Dataset Information:

The data set is comprised of around 27,500 PNG images of blood smear slides (split 50% healthy, 50% malaria infected). The images were taken at x100 zoom in RBG color space and were annotated by an expert slide reader. The data set can be found at https://data.lhncbc.nlm.nih.gov/public/Malaria/Thick_Smears_150/index.html

Project Idea: Train a Convolutional Neural Network to classify images of blood smear slides as either 'Malaria' or 'Uninfected'

Problem Statement: Diagnosing Malaria using traditional methods (like manual examination of microscopic blood smear images) can be time-consuming. Furthermore, late detection of malaria leads to higher mortality rates. For treatment to be most effective, Malaria must be identified as early as possible. It is possible that doctors could more accurately diagnose Malaria from blood smear images if they were assisted by a neural network. If a deep learning model were able to achieve better accuracy than a doctor, then this would be very useful in the medical field. Neural Networks could be used to verify the diagnosis of a doctor or assist with a diagnosis if the doctor was unsure. If the accuracy was reliable enough, it could be used as a replacement to traditional methods. Malaria is most deadly to children in Africa, where resources are severely limited and infection rates are high.

Potential Use Case: Trained Convolutional Neural Network is deployed (in areas with high Malaria mortality rates and/or limited resources) by a non-profit organization. These areas may not have enough Doctors available to keep up with the task of manually evaluating blood smears to determine if they are positive or negative for Malaria. In cases where the collection of blood smear images outpaces the capacity of manual microscopic detection, Neural Networks can be used to for malaria detection. Positive diagnoses could be passed to Doctor's for manual verification depending on the recall of the model. For instance, if recall is 100%, there will likely be false positives and Doctor's will need to verify positive results. However, Doctor's would not need to review negatives diagnoses, as a model with 100% recall would have no false negatives.

Why do I think deep learning could be useful and related?: Convolutional Neural Networks have proven to be particularly useful in image classification tasks. They have an ability to identify complex features from high-dimensional image data and achieve very high accuracies in many cases. This makes them well-suited for image classification tasks, where other methods may fall short.

First we will unzip the zipped data and define the directory and subdirectory paths. The function 'create_subset' randomly selects a subset from each subdirectory of size 'sample_size'. These images are copies into the subdirectories of the new directory 'cell_images_subset'. In this case, 1000 malaria-infected images and 1000 uninfected images are copied into the new subdirectory which will be used for testing.

In [1]:
```
import zipfile
import os
import random
import shutil
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten, Dropout
from tensorflow.keras.layers import BatchNormalization
```

```
from tensorflow.keras.regularizers import l2, l1
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping, ModelCheckpoint
from tensorflow.keras.models import load_model
from sklearn.metrics import confusion_matrix, classification_report

# set seeds
np.random.seed(42)
tf.random.set_seed(42)
random.seed(42)

# Path to the zipped folder
zip_path = 'cell_images.zip'
extract_path = 'cell_images'

# only unzip if it has not already been unzipped
if not os.path.exists(extract_path):
    # unzip the folder
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(extract_path)

# Define original and subset directories
original_dir = 'cell_images/cell_images'
subset_dir = 'cell_images_subset'

# Creates subsets to build the model, to compensate for lack of a GPU
def create_subset(original_dir, subset_dir, sample_size=1000):
    classes = ['Parasitized', 'Uninfected']
    # make sure the subset directory exists
    if not os.path.exists(subset_dir):
        os.makedirs(subset_dir, exist_ok=True)
        for cls in classes:
            os.makedirs(os.path.join(subset_dir, cls), exist_ok=True)
            files = os.listdir(os.path.join(original_dir, cls))
            sample_files = random.sample(files, sample_size)
            for file in sample_files:
                shutil.copy(os.path.join(original_dir, cls, file), os.path.join(subset_d
    else:
        print(f"The directory {subset_dir} already exists.")


# Create a subset of 1000 images per class
create_subset(original_dir, subset_dir, sample_size=1000)
```

```
The directory cell_images_subset already exists.
```

In [2]:

```
# create a function for data generator creation so target_size, directory, and batch_siz
def create_data_generators(base_dir, target_size, batch_size=32, augmentation=False):

    if augmentation: # create datagenerators with augmentation
        datagen = ImageDataGenerator(
            rescale=1.0/255.0,
            validation_split=0.2,
            rotation_range=30,
            width_shift_range=0.2,
            height_shift_range=0.2,
            shear_range=0.2,
            zoom_range=0.2,
            horizontal_flip=True,
            fill_mode='nearest'
        )

    else: # create data generators without augmentation
        datagen = ImageDataGenerator(rescale=1.0/255.0, validation_split=0.2)

    # Training data
    train_generator = datagen.flow_from_directory(
```

```
        base_dir,
        target_size=target_size, # taken from function args
        batch_size=batch_size, # taken from function args
        class_mode='binary',
        subset='training'
    )

    # Validation data
    validation_generator = datagen.flow_from_directory(
        base_dir,
        target_size=target_size, # taken from function args
        batch_size=batch_size, # taken from function args
        class_mode='binary',
        subset='validation'
    )

    return train_generator, validation_generator
```

In [3]:
```
# define a function to plot accuracy and loss
def plot_acc_loss(history):

    # define loss
    train_loss = history.history['loss']
    val_loss = history.history['val_loss']

    # define accuracy
    train_acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']

    # define epochs (x-axis)
    epochs = range(1, len(train_acc) + 1)

    # create 2 subplots
    plt.figure(figsize=(12,10))
    plt.subplot(2,1,1)

    # plot accuracy
    plt.plot(epochs, train_acc, color='red', label='Training Accuracy')
    plt.plot(epochs, val_acc, color='blue', label='Validation Accuracy')
    plt.title('Training/Validation Accuracy vs. Epochs', fontweight='bold')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.xticks(np.arange(1,len(train_acc) + 1))
    plt.legend()

    # plot loss
    plt.subplot(2,1,2)
    plt.plot(epochs, train_loss, color='red', label='Training Loss')
    plt.plot(epochs, val_loss, color='blue', label='Validation Loss')
    plt.title('Training/Validation Loss vs. Epochs', fontweight='bold')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.xticks(np.arange(1,len(val_acc) + 1))
    plt.legend()

    plt.tight_layout()
    plt.show()
```

In [4]:
```
# create generators with 64x64 targets initially
train_generator, validation_generator = create_data_generators(
    base_dir='cell_images_subset',
    target_size=(64,64),
    batch_size=32)
```

```
Found 1600 images belonging to 2 classes.
```

```
Found 400 images belonging to 2 classes.
```

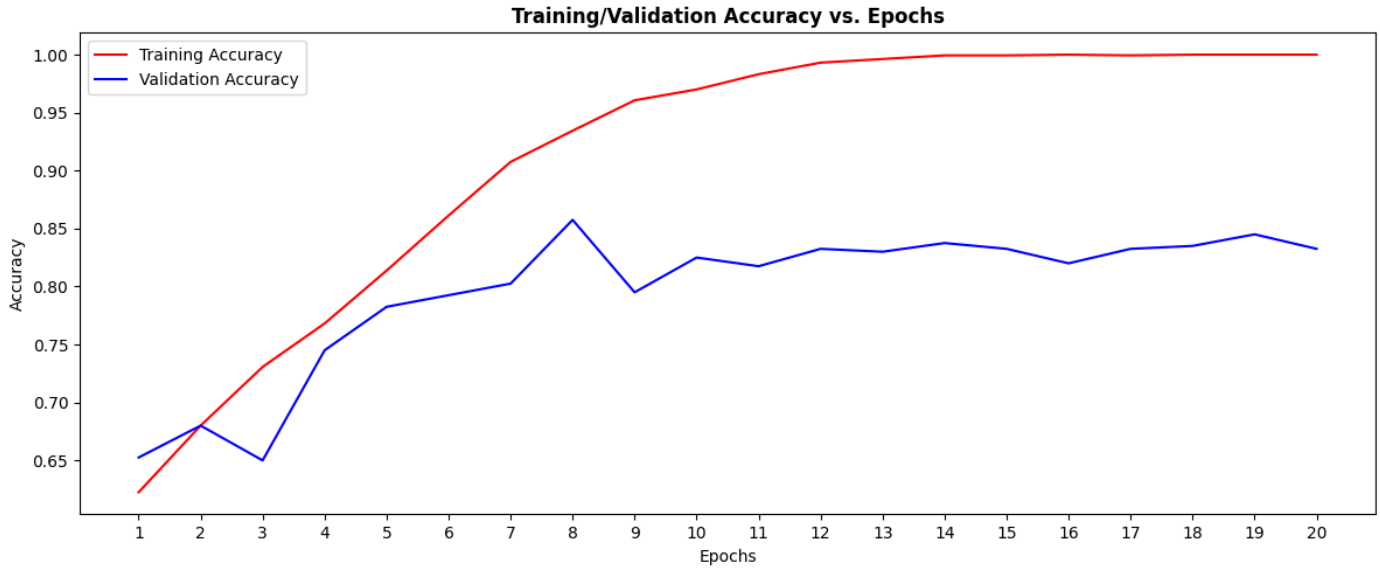
In [5]:

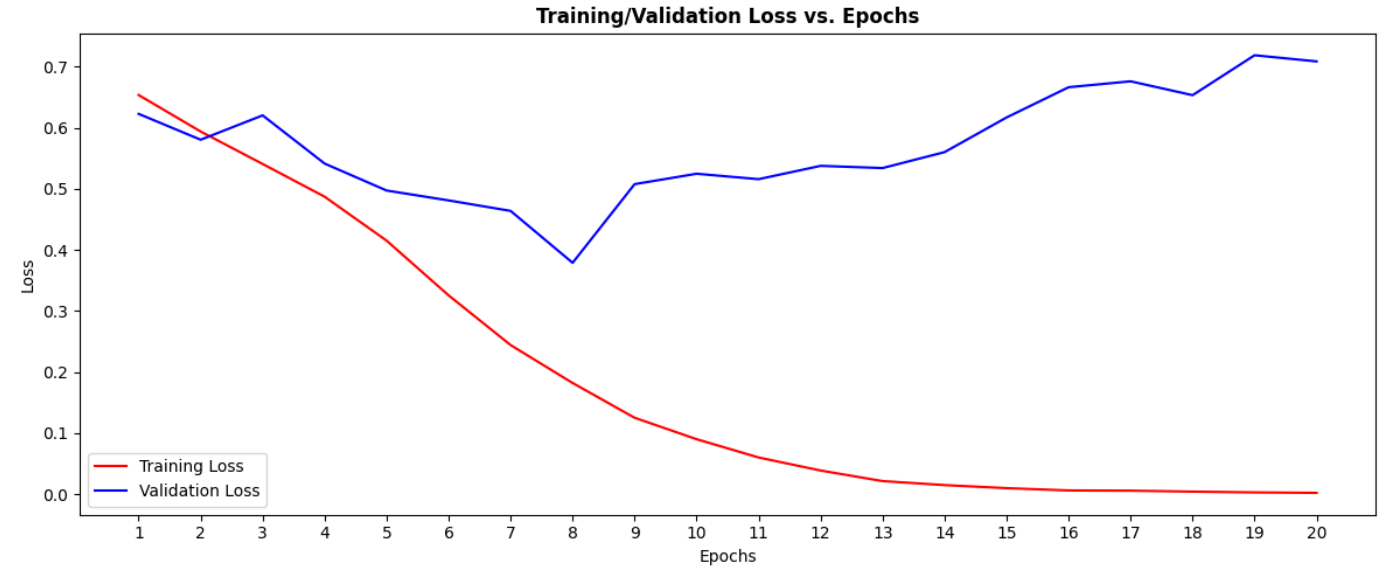
```python
# try building an initial Convolutional Neural Network using a subset of the data
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compiling the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Training the model
history_64_by_64 = model.fit(train_generator,
                             validation_data=validation_generator,
                             epochs=20,
                             verbose=2)
```

```
Epoch 1/20
50/50 - 5s - loss: 0.6535 - accuracy: 0.6225 - val_loss: 0.6227 - val_accuracy: 0.6525 -
5s/epoch - 107ms/step
Epoch 2/20
50/50 - 3s - loss: 0.5938 - accuracy: 0.6800 - val_loss: 0.5803 - val_accuracy: 0.6800 -
3s/epoch - 58ms/step
Epoch 3/20
50/50 - 3s - loss: 0.5406 - accuracy: 0.7306 - val_loss: 0.6202 - val_accuracy: 0.6500 -
3s/epoch - 61ms/step
Epoch 4/20
50/50 - 3s - loss: 0.4870 - accuracy: 0.7681 - val_loss: 0.5413 - val_accuracy: 0.7450 -
3s/epoch - 68ms/step
Epoch 5/20
50/50 - 3s - loss: 0.4150 - accuracy: 0.8138 - val_loss: 0.4971 - val_accuracy: 0.7825 -
3s/epoch - 56ms/step
Epoch 6/20
50/50 - 4s - loss: 0.3254 - accuracy: 0.8612 - val_loss: 0.4809 - val_accuracy: 0.7925 -
4s/epoch - 74ms/step
Epoch 7/20
50/50 - 3s - loss: 0.2440 - accuracy: 0.9075 - val_loss: 0.4637 - val_accuracy: 0.8025 -
3s/epoch - 64ms/step
Epoch 8/20
50/50 - 3s - loss: 0.1819 - accuracy: 0.9344 - val_loss: 0.3787 - val_accuracy: 0.8575 -
3s/epoch - 63ms/step
Epoch 9/20
50/50 - 4s - loss: 0.1249 - accuracy: 0.9606 - val_loss: 0.5076 - val_accuracy: 0.7950 -
4s/epoch - 77ms/step
Epoch 10/20
50/50 - 4s - loss: 0.0898 - accuracy: 0.9700 - val_loss: 0.5246 - val_accuracy: 0.8250 -
4s/epoch - 74ms/step
Epoch 11/20
50/50 - 3s - loss: 0.0599 - accuracy: 0.9831 - val_loss: 0.5157 - val_accuracy: 0.8175 -
3s/epoch - 57ms/step
Epoch 12/20
50/50 - 3s - loss: 0.0385 - accuracy: 0.9931 - val_loss: 0.5375 - val_accuracy: 0.8325 -
3s/epoch - 69ms/step
Epoch 13/20
50/50 - 4s - loss: 0.0212 - accuracy: 0.9962 - val_loss: 0.5339 - val_accuracy: 0.8300 -
4s/epoch - 78ms/step
Epoch 14/20
50/50 - 4s - loss: 0.0147 - accuracy: 0.9994 - val_loss: 0.5600 - val_accuracy: 0.8375 -
4s/epoch - 85ms/step
Epoch 15/20
50/50 - 3s - loss: 0.0097 - accuracy: 0.9994 - val_loss: 0.6168 - val_accuracy: 0.8325 -
```

```
3s/epoch - 67ms/step
Epoch 16/20
50/50 - 3s - loss: 0.0059 - accuracy: 1.0000 - val_loss: 0.6663 - val_accuracy: 0.8200 -
3s/epoch - 60ms/step
Epoch 17/20
50/50 - 4s - loss: 0.0055 - accuracy: 0.9994 - val_loss: 0.6760 - val_accuracy: 0.8325 -
4s/epoch - 73ms/step
Epoch 18/20
50/50 - 4s - loss: 0.0040 - accuracy: 1.0000 - val_loss: 0.6533 - val_accuracy: 0.8350 -
4s/epoch - 75ms/step
Epoch 19/20
50/50 - 4s - loss: 0.0027 - accuracy: 1.0000 - val_loss: 0.7187 - val_accuracy: 0.8450 -
4s/epoch - 73ms/step
Epoch 20/20
50/50 - 5s - loss: 0.0020 - accuracy: 1.0000 - val_loss: 0.7086 - val_accuracy: 0.8325 -
5s/epoch - 95ms/step
```

In [6]:
```
# display plot
plot_acc_loss(history_64_by_64)
```





Judging from the validation loss curve, it appears the model begins to overfit ~ epoch 8. The validation accuracy reached a peak of .8450 in this model.

Next we will try adding a dropout between each layer with a 25% dropout rate in an attempt to reduce overfitting.

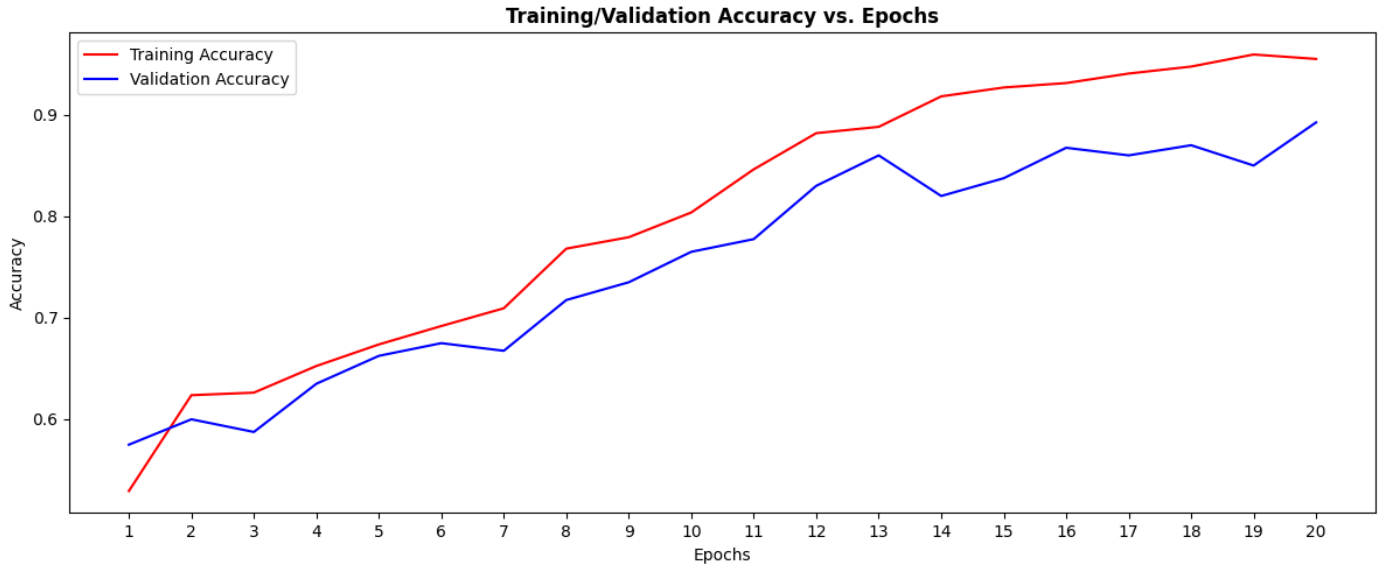
In [7]:
```
# build CNN with dropout
```

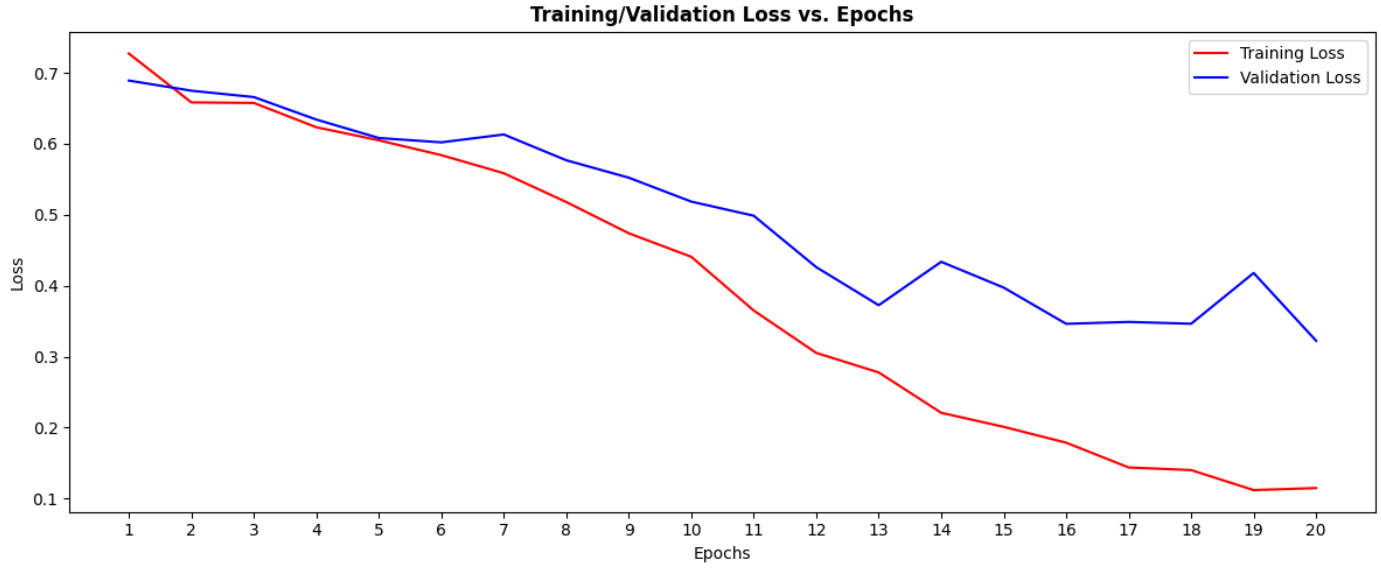
```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),
    MaxPooling2D((2, 2)),
    Dropout(0.25), # add 25% dropout layer
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Dropout(0.25),  # add 25% dropout layer
    Flatten(),
    Dense(64, activation='relu'),
    Dropout(0.25),  # add 25% dropout layer
    Dense(1, activation='sigmoid')
])

# Compiling the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Training the model
history_64_dropout = model.fit(train_generator,
                               validation_data=validation_generator,
                               epochs=20,
                               verbose=2)
```

```
Epoch 1/20
50/50 - 4s - loss: 0.7270 - accuracy: 0.5294 - val_loss: 0.6890 - val_accuracy: 0.5750 -
4s/epoch - 83ms/step
Epoch 2/20
50/50 - 4s - loss: 0.6582 - accuracy: 0.6237 - val_loss: 0.6747 - val_accuracy: 0.6000 -
4s/epoch - 81ms/step
Epoch 3/20
50/50 - 3s - loss: 0.6575 - accuracy: 0.6263 - val_loss: 0.6658 - val_accuracy: 0.5875 -
3s/epoch - 68ms/step
Epoch 4/20
50/50 - 4s - loss: 0.6233 - accuracy: 0.6525 - val_loss: 0.6341 - val_accuracy: 0.6350 -
4s/epoch - 81ms/step
Epoch 5/20
50/50 - 4s - loss: 0.6048 - accuracy: 0.6737 - val_loss: 0.6081 - val_accuracy: 0.6625 -
4s/epoch - 78ms/step
Epoch 6/20
50/50 - 4s - loss: 0.5839 - accuracy: 0.6919 - val_loss: 0.6020 - val_accuracy: 0.6750 -
4s/epoch - 82ms/step
Epoch 7/20
50/50 - 4s - loss: 0.5585 - accuracy: 0.7094 - val_loss: 0.6130 - val_accuracy: 0.6675 -
4s/epoch - 86ms/step
Epoch 8/20
50/50 - 4s - loss: 0.5178 - accuracy: 0.7681 - val_loss: 0.5767 - val_accuracy: 0.7175 -
4s/epoch - 86ms/step
Epoch 9/20
50/50 - 4s - loss: 0.4738 - accuracy: 0.7794 - val_loss: 0.5521 - val_accuracy: 0.7350 -
4s/epoch - 84ms/step
Epoch 10/20
50/50 - 4s - loss: 0.4407 - accuracy: 0.8037 - val_loss: 0.5185 - val_accuracy: 0.7650 -
4s/epoch - 85ms/step
Epoch 11/20
50/50 - 4s - loss: 0.3650 - accuracy: 0.8462 - val_loss: 0.4985 - val_accuracy: 0.7775 -
4s/epoch - 80ms/step
Epoch 12/20
50/50 - 4s - loss: 0.3053 - accuracy: 0.8819 - val_loss: 0.4261 - val_accuracy: 0.8300 -
4s/epoch - 79ms/step
Epoch 13/20
50/50 - 4s - loss: 0.2777 - accuracy: 0.8881 - val_loss: 0.3724 - val_accuracy: 0.8600 -
4s/epoch - 78ms/step
Epoch 14/20
50/50 - 4s - loss: 0.2208 - accuracy: 0.9181 - val_loss: 0.4336 - val_accuracy: 0.8200 -
4s/epoch - 83ms/step
Epoch 15/20
50/50 - 4s - loss: 0.2010 - accuracy: 0.9269 - val_loss: 0.3973 - val_accuracy: 0.8375 -
```

```
4s/epoch - 82ms/step
Epoch 16/20
50/50 - 4s - loss: 0.1787 - accuracy: 0.9312 - val_loss: 0.3462 - val_accuracy: 0.8675 -
4s/epoch - 81ms/step
Epoch 17/20
50/50 - 4s - loss: 0.1438 - accuracy: 0.9406 - val_loss: 0.3489 - val_accuracy: 0.8600 -
4s/epoch - 80ms/step
Epoch 18/20
50/50 - 4s - loss: 0.1401 - accuracy: 0.9475 - val_loss: 0.3463 - val_accuracy: 0.8700 -
4s/epoch - 78ms/step
Epoch 19/20
50/50 - 4s - loss: 0.1119 - accuracy: 0.9594 - val_loss: 0.4178 - val_accuracy: 0.8500 -
4s/epoch - 78ms/step
Epoch 20/20
50/50 - 4s - loss: 0.1148 - accuracy: 0.9550 - val_loss: 0.3223 - val_accuracy: 0.8925 -
4s/epoch - 81ms/step
```

In [8]:
```
# display plot
plot_acc_loss(history_64_dropout)
```





Adding the dropout layers appears to have reduced some of the overfitting and improved the accuracy by ~ 5%.

Let's try adding a l2 regularizer to the previous model with l2 = .001 and see if we observe any improvement.

