CNN Cancer Detection Kaggle Mini-Project

Description of the problem/data

For this week's mini-project, you will participate in this Kaggle competition: Histopathologic Cancer Detection

This Kaggle competition is a binary image classification problem where you will identify metastatic cancer in small image patches taken from larger digital pathology scans.

In this competition, you must create an algorithm to identify metastatic cancer in small image patches taken from larger digital pathology scans. The data for this competition is a slightly modified version of the PatchCamelyon (PCam) benchmark dataset (the original PCam dataset contains duplicate images due to its probabilistic sampling, however, the version presented on Kaggle does not contain duplicates).

PCam is highly interesting for both its size, simplicity to get started on, and approachability.

Dataset Description

In this dataset, you are provided with 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. You are predicting the labels for the images in the test folder. A positive label indicates that the center 32x32px 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. This outer region is provided to enable fully-convolutional models that do not use zero-padding, to ensure consistent behavior when applied to a whole-slide image.

Exploratory data analysis (EDA)

To begin, we'll import the necessary Python libraries and functions essential for constructing the CNN model. The deep learning framework we'll be using is `tensorflow` with its high-level API, `keras`, for training.

```
import warnings
warnings.filterwarnings('ignore')
import os
import numpy as np
import pandas as pd
import matplotlib.pylab as plt
import seaborn as sns

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import (
    Conv2D,
    MaxPool2D,
    BatchNormalization,
    Flatten,
    Dropout,
    Dense,
    Activation,
    Input
)

from tifffile import imread
from skimage import draw
from skimage.draw import rectangle_perimeter
```

The training folder, containing images holds approximately 220,000 samples, while the test folder includes around 57,000.

```
# Determine the total number of images available for training and
testing
base_dir = '/kaggle/input/histopathologic-cancer-detection'
train_dir = os.path.join(base_dir, 'train')
test_dir = os.path.join(base_dir, 'test')

num_train_images = len(os.listdir(train_dir))
num_test_images = len(os.listdir(test_dir))

print(f'#training images = {num_train_images}, #test images =
{num_test_images}')

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

The `train_labels.csv` file has been loaded into a `pandas` `DataFrame`, with its initial rows displayed below. This file maps each `id` to a corresponding `.tif` image in the training directory, along with its ground-truth label. For convenience in later file access, we'll append the `.tif` extension to each value in the `id` column so that it directly matches the image filenames.

```
# Load image metadata and labels into a DataFrame, formatting 'id' to
match filenames
train_labels_path = os.path.join(base_dir, 'train_labels.csv')
train_df = pd.read_csv(train_labels_path)

train_df['label'] = train_df['label'].astype(str)
train_df['id'] = train_df['id'].apply(lambda x: x if
x.endswith('.tif') else f'{x}.tif')

train_df.head()
```

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

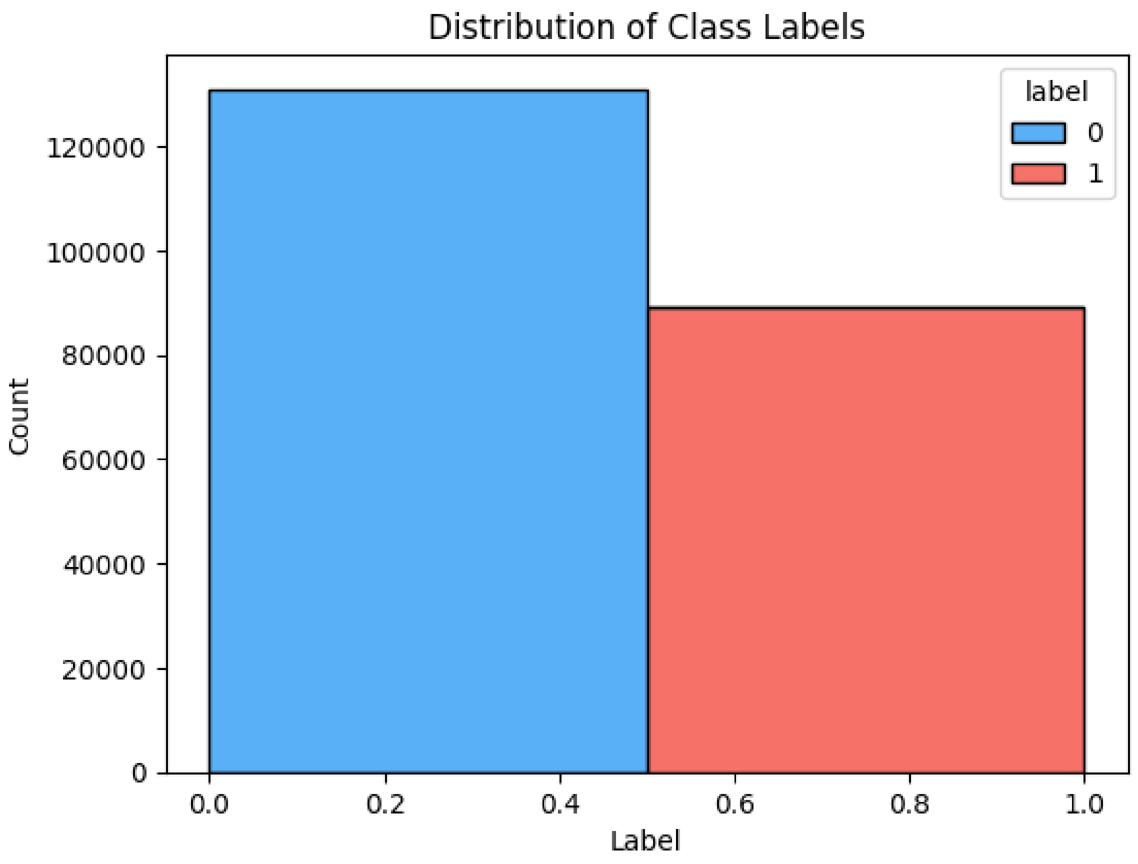
Each image is an RGB color file with dimensions 96x96 pixels. As demonstrated in the following code snippet, the images are loaded using the `imread()` function.

The plot below shows that the training dataset comprises approximately 130,000 benign and 89,000 cancerous images. Given the relatively balanced class distribution and the substantial dataset size, we've opted to forgo augmentation-based preprocessing.

