solution

October 29, 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.

[44]:
```
# Import necessary libraries
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cv2
import tensorflow as tf
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,␣
  ↪Dropout, Layer
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping
import glob
```

[9]:
```
# 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

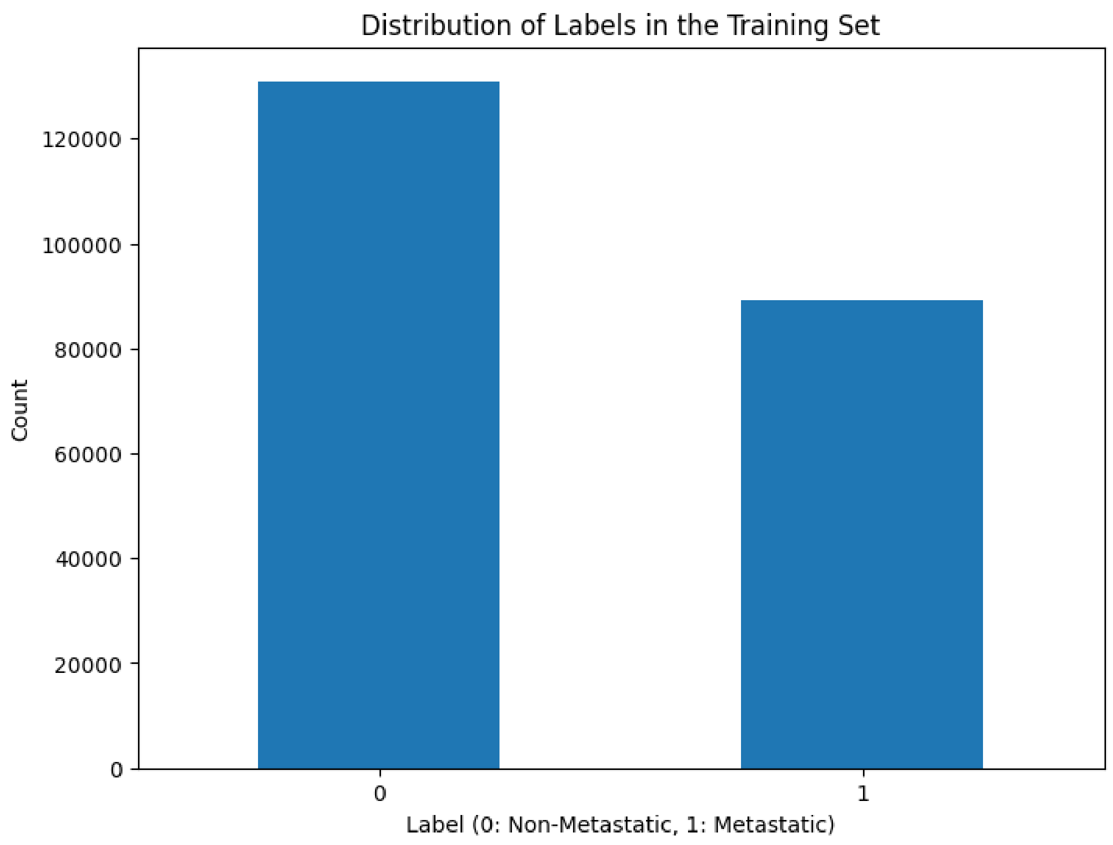
train_dir - directory of images corresponding to train_labels.csv

test_dir - directory of images to predict

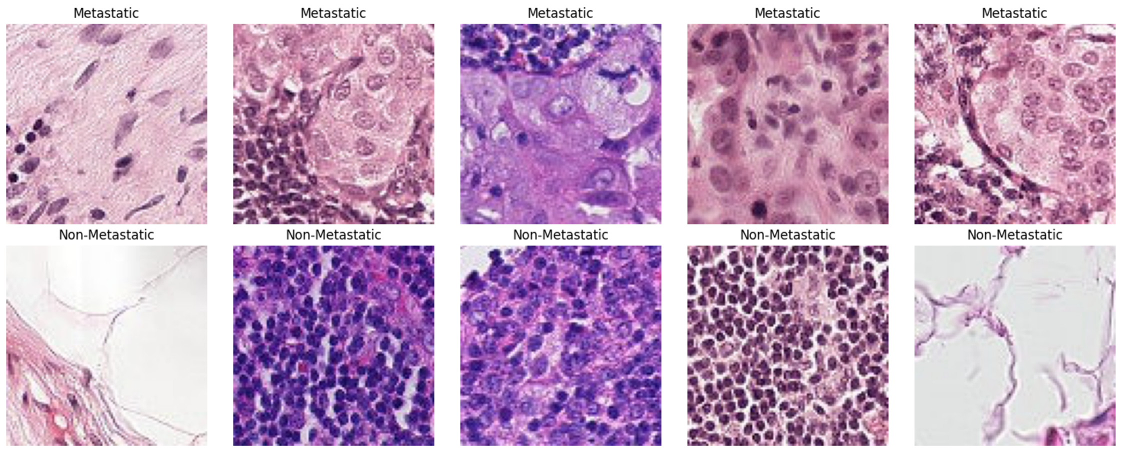
[10]:
```
# Load labels from CSV
train_labels = pd.read_csv(TRAIN_LABELS_CSV)
print(train_labels.head())