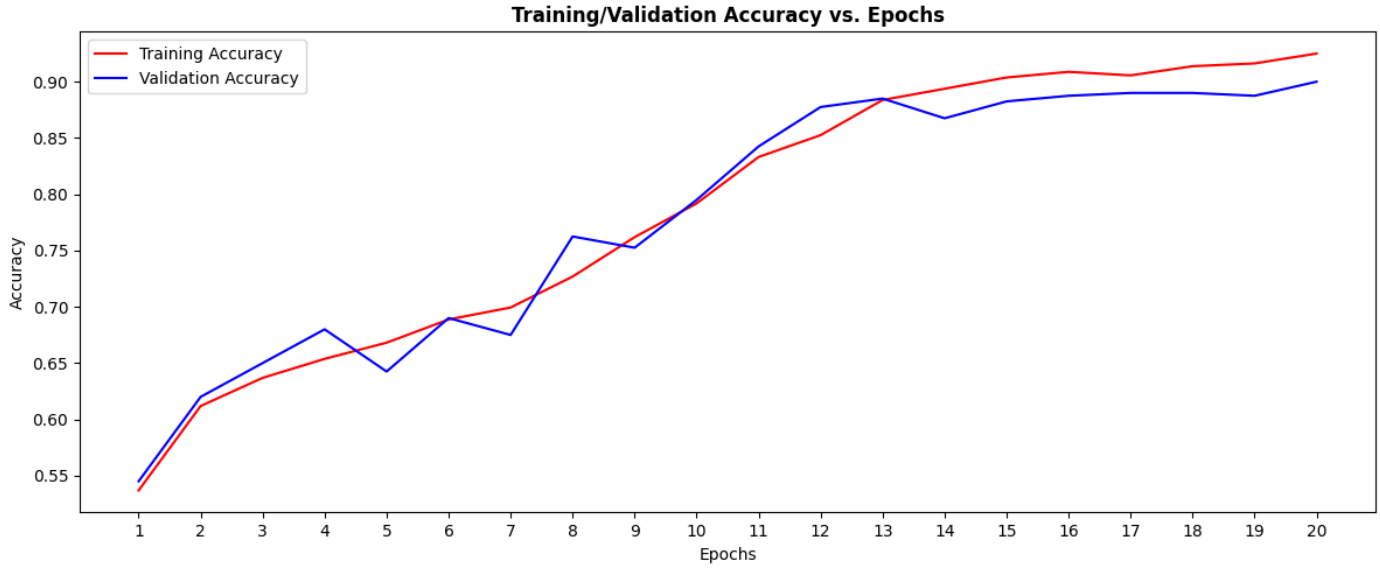
In [9]:
```
# Define the model with regularization and dropout, l2=.001
```

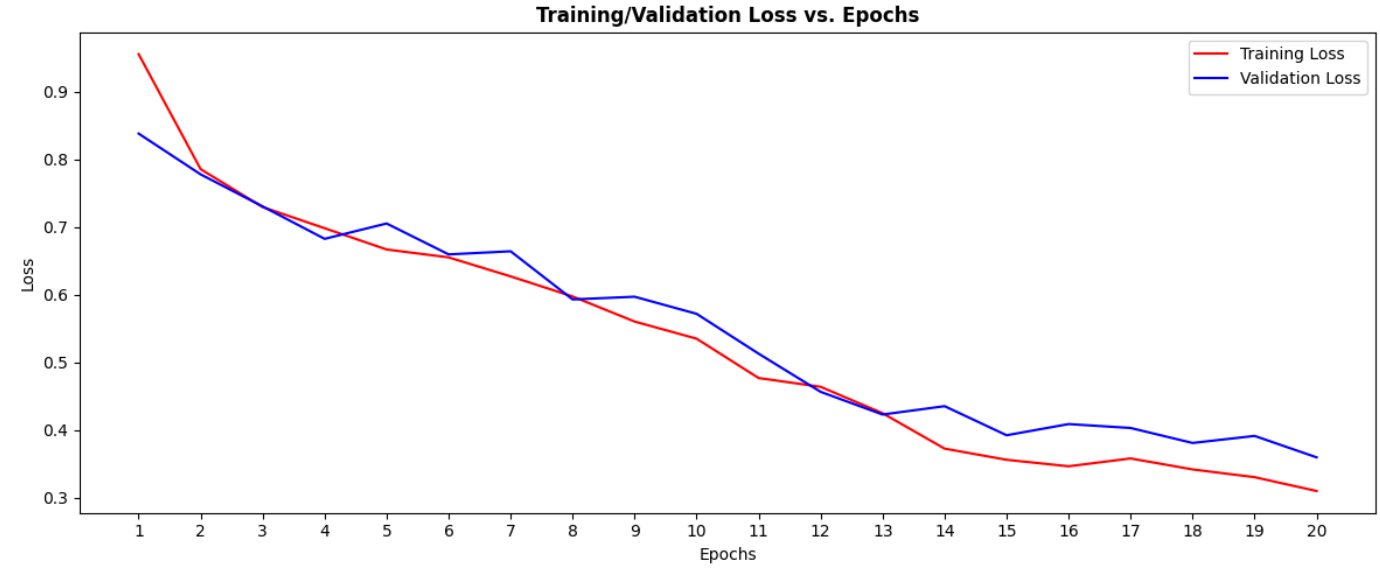
```python
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3), kernel_regularizer=l2
    MaxPooling2D((2, 2)),
    Dropout(0.25),
    Conv2D(64, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    MaxPooling2D((2, 2)),
    Dropout(0.25),
    Flatten(),
    Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
    Dropout(0.25),
    Dense(1, activation='sigmoid')
])

# Compiling the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Training the model
history_64_dropout_l2 = model.fit(train_generator,
                                  validation_data=validation_generator,
                                  epochs=20, verbose=2)
```

```
Epoch 1/20
50/50 - 5s - loss: 0.9553 - accuracy: 0.5369 - val_loss: 0.8380 - val_accuracy: 0.5450 -
5s/epoch - 102ms/step
Epoch 2/20
50/50 - 4s - loss: 0.7854 - accuracy: 0.6119 - val_loss: 0.7777 - val_accuracy: 0.6200 -
4s/epoch - 86ms/step
Epoch 3/20
50/50 - 4s - loss: 0.7294 - accuracy: 0.6369 - val_loss: 0.7304 - val_accuracy: 0.6500 -
4s/epoch - 84ms/step
Epoch 4/20
50/50 - 4s - loss: 0.6982 - accuracy: 0.6538 - val_loss: 0.6825 - val_accuracy: 0.6800 -
4s/epoch - 86ms/step
Epoch 5/20
50/50 - 4s - loss: 0.6668 - accuracy: 0.6681 - val_loss: 0.7052 - val_accuracy: 0.6425 -
4s/epoch - 82ms/step
Epoch 6/20
50/50 - 4s - loss: 0.6551 - accuracy: 0.6888 - val_loss: 0.6595 - val_accuracy: 0.6900 -
4s/epoch - 83ms/step
Epoch 7/20
50/50 - 4s - loss: 0.6270 - accuracy: 0.6994 - val_loss: 0.6641 - val_accuracy: 0.6750 -
4s/epoch - 82ms/step
Epoch 8/20
50/50 - 4s - loss: 0.5974 - accuracy: 0.7269 - val_loss: 0.5930 - val_accuracy: 0.7625 -
4s/epoch - 83ms/step
Epoch 9/20
50/50 - 4s - loss: 0.5603 - accuracy: 0.7619 - val_loss: 0.5970 - val_accuracy: 0.7525 -
4s/epoch - 81ms/step
Epoch 10/20
50/50 - 4s - loss: 0.5350 - accuracy: 0.7919 - val_loss: 0.5717 - val_accuracy: 0.7950 -
4s/epoch - 84ms/step
Epoch 11/20
50/50 - 4s - loss: 0.4769 - accuracy: 0.8331 - val_loss: 0.5129 - val_accuracy: 0.8425 -
4s/epoch - 82ms/step
Epoch 12/20
50/50 - 4s - loss: 0.4639 - accuracy: 0.8525 - val_loss: 0.4565 - val_accuracy: 0.8775 -
4s/epoch - 82ms/step
Epoch 13/20
50/50 - 4s - loss: 0.4249 - accuracy: 0.8838 - val_loss: 0.4230 - val_accuracy: 0.8850 -
4s/epoch - 83ms/step
Epoch 14/20
50/50 - 4s - loss: 0.3726 - accuracy: 0.8938 - val_loss: 0.4353 - val_accuracy: 0.8675 -
4s/epoch - 84ms/step
Epoch 15/20
50/50 - 4s - loss: 0.3561 - accuracy: 0.9038 - val_loss: 0.3923 - val_accuracy: 0.8825 -
4s/epoch - 83ms/step
```

```
Epoch 16/20
50/50 - 4s - loss: 0.3465 - accuracy: 0.9087 - val_loss: 0.4089 - val_accuracy: 0.8875 -
4s/epoch - 89ms/step
Epoch 17/20
50/50 - 5s - loss: 0.3581 - accuracy: 0.9056 - val_loss: 0.4031 - val_accuracy: 0.8900 -
5s/epoch - 93ms/step
Epoch 18/20
50/50 - 4s - loss: 0.3419 - accuracy: 0.9137 - val_loss: 0.3809 - val_accuracy: 0.8900 -
4s/epoch - 88ms/step
Epoch 19/20
50/50 - 4s - loss: 0.3304 - accuracy: 0.9162 - val_loss: 0.3914 - val_accuracy: 0.8875 -
4s/epoch - 82ms/step
Epoch 20/20
50/50 - 4s - loss: 0.3100 - accuracy: 0.9250 - val_loss: 0.3597 - val_accuracy: 0.9000 -
4s/epoch - 82ms/step
```

In [10]:
```
# display plot
plot_acc_loss(history_64_dropout_l2)
```





Introducing dropout and L2 regularization appeared to improve the model. The validation loss/accuracy curves are more steady and the accuracy reached .90 in the new model. The original model without dropout or l2 regularization had a peak accuracy of .8450 and showed signs of overfitting. Furthermore, the training and validation accuracy of this model are slightly closer to one another by the end of training when compared to the previous model which only had dropout (no l2 regularization).

Initially, we re-sized the targets to be 64x64. let's try re-sizing the images to 128x128 and compare the validation accuracy to the previous model.

In [11]:
```
train_generator, validation_generator = create_data_generators(
    base_dir='cell_images_subset',
    target_size=(128,128),
    batch_size=32)
```

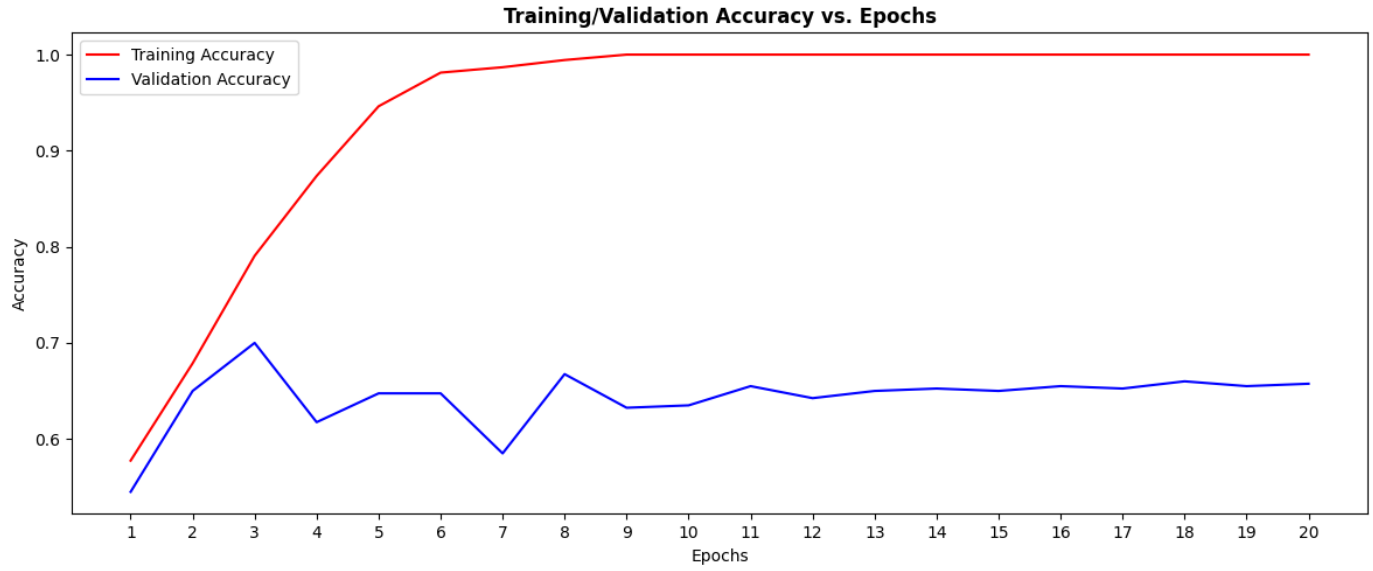
```
Found 1600 images belonging to 2 classes.
Found 400 images belonging to 2 classes.
```

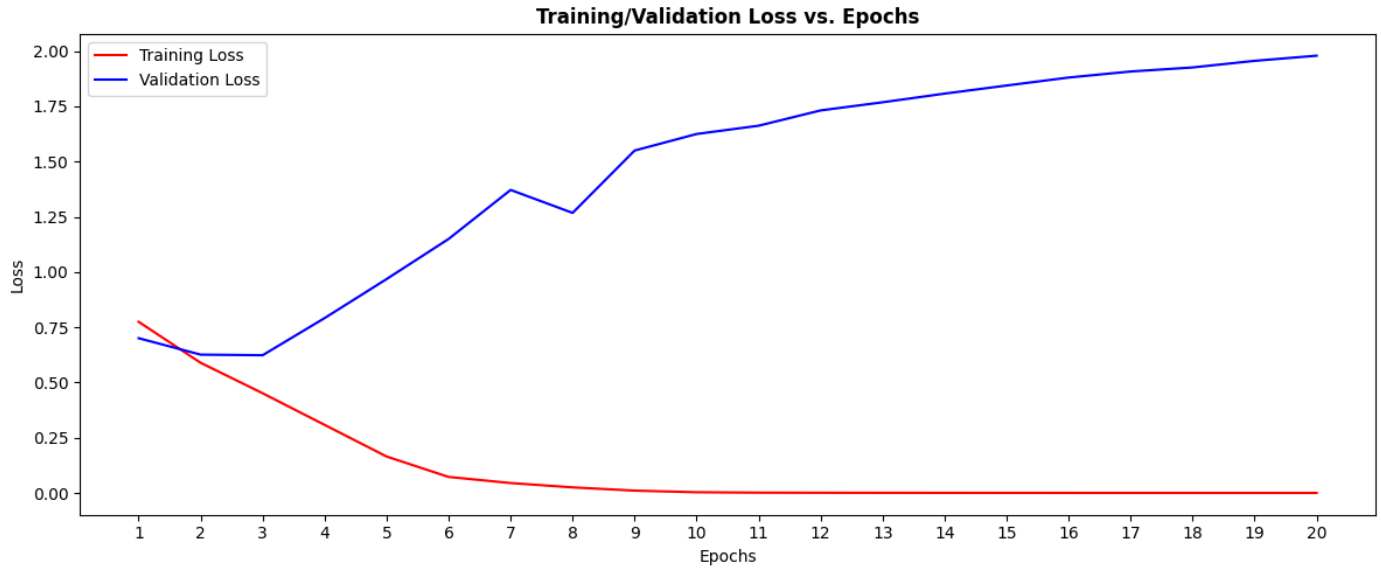
In [12]:
```
# build CNN
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compiling the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Training the model
history_128_by_128 = model.fit(train_generator,
                               validation_data=validation_generator,
                               epochs=20, verbose=2)
```

```
Epoch 1/20
50/50 - 13s - loss: 0.7748 - accuracy: 0.5775 - val_loss: 0.7006 - val_accuracy: 0.5450
- 13s/epoch - 252ms/step
Epoch 2/20
50/50 - 11s - loss: 0.5894 - accuracy: 0.6787 - val_loss: 0.6262 - val_accuracy: 0.6500
- 11s/epoch - 223ms/step
Epoch 3/20
50/50 - 12s - loss: 0.4514 - accuracy: 0.7906 - val_loss: 0.6235 - val_accuracy: 0.7000
- 12s/epoch - 247ms/step
Epoch 4/20
50/50 - 13s - loss: 0.3082 - accuracy: 0.8737 - val_loss: 0.7917 - val_accuracy: 0.6175
- 13s/epoch - 256ms/step
Epoch 5/20
50/50 - 13s - loss: 0.1650 - accuracy: 0.9463 - val_loss: 0.9683 - val_accuracy: 0.6475
- 13s/epoch - 255ms/step
Epoch 6/20
50/50 - 13s - loss: 0.0725 - accuracy: 0.9812 - val_loss: 1.1499 - val_accuracy: 0.6475
- 13s/epoch - 260ms/step
Epoch 7/20
50/50 - 13s - loss: 0.0450 - accuracy: 0.9869 - val_loss: 1.3717 - val_accuracy: 0.5850
- 13s/epoch - 265ms/step
Epoch 8/20
50/50 - 14s - loss: 0.0253 - accuracy: 0.9944 - val_loss: 1.2678 - val_accuracy: 0.6675
- 14s/epoch - 272ms/step
Epoch 9/20
50/50 - 14s - loss: 0.0106 - accuracy: 1.0000 - val_loss: 1.5500 - val_accuracy: 0.6325
- 14s/epoch - 290ms/step
Epoch 10/20
50/50 - 14s - loss: 0.0033 - accuracy: 1.0000 - val_loss: 1.6249 - val_accuracy: 0.6350
- 14s/epoch - 287ms/step
Epoch 11/20
50/50 - 14s - loss: 0.0015 - accuracy: 1.0000 - val_loss: 1.6624 - val_accuracy: 0.6550
- 14s/epoch - 281ms/step
Epoch 12/20
50/50 - 16s - loss: 9.8253e-04 - accuracy: 1.0000 - val_loss: 1.7316 - val_accuracy: 0.6
```

```
425 - 16s/epoch - 320ms/step
Epoch 13/20
50/50 - 13s - loss: 7.2491e-04 - accuracy: 1.0000 - val_loss: 1.7685 - val_accuracy: 0.6
500 - 13s/epoch - 267ms/step
Epoch 14/20
50/50 - 13s - loss: 5.7849e-04 - accuracy: 1.0000 - val_loss: 1.8077 - val_accuracy: 0.6
525 - 13s/epoch - 263ms/step
Epoch 15/20
50/50 - 13s - loss: 4.8164e-04 - accuracy: 1.0000 - val_loss: 1.8443 - val_accuracy: 0.6
500 - 13s/epoch - 253ms/step
Epoch 16/20
50/50 - 13s - loss: 4.0997e-04 - accuracy: 1.0000 - val_loss: 1.8805 - val_accuracy: 0.6
550 - 13s/epoch - 253ms/step
Epoch 17/20
50/50 - 13s - loss: 3.5030e-04 - accuracy: 1.0000 - val_loss: 1.9078 - val_accuracy: 0.6
525 - 13s/epoch - 266ms/step
Epoch 18/20
50/50 - 13s - loss: 3.0498e-04 - accuracy: 1.0000 - val_loss: 1.9261 - val_accuracy: 0.6
600 - 13s/epoch - 262ms/step
Epoch 19/20
50/50 - 13s - loss: 2.6496e-04 - accuracy: 1.0000 - val_loss: 1.9560 - val_accuracy: 0.6
550 - 13s/epoch - 268ms/step
Epoch 20/20
50/50 - 15s - loss: 2.3536e-04 - accuracy: 1.0000 - val_loss: 1.9793 - val_accuracy: 0.6
575 - 15s/epoch - 297ms/step
```

In [13]:
```
plot_acc_loss(history_128_by_128)
```





When re-sizing the images to 128x128, the model began to overfit very quickly. The training accuracy climbed to 100% by ~ epoch 10. The vaildation accuracy peaked at 70% and the validation loss steadily rose after epoch 3.

Thus far, the 64x64 model performed better, and required significantly less compute time.

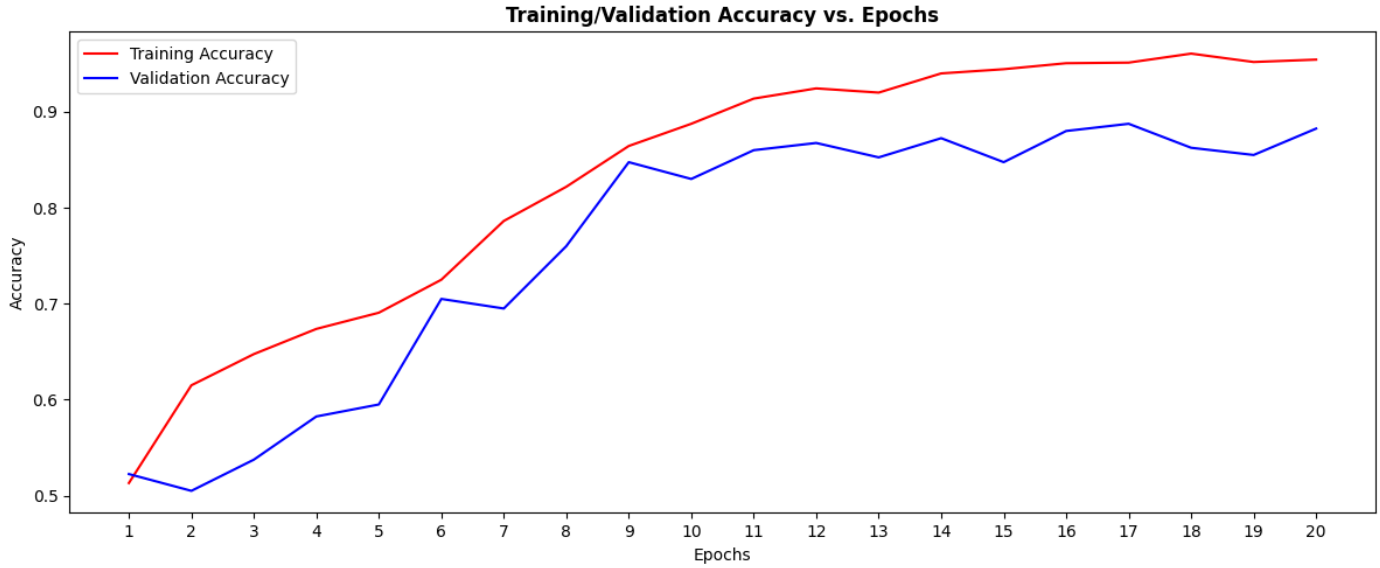
Next we will try introducing regularization and dropout in an attempt to mitigate overfitting in the 128x128 model.

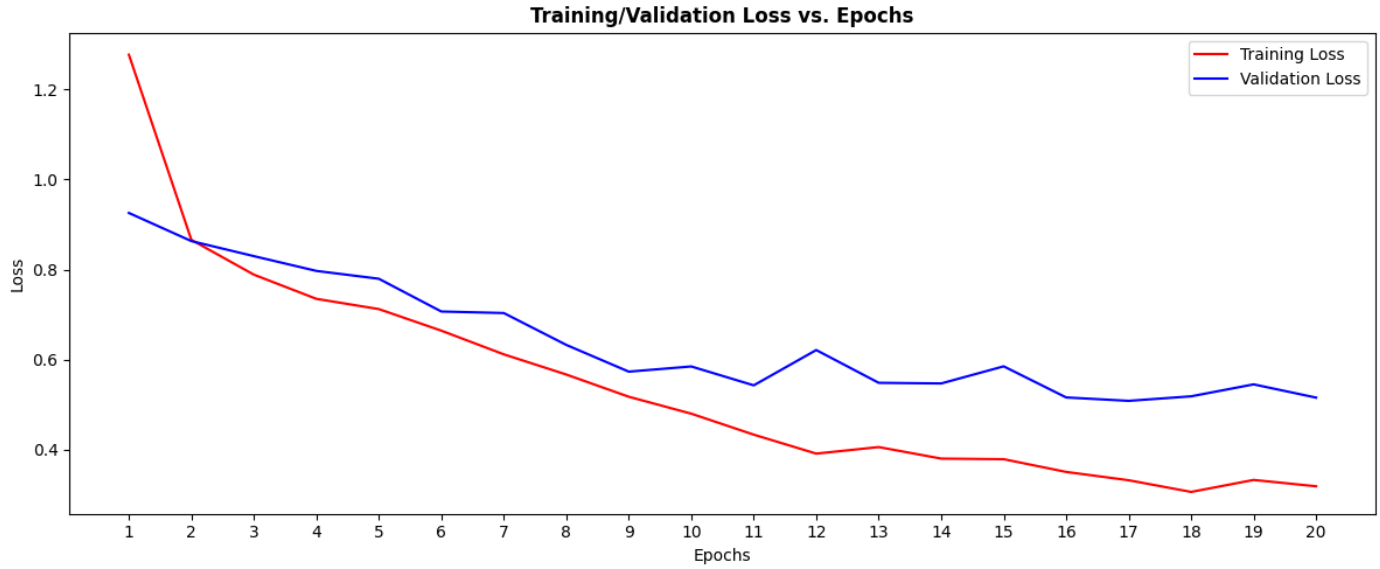
In [14]:
```
# Define the model with regularization and dropout
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3), kernel_regularizer=
    MaxPooling2D((2, 2)),
    Dropout(0.25),
    Conv2D(64, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    MaxPooling2D((2, 2)),
    Dropout(0.25),
    Flatten(),
    Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

history_128_l2_dropout = model.fit(train_generator,
                                   validation_data=validation_generator,
                                   epochs=20)
```

```
Epoch 1/20
50/50 [==============================] - 18s 348ms/step - loss: 1.2771 - accuracy: 0.513
1 - val_loss: 0.9256 - val_accuracy: 0.5225
Epoch 2/20
50/50 [==============================] - 16s 311ms/step - loss: 0.8658 - accuracy: 0.615
0 - val_loss: 0.8631 - val_accuracy: 0.5050
Epoch 3/20
50/50 [==============================] - 16s 322ms/step - loss: 0.7886 - accuracy: 0.647
5 - val_loss: 0.8297 - val_accuracy: 0.5375
Epoch 4/20
50/50 [==============================] - 17s 348ms/step - loss: 0.7349 - accuracy: 0.673
7 - val_loss: 0.7969 - val_accuracy: 0.5825
Epoch 5/20
50/50 [==============================] - 17s 343ms/step - loss: 0.7121 - accuracy: 0.690
6 - val_loss: 0.7795 - val_accuracy: 0.5950
Epoch 6/20
50/50 [==============================] - 16s 328ms/step - loss: 0.6642 - accuracy: 0.725
0 - val_loss: 0.7067 - val_accuracy: 0.7050
Epoch 7/20
50/50 [==============================] - 17s 330ms/step - loss: 0.6116 - accuracy: 0.786
2 - val_loss: 0.7033 - val_accuracy: 0.6950
Epoch 8/20
50/50 [==============================] - 17s 340ms/step - loss: 0.5667 - accuracy: 0.821
9 - val_loss: 0.6326 - val_accuracy: 0.7600
Epoch 9/20
50/50 [==============================] - 17s 342ms/step - loss: 0.5174 - accuracy: 0.864
4 - val_loss: 0.5732 - val_accuracy: 0.8475
Epoch 10/20
50/50 [==============================] - 18s 351ms/step - loss: 0.4798 - accuracy: 0.887
5 - val_loss: 0.5847 - val_accuracy: 0.8300
Epoch 11/20
50/50 [==============================] - 17s 334ms/step - loss: 0.4332 - accuracy: 0.913
7 - val_loss: 0.5427 - val_accuracy: 0.8600
```

```
Epoch 12/20
50/50 [==============================] - 17s 338ms/step - loss: 0.3911 - accuracy: 0.924
4 - val_loss: 0.6211 - val_accuracy: 0.8675
Epoch 13/20
50/50 [==============================] - 17s 340ms/step - loss: 0.4057 - accuracy: 0.920
0 - val_loss: 0.5483 - val_accuracy: 0.8525
Epoch 14/20
50/50 [==============================] - 17s 348ms/step - loss: 0.3801 - accuracy: 0.940
0 - val_loss: 0.5470 - val_accuracy: 0.8725
Epoch 15/20
50/50 [==============================] - 17s 335ms/step - loss: 0.3787 - accuracy: 0.944
4 - val_loss: 0.5849 - val_accuracy: 0.8475
Epoch 16/20
50/50 [==============================] - 17s 333ms/step - loss: 0.3505 - accuracy: 0.950
6 - val_loss: 0.5159 - val_accuracy: 0.8800
Epoch 17/20
50/50 [==============================] - 18s 352ms/step - loss: 0.3320 - accuracy: 0.951
3 - val_loss: 0.5083 - val_accuracy: 0.8875
Epoch 18/20
50/50 [==============================] - 17s 346ms/step - loss: 0.3060 - accuracy: 0.960
6 - val_loss: 0.5184 - val_accuracy: 0.8625
Epoch 19/20
50/50 [==============================] - 17s 347ms/step - loss: 0.3327 - accuracy: 0.951
9 - val_loss: 0.5449 - val_accuracy: 0.8550
Epoch 20/20
50/50 [==============================] - 17s 348ms/step - loss: 0.3185 - accuracy: 0.954
4 - val_loss: 0.5155 - val_accuracy: 0.8825
```

In [15]:
```
plot_acc_loss(history_128_l2_dropout)
```





The introduction of dropout and l2 regularization significantly mitigated the amount of overfitting in the model. We achieved a validation accuracy of almost 90%, which is a significant improvement over the previous model.

Next we can try increasing the complexity of the model by adding more convolutional layers and increasing the number of filters. We will also introduce data augmentation which can help mitigate the increased risk of overfitting that arises from using a more complex model.

