solution

October 19, 2024

1 Jupyter Notebook For DTSA-5511 WK3

2 Introduction

In this notebook, we tackle the challenge of identifying metastatic cancer in image patches derived from larger digital pathology scans using the PatchCamelyon (PCam) dataset. The PCam dataset requires a solution as a binary image classification task.

The goal of this competition is to develop an algorithm that accurately classifies image patches as either containing metastatic cancer or not.

In the following sections, we will outline our approach, including data preprocessing, model selection, training procedures, and evaluation metrics. By the end of this notebook, we aim to provide insights and results that contribute to the understanding of metastatic cancer detection using machine learning.

[2]:
```python
# Import necessary libraries
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cv2
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,␣
 ↪Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping
import glob
```

```
2024-10-19 13:42:01.464166: W
tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load
dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open
shared object file: No such file or directory; LD_LIBRARY_PATH:
/home/algorithmspath/.local/lib/python3.8/site-packages/cv2/../../lib64:
2024-10-19 13:42:01.464193: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]
Ignore above cudart dlerror if you do not have a GPU set up on your machine.
```

[3]:
```
# Set constants
IMG_SIZE = (128, 128)  # Image size for resizing
BATCH_SIZE = 32
EPOCHS = 10  # Adjust as needed

# Define file paths
TRAIN_LABELS_CSV = 'train_labels.csv'
TRAIN_DIR = 'train'
TEST_DIR = 'test'
SUBMISSION_CSV = 'submission.csv'
```

Dataset:

train_labels.csv - labels of metastatic; 220025 records

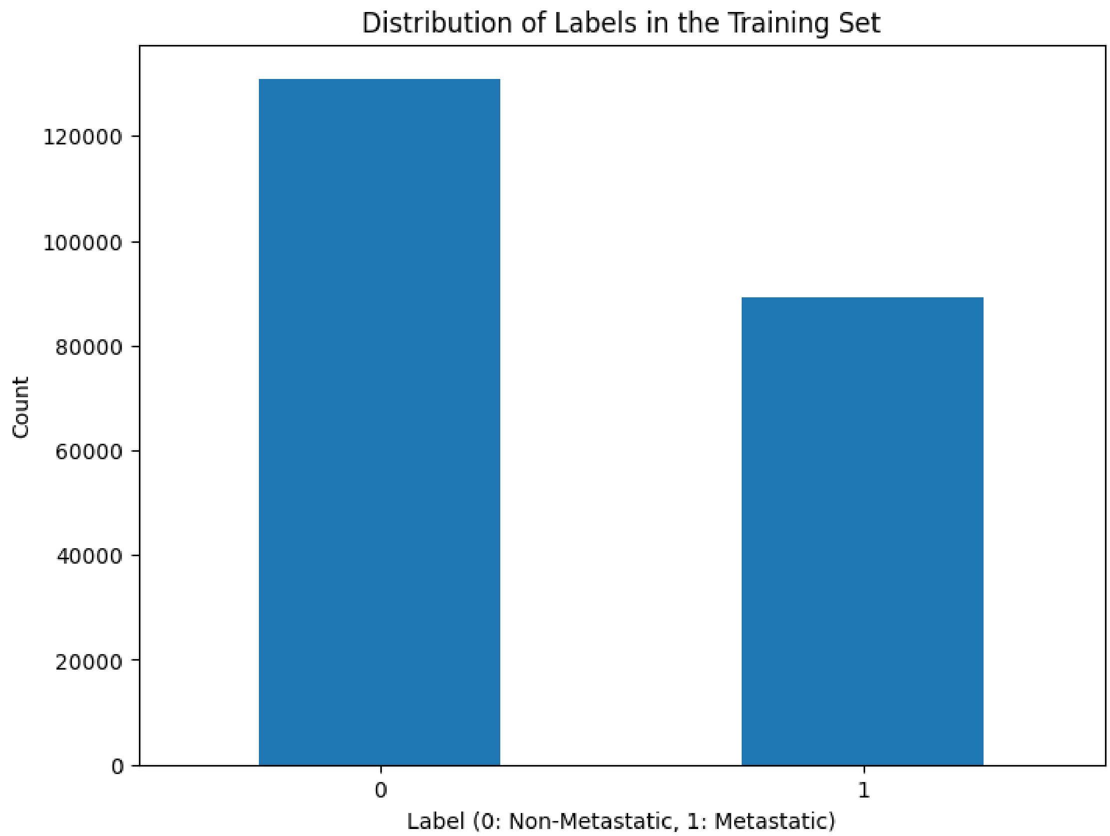
train_dir - directory of images corresponding to train_labels.csv

test_dir - directory of images to predict

[4]:
```
# Load labels from CSV
train_labels = pd.read_csv(TRAIN_LABELS_CSV)
print(train_labels.head())

print(train_labels.shape)
```