print(train_labels.shape)
```

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

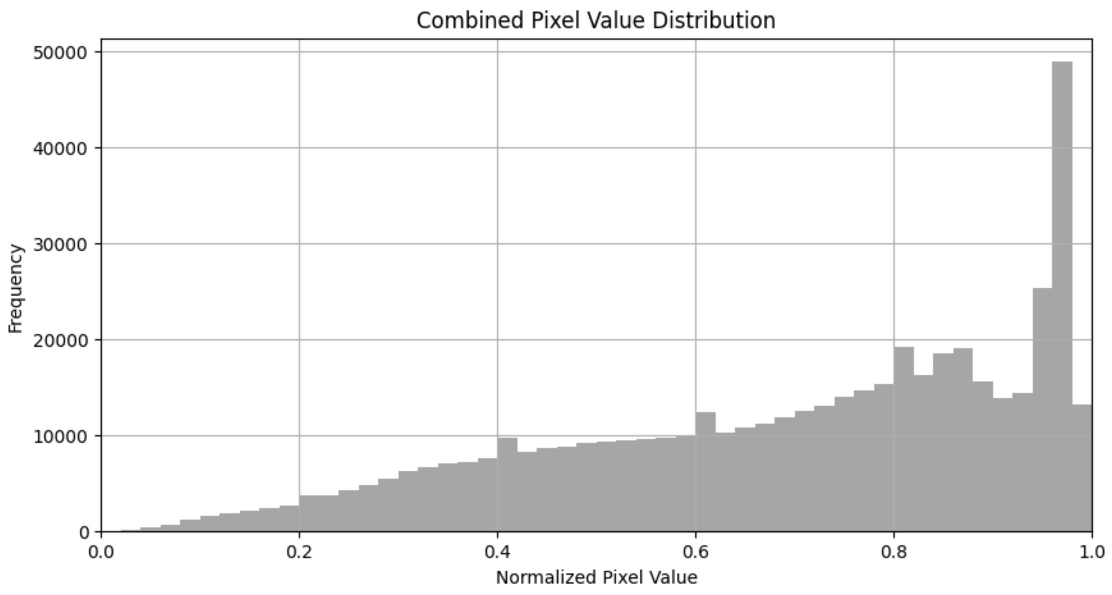
[11]:
```
# Exploratory Data Analysis (EDA)
# Check the distribution of classes
label_counts = train_labels['label'].value_counts()
plt.figure(figsize=(8, 6))
label_counts.plot(kind='bar')
plt.title('Distribution of Labels in the Training Set')
plt.xlabel('Label (0: Non-Metastatic, 1: Metastatic)')
plt.ylabel('Count')
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)
```



[13]:
```
# 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
        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)
```



[4]:
```
# 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)
```

[62]:
```
# Create a CNN model

class AddConstantLayer(Layer):
    def __init__(self, constant_value, **kwargs):
        super(AddConstantLayer, self).__init__(**kwargs)
        self.constant_value = constant_value

    def call(self, inputs):
        return inputs + self.constant_value

def create_model(dropout=0.5):
    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(dropout))
    model.add(Dense(1, activation='sigmoid'))  # Binary classification
    model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
    return model
```

[60]:
```
# 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
```

[60]:
```
<keras.engine.sequential.Sequential at 0x7f2bc4413460>
```

[14]:
```
# 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])
```

2.1 Model Architecture and Selection

The model is a standard CNN for classification. It uses successive convolution + pooling layers to extract the relevant features of the image for the classification task. Finally, the condensed representation is passed through a logistic regression unit for output. The full breakdown is available below for reference.

2.1.1 Architecture Breakdown

- **Input Layer**:
  - Shape: `(IMG_SIZE[0], IMG_SIZE[1], 3)` — **Image Input**.
- **First Convolutional Layer**:
  - Filters: 32, Size: `(3, 3)` — **Edge Detection**.
- **Second Convolutional Layer**:
  - Filters: 64, Size: `(3, 3)` — **Feature Learning**.
- **Third Convolutional Layer**:
  - Filters: 128, Size: `(3, 3)` — **Pattern Refinement**.
- **Max Pooling Layers**:
  - Pool Size: `(2, 2)` — **Dimensionality Reduction**.
- **Flatten Layer**:
  - Converts to 1D — **Preparing for Dense**.
- **Dense Layer**:
  - Units: 128 — **Non-linear Learning**.
- **Dropout Layer**:
  - Rate: `0.5` — **Prevent Overfitting**.
- **Output Layer**:
  - Units: 1, Activation: Sigmoid — **Binary Classification**.

2.2 Hyperparameter Tuning

The main hyperparameter in this model is dropout rate in dropout layer. However, other parameters like filter sizes in convolutional layers can also be considered as well. The learning rate during training can also be considered.

2.3 Comparison with Other Architectures

The main alternative architecture is to add more convolutional + pooling layers. However, this would come at an increased computational cost.

- **Shallow vs. Deep CNNs**: This model has a moderate depth (three convolutional layers). Compared to shallower models, it can capture more complex patterns in the data. However, it is also less complex than deeper networks (e.g., ResNet or Inception) that could potentially provide better accuracy at the cost of higher computational requirements.

3 Model Results and Analysis

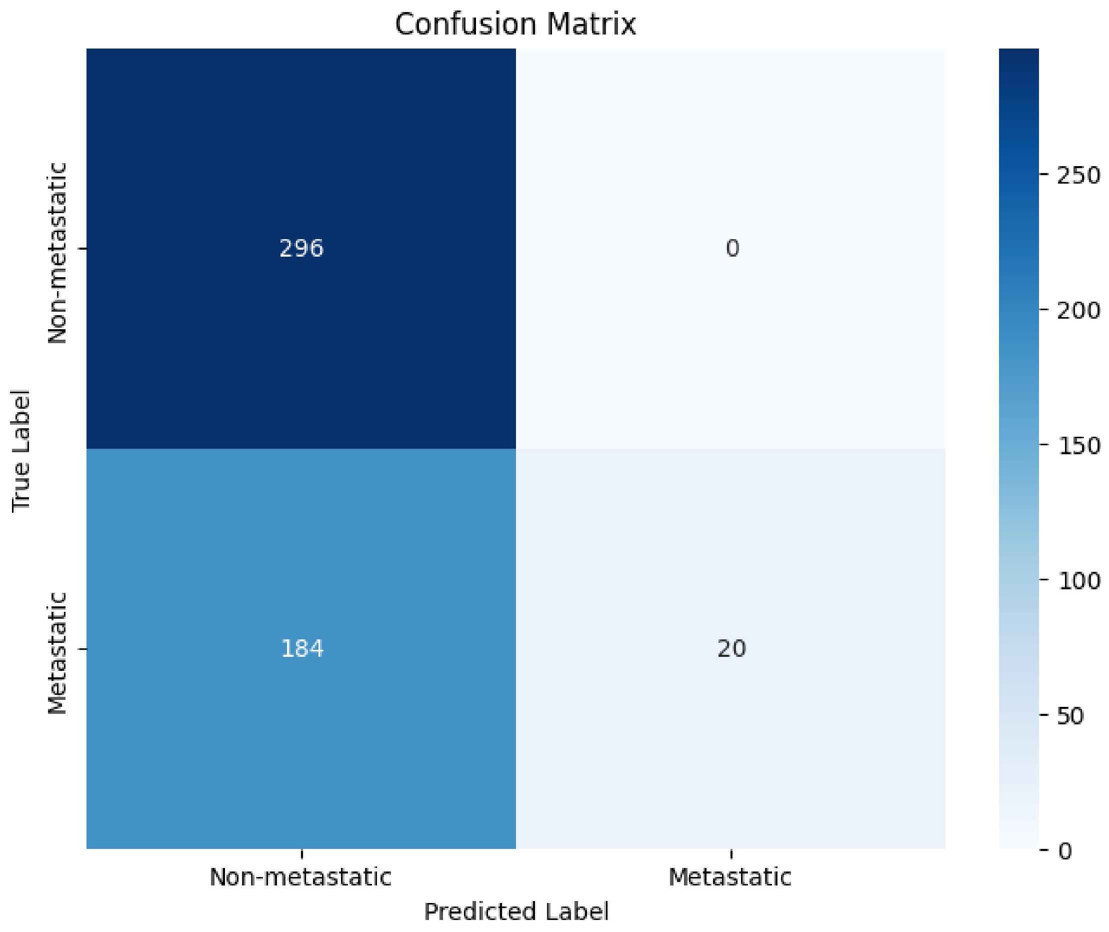
In the below cells, we apply the model on the validation dataset to evaluate its performance. We look at accurracy, precision, recall, and F1-score, and plot the confusion matrix.

[83]:
```python
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score, confusion_matrix
import numpy as np

# Function to evaluate the model
def evaluate_model(y_true, y_pred):
    """
    Evaluate a classification model.

    Parameters:
    y_true: True labels for the validation set.
    y_pred: Predicted labels for the validation set.

    Returns:
    dict: A dictionary containing evaluation metrics.
    """
    conf_matrix = confusion_matrix(y_true, y_pred)

    # Extract TP, TN, FP, FN
    TP = conf_matrix[1, 1]  # True Positives
    TN = conf_matrix[0, 0]  # True Negatives
    FP = conf_matrix[1, 0]  # False Positives
    FN = conf_matrix[0, 1]  # False Negatives

    # Calculate metrics
    accuracy = (TP + TN) / (TP + TN + FP + FN)
    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision +␣
↪recall) > 0 else 0


    results = {
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1 Score': f1,
        'Confusion Matrix': conf_matrix
    }

    return results
```

```
# Make predictions on the validation set
K = 500
X_val_images = np.array([
    load_image( os.path.join(TRAIN_DIR, image_id + '.tif') )
    for image_id in X_val[:K]
])


y_val_pred = model.predict(X_val_images)
y_val_pred_classes = (y_val_pred > 0.5).astype("int32")  # Convert␣
↪probabilities to binary class labels

# print(X_val.shape)
# print(X_train.shape)
# print(X_val_images.shape)
# print(y_val_pred.shape)
# print(y_val_pred)
# print(y_val_pred_classes)

# Evaluate the model
evaluation_results = evaluate_model(y_val[:K], y_val_pred_classes)
# print(evaluation_results)

accuracy = evaluation_results['Accuracy']
precision = evaluation_results['Precision']
recall = evaluation_results['Recall']
f1_score = evaluation_results['F1 Score']

print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1_score:.4f}")
```

```
Accuracy: 0.6320
Precision: 0.0980
Recall: 1.0000
F1 Score: 0.1786
```

[84]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(matrix, class_names):
    plt.figure(figsize=(8, 6))
    sns.heatmap(matrix, annot=True, fmt='d', cmap='Blues',␣
↪xticklabels=class_names, yticklabels=class_names)
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.show()


def plot_metrics(metrics):
    plt.figure(figsize=(10, 5))
    plt.bar(metrics.keys(), metrics.values(), color=['blue', 'orange', 'green',␣
 ↪'red'])
    plt.title('Model Evaluation Metrics')
    plt.ylabel('Score')
    plt.ylim(0, 1)
    plt.show()

# Define class names (assuming binary classification)
class_names = ['Non-metastatic', 'Metastatic']

plot_confusion_matrix(evaluation_results["Confusion Matrix"], class_names)
```



3.1 Interpretation of Results

- **Accuracy**: 0.6320
  The proportion of correctly predicted instances out of the total instances. This indicates that the model correctly identified 63.20% of the cases.
- **Precision**: 0.0980
  The ratio of true positive predictions to the total predicted positives. A precision of 9.80% means that when the model predicted a positive class, it was correct only 9.80% of the time.
- **Recall**: 1.0000
  The ratio of true positive predictions to the total actual positives. A recall of 100% indicates that the model identified all positive instances correctly.
- **F1 Score**: 0.1786
  The harmonic mean of precision and recall. An F1 score of 17.86% reflects a balance between precision and recall, indicating the model's overall performance in handling positive class predictions.

We see that the model has low precision, but high recall on the validation dataset. This means that a `P(+ | positive test) = 0.098` is low, and is therefore an unreliable diagnostic of cancer. We can troubleshoot this problem by further training our model and increasing our dataset size.

[25]:
```
# 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.
```

4 Metastatic Cancer Detection Model

4.1 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.

4.2 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.

4.3 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.

```
[ ]:
```