Model with increased complexity:

1. Additional Convolutional Layer with 128 filters
2. Additional Dropout Layer with rate of .5
3. Additional Dense Layer with 64 neurons

In [16]:
```
# create generators with augmentation
train_generator, validation_generator = create_data_generators(
    base_dir='cell_images_subset',
    target_size=(128,128),
    batch_size=32,
    augmentation=True)
```

```
Found 1600 images belonging to 2 classes.
Found 400 images belonging to 2 classes.
```

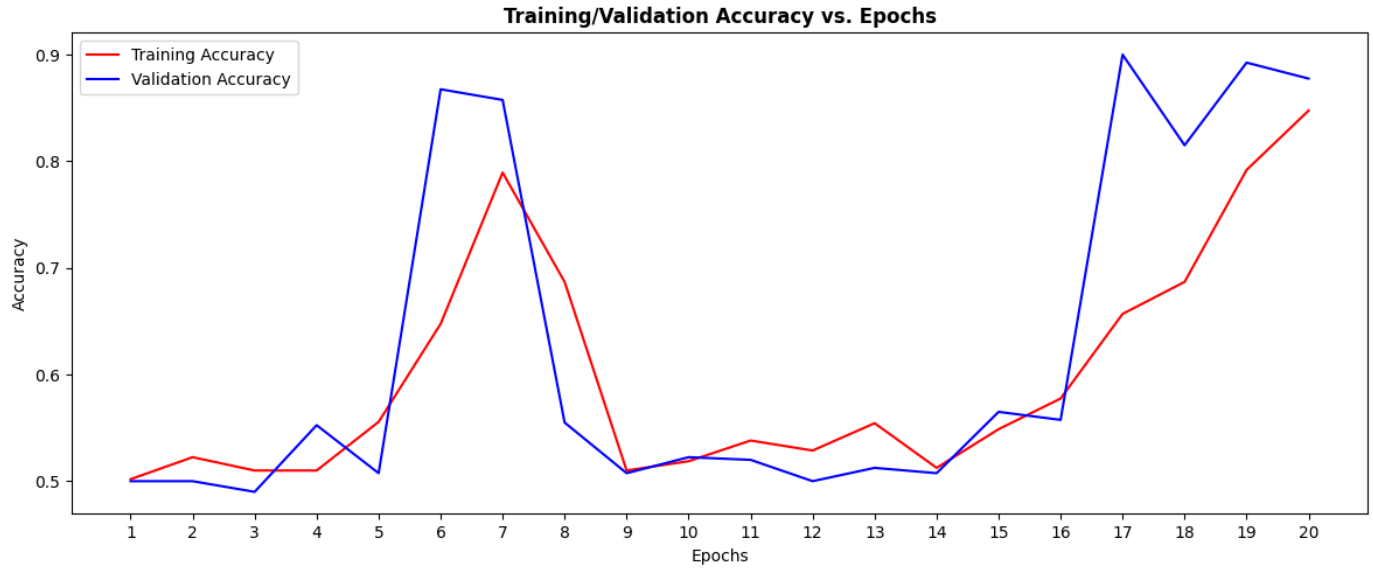
In [17]:
```
# Define the CNN model
```

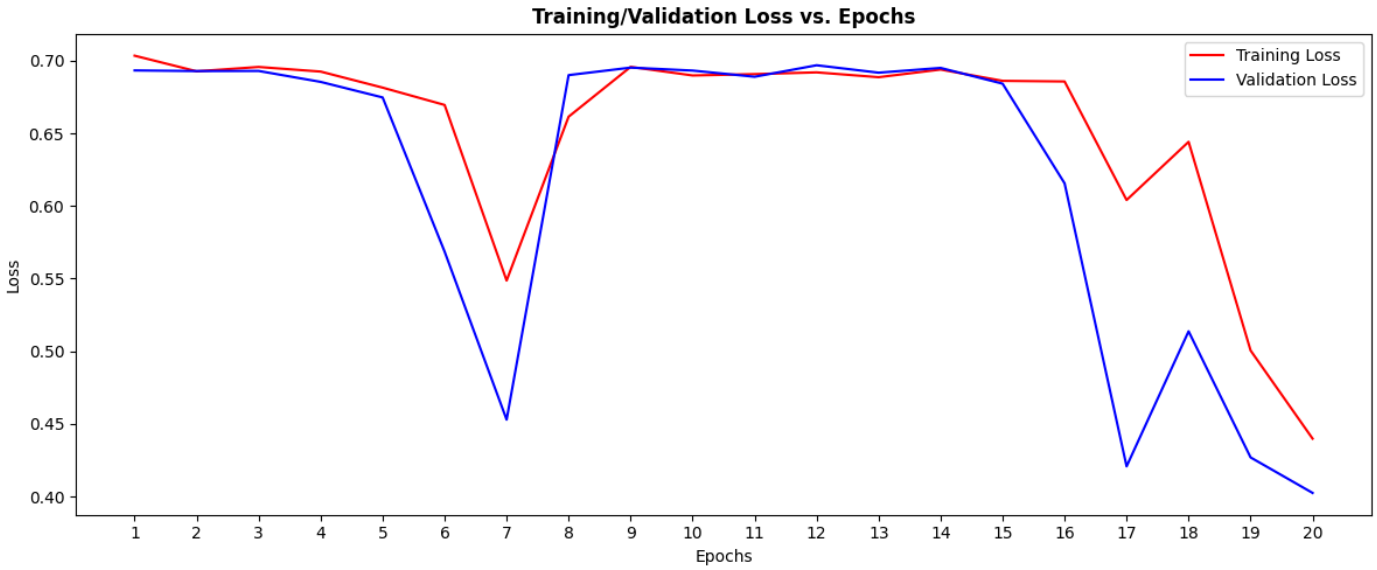
```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model with augmented data
history_complex = model.fit(train_generator,
                            validation_data=validation_generator,
                            epochs=20,
                            verbose=2)
```

```
Epoch 1/20
50/50 - 21s - loss: 0.7035 - accuracy: 0.5019 - val_loss: 0.6933 - val_accuracy: 0.5000
- 21s/epoch - 414ms/step
Epoch 2/20
50/50 - 19s - loss: 0.6927 - accuracy: 0.5225 - val_loss: 0.6929 - val_accuracy: 0.5000
- 19s/epoch - 381ms/step
Epoch 3/20
50/50 - 19s - loss: 0.6957 - accuracy: 0.5100 - val_loss: 0.6930 - val_accuracy: 0.4900
- 19s/epoch - 380ms/step
Epoch 4/20
50/50 - 19s - loss: 0.6925 - accuracy: 0.5100 - val_loss: 0.6855 - val_accuracy: 0.5525
- 19s/epoch - 386ms/step
Epoch 5/20
50/50 - 21s - loss: 0.6815 - accuracy: 0.5556 - val_loss: 0.6747 - val_accuracy: 0.5075
- 21s/epoch - 427ms/step
Epoch 6/20
50/50 - 20s - loss: 0.6696 - accuracy: 0.6475 - val_loss: 0.5685 - val_accuracy: 0.8675
- 20s/epoch - 399ms/step
Epoch 7/20
50/50 - 20s - loss: 0.5487 - accuracy: 0.7894 - val_loss: 0.4529 - val_accuracy: 0.8575
- 20s/epoch - 397ms/step
Epoch 8/20
50/50 - 19s - loss: 0.6615 - accuracy: 0.6869 - val_loss: 0.6901 - val_accuracy: 0.5550
- 19s/epoch - 386ms/step
Epoch 9/20
50/50 - 20s - loss: 0.6958 - accuracy: 0.5100 - val_loss: 0.6953 - val_accuracy: 0.5075
- 20s/epoch - 395ms/step
Epoch 10/20
50/50 - 20s - loss: 0.6899 - accuracy: 0.5188 - val_loss: 0.6932 - val_accuracy: 0.5225
- 20s/epoch - 398ms/step
Epoch 11/20
50/50 - 21s - loss: 0.6908 - accuracy: 0.5381 - val_loss: 0.6890 - val_accuracy: 0.5200
- 21s/epoch - 412ms/step
Epoch 12/20
50/50 - 19s - loss: 0.6920 - accuracy: 0.5288 - val_loss: 0.6969 - val_accuracy: 0.5000
- 19s/epoch - 388ms/step
Epoch 13/20
50/50 - 18s - loss: 0.6887 - accuracy: 0.5544 - val_loss: 0.6918 - val_accuracy: 0.5125
- 18s/epoch - 370ms/step
Epoch 14/20
50/50 - 17s - loss: 0.6940 - accuracy: 0.5125 - val_loss: 0.6951 - val_accuracy: 0.5075
- 17s/epoch - 350ms/step
```

```
Epoch 15/20
50/50 - 18s - loss: 0.6862 - accuracy: 0.5487 - val_loss: 0.6842 - val_accuracy: 0.5650
- 18s/epoch - 362ms/step
Epoch 16/20
50/50 - 19s - loss: 0.6857 - accuracy: 0.5775 - val_loss: 0.6156 - val_accuracy: 0.5575
- 19s/epoch - 379ms/step
Epoch 17/20
50/50 - 20s - loss: 0.6041 - accuracy: 0.6569 - val_loss: 0.4207 - val_accuracy: 0.9000
- 20s/epoch - 402ms/step
Epoch 18/20
50/50 - 19s - loss: 0.6442 - accuracy: 0.6869 - val_loss: 0.5137 - val_accuracy: 0.8150
- 19s/epoch - 381ms/step
Epoch 19/20
50/50 - 18s - loss: 0.5005 - accuracy: 0.7919 - val_loss: 0.4269 - val_accuracy: 0.8925
- 18s/epoch - 359ms/step
Epoch 20/20
50/50 - 18s - loss: 0.4398 - accuracy: 0.8475 - val_loss: 0.4024 - val_accuracy: 0.8775
- 18s/epoch - 351ms/step
```

In [18]:
```
plot_acc_loss(history_complex)
```





This Model achieved the highest validation accuracy we have seen thus far with a peak of .90. However, the validation curves are rather sporadic. Let's try introducing batch normalization into more complex model in an attempt to stabilize the learning process and smoothen our validation curves.

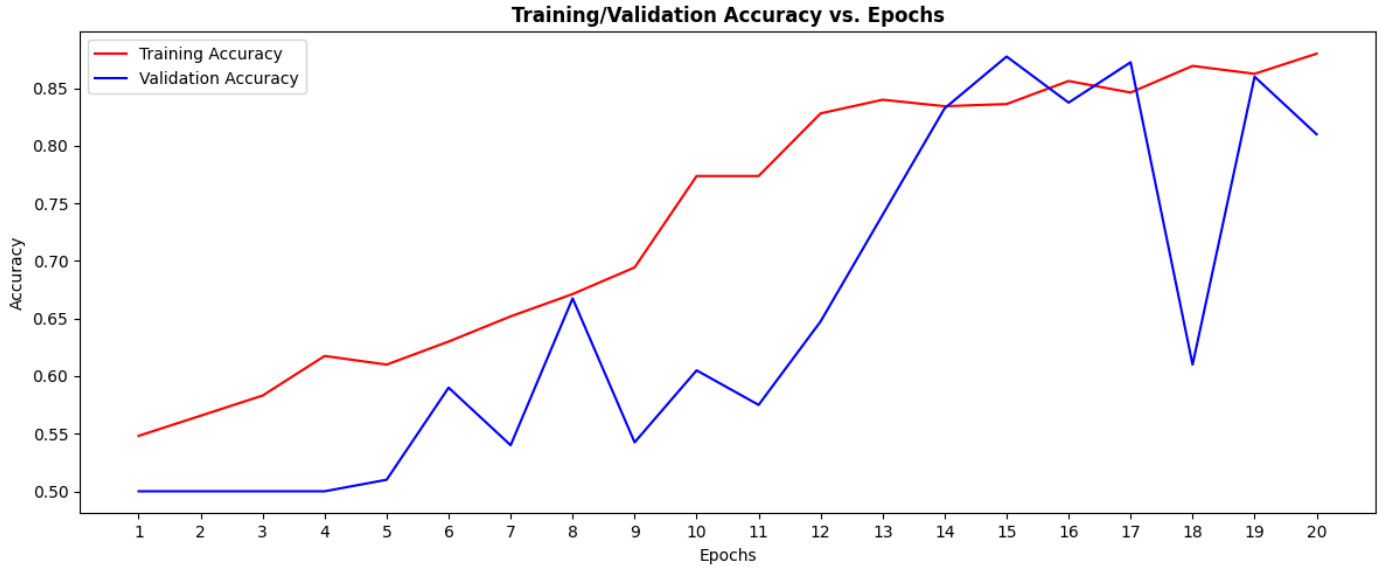
In [19]:
```
# Define the CNN model with Batch Normalization and L2 Regularization
model = Sequential([
```

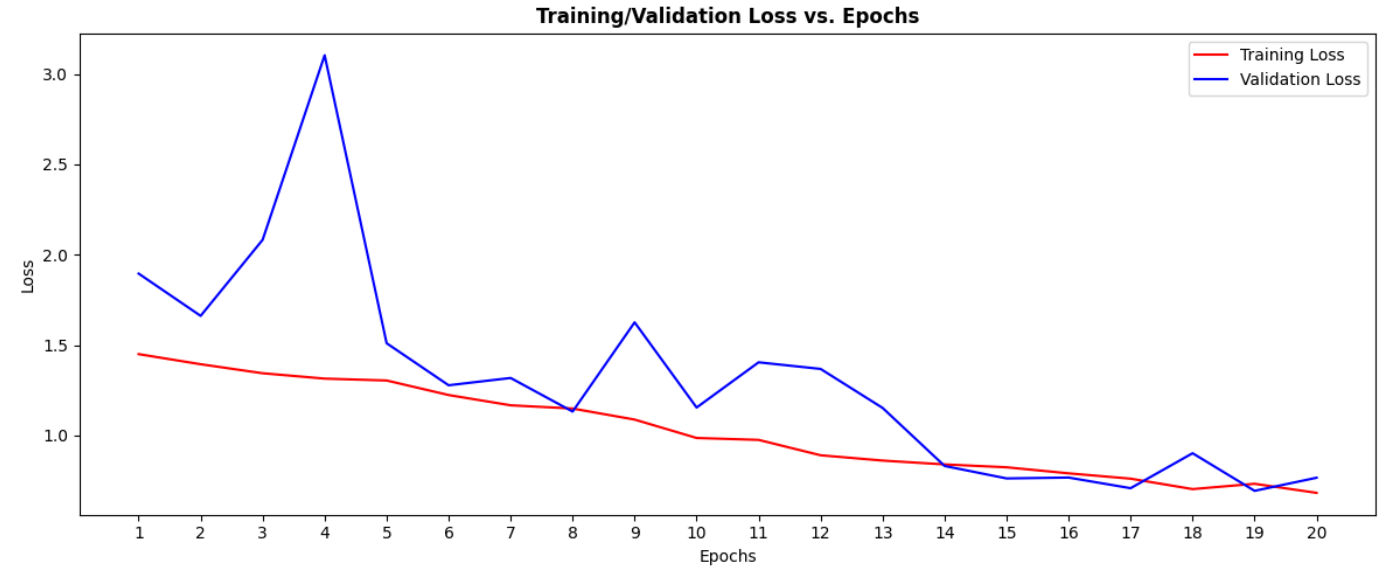
```
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3), kernel_regularizer=
    BatchNormalization(), # adding batch normalization between layers
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(), # adding batch normalization between layers
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(), # adding batch normalization between layers
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(), # adding batch normalization between layers
    Dropout(0.5),

    Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(), # adding batch normalization between layers
    Dropout(0.5),

    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model with augmented data
history_complex_bn = model.fit(train_generator,
                               validation_data=validation_generator,
                               epochs=20,
                               verbose=2)
```

```
Epoch 1/20
50/50 - 29s - loss: 1.4497 - accuracy: 0.5481 - val_loss: 1.8954 - val_accuracy: 0.5000
- 29s/epoch - 573ms/step
Epoch 2/20
50/50 - 27s - loss: 1.3938 - accuracy: 0.5656 - val_loss: 1.6615 - val_accuracy: 0.5000
- 27s/epoch - 535ms/step
Epoch 3/20
50/50 - 27s - loss: 1.3438 - accuracy: 0.5831 - val_loss: 2.0824 - val_accuracy: 0.5000
- 27s/epoch - 531ms/step
Epoch 4/20
50/50 - 25s - loss: 1.3142 - accuracy: 0.6175 - val_loss: 3.1040 - val_accuracy: 0.5000
- 25s/epoch - 504ms/step
Epoch 5/20
50/50 - 25s - loss: 1.3037 - accuracy: 0.6100 - val_loss: 1.5100 - val_accuracy: 0.5100
- 25s/epoch - 494ms/step
Epoch 6/20
50/50 - 26s - loss: 1.2233 - accuracy: 0.6300 - val_loss: 1.2774 - val_accuracy: 0.5900
- 26s/epoch - 515ms/step
Epoch 7/20
50/50 - 27s - loss: 1.1664 - accuracy: 0.6519 - val_loss: 1.3177 - val_accuracy: 0.5400
- 27s/epoch - 531ms/step
Epoch 8/20
50/50 - 27s - loss: 1.1486 - accuracy: 0.6712 - val_loss: 1.1318 - val_accuracy: 0.6675
- 27s/epoch - 548ms/step
Epoch 9/20
50/50 - 30s - loss: 1.0874 - accuracy: 0.6944 - val_loss: 1.6257 - val_accuracy: 0.5425
- 30s/epoch - 594ms/step
Epoch 10/20
50/50 - 28s - loss: 0.9856 - accuracy: 0.7738 - val_loss: 1.1541 - val_accuracy: 0.6050
- 28s/epoch - 553ms/step
```

```
Epoch 11/20
50/50 - 29s - loss: 0.9750 - accuracy: 0.7738 - val_loss: 1.4044 - val_accuracy: 0.5750
- 29s/epoch - 576ms/step
Epoch 12/20
50/50 - 30s - loss: 0.8899 - accuracy: 0.8281 - val_loss: 1.3678 - val_accuracy: 0.6475
- 30s/epoch - 594ms/step
Epoch 13/20
50/50 - 28s - loss: 0.8605 - accuracy: 0.8400 - val_loss: 1.1519 - val_accuracy: 0.7400
- 28s/epoch - 559ms/step
Epoch 14/20
50/50 - 29s - loss: 0.8392 - accuracy: 0.8344 - val_loss: 0.8307 - val_accuracy: 0.8325
- 29s/epoch - 585ms/step
Epoch 15/20
50/50 - 32s - loss: 0.8233 - accuracy: 0.8363 - val_loss: 0.7617 - val_accuracy: 0.8775
- 32s/epoch - 631ms/step
Epoch 16/20
50/50 - 30s - loss: 0.7899 - accuracy: 0.8562 - val_loss: 0.7665 - val_accuracy: 0.8375
- 30s/epoch - 593ms/step
Epoch 17/20
50/50 - 28s - loss: 0.7603 - accuracy: 0.8462 - val_loss: 0.7076 - val_accuracy: 0.8725
- 28s/epoch - 560ms/step
Epoch 18/20
50/50 - 28s - loss: 0.7026 - accuracy: 0.8694 - val_loss: 0.9012 - val_accuracy: 0.6100
- 28s/epoch - 561ms/step
Epoch 19/20
50/50 - 28s - loss: 0.7320 - accuracy: 0.8625 - val_loss: 0.6930 - val_accuracy: 0.8600
- 28s/epoch - 551ms/step
Epoch 20/20
50/50 - 28s - loss: 0.6818 - accuracy: 0.8800 - val_loss: 0.7655 - val_accuracy: 0.8100
- 28s/epoch - 567ms/step
```

In [20]:
```
plot_acc_loss(history_complex_bn)
```





This introduction of batch normalization appeared to make our validation curves more sporadic. It is possible that a batch size of 32 was not large enough to output relatively consistent means and variances for each batch. If there is a large disparity between the computed statistics of each batch, training can be destabilized and negatively affect model performance. When we train on the entire dataset, we will use a larger batch size and perhaps batch normalization will be more effective.

We will continue to build off of this model in attempt to continue improving its performance.

Our next model will implement early stopping and a learning rate scheduler. For our learning rate scheduler, we will use the keras callback method 'ReduceLROnPlateau'. This method will reduces the learning rate when a specified metric - in this case validation loss - stops improving for a specified number of epochs.

In [21]:
```
# Add Early stopping and learning rate scheduler to the model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3), kernel_regularizer=
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
```

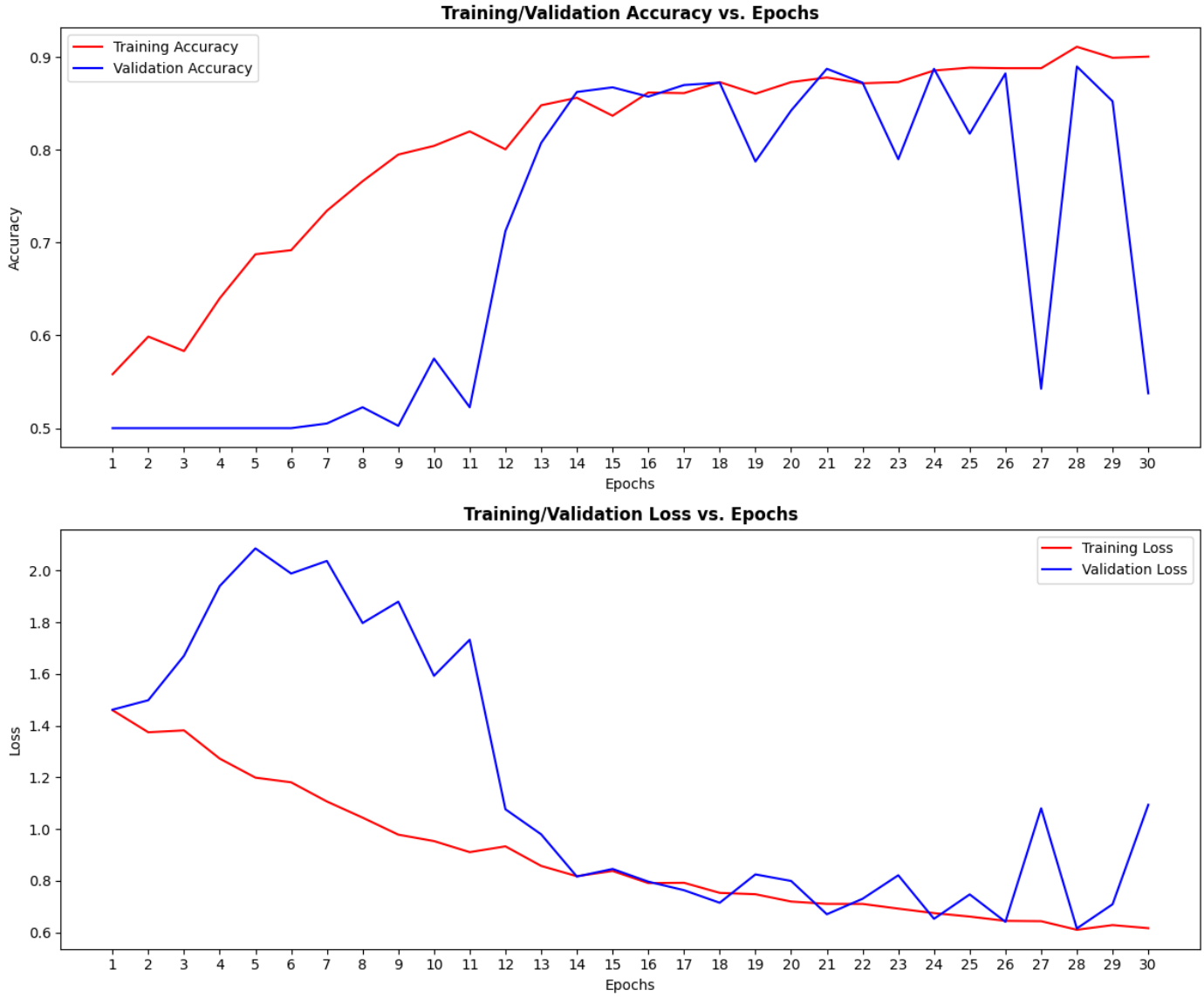
```
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.5),

    Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.5),

    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])


# Callbacks for learning rate adjustment, early stopping, and model checkpoint
callbacks = [
    ModelCheckpoint('best_model_v1.h5',
                    save_best_only=True,
                    monitor='val_accuracy'),
    ReduceLROnPlateau(monitor='val_accuracy',
                      factor=0.5,
                      patience=3,
                      min_lr=1e-6)
]

# Train the model with augmented data, increase epochs to 30 now that we have implemente
history_complex_lr_sched = model.fit(train_generator,
                                     validation_data=validation_generator,
                                     callbacks=callbacks,
                                     epochs=30,
                                     verbose=2)
```

```
Epoch 1/30
50/50 - 31s - loss: 1.4597 - accuracy: 0.5581 - val_loss: 1.4617 - val_accuracy: 0.5000
- lr: 0.0010 - 31s/epoch - 613ms/step
Epoch 2/30
C:\Users\gdgun\COMP4531\my_tensorflow_project\venv\lib\site-packages\keras\src\engine\tr
aining.py:3000: UserWarning: You are saving your model as an HDF5 file via `model.save()
`. This file format is considered legacy. We recommend using instead the native Keras fo
rmat, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
50/50 - 28s - loss: 1.3744 - accuracy: 0.5987 - val_loss: 1.4983 - val_accuracy: 0.5000
- lr: 0.0010 - 28s/epoch - 570ms/step
Epoch 3/30
50/50 - 28s - loss: 1.3817 - accuracy: 0.5831 - val_loss: 1.6701 - val_accuracy: 0.5000
- lr: 0.0010 - 28s/epoch - 556ms/step
Epoch 4/30
50/50 - 28s - loss: 1.2727 - accuracy: 0.6400 - val_loss: 1.9397 - val_accuracy: 0.5000
- lr: 0.0010 - 28s/epoch - 552ms/step
Epoch 5/30
50/50 - 28s - loss: 1.1991 - accuracy: 0.6875 - val_loss: 2.0853 - val_accuracy: 0.5000
- lr: 5.0000e-04 - 28s/epoch - 566ms/step
Epoch 6/30
50/50 - 27s - loss: 1.1809 - accuracy: 0.6919 - val_loss: 1.9881 - val_accuracy: 0.5000
- lr: 5.0000e-04 - 27s/epoch - 533ms/step
Epoch 7/30
50/50 - 28s - loss: 1.1069 - accuracy: 0.7344 - val_loss: 2.0367 - val_accuracy: 0.5050
- lr: 5.0000e-04 - 28s/epoch - 554ms/step
Epoch 8/30
```

```
50/50 - 25s - loss: 1.0445 - accuracy: 0.7663 - val_loss: 1.7966 - val_accuracy: 0.5225
- lr: 5.0000e-04 - 25s/epoch - 506ms/step
Epoch 9/30
50/50 - 25s - loss: 0.9786 - accuracy: 0.7950 - val_loss: 1.8792 - val_accuracy: 0.5025
- lr: 5.0000e-04 - 25s/epoch - 506ms/step
Epoch 10/30
50/50 - 24s - loss: 0.9537 - accuracy: 0.8044 - val_loss: 1.5926 - val_accuracy: 0.5750
- lr: 5.0000e-04 - 24s/epoch - 474ms/step
Epoch 11/30
50/50 - 27s - loss: 0.9110 - accuracy: 0.8200 - val_loss: 1.7322 - val_accuracy: 0.5225
- lr: 5.0000e-04 - 27s/epoch - 539ms/step
Epoch 12/30
50/50 - 26s - loss: 0.9334 - accuracy: 0.8006 - val_loss: 1.0768 - val_accuracy: 0.7125
- lr: 5.0000e-04 - 26s/epoch - 516ms/step
Epoch 13/30
50/50 - 25s - loss: 0.8579 - accuracy: 0.8481 - val_loss: 0.9800 - val_accuracy: 0.8075
- lr: 5.0000e-04 - 25s/epoch - 502ms/step
Epoch 14/30
50/50 - 27s - loss: 0.8180 - accuracy: 0.8562 - val_loss: 0.8167 - val_accuracy: 0.8625
- lr: 5.0000e-04 - 27s/epoch - 546ms/step
Epoch 15/30
50/50 - 27s - loss: 0.8383 - accuracy: 0.8369 - val_loss: 0.8460 - val_accuracy: 0.8675
- lr: 5.0000e-04 - 27s/epoch - 546ms/step
Epoch 16/30
50/50 - 26s - loss: 0.7914 - accuracy: 0.8619 - val_loss: 0.7970 - val_accuracy: 0.8575
- lr: 5.0000e-04 - 26s/epoch - 520ms/step
Epoch 17/30
50/50 - 26s - loss: 0.7926 - accuracy: 0.8612 - val_loss: 0.7639 - val_accuracy: 0.8700
- lr: 5.0000e-04 - 26s/epoch - 514ms/step
Epoch 18/30
50/50 - 24s - loss: 0.7536 - accuracy: 0.8731 - val_loss: 0.7153 - val_accuracy: 0.8725
- lr: 5.0000e-04 - 24s/epoch - 479ms/step
Epoch 19/30
50/50 - 26s - loss: 0.7483 - accuracy: 0.8606 - val_loss: 0.8250 - val_accuracy: 0.7875
- lr: 5.0000e-04 - 26s/epoch - 518ms/step
Epoch 20/30
50/50 - 25s - loss: 0.7200 - accuracy: 0.8731 - val_loss: 0.7996 - val_accuracy: 0.8425
- lr: 5.0000e-04 - 25s/epoch - 494ms/step
Epoch 21/30
50/50 - 24s - loss: 0.7111 - accuracy: 0.8781 - val_loss: 0.6709 - val_accuracy: 0.8875
- lr: 5.0000e-04 - 24s/epoch - 471ms/step
Epoch 22/30
50/50 - 23s - loss: 0.7109 - accuracy: 0.8719 - val_loss: 0.7308 - val_accuracy: 0.8725
- lr: 5.0000e-04 - 23s/epoch - 466ms/step
Epoch 23/30
50/50 - 24s - loss: 0.6927 - accuracy: 0.8731 - val_loss: 0.8216 - val_accuracy: 0.7900
- lr: 5.0000e-04 - 24s/epoch - 477ms/step
Epoch 24/30
50/50 - 25s - loss: 0.6754 - accuracy: 0.8856 - val_loss: 0.6533 - val_accuracy: 0.8875
- lr: 5.0000e-04 - 25s/epoch - 510ms/step
Epoch 25/30
50/50 - 26s - loss: 0.6620 - accuracy: 0.8888 - val_loss: 0.7477 - val_accuracy: 0.8175
- lr: 2.5000e-04 - 26s/epoch - 512ms/step
Epoch 26/30
50/50 - 26s - loss: 0.6454 - accuracy: 0.8881 - val_loss: 0.6416 - val_accuracy: 0.8825
- lr: 2.5000e-04 - 26s/epoch - 526ms/step
Epoch 27/30
50/50 - 29s - loss: 0.6441 - accuracy: 0.8881 - val_loss: 1.0806 - val_accuracy: 0.5425
- lr: 2.5000e-04 - 29s/epoch - 583ms/step
Epoch 28/30
50/50 - 28s - loss: 0.6110 - accuracy: 0.9112 - val_loss: 0.6160 - val_accuracy: 0.8900
- lr: 1.2500e-04 - 28s/epoch - 564ms/step
Epoch 29/30
50/50 - 27s - loss: 0.6290 - accuracy: 0.8994 - val_loss: 0.7096 - val_accuracy: 0.8525
- lr: 1.2500e-04 - 27s/epoch - 550ms/step
Epoch 30/30
```

```
50/50 - 27s - loss: 0.6171 - accuracy: 0.9006 - val_loss: 1.0941 - val_accuracy: 0.5375
- lr: 1.2500e-04 - 27s/epoch - 546ms/step
```

In [22]:
```
plot_acc_loss(history_complex_lr_sched)
```



This model showed a steady increase in the validation accuracy up until ~ epoch 17. After that there were steep drops and spikes in both the validation accuracy and loss curves.

Let's try using one of the better-performing models from earlier and modify the target_size to 64x64.

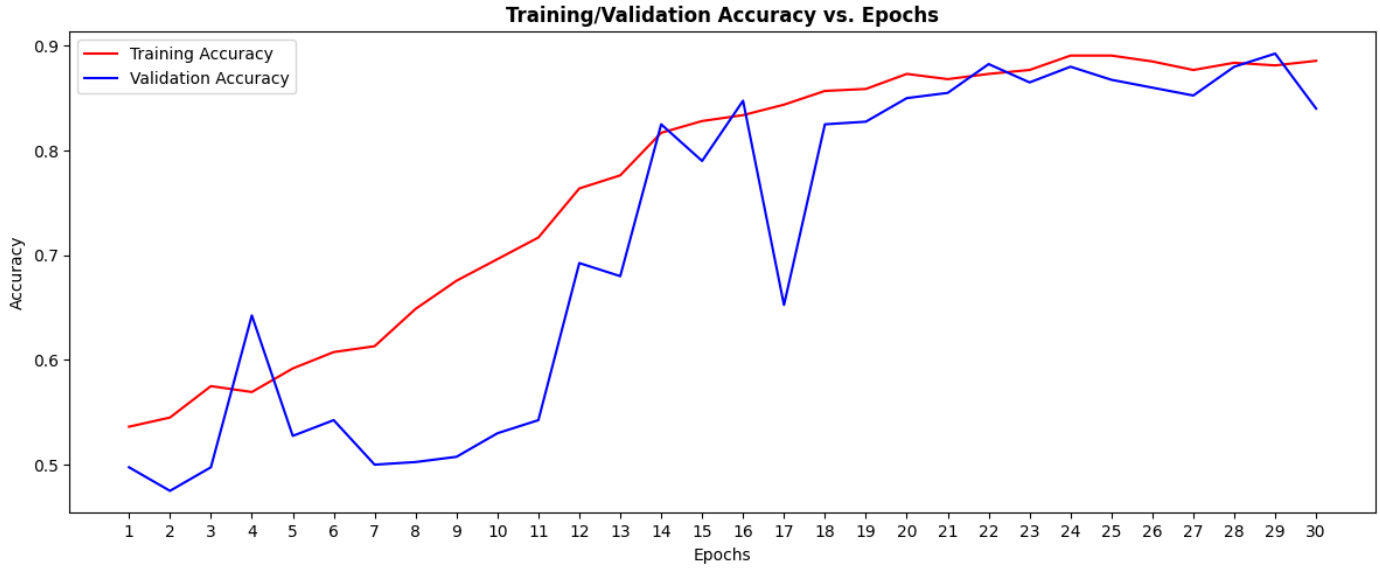
In [23]:
```
# re-define targets to be 64x64
train_generator, validation_generator = create_data_generators(base_dir='cell_images_sub

# Define the CNN model with Batch Normalization and L2 Regularization
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3), kernel_regularizer=l2
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
```

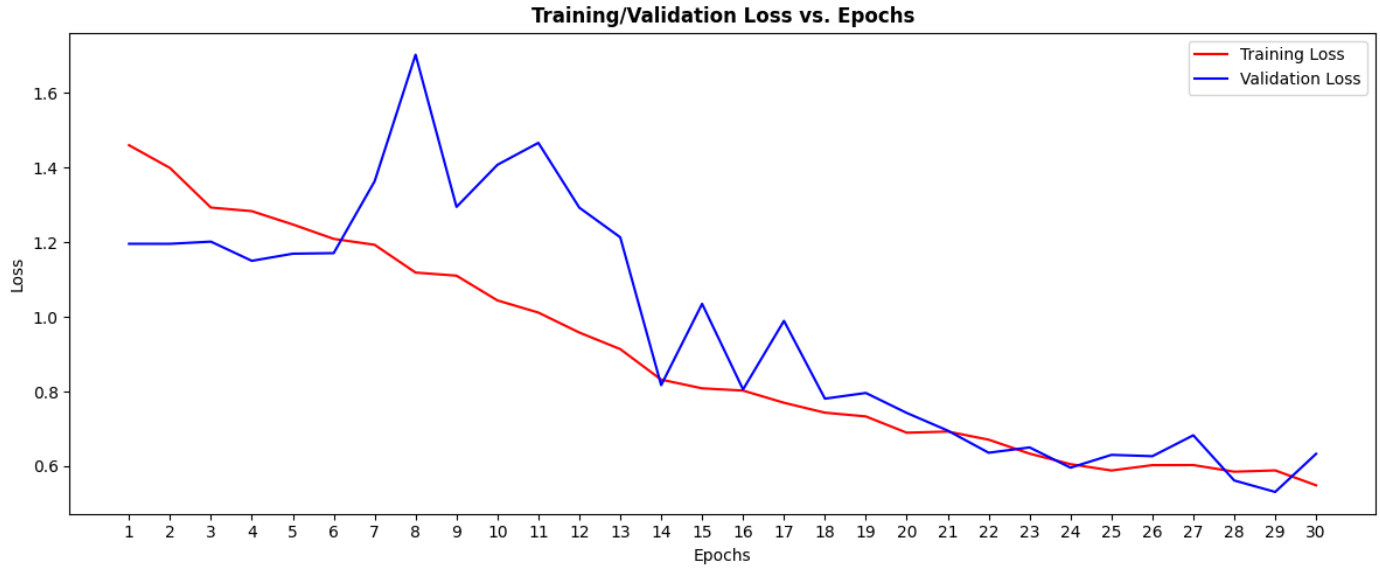
```
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.5),

    Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.5),

    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Callbacks for early stopping, and model checkpoint
callbacks = [
    ModelCheckpoint('best_model_v2.h5',
                    save_best_only=True,
                    monitor='val_accuracy')
]


# Train the model with augmented data
history_complex_64 = model.fit(train_generator,
                               validation_data=validation_generator,
                               callbacks=callbacks,
                               epochs=30,
                               verbose=2)
```

```
Found 1600 images belonging to 2 classes.
Found 400 images belonging to 2 classes.
Epoch 1/30
50/50 - 11s - loss: 1.4598 - accuracy: 0.5362 - val_loss: 1.1957 - val_accuracy: 0.4975
- 11s/epoch - 217ms/step
Epoch 2/30
50/50 - 8s - loss: 1.3989 - accuracy: 0.5450 - val_loss: 1.1956 - val_accuracy: 0.4750 -
8s/epoch - 166ms/step
Epoch 3/30
50/50 - 9s - loss: 1.2928 - accuracy: 0.5750 - val_loss: 1.2015 - val_accuracy: 0.4975 -
9s/epoch - 172ms/step
Epoch 4/30
50/50 - 9s - loss: 1.2831 - accuracy: 0.5694 - val_loss: 1.1501 - val_accuracy: 0.6425 -
9s/epoch - 178ms/step
Epoch 5/30
50/50 - 9s - loss: 1.2475 - accuracy: 0.5919 - val_loss: 1.1691 - val_accuracy: 0.5275 -
9s/epoch - 180ms/step
Epoch 6/30
50/50 - 8s - loss: 1.2089 - accuracy: 0.6075 - val_loss: 1.1707 - val_accuracy: 0.5425 -
8s/epoch - 167ms/step
Epoch 7/30
50/50 - 8s - loss: 1.1930 - accuracy: 0.6131 - val_loss: 1.3632 - val_accuracy: 0.5000 -
8s/epoch - 169ms/step
Epoch 8/30
50/50 - 9s - loss: 1.1186 - accuracy: 0.6488 - val_loss: 1.7022 - val_accuracy: 0.5025 -
9s/epoch - 174ms/step
Epoch 9/30
50/50 - 8s - loss: 1.1103 - accuracy: 0.6756 - val_loss: 1.2945 - val_accuracy: 0.5075 -
8s/epoch - 163ms/step
Epoch 10/30
50/50 - 8s - loss: 1.0440 - accuracy: 0.6963 - val_loss: 1.4073 - val_accuracy: 0.5300 -
8s/epoch - 161ms/step
Epoch 11/30
```

```
50/50 - 8s - loss: 1.0114 - accuracy: 0.7169 - val_loss: 1.4661 - val_accuracy: 0.5425 -
8s/epoch - 160ms/step
Epoch 12/30
50/50 - 8s - loss: 0.9580 - accuracy: 0.7638 - val_loss: 1.2925 - val_accuracy: 0.6925 -
8s/epoch - 164ms/step
Epoch 13/30
50/50 - 9s - loss: 0.9136 - accuracy: 0.7763 - val_loss: 1.2132 - val_accuracy: 0.6800 -
9s/epoch - 177ms/step
Epoch 14/30
50/50 - 8s - loss: 0.8318 - accuracy: 0.8169 - val_loss: 0.8171 - val_accuracy: 0.8250 -
8s/epoch - 167ms/step
Epoch 15/30
50/50 - 8s - loss: 0.8085 - accuracy: 0.8281 - val_loss: 1.0349 - val_accuracy: 0.7900 -
8s/epoch - 166ms/step
Epoch 16/30
50/50 - 8s - loss: 0.8025 - accuracy: 0.8338 - val_loss: 0.8055 - val_accuracy: 0.8475 -
8s/epoch - 164ms/step
Epoch 17/30
50/50 - 8s - loss: 0.7699 - accuracy: 0.8438 - val_loss: 0.9893 - val_accuracy: 0.6525 -
8s/epoch - 166ms/step
Epoch 18/30
50/50 - 8s - loss: 0.7433 - accuracy: 0.8569 - val_loss: 0.7809 - val_accuracy: 0.8250 -
8s/epoch - 167ms/step
Epoch 19/30
50/50 - 8s - loss: 0.7333 - accuracy: 0.8587 - val_loss: 0.7961 - val_accuracy: 0.8275 -
8s/epoch - 163ms/step
Epoch 20/30
50/50 - 9s - loss: 0.6896 - accuracy: 0.8731 - val_loss: 0.7426 - val_accuracy: 0.8500 -
9s/epoch - 180ms/step
Epoch 21/30
50/50 - 9s - loss: 0.6928 - accuracy: 0.8681 - val_loss: 0.6953 - val_accuracy: 0.8550 -
9s/epoch - 181ms/step
Epoch 22/30
50/50 - 10s - loss: 0.6710 - accuracy: 0.8731 - val_loss: 0.6360 - val_accuracy: 0.8825
- 10s/epoch - 190ms/step
Epoch 23/30
50/50 - 9s - loss: 0.6336 - accuracy: 0.8769 - val_loss: 0.6502 - val_accuracy: 0.8650 -
9s/epoch - 177ms/step
Epoch 24/30
50/50 - 8s - loss: 0.6054 - accuracy: 0.8906 - val_loss: 0.5960 - val_accuracy: 0.8800 -
8s/epoch - 163ms/step
Epoch 25/30
50/50 - 8s - loss: 0.5884 - accuracy: 0.8906 - val_loss: 0.6303 - val_accuracy: 0.8675 -
8s/epoch - 164ms/step
Epoch 26/30
50/50 - 15s - loss: 0.6028 - accuracy: 0.8850 - val_loss: 0.6267 - val_accuracy: 0.8600
- 15s/epoch - 293ms/step
Epoch 27/30
50/50 - 17s - loss: 0.6030 - accuracy: 0.8769 - val_loss: 0.6828 - val_accuracy: 0.8525
- 17s/epoch - 332ms/step
Epoch 28/30
50/50 - 17s - loss: 0.5852 - accuracy: 0.8838 - val_loss: 0.5617 - val_accuracy: 0.8800
- 17s/epoch - 334ms/step
Epoch 29/30
50/50 - 17s - loss: 0.5887 - accuracy: 0.8813 - val_loss: 0.5310 - val_accuracy: 0.8925
- 17s/epoch - 349ms/step
Epoch 30/30
50/50 - 17s - loss: 0.5488 - accuracy: 0.8856 - val_loss: 0.6332 - val_accuracy: 0.8400
- 17s/epoch - 332ms/step
```

In [24]:
```
plot_acc_loss(history_complex_64)
```





This model performed similarly to the model with the similar structure that used target_size = 128x128. The validation curves were especially also unpredictable. It was not the best performing model that used 64x64 target sizes.

Now that we have run a number of different models on a subset of the data, we have narrowed down some different model structures that have performed well on the subset. We will now test a few of the higher-performing models on the whole data set. We will leverage the entirety of the dataset to build our best model (all 27,560 images).

In the code below, we will split the data into three separate directories. 70% of the data will be allocated for training, 20% for validation, and 10% for testing.

In [25]:
```
import os
import shutil
import random

# Specify the original directories
original_dir = 'cell_images/cell_images'

# Specify the directories for the split
train_dir = 'cell_images/train'
val_dir = 'cell_images/validation'
test_dir = 'cell_images/test'

# Ensure the directories exist
```

```
for dir_path in [train_dir, val_dir, test_dir]:
    if not os.path.exists(dir_path):
        os.makedirs(os.path.join(dir_path, 'Parasitized'))
        os.makedirs(os.path.join(dir_path, 'Uninfected'))

# Specify the categories
categories = ['Parasitized', 'Uninfected']

# make sure directories are not overwritten if they already exist(for when I re-run)
def ensure_directory_contents(dir_path, category, files):
    category_dir = os.path.join(dir_path, category)
    if not os.listdir(category_dir):  # Check if the directory is empty
        for file in files:
            shutil.copy(os.path.join(original_dir, category, file), os.path.join(categor


# Splitting images into 70% training, 20% validation, 10% testing
for category in categories:
    # list files
    files = os.listdir(os.path.join(original_dir, category))

    # suffle files
    random.shuffle(files)

    # compute splits
    train_size = int(0.7 * len(files))
    val_size = int(0.2 * len(files))

    # split
    train_files = files[:train_size]
    val_files = files[train_size:train_size+val_size]
    test_files = files[train_size+val_size:]

    # Only copy files if directories are empty
    ensure_directory_contents(train_dir, category, train_files)
    ensure_directory_contents(val_dir, category, val_files)
    ensure_directory_contents(test_dir, category, test_files)
```

In [26]:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import random
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten, Dropout
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.regularizers import l2, l1
from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping, ModelCheckpoint
from tensorflow.keras.models import load_model
from sklearn.metrics import confusion_matrix, classification_report
from tabulate import tabulate
```

In [27]:

```
# new data generator function
def create_data_generators_wholeset(base_dir, target_size, batch_size=128, augmentation=

    # Define class mapping
    class_indices = {'Uninfected': 0, 'Parasitized': 1}

    if augmentation:  # Create data generator with augmentation for training
        train_datagen = ImageDataGenerator(
            rescale=1.0/255.0,
            rotation_range=30,
            width_shift_range=0.2,
```

```
            height_shift_range=0.2,
            shear_range=0.2,
            zoom_range=0.2,
            horizontal_flip=True,
            fill_mode='nearest'
        )
    else:  # Standard rescaling generator for training
        train_datagen = ImageDataGenerator(rescale=1.0/255.0)

    # For validation and test, always use the same datagen without augmentation
    test_val_datagen = ImageDataGenerator(rescale=1.0/255.0)

    # Training data
    train_generator = train_datagen.flow_from_directory(
        directory=f'{base_dir}/train',
        target_size=target_size,  # Taken from function args
        batch_size=batch_size,  # Taken from function args
        class_mode='binary',
        shuffle=True,  # Shuffle for training data
        classes=list(class_indices.keys())  # Use pre-defined class mapping
    )

    # Validation data
    validation_generator = test_val_datagen.flow_from_directory(
        directory=f'{base_dir}/validation',
        target_size=target_size,  # Taken from function args
        batch_size=batch_size,  # Taken from function args
        class_mode='binary',
        shuffle=False,  # No shuffling for validation data
        classes=list(class_indices.keys())  # Use pre-defined class mapping
    )

    # Test data
    test_generator = test_val_datagen.flow_from_directory(
        directory=f'{base_dir}/test',
        target_size=target_size,  # Taken from function args
        batch_size=batch_size,  # Taken from function args
        class_mode='binary',
        shuffle=False,  # No shuffling for test data
        classes=list(class_indices.keys())  # Use pre-defined class mapping
    )

    return train_generator, validation_generator, test_generator
```

We will select 4 models to re-train on the entirety of the dataset. When training with the entire data_set we will use larger batch_sizes (128 or 256 depending on the model).

MODEL 1 DETAILS Target Size: 64x64 Batch Size: 128 No Data Augmentation 2 Convolutional Layers 2 Dense Layers 25% Dropout between each layer L2 regularization = .001 for each layer (except output layer)

In [28]:
```
# Simple model with 64x64 targets initially, higher batch_size
train_generator, validation_generator, test_generator = create_data_generators_wholeset(
    base_dir='cell_images',
    target_size=(64,64),
    batch_size=128,
    augmentation=False
)

# Define the model with regularization and dropout
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3), kernel_regularizer=l2
    MaxPooling2D((2, 2)),
    Dropout(0.25),
```

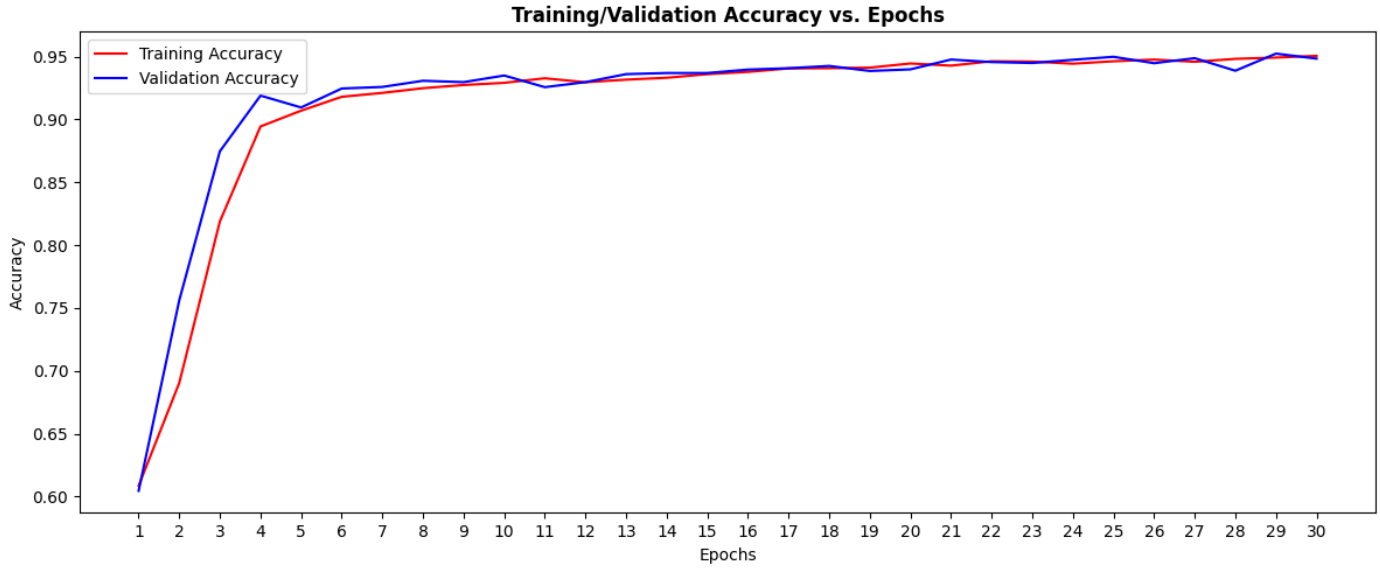
```
    Conv2D(64, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    MaxPooling2D((2, 2)),
    Dropout(0.25),
    Flatten(),
    Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
    Dropout(0.25),
    Dense(1, activation='sigmoid')
])

# Compiling the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

callbacks=[ModelCheckpoint('best_model_wholeset_v1.h5',
                           save_best_only=True,
                           monitor='val_accuracy')]

# Training the model
history_64_whole_set = model.fit(train_generator,
                                 validation_data=validation_generator,
                                 callbacks=callbacks,
                                 epochs=30,
                                 verbose=2)
```

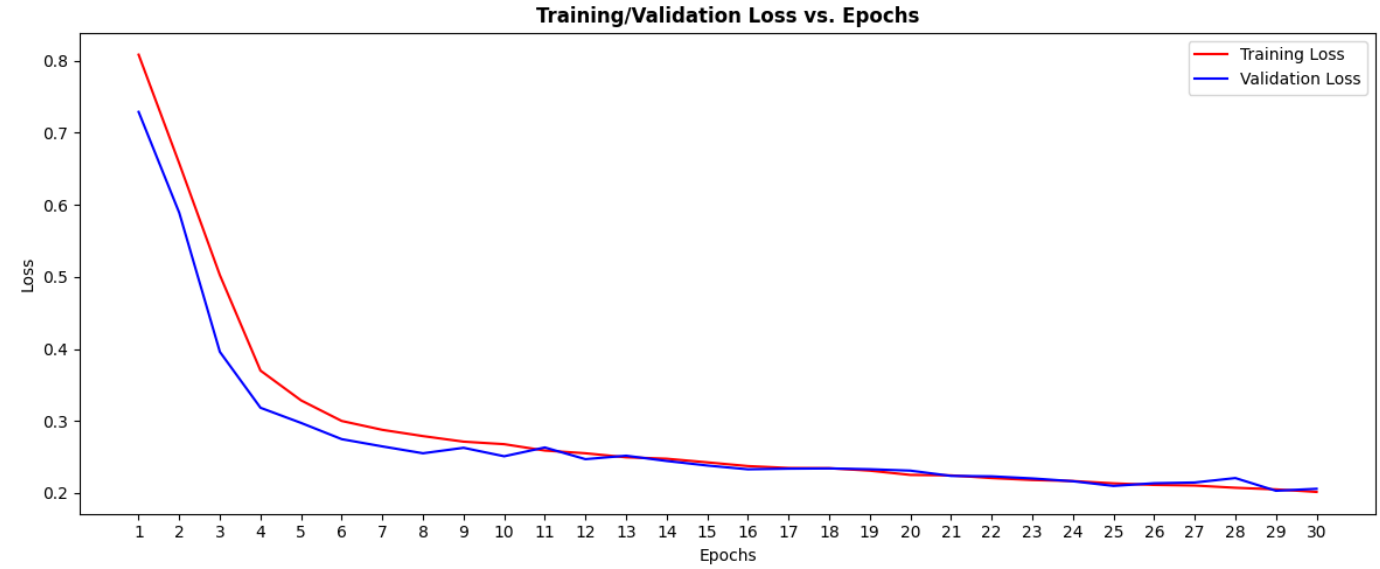
```
Found 19291 images belonging to 2 classes.
Found 5511 images belonging to 2 classes.
Found 2756 images belonging to 2 classes.
Epoch 1/30
151/151 - 81s - loss: 0.8084 - accuracy: 0.6086 - val_loss: 0.7290 - val_accuracy: 0.604
6 - 81s/epoch - 534ms/step
Epoch 2/30
151/151 - 49s - loss: 0.6575 - accuracy: 0.6904 - val_loss: 0.5890 - val_accuracy: 0.756
1 - 49s/epoch - 322ms/step
Epoch 3/30
151/151 - 49s - loss: 0.5025 - accuracy: 0.8188 - val_loss: 0.3960 - val_accuracy: 0.874
6 - 49s/epoch - 322ms/step
Epoch 4/30
151/151 - 53s - loss: 0.3700 - accuracy: 0.8944 - val_loss: 0.3184 - val_accuracy: 0.918
9 - 53s/epoch - 351ms/step
Epoch 5/30
151/151 - 59s - loss: 0.3286 - accuracy: 0.9069 - val_loss: 0.2973 - val_accuracy: 0.909
5 - 59s/epoch - 387ms/step
Epoch 6/30
151/151 - 68s - loss: 0.3001 - accuracy: 0.9179 - val_loss: 0.2748 - val_accuracy: 0.924
5 - 68s/epoch - 450ms/step
Epoch 7/30
151/151 - 65s - loss: 0.2878 - accuracy: 0.9211 - val_loss: 0.2648 - val_accuracy: 0.925
8 - 65s/epoch - 429ms/step
Epoch 8/30
151/151 - 65s - loss: 0.2791 - accuracy: 0.9248 - val_loss: 0.2552 - val_accuracy: 0.930
7 - 65s/epoch - 430ms/step
Epoch 9/30
151/151 - 66s - loss: 0.2714 - accuracy: 0.9273 - val_loss: 0.2628 - val_accuracy: 0.929
6 - 66s/epoch - 436ms/step
Epoch 10/30
151/151 - 66s - loss: 0.2679 - accuracy: 0.9290 - val_loss: 0.2511 - val_accuracy: 0.934
9 - 66s/epoch - 440ms/step
Epoch 11/30
151/151 - 61s - loss: 0.2590 - accuracy: 0.9327 - val_loss: 0.2633 - val_accuracy: 0.925
6 - 61s/epoch - 406ms/step
Epoch 12/30
151/151 - 59s - loss: 0.2552 - accuracy: 0.9295 - val_loss: 0.2471 - val_accuracy: 0.929
6 - 59s/epoch - 390ms/step
Epoch 13/30
151/151 - 58s - loss: 0.2497 - accuracy: 0.9316 - val_loss: 0.2518 - val_accuracy: 0.935
9 - 58s/epoch - 384ms/step
Epoch 14/30
```

```
151/151 - 61s - loss: 0.2477 - accuracy: 0.9331 - val_loss: 0.2446 - val_accuracy: 0.936
9 - 61s/epoch - 406ms/step
Epoch 15/30
151/151 - 58s - loss: 0.2426 - accuracy: 0.9359 - val_loss: 0.2382 - val_accuracy: 0.936
9 - 58s/epoch - 385ms/step
Epoch 16/30
151/151 - 64s - loss: 0.2375 - accuracy: 0.9377 - val_loss: 0.2330 - val_accuracy: 0.939
6 - 64s/epoch - 422ms/step
Epoch 17/30
151/151 - 59s - loss: 0.2348 - accuracy: 0.9405 - val_loss: 0.2339 - val_accuracy: 0.940
7 - 59s/epoch - 390ms/step
Epoch 18/30
151/151 - 59s - loss: 0.2345 - accuracy: 0.9407 - val_loss: 0.2342 - val_accuracy: 0.942
5 - 59s/epoch - 393ms/step
Epoch 19/30
151/151 - 60s - loss: 0.2311 - accuracy: 0.9411 - val_loss: 0.2331 - val_accuracy: 0.938
5 - 60s/epoch - 395ms/step
Epoch 20/30
151/151 - 58s - loss: 0.2252 - accuracy: 0.9444 - val_loss: 0.2311 - val_accuracy: 0.939
8 - 58s/epoch - 386ms/step
Epoch 21/30
151/151 - 61s - loss: 0.2245 - accuracy: 0.9427 - val_loss: 0.2238 - val_accuracy: 0.947
6 - 61s/epoch - 405ms/step
Epoch 22/30
151/151 - 60s - loss: 0.2208 - accuracy: 0.9462 - val_loss: 0.2232 - val_accuracy: 0.945
6 - 60s/epoch - 396ms/step
Epoch 23/30
151/151 - 59s - loss: 0.2181 - accuracy: 0.9459 - val_loss: 0.2203 - val_accuracy: 0.944
8 - 59s/epoch - 390ms/step
Epoch 24/30
151/151 - 59s - loss: 0.2167 - accuracy: 0.9443 - val_loss: 0.2165 - val_accuracy: 0.947
4 - 59s/epoch - 393ms/step
Epoch 25/30
151/151 - 59s - loss: 0.2135 - accuracy: 0.9462 - val_loss: 0.2099 - val_accuracy: 0.949
7 - 59s/epoch - 389ms/step
Epoch 26/30
151/151 - 60s - loss: 0.2113 - accuracy: 0.9475 - val_loss: 0.2137 - val_accuracy: 0.944
7 - 60s/epoch - 394ms/step
Epoch 27/30
151/151 - 58s - loss: 0.2105 - accuracy: 0.9458 - val_loss: 0.2147 - val_accuracy: 0.948
6 - 58s/epoch - 385ms/step
Epoch 28/30
151/151 - 59s - loss: 0.2074 - accuracy: 0.9481 - val_loss: 0.2208 - val_accuracy: 0.938
7 - 59s/epoch - 393ms/step
Epoch 29/30
151/151 - 58s - loss: 0.2051 - accuracy: 0.9491 - val_loss: 0.2032 - val_accuracy: 0.952
3 - 58s/epoch - 385ms/step
Epoch 30/30
151/151 - 59s - loss: 0.2016 - accuracy: 0.9505 - val_loss: 0.2059 - val_accuracy: 0.948
3 - 59s/epoch - 392ms/step
```

In [29]:

```
# display plot
plot_acc_loss(history_64_whole_set)
```





Now we will evaluate the model and make predictions with the model trained on the entire data set.

Initially when I did this step, the accuracy for the predictions was 50%, even though the test accuracy reported by my model.evaluate function was > 90% for each of the models. Originally, when I created my test generator, I did not specify shuffle = False. The indices of predicted labels that I was applying to the unseen test data did not match. This led to obtaining the same accuracy that would result for random predictions.

After specifying shuffle = False in the create_data_generators_wholeset() function, the accuracy that resulted from the predictions aligned with that of the model.evaluate() function.

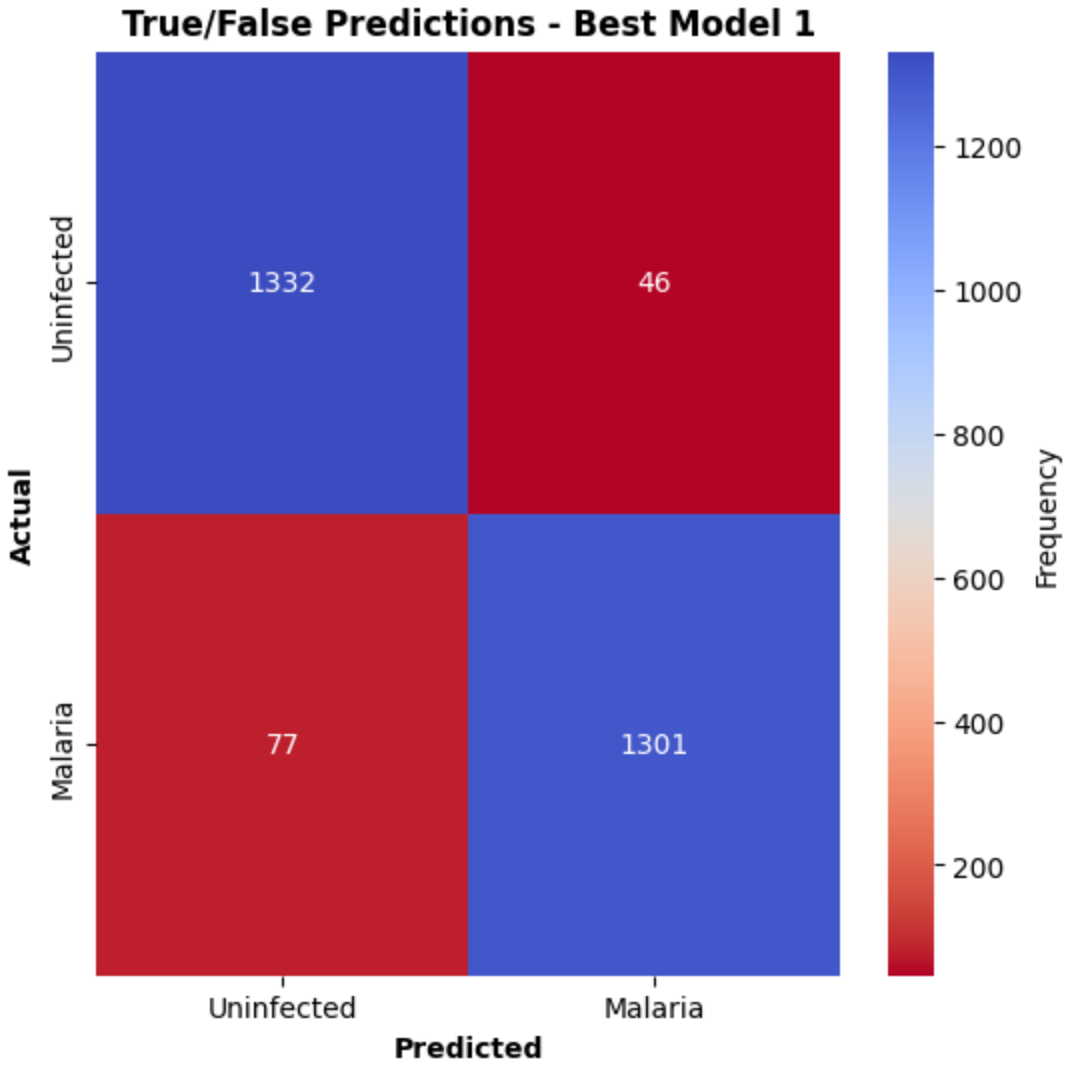
In [30]:
```
from tensorflow.keras.models import load_model
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
from tabulate import tabulate

# Load the best model
best_model_path1 = 'best_model_wholeset_v1.h5'
best_model1 = load_model(best_model_path1)

# Evaluate the model on the test data
test_loss1, test_accuracy1 = best_model1.evaluate(test_generator, verbose=0)
print(f"Test Loss: {test_loss1:.4f}, Test Accuracy: {test_accuracy1:.4f}")

# Make predictions
```

```
predictions1 = best_model1.predict(test_generator)
predicted_labels1 = np.round(predictions1).flatten()

# True labels
true_labels1 = test_generator.classes

# check how classes are labeled, before changing 0s and 1s labels
class_indices = test_generator.class_indices
print(f"\nClass Indices: {class_indices}\n")

# Confusion matrix and classification report
cm1 = confusion_matrix(true_labels1, predicted_labels1)
print(f"\nConfusion Matrix:\n{cm1}")

report1 = classification_report(true_labels1, predicted_labels1,
                                target_names=['Uninfected', 'Malaria'])
print("\nClassification Report:\n", report1)
```

```
Test Loss: 0.1789, Test Accuracy: 0.9554
22/22 [==============================] - 4s 154ms/step

Class Indices: {'Uninfected': 0, 'Parasitized': 1}


Confusion Matrix:
[[1332   46]
 [  77 1301]]

Classification Report:
              precision    recall  f1-score   support

  Uninfected       0.95      0.97      0.96      1378
     Malaria       0.97      0.94      0.95      1378

    accuracy                           0.96      2756
   macro avg       0.96      0.96      0.96      2756
weighted avg       0.96      0.96      0.96      2756
```

In [31]:

```
# Plot confusion matrix using seaborn to see labels
plt.figure(figsize=(6, 6))
sns.heatmap(cm1, annot=True, fmt='d', cmap='coolwarm_r',
            xticklabels=['Uninfected', 'Malaria'], yticklabels=['Uninfected', 'Malaria']
            cbar_kws={'label': 'Frequency'})
plt.ylabel('Actual', fontweight='bold')
plt.xlabel('Predicted', fontweight='bold')
plt.title('True/False Predictions - Best Model 1', fontweight='bold')
plt.show()
```



MODEL 2 DETAILS Target Size: 128x128 Batch Size: 128 No Data Augmentation 2 Convolutional Layers 2 Dense Layers 25% Dropout after each convolutional layer, 50% Dropout before output layer L2 regularization = .001 between each layer

In [32]:

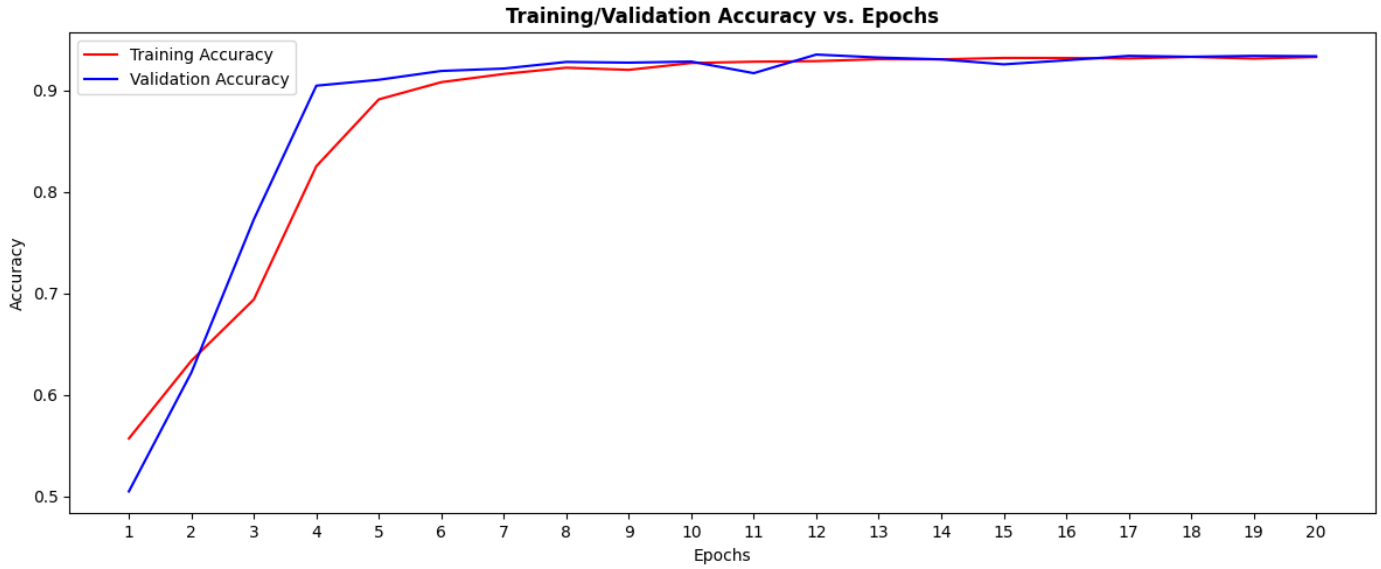
```
# with 128x128 targets, dropout, l2 regularization, higher batch_size
train_generator, validation_generator, test_generator = create_data_generators_wholeset(
    base_dir='cell_images',
    target_size=(128,128),
    batch_size=128,
    augmentation=False
)

# Define the model with regularization and dropout that performed well on subset
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3), kernel_regularizer=
    MaxPooling2D((2, 2)),
    Dropout(0.25),
    Conv2D(64, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    MaxPooling2D((2, 2)),
    Dropout(0.25),
    Flatten(),
    Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

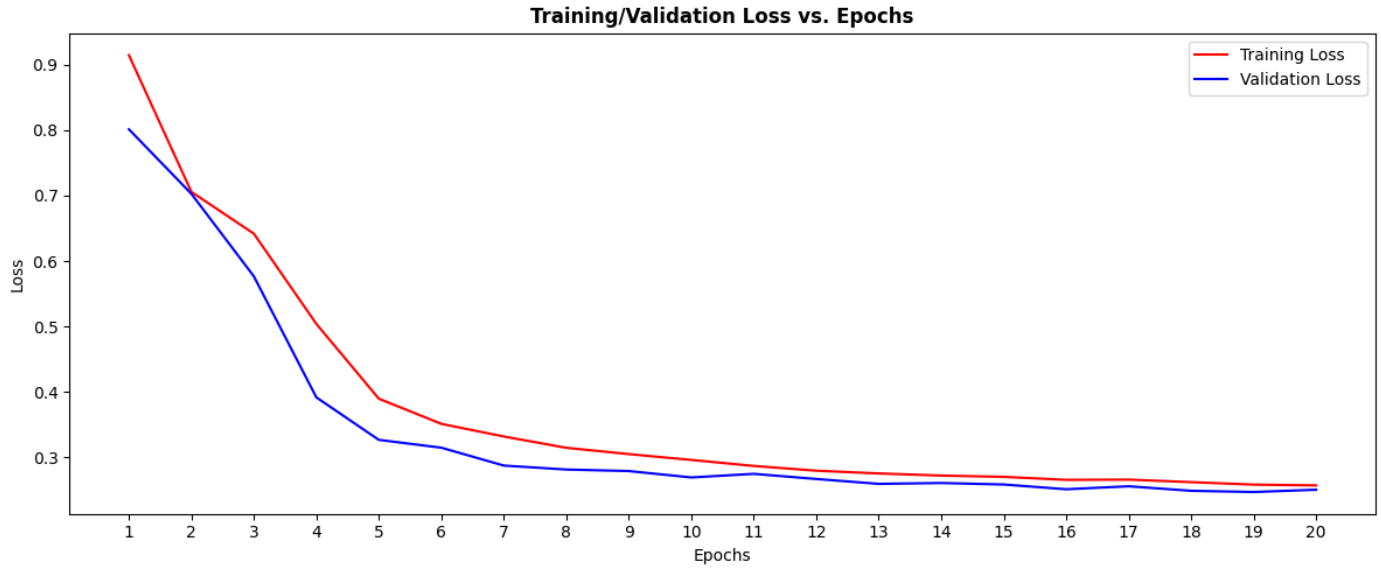
```
callbacks=[
    EarlyStopping(monitor='val_accuracy',
                  patience=8,
                  restore_best_weights=True),
    ModelCheckpoint('best_model_wholeset_v2.h5',
                    save_best_only=True,
                    monitor='val_accuracy')
]

history_128_l2_dropout_wholeset = model.fit(train_generator,
                                            validation_data=validation_generator,
                                            callbacks=callbacks,
                                            epochs=30,
                                            verbose=2)
```

```
Found 19291 images belonging to 2 classes.
Found 5511 images belonging to 2 classes.
Found 2756 images belonging to 2 classes.
Epoch 1/30
151/151 - 187s - loss: 0.9147 - accuracy: 0.5569 - val_loss: 0.8014 - val_accuracy: 0.50
48 - 187s/epoch - 1s/step
C:\Users\gdgun\COMP4531\my_tensorflow_project\venv\lib\site-packages\keras\src\engine\tr
aining.py:3000: UserWarning: You are saving your model as an HDF5 file via `model.save()
`. This file format is considered legacy. We recommend using instead the native Keras fo
rmat, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
Epoch 2/30
151/151 - 186s - loss: 0.7058 - accuracy: 0.6336 - val_loss: 0.7028 - val_accuracy: 0.62
20 - 186s/epoch - 1s/step
Epoch 3/30
151/151 - 186s - loss: 0.6419 - accuracy: 0.6940 - val_loss: 0.5767 - val_accuracy: 0.77
30 - 186s/epoch - 1s/step
Epoch 4/30
151/151 - 186s - loss: 0.5041 - accuracy: 0.8254 - val_loss: 0.3919 - val_accuracy: 0.90
47 - 186s/epoch - 1s/step
Epoch 5/30
151/151 - 186s - loss: 0.3898 - accuracy: 0.8912 - val_loss: 0.3269 - val_accuracy: 0.91
05 - 186s/epoch - 1s/step
Epoch 6/30
151/151 - 185s - loss: 0.3514 - accuracy: 0.9082 - val_loss: 0.3150 - val_accuracy: 0.91
93 - 185s/epoch - 1s/step
Epoch 7/30
151/151 - 185s - loss: 0.3321 - accuracy: 0.9163 - val_loss: 0.2876 - val_accuracy: 0.92
16 - 185s/epoch - 1s/step
Epoch 8/30
151/151 - 184s - loss: 0.3148 - accuracy: 0.9225 - val_loss: 0.2816 - val_accuracy: 0.92
81 - 184s/epoch - 1s/step
Epoch 9/30
151/151 - 183s - loss: 0.3052 - accuracy: 0.9203 - val_loss: 0.2793 - val_accuracy: 0.92
74 - 183s/epoch - 1s/step
Epoch 10/30
151/151 - 183s - loss: 0.2963 - accuracy: 0.9271 - val_loss: 0.2695 - val_accuracy: 0.92
85 - 183s/epoch - 1s/step
Epoch 11/30
151/151 - 184s - loss: 0.2871 - accuracy: 0.9284 - val_loss: 0.2750 - val_accuracy: 0.91
71 - 184s/epoch - 1s/step
Epoch 12/30
151/151 - 184s - loss: 0.2798 - accuracy: 0.9289 - val_loss: 0.2672 - val_accuracy: 0.93
54 - 184s/epoch - 1s/step
Epoch 13/30
151/151 - 183s - loss: 0.2756 - accuracy: 0.9308 - val_loss: 0.2597 - val_accuracy: 0.93
25 - 183s/epoch - 1s/step
Epoch 14/30
151/151 - 198s - loss: 0.2723 - accuracy: 0.9307 - val_loss: 0.2610 - val_accuracy: 0.93
07 - 198s/epoch - 1s/step
```

```
Epoch 15/30
151/151 - 185s - loss: 0.2704 - accuracy: 0.9321 - val_loss: 0.2588 - val_accuracy: 0.92
58 - 185s/epoch - 1s/step
Epoch 16/30
151/151 - 184s - loss: 0.2659 - accuracy: 0.9320 - val_loss: 0.2515 - val_accuracy: 0.92
98 - 184s/epoch - 1s/step
Epoch 17/30
151/151 - 184s - loss: 0.2662 - accuracy: 0.9316 - val_loss: 0.2560 - val_accuracy: 0.93
41 - 184s/epoch - 1s/step
Epoch 18/30
151/151 - 184s - loss: 0.2623 - accuracy: 0.9331 - val_loss: 0.2491 - val_accuracy: 0.93
32 - 184s/epoch - 1s/step
Epoch 19/30
151/151 - 184s - loss: 0.2585 - accuracy: 0.9314 - val_loss: 0.2473 - val_accuracy: 0.93
41 - 184s/epoch - 1s/step
Epoch 20/30
151/151 - 184s - loss: 0.2573 - accuracy: 0.9330 - val_loss: 0.2507 - val_accuracy: 0.93
38 - 184s/epoch - 1s/step
```

In [33]:
```
# display plot
plot_acc_loss(history_128_l2_dropout_wholeset)
```





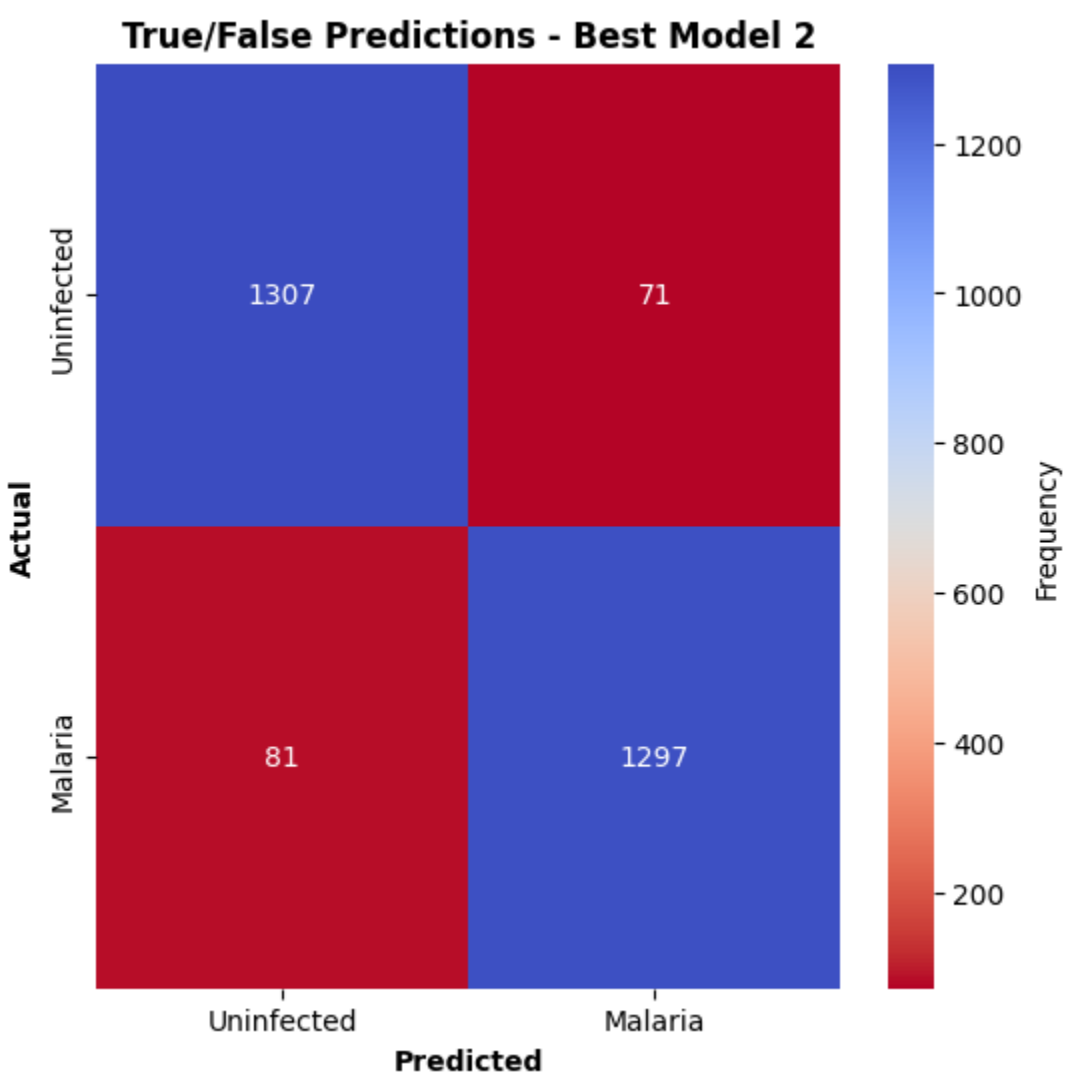
In [34]:
```
# Load the best model
best_model_path2 = 'best_model_wholeset_v2.h5'
best_model2 = load_model(best_model_path2)

# Evaluate the model on the test data
test_loss2, test_accuracy2 = best_model2.evaluate(test_generator, verbose=0)
```

```
print(f"Test Loss: {test_loss2:.4f}, Test Accuracy: {test_accuracy2:.4f}")

# Make predictions
predictions2 = best_model2.predict(test_generator)
predicted_labels2 = np.round(predictions2).flatten()

# True labels
true_labels2 = test_generator.classes

# check how classes are labeled, before changing 0s and 1s labels
class_indices = test_generator.class_indices
print(f"\nClass Indices: {class_indices}\n")

# Confusion matrix and classification report
cm2 = confusion_matrix(true_labels2, predicted_labels2)
print(f"\nConfusion Matrix:\n{cm2}")

report2 = classification_report(true_labels2, predicted_labels2,
                                target_names=['Uninfected', 'Malaria'])
print("\nClassification Report:\n", report2)
```

```
Test Loss: 0.2313, Test Accuracy: 0.9448
22/22 [==============================] - 7s 317ms/step

Class Indices: {'Uninfected': 0, 'Parasitized': 1}


Confusion Matrix:
[[1307   71]
 [  81 1297]]

Classification Report:
              precision    recall  f1-score   support

  Uninfected       0.94      0.95      0.95      1378
     Malaria       0.95      0.94      0.94      1378

    accuracy                           0.94      2756
   macro avg       0.94      0.94      0.94      2756
weighted avg       0.94      0.94      0.94      2756
```

In [35]:

```
# Plot confusion matrix using seaborn to see labels
plt.figure(figsize=(6, 6))
sns.heatmap(cm2, annot=True, fmt='d', cmap='coolwarm_r',
            xticklabels=['Uninfected', 'Malaria'],
            yticklabels=['Uninfected', 'Malaria'],
            cbar_kws={'label': 'Frequency'})
plt.ylabel('Actual', fontweight='bold')
plt.xlabel('Predicted', fontweight='bold')
plt.title('True/False Predictions - Best Model 2', fontweight='bold')
plt.show()
```



MODEL 3 DETAILS Target Size: 128x128 Batch Size: 256 Data Augmentation used 3 Convolutional Layers 3 Dense Layers 50% Dropout between each dense layer

In [36]:

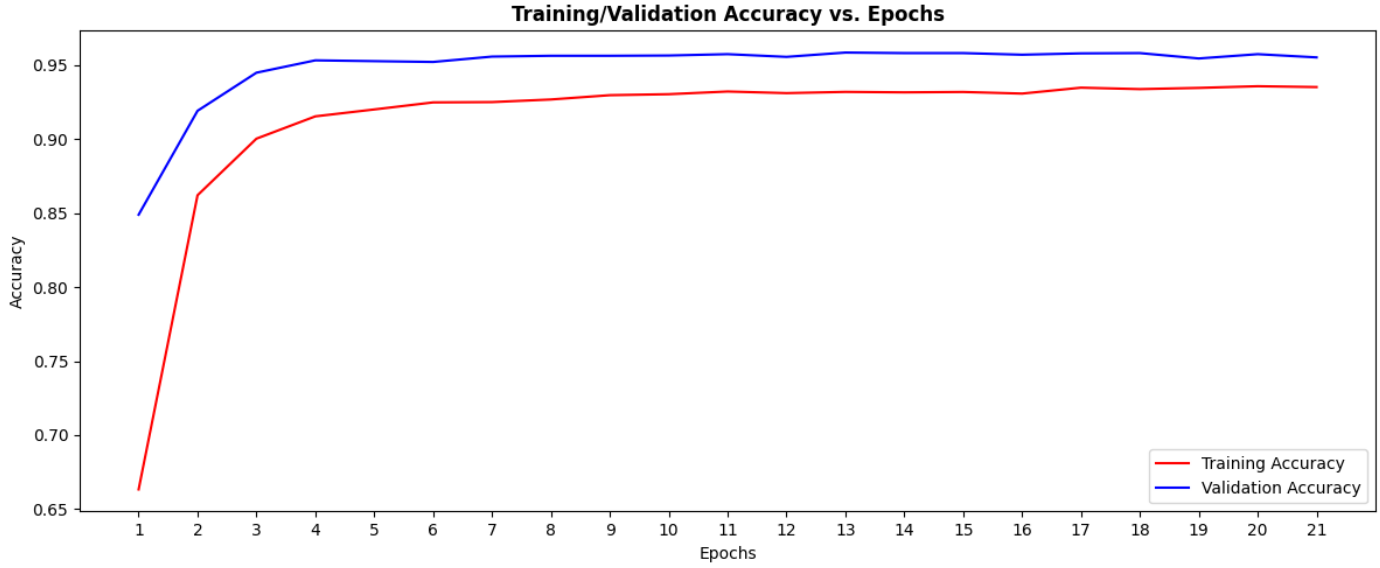
```
# more complex model, with augmentation, batch_size of 256
train_generator, validation_generator, test_generator = create_data_generators_wholeset(
    base_dir='cell_images',
    target_size=(128,128),
    batch_size=256,
    augmentation=True)

# Define the CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])



# Compile the model
```

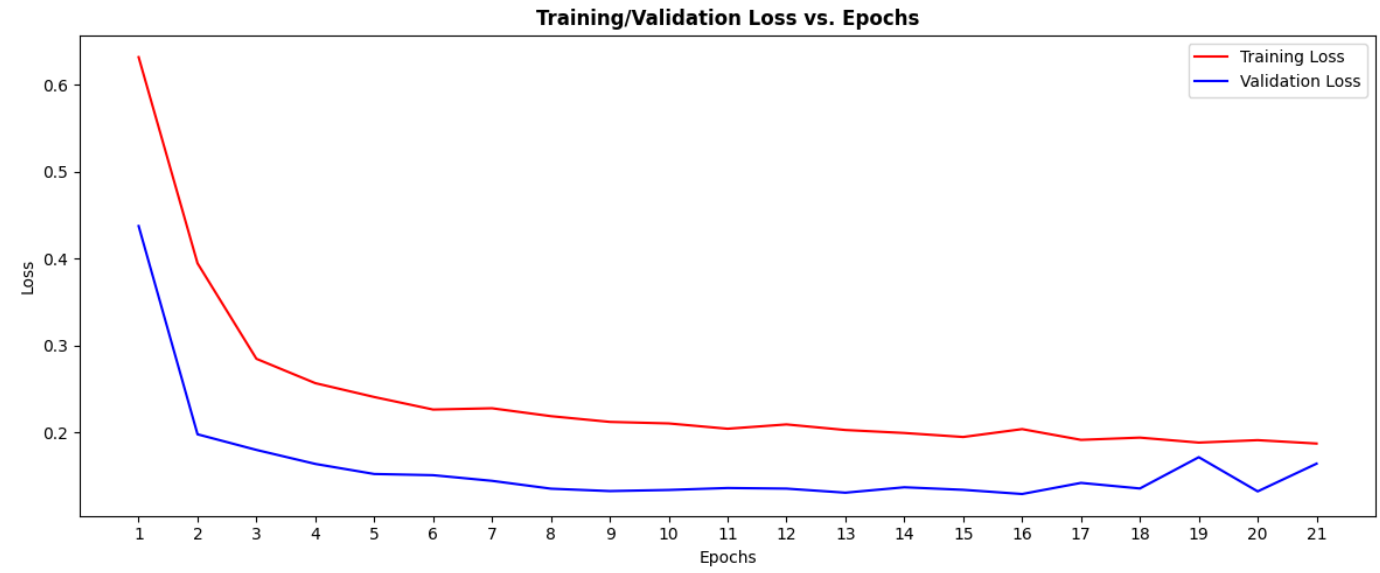
```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])


callbacks=[
    EarlyStopping(monitor='val_accuracy',
                  patience=8,
                  restore_best_weights=True),
    ModelCheckpoint('best_model_wholeset_v3.h5',
                    save_best_only=True,
                    monitor='val_accuracy')
]

# Train the model with augmented data
history_complex_wholeset = model.fit(train_generator,
                            validation_data=validation_generator,
                            callbacks=callbacks,
                            epochs=30,
                            verbose=2)
```

```
Found 19291 images belonging to 2 classes.
Found 5511 images belonging to 2 classes.
Found 2756 images belonging to 2 classes.
Epoch 1/30
76/76 - 232s - loss: 0.6315 - accuracy: 0.6633 - val_loss: 0.4373 - val_accuracy: 0.8490
- 232s/epoch - 3s/step
Epoch 2/30
C:\Users\gdgun\COMP4531\my_tensorflow_project\venv\lib\site-packages\keras\src\engine\tr
aining.py:3000: UserWarning: You are saving your model as an HDF5 file via `model.save()
`. This file format is considered legacy. We recommend using instead the native Keras fo
rmat, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
76/76 - 228s - loss: 0.3944 - accuracy: 0.8622 - val_loss: 0.1977 - val_accuracy: 0.9193
- 228s/epoch - 3s/step
Epoch 3/30
76/76 - 233s - loss: 0.2846 - accuracy: 0.9004 - val_loss: 0.1798 - val_accuracy: 0.9450
- 233s/epoch - 3s/step
Epoch 4/30
76/76 - 231s - loss: 0.2566 - accuracy: 0.9155 - val_loss: 0.1637 - val_accuracy: 0.9534
- 231s/epoch - 3s/step
Epoch 5/30
76/76 - 228s - loss: 0.2406 - accuracy: 0.9202 - val_loss: 0.1521 - val_accuracy: 0.9528
- 228s/epoch - 3s/step
Epoch 6/30
76/76 - 227s - loss: 0.2262 - accuracy: 0.9249 - val_loss: 0.1507 - val_accuracy: 0.9523
- 227s/epoch - 3s/step
Epoch 7/30
76/76 - 227s - loss: 0.2277 - accuracy: 0.9251 - val_loss: 0.1442 - val_accuracy: 0.9559
- 227s/epoch - 3s/step
Epoch 8/30
76/76 - 229s - loss: 0.2186 - accuracy: 0.9269 - val_loss: 0.1352 - val_accuracy: 0.9565
- 229s/epoch - 3s/step
Epoch 9/30
76/76 - 229s - loss: 0.2120 - accuracy: 0.9298 - val_loss: 0.1324 - val_accuracy: 0.9565
- 229s/epoch - 3s/step
Epoch 10/30
76/76 - 226s - loss: 0.2102 - accuracy: 0.9305 - val_loss: 0.1337 - val_accuracy: 0.9566
- 226s/epoch - 3s/step
Epoch 11/30
76/76 - 231s - loss: 0.2041 - accuracy: 0.9323 - val_loss: 0.1360 - val_accuracy: 0.9575
- 231s/epoch - 3s/step
Epoch 12/30
76/76 - 226s - loss: 0.2091 - accuracy: 0.9312 - val_loss: 0.1353 - val_accuracy: 0.9557
- 226s/epoch - 3s/step
Epoch 13/30
76/76 - 227s - loss: 0.2027 - accuracy: 0.9320 - val_loss: 0.1306 - val_accuracy: 0.9586
```

```
- 227s/epoch - 3s/step
Epoch 14/30
76/76 - 226s - loss: 0.1992 - accuracy: 0.9317 - val_loss: 0.1368 - val_accuracy: 0.9583
- 226s/epoch - 3s/step
Epoch 15/30
76/76 - 227s - loss: 0.1947 - accuracy: 0.9320 - val_loss: 0.1339 - val_accuracy: 0.9583
- 227s/epoch - 3s/step
Epoch 16/30
76/76 - 226s - loss: 0.2037 - accuracy: 0.9309 - val_loss: 0.1291 - val_accuracy: 0.9572
- 226s/epoch - 3s/step
Epoch 17/30
76/76 - 226s - loss: 0.1914 - accuracy: 0.9349 - val_loss: 0.1418 - val_accuracy: 0.9581
- 226s/epoch - 3s/step
Epoch 18/30
76/76 - 226s - loss: 0.1939 - accuracy: 0.9339 - val_loss: 0.1354 - val_accuracy: 0.9583
- 226s/epoch - 3s/step
Epoch 19/30
76/76 - 227s - loss: 0.1882 - accuracy: 0.9347 - val_loss: 0.1714 - val_accuracy: 0.9546
- 227s/epoch - 3s/step
Epoch 20/30
76/76 - 226s - loss: 0.1910 - accuracy: 0.9359 - val_loss: 0.1321 - val_accuracy: 0.9575
- 226s/epoch - 3s/step
Epoch 21/30
76/76 - 227s - loss: 0.1871 - accuracy: 0.9353 - val_loss: 0.1640 - val_accuracy: 0.9554
- 227s/epoch - 3s/step
```

In [37]:
```
# display plot
plot_acc_loss(history_complex_wholeset)
```

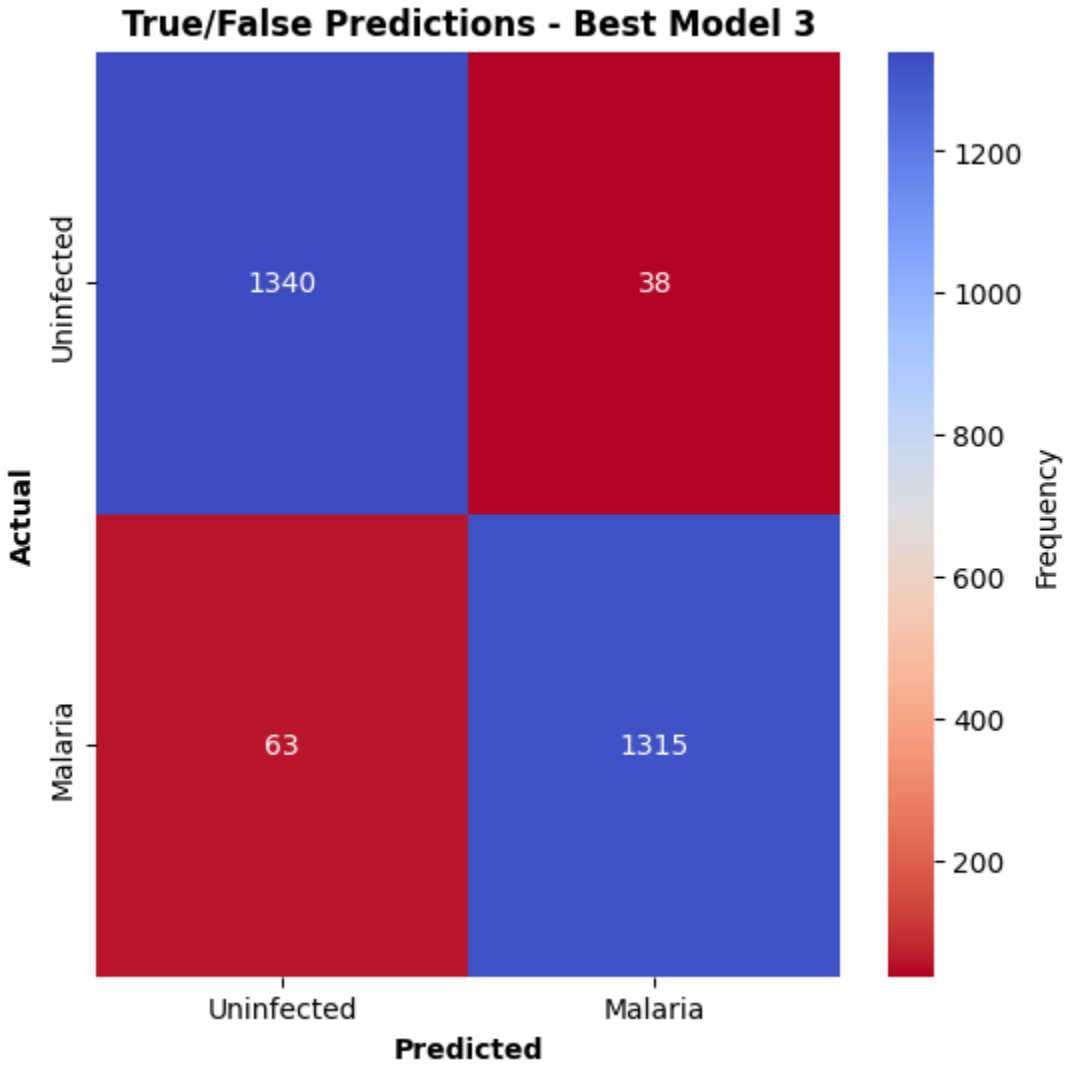




In [38]:
```
# Load the best model
best_model_path3 = 'best_model_wholeset_v3.h5'
best_model3 = load_model(best_model_path3)

# Evaluate the model on the test data
test_loss3, test_accuracy3 = best_model3.evaluate(test_generator, verbose=0)
print(f"Test Loss: {test_loss3:.4f}, Test Accuracy: {test_accuracy3:.4f}")

# Make predictions
predictions3 = best_model3.predict(test_generator)
predicted_labels3 = np.round(predictions3).flatten()

# True labels
true_labels3 = test_generator.classes

# check how classes are labeled, before changing 0s and 1s labels
class_indices = test_generator.class_indices
print(f"\nClass Indices: {class_indices}\n")

# Confusion matrix and classification report
cm3 = confusion_matrix(true_labels3, predicted_labels3)
print(f"\nConfusion Matrix:\n{cm3}")

report3 = classification_report(true_labels3, predicted_labels3,
                                target_names=['Uninfected', 'Malaria'])
print("\nClassification Report:\n", report3)
```

```
Test Loss: 0.1038, Test Accuracy: 0.9634
11/11 [==============================] - 7s 649ms/step

Class Indices: {'Uninfected': 0, 'Parasitized': 1}


Confusion Matrix:
[[1340   38]
 [  63 1315]]

Classification Report:
              precision    recall  f1-score   support

  Uninfected       0.96      0.97      0.96      1378
     Malaria       0.97      0.95      0.96      1378

    accuracy                           0.96      2756
   macro avg       0.96      0.96      0.96      2756
weighted avg       0.96      0.96      0.96      2756
```

In [39]:
```
# Plot confusion matrix using seaborn to see labels
plt.figure(figsize=(6, 6))
sns.heatmap(cm3, annot=True, fmt='d', cmap='coolwarm_r',
            xticklabels=['Uninfected', 'Malaria'],
            yticklabels=['Uninfected', 'Malaria'],
            cbar_kws={'label': 'Frequency'})
plt.ylabel('Actual', fontweight='bold')
plt.xlabel('Predicted', fontweight='bold')
plt.title('True/False Predictions - Best Model 3', fontweight='bold')
plt.show()
```



MODEL 4 DETAILS Target Size: 128x128 Batch Size: 256 Data Augmentation used 3 Convolutional Layers 3 Dense Layers 25% Dropout after each convolutional layer, 50% Dropout between Dense layers L2 regularization = .001 for each layer (except output layer) Batch Normalization used on each layer

In [40]:

```
# adds increased dropout, L2 regularization, and batchnormalization to previous model
train_generator, validation_generator, test_generator = create_data_generators_wholeset(
    base_dir='cell_images',
    target_size=(128,128),
    batch_size=256,
    augmentation=True)

model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3), kernel_regularizer=
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
```

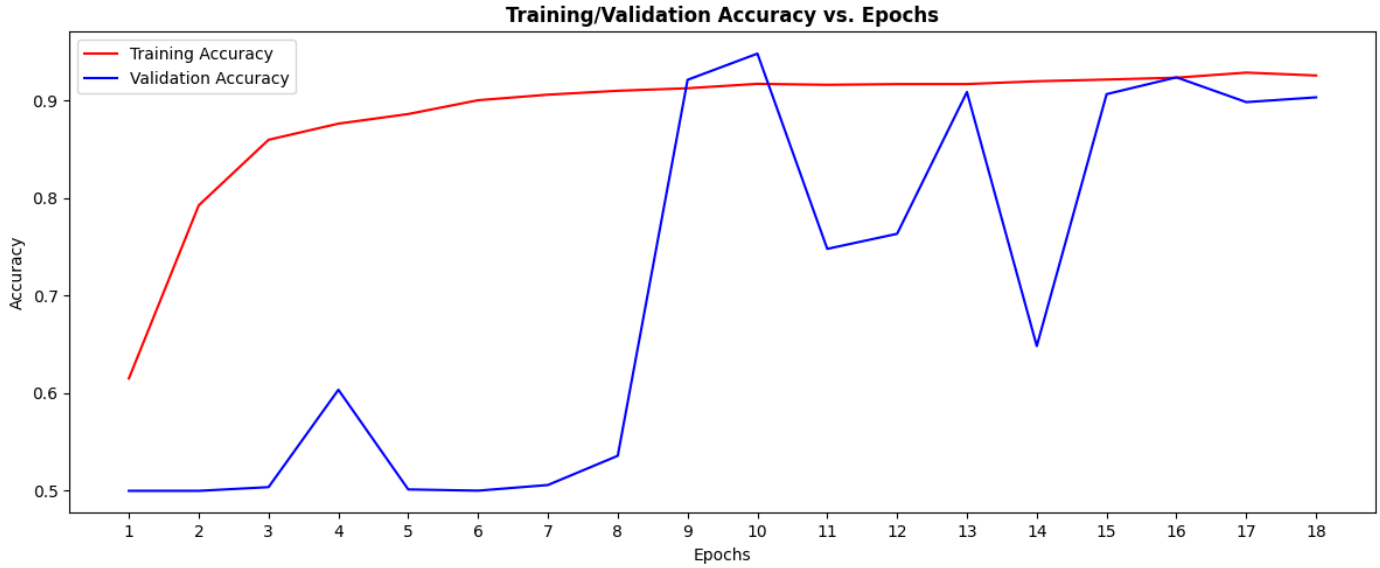
```
    BatchNormalization(),
    Dropout(0.5),

    Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.5),

    Dense(1, activation='sigmoid')
])


# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

callbacks=[
     EarlyStopping(monitor='val_accuracy',
                   patience=8,
                   restore_best_weights=True),
     ModelCheckpoint('best_model_wholeset_v4.h5',
                     save_best_only=True,
                     monitor='val_accuracy')
]

# Train the model with augmented data
history_complex_bn_wholeset = model.fit(train_generator,
                              validation_data=validation_generator,
                              callbacks=callbacks,
                              epochs=30,
                              verbose=2)
```

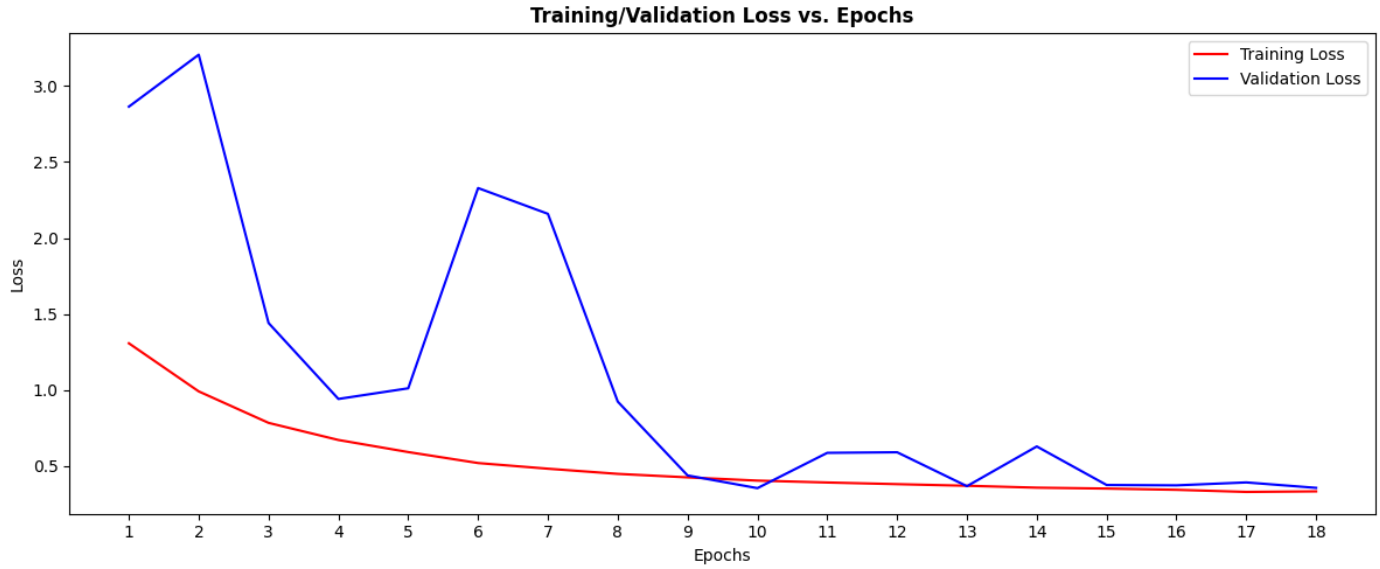
```
Found 19291 images belonging to 2 classes.
Found 5511 images belonging to 2 classes.
Found 2756 images belonging to 2 classes.
Epoch 1/30
76/76 - 324s - loss: 1.3080 - accuracy: 0.6152 - val_loss: 2.8636 - val_accuracy: 0.4999
- 324s/epoch - 4s/step
Epoch 2/30
C:\Users\gdgun\COMP4531\my_tensorflow_project\venv\lib\site-packages\keras\src\engine\tr
aining.py:3000: UserWarning: You are saving your model as an HDF5 file via `model.save()
`. This file format is considered legacy. We recommend using instead the native Keras fo
rmat, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
76/76 - 321s - loss: 0.9919 - accuracy: 0.7925 - val_loss: 3.2051 - val_accuracy: 0.4999
- 321s/epoch - 4s/step
Epoch 3/30
76/76 - 354s - loss: 0.7850 - accuracy: 0.8598 - val_loss: 1.4415 - val_accuracy: 0.5037
- 354s/epoch - 5s/step
Epoch 4/30
76/76 - 336s - loss: 0.6725 - accuracy: 0.8763 - val_loss: 0.9422 - val_accuracy: 0.6035
- 336s/epoch - 4s/step
Epoch 5/30
76/76 - 320s - loss: 0.5931 - accuracy: 0.8862 - val_loss: 1.0122 - val_accuracy: 0.5014
- 320s/epoch - 4s/step
Epoch 6/30
76/76 - 319s - loss: 0.5209 - accuracy: 0.9003 - val_loss: 2.3285 - val_accuracy: 0.5001
- 319s/epoch - 4s/step
Epoch 7/30
76/76 - 320s - loss: 0.4836 - accuracy: 0.9060 - val_loss: 2.1591 - val_accuracy: 0.5059
- 320s/epoch - 4s/step
Epoch 8/30
76/76 - 324s - loss: 0.4497 - accuracy: 0.9100 - val_loss: 0.9248 - val_accuracy: 0.5358
- 324s/epoch - 4s/step
Epoch 9/30
76/76 - 322s - loss: 0.4266 - accuracy: 0.9126 - val_loss: 0.4394 - val_accuracy: 0.9212
- 322s/epoch - 4s/step
```

```
Epoch 10/30
76/76 - 320s - loss: 0.4054 - accuracy: 0.9171 - val_loss: 0.3559 - val_accuracy: 0.9481
- 320s/epoch - 4s/step
Epoch 11/30
76/76 - 319s - loss: 0.3930 - accuracy: 0.9161 - val_loss: 0.5880 - val_accuracy: 0.7480
- 319s/epoch - 4s/step
Epoch 12/30
76/76 - 320s - loss: 0.3821 - accuracy: 0.9169 - val_loss: 0.5918 - val_accuracy: 0.7634
- 320s/epoch - 4s/step
Epoch 13/30
76/76 - 321s - loss: 0.3716 - accuracy: 0.9169 - val_loss: 0.3688 - val_accuracy: 0.9087
- 321s/epoch - 4s/step
Epoch 14/30
76/76 - 322s - loss: 0.3590 - accuracy: 0.9198 - val_loss: 0.6300 - val_accuracy: 0.6483
- 322s/epoch - 4s/step
Epoch 15/30
76/76 - 324s - loss: 0.3531 - accuracy: 0.9215 - val_loss: 0.3766 - val_accuracy: 0.9066
- 324s/epoch - 4s/step
Epoch 16/30
76/76 - 325s - loss: 0.3450 - accuracy: 0.9234 - val_loss: 0.3743 - val_accuracy: 0.9238
- 325s/epoch - 4s/step
Epoch 17/30
76/76 - 320s - loss: 0.3310 - accuracy: 0.9286 - val_loss: 0.3935 - val_accuracy: 0.8984
- 320s/epoch - 4s/step
Epoch 18/30
76/76 - 337s - loss: 0.3347 - accuracy: 0.9257 - val_loss: 0.3578 - val_accuracy: 0.9033
- 337s/epoch - 4s/step
```

In [41]:

```
# display plot
plot_acc_loss(history_complex_bn_wholeset)
```





In [42]:

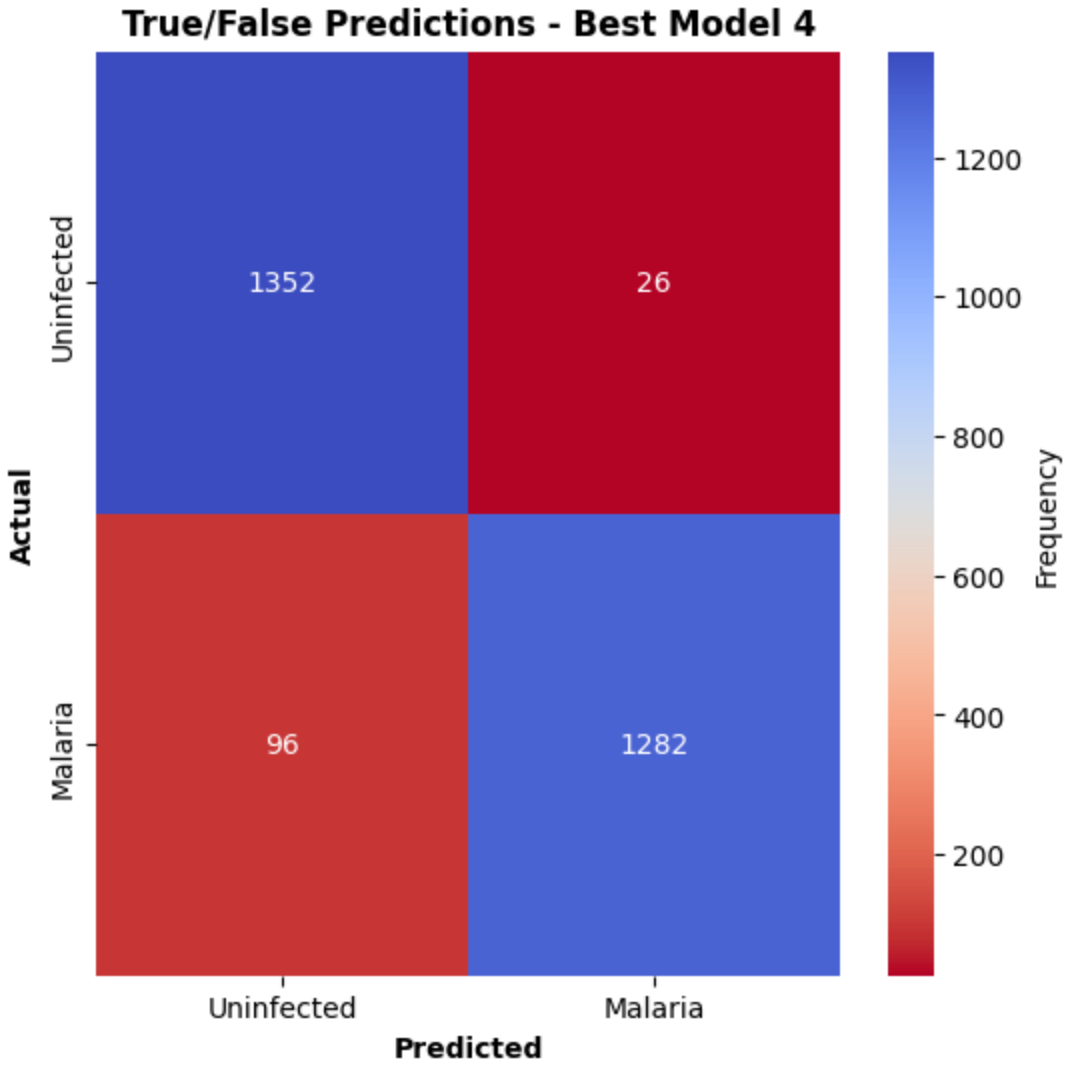
```
# Load the best model
best_model_path4 = 'best_model_wholeset_v4.h5'
best_model4 = load_model(best_model_path4)

# Evaluate the model on the test data
test_loss4, test_accuracy4 = best_model4.evaluate(test_generator, verbose=0)
print(f"Test Loss: {test_loss4:.4f}, Test Accuracy: {test_accuracy4:.4f}")

# Make predictions
predictions4 = best_model4.predict(test_generator)
predicted_labels4 = np.round(predictions4).flatten()

# True labels
true_labels4 = test_generator.classes

# check how classes are labeled, before changing 0s and 1s labels
class_indices = test_generator.class_indices
print(f"\nClass Indices: {class_indices}\n")

# Confusion matrix and classification report
cm4 = confusion_matrix(true_labels4, predicted_labels4)
print(f"\nConfusion Matrix:\n{cm4}")

report4 = classification_report(true_labels4, predicted_labels4,
                                target_names=['Uninfected', 'Malaria'])
print("\nClassification Report:\n", report4)
```

```
Test Loss: 0.3428, Test Accuracy: 0.9557
```

```
11/11 [==============================] - 10s 890ms/step

Class Indices: {'Uninfected': 0, 'Parasitized': 1}


Confusion Matrix:
[[1352   26]
 [  96 1282]]

Classification Report:
              precision    recall  f1-score   support

  Uninfected       0.93      0.98      0.96      1378
     Malaria       0.98      0.93      0.95      1378

    accuracy                           0.96      2756
   macro avg       0.96      0.96      0.96      2756
weighted avg       0.96      0.96      0.96      2756
```

In [43]:

```python
# Plot confusion matrix using seaborn to see labels
plt.figure(figsize=(6, 6))
sns.heatmap(cm4, annot=True, fmt='d', cmap='coolwarm_r',
            xticklabels=['Uninfected', 'Malaria'],
            yticklabels=['Uninfected', 'Malaria'],
            cbar_kws={'label': 'Frequency'})
plt.ylabel('Actual', fontweight='bold')
plt.xlabel('Predicted', fontweight='bold')
plt.title('True/False Predictions - Best Model 4', fontweight='bold')
plt.show()
```

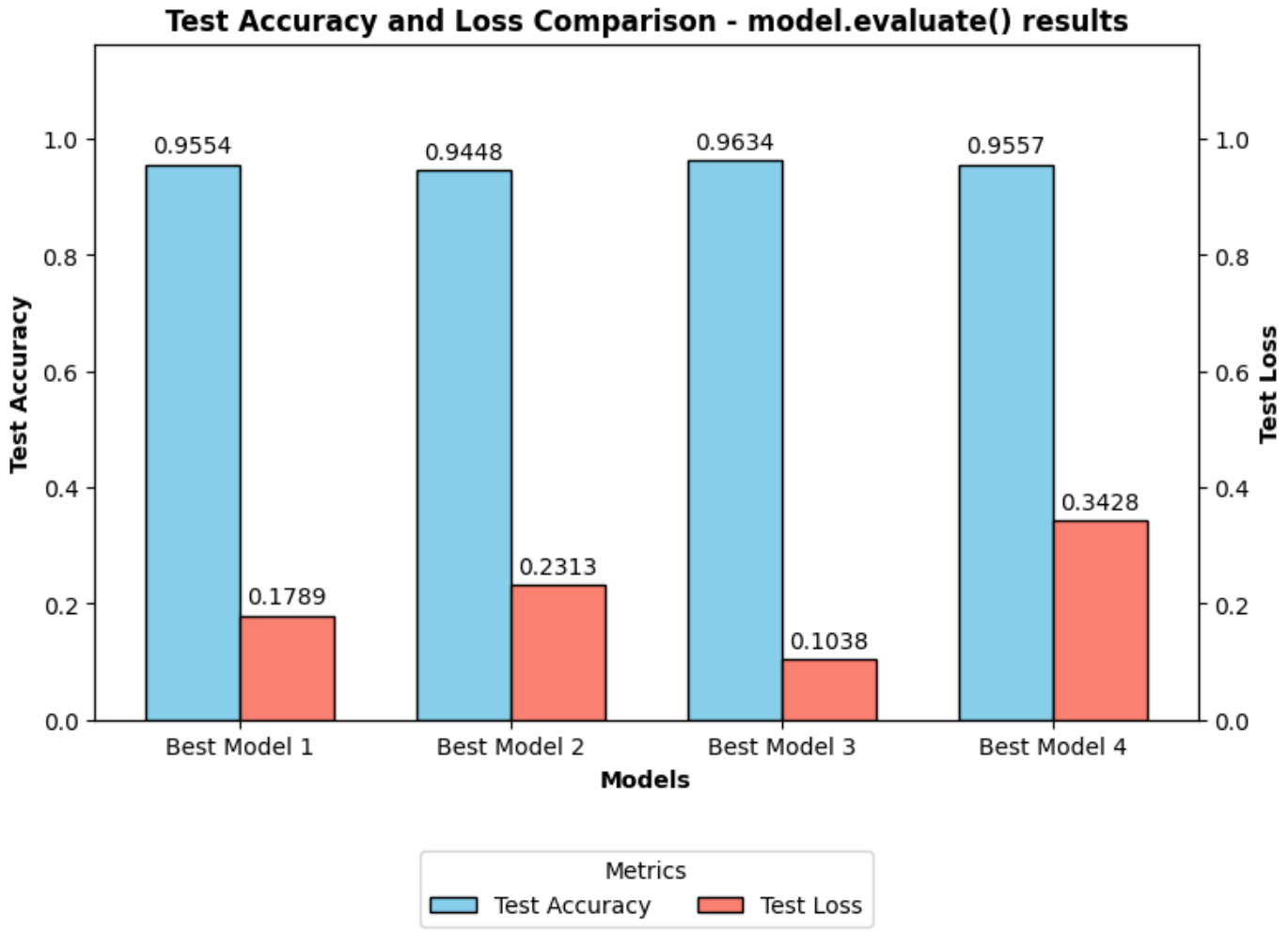


Now we will the performance of each of the 4 models that we ran before selecting a final model.

First, we will compare the results of the model.evaluate() function on each of the 4 models.

In [44]:
```
# define accuracies, losses, and names of models
test_accuracies = [test_accuracy1, test_accuracy2, test_accuracy3, test_accuracy4]
test_losses = [test_loss1, test_loss2, test_loss3, test_loss4]
model_names = ['Best Model 1', 'Best Model 2', 'Best Model 3', 'Best Model 4']

x = np.arange(len(model_names))  # the label locations
width = 0.35  # the width of the bars

fig, ax1 = plt.subplots(figsize=(8, 6))

# Plot test accuracy
bars1 = ax1.bar(x - width/2, test_accuracies, width,
                label='Test Accuracy', color='skyblue', edgecolor='black')

# Create a second y-axis for test loss
ax2 = ax1.twinx()
bars2 = ax2.bar(x + width/2, test_losses, width, label='Test Loss',
                color='salmon', edgecolor='black')

# Add some text for labels, title, and custom x-axis tick labels, etc.
ax1.set_xlabel('Models', fontweight='bold')
ax1.set_ylabel('Test Accuracy', fontweight='bold')
ax2.set_ylabel('Test Loss', fontweight='bold')
ax1.set_title('Test Accuracy and Loss Comparison - model.evaluate() results',
              fontweight='bold')
ax1.set_xticks(x)
ax1.set_xticklabels(model_names)

# Set both y-axes to the same scale
min_value = 0
max_value = max(max(test_accuracies), max(test_losses))

# Adjust y-axis limits for both ax1 and ax2
ax1.set_ylim(min_value, max_value + .2)
ax2.set_ylim(min_value, max_value + 0.2)

# Combine legends
lines1, labels1 = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax1.legend(lines1 + lines2, labels1 + labels2, loc='center',
           bbox_to_anchor=(0.5, -0.25), ncol=2, title='Metrics')

# Add labels
for bar in bars1:
    height = bar.get_height()
    ax1.annotate(f'{height:.4f}',
                 xy=(bar.get_x() + bar.get_width() / 2, height),
                 xytext=(0, 3),
                 textcoords="offset points",
                 ha='center', va='bottom')

for bar in bars2:
    height = bar.get_height()
    ax2.annotate(f'{height:.4f}',
                 xy=(bar.get_x() + bar.get_width() / 2, height),
                 xytext=(0, 3),
                 textcoords="offset points",
                 ha='center', va='bottom')

fig.tight_layout()
```

```
plt.show()
```



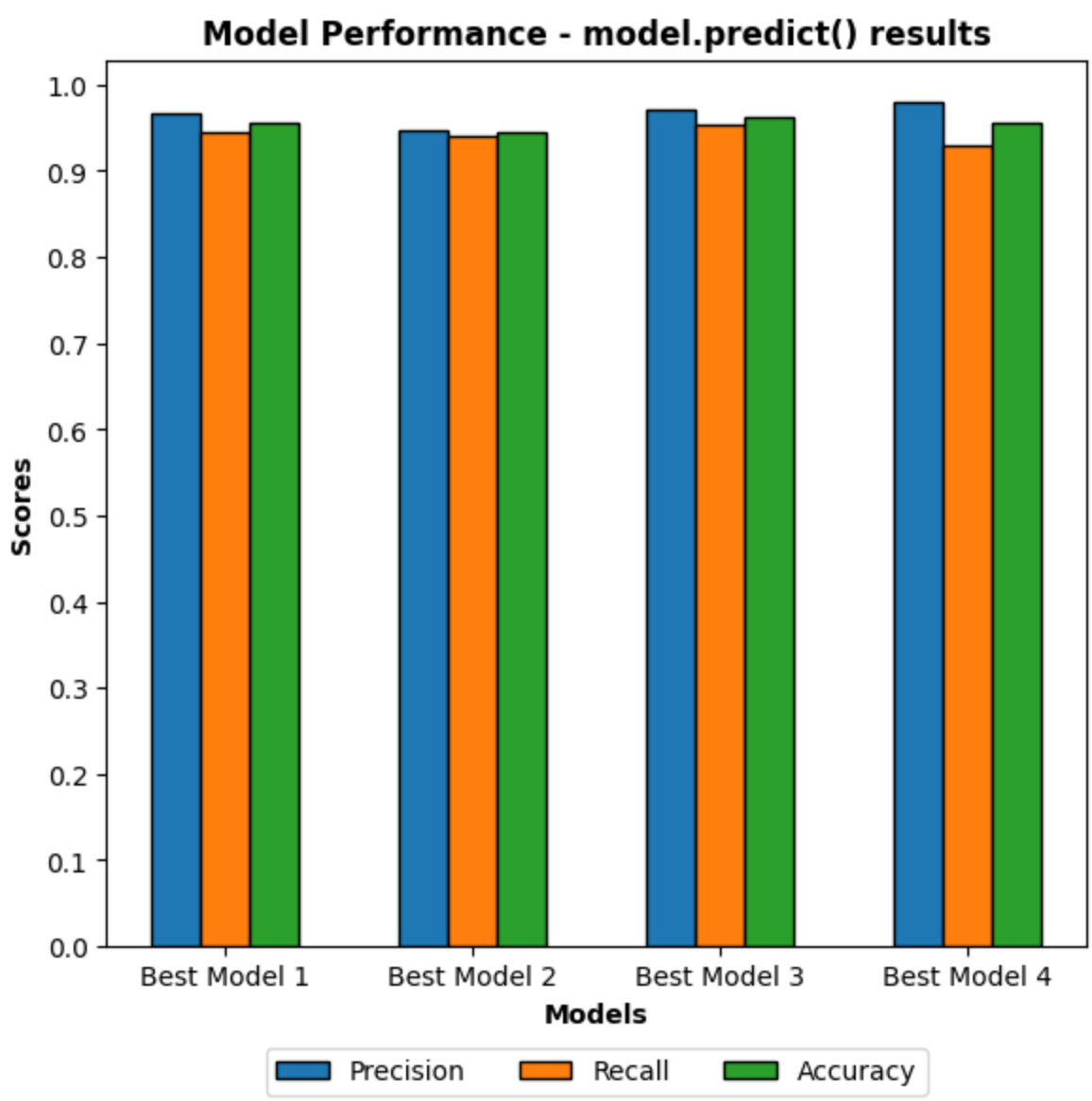
Based on the results of model.evaluate(), our best performing model appears to be Model 3. It has the highest testing accuracy as well as the lowest testing loss.

Next we will compare the results of the model.predict() function. The function below extracts data from the confusion matrices computer earlier that we want to plot for comparison.

In [45]:
```
# takes in confusion matrix and outputs precision, recall, and accuracy for plotting
def compute_metrics(cm):
    precision = cm[1][1]/(cm[1][1] + cm[0][1])
    recall = cm[1][1]/(cm[1][1] + cm[1][0])
    accuracy = (cm[1][1] + cm[0][0])/cm.sum()

    return precision, recall, accuracy

# extract metrics for confusion matrices
prec1, rec1, acc1 = compute_metrics(cm1)
prec2, rec2, acc2 = compute_metrics(cm2)
prec3, rec3, acc3 = compute_metrics(cm3)
prec4, rec4, acc4 = compute_metrics(cm4)
```

In [46]:
```
# Metrics for each group
precisions = [prec1, prec2, prec3, prec4]
recalls = [rec1, rec2, rec3, rec4]
accuracies = [acc1, acc2, acc3, acc4]

# Labels for each group
labels = ['Best Model 1', 'Best Model 2', 'Best Model 3', 'Best Model 4']
```

```
x = np.arange(len(labels))
width = 0.2

fig, ax = plt.subplots(figsize=(6, 6))

# Plotting the bars for each metric
rects1 = ax.bar(x - width, precisions, width,
                label='Precision', edgecolor='black')
rects2 = ax.bar(x, recalls, width,
                label='Recall', edgecolor='black')
rects3 = ax.bar(x + width, accuracies, width,
                label='Accuracy', edgecolor='black')

# Adding labels and title
ax.set_ylabel('Scores', fontweight='bold')
ax.set_yticks(np.arange(0, 1.1, .1))
ax.set_xlabel('Models', fontweight='bold')
ax.set_title('Model Performance - model.predict() results', fontweight='bold')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend()
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.1), ncol=3)

fig.tight_layout()
plt.show()


# display a table of the results
predict_metrics = {
    'Model': labels,
    'Precision': np.round(precisions,4),
    'Recall': np.round(recalls,4),
    'Accuracy': np.round(accuracies,4)
}

# Create a DataFrame
df_metrics = pd.DataFrame(predict_metrics)
df_metrics.set_index('Model', inplace=True)

# bold headers and indices
headers = [f'\033[1m{header}\033[0m' for header in df_metrics.columns]
index = [f'\033[1m{idx}\033[0m' for idx in df_metrics.index]

# fancy grid with tabulate
print('\nPrediction Metrics Table:')
print(tabulate(df_metrics, headers=headers,
               tablefmt='fancy_grid',showindex=index))
```



Prediction Metrics Table:

	Precision	Recall	Accuracy
Best Model 1	0.9659	0.9441	0.9554
Best Model 2	0.9481	0.9412	0.9448
Best Model 3	0.9719	0.9543	0.9634
Best Model 4	0.9801	0.9303	0.9557

In [47]:
```
# model with best precision
best_prec = np.max(df_metrics['Precision'])
best_prec_model = df_metrics[df_metrics['Precision']==best_prec].index[0]

# model with best recall
best_rec = np.max(df_metrics['Recall'])
best_rec_model = df_metrics[df_metrics['Recall']==best_rec].index[0]

# model with best accuracy
best_acc = np.max(df_metrics['Accuracy'])
best_acc_model = df_metrics[df_metrics['Accuracy']==best_acc].index[0]

# print results
print(f'Highest Precision - {best_prec_model}: {best_prec}')
```

```
print(f'Highest Recall - {best_rec_model}: {best_rec}')
print(f'Highest Accuracy - {best_acc_model}: {best_acc}')
```

```
Highest Precision - Best Model 4: 0.9801
Highest Recall - Best Model 3: 0.9543
Highest Accuracy - Best Model 3: 0.9634
```

We may want to modify the threshold at which a prediction is converted to a 0 or 1. In this case for instance, falsely classifing an image as 'Uninfected' when in fact the image was taken from a patient with Malaria, could have drastic consequences (depending on the application of the model). A person infected with Malaria may not be given the early care they need in order to survive if they are falsely diagnosed as healthy. In our case, we will assume False negatives would have far worse effects than False Positives. Particularly if positive results are re-verified using traditional methods.

Because the ultimate goal of our model is to minimize deaths caused by Malaria, we may be willing to sacrifice some model accuracy in order to increase the recall of the model. The recall of the model measures how well the model predicts instances of Malaria from the total number of actual instances of Malaria. Higher recall means less False negatives diagnoses.

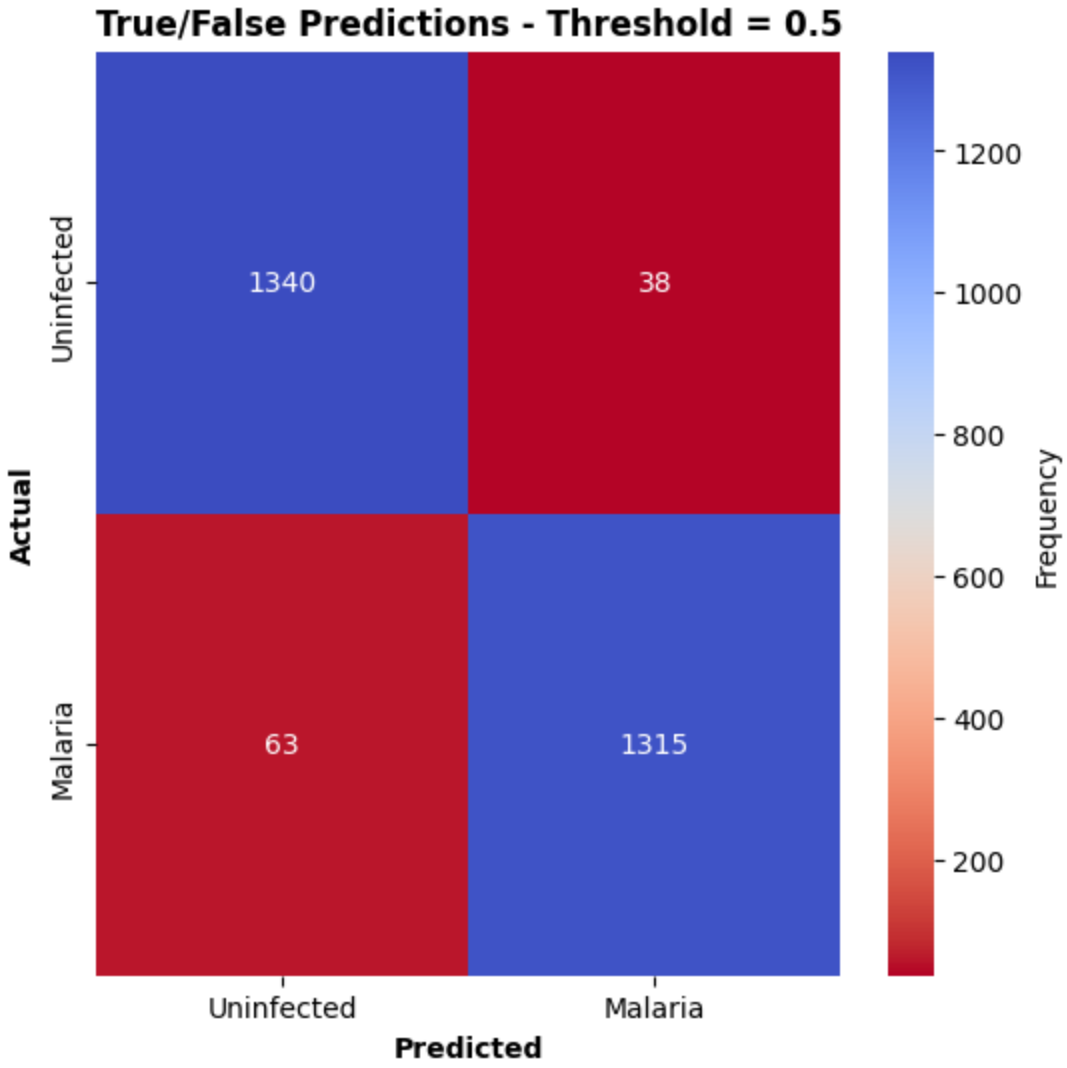
Now that we have our highest performing model (model 3), we can modify the classification theshold at which the output probability is converted to a positive ('Malaria') prediction.

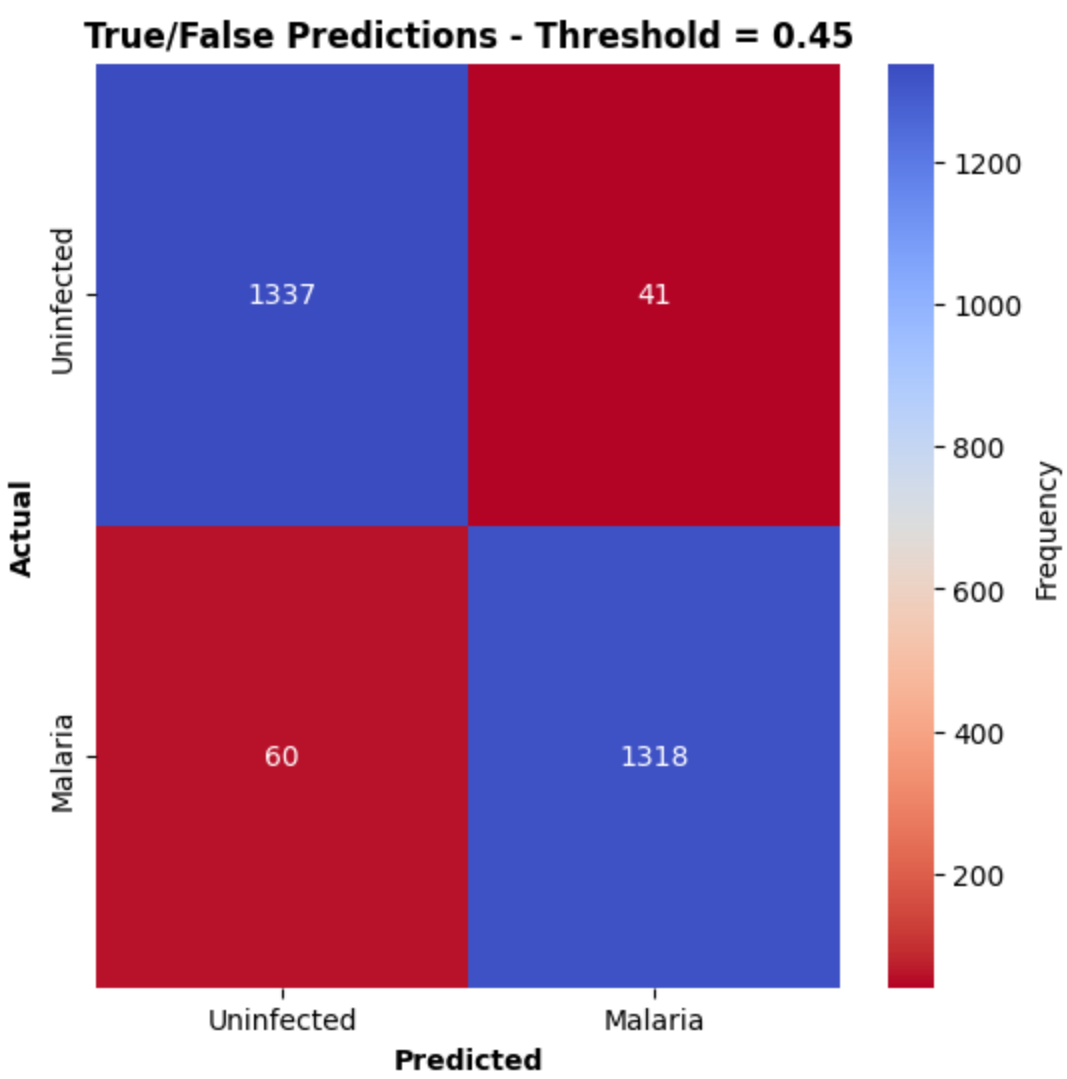
By default, the classification threshold is .5 in a binary classification model. Predictions greater than .5 are converted to 1s and predictions less than .5 are converted to 0s. By lowering the classification threshold, we will increase the amount of positive ('Malaria') predictions. This will increase the amount of False Positives and decrease the amount of False Negatives.

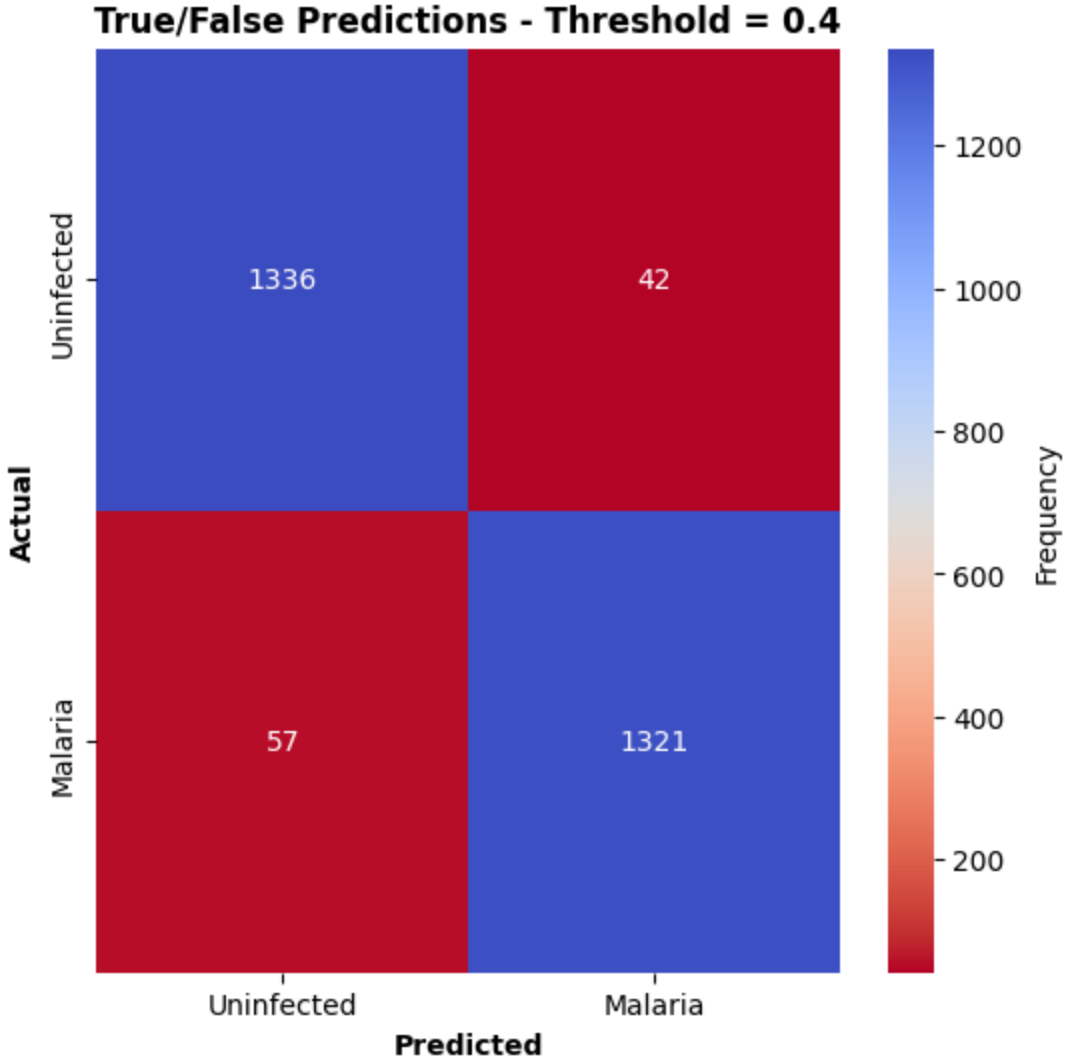
In [48]:
```
# Load model with highest accuracy
best_model_path = 'best_model_wholeset_v3.h5'
best_model = load_model(best_model_path)

# Make predictions
predictions = best_model.predict(test_generator)

# define a list of different thresholds
thresholds = [0.50, 0.45, 0.40, 0.35, 0.30, 0.25, .20, .15]

thresholds_dict = {}
for threshold in thresholds:
    # generate predictions based on each threshold
    predicted_labels = (predictions > threshold).astype(int).flatten()

    # True labels
    true_labels = test_generator.classes

    # Compute metrics
    cm = confusion_matrix(true_labels, predicted_labels)

    # Store results in dictionary
    thresholds_dict[threshold] = cm

    # Plot confusion matrix using seaborn
    plt.figure(figsize=(6, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='coolwarm_r',
                xticklabels=['Uninfected', 'Malaria'],
                yticklabels=['Uninfected', 'Malaria'],
                cbar_kws={'label': 'Frequency'})
    plt.ylabel('Actual', fontweight='bold')
    plt.xlabel('Predicted', fontweight='bold')
```

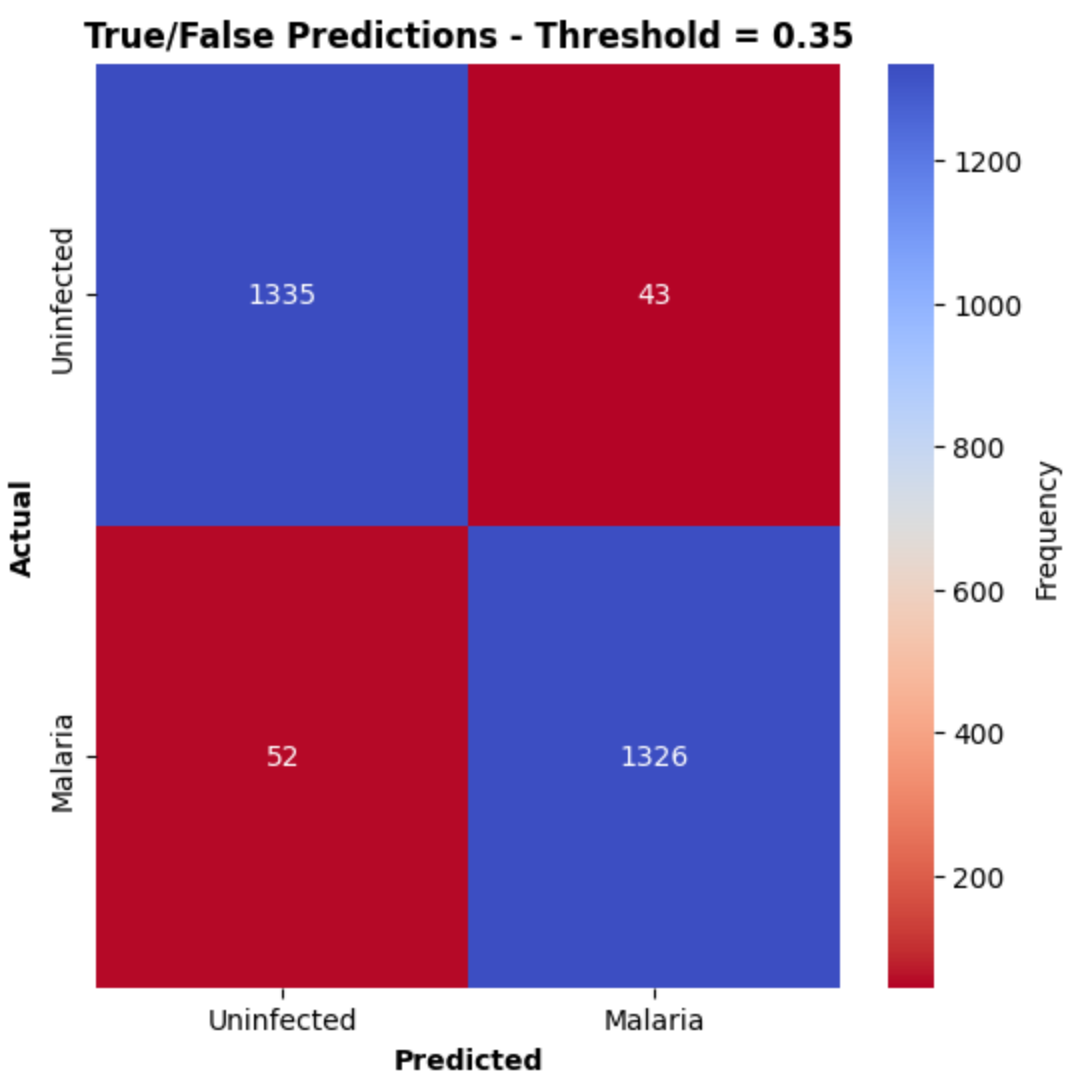
```
    plt.title(f'True/False Predictions - Threshold = {threshold}', fontweight='bold')
    plt.show()
```

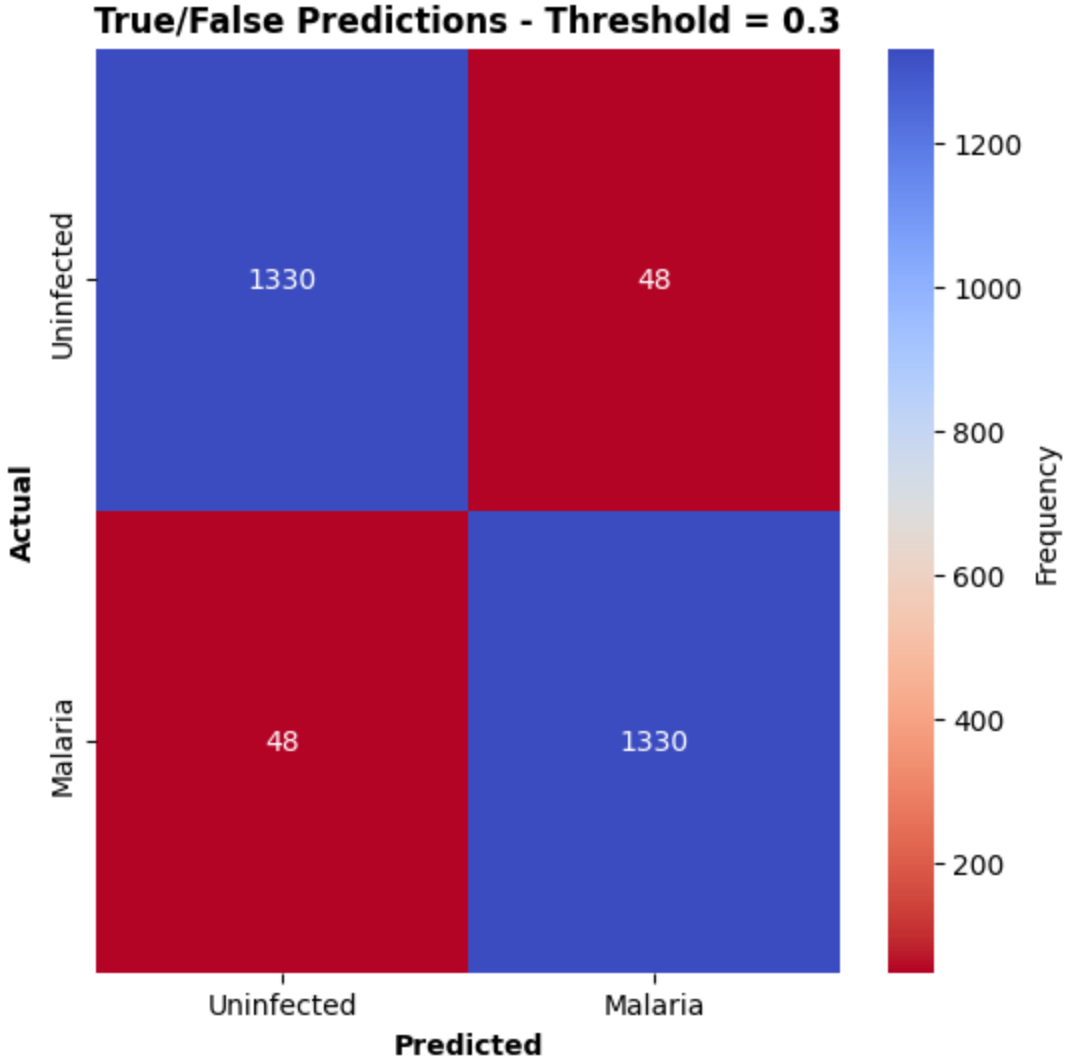
```
11/11 [==============================] - 7s 653ms/step
```

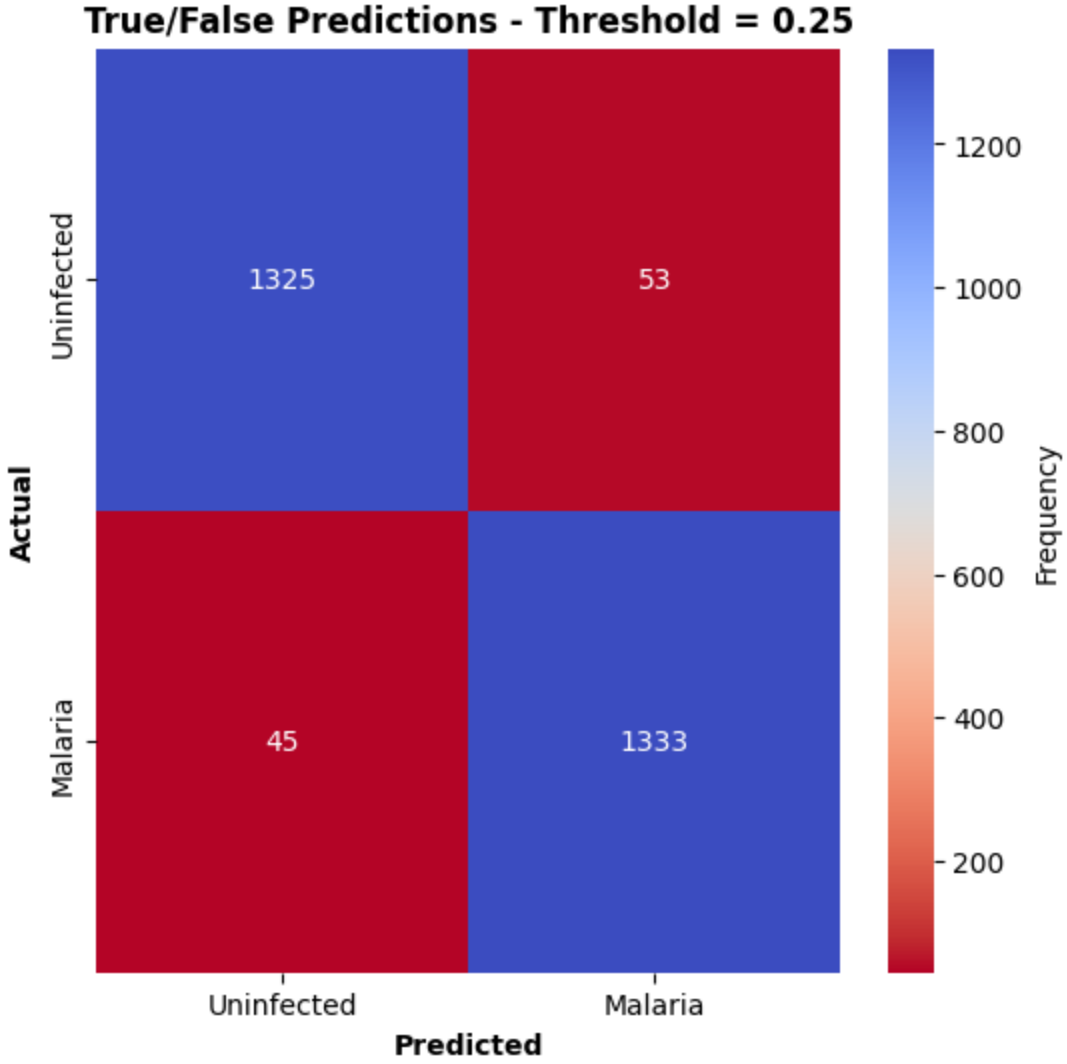


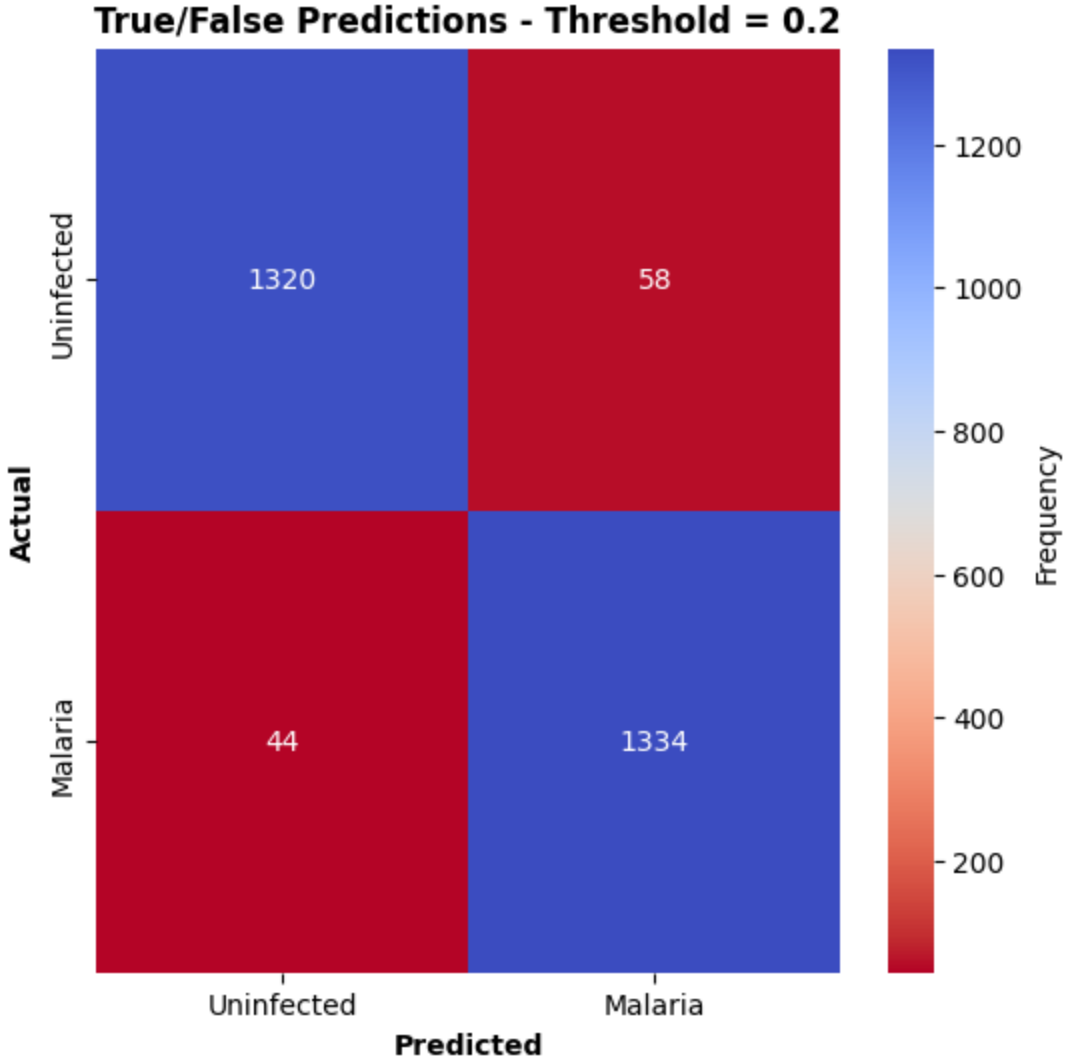

True/False Predictions - Threshold = 0.45
Actual
Uninfected
Malaria
1337
41
60
1318
Uninfected
Malaria
Predicted
1200
1000
800
600
400
200
Frequency

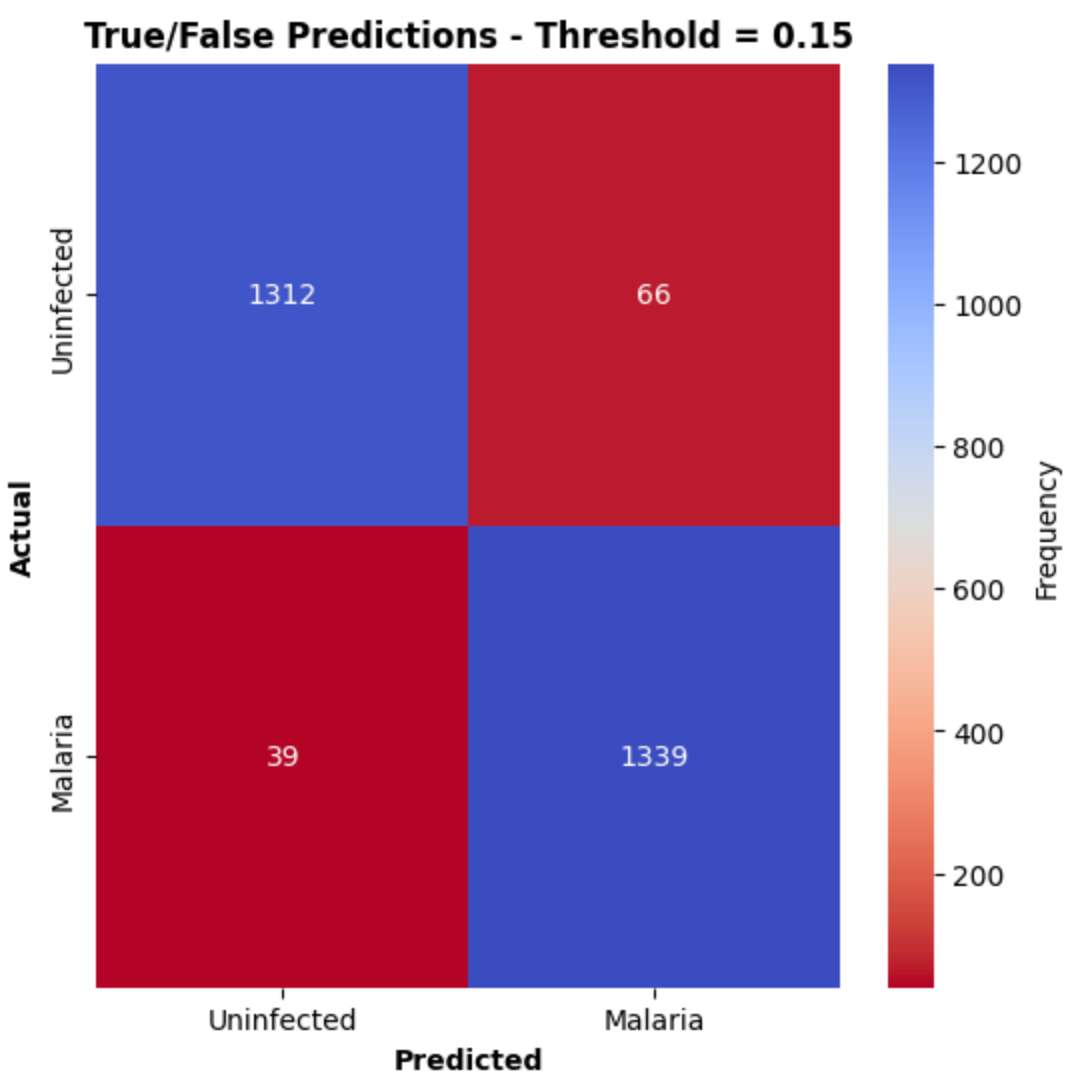

True/False Predictions - Threshold = 0.4
Actual
Uninfected
Malaria
1336
42
57
1321
Uninfected
Malaria
Predicted
1200
1000
800
600
400
200
Frequency


True/False Predictions - Threshold = 0.35
Actual
Uninfected
Malaria
1335
43
52
1326
Uninfected
Malaria
Predicted
1200
1000
800
600
400
200
Frequency


True/False Predictions - Threshold = 0.3
Actual
Uninfected
Malaria
1330
48
48
1330
Uninfected
Malaria
Predicted
1200
1000
800
600
400
200
Frequency


True/False Predictions - Threshold = 0.25
Actual
Uninfected
Malaria
1325
53
45
1333
Uninfected
Malaria
Predicted
1200
1000
800
600
400
200
Frequency


True/False Predictions - Threshold = 0.2
Actual
Uninfected
Malaria
1320
58
44
1334
Uninfected
Malaria
Predicted
1200
1000
800
600
400
200
Frequency



In [49]:

```
# compute recall from each modified threshold
prec_50, rec_50, acc_50 = compute_metrics(thresholds_dict[.50])
prec_45, rec_45, acc_45 = compute_metrics(thresholds_dict[.45])
prec_40, rec_40, acc_40 = compute_metrics(thresholds_dict[.40])
prec_35, rec_35, acc_35 = compute_metrics(thresholds_dict[.35])
prec_30, rec_30, acc_30 = compute_metrics(thresholds_dict[.30])
prec_25, rec_25, acc_25 = compute_metrics(thresholds_dict[.25])
prec_20, rec_20, acc_20 = compute_metrics(thresholds_dict[.20])
prec_15, rec_15, acc_15 = compute_metrics(thresholds_dict[.15])

prec_list = [prec_50, prec_45, prec_40, prec_35, prec_30, prec_25, prec_20, prec_15]
rec_list = [rec_50, rec_45, rec_40, rec_35, rec_30, rec_25, rec_20, rec_15]
acc_list = [acc_50, acc_45, acc_40, acc_35, acc_30, acc_25, acc_20, acc_15]

# print out model recall for each threshold
for thresh, rec in zip(thresholds, rec_list):
    print(f'Classification Threshold {thresh}: Recall = {np.round(rec,4)}')

# display a table of the results
threshold_metrics = {
    'Threshold': thresholds,
    'Precision': np.round(prec_list,4),
    'Recall': np.round(rec_list,4),
    'Accuracy': np.round(acc_list,4)
}

# Create a DataFrame
threshold_metrics = pd.DataFrame(threshold_metrics)
threshold_metrics.set_index('Threshold', inplace=True)
```

```
# bold headers and indices
headers = [f'\033[1m{header}\033[0m' for header in threshold_metrics.columns]
index = [f'\033[1m{idx}\033[0m' for idx in threshold_metrics.index]

# fancy grid using tabulate
print('\nThreshold Metrics Table - Best Model 3:')
print(tabulate(threshold_metrics, headers=headers,
               tablefmt='fancy_grid',showindex=index))
```

```
Classification Threshold 0.5: Recall = 0.9543
Classification Threshold 0.45: Recall = 0.9565
Classification Threshold 0.4: Recall = 0.9586
Classification Threshold 0.35: Recall = 0.9623
Classification Threshold 0.3: Recall = 0.9652
Classification Threshold 0.25: Recall = 0.9673
Classification Threshold 0.2: Recall = 0.9681
Classification Threshold 0.15: Recall = 0.9717

Threshold Metrics Table - Best Model 3:
```

	Precision	Recall	Accuracy
0.5	0.9719	0.9543	0.9634
0.45	0.9698	0.9565	0.9634
0.4	0.9692	0.9586	0.9641
0.35	0.9686	0.9623	0.9655
0.3	0.9652	0.9652	0.9652
0.25	0.9618	0.9673	0.9644
0.2	0.9583	0.9681	0.963
0.15	0.953	0.9717	0.9619

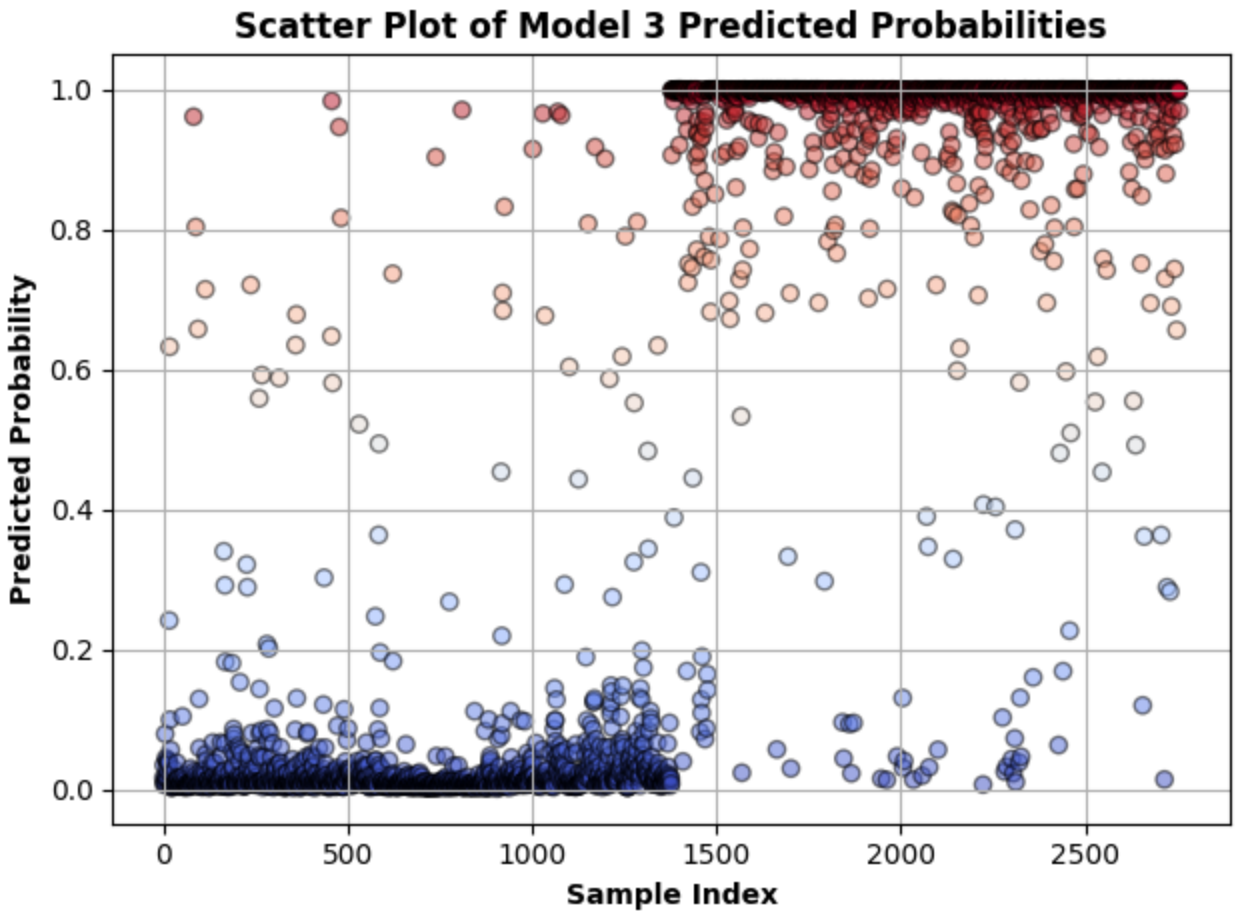
I was surprised to find that the different classification thresholds did not have a large effect on the output predictions. The lower threshold predictions did make more predictions for Malaria but only slightly more. For instance, the lowest threshold model of .15 only predicted 71 more instances of Malaria than the highest threshold model of .50 (threshold = .15 predicted 1444 Malaria instances, threshold = .50 predicted 1371 Malaria instances). Model recall was increased, but not as much as I initially expected. The increase in model recall came at the cost of precision/accuracy as shown in the table above.

These puzzling results are likely the result of our model making very confident predictions. For instance of the model was generating outputs that were almost always < .1 or greater than .9, the modified thresholds would not have a very large effect.

We can create a scatterplot of the predicted probabilities to see if this is the case.

In [50]:
```
# define x,y
pred_probs = predictions.flatten()
indices = np.arange(len(pred_probs))

# create scatterlot of output probs of Model 3
plt.scatter(indices, pred_probs, c=pred_probs, cmap='coolwarm',
            alpha=.5, edgecolor='k')
plt.title('Scatter Plot of Model 3 Predicted Probabilities', fontweight='bold')
plt.xlabel('Sample Index', fontweight='bold')
```

```
plt.ylabel('Predicted Probability', fontweight='bold')

plt.grid(True)
plt.tight_layout()
plt.show()
```



Based on the scatterplot, the vast majority of the predictions are made with great confidence. This is why modifying our classification threshold had little effect.

Results:

The model that performed the best after validation and testing was the third model trained on the entire dataset 'best_model_wholeset_v3.h5'. This model had the highest accuracy (.9655) and highest recall (.963) when making predictions on the previously unseen testing data.

The architecture of Model 3 was as follows: MODEL 3 DETAILS Target Size: 128x128 Batch Size: 256 Data Augmentation used 3 Convolutional Layers 3 Dense Layers 50% Dropout between each dense layer

Depending on the applicaiton of this model, if minimizing False negatives is the utmost priority, then the classification threshold can be lowered. This will come at the cost of precision and likely accuracy.