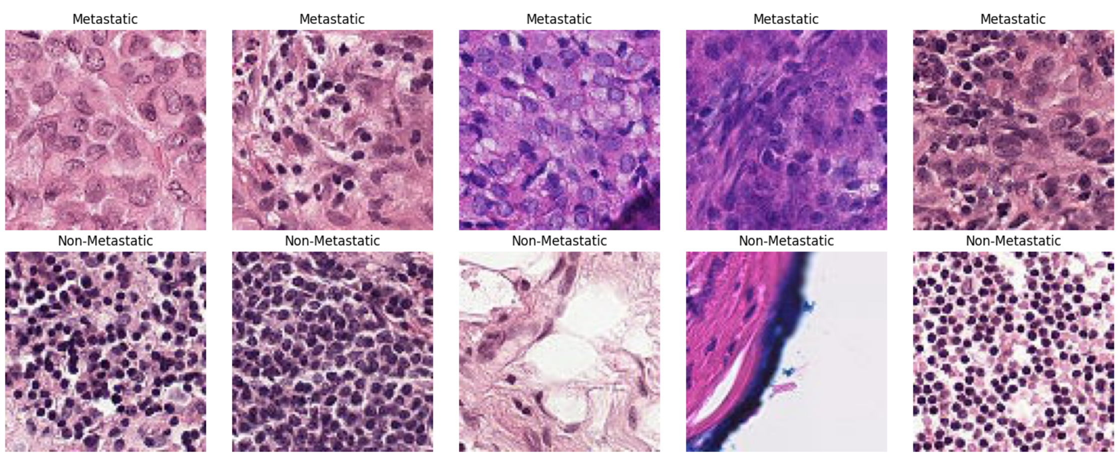
```
                                         id  label
0  f38a6374c348f90b587e046aac6079959adf3835      0
1  c18f2d887b7ae4f6742ee445113fa1aef383ed77      1
2  755db6279dae599ebb4d39a9123cce439965282d      0
3  bc3f0c64fb968ff4a8bd33af6971ecae77c75e08      0
4  068aba587a4950175d04c680d38943fd488d6a9d      0
(220025, 2)
```

[14]:
```
# Exploratory Data Analysis (EDA)
# Check the distribution of classes
label_counts = train_labels['label'].value_counts()
plt.figure(figsize=(8, 6))
label_counts.plot(kind='bar')
plt.title('Distribution of Labels in the Training Set')
plt.xlabel('Label (0: Non-Metastatic, 1: Metastatic)')
plt.ylabel('Count')
plt.xticks(rotation=0)
plt.show()
```



