In this notebook, I will be working through the Histopathologic Cancer Detection Kaggle competition. It can be accessed at https://www.kaggle.com/c/histopathologic-cancer-detection . It involves identifying metastatic cancer in small image patches taken from larger digital pathology scans, predicting whether the image patch contains a tumor based on center-region pixel information. The final model must correctly classify new images into positive (if it contains tumor) and negative (if it does not) categories. The project is available at https://github.com/giosofteng/hcd .

The data consists of 96x96px tif images. However, only the center 32x32px regions are to be used for analysis. A positive label indicates that at least one pixel in this square region is of a tumor. Additionally, the data is rather clean and contains no duplicates.

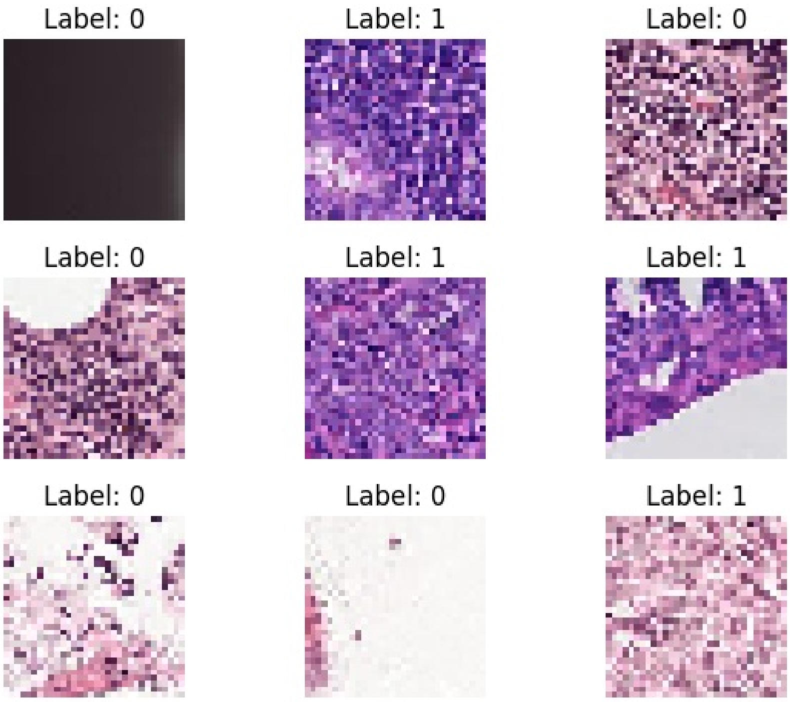
Now, let us import and display the data. Let us also do some basic pre-processing to improve model performance.

In [1]:
```python
import random
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# for reproducibility
tf.config.experimental.enable_op_determinism()
tf.random.set_seed(42)
np.random.seed(42)
random.seed(42)

# use better formatting
pd.set_option('display.expand_frame_repr', False)

# import labels
df_labels = pd.read_csv('data/train_labels.csv')
# make `label` column type string--needed for Keras
df_labels['label'] = df_labels['label'].astype(str)
# add `file` column--useful for Keras ImageDataGenerator
df_labels['file'] = df_labels['id'] + '.tif'
# display data
print(f'Data Shape: {df_labels.shape}\n')
print(f'Data Sample:\n{df_labels.head()}\n')

# init ImageDataGenerator to load images
# normalize pixel values and resize images to 32x32px to improve model performance
data_generator = ImageDataGenerator(rescale=1/255).flow_from_dataframe(
    dataframe=df_labels,
    directory='data/train',
    x_col='file',
    y_col='label',
    target_size=(32, 32),
    class_mode='binary',
    batch_size=32,
    seed=42
)
```

```
Data Shape: (220025, 3)

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

Found 220025 validated image filenames belonging to 2 classes.
```

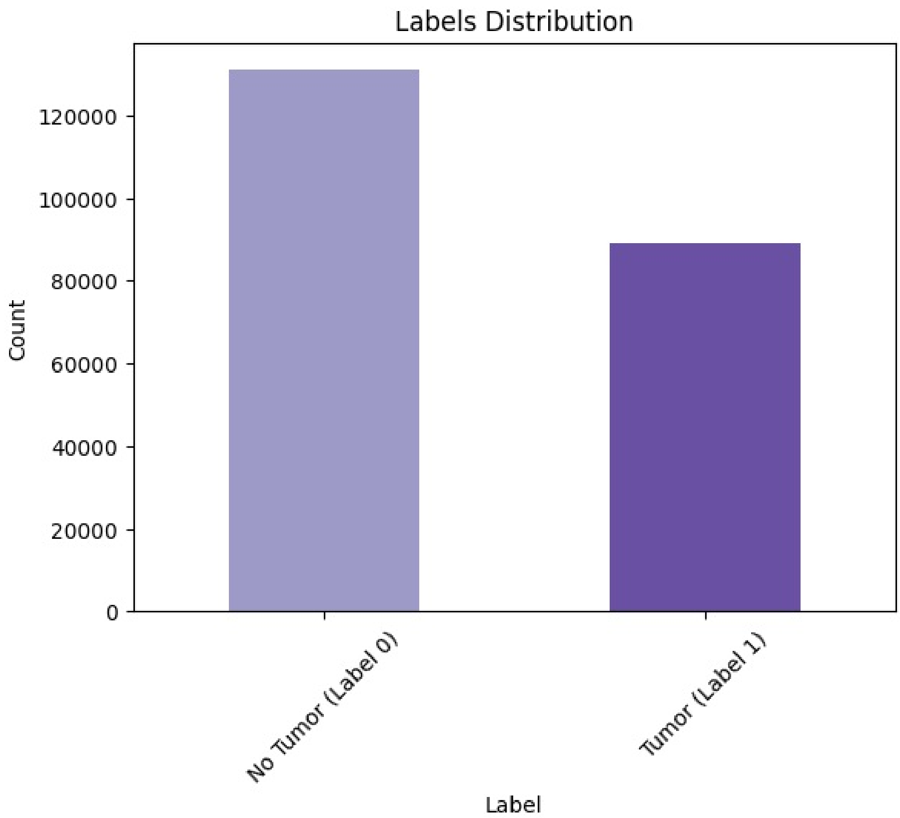
Let us also display some sample images from the dataset:

In [2]:
```python
import matplotlib.pyplot as plt

# get a batch of images with labels
images, labels = next(data_generator)

# display 9 images
for i in range(9):
    plt.subplot(3, 3, i+1)
    plt.axis('off')
    plt.title(f'Label: {int(labels[i])}')
    plt.imshow(images[i])
plt.tight_layout()
plt.show()
```



Let us display a few more important metrics using some plots:

In [3]:
```
# plot labels distribution via histogram
df_labels['label'].value_counts().plot(kind='bar', color=plt.cm.Purples([0.5, 0.75]))
plt.title('Labels Distribution')
plt.xlabel('Label')
plt.ylabel('Count')
plt.xticks(ticks=[0, 1], labels=['No Tumor (Label 0)', 'Tumor (Label 1)'], rotation=45)
plt.show()
```



So, it seems that most of our data is of non-tumor tissue. Still, we have plenty of samples for both categories.

Let us now move on to designing and implementing a simple CNN architecture to tackle this problem. Let us use 3 convolution layers of increasing filter sizes (32, 64, and 128) with max pooling after each to reduce dimensionality. Let us also place two fully connected layers at the end and apply dropout to avoid overfitting. Let us use ReLU activations except for the output layer.

In [4]:
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam
```

```python
# init model
model_sequential = Sequential()
# add input layer
model_sequential.add(Input(shape=(32, 32, 3)))
# add 3 convolution and max pooling layers of sizes 32, 64, 128
model_sequential.add(Conv2D(32, (3, 3), activation='relu'))
model_sequential.add(MaxPooling2D((2, 2)))
model_sequential.add(Conv2D(64, (3, 3), activation='relu'))
model_sequential.add(MaxPooling2D((2, 2)))
model_sequential.add(Conv2D(128, (3, 3), activation='relu'))
model_sequential.add(MaxPooling2D((2, 2)))
# add flatten and dense layers + dropout for regularization
model_sequential.add(Flatten())
model_sequential.add(Dense(128, activation='relu'))
model_sequential.add(Dropout(0.5))
model_sequential.add(Dense(64, activation='relu'))
model_sequential.add(Dropout(0.5))
# add output layer
model_sequential.add(Dense(1, activation='sigmoid'))
# compile model
model_sequential.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='binary_crossentropy',
    metrics=['accuracy']
)
# print model architecture
model_sequential.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 128)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 128)	65,664
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8,256
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 1)	65

Total params: 167,233 (653.25 KB)
Trainable params: 167,233 (653.25 KB)
Non-trainable params: 0 (0.00 B)

With our model in place, let us evaluate its performance:

In [5]:
```python
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping

# split data into train and validate sets (80/20)
df_train, df_validate = train_test_split(
    df_labels, test_size=0.2, stratify=df_labels['label'], random_state=42)

# train data generator + data augmentation for better performance
train_generator = ImageDataGenerator(
    rescale=1/255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
).flow_from_dataframe(
    dataframe=df_train,
    directory='data/train',
```

```python
    x_col='file',
    y_col='label',
    target_size=(32, 32),
    class_mode='binary',
    batch_size=32,
    seed=42
)
# validate data generator (without augmentation)
validate_generator = ImageDataGenerator(rescale=1/255).flow_from_dataframe(
    dataframe=df_validate,
    directory='data/train',
    x_col='file',
    y_col='label',
    target_size=(32, 32),
    class_mode='binary',
    batch_size=32,
    seed=42
)

# train model with early stopping
history_sequential = model_sequential.fit(
    train_generator,
    steps_per_epoch=len(train_generator) // 32,
    epochs=50,
    validation_data=validate_generator,
    validation_steps=len(validate_generator) // 32,
    callbacks=[EarlyStopping(
        monitor='val_loss',
        patience=5,
        restore_best_weights=True
    )]
)
# evaluate model and print results
validate_loss, validate_accuracy = model_sequential.evaluate(validate_generator)
print(f'Validation Loss: {validate_loss:.2f}')
print(f'Validation Accuracy: {validate_accuracy:.2f}')
```

```
Found 176020 validated image filenames belonging to 2 classes.
Found 44005 validated image filenames belonging to 2 classes.
Epoch 1/50
/Users/george/projects/histopathologic-cancer-detection/venv/lib/python3.12/site-packages/keras/src/trainers/dat
a_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs
)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max_queue_size`. Do not pass th
ese arguments to `fit()`, as they will be ignored.
  self._warn_if_super_not_called()
171/171 ━━━━━━━━━━━━━━━━━━━━ 6s 28ms/step - accuracy: 0.5468 - loss: 0.6857 - val_accuracy: 0.6119 - val_loss: 0
.6511
Epoch 2/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 5s 28ms/step - accuracy: 0.6258 - loss: 0.6452 - val_accuracy: 0.6853 - val_loss: 0
.6524
Epoch 3/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 5s 27ms/step - accuracy: 0.7464 - loss: 0.5569 - val_accuracy: 0.6592 - val_loss: 0
.8146
Epoch 4/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 5s 26ms/step - accuracy: 0.7476 - loss: 0.5471 - val_accuracy: 0.6722 - val_loss: 0
.8111
Epoch 5/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 5s 27ms/step - accuracy: 0.7671 - loss: 0.5294 - val_accuracy: 0.5916 - val_loss: 1
.0093
Epoch 6/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 26ms/step - accuracy: 0.7685 - loss: 0.5251 - val_accuracy: 0.6308 - val_loss: 0
.9466
1376/1376 ━━━━━━━━━━━━━━━━━━━━ 17s 12ms/step - accuracy: 0.5972 - loss: 0.6574
Validation Loss: 0.66
Validation Accuracy: 0.59
```

With our initial model evaluated, we can see that there is still plenty of room for improvement!

Let us divide the learning rate by 10 and see how our results compare:

In [6]:

```python
# re-compile model with slower learning rate
model_sequential.compile(
    optimizer=Adam(learning_rate=0.00001),
    loss='binary_crossentropy',
    metrics=['accuracy']
)
# re-train model and print results
history_sequential_slower = model_sequential.fit(
    train_generator,
    steps_per_epoch=len(train_generator) // 32,
    epochs=50,
    validation_data=validate_generator,
    validation_steps=len(validate_generator) // 32,
```

```
    callbacks=[EarlyStopping(
        monitor='val_loss',
        patience=5,
        restore_best_weights=True
    )]
)
validate_loss_slower, validate_accuracy_slower = model_sequential.evaluate(validate_generator)
print(f'Validation Loss: {validate_loss_slower:.2f}')
print(f'Validation Accuracy: {validate_accuracy_slower:.2f}')
```

```
Epoch 1/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 5s 26ms/step - accuracy: 0.6067 - loss: 0.6516 - val_accuracy: 0.5887 - val_loss: 0
.6502
Epoch 2/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.5999 - loss: 0.6544 - val_accuracy: 0.6243 - val_loss: 0
.6203
Epoch 3/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 25ms/step - accuracy: 0.6126 - loss: 0.6405 - val_accuracy: 0.5879 - val_loss: 0
.6490
Epoch 4/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.6307 - loss: 0.6377 - val_accuracy: 0.6003 - val_loss: 0
.6477
Epoch 5/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.6510 - loss: 0.6341 - val_accuracy: 0.6192 - val_loss: 0
.6341
Epoch 6/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.6736 - loss: 0.6193 - val_accuracy: 0.6308 - val_loss: 0
.6336
Epoch 7/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.7109 - loss: 0.5958 - val_accuracy: 0.6112 - val_loss: 0
.6958
1376/1376 ━━━━━━━━━━━━━━━━━━━━ 12s 8ms/step - accuracy: 0.5949 - loss: 0.6405
Validation Loss: 0.64
Validation Accuracy: 0.59
```

We do not see a big change in results. How about with a 10x faster learning rate instead:

In [7]:
```
# re-compile model with slower learning rate
model_sequential.compile(
    optimizer=Adam(learning_rate=0.001),
    loss='binary_crossentropy',
    metrics=['accuracy']
)
# re-train model and print results
history_sequential_faster = model_sequential.fit(
    train_generator,
    steps_per_epoch=len(train_generator) // 32,
    epochs=50,
    validation_data=validate_generator,
    validation_steps=len(validate_generator) // 32,
    callbacks=[EarlyStopping(
        monitor='val_loss',
        patience=5,
        restore_best_weights=True
    )]
)
validate_loss_faster, validate_accuracy_faster = model_sequential.evaluate(validate_generator)
print(f'Validation Loss: {validate_loss_faster:.2f}')
print(f'Validation Accuracy: {validate_accuracy_faster:.2f}')
```

```
Epoch 1/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 5s 25ms/step - accuracy: 0.6005 - loss: 0.6563 - val_accuracy: 0.6846 - val_loss: 0
.6170
Epoch 2/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.7219 - loss: 0.5733 - val_accuracy: 0.6359 - val_loss: 0
.7695
Epoch 3/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.7670 - loss: 0.5288 - val_accuracy: 0.6533 - val_loss: 0
.7312
Epoch 4/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 24ms/step - accuracy: 0.7499 - loss: 0.5346 - val_accuracy: 0.6177 - val_loss: 0
.9083
Epoch 5/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 25ms/step - accuracy: 0.7445 - loss: 0.5378 - val_accuracy: 0.6374 - val_loss: 0
.7460
Epoch 6/50
171/171 ━━━━━━━━━━━━━━━━━━━━ 4s 25ms/step - accuracy: 0.7744 - loss: 0.5030 - val_accuracy: 0.6323 - val_loss: 0
.9474
1376/1376 ━━━━━━━━━━━━━━━━━━━━ 12s 9ms/step - accuracy: 0.6680 - loss: 0.6334
Validation Loss: 0.63
Validation Accuracy: 0.67
```

Not good enough! Let us try a more complex model instead, leveraging transfer learning using the MobileNetV2 architecture.

In [8]:

```python
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import BatchNormalization, GlobalAveragePooling2D

# do not resize input images--MobileNetV2 does not accept 32x32 input
# reduce batch size for memory considerations
train_generator_large = ImageDataGenerator(
    rescale=1/255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True
).flow_from_dataframe(
    dataframe=df_train,
    directory='data/train',
    x_col='file',
    y_col='label',
    target_size=(96, 96),
    class_mode='binary',
    batch_size=16,
    seed=42
)
validate_generator_large = ImageDataGenerator(rescale=1/255).flow_from_dataframe(
    dataframe=df_validate,
    directory='data/train',
    x_col='file',
    y_col='label',
    target_size=(96, 96),
    class_mode='binary',
    batch_size=16,
    seed=42
)

# init Xception pre-trained model minus top layer and freeze it
mobilenetv2 = MobileNetV2(weights='imagenet', include_top=False, input_shape=(96, 96, 3))
mobilenetv2.trainable = False

# build improved model
model_transfer = Sequential()
model_transfer.add(mobilenetv2)
model_transfer.add(GlobalAveragePooling2D())
model_transfer.add(Dense(256, activation='relu'))
model_transfer.add(BatchNormalization())
model_transfer.add(Dropout(0.5))
model_transfer.add(Dense(64, activation='relu'))
model_transfer.add(Dropout(0.5))
model_transfer.add(Dense(1, activation='sigmoid'))

# compile improved model
model_transfer.compile(
    optimizer=Adam(learning_rate=0.0001),
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# unfreeze last 20 layers to improve results
for layer in mobilenetv2.layers[-20:]:
    layer.trainable = True

# print improved model architecture
model_transfer.summary()
```

```
Found 176020 validated image filenames belonging to 2 classes.
Found 44005 validated image filenames belonging to 2 classes.
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
mobilenetv2_1.00_96 (Functional)	(None, 3, 3, 1280)	2,257,984
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense_3 (Dense)	(None, 256)	327,936
batch_normalization (BatchNormalization)	(None, 256)	1,024
dropout_2 (Dropout)	(None, 256)	0
dense_4 (Dense)	(None, 64)	16,448
dropout_3 (Dropout)	(None, 64)	0
dense_5 (Dense)	(None, 1)	65

Total params: 2,603,457 (9.93 MB)

Trainable params: 1,551,041 (5.92 MB)

Non-trainable params: 1,052,416 (4.01 MB)

Let us now train the new model and display the results:

In [10]:
```
history_model_transfer = model_transfer.fit(
    train_generator_large,
    steps_per_epoch=len(train_generator_large) // 16,
    epochs=50,
    validation_data=validate_generator_large,
    validation_steps=len(validate_generator_large) // 16,
    callbacks=[EarlyStopping(
        monitor='val_loss',
        patience=5,
        restore_best_weights=True
    )]
)
validate_loss_improved, validate_accuracy_improved = model_transfer.evaluate(validate_generator_large)
print(f'Validation Loss: {validate_loss_improved:.2f}')
print(f'Validation Accuracy: {validate_accuracy_improved:.2f}')
```

```
Epoch 1/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 54s 79ms/step - accuracy: 0.7719 - loss: 0.5388 - val_accuracy: 0.8012 - val_loss:
0.5329
Epoch 2/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 48s 69ms/step - accuracy: 0.7948 - loss: 0.4787 - val_accuracy: 0.8297 - val_loss:
0.4156
Epoch 3/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 55s 80ms/step - accuracy: 0.7886 - loss: 0.4667 - val_accuracy: 0.8282 - val_loss:
0.4383
Epoch 4/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 50s 73ms/step - accuracy: 0.8023 - loss: 0.4452 - val_accuracy: 0.8154 - val_loss:
0.4210
Epoch 5/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 51s 75ms/step - accuracy: 0.8169 - loss: 0.4229 - val_accuracy: 0.8359 - val_loss:
0.3976
Epoch 6/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 54s 79ms/step - accuracy: 0.8098 - loss: 0.4282 - val_accuracy: 0.8505 - val_loss:
0.3467
Epoch 7/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 54s 79ms/step - accuracy: 0.8190 - loss: 0.4046 - val_accuracy: 0.8443 - val_loss:
0.3502
Epoch 8/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 57s 83ms/step - accuracy: 0.8207 - loss: 0.4079 - val_accuracy: 0.8355 - val_loss:
0.4099
Epoch 9/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 55s 81ms/step - accuracy: 0.8234 - loss: 0.4186 - val_accuracy: 0.8348 - val_loss:
0.4306
Epoch 10/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 55s 81ms/step - accuracy: 0.8373 - loss: 0.3884 - val_accuracy: 0.8282 - val_loss:
0.4366
Epoch 11/50
687/687 ━━━━━━━━━━━━━━━━━━━━ 56s 82ms/step - accuracy: 0.8326 - loss: 0.3813 - val_accuracy: 0.8735 - val_loss:
0.3512
2751/2751 ━━━━━━━━━━━━━━━━━━━━ 122s 44ms/step - accuracy: 0.8498 - loss: 0.3553
Validation Loss: 0.35
Validation Accuracy: 0.85
```

Much better!

So, to summarize, we attempted to create a neural network capable of correctly classifying histopathologic cancer images. We started with a custom CNN model—simple but decently effective—and later implemented a more complex model that utilized transfer learning with MobileNetV2 architecture. We experimented with several different learning rates and did plenty of pre-processing on our data, such as modifying its formatting to comply with Keras library's requirements, as well as implementing data augmentation to improve our models' generalization. It should be noted that our second model, because of its dependence on MobileNetV2, could not rely on image resizing. During the training process, we split our data into train and validate sets to measure accuracy early-on and made decision accordingly. Our training process also took advantage of early stoppage to save on development time.

From everything that we tried, learning rate adjustments did not yield significant improvements. However, data augmentation and, more importantly, transfer learning definitely did. In fact, I strongly believe that by further iterating on the second model and trying other architectures in addition to MobileNetV2, we could potentially see even better results!

This following code is simply to submit our results to Kaggle. Our public score was 0.7991!

In [14]:
```python
import gc

gc.collect()

df_sample = pd.read_csv('data/sample_submission.csv')
df_sample['id'] = df_sample['id'].astype(str) + '.tif'

test_generator = ImageDataGenerator(rescale=1/255).flow_from_dataframe(
    dataframe=df_sample,
    directory='data/test',
    x_col='id',
    y_col=None,
    target_size=(96, 96),
    batch_size=4,
    class_mode=None,
    shuffle=False
)

predicts = []
for i in range(len(test_generator)):
    predicts_chunk = model_transfer.predict(test_generator[i])
    predicts.extend(predicts_chunk)
predicts = np.concatenate(predicts).ravel()
df_sample['id'] = df_sample['id'].str.replace('.tif', '')
df_sample['label'] = (predicts > 0.5).astype(int).reshape(-1)

df_sample.to_csv('submission.csv', index=False)
```

```
Found 57458 validated image filenames.
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step
2024-10-31 22:02:49.750191: E tensorflow/core/framework/node_def_util.cc:676] NodeDef mentions attribute use_unbounded_threadpool which is not in the op definition: Op<name=MapDataset; signature=input_dataset:variant, other_arguments: -> handle:variant; attr=f:func; attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1; attr=output_shapes:list(shape),min=1; attr=use_inter_op_parallelism:bool,default=true; attr=preserve_cardinality:bool,default=false; attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This may be expected if your graph generating binary is newer  than this binary. Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
2024-10-31 22:02:49.750488: E tensorflow/core/framework/node_def_util.cc:676] NodeDef mentions attribute use_unbounded_threadpool which is not in the op definition: Op<name=MapDataset; signature=input_dataset:variant, other_arguments: -> handle:variant; attr=f:func; attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1; attr=output_shapes:list(shape),min=1; attr=use_inter_op_parallelism:bool,default=true; attr=preserve_cardinality:bool,default=false; attr=force_synchronous:bool,default=false; attr=metadata:string,default=""> This may be expected if your graph generating binary is newer  than this binary. Unknown attributes will be ignored. NodeDef: {{node ParallelMapDatasetV2/_14}}
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step
```