model1 = CNNModel1()
model1.build(input_shape=(batch_size,im_size[0],im_size[1],3))
model1.summary()
```

Model: "base_model1"

| Layer (type) | Output Shape | Param # |
|-----------------------------|--------------|---------|
| <hr/> | | |
| conv_block (ConvBlock) | multiple | 2768 |
| conv_block_1 (ConvBlock) | multiple | 14144 |
| top_block (TopBlock) | multiple | 1384961 |
| <hr/> | | |
| Total params: 1,401,873 | | |
| Trainable params: 1,401,745 | | |
| Non-trainable params: 128 | | |

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



CNNModel2

- 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)

# instantiate and build the model with BatchNorm
model2 = CNNModel2()
model2.build(input_shape=(batch_size,im_size,im_size,3))
model2.summary()

```

Model: "base_model2"

| Layer (type) | Output Shape | Param # |
|-----------------------------|--------------|---------|
| <hr/> | | |
| conv_block_2 (ConvBlock) | multiple | 10400 |
| conv_block_3 (ConvBlock) | multiple | 18752 |
| top_block_1 (TopBlock) | multiple | 2769921 |
| <hr/> | | |
| Total params: 2,799,073 | | |
| Trainable params: 2,798,817 | | |
| Non-trainable params: 256 | | |

We shall also use couple of popular architectures, namely, VGG16, VGG19 and ResNet50 and add a couple of layers with the models by removing the top layer, for the classification.

Model with VGG16 Backbone

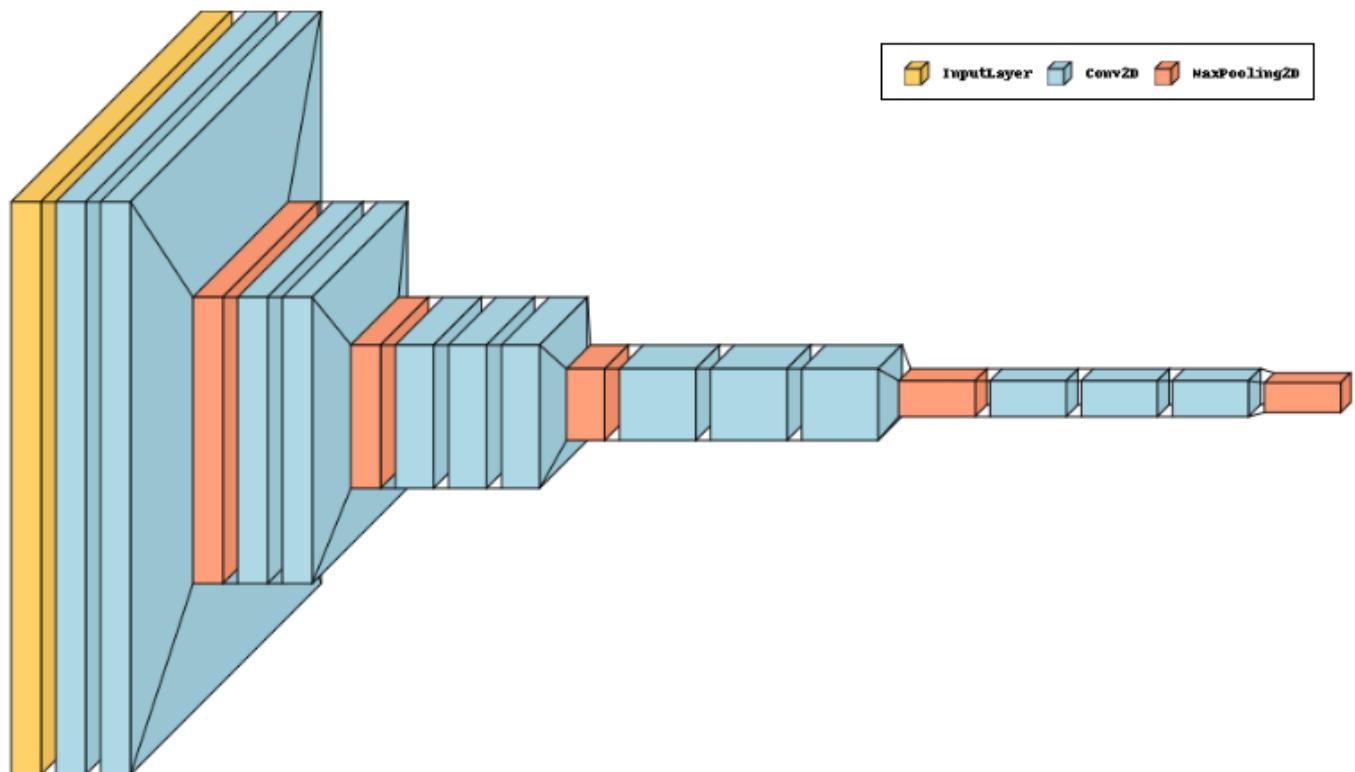
In [13]:

```

base_model = tf.keras.applications.VGG16(
    input_shape=(im_size[0],im_size[1],3),
    include_top=False,
    weights='imagenet'
)

#color_map = get_color_map()
#visualkeras.layered_view(base_model, color_map=color_map, legend=True)

```



In [31]:

```
np.random.seed(1)
tf.random.set_seed(1)
model_vgg16 = Sequential([
    base_model,
    Flatten(),
    BatchNormalization(),
    Dense(16, activation='relu'),
    Dropout(0.3),
    Dense(8, activation='relu'),
    Dropout(0.3),
    BatchNormalization(),
    Dense(1, activation='sigmoid')
], name='vgg16_backbone')
model_vgg16.summary()
```

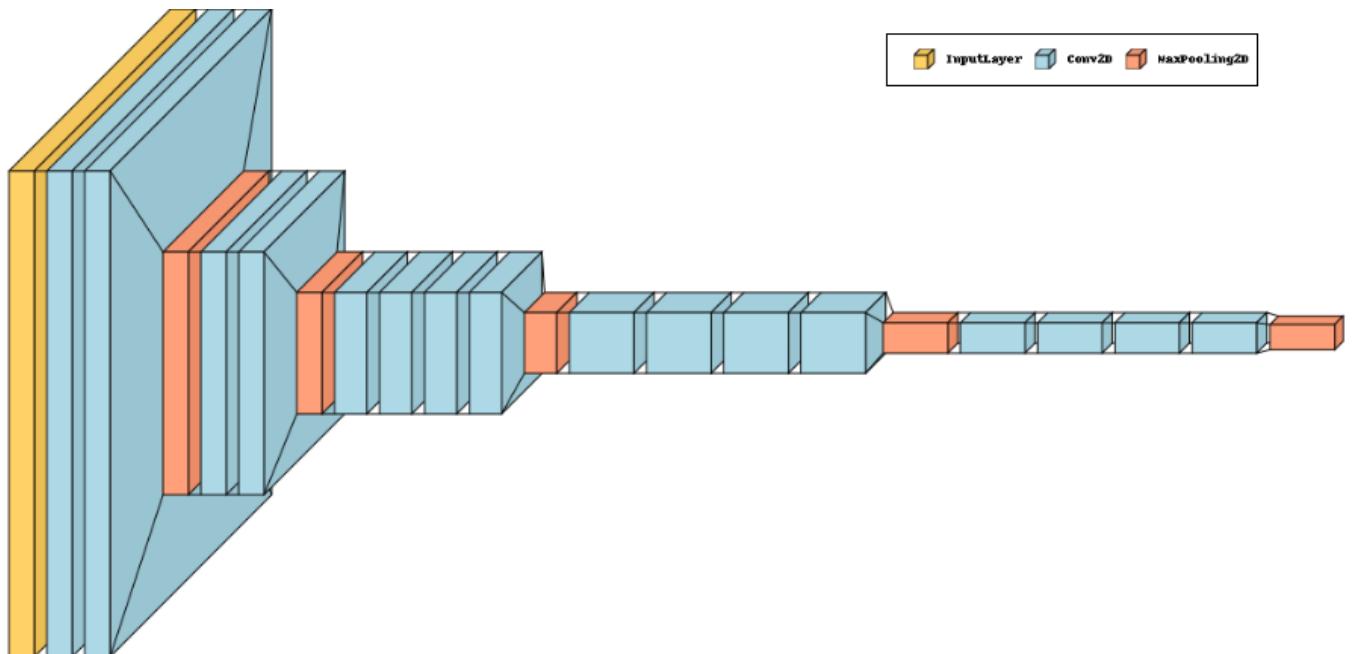
Model: "vgg16_backbone"

| Layer (type) | Output Shape | Param # |
|--|-------------------|----------|
| vgg16 (Functional) | (None, 3, 3, 512) | 14714688 |
| flatten_3 (Flatten) | (None, 4608) | 0 |
| batch_normalization_6 (BatchNormalization) | (None, 4608) | 18432 |
| dense_9 (Dense) | (None, 16) | 73744 |
| dropout_6 (Dropout) | (None, 16) | 0 |
| dense_10 (Dense) | (None, 8) | 136 |
| dropout_7 (Dropout) | (None, 8) | 0 |
| batch_normalization_7 (BatchNormalization) | (None, 8) | 32 |
| dense_11 (Dense) | (None, 1) | 9 |
| <hr/> | | |
| Total params: 14,807,041 | | |
| Trainable params: 14,797,809 | | |
| Non-trainable params: 9,232 | | |

Model with VGG19 Backbone

In []:

```
base_model = tf.keras.applications.VGG19(  
    input_shape=(im_size[0],im_size[1],3),  
    include_top=False,  
    weights='imagenet'  
)  
#color_map = get_color_map()  
#visualkeras.Layered_view(base_model, color_map=color_map, Legend=True)
```



In [15]:

```

np.random.seed(1)
tf.random.set_seed(1)

model_vgg19 = Sequential([
    base_model,
    Flatten(),
    BatchNormalization(),
    Dense(16, activation='relu'),
    Dropout(0.3),
    Dense(8, activation='relu'),
    Dropout(0.3),
    BatchNormalization(),
    Dense(1, activation='sigmoid')
], name='vgg19_backbone')

model_vgg19.summary()

```

Model: "vgg19_backbone"

| Layer (type) | Output Shape | Param # |
|--|-------------------|----------|
| <hr/> | | |
| vgg19 (Functional) | (None, 3, 3, 512) | 20024384 |
| flatten_1 (Flatten) | (None, 4608) | 0 |
| batch_normalization_2 (BatchNormalization) | (None, 4608) | 18432 |
| dense_3 (Dense) | (None, 16) | 73744 |
| dropout_2 (Dropout) | (None, 16) | 0 |
| dense_4 (Dense) | (None, 8) | 136 |
| dropout_3 (Dropout) | (None, 8) | 0 |
| batch_normalization_3 (BatchNormalization) | (None, 8) | 32 |
| dense_5 (Dense) | (None, 1) | 9 |
| <hr/> | | |
| Total params: 20,116,737 | | |
| Trainable params: 20,107,505 | | |
| Non-trainable params: 9,232 | | |

Model with ResNet50 Backbone

In [14]:

```
base_model = tf.keras.applications.ResNet50(  
    input_shape=(im_size[0],im_size[1],3),  
    include_top=False,  
    weights='imagenet'  
)  
#color_map = get_color_map()  
#visualkeras.Layered_view(base_model, color_map=color_map, Legend=True)
```

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 [=====] - 1s 0us/step



In [15]:

```
np.random.seed(1)
tf.random.set_seed(1)

model_resnet50 = Sequential([
    base_model,
    Flatten(),
    BatchNormalization(),
    Dense(16, activation='relu'),
    Dropout(0.5),
    Dense(8, activation='relu'),
    Dropout(0.5),
    BatchNormalization(),
    Dense(1, activation='sigmoid')
], 'resnet50_backbone')

model_resnet50.summary()
```

Model: "resnet50_backbone"

| Layer (type) | Output Shape | Param # |
|--|--------------------|----------|
| resnet50 (Functional) | (None, 3, 3, 2048) | 23587712 |
| flatten_1 (Flatten) | (None, 18432) | 0 |
| batch_normalization_4 (BatchNormalization) | (None, 18432) | 73728 |
| dense_2 (Dense) | (None, 16) | 294928 |
| dropout_1 (Dropout) | (None, 16) | 0 |
| dense_3 (Dense) | (None, 8) | 136 |
| dropout_2 (Dropout) | (None, 8) | 0 |
| batch_normalization_5 (BatchNormalization) | (None, 8) | 32 |
| dense_4 (Dense) | (None, 1) | 9 |

| | |
|-----------------------|------------|
| Total params: | 23,956,545 |
| Trainable params: | 23,866,545 |
| Non-trainable params: | 90,000 |

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]:

```
# scale the images to have pixel values in between [0-1]
# create 75-25 train-validation split of the training dataset for model evaluation
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. We shall use the `BCE` loss and accuracy metrics on the held-out validation dataset for model evaluation.

Results with Baseline Models

With CNNModel1

In []:

```
# compile and fit the first model with Adam optimizer, BCE Loss and to be evaluated with accuracy metric
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
```

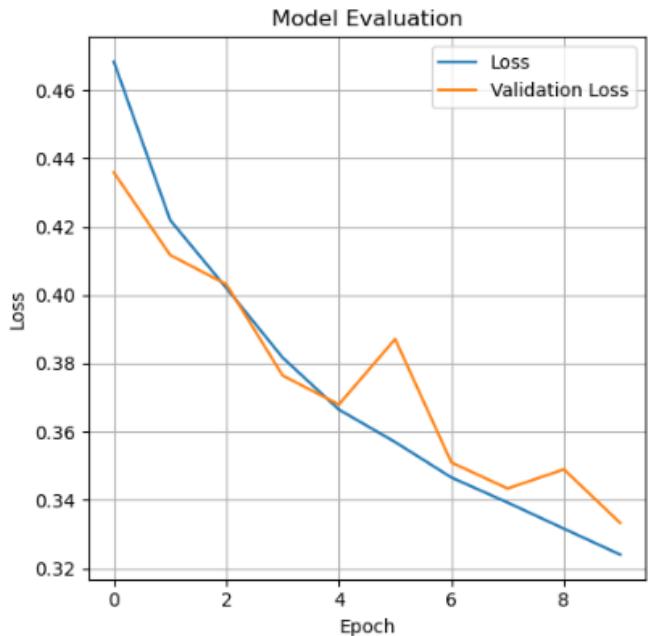
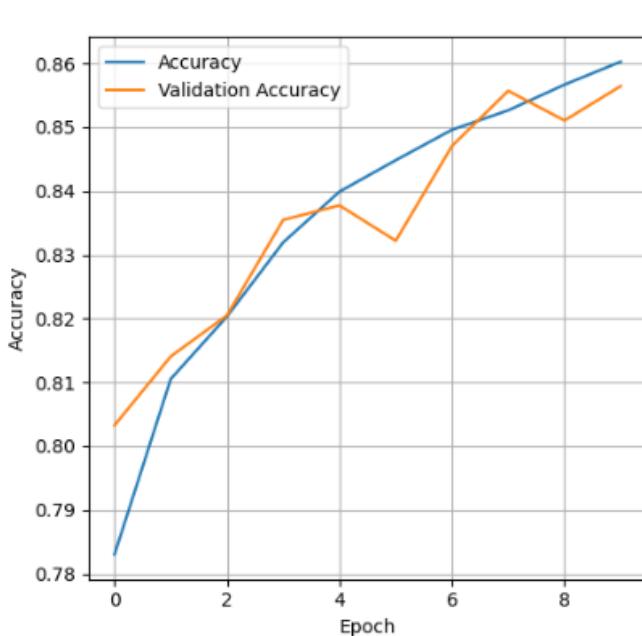
As can be seen, the validation loss went as low as 0.3611 and validation accuracy went upto 84%. The next figure shows how both the training and validation loss decreased over epochs, whereas both the training and validation accuracy increased steadily over epochs during training, with the model.

In [8]:

```
def plot_hist(hist):
    plt.figure(figsize=(10,5))
    plt.subplot(121)
    plt.plot(hist.history["accuracy"])
    plt.plot(hist.history['val_accuracy'])
    plt.legend(["Accuracy", "Validation Accuracy"])
    plt.ylabel("Accuracy")
    plt.xlabel("Epoch")
    plt.grid()
    plt.subplot(122)
    plt.plot(hist.history['loss'])
    plt.plot(hist.history['val_loss'])
    plt.title("Model Evaluation")
    plt.ylabel("Loss")
    plt.xlabel("Epoch")
    plt.grid()
    plt.legend(["Loss", "Validation Loss"])
    plt.tight_layout()
    plt.show()
```

In []:

```
plot_hist(hist)
```



with CNNModel2

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]:

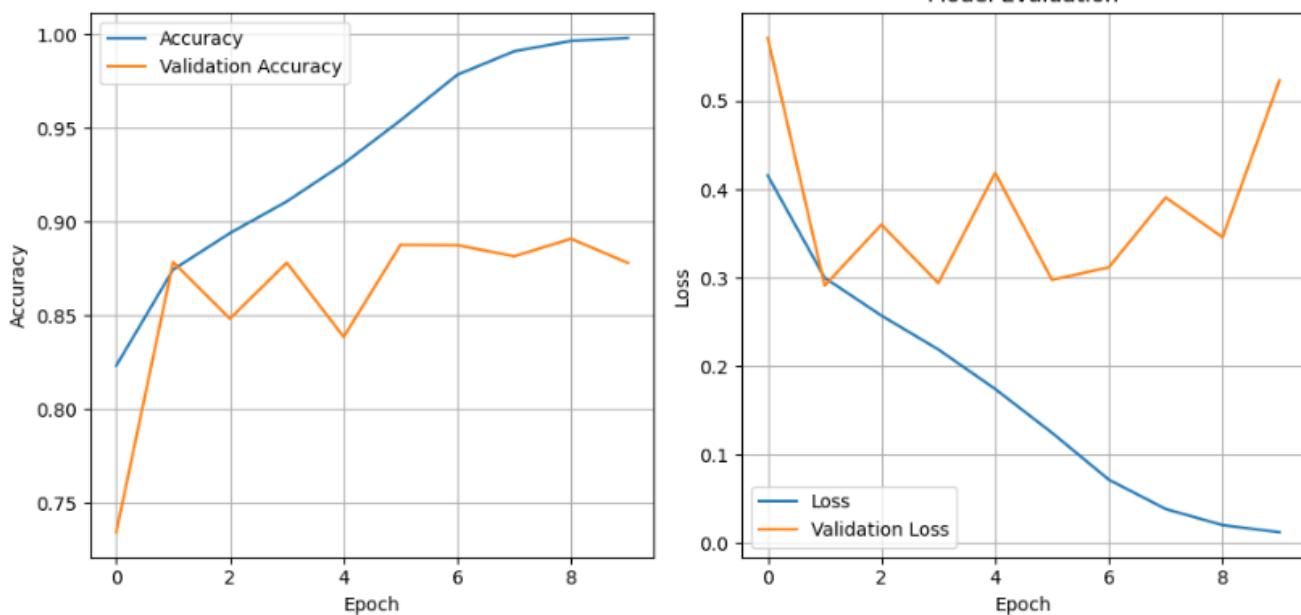
```
# compile and fit the second model with Adam optimizer, BCE Loss and to be evaluated with accuracy metric
opt = Adam(learning_rate=0.0001)
model.compile(optimizer=opt, loss='binary_crossentropy', metrics=['accuracy'])
hist = model.fit(data_train, validation_data=data_validate, epochs=10)
```

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 0.4302 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 []:

```
plot_hist(hist)
```



Results with the model with VGG16 backbone

We shall attempt transfer learning / fine tuning (by starting with pretrained weights on imagenet and keeping the VGG16 backbone weights frozen) and also train VGG16 from scratch and compare the results obtained in terms of accuracy and **ROC AUC** (area under the curve).

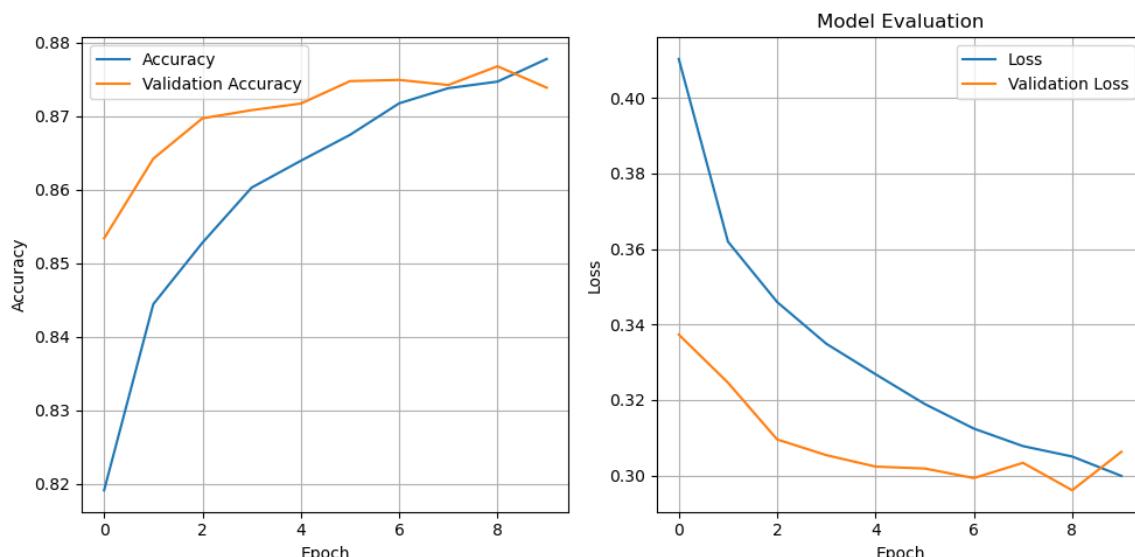
Transfer Learning / Fine Tuning

In [32]:

```
opt = tf.keras.optimizers.Adam(0.001)
base_model.trainable = False
model_vgg16.compile(loss='binary_crossentropy', optimizer=opt, metrics=[ 'accuracy', tf.keras.metrics.AUC()])
hist = model_vgg16.fit(train_data, epochs = 20, validation_data = val_data, verbose=1)
```

In [18]:

```
plot_hist(hist)
```



Training from Scratch

In [34]:

```
base_model.trainable = True
model_vgg16.compile(loss='binary_crossentropy', optimizer=opt, metrics=[ 'accuracy', tf.keras.metrics.AUC()])
```

In [35]:

```
%time  
hist = model_vgg16.fit(train_data, epochs = 20, validation_data = val_data, verbose=1)
```

Epoch 1/20

645/645 [=====] - 867s 1s/step - loss: 0.5446 - accuracy: 0.7338 - auc_6: 0.7940 - val_loss: 0.5420 - val_accuracy: 0.7202 - val_auc_6: 0.7990

Epoch 2/20

645/645 [=====] - 524s 812ms/step - loss: 0.5008 - accuracy: 0.7689 - auc_6: 0.8329 - val_loss: 0.9227 - val_accuracy: 0.6275 - val_auc_6: 0.7825

Epoch 3/20

645/645 [=====] - 543s 842ms/step - loss: 0.4721 - accuracy: 0.7896 - auc_6: 0.8551 - val_loss: 0.5896 - val_accuracy: 0.7274 - val_auc_6: 0.8574

Epoch 4/20

645/645 [=====] - 544s 842ms/step - loss: 0.4151 - accuracy: 0.8228 - auc_6: 0.8908 - val_loss: 156.7007 - val_accuracy: 0.6567 - val_auc_6: 0.8239

Epoch 5/20

645/645 [=====] - 534s 828ms/step - loss: 0.3698 - accuracy: 0.8484 - auc_6: 0.9156 - val_loss: 0.3955 - val_accuracy: 0.8148 - val_auc_6: 0.9197

Epoch 6/20

645/645 [=====] - 517s 802ms/step - loss: 0.3137 - accuracy: 0.8809 - auc_6: 0.9401 - val_loss: 124019.8984 - val_accuracy: 0.7569 - val_auc_6: 0.8264

Epoch 7/20

645/645 [=====] - 486s 752ms/step - loss: 0.2689 - accuracy: 0.9010 - auc_6: 0.9564 - val_loss: 0.2424 - val_accuracy: 0.9119 - val_auc_6: 0.9729

Epoch 8/20

645/645 [=====] - 474s 735ms/step - loss: 0.2460 - accuracy: 0.9096 - auc_6: 0.9634 - val_loss: 0.2357 - val_accuracy: 0.8903 - val_auc_6: 0.9777

Epoch 9/20

645/645 [=====] - 474s 734ms/step - loss: 0.2245 - accuracy: 0.9210 - auc_6: 0.9694 - val_loss: 0.3355 - val_accuracy: 0.8652 - val_auc_6: 0.9742

Epoch 10/20

645/645 [=====] - 476s 738ms/step - loss: 0.2222 - accuracy: 0.9214 - auc_6: 0.9699 - val_loss: 3.3083 - val_accuracy: 0.8834 - val_auc_6: 0.9826

Epoch 11/20

645/645 [=====] - 515s 798ms/step - loss: 0.1935 - accuracy: 0.9338 - auc_6: 0.9767 - val_loss: 0.1701 - val_accuracy: 0.9417 - val_auc_6: 0.9842

Epoch 12/20

645/645 [=====] - 532s 825ms/step - loss: 0.1894 - accuracy: 0.9345 - auc_6: 0.9776 - val_loss: 0.1957 - val_accuracy: 0.9417 - val_auc_6: 0.9817

Epoch 13/20

645/645 [=====] - 474s 734ms/step - loss: 0.1702 - accuracy: 0.9420 - auc_6: 0.9814 - val_loss: 0.3708 - val_accuracy: 0.9157 - val_auc_6: 0.9851

Epoch 14/20

645/645 [=====] - 470s 728ms/step - loss: 0.1556 - accuracy: 0.9480 - auc_6: 0.9840 - val_loss: 1.0746 - val_accuracy: 0.9535 - val_auc_6: 0.9858

Epoch 15/20

645/645 [=====] - 471s 729ms/step - loss: 0.1523 - accuracy: 0.9495 - auc_6: 0.9845 - val_loss: 2.1778 - val_accuracy: 0.95

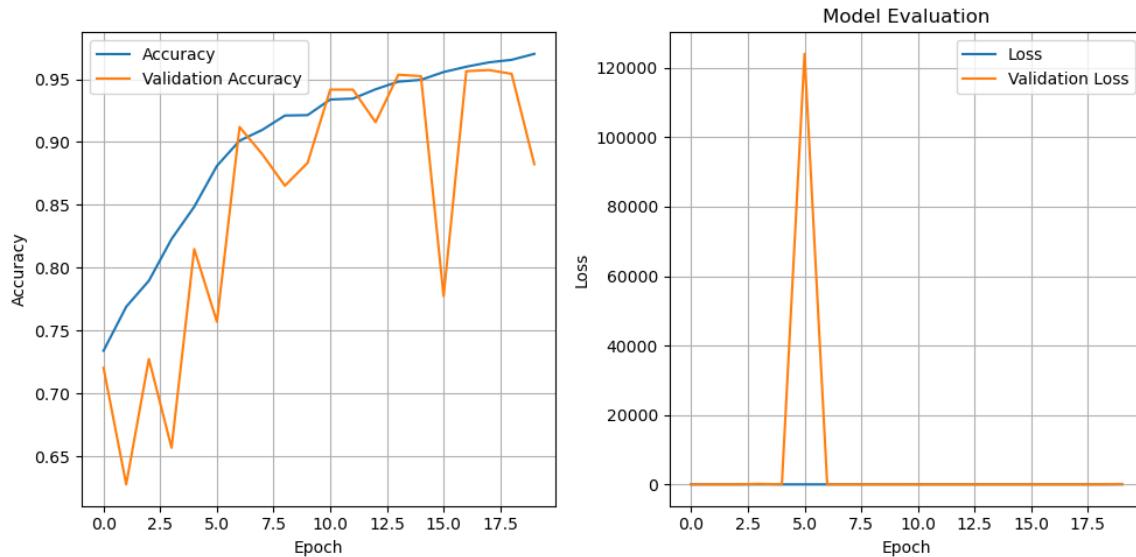
```

24 - val_auc_6: 0.9873
Epoch 16/20
645/645 [=====] - 471s 729ms/step - loss: 0.1333
- accuracy: 0.9557 - auc_6: 0.9878 - val_loss: 0.9888 - val_accuracy: 0.77
73 - val_auc_6: 0.9137
Epoch 17/20
645/645 [=====] - 474s 735ms/step - loss: 0.1226
- accuracy: 0.9599 - auc_6: 0.9896 - val_loss: 0.1557 - val_accuracy: 0.95
63 - val_auc_6: 0.9855
Epoch 18/20
645/645 [=====] - 476s 737ms/step - loss: 0.1123
- accuracy: 0.9634 - auc_6: 0.9909 - val_loss: 0.5402 - val_accuracy: 0.95
73 - val_auc_6: 0.9860
Epoch 19/20
645/645 [=====] - 508s 787ms/step - loss: 0.1068
- accuracy: 0.9654 - auc_6: 0.9915 - val_loss: 0.1696 - val_accuracy: 0.95
42 - val_auc_6: 0.9841
Epoch 20/20
645/645 [=====] - 471s 730ms/step - loss: 0.0949
- accuracy: 0.9701 - auc_6: 0.9929 - val_loss: 116.0880 - val_accuracy: 0.
8824 - val_auc_6: 0.9839
CPU times: user 2h 47min 57s, sys: 11min 9s, total: 2h 59min 7s
Wall time: 2h 52min 26s

```

In [36]:

```
plot_hist(hist)
```



Results with the model with VGG19 Backbone

This is the model that performed best and obtained ~88% public ROC score on the unseen test dataset.

Training from Scratch

In [16]:

```
opt = tf.keras.optimizers.Adam(0.001)
base_model.trainable = True
model_vgg19.compile(loss='binary_crossentropy', optimizer=opt, metrics=[ 'accuracy', tf.keras.metrics.AUC()])
```

In [17]:

```
%time  
hist = model_vgg19.fit(train_data, epochs = 20, validation_data = val_data, verbose=1)
```

Epoch 1/20

645/645 [=====] - 1362s 2s/step - loss: 0.5519 -
accuracy: 0.7240 - auc_1: 0.7850 - val_loss: 1294.4092 - val_accuracy: 0.7
835 - val_auc_1: 0.8600

Epoch 2/20

645/645 [=====] - 585s 907ms/step - loss: 0.4525
- accuracy: 0.7966 - auc_1: 0.8673 - val_loss: 1.1149 - val_accuracy: 0.65
79 - val_auc_1: 0.7463

Epoch 3/20

645/645 [=====] - 557s 863ms/step - loss: 0.3838
- accuracy: 0.8365 - auc_1: 0.9082 - val_loss: 1.0965 - val_accuracy: 0.73
96 - val_auc_1: 0.7908

Epoch 4/20

645/645 [=====] - 557s 864ms/step - loss: 0.3291
- accuracy: 0.8682 - auc_1: 0.9331 - val_loss: 0.5429 - val_accuracy: 0.71
61 - val_auc_1: 0.9106

Epoch 5/20

645/645 [=====] - 555s 859ms/step - loss: 0.2725
- accuracy: 0.8989 - auc_1: 0.9540 - val_loss: 0.2246 - val_accuracy: 0.91
34 - val_auc_1: 0.9708

Epoch 6/20

645/645 [=====] - 555s 860ms/step - loss: 0.2316
- accuracy: 0.9171 - auc_1: 0.9663 - val_loss: 0.2160 - val_accuracy: 0.91
62 - val_auc_1: 0.9709

Epoch 7/20

645/645 [=====] - 554s 858ms/step - loss: 0.2149
- accuracy: 0.9233 - auc_1: 0.9707 - val_loss: 0.2129 - val_accuracy: 0.92
02 - val_auc_1: 0.9773

Epoch 8/20

645/645 [=====] - 558s 865ms/step - loss: 0.1930
- accuracy: 0.9333 - auc_1: 0.9760 - val_loss: 0.1660 - val_accuracy: 0.93
95 - val_auc_1: 0.9820

Epoch 9/20

645/645 [=====] - 554s 858ms/step - loss: 0.1804
- accuracy: 0.9386 - auc_1: 0.9788 - val_loss: 19353.1387 - val_accuracy:
0.9063 - val_auc_1: 0.9763

Epoch 10/20

645/645 [=====] - 555s 859ms/step - loss: 0.1704
- accuracy: 0.9424 - auc_1: 0.9810 - val_loss: 35.3024 - val_accuracy: 0.8
789 - val_auc_1: 0.9433

Epoch 11/20

645/645 [=====] - 551s 854ms/step - loss: 0.1642
- accuracy: 0.9435 - auc_1: 0.9823 - val_loss: 1148.5889 - val_accuracy:
0.8950 - val_auc_1: 0.9636

Epoch 12/20

645/645 [=====] - 552s 856ms/step - loss: 0.1558
- accuracy: 0.9476 - auc_1: 0.9840 - val_loss: 0.1572 - val_accuracy: 0.94
45 - val_auc_1: 0.9827

Epoch 13/20

645/645 [=====] - 549s 851ms/step - loss: 0.1430
- accuracy: 0.9524 - auc_1: 0.9862 - val_loss: 0.1498 - val_accuracy: 0.94
57 - val_auc_1: 0.9855

Epoch 14/20

645/645 [=====] - 552s 856ms/step - loss: 0.1342
- accuracy: 0.9556 - auc_1: 0.9877 - val_loss: 0.1678 - val_accuracy: 0.94
12 - val_auc_1: 0.9827

Epoch 15/20

645/645 [=====] - 550s 852ms/step - loss: 0.1294
- accuracy: 0.9565 - auc_1: 0.9884 - val_loss: 205.0925 - val_accuracy: 0.

```

9417 - val_auc_1: 0.9852
Epoch 16/20
645/645 [=====] - 550s 853ms/step - loss: 0.1144
- accuracy: 0.9623 - auc_1: 0.9907 - val_loss: 183.5535 - val_accuracy: 0.
9473 - val_auc_1: 0.9856
Epoch 17/20
645/645 [=====] - 553s 857ms/step - loss: 0.1083
- accuracy: 0.9647 - auc_1: 0.9917 - val_loss: 132.3142 - val_accuracy: 0.
9428 - val_auc_1: 0.9800
Epoch 18/20
645/645 [=====] - 559s 867ms/step - loss: 0.1038
- accuracy: 0.9657 - auc_1: 0.9922 - val_loss: 1007.8879 - val_accuracy:
0.9205 - val_auc_1: 0.9828
Epoch 19/20
645/645 [=====] - 557s 864ms/step - loss: 0.1012
- accuracy: 0.9666 - auc_1: 0.9926 - val_loss: 121.1083 - val_accuracy: 0.
9210 - val_auc_1: 0.9599
Epoch 20/20
645/645 [=====] - 578s 896ms/step - loss: 0.0885
- accuracy: 0.9709 - auc_1: 0.9941 - val_loss: 2187.4048 - val_accuracy:
0.9541 - val_auc_1: 0.9863
CPU times: user 3h 8min 54s, sys: 13min 30s, total: 3h 22min 24s
Wall time: 3h 19min 3s

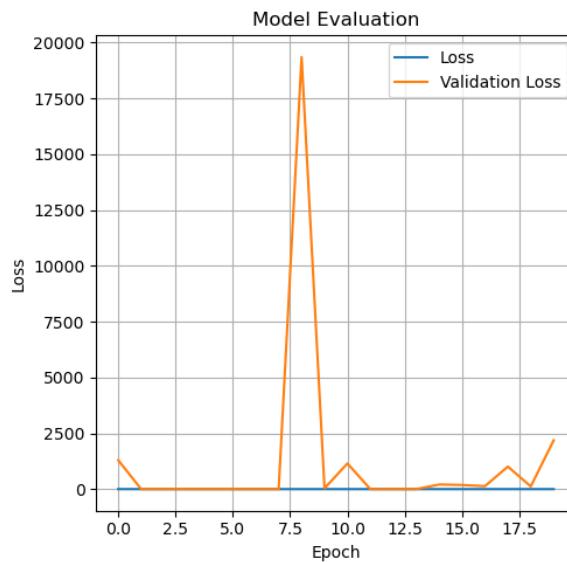
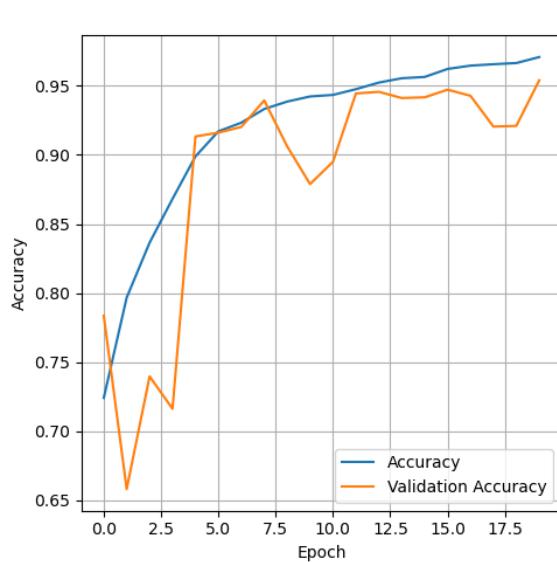
```

In [18]:

```
model_vgg19.save('model_vgg19_20.h5')
```

In [19]:

```
plot_hist(hist)
```



Results with the model with ResNet50 Backbone

In [16]:

```

base_model.trainable = True
opt = tf.keras.optimizers.Adam(0.001)
model_resnet50.compile(loss='binary_crossentropy', optimizer=opt, metrics=[ 'accuracy',
tf.keras.metrics.AUC()])

```

In [17]:

```
%time  
hist = model_resnet50.fit(train_data, validation_data=val_data, epochs=20, verbose=1)
```

Epoch 1/20

645/645 [=====] - 1317s 2s/step - loss: 0.3414 -
accuracy: 0.8588 - auc: 0.9298 - val_loss: 1.1011 - val_accuracy: 0.5951 -
val_auc: 0.3934

Epoch 2/20

645/645 [=====] - 593s 919ms/step - loss: 0.2412
- accuracy: 0.9122 - auc: 0.9653 - val_loss: 0.5169 - val_accuracy: 0.8323
- val_auc: 0.9167

Epoch 3/20

645/645 [=====] - 450s 697ms/step - loss: 0.2124
- accuracy: 0.9235 - auc: 0.9722 - val_loss: 0.2642 - val_accuracy: 0.8980
- val_auc: 0.9597

Epoch 4/20

645/645 [=====] - 448s 695ms/step - loss: 0.1941
- accuracy: 0.9305 - auc: 0.9766 - val_loss: 0.3500 - val_accuracy: 0.8499
- val_auc: 0.9337

Epoch 5/20

645/645 [=====] - 446s 692ms/step - loss: 0.1791
- accuracy: 0.9353 - auc: 0.9795 - val_loss: 0.6030 - val_accuracy: 0.7776
- val_auc: 0.8575

Epoch 6/20

645/645 [=====] - 449s 696ms/step - loss: 0.1673
- accuracy: 0.9416 - auc: 0.9823 - val_loss: 1.3558 - val_accuracy: 0.7927
- val_auc: 0.8046

Epoch 7/20

645/645 [=====] - 453s 701ms/step - loss: 0.1497
- accuracy: 0.9474 - auc: 0.9852 - val_loss: 0.4392 - val_accuracy: 0.8672
- val_auc: 0.9208

Epoch 8/20

645/645 [=====] - 463s 718ms/step - loss: 0.1431
- accuracy: 0.9497 - auc: 0.9863 - val_loss: 0.3357 - val_accuracy: 0.9010
- val_auc: 0.9642

Epoch 9/20

645/645 [=====] - 454s 703ms/step - loss: 0.1331
- accuracy: 0.9531 - auc: 0.9879 - val_loss: 1.3040 - val_accuracy: 0.7925
- val_auc: 0.8910

Epoch 10/20

645/645 [=====] - 490s 760ms/step - loss: 0.1234
- accuracy: 0.9571 - auc: 0.9893 - val_loss: 0.3096 - val_accuracy: 0.9203
- val_auc: 0.9672

Epoch 11/20

645/645 [=====] - 451s 698ms/step - loss: 0.1124
- accuracy: 0.9605 - auc: 0.9909 - val_loss: 0.9762 - val_accuracy: 0.7699
- val_auc: 0.8315

Epoch 12/20

645/645 [=====] - 458s 710ms/step - loss: 0.1117
- accuracy: 0.9607 - auc: 0.9909 - val_loss: 1.2930 - val_accuracy: 0.7756
- val_auc: 0.7858

Epoch 13/20

645/645 [=====] - 503s 779ms/step - loss: 0.1023
- accuracy: 0.9643 - auc: 0.9922 - val_loss: 0.3675 - val_accuracy: 0.9167
- val_auc: 0.9695

Epoch 14/20

645/645 [=====] - 507s 786ms/step - loss: 0.0978
- accuracy: 0.9653 - auc: 0.9927 - val_loss: 0.8825 - val_accuracy: 0.7811
- val_auc: 0.9036

Epoch 15/20

645/645 [=====] - 477s 740ms/step - loss: 0.0956
- accuracy: 0.9667 - auc: 0.9931 - val_loss: 0.6381 - val_accuracy: 0.8942

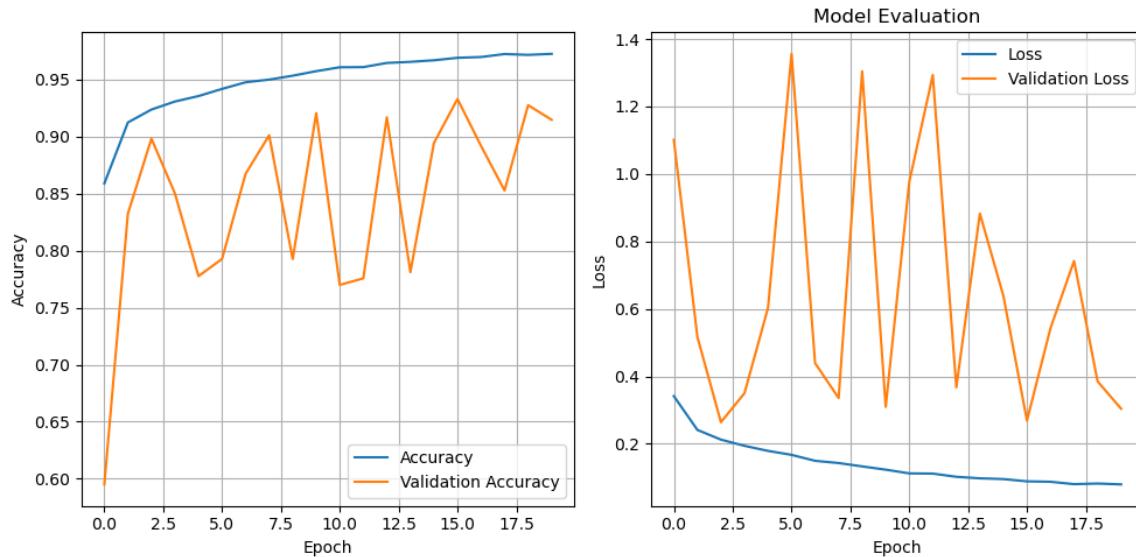
```

- val_auc: 0.9421
Epoch 16/20
645/645 [=====] - 451s 699ms/step - loss: 0.0888
- accuracy: 0.9688 - auc: 0.9938 - val_loss: 0.2691 - val_accuracy: 0.9328
- val_auc: 0.9725
Epoch 17/20
645/645 [=====] - 461s 714ms/step - loss: 0.0876
- accuracy: 0.9695 - auc: 0.9940 - val_loss: 0.5430 - val_accuracy: 0.8916
- val_auc: 0.9415
Epoch 18/20
645/645 [=====] - 451s 699ms/step - loss: 0.0805
- accuracy: 0.9721 - auc: 0.9948 - val_loss: 0.7418 - val_accuracy: 0.8526
- val_auc: 0.9025
Epoch 19/20
645/645 [=====] - 445s 690ms/step - loss: 0.0823
- accuracy: 0.9714 - auc: 0.9946 - val_loss: 0.3851 - val_accuracy: 0.9273
- val_auc: 0.9635
Epoch 20/20
645/645 [=====] - 442s 684ms/step - loss: 0.0798
- accuracy: 0.9723 - auc: 0.9948 - val_loss: 0.3040 - val_accuracy: 0.9145
- val_auc: 0.9710
CPU times: user 2h 45min 23s, sys: 10min 53s, total: 2h 56min 16s
Wall time: 2h 51min 7s

```

In [18]:

```
plot_hist(hist)
```



As seen from the above figure, the ResNet50 model's accuracy and loss both are quite fluctuating not stable and need more training.

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) # scale the test images to have pixel values in [0-1]

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)

# predict with the model
predictions = model1.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]:

```

# create submission dataframe for kaggle submission
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 baseline model (CNNModel1) ~82.7% ROC score was obtained on the unseen test dataset in *kaggle*, as shown below, which was more than the one obtained with the second baseline model (CNNModel2).
- The ROC scores obtained with the models with VGG16, VGG19 backbones were much better and the results obtained with them by training from scratch were better than those obtained with transfer learning / fine tuning (since we have a huge dataset) - the best score was obtained with the model with VGG19 backbone trained from scratch.

Git Repository

<https://github.com/sandipan/Coursera-Deep-Learning-Histopathologic-Cancer-Detection-Project>
[\(https://github.com/sandipan/Coursera-Deep-Learning-Histopathologic-Cancer-Detection-Project\)](https://github.com/sandipan/Coursera-Deep-Learning-Histopathologic-Cancer-Detection-Project)

Kaggle Notebook

<https://www.kaggle.com/code/sandipanumbc/vgg16-19-resnet50>
[\(https://www.kaggle.com/code/sandipanumbc/vgg16-19-resnet50\)](https://www.kaggle.com/code/sandipanumbc/vgg16-19-resnet50)

My Blog

<https://sandipanweb.wordpress.com/2023/03/26/histopathologic-cancer-detection-with-cnn/>
[\(https://sandipanweb.wordpress.com/2023/03/26/histopathologic-cancer-detection-with-cnn/\)](https://sandipanweb.wordpress.com/2023/03/26/histopathologic-cancer-detection-with-cnn/)

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 have also trained popular CNN architectures such as VGG1/19 both using *pre-trained* weights from *imagenet*, with *transfer learning / fine tuning* and also training them from scratch, and we obtained much better accuracy, particularly with these models trained from scratch. We could try more recent and complex models such as ResNet50/101, InceptionV3 or EfficientNet too.