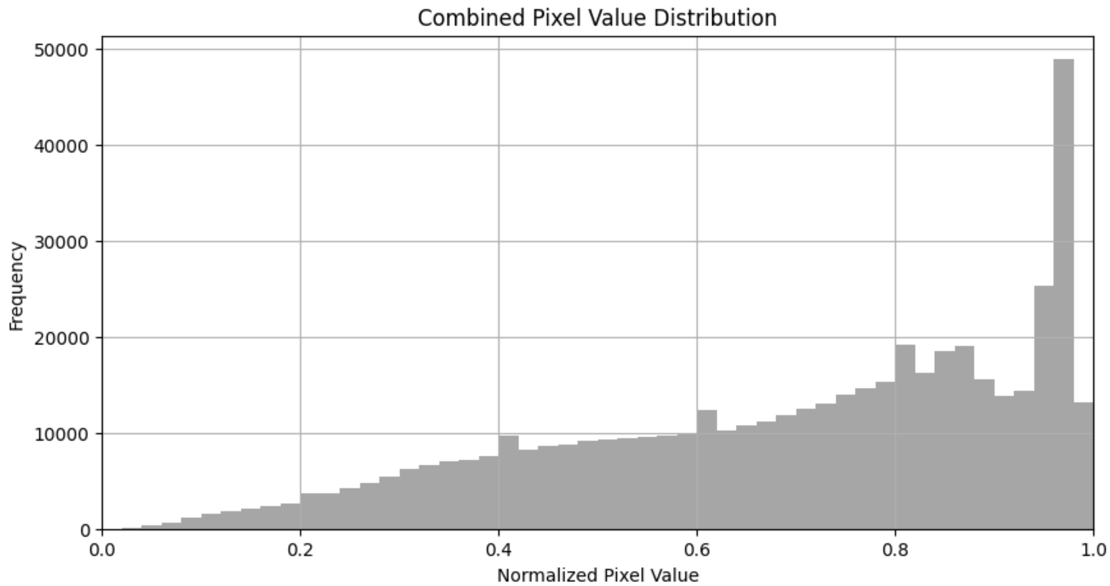
[12]:
```
import os
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image

# Function to load and display images
def display_images(train_dir, labels_df, num_images=5):
    # Separate the metastatic and non-metastatic images
    metastatic_images = labels_df[labels_df['label'] == 1]['id'].values
    non_metastatic_images = labels_df[labels_df['label'] == 0]['id'].values

    # Randomly select images
    selected_metastatic = np.random.choice(metastatic_images, num_images,␣
↪replace=False)
    selected_non_metastatic = np.random.choice(non_metastatic_images,␣
↪num_images, replace=False)

    plt.figure(figsize=(15, 6))

    # Display metastatic images
```

```
    for i, img_id in enumerate(selected_metastatic):
        img_path = os.path.join(train_dir, img_id + '.tif')  # Adjust extension␣
 ↪if necessary
        img = Image.open(img_path)
        plt.subplot(2, num_images, i + 1)
        plt.imshow(img)
        plt.axis('off')
        plt.title('Metastatic')

    # Display non-metastatic images
    for i, img_id in enumerate(selected_non_metastatic):
        img_path = os.path.join(train_dir, img_id + '.tif')  # Adjust extension␣
 ↪if necessary
        img = Image.open(img_path)
        plt.subplot(2, num_images, i + 1 + num_images)
        plt.imshow(img)
        plt.axis('off')
        plt.title('Non-Metastatic')

    plt.tight_layout()
    plt.show()

# Display images from the dataset
# print(os.listdir(TRAIN_DIR))
display_images(TRAIN_DIR, train_labels, num_images=5)
```



[21]:
```
# Function to load and resize images
def load_image(image_path):
    img = cv2.imread(image_path, cv2.IMREAD_COLOR)
    if img is not None:
        img = cv2.resize(img, IMG_SIZE)  # Resize to your desired size
```

```python
        img = img.astype(np.float32) / 255.0  # Scale to [0, 1]
        return img
    return None

# Load the first 10 images
def load_first_n_images(n=10):
    images = []
    for i in range(n):
        img_path = os.path.join(TRAIN_DIR, train_labels['id'].iloc[i] + '.tif')␣
 ↪ # Adjust extension if necessary
        img = load_image(img_path)
        if img is not None:
            images.append(img)
    return np.array(images)

# Plot combined pixel value distribution
def plot_combined_pixel_distribution(images, num_bins=50):
    # Flatten the images to get all pixel values
    pixel_values = images.flatten()  # Shape: (num_images * height * width *␣
 ↪channels,)

    plt.figure(figsize=(10, 5))
    plt.hist(pixel_values, bins=num_bins, color='gray', alpha=0.7)
    plt.title('Combined Pixel Value Distribution')
    plt.xlabel('Normalized Pixel Value')
    plt.ylabel('Frequency')
    plt.xlim(0, 1)  # Adjust for normalized pixel values
    plt.grid()
    plt.show()

# Load the first 10 images and plot their combined pixel value distribution
images = load_first_n_images(10)
plot_combined_pixel_distribution(images)
```



[15]:
```
# Load images in batches

# Load image and perform normalization
def load_image(image_path):
    img = cv2.imread(image_path, cv2.IMREAD_COLOR)
    if img is not None:
        img = cv2.resize(img, IMG_SIZE)

        # Normalize the image
        img = img.astype(np.float32) / 255.0  # Scale to [0, 1]
        img = (img - MEAN) / STD  # Normalize using mean and std
        return img
    return None

def generate_data(batch_size):
    while True:
        for start in range(0, len(train_labels), batch_size):
            end = min(start + batch_size, len(train_labels))
            batch_images = []
            batch_labels = []
            for i in range(start, end):
                img_path = os.path.join(TRAIN_DIR, train_labels['id'].iloc[i] +␣
↪'.tif')
                img = load_image(img_path)
                if img is not None:
                    batch_images.append(img)
                    batch_labels.append(train_labels['label'].iloc[i])
```

```
            yield np.array(batch_images), np.array(batch_labels)
```

