Breast Cancer Classification with CNN

Brief description:
This notebook presents a custom convolutional neural network (CNN) built from scratch to classify Invasive Ductal Carcinoma (IDC) in breast histopathology images. The project includes comprehensive data preprocessing, addresses class imbalance, and evaluates model performance using key metrics like AUC, precision, and recall. This end-to-end pipeline demonstrates my skills in deep learning for computer vision and in this particular case medical image analysis.

Author: Juba Amari
Date: 2025-07-13
Context:
Developed as a key piece in my apprenticeship search portfolio, showcasing my ability to tackle real-world machine learning and deep learning challenges.

The original dataset is available here

And you can check out the evolution of the project on my Github page

Motivation

Breast cancer affects millions of women worldwide, and Invasive Ductal Carcinoma (IDC) is its most common and aggressive form. Detecting IDC early can save lives, but analyzing medical images manually is slow and prone to mistakes. Imagine if we could empower doctors with AI tools that quickly and accurately spot cancer signs — helping them make faster, more reliable diagnoses. This project aims to build exactly that: a custom deep learning model to support early IDC detection and make a real difference in healthcare.

Let's start! all the data was in an archive with the patients numbers as folders and inside the IDC positive and IDC negatives patches in two separate folders, we take all these out and merge them in 2 folders positive and negative in our destination path.

This section depends on local files and may not run on Colab or your local machine, but if needed, the reader can change the src and dst variables to make this work locally. The src path contains the extracted archive used as a dataset.

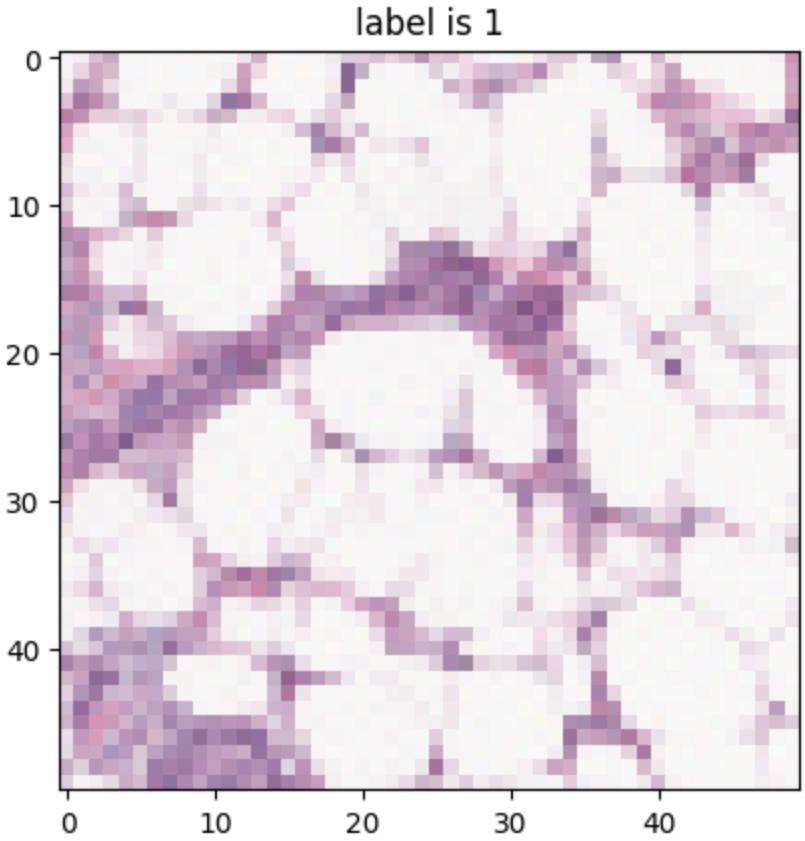
In [3]:
```
import os
import shutil
import numpy as np
import matplotlib.pyplot as plt
import cv2
np.random.seed(42)
```

```
src = '../data/raw/'
dst = '../data/clean'

os.makedirs(f'{dst}/0', exist_ok=True)
os.makedirs(f'{dst}/1', exist_ok=True)

for patient_dir in os.listdir(src):
    patient_path = os.path.join(src, patient_dir)
    if not os.path.isdir(patient_path): continue

    zero_dir = os.path.join(patient_path,"0")
    one_dir = os.path.join(patient_path,"1")

    dir_name = os.path.splitext(patient_dir)[0]
    if dir_name.isdigit():
        for file in os.listdir(zero_dir):
            if file not in os.path.join(dst, "0"):
                filepath = os.path.join(zero_dir, file)
                shutil.move(filepath, os.path.join(dst, "0") )


        for file in os.listdir(one_dir):
            if file not in os.path.join(dst, "1"):
                filepath = os.path.join(one_dir, file)
                shutil.move(filepath, os.path.join(dst, "1") )
```

now we can start spliting the data and analizing it

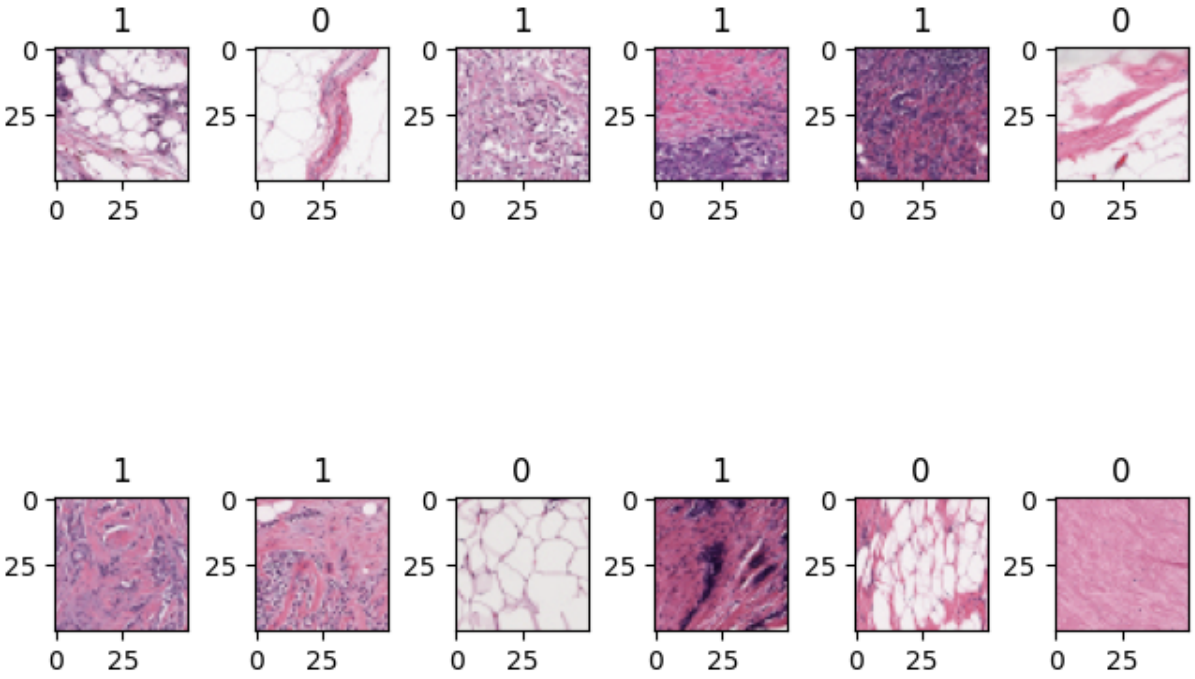
let's get an image

In [ ]:
```
def get_random_image():
    dataset = []
    for label in ["0","1"]:
        folder = os.path.join(dst,label)
        images_path = [os.path.join(folder,f) for f in os.listdir(folder)]
        dataset.extend((path, label) for path in images_path)

    idx = np.random.choice(len(dataset))
    label = dataset[idx][1]
    im = dataset[idx][0]
    im = cv2.imread(im)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)

    return im, label

def plot_random_image():
    im, label = get_random_image()
    plt.imshow(im)
    plt.title(f"label is {label}")

plot_random_image()
```



Let's now see some random images to get an idea of what our dataset looks like

In [ ]:
```
def plot_random_images(num_images=12, dataset = []):
    if len(dataset)==0:
        for label in ["0","1"]:
            folder = os.path.join(dst,label)
            images_path = [os.path.join(folder,f) for f in os.listdir(folder)]
            dataset.extend((path, label) for path in images_path)

    plt.figure(layout='constrained')

    sample = np.random.choice(len(dataset), size=num_images)
    for i,idx in enumerate(sample):
        image_path, label = dataset[idx]
        im = cv2.imread(image_path)
        im_rgb = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)

        plt.subplot(2,num_images//2, i+1)
        plt.imshow(im_rgb)
        plt.title(label)

    plt.show()

plot_random_images()
```



Now we're gonna separate all the images into training, validation and test sets, I've choosen a ratio of 70/15/15 for these, we're gonna have a list of everything and use NumPy's random.shuffle() method to shuffle them to avoid any possible data leak

In [ ]:
```
dataset = []
for label in ["0","1"]:
    folder = os.path.join(dst,label)
    images_path = [os.path.join(folder,f) for f in os.listdir(folder)]
    dataset.extend((path, label) for path in images_path)

pos_dataset=[]
neg_dataset=[]
for name, label in dataset:
    if label=="0":
        neg_dataset.append((name, label))
    else :
        pos_dataset.append((name, label))

train_ratio = 0.7
index_cut_pos = int(len(pos_dataset)*train_ratio)
index_cut_neg = int(len(neg_dataset)*train_ratio)

#print(len(pos_dataset))
#print(len(neg_dataset))


#we shuffle the dataset for more randomness
np.random.shuffle(pos_dataset)
np.random.shuffle(neg_dataset)

X_train_pos = pos_dataset[:index_cut_pos]
X_train_neg = neg_dataset[:index_cut_neg]
X_train = np.concatenate((X_train_pos, X_train_neg))
```

```
X_val_pos = pos_dataset[index_cut_pos:]
X_val_neg = neg_dataset[index_cut_neg:]


validation_pos_cut = len(X_val_pos)//2
validation_neg_cut = len(X_val_neg)//2

X_test = np.concatenate([X_val_pos[validation_pos_cut:], X_val_neg[validation_neg_c
X_val = np.concatenate([X_val_pos[:validation_pos_cut], X_val_neg[:validation_neg_c
```

```
X_train size: 194266
Total string length: 365
['../data/clean\\1\\12868_idx5_x1751_y1801_class1.png' '1']
```

In [5]:
```
#paths
train_path = os.path.join(dst,"train")
val_path = os.path.join(dst, "validation")
test_path = os.path.join(dst, "test")
```

In [13]:
```
def copy_data(data, destination_path):
    print(f"Copying {data} set with {len(data)} images...")
    for i, (name, label) in enumerate(data):
        #for testing purposes
        #if i>=10000:
        #    break

        x_dir = os.path.join(destination_path,label)
        os.makedirs(x_dir, exist_ok=True)

        filename = os.path.basename(name)
        dst_path = os.path.join(x_dir,filename)
        shutil.copyfile(name,dst_path)

        if i%500==0:
            print(f'{i}/{len(data)} images have been copied')
    print('everything has been copied')

#copy_data(X_train,train_path)
#copy_data(X_val,val_path)
#copy_data(X_test, test_path)
```

Problem : this is a naive approach of randomly distributing patches without taking into account the whole slide, this may cause data leakage as the patches are not independant and the model may see in validation some patches from the same patient seen in training.

As the patches are correlated we will have some overly optimistic results, we will redo the data split, this time spliting by patients, then extracting all positive and negative patches, this may or may not cause some problematic distribution of + and - patches because not all patients have the same number of + patches, and we will adress this problem if we are faced to it.

But as of right now, we prefer avoiding data leakage.

We start by creating a python dictionary where we have patient ids as keys and all their patches in a list as values, these patches are tuples of the form (image, 1 or 0).

In [94]:
```
patients = {}

for label in ["0", "1"]:
    current_dir = os.path.join(dst, label)

    for patch in os.listdir(current_dir):
        patch_path = os.path.join(current_dir, patch)
        patch_name = os.path.basename(patch)
        id = patch_name.split('_')[0]

        if id not in patients:
            patients[id] = []

        patients[id].append((patch_path, label))

#now we have a dictionary with the patients and their patches
patch_count = {k: len(v) for k, v in patients.items()}
print(f"The number of patiens is {len(patch_count)}")
print(f"the total number of patches is {sum(patch_count.values())}")
print(f"The average number of patches per patient is {sum(patch_count.values())/len
```

```
The number of patiens is 279
the total number of patches is 277524
The average number of patches per patient is 994.7096774193549
```

perfect, now we split by patients with that previous 70/15/15 ratio then copy the data using our previous copying function.

In [37]:
```
def split_data(data, train_ratio=0.7, seed=42):
    np.random.seed(seed)
    patient_ids = list(data.keys())
    np.random.shuffle(patient_ids)
    split_index = int(len(patient_ids)*train_ratio)

    training = set(patient_ids[:split_index])
    val = patient_ids[split_index:]
    test_split_index = len(val)//2
    testing = set(val[test_split_index:])
    val = set(val[:test_split_index])
    #we convert to sets for faster in checks


    X_train = []
    X_val = []
    X_test = []

    for k, v in data.items():
        if k in training:
            X_train.extend(patch for patch in v)
        elif k in val :
            X_val.extend(patch for patch in v)
        else:
```

```
            X_test.extend(patch for patch in v)

    return np.array(X_train), np.array(X_val), np.array(X_test)
```

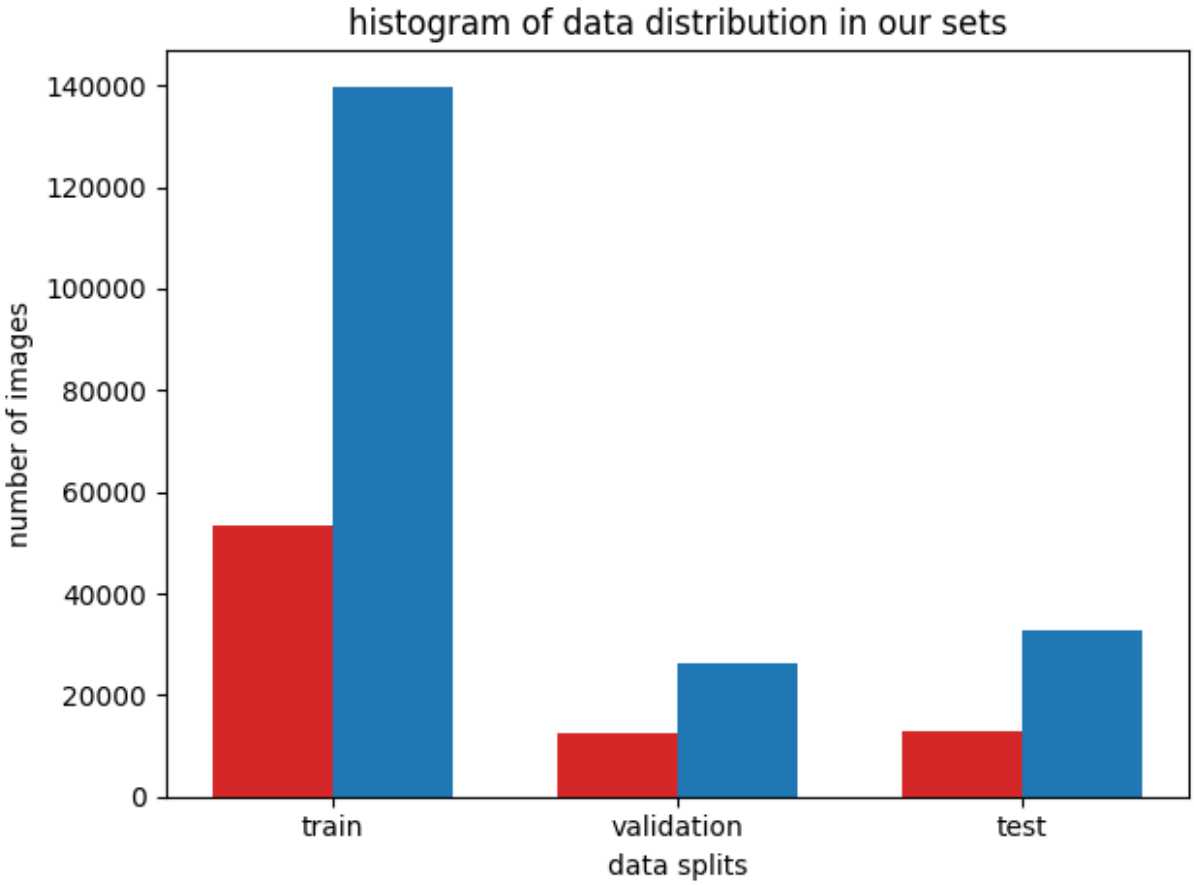
In [ ]:
```
X_train, X_val, X_test = split_data(patients)

copy_data(X_train, train_path)
copy_data(X_val, val_path)
copy_data(X_test, test_path)
```

Here I've cleared the above cell's output for better readability but all the files were in fact copied

Let's verify we have some equal distribution of the positive and negative images between our different folders

In [41]:
```
plt.figure(layout='constrained')


train_1 = len(os.listdir(os.path.join(dst,"train/1")))
train_0 = len(os.listdir(os.path.join(dst,"train/0")))
val_1 = len(os.listdir(os.path.join(dst,"validation/1")))
val_0 = len(os.listdir(os.path.join(dst,"validation/0")))
test_1 = len(os.listdir(os.path.join(dst,"test/1")))
test_0 = len(os.listdir(os.path.join(dst,"test/0")))

print(type(train_1), type(val_1), type(test_1))
print(type(train_0), type(val_0), type(test_0))

labels = ['train','validation','test']
x= np.arange(len(labels))
bar_width = 0.35
print(x)

positive_count = [train_1,val_1,test_1]
negative_count = [train_0,val_0,test_0]

plt.bar(x - bar_width/2, positive_count, width=bar_width, color='#d62728' )
plt.bar(x + bar_width/2, negative_count, width=bar_width, color='#1f77b4')

plt.xticks(x, labels)
plt.ylabel('number of images')
plt.xlabel('data splits')
plt.title('histogram of data distribution in our sets')
plt.tight_layout()

plt.show()
```

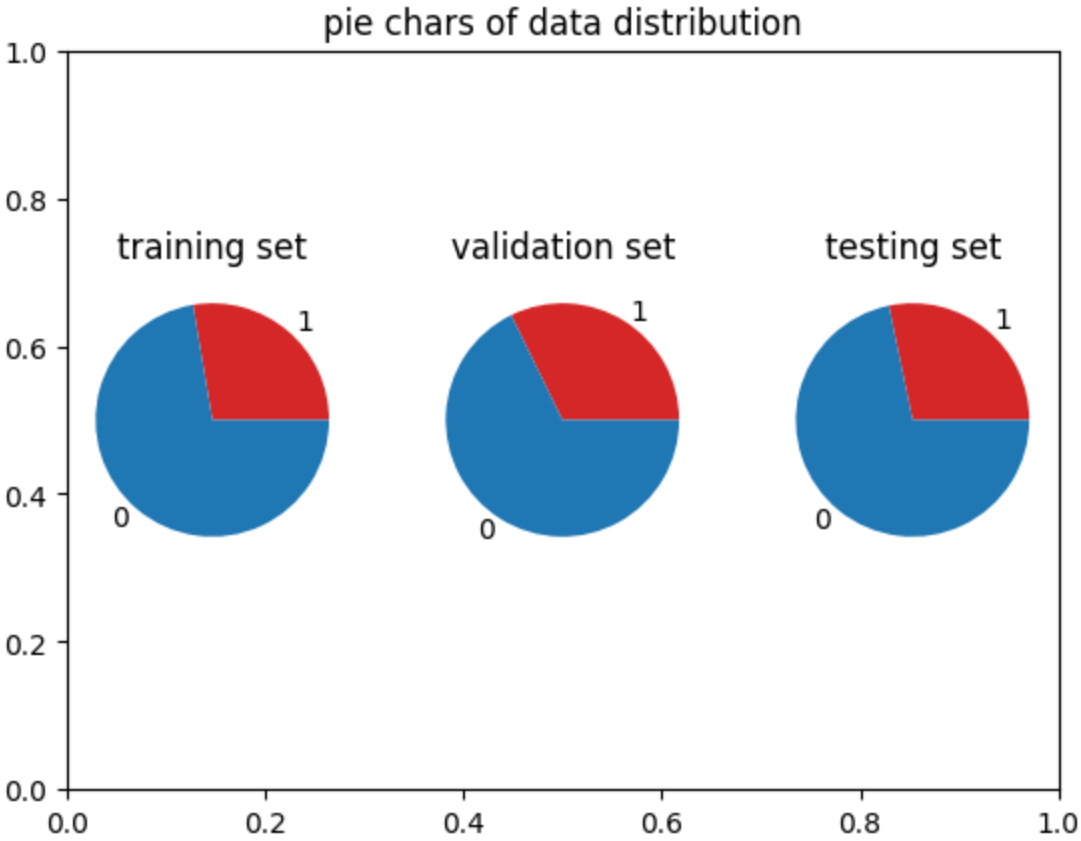
```
<class 'int'> <class 'int'> <class 'int'>
<class 'int'> <class 'int'> <class 'int'>
[0 1 2]
C:\Users\PC\AppData\Local\Temp\ipykernel_19816\126570909.py:29: UserWarning: The fig
ure layout has changed to tight
  plt.tight_layout()
```



Some pie charts to better see the proportions, if everything is fine they should all look the same

In [42]:
```
plt.figure()
plt.title('pie chars of data distribution')

plt.subplot(1,3,1)
plt.pie(x=[train_1, train_0], labels=['1','0'], colors=['#d62728', '#1f77b4'])
plt.title('training set')
plt.subplot(1,3,2)
plt.pie(x=[val_1, val_0], labels=['1','0'], colors=['#d62728', '#1f77b4'])
plt.title('validation set')
plt.subplot(1,3,3)
plt.pie(x=[test_1, test_0], labels=['1','0'], colors=['#d62728', '#1f77b4'])
plt.title('testing set')
```

Out[42]:
```
Text(0.5, 1.0, 'testing set')
```



We have about the same proportions in the different folders, so no problem on this side. Now that the data is well organised and splited into the 3 sets we need, we can start building our prediction model, using tensorflow and keras's high level API, we will create our own Convolutional Neural Network architecture from scratch and tweak it until we get the best results.

But before we start model building, we need to preprocess the data we have.

In [2]:
```
#imports needed
import os
import shutil
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.preprocessing import image_dataset_from_directory
from tensorflow.keras import layers
import pydot
import graphviz
import pandas as pd
```

In [ ]:
```
#for reproducability
def set_seed(seed=42):
    np.random.seed(seed)
    tf.random.set_seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    os.environ['TF_DETERMINISTIC_OPS'] = '1'
set_seed()
```

```
#because we already organised the data into clear folders, we can directly use this
X_train = image_dataset_from_directory(
    train_path,
    labels='inferred',
    label_mode='binary',
    color_mode='rgb',
    batch_size=None,
    image_size=(50,50),
    shuffle=True,
    interpolation='nearest'
)

X_val = image_dataset_from_directory(
    val_path,
    labels='inferred',
    label_mode='binary',
    color_mode='rgb',
    batch_size=None,
    image_size=(50,50),
    shuffle=True,
    interpolation='nearest'
)
#autotune for when we map the data
AUTOTUNE = tf.data.experimental.AUTOTUNE
def preprocess(image, label):
    image = tf.image.convert_image_dtype(image, tf.float32) #we normalize the tenso
    return image, label
```

```
Found 193299 files belonging to 2 classes.
Found 38496 files belonging to 2 classes.
```

We check the proportion of positives and negatives to make sure everything was loaded correctly

In [7]:
```
from collections import Counter

counter = Counter()

for _, label in X_train:
    label_value = int(label.numpy())
    counter[label_value] += 1

print(counter)
```

```
C:\Users\PC\AppData\Local\Temp\ipykernel_1404\1742459193.py:6: DeprecationWarning: C
onversion of an array with ndim > 0 to a scalar is deprecated, and will error in fut
ure. Ensure you extract a single element from your array before performing this oper
ation. (Deprecated NumPy 1.25.)
  label_value = int(label.numpy())
Counter({0: 139851, 1: 53448})
```

A ratio of about 1:2.6, we are dealing with a significant class imbalance, we will take that into account in the class weights of the loss function in order for the model not to get lazy and predict all negatives, our pricipal objective here is a high Recall i.e catching all posives and avoiding false negatives.

Let's now write the data augmentation pipeline, this will help our model generalize quicker but we won't be changing anything linked to the images's colors, brightness, saturation... because it may lose some important information we want to keep, just flips and rotations should do the job.

In [53]:
```
data_augmentation = tf.keras.Sequential([
    layers.RandomFlip(mode='horizontal_and_vertical'),
    layers.RandomRotation(factor=(-0.3, 0.3)),
]) #we don't alter the colors because on a former try it used to give very optimist

def augmentation(image, label):
    return (data_augmentation(image), label)
```

We can finally preprocess with everything we have defined

In [54]:
```
print(tf.data.experimental.cardinality(X_train).numpy())
print(tf.data.experimental.cardinality(X_val).numpy())

X_train = (
    X_train
    .map(preprocess, num_parallel_calls=AUTOTUNE, deterministic=True)
    .map(augmentation , num_parallel_calls=AUTOTUNE, deterministic=True)
    .shuffle(buffer_size=1000)
    .batch(32)
    .cache()
    .prefetch(buffer_size=AUTOTUNE)
)
X_val = (
    X_val
    .map(preprocess, num_parallel_calls=AUTOTUNE, deterministic=True)
    .batch(32)
    .cache()
    .prefetch(buffer_size=AUTOTUNE)
)

print(tf.data.experimental.cardinality(X_train).numpy())
print(tf.data.experimental.cardinality(X_val).numpy())
```

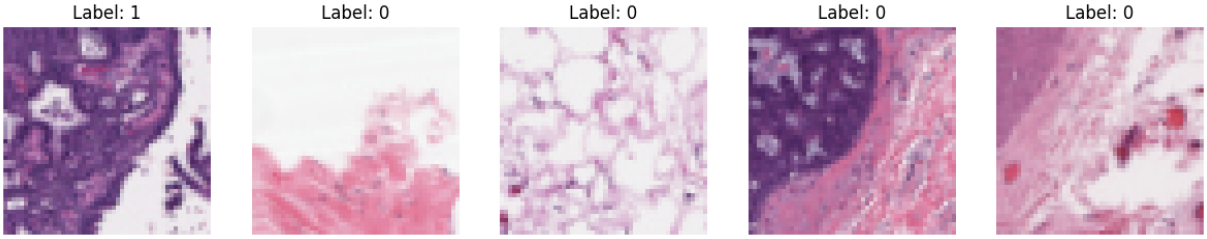
```
193299
38496
6041
1203
```

We check some of the images to make sure nothing broke in augmentation

In [68]:
```
for images, labels in X_train.take(1):
    fig, axes = plt.subplots(1, 5, figsize=(15, 5))
```

```
    for i in range(5):
        axes[i].imshow(images[i].numpy())
        axes[i].set_title(f"Label: {int(labels[i].numpy())}")
        axes[i].axis('off')
    plt.show()
```

```
C:\Users\PC\AppData\Local\Temp\ipykernel_1404\2739197042.py:5: DeprecationWarning: C
onversion of an array with ndim > 0 to a scalar is deprecated, and will error in fut
ure. Ensure you extract a single element from your array before performing this oper
ation. (Deprecated NumPy 1.25.)
  axes[i].set_title(f"Label: {int(labels[i].numpy())}")
```



The data preprocessing is done, now we'll start the model building, first step is feature extraction, then the prediction model.

For feature extraction we will use 3 layers of keras's Conv2D with filters starting at 32, batch normalization, ReLu activation and maximum pooling. (We use GlobalAveragePooling instead of flatten because it gives better results after testing both) Then, the body will have three 64 units dense layers, ending in a sigmoid activation to get a probability.

In [23]:
```
feature_extraction = [
        #input layer will be size 50x50x3
        layers.Conv2D(input_shape=(50, 50, 3), filters=32, kernel_size=(3,3), strid
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.MaxPool2D(pool_size=(2,2), strides=None, padding='valid'), #25x25x32

        layers.Conv2D(filters=64, kernel_size=(3,3), use_bias=False), #25x25x64
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.MaxPool2D(pool_size=(2,2), strides=None, padding='valid'), #12x12x64

        layers.Conv2D(filters=128, kernel_size=(3,3), use_bias=False), #12x12x128
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.MaxPool2D(pool_size=(2,2), strides=None, padding='valid'), #6x6x128

        layers.GlobalAveragePooling2D()
    ]

model_body = [
        layers.Dense(units=64, input_shape=[4608], kernel_regularizer='l2'), #L2 pr
        layers.BatchNormalization(), #to stabilize training
        layers.Activation('relu'),
        layers.Dropout(0.5), #after a first test at 30% dropout I noticed it wasn't

        layers.Dense(units=64, kernel_regularizer='l2'),
```

```
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.Dropout(0.5),

        layers.Dense(units=64, kernel_regularizer='l2'),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.Dropout(0.5),

        layers.Dense(units=1, activation='sigmoid')
    ]
```

Now we assemble the model, compile and we can generate a graphical representation.

As a metric to evaluate, Recall is the most important, because we are detecting cancer cases, what we want to avoid the most are false negatives, as the consequences can be very bad while a false positive may only require some more testing.

In [24]:
```
model = tf.keras.Sequential([
    *feature_extraction,
    *model_body
])

model.compile(
    optimizer='adam',
    loss=tf.keras.losses.BinaryFocalCrossentropy(gamma=2.0, from_logits=False),
    metrics=['binary_accuracy',
             tf.keras.metrics.Recall(name='recall'),
             tf.keras.metrics.Precision(name='precision'),
             tf.keras.metrics.AUC(name='auc')
         ]
)

#we add a callback
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor= "val_loss",
    min_delta= 0.01,
    patience= 5,
    mode='min',
    restore_best_weights=True,
    start_from_epoch=10
)
```

Here's a quick overview of our model's structure

In [25]:
```
model.build()
model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 50, 50, 32)	864
batch_normalization_12 (BatchNormalization)	(None, 50, 50, 32)	128
activation_12 (Activation)	(None, 50, 50, 32)	0
max_pooling2d_6 (MaxPooling2D)	(None, 25, 25, 32)	0
conv2d_7 (Conv2D)	(None, 23, 23, 64)	18,432
batch_normalization_13 (BatchNormalization)	(None, 23, 23, 64)	256
activation_13 (Activation)	(None, 23, 23, 64)	0
max_pooling2d_7 (MaxPooling2D)	(None, 11, 11, 64)	0
conv2d_8 (Conv2D)	(None, 9, 9, 128)	73,728
batch_normalization_14 (BatchNormalization)	(None, 9, 9, 128)	512
activation_14 (Activation)	(None, 9, 9, 128)	0
max_pooling2d_8 (MaxPooling2D)	(None, 4, 4, 128)	0
global_average_pooling2d_2 (GlobalAveragePooling2D)	(None, 128)	0
dense_8 (Dense)	(None, 64)	8,256
batch_normalization_15 (BatchNormalization)	(None, 64)	256
activation_15 (Activation)	(None, 64)	0
dropout_6 (Dropout)	(None, 64)	0
dense_9 (Dense)	(None, 64)	4,160
batch_normalization_16 (BatchNormalization)	(None, 64)	256
activation_16 (Activation)	(None, 64)	0
dropout_7 (Dropout)	(None, 64)	0
dense_10 (Dense)	(None, 64)	4,160
batch_normalization_17 (BatchNormalization)	(None, 64)	256
activation_17 (Activation)	(None, 64)	0
dropout_8 (Dropout)	(None, 64)	0

dense_11 (Dense)	(None, 1)	65

Total params: 111,329 (434.88 KB)

Trainable params: 110,497 (431.63 KB)

Non-trainable params: 832 (3.25 KB)

before we fit the model, we will write a callback for saving our model every epoch so we can let the training go and comeback later

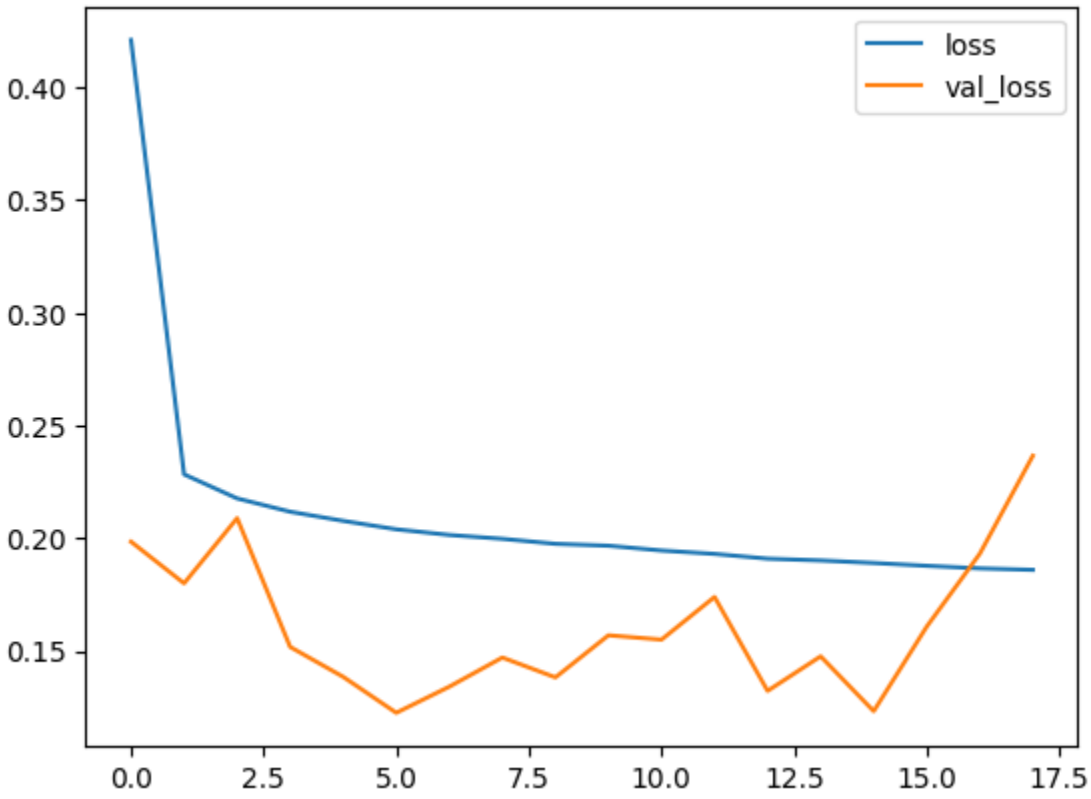
In [74]:
```
os.makedirs("../model/model2/checkpoints", exist_ok=True)

checkpoint = tf.keras.callbacks.ModelCheckpoint(
    "../model/model2/checkpoints/{epoch:02d}-{val_loss:.2f}.keras",
    monitor='val_loss',
    verbose=1,
    save_best_only=False,
    save_weights_only=False,
    save_freq='epoch'
)
```

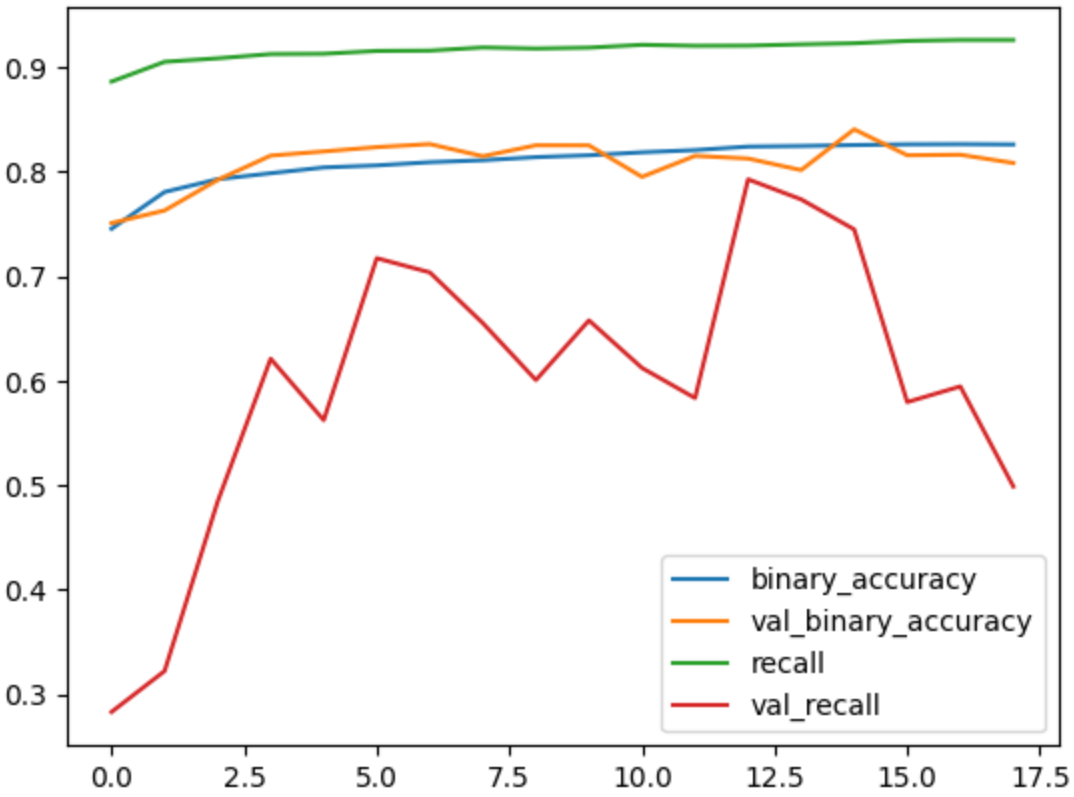
We can now fit our custom ConvNet !

In [75]:
```
history = model.fit(
    X_train,
    epochs=60,
    verbose=1,
    callbacks=[early_stopping, checkpoint],
    validation_split=0,
    validation_data=X_val,
    validation_steps=None,
    class_weight={0:1.0, 1:5},
    validation_freq=1
)
```

```
Epoch 17/60
6040/6041 ━━━━━━━━━━━━━━━━━━━━ 0s 73ms/step - auc: 0.9359 - binary_accuracy: 0.8275
- loss: 0.1860 - precision: 0.6276 - recall: 0.9250
Epoch 17: saving model to ../model/model2/checkpoints/17-0.19.keras
6041/6041 ━━━━━━━━━━━━━━━━━━━━ 452s 75ms/step - auc: 0.9359 - binary_accuracy: 0.827
5 - loss: 0.1860 - precision: 0.6276 - recall: 0.9250 - val_auc: 0.8785 - val_binary
_accuracy: 0.8162 - val_loss: 0.1931 - val_precision: 0.7834 - val_recall: 0.5944
Epoch 18/60
6040/6041 ━━━━━━━━━━━━━━━━━━━━ 0s 72ms/step - auc: 0.9371 - binary_accuracy: 0.8271
- loss: 0.1846 - precision: 0.6267 - recall: 0.9260
Epoch 18: saving model to ../model/model2/checkpoints/18-0.24.keras
6041/6041 ━━━━━━━━━━━━━━━━━━━━ 448s 74ms/step - auc: 0.9371 - binary_accuracy: 0.827
1 - loss: 0.1846 - precision: 0.6267 - recall: 0.9260 - val_auc: 0.8350 - val_binary
_accuracy: 0.8083 - val_loss: 0.2366 - val_precision: 0.8426 - val_recall: 0.4986
```

In [ ]:
```
history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot()
history_df.loc[:, ['binary_accuracy', 'val_binary_accuracy', 'recall', 'val_recall'
```





So, we started to overfit at about epoch 13, our early stopping then stopped the training, the curves look good but not incredible. We may now evaluate the model on testing data to see its live performance.

In [86]:
```
X_test = image_dataset_from_directory(
    test_path,
    labels='inferred',
    label_mode='binary',
    color_mode='rgb',
    image_size=(50,50),
    batch_size=None,
    shuffle=True,
    interpolation='nearest'
)

X_test = (
    X_test
    .map(preprocess, num_parallel_calls=AUTOTUNE, deterministic=True)
    .batch(64)
    .cache()
    .prefetch(AUTOTUNE)
)
```

```
Found 45729 files belonging to 2 classes.
```

In [ ]:
```
model = tf.keras.models.load_model('../model/model2/checkpoints/18-0.24.keras')
```

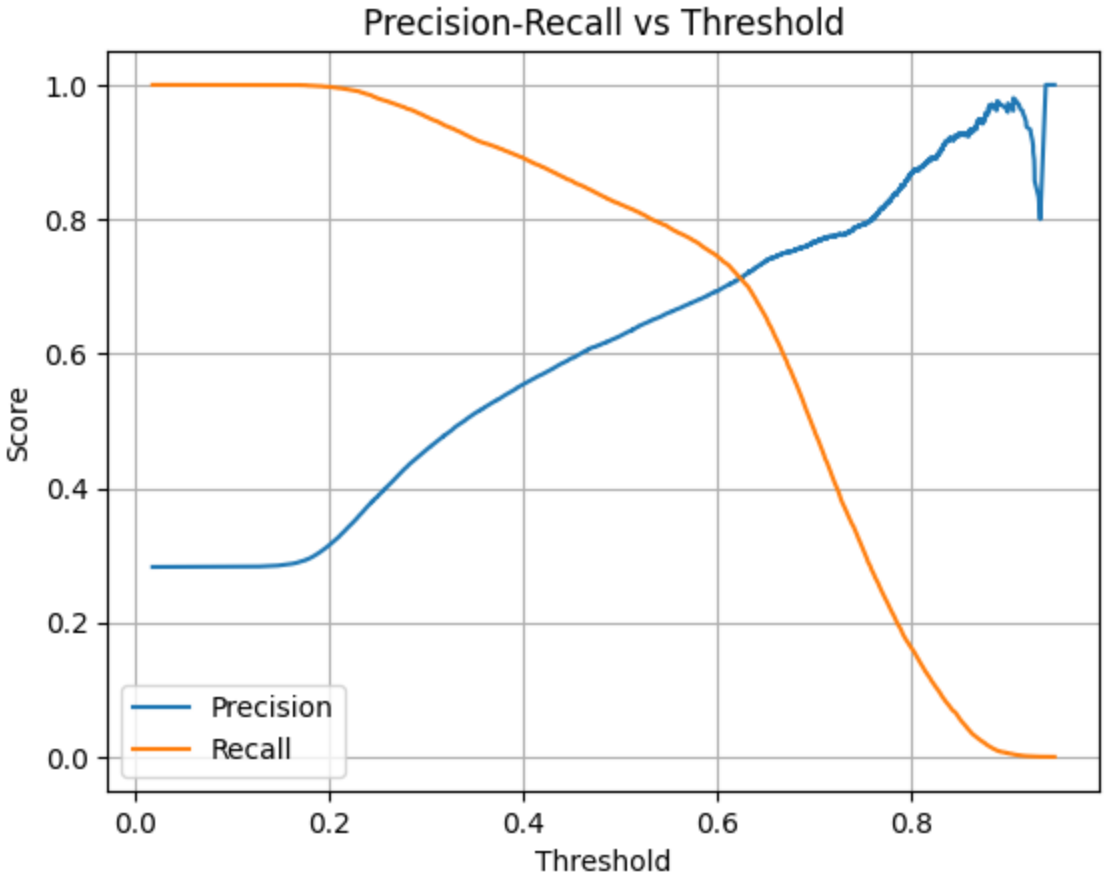
In [81]:
```
test_history = model.evaluate(
    X_test,
    batch_size=None,
```

```
    verbose=1,
    steps=None,
    return_dict=True
)
```

```
715/715 ━━━━━━━━━━━━━━━━━━━━ 95s 133ms/step - auc: 0.8887 - binary_accuracy: 0.8114
- loss: 0.1362 - precision: 0.6256 - recall: 0.8251
```

We may later want to change our threshold for maximizing recall while minimizing the loss in precision, as recall is the most important metric for us here

In [ ]:
```
from sklearn.metrics import precision_recall_curve
y_true = []
y_score = []

for images, labels in X_test:
    probs = model.predict(images, verbose=0)
    y_score.extend(probs.flatten())
    y_true.extend(labels.numpy().flatten())

y_true = np.array(y_true)
y_score = np.array(y_score)

precision, recall, thresholds = precision_recall_curve(y_true, y_score)

plt.figure()
plt.plot(thresholds, precision[:-1], label='Precision')
plt.plot(thresholds, recall[:-1], label='Recall')
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.title('Precision-Recall vs Threshold')
plt.legend()
plt.grid(True)
plt.show()
```



As we see here a good threshold would around 0.65, but we can make an even better model, this one seems to overcompensate with the weight classes, we'll do some adjustments, put the class_weight at {0:1, 1:4.2}, halve the batch size for training to maybe achieve better generalization, change the minimum variation of our early stoping to give the model more time to converge if needed and add a callback to make it easier to escape early plateau's.

We will continue all this on Google Colab for faster training because we have access to GPU acceleration, so some file paths may change in the next part of this notebook because I am merging both notebooks at the end.

In [1]:
```
#imports
import os
import numpy as np
import pandas as pd
import cv2
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.preprocessing import image_dataset_from_directory
import sklearn
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_curve
from sklearn.metrics import f1_score
from sklearn.metrics import precision_recall_curve
```

In [10]:
```
print(tf.config.list_physical_devices('GPU'))
```

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

In [ ]:
```
#paths
ds_path = '/kaggle/input/idc-histology-cleaned-and-splitted'

train_path= '/kaggle/input/idc-histology-cleaned-and-splitted/train'
val_path= '/kaggle/input/idc-histology-cleaned-and-splitted/validation'
test_path= '/kaggle/input/idc-histology-cleaned-and-splitted/test'
```

Same preprocessing as before

In [ ]:
```
#for reproducability
def set_seed(seed=42):
    np.random.seed(seed)
    tf.random.set_seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    os.environ['TF_DETERMINISTIC_OPS'] = '1'
set_seed()

X_train = image_dataset_from_directory(
    train_path,
    labels='inferred',
    label_mode='binary',
    color_mode='rgb',
    batch_size=None,
    image_size=(50,50),
    shuffle=True,
    interpolation='nearest'
)

X_val = image_dataset_from_directory(
    val_path,
    labels='inferred',
    label_mode='binary',
    color_mode='rgb',
    batch_size=None,
    image_size=(50,50),
```

```
    shuffle=True,
    interpolation='nearest'
)
```

```
Found 193299 files belonging to 2 classes.
I0000 00:00:1752358281.510962     195 gpu_device.cc:2022] Created device /job:localh
ost/replica:0/task:0/device:GPU:0 with 15513 MB memory:  -> device: 0, name: Tesla P
100-PCIE-16GB, pci bus id: 0000:00:04.0, compute capability: 6.0
Found 38496 files belonging to 2 classes.
```

In [5]:
```
#preprocessing
AUTOTUNE = tf.data.experimental.AUTOTUNE
def preprocess(image, label):
    image = tf.image.convert_image_dtype(image, tf.float32)
    return image, label

data_augmentation = tf.keras.Sequential([
    layers.RandomFlip(mode='horizontal_and_vertical'),
    layers.RandomRotation(factor=(-0.3, 0.3)),
])

def augmentation(image, label):
    return (data_augmentation(image), label)
```

In [ ]:
```
print(tf.data.experimental.cardinality(X_train).numpy())
print(tf.data.experimental.cardinality(X_val).numpy())

X_train = (
    X_train
    .map(preprocess, num_parallel_calls=AUTOTUNE, deterministic=True)
    .map(augmentation , num_parallel_calls=AUTOTUNE, deterministic=True)
    .shuffle(buffer_size=1000)
    .batch(16)#here we're using some smaller batches
    .cache()
    .prefetch(buffer_size=AUTOTUNE)
)
X_val = (
    X_val
    .map(preprocess, num_parallel_calls=AUTOTUNE, deterministic=True)
    .batch(16)
    .cache()
    .prefetch(buffer_size=AUTOTUNE)
)

print(tf.data.experimental.cardinality(X_train).numpy())
print(tf.data.experimental.cardinality(X_val).numpy())
```

```
193299
38496
12082
2406
```

In [ ]:
```
feature_extraction = [
        layers.Conv2D(filters=32, kernel_size=(3,3), strides=1, padding='same', use
        layers.BatchNormalization(),
        layers.Activation('relu'),
```

```
        layers.MaxPool2D(pool_size=(2,2), strides=None, padding='valid'),

        layers.Conv2D(filters=64, kernel_size=(3,3), use_bias=False),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.MaxPool2D(pool_size=(2,2), strides=None, padding='valid'),

        layers.Conv2D(filters=128, kernel_size=(3,3), use_bias=False),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.MaxPool2D(pool_size=(2,2), strides=None, padding='valid'),

        layers.GlobalAveragePooling2D()
    ]

model_body = [
        layers.Dense(units=64, input_shape=[4608], kernel_regularizer='l2'),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.Dropout(0.5),

        layers.Dense(units=64, kernel_regularizer='l2'),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.Dropout(0.5),

        layers.Dense(units=64, kernel_regularizer='l2'),
        layers.BatchNormalization(),
        layers.Activation('relu'),
        layers.Dropout(0.5),

        layers.Dense(units=1, activation='sigmoid')
    ]
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarni
ng: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequent
ial models, prefer using an `Input(shape)` object as the first layer in the model in
stead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [18]:
```
model = tf.keras.Sequential([
    *feature_extraction,
    *model_body
])

model.compile(
    optimizer='adam',
    loss=tf.keras.losses.BinaryFocalCrossentropy(gamma=2.0, from_logits=False),
    metrics=['binary_accuracy',
            tf.keras.metrics.Recall(name='recall'),
            tf.keras.metrics.Precision(name='precision'),
            tf.keras.metrics.AUC(name='auc')
        ]
)
```

```
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor= "val_loss",
    min_delta= 0.001,
    patience= 5,
    mode='min',
    restore_best_weights=True,
    start_from_epoch=10
)

checkpoint = tf.keras.callbacks.ModelCheckpoint(
    "/kaggle/working/model/checkpoints/{epoch:02d}-{val_loss:.2f}.keras",
    monitor='val_loss',
    verbose=1,
    save_best_only=False,
    save_weights_only=False,
    save_freq='epoch'
)

reduce_lr =tf.keras.callbacks.ReduceLROnPlateau(
    monitor="val_loss", factor=0.5, patience=3, min_lr=1e-6, verbose=1
)
```

In [25]:
```
history = model.fit(
    X_train,
    epochs=60,
    verbose=1,
    callbacks=[early_stopping, checkpoint, reduce_lr],
    validation_split=0,
    validation_data=X_val,
    validation_steps=None,
    class_weight={0:1.0, 1:4.2},
    validation_freq=1
)

model.save("../model/model4/model.keras")
```

```
3 - loss: 0.1557 - precision: 0.6497 - recall: 0.9243
Epoch 21: saving model to ../model/model4/checkpoints/21-0.11.keras
12082/12082 ━━━━━━━━━━━━━━━━━━━━ 200s 17ms/step - auc: 0.9390 - binary_accuracy: 0.8
413 - loss: 0.1557 - precision: 0.6497 - recall: 0.9243 - val_auc: 0.9059 - val_bina
ry_accuracy: 0.8528 - val_loss: 0.1065 - val_precision: 0.7566 - val_recall: 0.8014
- learning_rate: 1.2500e-04
```

Due to collab crashing, the model training stopped, but we can resume it on Kaggle from epoch 21 where it ended

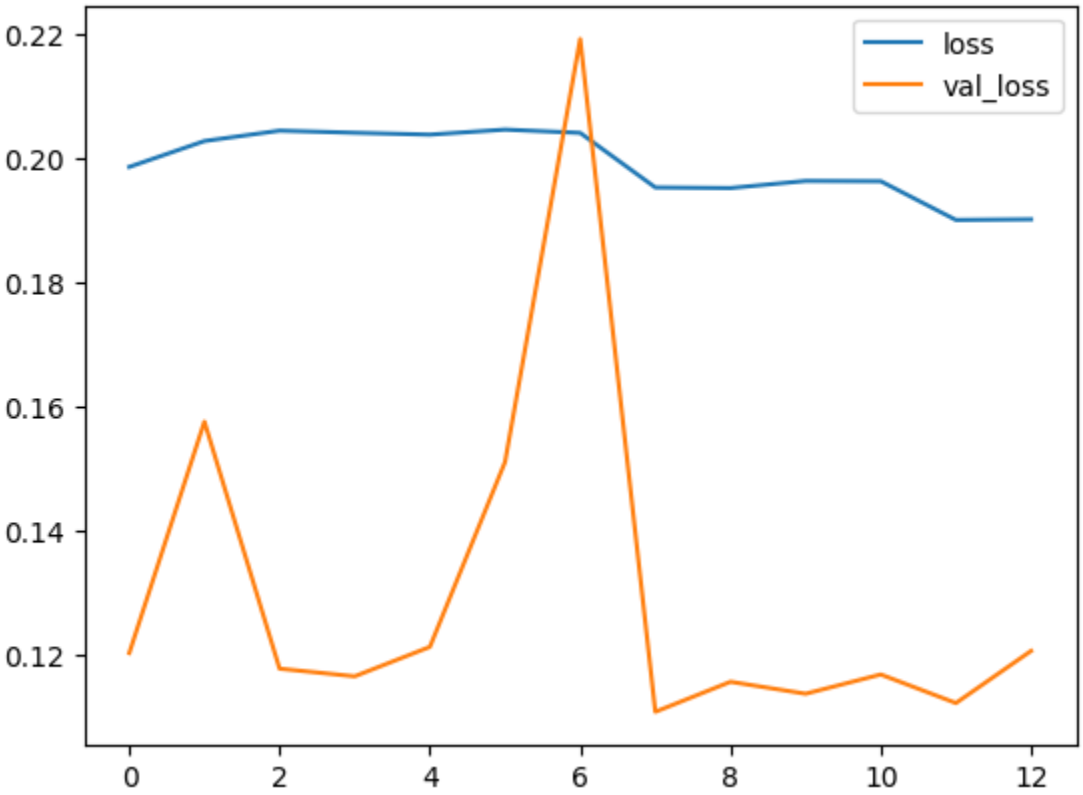
In [20]:
```
model= tf.keras.models.load_model('/kaggle/input/cnn_model_bc/keras/default/1/model

history = model.fit(
    X_train,
    epochs=60,
    verbose=1,
    callbacks=[early_stopping, checkpoint, reduce_lr],
    initial_epoch=21,
    validation_split=0,
    validation_data=X_val,
    validation_steps=None,
    class_weight={0:1.0, 1:4.2},
    validation_freq=1
)

model.save('/kaggle/working/model/model_v5.keras')
```

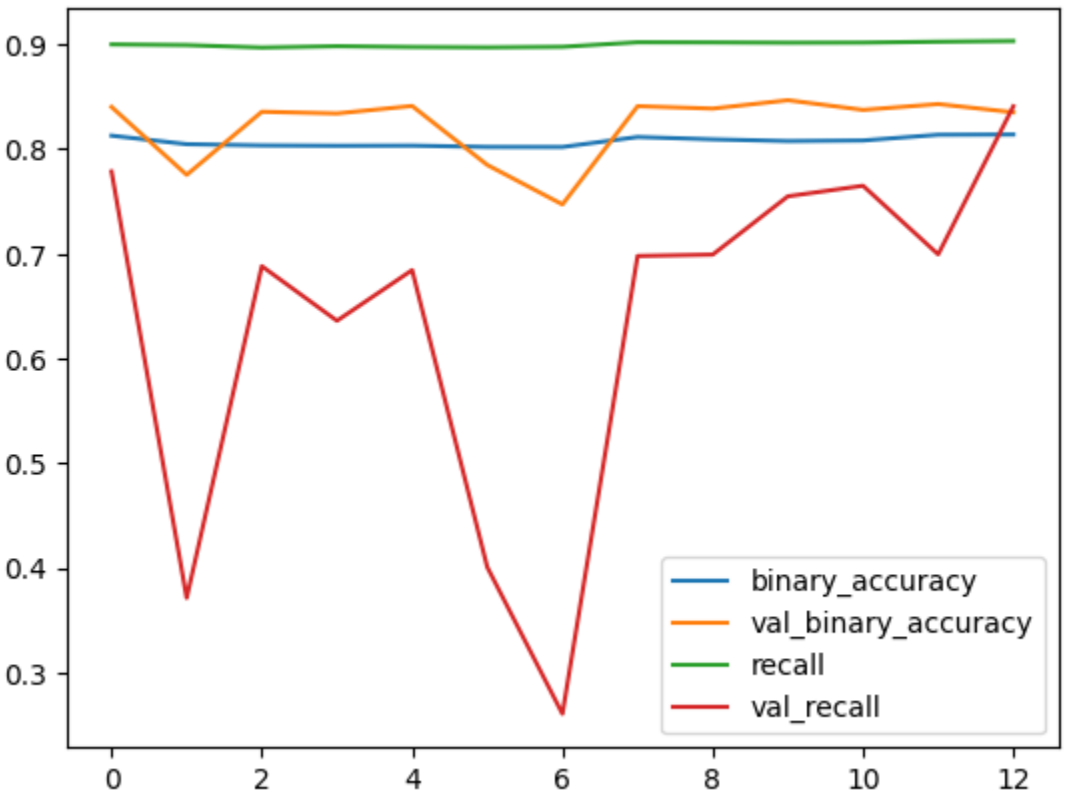
```
Epoch 22/60
2025-07-12 22:44:55.371271: E tensorflow/core/framework/node_def_util.cc:676] NodeDe
f mentions attribute use_unbounded_threadpool which is not in the op definition: Op<
name=MapDataset; signature=input_dataset:variant, other_arguments: -> handle:variant
; attr=f:func; attr=Targuments:list(type),min=0; attr=output_types:list(type),min=1;
attr=output_shapes:list(shape),min=1; attr=use_inter_op_parallelism:bool,default=tru
e; attr=preserve_cardinality:bool,default=false; attr=force_synchronous:bool,default
=false; attr=metadata:string,default=""> This may be expected if your graph generati
ng binary is newer  than this binary. Unknown attributes will be ignored. NodeDef:
{{node ParallelMapDatasetV2/_5}}
```

```
- learning_rate: 0.0010
Epoch 29/60
12082/12082 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - auc: 0.9157 - binary_accuracy: 0.814
2 - loss: 0.1967 - precision: 0.6110 - recall: 0.9014
Epoch 29: saving model to /kaggle/working/model/checkpoints/29-0.11.keras
12082/12082 ━━━━━━━━━━━━━━━━━━━━ 134s 11ms/step - auc: 0.9157 - binary_accuracy: 0.8
142 - loss: 0.1967 - precision: 0.6110 - recall: 0.9014 - val_auc: 0.8972 - val_bina
ry_accuracy: 0.8409 - val_loss: 0.1107 - val_precision: 0.7849 - val_recall: 0.6979
- learning_rate: 5.0000e-04
Epoch 30/60
12081/12082 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - auc: 0.9166 - binary_accuracy: 0.811
8 - loss: 0.1950 - precision: 0.6076 - recall: 0.9014
Epoch 30: saving model to /kaggle/working/model/checkpoints/30-0.12.keras
12082/12082 ━━━━━━━━━━━━━━━━━━━━ 137s 11ms/step - auc: 0.9166 - binary_accuracy: 0.8
118 - loss: 0.1950 - precision: 0.6076 - recall: 0.9014 - val_auc: 0.8874 - val_bina
ry_accuracy: 0.8386 - val_loss: 0.1156 - val_precision: 0.7778 - val_recall: 0.6992
- learning_rate: 5.0000e-04
Epoch 31/60
12080/12082 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - auc: 0.9151 - binary_accuracy: 0.808
5 - loss: 0.1968 - precision: 0.6030 - recall: 0.8995
Epoch 31: saving model to /kaggle/working/model/checkpoints/31-0.11.keras
12082/12082 ━━━━━━━━━━━━━━━━━━━━ 137s 11ms/step - auc: 0.9151 - binary_accuracy: 0.8
085 - loss: 0.1968 - precision: 0.6030 - recall: 0.8995 - val_auc: 0.9045 - val_bina
ry_accuracy: 0.8466 - val_loss: 0.1136 - val_precision: 0.7660 - val_recall: 0.7549
- learning_rate: 5.0000e-04
Epoch 32/60
12078/12082 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - auc: 0.9154 - binary_accuracy: 0.810
0 - loss: 0.1965 - precision: 0.6048 - recall: 0.9017
Epoch 32: saving model to /kaggle/working/model/checkpoints/32-0.12.keras

Epoch 32: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
12082/12082 ━━━━━━━━━━━━━━━━━━━━ 137s 11ms/step - auc: 0.9154 - binary_accuracy: 0.8
100 - loss: 0.1965 - precision: 0.6048 - recall: 0.9017 - val_auc: 0.8993 - val_bina
ry_accuracy: 0.8373 - val_loss: 0.1167 - val_precision: 0.7396 - val_recall: 0.7649
- learning_rate: 5.0000e-04
Epoch 33/60
12081/12082 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - auc: 0.9186 - binary_accuracy: 0.814
9 - loss: 0.1905 - precision: 0.6124 - recall: 0.9006
Epoch 33: saving model to /kaggle/working/model/checkpoints/33-0.11.keras
12082/12082 ━━━━━━━━━━━━━━━━━━━━ 137s 11ms/step - auc: 0.9186 - binary_accuracy: 0.8
149 - loss: 0.1905 - precision: 0.6124 - recall: 0.9006 - val_auc: 0.8915 - val_bina
ry_accuracy: 0.8429 - val_loss: 0.1121 - val_precision: 0.7894 - val_recall: 0.6995
- learning_rate: 2.5000e-04
Epoch 34/60
12081/12082 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - auc: 0.9190 - binary_accuracy: 0.815
9 - loss: 0.1899 - precision: 0.6135 - recall: 0.9025
Epoch 34: saving model to /kaggle/working/model/checkpoints/34-0.12.keras
12082/12082 ━━━━━━━━━━━━━━━━━━━━ 137s 11ms/step - auc: 0.9190 - binary_accuracy: 0.8
159 - loss: 0.1899 - precision: 0.6135 - recall: 0.9025 - val_auc: 0.8959 - val_bina
ry_accuracy: 0.8350 - val_loss: 0.1205 - val_precision: 0.7047 - val_recall: 0.8407
- learning_rate: 2.5000e-04
```

In [21]:
```
history_df = pd.DataFrame(history.history)
history_df.loc[:, ['loss', 'val_loss']].plot()
history_df.loc[:, ['binary_accuracy', 'val_binary_accuracy', 'recall', 'val_recall'
```

Out[21]: <Axes: >





pretty good results! let's evaluate on testing set

In [6]:
```
X_test = image_dataset_from_directory(
    test_path,
```

```
    labels='inferred',
    label_mode='binary',
    color_mode='rgb',
    batch_size=None,
    image_size=(50,50),
    shuffle=True,
    interpolation='nearest'
)
X_test = (
    X_test
    .map(preprocess, num_parallel_calls=AUTOTUNE, deterministic=True)
    .batch(16)
    .cache()
    .prefetch(buffer_size=AUTOTUNE)
)
```

```
Found 45729 files belonging to 2 classes.
```

In [7]:
```
model = tf.keras.models.load_model('../model/model3/model_v5.keras')
```

In [8]:
```
model.evaluate(
    X_test,
    batch_size=None,
    verbose=1,
    steps=None,
    return_dict=True
)
```

```
2859/2859 ━━━━━━━━━━━━━━━━━━━━ 89s 31ms/step - auc: 0.9026 - binary_accuracy: 0.8406
- loss: 0.1080 - precision: 0.6987 - recall: 0.7772
```

Out[8]:
```
{'auc': 0.9044559001922607,
 'binary_accuracy': 0.8445625305175781,
 'loss': 0.1068098247051239,
 'precision': 0.7008564472198486,
 'recall': 0.7851284146308899}
```

I am more than happy with these results, the model learned to generalize very well, AUC shows an excellent ability to distinguish between positive and negative classes, 84.4% accuracy is very good for this imbalanced data, precision is okay and most importantly we have 78.5% of Recall which is very good, we will rarely have a false negative while keeping a reasonable false positives rate.

let's see this more clearly on a confusion matrix.

But scikit-learn's classification metrics can't handle a mix of binary and continuous targets so we need to find a good threshold to binarize our predictions and we'll use the recall and precision curve and then look for the best f1 score, and ROC curve so we can compare both.

In [9]:
```
y_pred = model.predict(X_test)
y_true = np.concatenate([labels for _,labels in X_test])
```

```
2859/2859 ━━━━━━━━━━━━━━━━━━━━ 17s 6ms/step
```

ROC curve method

In [10]:
```
fpr, tpr, thresholds = roc_curve(y_true, y_pred)
j_scores = tpr - fpr
best_threshold_roc = thresholds[np.argmax(j_scores)]
print(f"Best threshold by ROC Youden's J: {best_threshold_roc}")
```

```
Best threshold by ROC Youden's J: 0.4114065170288086
```

best f1 score calculator

In [11]:
```
f1_scores = []
thresholds = np.linspace(0.4,0.6,1000)

for t in thresholds:
    preds = (y_pred >= t).astype(int)
    f1 = f1_score(y_true, preds)
    f1_scores.append(f1)

best_threshold = thresholds[np.argmax(f1_scores)]
print(f"Best threshold by F1 score: {best_threshold}")
```

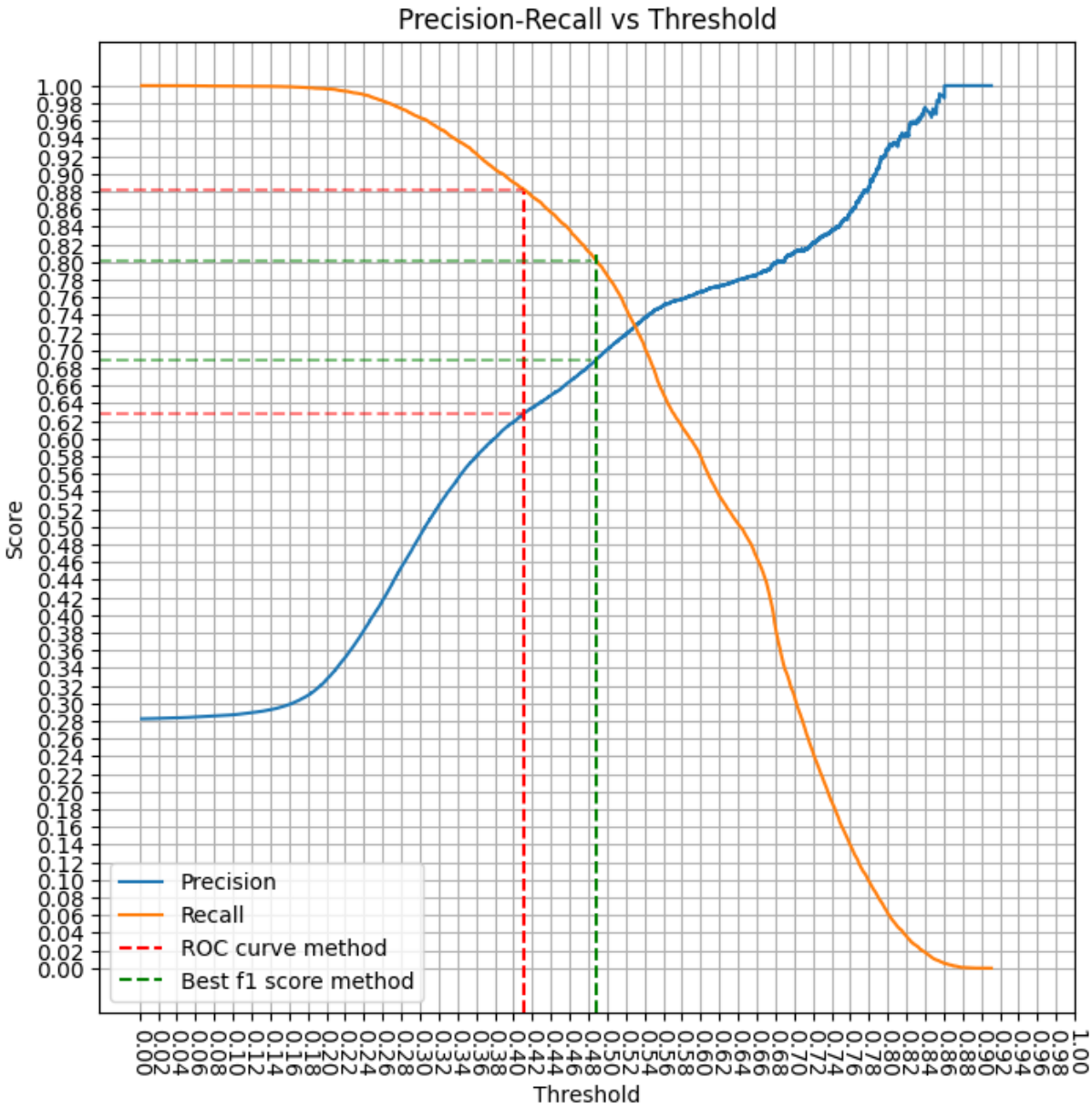
```
Best threshold by F1 score: 0.48848848848848847
```

And we can see both these thresholds on a recall, precision curve

In [28]:
```
y_true = []
y_score = []

for images, labels in X_test:
    probs = model.predict(images, verbose=0)
    y_score.extend(probs.flatten())       # predicted probabilities
    y_true.extend(labels.numpy().flatten())  # true labels

y_true = np.array(y_true)
y_score = np.array(y_score)

precision, recall, thresholds = precision_recall_curve(y_true, y_score)
```

In [76]:
```
plt.figure(figsize=(8,8))
plt.plot(thresholds, precision[:-1], label='Precision')
plt.plot(thresholds, recall[:-1], label='Recall')
plt.axvline(x=0.4114,ymin=0,ymax=0.845, color='red', linestyle='--', label='ROC cur
plt.axvline(x=0.4885, ymin=0, ymax=0.78 ,color='green', linestyle='--', label='Best

idx_roc = np.abs(thresholds - 0.4114).argmin()
idx_f1 = np.abs(thresholds - 0.4885).argmin()

prec_roc = precision[idx_roc]
rec_roc = recall[idx_roc]
prec_f1 = precision[idx_f1]
rec_f1 = recall[idx_f1]

plt.axhline(y=prec_roc, color='red', linestyle='--', alpha=0.5, xmin=0, xmax=0.435)
plt.axhline(y=rec_roc, color='red', linestyle='--', alpha=0.5, xmin=0, xmax=0.435)
plt.axhline(y=prec_f1, color='green', linestyle='--', alpha=0.5, xmin=0, xmax=0.505
plt.axhline(y=rec_f1, color='green', linestyle='--', alpha=0.5, xmin=0, xmax=0.505)
```

```
plt.xticks(np.linspace(0,1,51), rotation=-90)
plt.yticks(np.linspace(0,1,51))
plt.xlabel('Threshold')
plt.ylabel('Score')
plt.title('Precision-Recall vs Threshold')
plt.legend()
plt.grid(True)
plt.show()
```



In [47]:
```
def plot_confusion(X_test, threshold):
    y_pred = model.predict(X_test)
    y_true = np.concatenate([labels for _,labels in X_test])

    def binarize(y):
        if y>=threshold:
            return 1
        else:
            return 0
```

```
    y_pred = [binarize(y) for y in y_pred]
    mat = confusion_matrix(
        y_true=y_true,
        y_pred=y_pred
    )
    return mat
```

In [85]:
```
print(f"confusion matrix with 0.4114 is \n{plot_confusion(X_test,0.4114)}")
print(f"confusion matrix with 0.4885 is \n{plot_confusion(X_test,0.4885)}")
```

```
2859/2859 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step
confusion matrix with 0.4114 is
[[26079  6726]
 [ 1519 11405]]
2859/2859 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step
confusion matrix with 0.4885 is
[[28147  4658]
 [ 2559 10365]]
```

We get slightly different results, using the folowing formulas we can compute all the metrics

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1} = \frac{2TP}{2TP + FP + FN}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

In [86]:
```
data = {
    'Threshold': ['0.4114 (ROC-Youden)', '0.4885 (F1-optimal)'],
    'Precision': [0.6288, 0.6901],
    'Recall': [0.8824, 0.8021],
    'F1 Score': [0.7344, 0.7419],
    'Accuracy': [0.819, 0.8433],
    'False Positives': [6726, 4658],
    'False Negatives': [1519, 2559]
}

df = pd.DataFrame(data)
df.set_index('Threshold', inplace=True)
display(df)
```

Threshold	Precision	Recall	F1 Score	Accuracy	False Positives	False Negatives
0.4114 (ROC-Youden)	0.6288	0.8824	0.7344	0.8190	6726	1519
0.4885 (F1-optimal)	0.6901	0.8021	0.7419	0.8433	4658	2559

Interpretation:

The ROC-based threshold (0.4114) achieves higher recall, which is desirable in medical contexts where missing a positive case is dangerous.

The F1-optimal threshold (0.4885) offers a better balance of precision and recall, leading to a higher F1 score and accuracy overall.

False positives are significantly reduced with the F1-optimal threshold, making predictions more trustworthy in real-world settings.

We chose to favor the F1-optimal threshold as it maintains strong recall while improving precision and reducing false alarms, offering a robust trade-off for practical deployment.

Conclusion

This project successfully demonstrates the design and training of a custom CNN to classify Invasive Ductal Carcinoma (IDC) from breast histopathology images, achieving strong performance metrics including an AUC of 0.90 and balanced precision and recall. The model provides a solid foundation for further exploration and potential clinical application.

Next Steps

- Implement Grad-CAM visualizations for model explainability
- Experiment with more advanced CNN architectures (e.g., EfficientNet)
- Develop a user-friendly interface or deploy the model as a web app

---

Thank you for reviewing my work! Feel free to reach out for questions, feedback, or collaboration.