```
# Load the training image and display its dimensions (height, width,
channels)
imread(f'{train_dir}/{train_df.id[0]}').shape
```

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

```
train_df['label'] = train_df['label'].astype(int)

# Define blue and red palette
custom_palette = {0: '#2196F3', 1: '#F44336'}  # blue for benign, red
for cancerous

sns.histplot(
    data=train_df,
    x='label',
    hue='label',
    multiple='stack',
    bins=2,
    palette=custom_palette
)
plt.title('Distribution of Class Labels')
plt.xlabel('Label')
plt.ylabel('Count')
plt.show()

label_counts = train_df['label'].value_counts()
print(label_counts)
```



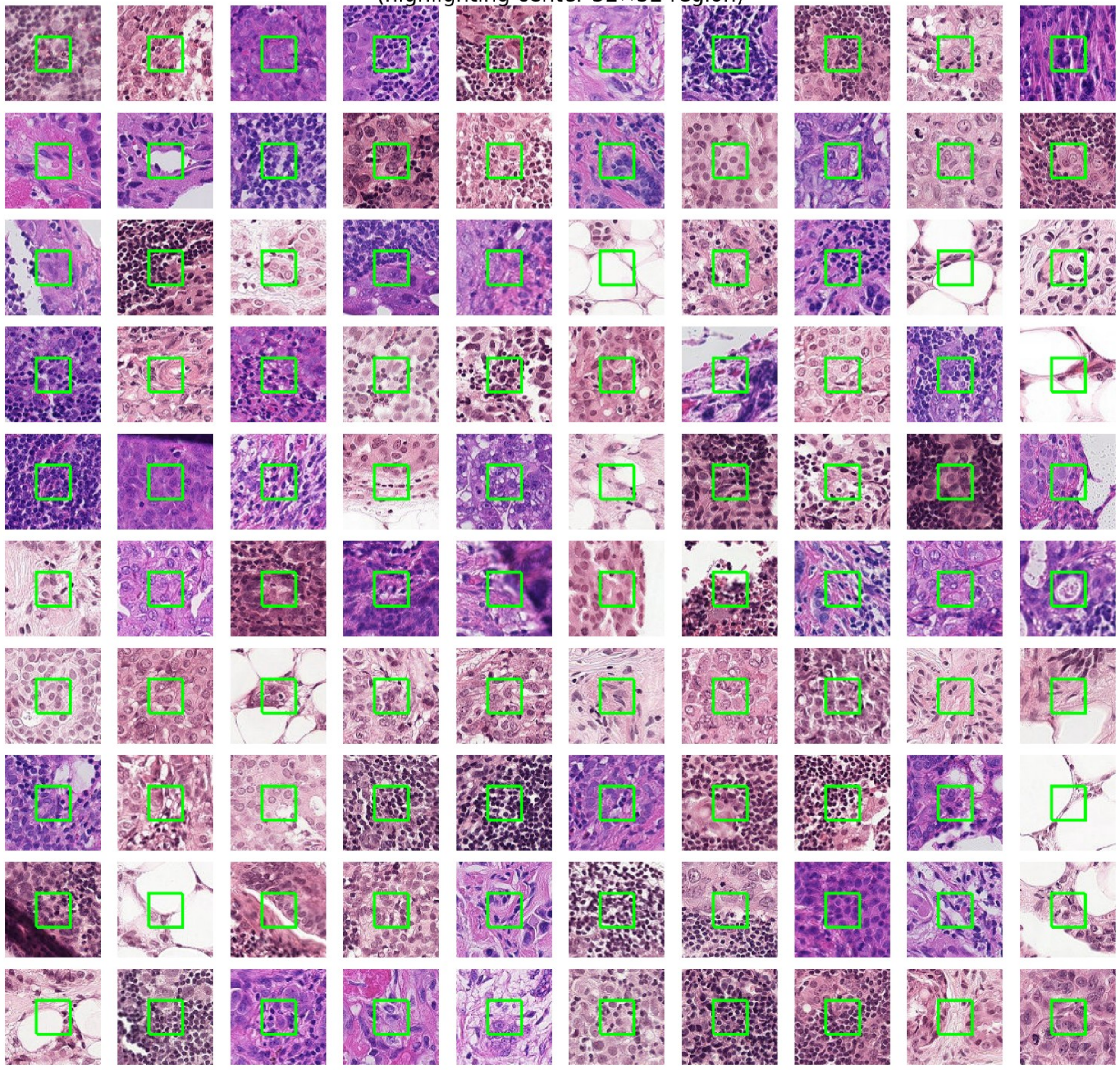
```
label
0    130908
1     89117
Name: count, dtype: int64
```

We now visualize and sample 100 randomly selected training images. 50 benign (label 0) and 50 cancerous (label 1), highlighting their central 32×32 region to examine potential visual distinctions between the two classes.

```python
def plot_images(df, label, n=100, highlight_size=32):
    # Compute coordinates for center highlight box
    center = 96 // 2
    half_size = highlight_size // 2
    start = (center - half_size, center - half_size)
    end = (center + half_size, center + half_size)
    row, col = rectangle_perimeter(start=start, end=end)

    # Sample images for the given label
    df_subset = df[df['label'] == label].sample(n, random_state=42)
    image_ids = df_subset['id'].values

    # Set up the plot grid
    plt.figure(figsize=(15, 15))
    for i, image_id in enumerate(image_ids):
        image = imread(f'{train_dir}/{image_id}')

        # Overlay green border around central region
        for offset in range(-1, 2):
            image[row + offset, col + offset] = [0, 255, 0]

        plt.subplot(10, 10, i + 1)
        plt.imshow(image)
        plt.axis('off')

    plt.suptitle(
        f'Sample training images with label = {label}\n(highlighting
center {highlight_size}×{highlight_size} region)',
        fontsize=20
    )
    plt.tight_layout()
    plt.show()

# Apply function to each unique class label
for label_val in sorted(train_df['label'].unique()):
    plot_images(train_df, label_val)
```

Sample training images with label = 0
(highlighting center 32×32 region)

Sample training images with label = 1
(highlighting center 32×32 region)



Procedure and Analysis

- Image pixel values are normalized to the [0, 1] range during batch loading.
- The original 96 x 96 images are resized to 64 x 64 to reduce memory usage.
- Since the dataset is relatively balanced and sufficiently large, augmentation-based preprocessing is not applied.
- A range of models were trained to classify the images, progressing from basic baseline models to more sophisticated architectures.

Baseline Models

We begin by implementing a set of baseline models using Keras model subclassing, incorporating configurable and reusable blocks alongside functional APIs.

- These models follow a traditional CNN architecture, consisting of several convolution and pooling blocks to enhance translation invariance, expand the receptive field, and reduce parameter count. This is followed by a flattening layer, dense layers, and a final binary classification output.
- The ConvBlock class defines a reusable convolutional block with two Conv2D layers of equal size, optionally followed by BatchNormalization layers to mitigate internal covariate shift. A MaxPool2D layer concludes the block. Parameters such as filter size, kernel size, activation function, pooling size, and batch normalization usage are all configurable with sensible defaults.
- The TopBlock class serves as the final stage, containing a Flatten layer and two Dense layers. The last Dense layer acts as the classifier with a single output neuron. The size of the intermediate Dense layer is also configurable.
- ReLU is used as the default activation function throughout, except for the final classification layer, which uses a sigmoid activation to produce a probability score indicating the presence of tumor cells.

The `batch_size` and `im_size` are configurable hyperparameters. In this setup, we use a batch size of 64 and resize all images to 64 × 64 to conserve memory.

```
batch_size, im_size = 64, 64

# Custom convolutional block
class ConvBlock(tf.keras.layers.Layer):
    def __init__(self, n_filter, kernel_sz=(3, 3), activation='relu',
                 pool_sz=(2, 2), batch_norm=False):
        super().__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)

# Custom top classification block
class TopBlock(tf.keras.layers.Layer):
    def __init__(self, n_units=256, activation='relu',
                 drop_out=False, drop_rate=0.5):
        super().__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 self.drop_out and training:
            x = self.dropout(x)
        x = self.dense(x)
        return self.classifier(x)

# CNN model using the custom blocks
class CNNModel1(tf.keras.Model):
    def __init__(self, input_shape=(64, 64, 3), n_class=1):
        super().__init__()
        self.conv_block_1 = ConvBlock(n_filter=16)
        self.conv_block_2 = ConvBlock(n_filter=32)
        self.top_block = TopBlock(n_units=256)

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

A series of models will be constructed by varying Conv2D filter sizes and Dense layer dimensions, exploring configurations with and without normalization.

CNNModel1

- The first model, CNNModel1, excludes BatchNormalization and employs ConvBlock sizes of 16 and 32, with a Dense layer containing 256 units. This configuration is illustrated in the subsequent code.

```
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 the model
model1 = CNNModel1()

# Run a dummy forward pass to build the model
dummy_input = tf.random.normal((batch_size, im_size, im_size, 3))
_ = model1(dummy_input) # Trigger dynamic shape inference

# Display the model summary
model1.summary()

Model: "cnn_model1"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ conv_block (ConvBlock)          │ ?                      │
2,768 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ conv_block_1 (ConvBlock)        │ ?                      │
13,888 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ top_block (TopBlock)            │ ?                      │
1,384,961 │
└─────────────────────────────────┴────────────────────────┴──────────
─────┘

 Total params: 1,401,617 (5.35 MB)

 Trainable params: 1,401,617 (5.35 MB)

 Non-trainable params: 0 (0.00 B)
```

CNNModel2

- CNNModel2 is the second model, incorporating BatchNormalization. It uses ConvBlock layers with 32 filters each and a Dense layer with 512 units. The model is defined in the following code snippet

```
class CNNModel2(tf.keras.Model):
    def __init__(self, input_shape=(im_size, im_size, 3), n_class=1):
        super().__init__()

        # Convolutional blocks with BatchNormalization enabled
        self.conv_block_1 = ConvBlock(n_filter=32, batch_norm=True)
        self.conv_block_2 = ConvBlock(n_filter=32, batch_norm=True)

        # Top classification block with 512 units
        self.top_block = TopBlock(n_units=512)

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


# Instantiate and build the model
model2 = CNNModel2()
dummy_input = tf.random.normal((batch_size, im_size, im_size, 3))
_ = model2(dummy_input)  # Trigger dynamic shape inference
model2.summary()

Model: "cnn_model2"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━
━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━
━━━━━┩
│ conv_block_2 (ConvBlock)        │ ?                      │
10,400 │
├─────────────────────────────────┼────────────────────────┼───────────────
─────┤
│ conv_block_3 (ConvBlock)        │ ?                      │
18,752 │
├─────────────────────────────────┼────────────────────────┼───────────────
─────┤
│ top_block_1 (TopBlock)          │ ?                      │
2,769,921 │
└─────────────────────────────────┴────────────────────────┴───────────────
─────┘

 Total params: 2,799,073 (10.68 MB)

 Trainable params: 2,798,817 (10.68 MB)

 Non-trainable params: 256 (1.00 KB)
```

In addition, we utilize popular pretrained models such as VGG16, VGG19, and ResNet50. Their top layers are removed, and custom classification layers are added to adapt them to our task.

VGG16 Backbone Model

```
# Set reproducibility seeds
np.random.seed(1)
tf.random.set_seed(1)

# Define input shape
input_shape = (im_size, im_size, 3)  # assuming im_size is a scalar
like 64

# Load VGG16 base model without top layers
base_model = tf.keras.applications.VGG16(
    input_shape=input_shape,
    include_top=False,
    weights='imagenet'
)

# Optionally freeze base model layers to use it as a fixed feature
extractor
base_model.trainable = False

# Build the model using Sequential API
model_vgg16 = Sequential([
    base_model,                          # Pretrained convolutional base
    Flatten(),                           # Flatten feature maps to 1D
vector
    BatchNormalization(),               # Normalize activations
    Dense(16, activation='relu'),       # First dense layer
    Dropout(0.3),                        # Regularization
    Dense(8, activation='relu'),        # Second dense layer
    Dropout(0.3),                        # Additional regularization
    BatchNormalization(),               # Normalize before final output
    Dense(1, activation='sigmoid')      # Output layer for binary
classification
], name='vgg16_backbone')

# Display model summary
model_vgg16.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

Model: "vgg16_backbone"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
```

```
│ vgg16 (Functional)                │ (None, 2, 2, 512)        │
14,714,688 │
├───────────────────────────────────┼──────────────────────────┼──────────
───────┤
│ flatten_2 (Flatten)               │ (None, 2048)             │
0 │
├───────────────────────────────────┼──────────────────────────┼──────────
───────┤
│ batch_normalization_8             │ (None, 2048)             │
8,192 │
│ (BatchNormalization)              │                          │
│
├───────────────────────────────────┼──────────────────────────┼──────────
───────┤
│ dense_4 (Dense)                   │ (None, 16)               │
32,784 │
├───────────────────────────────────┼──────────────────────────┼──────────
───────┤
│ dropout_2 (Dropout)               │ (None, 16)               │
0 │
├───────────────────────────────────┼──────────────────────────┼──────────
───────┤
│ dense_5 (Dense)                   │ (None, 8)                │
136 │
├───────────────────────────────────┼──────────────────────────┼──────────
───────┤
│ dropout_3 (Dropout)               │ (None, 8)                │
0 │
├───────────────────────────────────┼──────────────────────────┼──────────
───────┤
│ batch_normalization_9             │ (None, 8)                │
32 │
│ (BatchNormalization)              │                          │
│
├───────────────────────────────────┼──────────────────────────┼──────────
───────┤
│ dense_6 (Dense)                   │ (None, 1)                │
9 │
└───────────────────────────────────┴──────────────────────────┴──────────
───────┘

 Total params: 14,755,841 (56.29 MB)

 Trainable params: 37,041 (144.69 KB)

 Non-trainable params: 14,718,800 (56.15 MB)
```

VGG19 Backbone Model

```
# Set reproducibility seeds
np.random.seed(1)
tf.random.set_seed(1)

# Define input shape
input_shape = (im_size, im_size, 3)  # assuming im_size is a scalar
like 64

# Load VGG19 base model without top layers
base_model = tf.keras.applications.VGG19(
    input_shape=input_shape,
    include_top=False,
    weights='imagenet'
)

# Optionally freeze base model layers
base_model.trainable = False

# Build custom classification head
inputs = Input(shape=input_shape)
x = base_model(inputs, training=False)
x = Flatten()(x)
x = BatchNormalization()(x)
x = Dense(16, activation='relu')(x)
x = Dropout(0.3)(x)
x = Dense(8, activation='relu')(x)
x = Dropout(0.3)(x)
x = BatchNormalization()(x)
outputs = Dense(1, activation='sigmoid')(x)

# Assemble the full model
model_vgg19 = Model(inputs, outputs, name='vgg19_backbone')

# Display model summary
model_vgg19.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_notop.h5
80134624/80134624 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

Model: "vgg19_backbone"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape            ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ input_layer_3 (InputLayer)       │ (None, 64, 64, 3)       │
```

```
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ vgg19 (Functional)              │ (None, 2, 2, 512)       │
20,024,384 │
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ flatten_3 (Flatten)             │ (None, 2048)            │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ batch_normalization_10          │ (None, 2048)            │
8,192 │
│ (BatchNormalization)            │                         │
│
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dense_7 (Dense)                 │ (None, 16)              │
32,784 │
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dropout_4 (Dropout)             │ (None, 16)              │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dense_8 (Dense)                 │ (None, 8)               │
136 │
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dropout_5 (Dropout)             │ (None, 8)               │
0 │
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ batch_normalization_11          │ (None, 8)               │
32 │
│ (BatchNormalization)            │                         │
│
├─────────────────────────────────┼─────────────────────────┼──────────
──────┤
│ dense_9 (Dense)                 │ (None, 1)               │
9 │
└─────────────────────────────────┴─────────────────────────┴──────────
──────┘

 Total params: 20,065,537 (76.54 MB)

 Trainable params: 37,041 (144.69 KB)

 Non-trainable params: 20,028,496 (76.40 MB)
```

ResNet50 Backbone Model

```python
# Set reproducibility seeds
np.random.seed(1)
tf.random.set_seed(1)

# Define input shape
input_shape = (im_size, im_size, 3)  # assuming im_size is a scalar
like 64

# Load ResNet50 base model without top layers
base_model = tf.keras.applications.ResNet50(
    input_shape=input_shape,
    include_top=False,
    weights='imagenet'
)

# Optionally freeze base model layers to use it as a fixed feature
extractor
base_model.trainable = False

# Build custom classification head
inputs = Input(shape=input_shape)
x = base_model(inputs, training=False)  # ensure batchnorm layers in
ResNet behave correctly
x = Flatten()(x)                        # Flatten feature maps
x = BatchNormalization()(x)            # Normalize activations
x = Dense(16, activation='relu')(x)    # First dense layer
x = Dropout(0.5)(x)                     # Regularization
x = Dense(8, activation='relu')(x)     # Second dense layer
x = Dropout(0.5)(x)                     # Additional regularization
x = BatchNormalization()(x)            # Normalize before final output
outputs = Dense(1, activation='sigmoid')(x)  # Output layer for binary
classification

# Assemble the full model
model_resnet50 = Model(inputs, outputs, name='resnet50_backbone')

# Display model summary
model_resnet50.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/resnet/
resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5
94765736/94765736 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step

Model: "resnet50_backbone"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━
━━━━━┓
┃ Layer (type)                     ┃ Output Shape           ┃
```

```
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━
━━━━━┩
│ input_layer_5 (InputLayer)      │ (None, 64, 64, 3)      │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ resnet50 (Functional)           │ (None, 2, 2, 2048)     │
23,587,712 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ flatten_4 (Flatten)             │ (None, 8192)           │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ batch_normalization_12          │ (None, 8192)           │
32,768 │
│ (BatchNormalization)            │                        │
│
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dense_10 (Dense)                │ (None, 16)             │
131,088 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dropout_6 (Dropout)             │ (None, 16)             │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dense_11 (Dense)                │ (None, 8)              │
136 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dropout_7 (Dropout)             │ (None, 8)              │
0 │
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ batch_normalization_13          │ (None, 8)              │
32 │
│ (BatchNormalization)            │                        │
│
├─────────────────────────────────┼────────────────────────┼──────────
─────┤
│ dense_12 (Dense)                │ (None, 1)              │
9 │
└─────────────────────────────────┴────────────────────────┴──────────
─────┘

 Total params: 23,751,745 (90.61 MB)
 Trainable params: 147,633 (576.69 KB)

 Non-trainable params: 23,604,112 (90.04 MB)
```

Preparing and Loading Images for Training

To streamline image loading during training, we'll utilize the `ImageDataGenerator` class in combination with the `flow_from_dataframe()` method. Furthermore, 25% of the training images will be set aside to assess validation performance.

```
# Convert labels to string type for compatibility
train_df['label'] = train_df['label'].astype(str)

# Initialize ImageDataGenerator with rescaling and validation split
generator = ImageDataGenerator(
    rescale=1.0 / 255,
    validation_split=0.25
)

# Configure training data generator
train_data = generator.flow_from_dataframe(
    dataframe          = train_df,
    x_col              = 'id',                # image filenames
    y_col              = 'label',             # class labels
    directory          = train_dir,
    subset             = 'training',
    class_mode         = 'binary',
    batch_size         = batch_size,
    target_size        = (im_size, im_size),  # resize images
    validate_filenames = False                # skip file existence
check
)

# Configure validation data generator
val_data = generator.flow_from_dataframe(
    dataframe          = train_df,
    x_col              = 'id',
    y_col              = 'label',
    directory          = train_dir,
    subset             = 'validation',
    class_mode         = 'binary',
    batch_size         = batch_size,
    target_size        = (im_size, im_size),
    validate_filenames = False
)

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

Results

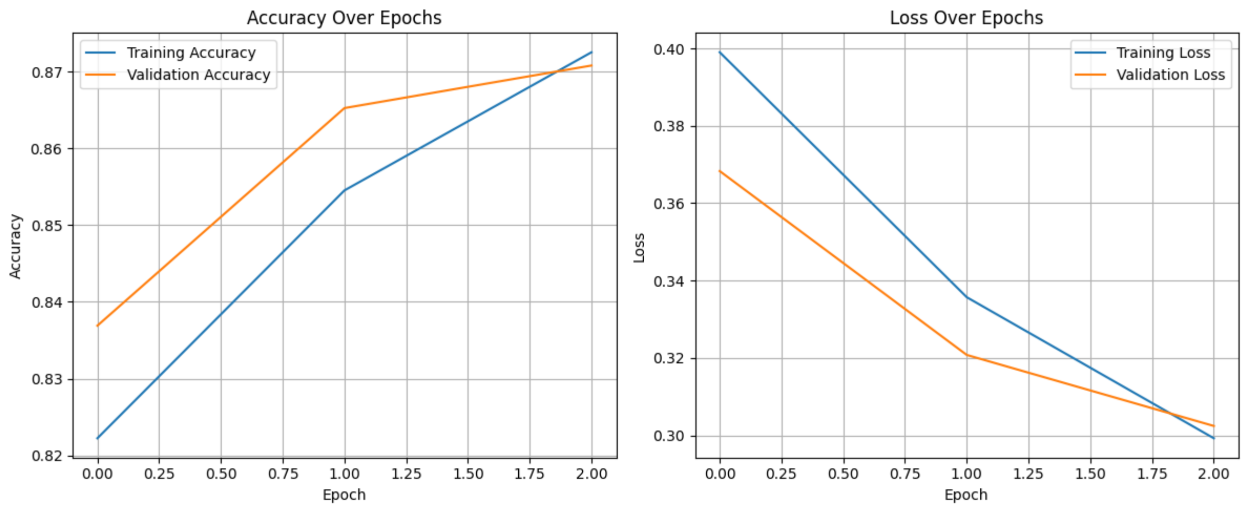
The model excluding BatchNormalization layers is initially trained using the Adam optimizer with a learning rate of 0.001, as higher rates tend to cause divergence. Binary crossentropy (BCE) is used as the loss function, and training is conducted over 3 epochs. Model performance will be evaluated on the held-out validation set using BCE loss and accuracy metrics.

CNNModel1 outcome evaluation (baseline model)

```
# Model configuration
optimizer = Adam(learning_rate=0.001)
loss_function = 'binary_crossentropy'
evaluation_metrics = ['accuracy']

# Compile the model
model1.compile(
    optimizer=optimizer,
    loss=loss_function,
    metrics=evaluation_metrics
)

# Train the model
hist = model1.fit(
    train_data,
    validation_data=val_data,
    epochs=3
)

Epoch 1/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 1667s 645ms/step - accuracy: 0.8124 -
loss: 0.4163 - val_accuracy: 0.8369 - val_loss: 0.3683
Epoch 2/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 1021s 396ms/step - accuracy: 0.8506 -
loss: 0.3433 - val_accuracy: 0.8653 - val_loss: 0.3208
Epoch 3/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 664s 257ms/step - accuracy: 0.8715 -
loss: 0.3026 - val_accuracy: 0.8708 - val_loss: 0.3024
```

The validation loss reached a minimum of 0.3024, while validation accuracy peaked at 87%. The following figure illustrates a consistent decline in both training and validation loss across epochs, accompanied by a steady rise in accuracy for both sets—indicating progressive improvement in the model's performance during training.

```
def plot_training_history(history):
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Accuracy plot
    axes[0].plot(history.history['accuracy'], label='Training
Accuracy')
    axes[0].plot(history.history['val_accuracy'], label='Validation
Accuracy')
    axes[0].set_title('Accuracy Over Epochs')
    axes[0].set_xlabel('Epoch')
    axes[0].set_ylabel('Accuracy')
    axes[0].legend()
    axes[0].grid(True)

    # Loss plot
    axes[1].plot(history.history['loss'], label='Training Loss')
    axes[1].plot(history.history['val_loss'], label='Validation Loss')
    axes[1].set_title('Loss Over Epochs')
    axes[1].set_xlabel('Epoch')
    axes[1].set_ylabel('Loss')
    axes[1].legend()
    axes[1].grid(True)

    plt.tight_layout()
    plt.show()

plot_training_history(hist)
```

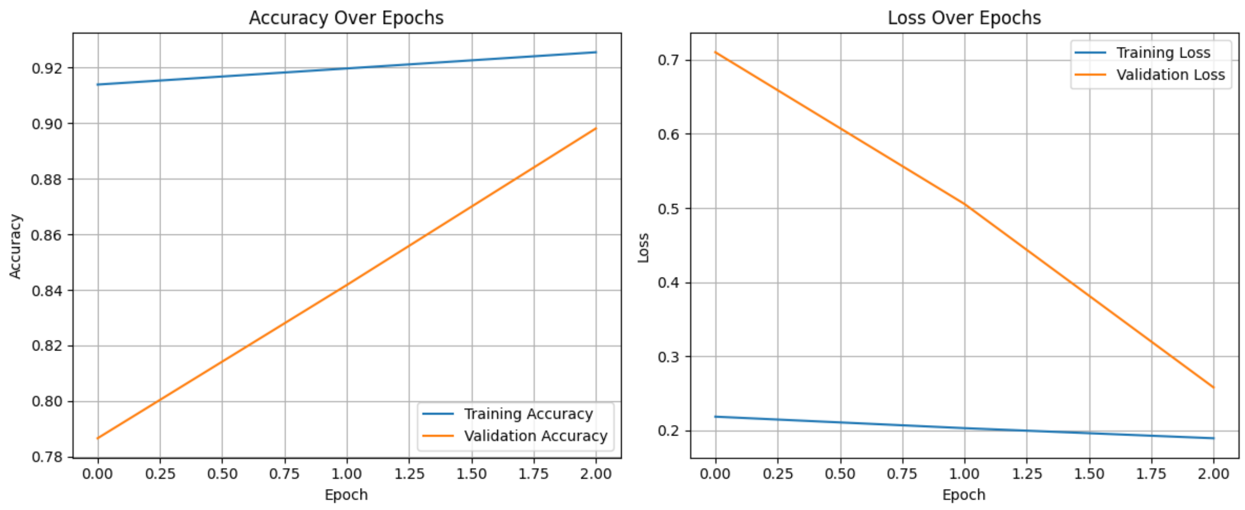


CNNModel2 outcome evaluation (baseline model)

Next, we evaluated the model with batch normalization enabled and the other hyperparameters tuned as previously described. The optimizer and number of training epochs remained unchanged from the earlier configuration

```
# Model configuration
optimizer = Adam(learning_rate=0.001)
loss_function = 'binary_crossentropy'
evaluation_metrics = ['accuracy']

# Compile the model
```

```
model2.compile(
    optimizer=optimizer,
    loss=loss_function,
    metrics=evaluation_metrics
)

# Train the model
hist = model2.fit(
    train_data,
    validation_data=val_data,
    epochs=3
)

plot_training_history(hist)

Epoch 1/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 1528s 590ms/step - accuracy: 0.9119 -
loss: 0.2215 - val_accuracy: 0.7866 - val_loss: 0.7102
Epoch 2/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 1504s 583ms/step - accuracy: 0.9205 -
loss: 0.2002 - val_accuracy: 0.8418 - val_loss: 0.5054
Epoch 3/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 1513s 586ms/step - accuracy: 0.9260 -
loss: 0.1884 - val_accuracy: 0.8982 - val_loss: 0.2578
```



Validation loss reached 0.2578, with validation accuracy rising to 89.2%. Training curves show a steady decline in loss and a corresponding increase in accuracy, reflecting effective learning.

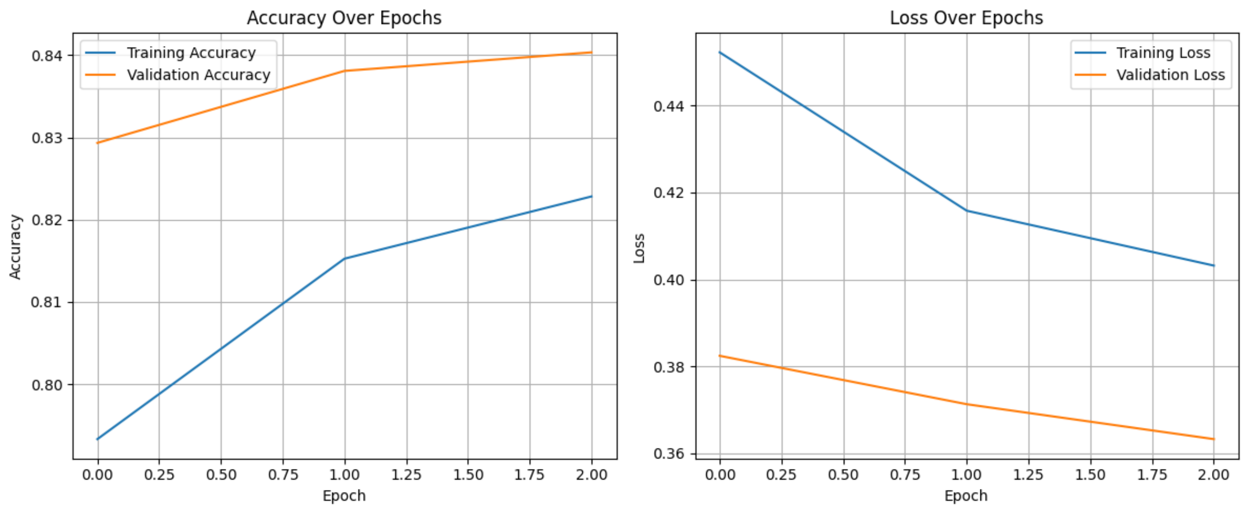
Validation metrics, while generally improving, showed mild fluctuations. This suggests sensitivity to training dynamics and possible early signs of overfitting. Overall, the model demonstrates strong generalization and solid alignment between training and validation performance.

VGG16 backbone outcome evaluation

Our approach involves evaluating two strategies with the VGG16 model: one using transfer learning with frozen pretrained weights from ImageNet, and the other training the model entirely from scratch. We will assess and compare their performance using accuracy and ROC AUC (Area Under the Curve) as evaluation metrics.

Transfer Learning Technique

```
# Define optimizer
optimizer = tf.keras.optimizers.Adam(0.001)

# Freeze the base model
base_model.trainable = False

# Compile the model
model_vgg16.compile(
    loss='binary_crossentropy',
    optimizer=optimizer,
    metrics=[
        'accuracy',
        tf.keras.metrics.AUC(name='roc_auc')
    ]
)

# Train the model
hist = model_vgg16.fit(
    train_data,
    epochs=3,
    validation_data=val_data,
    verbose=1
)

plot_training_history(hist)

Epoch 1/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 4405s 2s/step - accuracy: 0.7700 -
loss: 0.4886 - roc_auc: 0.8370 - val_accuracy: 0.8293 - val_loss:
0.3824 - val_roc_auc: 0.9039
Epoch 2/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 4428s 2s/step - accuracy: 0.8135 -
loss: 0.4173 - roc_auc: 0.8862 - val_accuracy: 0.8381 - val_loss:
0.3713 - val_roc_auc: 0.9100
Epoch 3/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 4417s 2s/step - accuracy: 0.8230 -
loss: 0.4031 - roc_auc: 0.8948 - val_accuracy: 0.8403 - val_loss:
0.3633 - val_roc_auc: 0.9134
```

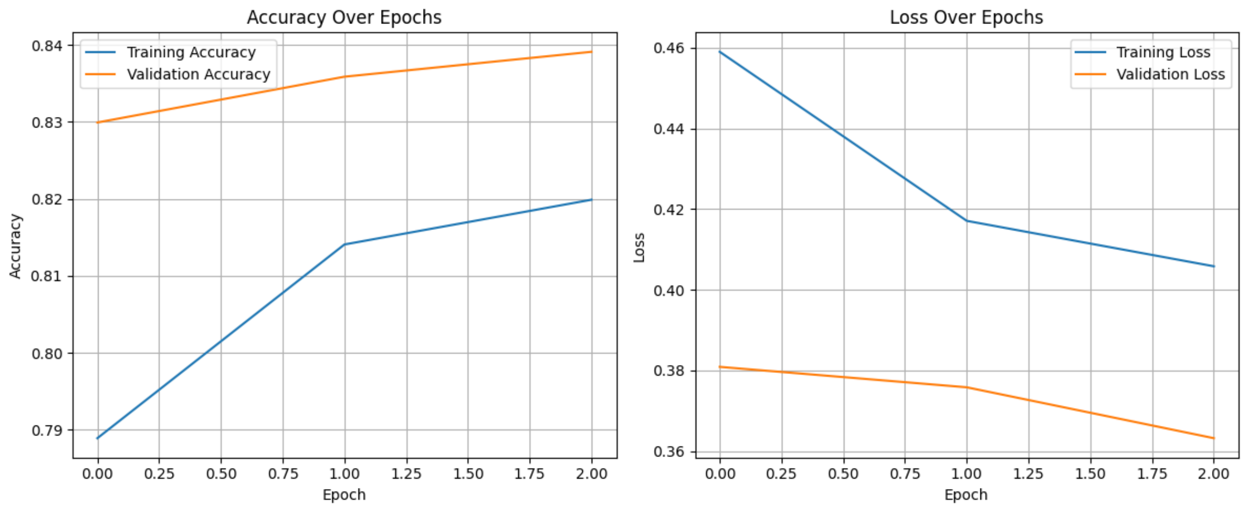


Entirely from Scratch Technique

```
# Define optimizer
optimizer = tf.keras.optimizers.Adam(0.001)

# Unfreeze base model for fine-tuning
base_model.trainable = True

# Recompile the model with the same optimizer and metrics
model_vgg16.compile(
    optimizer=optimizer,
    loss='binary_crossentropy',
    metrics=[
        'accuracy',
        tf.keras.metrics.AUC(name='roc_auc')
    ]
)

# Fine-tune the model for a few epochs
hist = model_vgg16.fit(
    train_data,
    validation_data=val_data,
    epochs=3,
    verbose=1
)

# Visualize training performance
plot_training_history(hist)
```

```
Epoch 1/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 5330s 2s/step - accuracy: 0.7588 -
loss: 0.4997 - roc_auc: 0.8302 - val_accuracy: 0.8299 - val_loss:
0.3809 - val_roc_auc: 0.9039
Epoch 2/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 5220s 2s/step - accuracy: 0.8119 -
```

```
loss: 0.4210 - roc_auc: 0.8845 - val_accuracy: 0.8359 - val_loss:
0.3758 - val_roc_auc: 0.9097
Epoch 3/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 5152s 2s/step - accuracy: 0.8196 -
loss: 0.4059 - roc_auc: 0.8935 - val_accuracy: 0.8391 - val_loss:
0.3632 - val_roc_auc: 0.9129
```

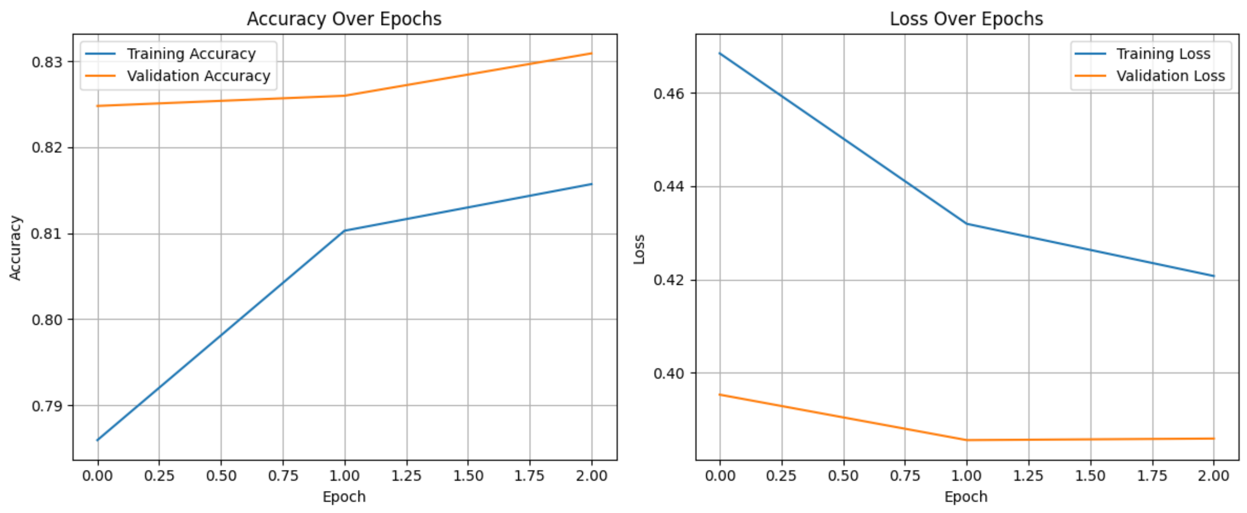


VGG19 backbone outcome evaluation

Entirely from Scratch Technique

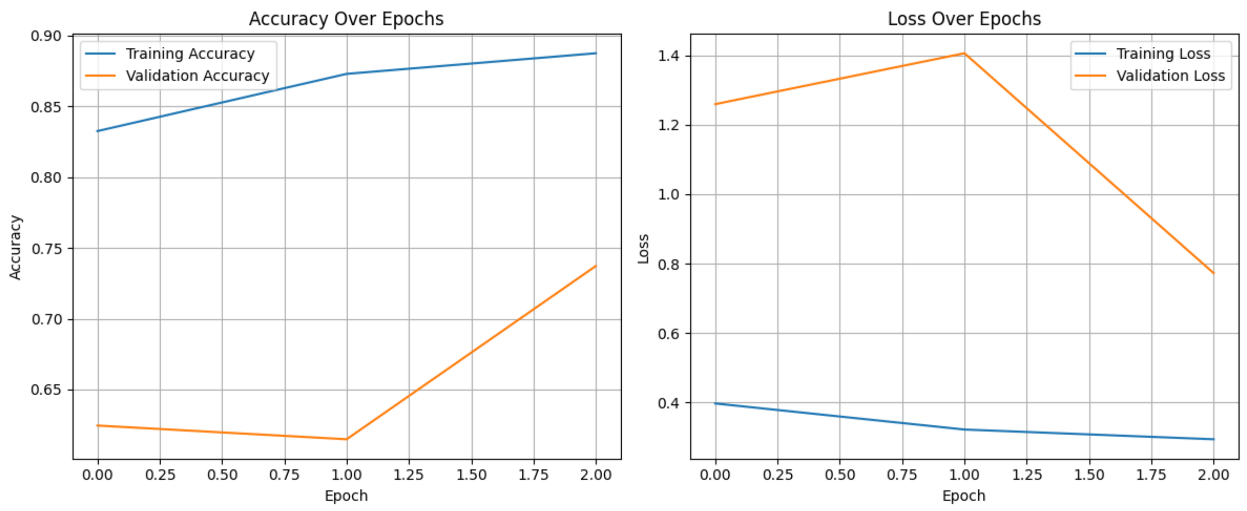
```python
# Optimizer configuration
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)

# Unfreeze base model for fine-tuning
base_model.trainable = True

# Compile the VGG19 model
model_vgg19.compile(
    optimizer=optimizer,
    loss='binary_crossentropy',
    metrics=[
        'accuracy',
        tf.keras.metrics.AUC(name='roc_auc')
    ]
)

# Train the model
hist = model_vgg19.fit(
    train_data,
    validation_data=val_data,
    epochs=3,
    verbose=1
)
```

```
# Save the trained model
model_vgg19.save('model_vgg19_20.h5')

# Plot training performance
plot_training_history(hist)

Epoch 1/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 6337s 2s/step - accuracy: 0.7538 -
loss: 0.5091 - roc_auc: 0.8246 - val_accuracy: 0.8248 - val_loss:
0.3953 - val_roc_auc: 0.8980
Epoch 2/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 6307s 2s/step - accuracy: 0.8088 -
loss: 0.4333 - roc_auc: 0.8787 - val_accuracy: 0.8260 - val_loss:
0.3856 - val_roc_auc: 0.9028
Epoch 3/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 6384s 2s/step - accuracy: 0.8149 -
loss: 0.4217 - roc_auc: 0.8852 - val_accuracy: 0.8309 - val_loss:
0.3859 - val_roc_auc: 0.9039
```



ResNet50 backbone outcome evaluation

```
# Unfreeze base model for fine-tuning
base_model.trainable = True

# Optimizer configuration
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)

# Compile the ResNet50 model
model_resnet50.compile(
    optimizer=optimizer,
    loss='binary_crossentropy',
    metrics=[
        'accuracy',
        tf.keras.metrics.AUC(name='roc_auc')
    ]
)

# Train the model
hist = model_resnet50.fit(
    train_data,
    validation_data=val_data,
    epochs=3,
    verbose=1
)

# Plot training performance
plot_training_history(hist)

Epoch 1/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 11329s 4s/step - accuracy: 0.7971 -
loss: 0.4550 - roc_auc: 0.8656 - val_accuracy: 0.6245 - val_loss:
1.2599 - val_roc_auc: 0.8092
Epoch 2/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 11209s 4s/step - accuracy: 0.8685 -
loss: 0.3288 - roc_auc: 0.9355 - val_accuracy: 0.6149 - val_loss:
1.4067 - val_roc_auc: 0.8212
Epoch 3/3
2579/2579 ━━━━━━━━━━━━━━━━━━━━ 11098s 4s/step - accuracy: 0.8859 -
loss: 0.2973 - roc_auc: 0.9475 - val_accuracy: 0.7372 - val_loss:
0.7726 - val_roc_auc: 0.8902
```



As shown in the figure, the ResNet50 model demonstrates steady improvement in training accuracy and a consistent decline in training loss over the epochs. Validation accuracy also increases, while validation loss initially rises before dropping sharply, suggesting that the model is beginning to generalize better. Although some early fluctuations are present, the overall trajectory indicates that the model is learning effectively and may benefit from additional training to further stabilize validation performance.

Evaluation: Predictions on Test Data

As before, a DataFrame containing the test image IDs was created, and the images were automatically loaded during the prediction phase using the `flow_from_dataframe()` function. The resulting predictions were submitted to Kaggle for scoring, although the leaderboard was not enabled for ranking.

```python
import os

# Prepare test image filenames
image_filenames = os.listdir(test_dir)
images_test_df = pd.DataFrame({'id': image_filenames})

# Initialize ImageDataGenerator for test set
test_datagen = ImageDataGenerator(rescale=1.0 / 255)

# Create test data generator
test_generator = test_datagen.flow_from_dataframe(
    dataframe=images_test_df,
    directory=test_dir,
    x_col='id',
    y_col=None,
    target_size=(im_size, im_size),
    batch_size=1,
    class_mode=None,
    shuffle=False
)

# Generate predictions
predictions = model1.predict(test_generator, verbose=1)
```

```
Found 57458 validated image filenames.
57458/57458 ━━━━━━━━━━━━━━━━━━━━ 294s 5ms/step
```

```python
# Remove single dimensions from predictions
predictions = np.squeeze(predictions)

# Display the new shape
print(f"Predictions shape: {predictions.shape}")
```

```
Predictions shape: (57458,)
```

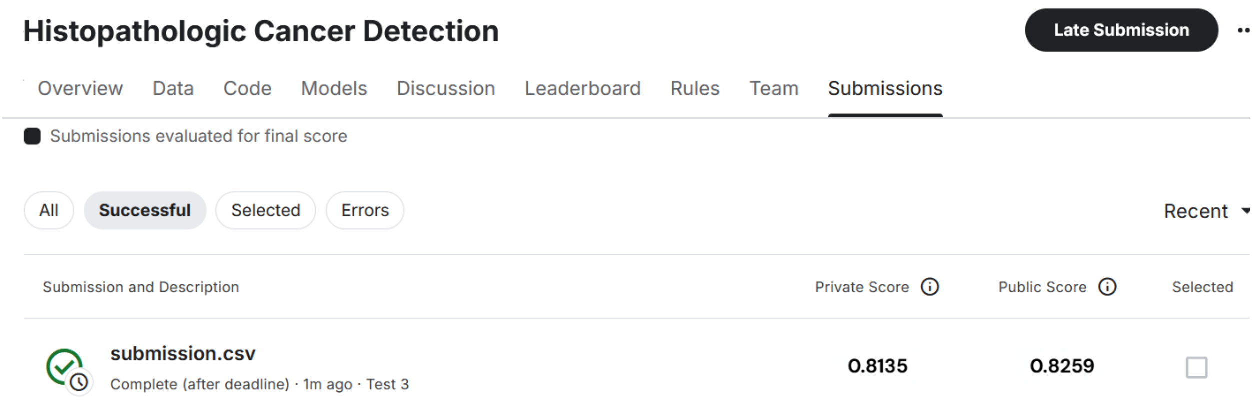
```python
# Create submission DataFrame for Kaggle
submission_df = pd.DataFrame({
    'id': images_test_df['id'].str.replace('.tif', '', regex=False),
    'label': (predictions.flatten() >= 0.5).astype(int)
})

# Show label distribution
print(submission_df['label'].value_counts())
```

```
# Save to CSV
submission_df.to_csv('submission.csv', index=False)

# Preview
print(submission_df.head())

label
0    55709
1     1749
Name: count, dtype: int64
                                         id  label
0  a7ea26360815d8492433b14cd8318607bcf99d9e      0
1  59d21133c845dff1ebc7a0c7cf40c145ea9e9664      0
2  5fde41ce8c6048a5c2f38eca12d6528fa312cdbb      0
3  bd953a3b1db1f7041ee95ff482594c4f46c73ed0      0
4  523fc2efd7aba53e597ab0f69cc2cbded7a6ce62      0
```

Project GitHub Repository

https://github.com/k-khuu/CNN-Cancer-Detection-Kaggle-Mini-Project

Kaggle Position



Histopathologic Cancer Detection

■ Submissions evaluated for final score

Submission and Description	Private Score	Public Score	Selected
submission.csv Complete (after deadline) · 1m ago · Test 3	0.8135	0.8259	☐

Conclusion

The CNN model without BatchNormalization (CNNModel1) with a 82.59% score outperformed the BatchNormalized version (CNNModel2) when trained for just 3 epochs. CNNModel2 may generalize better if trained longer, such as for 20 epochs. Additionally, we trained popular architectures like VGG1 and VGG19 using both transfer learning with ImageNet weights and training from scratch. Models trained from scratch showed notable accuracy gains. For further improvement, exploring advanced architectures like ResNet50/101, InceptionV3, or EfficientNet is recommended.