[16]:
```
# Create a CNN model
def create_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_SIZE[0],␣
 ↪IMG_SIZE[1], 3)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))  # Binary classification
    model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
    return model
```

[17]:
```
# Prepare training and validation datasets
X_train, X_val, y_train, y_val = train_test_split(train_labels['id'],␣
 ↪train_labels['label'], test_size=0.2, random_state=42)

# Initialize the model
model = create_model()
model
```

[17]: <keras.engine.sequential.Sequential at 0x7feee3f40130>

[18]:
```
# Train the model
# Note: Training done outside of notebook environment.
early_stopping = EarlyStopping(monitor='val_loss', patience=3)
train_gen = generate_data(BATCH_SIZE)

model.fit(train_gen,
          steps_per_epoch=len(X_train) // BATCH_SIZE,
          validation_data=generate_data(BATCH_SIZE),
          validation_steps=len(X_val) // BATCH_SIZE,
          epochs=EPOCHS,
          callbacks=[early_stopping])
```

```
2024-10-09 10:49:16.155514: I
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR
Optimization Passes are enabled (registered 2)
```

```
Epoch 1/10
```

```
2024-10-09 10:49:17.353270: W
tensorflow/core/framework/cpu_allocator_impl.cc:80] Allocation of 65028096
exceeds 10% of free system memory.
2024-10-09 10:49:17.709597: W
tensorflow/core/framework/cpu_allocator_impl.cc:80] Allocation of 30482432
exceeds 10% of free system memory.
2024-10-09 10:49:18.037437: W
tensorflow/core/framework/cpu_allocator_impl.cc:80] Allocation of 23482368
exceeds 10% of free system memory.
2024-10-09 10:49:18.152410: W
tensorflow/core/framework/cpu_allocator_impl.cc:80] Allocation of 30482432
exceeds 10% of free system memory.
2024-10-09 10:49:18.192791: W
tensorflow/core/framework/cpu_allocator_impl.cc:80] Allocation of 25719552
exceeds 10% of free system memory.
```

```
  91/5500 […] - ETA: 48:00 - loss: 6.9329 -
accuracy: 0.6078
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[18], line 5
      2 early_stopping = EarlyStopping(monitor='val_loss', patience=3)
      3 train_gen = generate_data(BATCH_SIZE)
----> 5 model.fit(train_gen,
      6           steps_per_epoch=len(X_train) // BATCH_SIZE,
      7           validation_data=generate_data(BATCH_SIZE),
      8           validation_steps=len(X_val) // BATCH_SIZE,
      9           epochs=EPOCHS,
     10           callbacks=[early_stopping])

File ~/.local/lib/python3.8/site-packages/keras/engine/training.py:1184, in
 ↪Model.fit(self, x, y, batch_size, epochs, verbose, callbacks,
 ↪validation_split, validation_data, shuffle, class_weight, sample_weight,
 ↪initial_epoch, steps_per_epoch, validation_steps, validation_batch_size,
 ↪validation_freq, max_queue_size, workers, use_multiprocessing)
   1177 with tf.profiler.experimental.Trace(
   1178     'train',
   1179     epoch_num=epoch,
   1180     step_num=step,
   1181     batch_size=batch_size,
   1182     _r=1):
   1183   callbacks.on_train_batch_begin(step)
-> 1184   tmp_logs = self.train_function(iterator)
   1185   if data_handler.should_sync:
   1186     context.async_wait()

File ~/.local/lib/python3.8/site-packages/tensorflow/python/eager/def_function.
 ↪py:885, in Function.__call__(self, *args, **kwds)
```

```
    882 compiler = "xla" if self._jit_compile else "nonXla"
    884 with OptionalXlaContext(self._jit_compile):
--> 885   result = self._call(*args, **kwds)
    887 new_tracing_count = self.experimental_get_tracing_count()
    888 without_tracing = (tracing_count == new_tracing_count)

File ~/.local/lib/python3.8/site-packages/tensorflow/python/eager/def_function.py:917, in Function._call(self, *args, **kwds)
    914   self._lock.release()
    915   # In this case we have created variables on the first call, so we run the
    916   # defunned version which is guaranteed to never create variables.
--> 917   return self._stateless_fn(*args, **kwds)  # pylint: disable=not-callable
    918 elif self._stateful_fn is not None:
    919   # Release the lock early so that multiple threads can perform the cal
    920   # in parallel.
    921   self._lock.release()

File ~/.local/lib/python3.8/site-packages/tensorflow/python/eager/function.py:3039, in Function.__call__(self, *args, **kwargs)
   3036 with self._lock:
   3037   (graph_function,
   3038    filtered_flat_args) = self._maybe_define_function(args, kwargs)
-> 3039 return graph_function._call_flat(
   3040     filtered_flat_args, captured_inputs=graph_function.captured_inputs)

File ~/.local/lib/python3.8/site-packages/tensorflow/python/eager/function.py:1963, in ConcreteFunction._call_flat(self, args, captured_inputs, cancellation_manager)
   1959 possible_gradient_type = gradients_util.PossibleTapeGradientTypes(args)
   1960 if (possible_gradient_type == gradients_util.POSSIBLE_GRADIENT_TYPES_NONE
   1961     and executing_eagerly):
   1962   # No tape is watching; skip to running the function.
-> 1963   return self._build_call_outputs(self._inference_function.call(
   1964       ctx, args, cancellation_manager=cancellation_manager))
   1965 forward_backward = self._select_forward_and_backward_functions(
   1966     args,
   1967     possible_gradient_type,
   1968     executing_eagerly)
   1969 forward_function, args_with_tangents = forward_backward.forward()

File ~/.local/lib/python3.8/site-packages/tensorflow/python/eager/function.py:591, in _EagerDefinedFunction.call(self, ctx, args, cancellation_manager)
    589 with _InterpolateFunctionError(self):
    590   if cancellation_manager is None:
--> 591     outputs = execute.execute(
    592         str(self.signature.name),
```

```
    593         num_outputs=self._num_outputs,
    594         inputs=args,
    595         attrs=attrs,
    596         ctx=ctx)
    597   else:
    598     outputs = execute.execute_with_cancellation(
    599         str(self.signature.name),
    600         num_outputs=self._num_outputs,
   (…)
    603         ctx=ctx,
    604         cancellation_manager=cancellation_manager)

File ~/.local/lib/python3.8/site-packages/tensorflow/python/eager/execute.py:59
 ↪in quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
     57 try:
     58   ctx.ensure_initialized()
---> 59   tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name, op_name
     60                                       inputs, attrs, num_outputs)
     61 except core._NotOkStatusException as e:
     62   if name is not None:

KeyboardInterrupt:
```

