Classification with BHI Dataset and VGG-style network

In this experiment you will set up a VGG-style network to classify histopathologic scans of breast tissue from the BHI (https://www.kaggle.com/paultimothymooney/breast-histopathology-images) dataset.

In [1]:
```
import tensorflow.keras as keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Conv2D, Dense, Flatten, MaxPooling2D
from tensorflow.keras.optimizers import SGD, Adam
from matplotlib import pyplot as plt
import numpy as np
```

```
WARNING:tensorflow:From C:\Users\Johan\anaconda3\envs\py311\Lib\site-packages\keras\src\losses.py:2976: The name
tf.losses.sparse_softmax_cross_entropy is deprecated. Please use tf.compat.v1.losses.sparse_softmax_cross_entropy
instead.
```

Here we use a Keras utility function to load the dataset. I already organized the data into HDF5 files which are a good format for storing array data.

In [2]:
```
from tensorflow.keras.utils import get_file
x_train_path = get_file('idc_train.h5','https://storage.googleapis.com/data401-datasets/idc_train.h5')
x_test_path = get_file('idc_test.h5','https://storage.googleapis.com/data401-datasets/idc_test.h5')
```

We read the data from the HDF5 files into Numpy arrays.

I crop the images so they are all 48x48.

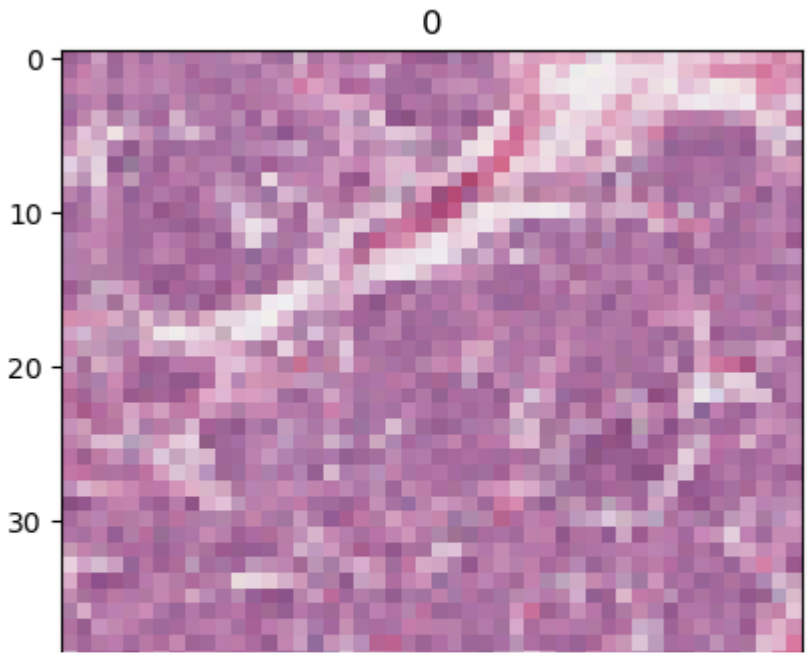
In [3]:
```
import h5py as h5
with h5.File(x_train_path,'r') as f:
  x_train = f['X'][::2,1:49,1:49] # load half the data to avoid out-of-memory errors
  y_train = f['y'][::2]
with h5.File(x_test_path,'r') as f:
  x_test = f['X'][:,1:49,1:49]
  y_test = f['y'][:]
```

In [4]:
```
x_train.shape,y_train.shape,x_test.shape,y_test.shape
```

Out[4]:
```
((61925, 48, 48, 3), (61925,), (13761, 48, 48, 3), (13761,))
```

Showing a few images from the dataset.

In [5]:
```
for i in range(5):
  plt.imshow(np.squeeze(x_train[i]))
  plt.title(y_train[i])
  plt.show()
```



Data preprocessing

1. Convert the train and test images to floating point and divide by 255.
2. Compute the average value of the entire training image set.
3. Subtract the average value from the training and testing images.

In [6]:
```
x_test = x_test.astype("float32")/255
x_train = x_train.astype("float32")/255

x_train_avg = np.mean(x_train)

x_train = x_train-x_train_avg
x_test = x_test-x_train_avg
```

Build a VGG-style binary classifier model. For example, your network could contain the following:

1. 32 convolutional filters of size 3x3, zero padding, ReLU activation
2. 2x2 max pooling with stride 2
3. 64 filters
4. max pool
5. 128 filters
6. max pool
7. 256 filters
8. max pool
9. flatten
10. Fully-connected layer with 128 outputs
11. Final binary classification layer

In [59]:
```
model = Sequential([
        Input(x_train.shape[1:]),
        Conv2D(1,3,activation='relu',padding='same',name='conv1'),
        MaxPooling2D(2,2),
        Flatten(),
        Dense(2,activation='softmax',name='z')
])
model.summary()
```

```
Model: "sequential_15"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1 (Conv2D)              (None, 48, 48, 1)         28

 max_pooling2d_35 (MaxPooli  (None, 24, 24, 1)         0
 ng2D)

 flatten_15 (Flatten)        (None, 576)               0

 z (Dense)                   (None, 2)                 1154

=================================================================
Total params: 1182 (4.62 KB)
Trainable params: 1182 (4.62 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Set up the model to optimize the sparse categorical cross-entropy loss using Adam optimizer and learning rate of $.0003$. Calculate accuracy metrics during training.

In [65]:

```
learning_rate = 3e-4


opt = Adam(learning_rate=learning_rate)

model.compile(loss='sparse_categorical_crossentropy',optimizer=opt,metrics='accuracy')
```

Now `fit` the model to the data using a batch size of 32 and 10% validation split over 10 epochs.

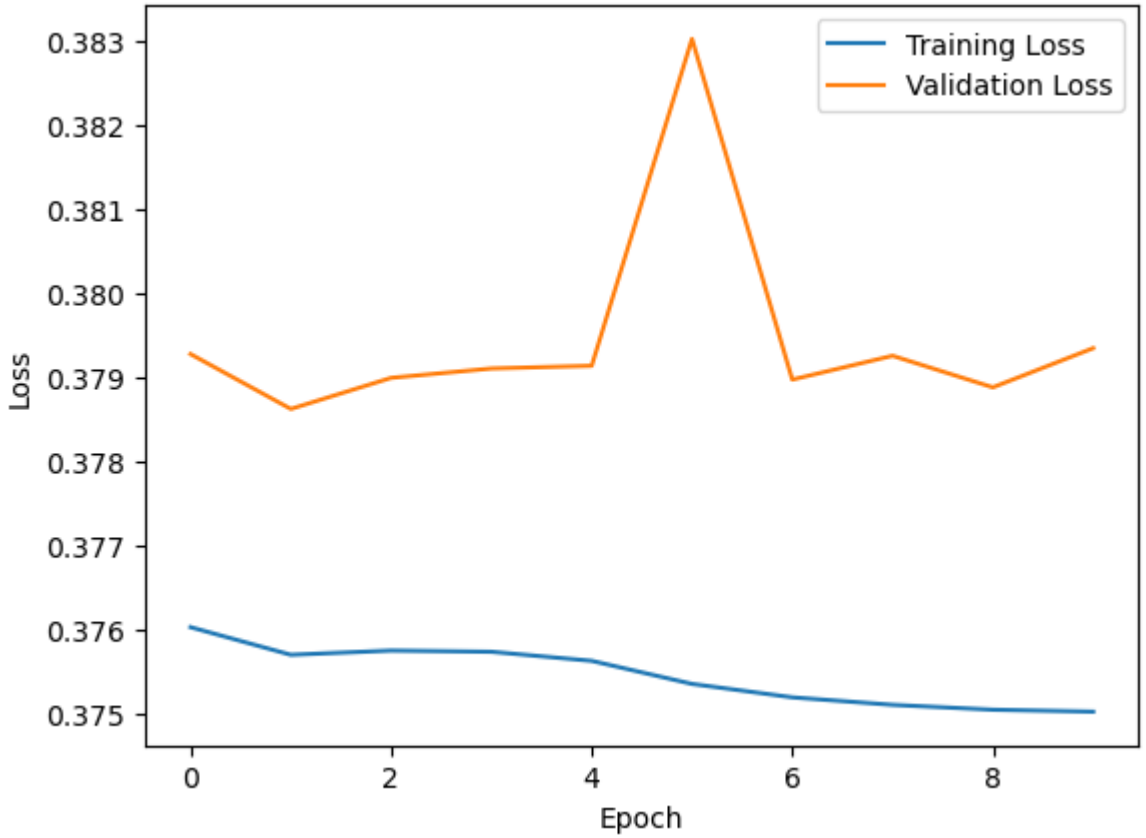
In [70]:
```
batch_size = 32
epochs = 10

history = model.fit(x_train,y_train,batch_size=batch_size,epochs=epochs,validation_split=0.1,verbose=True)
```

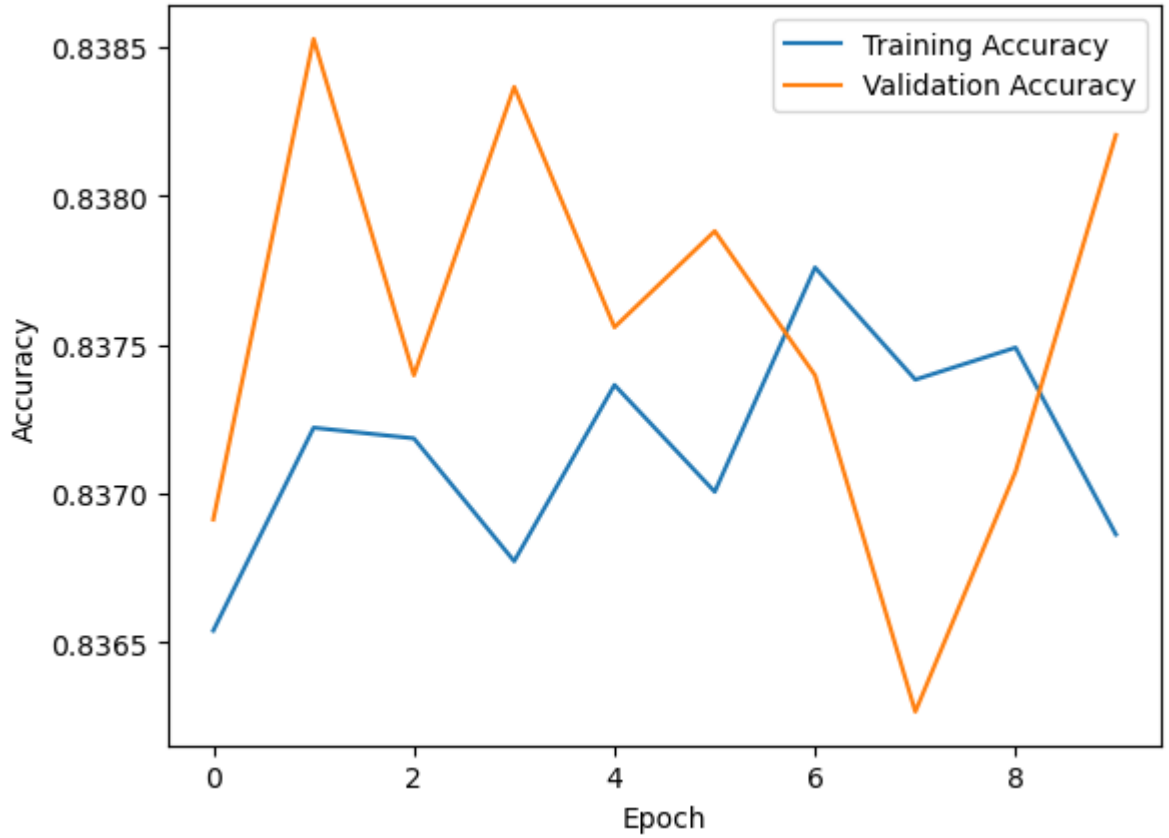
```
Epoch 1/10
1742/1742 [==============================] - 5s 3ms/step - loss: 0.3760 - accuracy: 0.8365 - val_loss: 0.3793 - v
al_accuracy: 0.8369
Epoch 2/10
1742/1742 [==============================] - 4s 2ms/step - loss: 0.3757 - accuracy: 0.8372 - val_loss: 0.3786 - v
al_accuracy: 0.8385
Epoch 3/10
1742/1742 [==============================] - 4s 2ms/step - loss: 0.3757 - accuracy: 0.8372 - val_loss: 0.3790 - v
al_accuracy: 0.8374
Epoch 4/10
1742/1742 [==============================] - 5s 3ms/step - loss: 0.3757 - accuracy: 0.8368 - val_loss: 0.3791 - v
al_accuracy: 0.8384
Epoch 5/10
1742/1742 [==============================] - 4s 2ms/step - loss: 0.3756 - accuracy: 0.8374 - val_loss: 0.3791 - v
al_accuracy: 0.8376
Epoch 6/10
1742/1742 [==============================] - 4s 2ms/step - loss: 0.3753 - accuracy: 0.8370 - val_loss: 0.3830 - v
al_accuracy: 0.8379
Epoch 7/10
1742/1742 [==============================] - 4s 2ms/step - loss: 0.3752 - accuracy: 0.8378 - val_loss: 0.3790 - v
al_accuracy: 0.8374
Epoch 8/10
1742/1742 [==============================] - 4s 2ms/step - loss: 0.3751 - accuracy: 0.8374 - val_loss: 0.3793 - v
al_accuracy: 0.8363
Epoch 9/10
1742/1742 [==============================] - 4s 2ms/step - loss: 0.3750 - accuracy: 0.8375 - val_loss: 0.3789 - v
al_accuracy: 0.8371
Epoch 10/10
1742/1742 [==============================] - 4s 2ms/step - loss: 0.3750 - accuracy: 0.8369 - val_loss: 0.3793 - v
al_accuracy: 0.8382
```

Plot loss and accuracy over the training run.

In [71]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['Training Loss','Validation Loss'])
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



In [72]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.legend(['Training Accuracy','Validation Accuracy'])
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()
```



Compute accuracy of the model on the training and testing sets.

In [73]:
```python
# Evaluate the model on the test data
loss, accuracy = model.evaluate(x_test, y_test)

print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

```
431/431 [==============================] - 1s 1ms/step - loss: 0.3854 - accuracy: 0.8315
Test Loss: 0.3854
Test Accuracy: 83.15%
```

Try a different setting to see if you can improve the test set accuracy at all. Write about the results here.

```
Results:
    - I played with the VGG blueprint a bit.
    - In our default setting as described above the VGG setup we already encounter overfitting.
      Which is caused by a too complex model which now fits irrelevent data.
    - Increasing the conv sizes and amount of filter and pooling steps resulted in an expected behaviour.
      Because we increased the number of parameters to train, we did not improve the performance at all and ran into
even more overfitting.
    - Decreasing the amount of parameters the model has to fit to, already resulted in a better validation loss
after the ten epochs. And The Accuracy of the model got increased as well.
    - continuously decreasing the amount of parameters to learn further decreased the validation loss and increased
the test accuracy.
    - I continued this until I ran into the issue of underfitting which led to almost no significant change of the
training loss curve. This indicates that the model is not learning effectively from the data.
```