[25]:

```python
# Model inference on test data
def predict_test_data(test_dir):
    test_files = glob.glob(os.path.join(test_dir, '*.tif'))

    predictions = []
    ids = []

    for img_path in test_files[:]:
        img = load_image(img_path)
        if img is not None:
            img = np.expand_dims(img, axis=0) / 255.0  # Normalize
            pred = model.predict(img)
            predictions.append(pred[0][0])
        else:
            predictions.append(None)  # Handle non-loadable images
        ids.append( os.path.basename(img_path).replace('.tif', '') )

    return pd.DataFrame({
        'id': ids,
        'label': [ round(x) for x in predictions ]
    })

# Predict and save to submission file
submission_df = predict_test_data(TEST_DIR)
```

```
print(len(os.listdir(TEST_DIR)))

submission_df.to_csv(SUBMISSION_CSV, index=False)
print("Submission file saved.")
```

```
57458
Submission file saved.
```

3 Metastatic Cancer Detection Model

3.1 Model Architecture

The model is called a Convolutional Neural Network (CNN). It has several important parts:

1. **Convolutional Layers**: There are three of these layers. They look for features in the images using filters. The first layer has 32 filters, the second has 64, and the third has 128.
2. **Max Pooling Layers**: After each convolutional layer, there is a max pooling layer. This layer makes the images smaller and helps the model work faster.
3. **Flatten Layer**: This layer takes the 2D image data and turns it into a single line of numbers. This is needed for the next layers.
4. **Dense Layer**: This layer connects all the numbers together. It has 128 units and uses a special function called ReLU to help the model learn better.
5. **Output Layer**: The final layer has one unit that tells us if there is cancer or not. It gives a score between 0 and 1. If the score is close to 1, it means cancer is present.

3.2 Training Process

1. **Data Preparation**: We load the images in small groups so we don't use too much memory.
2. **Training**: We teach the model using training data. It learns to recognize signs of cancer.
3. **Validation**: We check how well the model is doing with a different set of images. This helps us see if it is learning correctly.
4. **Note on Training**: Training is done on separate environment, outside of notebook.

3.3 Inference Steps

1. **Load Test Images**: We get the images we want to test.
2. **Make Predictions**: The model looks at each test image and decides if it shows cancer or not.
3. **Save Results**: We save these predictions in a file to send out.

This model helps us find metastatic cancer in small image patches from larger scans. The model achieves 67% accuracy on test.

[ ]: