UNIVERSITI MALAYA

Faculty of Computer Science and Information Technology

WIX3001 SOFT COMPUTING

GROUP ASSIGNMENT 2: NEURAL NETWORK

SEMESTER 2 SESSION 2023/2024

LECTURER: DR LIEW WEI SHIUNG

NO	NAME	MATRIC NUMBER
1.	QUAH JUN CHUAN	22004851
2.	TER ZHEN HUANG	22004736
3.	CHENG YEE ERN	22004791
4.	RACHEL LIM	22052702

Table of Contents

1. Dataset

1.1 Introduction

Today, medical imaging has transformed the world of healthcare with the advancement of technology in providing critical insights into a variety of diseases and examining health conditions. Among them, analyzing and detecting bone fractures with X-ray images has emerged as a critical area of research. However, conventional methods of manually examining these X-ray images are not only time-consuming but also prone to errors.

In response to this challenge, our project aims to automate fracture detection by developing a classification model that can analyze X-ray images and identify whether the bone is fractured by assessing the severity and complexity of bone structures from the X-ray images. By harnessing the power of computer vision, we seek to enhance the accuracy and efficiency of fracture detection processes, ultimately leading to improved healthcare diagnosis outcomes.

1.2 About Dataset

The dataset chosen for this project “Bone Fracture Dataset” is obtained from a Kaggle Repository. The dataset comprises two main files: one for training and another for testing bone fracture detection models. Within these files are images categorized into fractured and non-fractured bones. The training set consists of 4480 fractured samples and 4383 non-fractured samples, totaling 8863 images. Conversely, the testing set contains 360 fractured samples and 240 non-fractured samples, totaling 600 images. These images will be the data used for training and evaluating our computer vision algorithms that are aimed at automated bone fracture detection.

1.3 Solution Approach

For this project, we will be utilizing the Convolutional Neural Network (CNN) for bone fracture classification by using the X-ray images as our input. Below is the CNN architecture diagram for our project.

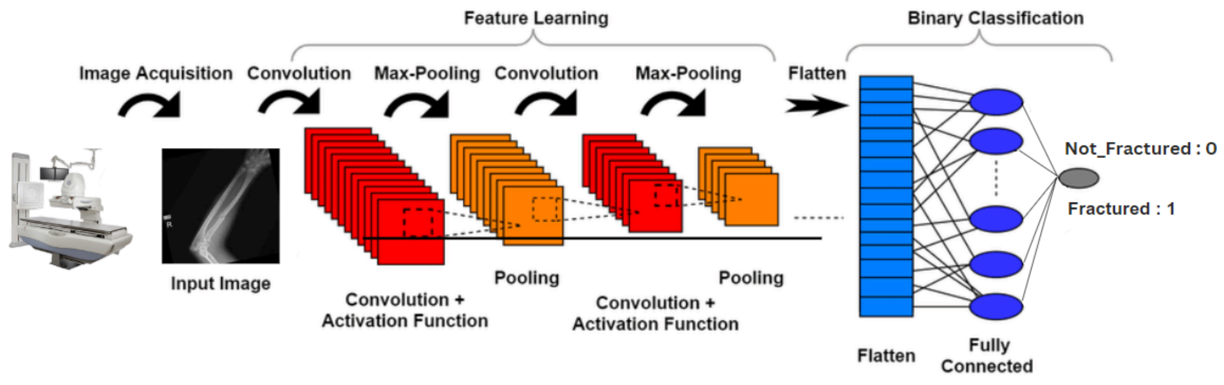


Diagram 1.3.1: Convolutional Neural Network (CNN) Architecture Diagram

CNN excels at automatically extracting meaningful features from the input images, enabling accurate classification decisions (Sharma, 2021). CNNs are tailored as below for our bone fracture classification project task:

- **Convolutional Layers:** The backbone of the CNN architecture, convolutional layers systematically apply filters to the input X-ray images. These filters capture important features such as bone edges, textures, and patterns associated with fractures. By convolving these filters across the entire image, the CNN learns to identify relevant features regardless of their spatial location.
- **Activation Function:** Following each convolutional layer, an activation function like ReLU introduces non-linearity, allowing the CNN to model complex relationships between image features. This activation function helps enhance the network's ability to detect subtle variations indicative of fractures within the X-ray images.
- **Pooling Layers:** Pooling layers play a crucial role in reducing the spatial dimensions of the feature maps obtained from the convolutional layers. By downsampling the feature maps, pooling layers retain the most salient information while simultaneously reducing computational complexity. This downsampling process ensures that the CNN remains efficient and capable of processing large volumes of medical imaging data.
- **Flattening Layer:** After applying multiple convolutional and pooling layers, the spatial dimensions of the feature maps are reduced while preserving their depth (number of channels). Before passing these feature maps to the fully connected layers, they need to be flattened into a one-dimensional vector. It will reshape the 3D feature maps into a single vector to ensure compatibility with the fully connected layers.
- **Fully Connected Layers:** Towards the end of the CNN architecture, fully connected layers process the high-level features extracted from the convolutional and pooling layers. These layers perform the final classification task by combining the learned features and making predictions about whether the input X-ray image depicts a fractured bone.
- **Output Layer:** In the case of bone fracture classification, the CNN's output layer consists of a single neuron with a sigmoid activation function. This neuron outputs the likelihood of the input image containing a fracture, providing a binary classification decision.

Optionally, we will also add regularization techniques to one of the architectures to evaluate whether they will increase the performance of the model by reducing overfitting and improving the generalization ability of neural networks. These techniques include:

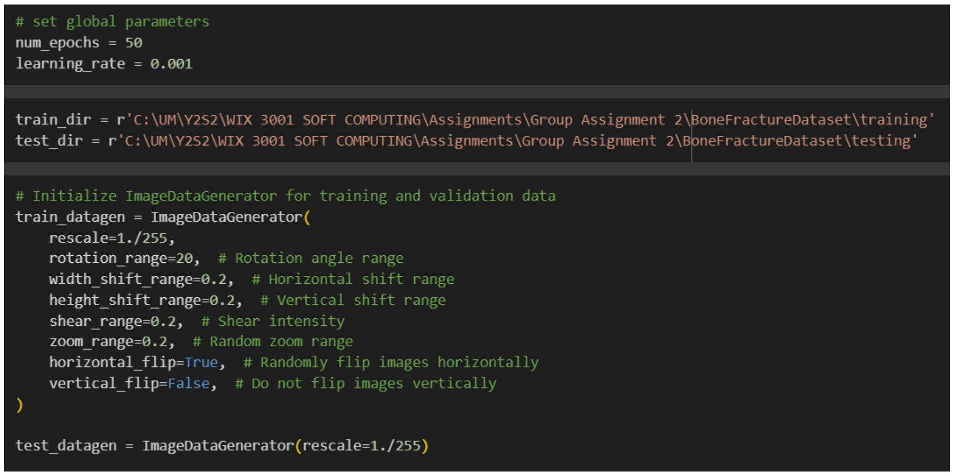
- **Dropout**: Dropout is a regularization technique used to prevent overfitting in deep neural networks, including CNN. During training, dropout randomly sets a fraction of the input units to zero, effectively "dropping out" some neurons from the network. This prevents neurons from relying too much on specific input features, forcing them to learn more robust and generalizable representations. By randomly dropping neurons during training, dropout helps prevent the network from memorizing the training data and improves its ability to generalize to unseen data.

- **Batch Normalization**: Batch normalization is another regularization technique that improves the stability and speed of training deep neural networks. It normalizes the activations of each layer by subtracting the batch mean and dividing by the batch standard deviation. This normalization step helps alleviate the internal covariate shift problem, where the distribution of activations in intermediate layers of the network shifts during training.

2. CNN Model Development

2.1 Data Loading and Preprocessing

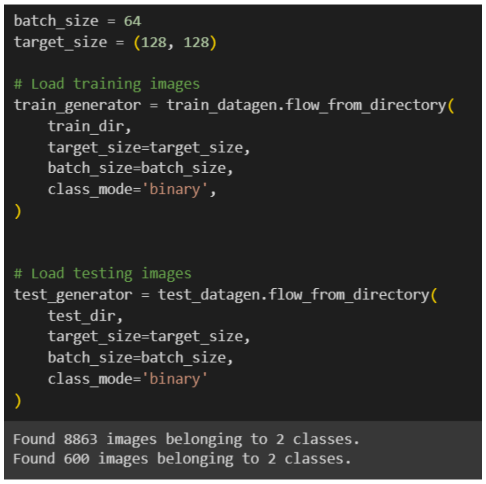
Based on Diagram 2.1.1, we began by setting up the global parameters of 50 epochs and a learning rate of 0.001. Our dataset is divided into two parts: training data and testing data, each pointing to their respective directory. We then initialize an ImageDataGenerator for our training and testing data, allowing for the preprocessing of images and the generation of batches of augmented data during training. In the Training Data Generator, we rescale the pixel values of the images to the range [0,1] by dividing them by 255, apply rotation of image angles up to 20 degrees, randomly shift images vertically and horizontally by up to 20%, apply shear intensity with a maximum intensity of 20%, zoom into images randomly by up to 20%, horizontally flip images randomly, and disable vertical flipping to maintain objects in a consistent orientation. The Testing Data Generator involves rescaling the images similarly to the training images to accurately evaluate the model's performance on unseen examples.



```
# set global parameters
num_epochs = 50
learning_rate = 0.001

train_dir = r'C:\UM\Y2S2\WIX 3001 SOFT COMPUTING\Assignments\Group Assignment 2\BoneFractureDataset\training'
test_dir = r'C:\UM\Y2S2\WIX 3001 SOFT COMPUTING\Assignments\Group Assignment 2\BoneFractureDataset\testing'

# Initialize ImageDataGenerator for training and validation data
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,  # Rotation angle range
    width_shift_range=0.2,  # Horizontal shift range
    height_shift_range=0.2,  # Vertical shift range
    shear_range=0.2,  # Shear intensity
    zoom_range=0.2,  # Random zoom range
    horizontal_flip=True,  # Randomly flip images horizontally
    vertical_flip=False,  # Do not flip images vertically
)

test_datagen = ImageDataGenerator(rescale=1./255)
```

Diagram 2.1.1 Data preprocessing

Next, we use the flow_from_directory method of the ImageDataGenerator objects to load both training and testing images from the specified directories (train_dir and test_dir) based on the batch_size of 64 and traget_size of (128, 128) pixels predefined. Upon running the code, we found 8863 images from the training data and 600 images from the testing data.

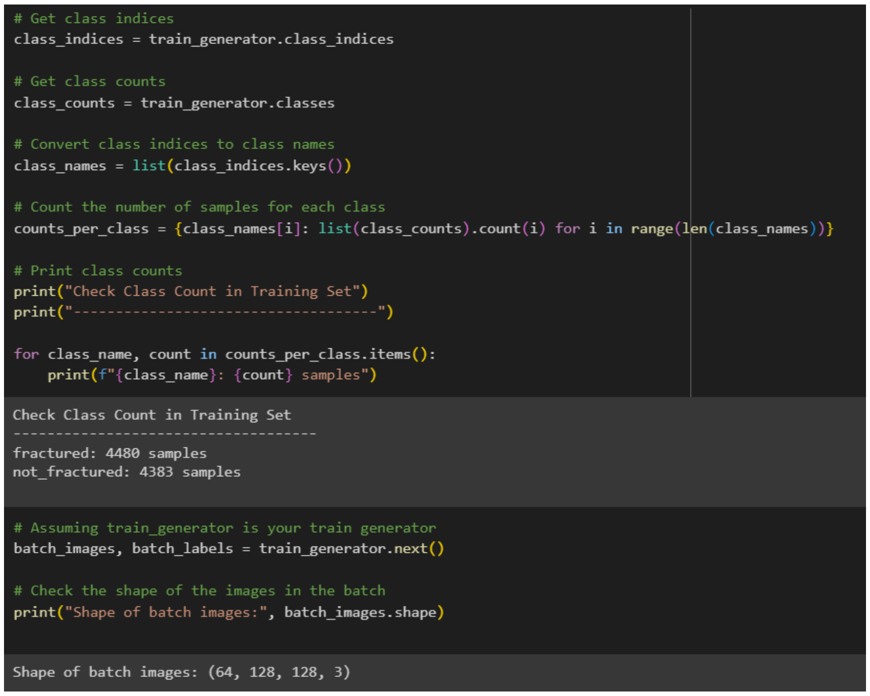


```
batch_size = 64
target_size = (128, 128)

# Load training images
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=target_size,
    batch_size=batch_size,
    class_mode='binary',
)


# Load testing images
test_generator = test_datagen.flow_from_directory(
    test_dir,
    target_size=target_size,
    batch_size=batch_size,
    class_mode='binary'
)
```

```
Found 8863 images belonging to 2 classes.
Found 600 images belonging to 2 classes.
```

Diagram 2.1.2 Data loading

Related to the training data generator, we map their class names to their corresponding integer indices, such as 'fractured': 0 and 'non_fractured': 1. We built an array for class_counts and a list for class_names. We also counted and printed the count of samples for each class and generated the next batch of data from the training generator which printed out the shape of the images in the batch.



```
# Get class indices
class_indices = train_generator.class_indices

# Get class counts
class_counts = train_generator.classes

# Convert class indices to class names
class_names = list(class_indices.keys())

# Count the number of samples for each class
counts_per_class = {class_names[i]: list(class_counts).count(i) for i in range(len(class_names))}

# Print class counts
print("Check Class Count in Training Set")
print("------------------------------------")

for class_name, count in counts_per_class.items():
    print(f"{class_name}: {count} samples")
```

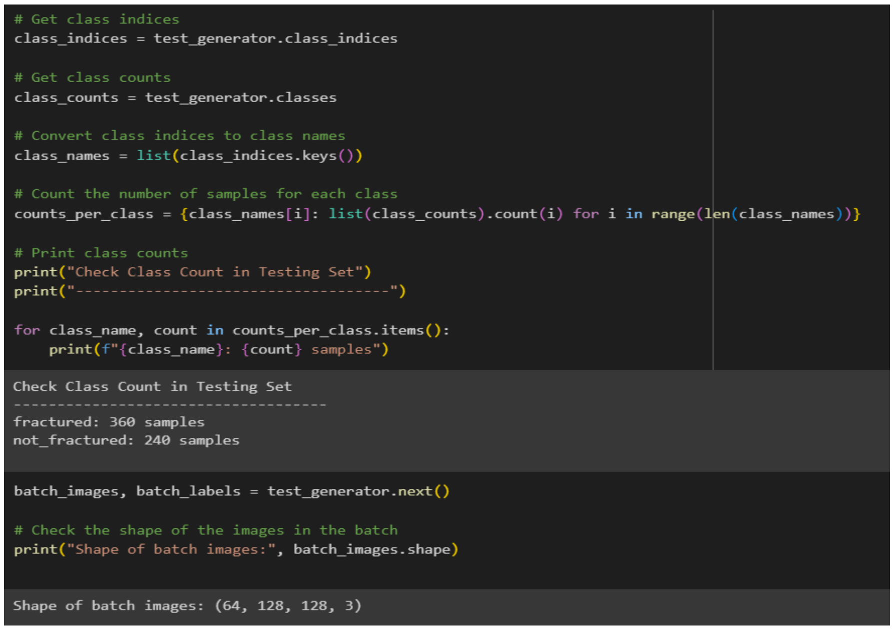
```
Check Class Count in Training Set
------------------------------------
fractured: 4480 samples
not_fractured: 4383 samples
```

```
# Assuming train_generator is your train generator
batch_images, batch_labels = train_generator.next()

# Check the shape of the images in the batch
print("Shape of batch images:", batch_images.shape)
```

```
Shape of batch images: (64, 128, 128, 3)
```

Diagram 2.1.3 Batch generation summary of training data

The code lines in diagram 2.1.4 are similar to the code lines in diagram 2.1.3, except we changed into test_generator to deal with testing data instead of training data like the previous segment of codes.



```
# Get class indices
class_indices = test_generator.class_indices

# Get class counts
class_counts = test_generator.classes

# Convert class indices to class names
class_names = list(class_indices.keys())

# Count the number of samples for each class
counts_per_class = {class_names[i]: list(class_counts).count(i) for i in range(len(class_names))}

# Print class counts
print("Check Class Count in Testing Set")
print("------------------------------------")

for class_name, count in counts_per_class.items():
    print(f"{class_name}: {count} samples")
```

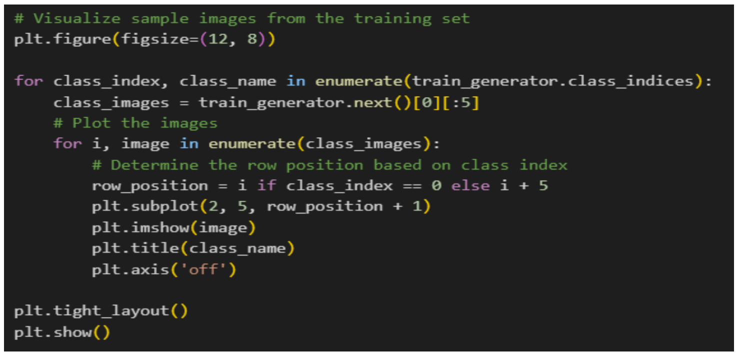
```
Check Class Count in Testing Set
------------------------------------
fractured: 360 samples
not_fractured: 240 samples
```

```
batch_images, batch_labels = test_generator.next()

# Check the shape of the images in the batch
print("Shape of batch images:", batch_images.shape)
```

```
Shape of batch images: (64, 128, 128, 3)
```

Diagram 2.1.4 Batch generation summary of testing data

Next, we visualize sample images from the training set, 5 samples from each class is shown in the figure below, labeled as Diagram 2.1.6.



```
# Visualize sample images from the training set
plt.figure(figsize=(12, 8))

for class_index, class_name in enumerate(train_generator.class_indices):
    class_images = train_generator.next()[0][:5]
    # Plot the images
    for i, image in enumerate(class_images):
        # Determine the row position based on class index
        row_position = i if class_index == 0 else i + 5
        plt.subplot(2, 5, row_position + 1)
        plt.imshow(image)
        plt.title(class_name)
        plt.axis('off')

plt.tight_layout()
plt.show()
```

Diagram 2.1.5 Visualization of sample image

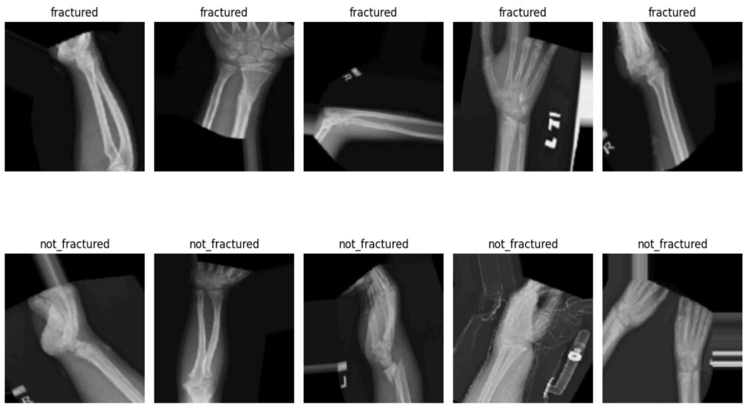
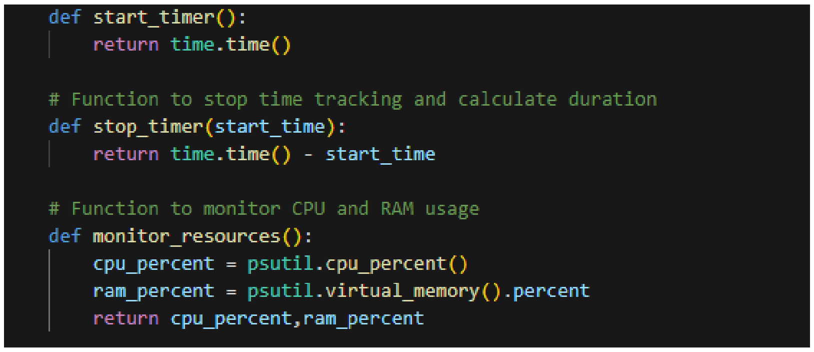


Diagram 2.1.6 Sample image

2.2 Model Architecture Design and Training

Before designing the model, we define 3 functions aimed at tracking time and monitoring system resources during the execution of tasks. The start_timer() function initiates time tracking by capturing the current time using the time.time() function. Conversely, the stop_timer(start_time) function concludes time tracking and calculates the duration by subtracting the start time from the current time. These functions facilitate accurate measurement of the time taken for specific operations or tasks. Additionally, the monitor_resources() function utilizes the psutil library to monitor CPU and RAM usage, providing insights into resource utilization during task execution.

By default, TensorFlow code will execute on the CPU unless explicitly configured to use a GPU. Although GPU monitoring is crucial for deep learning tasks due to the heavy computational load typically offloaded to the GPU, keeping track of CPU and RAM usage can also offer a comprehensive approach to system resource management and tracking model complexity for performance optimization. Thus, GPU tracking is excluded from our project this time.



```python
def start_timer():
    return time.time()

# Function to stop time tracking and calculate duration
def stop_timer(start_time):
    return time.time() - start_time

# Function to monitor CPU and RAM usage
def monitor_resources():
    cpu_percent = psutil.cpu_percent()
    ram_percent = psutil.virtual_memory().percent
    return cpu_percent,ram_percent
```

Diagram 2.2.1 Time and Resource Monitoring

Referring to Symbl (2022), for all the architectures used, a basic model is defined for both where this basic model will also be designed as our Model 1. After that, more layers or filters will be added across other models.

- Input Layer:
  - Both PyTorchModel and the TensorFlow model accept input images of size 128x128 pixels with 3 channels (RGB).

- Convolutional Layer:
  - PyTorchModel: Utilizes a single convolutional layer (conv1) with 16 output channels and a kernel size of 3x3. ReLU activation is applied after convolution.
  - TensorFlow Model: Employs a convolutional layer with 16 filters of size 3x3. The layer applies ReLU activation.

- Pooling Layer:
  - Both models employ max pooling with a pool size of 2x2 and a stride of 2, reducing the spatial dimensions of the feature maps by half. This layer reduces

the spatial dimensions of the output of the previous layer, which helps in reducing computation while retaining important features.

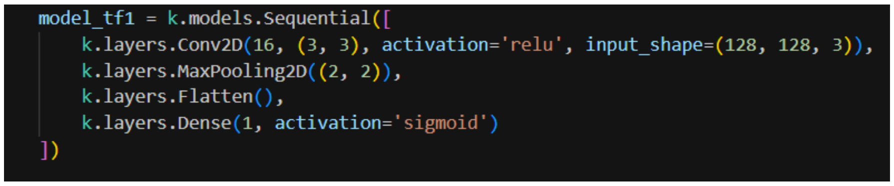
- Flattening:
  - PyTorchModel: The output from the convolutional and pooling layers is flattened into a vector using the flatten operation, resulting in a shape compatible with the fully connected layer.
  - TensorFlow Model: Utilizes the Flatten layer to flatten the output from the convolutional and pooling layers into a vector.

- Fully Connected Layer:
  - PyTorchModel: Includes a single fully connected layer (fc1) with 16 * 64 * 64 input features and 1 output feature, followed by a sigmoid activation function.
  - TensorFlow Model: Consists of a single dense layer with 1 neuron and a sigmoid activation function.

- Output:
  - Both models produce a single output representing the probability of the input belonging to a particular class, with a sigmoid activation function applied to the final layer's output

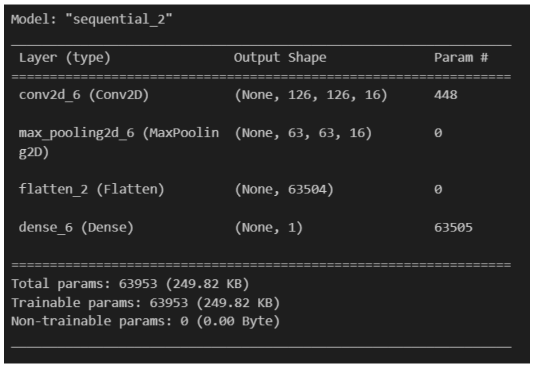
2.2.1 Model Architecture 1

- The first layer is a convolutional layer (Conv2D).
- A max-pooling layer (MaxPooling2D) is added.
- The output of the max-pooling layer is flattened to be fed into the fully connected layer.
- The fully connected layer with 1 neuron/output feature is produced.



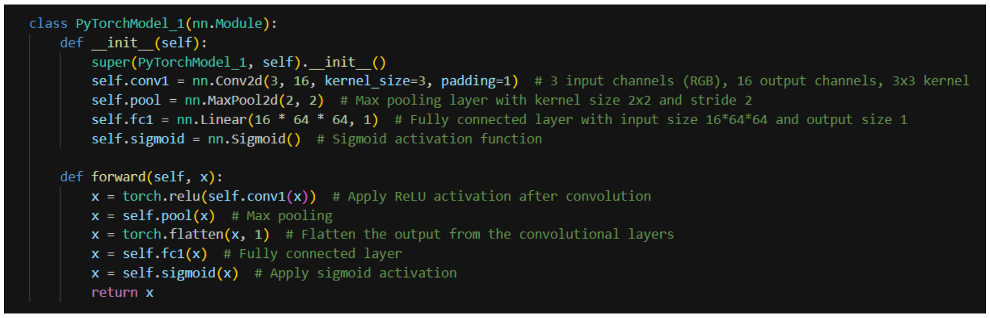
```
model_tf1 = k.models.Sequential([
    k.layers.Conv2D(16, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    k.layers.MaxPooling2D((2, 2)),
    k.layers.Flatten(),
    k.layers.Dense(1, activation='sigmoid')
])
```

Diagram 2.2.1.1: Tensorflow Model 1



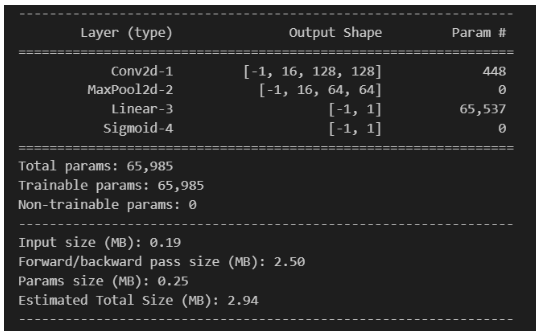
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 126, 126, 16)      448

 max_pooling2d_6 (MaxPoolin  (None, 63, 63, 16)        0
 g2D)

 flatten_2 (Flatten)         (None, 63504)             0

 dense_6 (Dense)             (None, 1)                 63505

=================================================================
Total params: 63953 (249.82 KB)
Trainable params: 63953 (249.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Diagram 2.2.1.2: Tensorflow Model 1 Summary



```python
class PyTorchModel_1(nn.Module):
    def __init__(self):
        super(PyTorchModel_1, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)  # 3 input channels (RGB), 16 output channels, 3x3 kernel
        self.pool = nn.MaxPool2d(2, 2)  # Max pooling layer with kernel size 2x2 and stride 2
        self.fc1 = nn.Linear(16 * 64 * 64, 1)  # Fully connected layer with input size 16*64*64 and output size 1
        self.sigmoid = nn.Sigmoid()  # Sigmoid activation function

    def forward(self, x):
        x = torch.relu(self.conv1(x))  # Apply ReLU activation after convolution
        x = self.pool(x)  # Max pooling
        x = torch.flatten(x, 1)  # Flatten the output from the convolutional layers
        x = self.fc1(x)  # Fully connected layer
        x = self.sigmoid(x)  # Apply sigmoid activation
        return x
```

Diagram 2.2.1.3: Pytorch Model 1

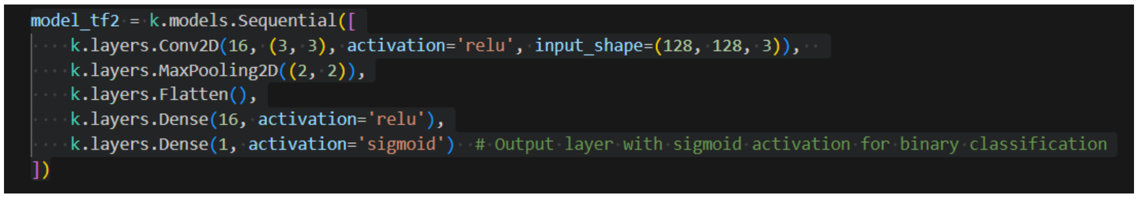


```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 128, 128]             448
         MaxPool2d-2           [-1, 16, 64, 64]               0
            Linear-3                    [-1, 1]          65,537
           Sigmoid-4                    [-1, 1]               0
================================================================
Total params: 65,985
Trainable params: 65,985
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 2.50
Params size (MB): 0.25
Estimated Total Size (MB): 2.94
----------------------------------------------------------------
```

Diagram 2.2.1.4: Pytorch Model Summary 1

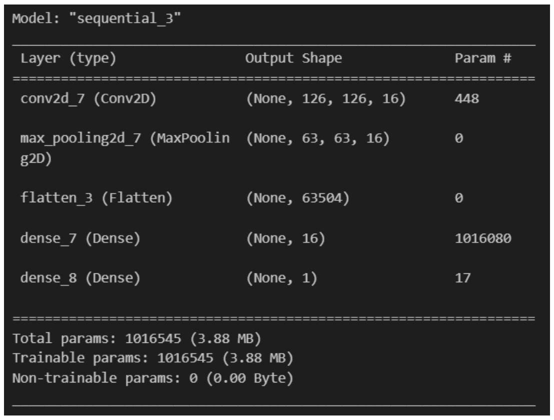
2.2.2 Model Architecture 2

- The convolutional layer, max-pooling layer and flattening of the output of the max-pooling layer remain the same as Architecture 1
- An additional fully connected layer with 16 neurons/output features
- Finally, a fully connected layer with a single neuron/output feature is added.



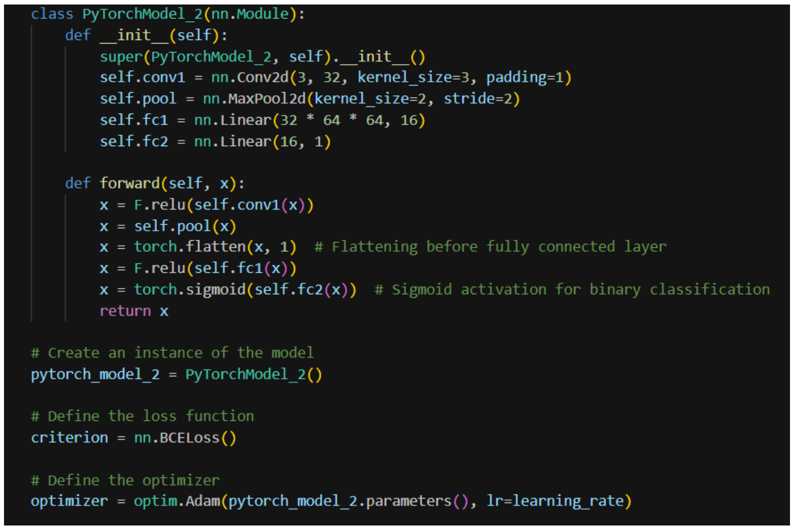
```python
model_tf2 = k.models.Sequential([
    k.layers.Conv2D(16, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    k.layers.MaxPooling2D((2, 2)),
    k.layers.Flatten(),
    k.layers.Dense(16, activation='relu'),
    k.layers.Dense(1, activation='sigmoid')  # Output layer with sigmoid activation for binary classification
])
```

Diagram 2.2.2.1: Tensorflow Model 2



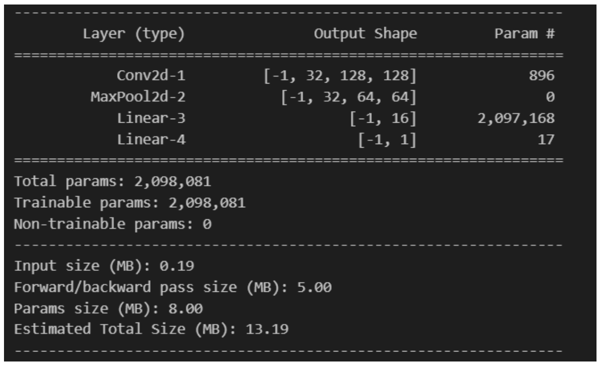
```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_7 (Conv2D)           (None, 126, 126, 16)      448

 max_pooling2d_7 (MaxPoolin  (None, 63, 63, 16)        0
 g2D)

 flatten_3 (Flatten)         (None, 63504)             0

 dense_7 (Dense)             (None, 16)                1016080

 dense_8 (Dense)             (None, 1)                 17

=================================================================
Total params: 1016545 (3.88 MB)
Trainable params: 1016545 (3.88 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Diagram 2.2.2.2: Tensorflow Model 2 Summary



```
class PyTorchModel_2(nn.Module):
    def __init__(self):
        super(PyTorchModel_2, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(32 * 64 * 64, 16)
        self.fc2 = nn.Linear(16, 1)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = torch.flatten(x, 1)  # Flattening before fully connected layer
        x = F.relu(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))  # Sigmoid activation for binary classification
        return x

# Create an instance of the model
pytorch_model_2 = PyTorchModel_2()

# Define the loss function
criterion = nn.BCELoss()

# Define the optimizer
optimizer = optim.Adam(pytorch_model_2.parameters(), lr=learning_rate)
```

Diagram 2.2.2.3: Pytorch Model 2

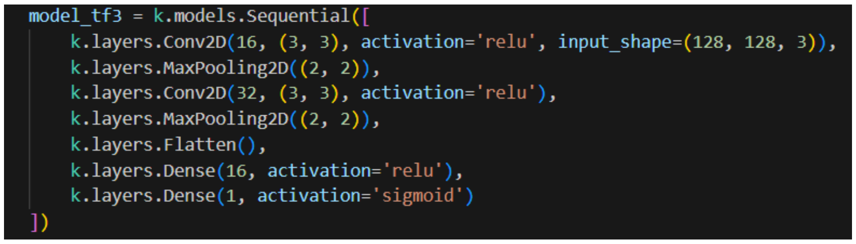


```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 128, 128]             896
         MaxPool2d-2           [-1, 32, 64, 64]               0
            Linear-3                   [-1, 16]       2,097,168
            Linear-4                    [-1, 1]              17
================================================================
Total params: 2,098,081
Trainable params: 2,098,081
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 5.00
Params size (MB): 8.00
Estimated Total Size (MB): 13.19
----------------------------------------------------------------
```

Diagram 2.2.2.4: Pytorch Model 2 Summary

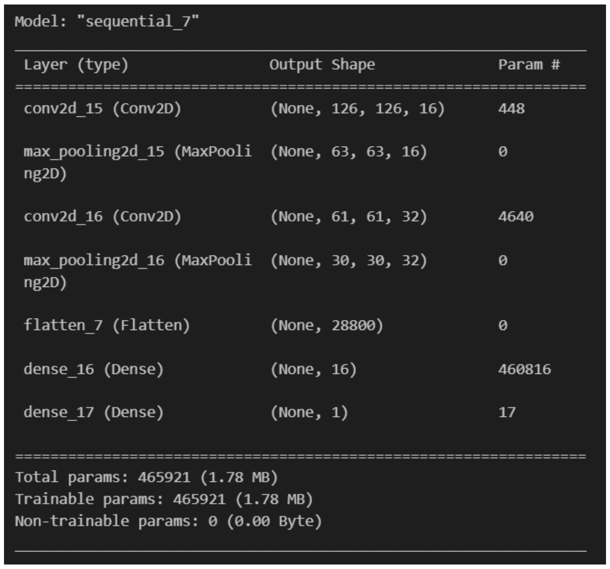
2.2.3 Model Architecture 3

- The convolutional layer and max-pooling layer remain the same as the previous architectures.
- Following the max-pooling layer, an additional convolutional layer and max-pooling layer with the same configuration are added. This further increases the depth of the model, allowing it to learn more abstract features.
- After the additional convolutional and max-pooling layer, the flattening of output and the fully connected layers remain the same as in Architecture 2.



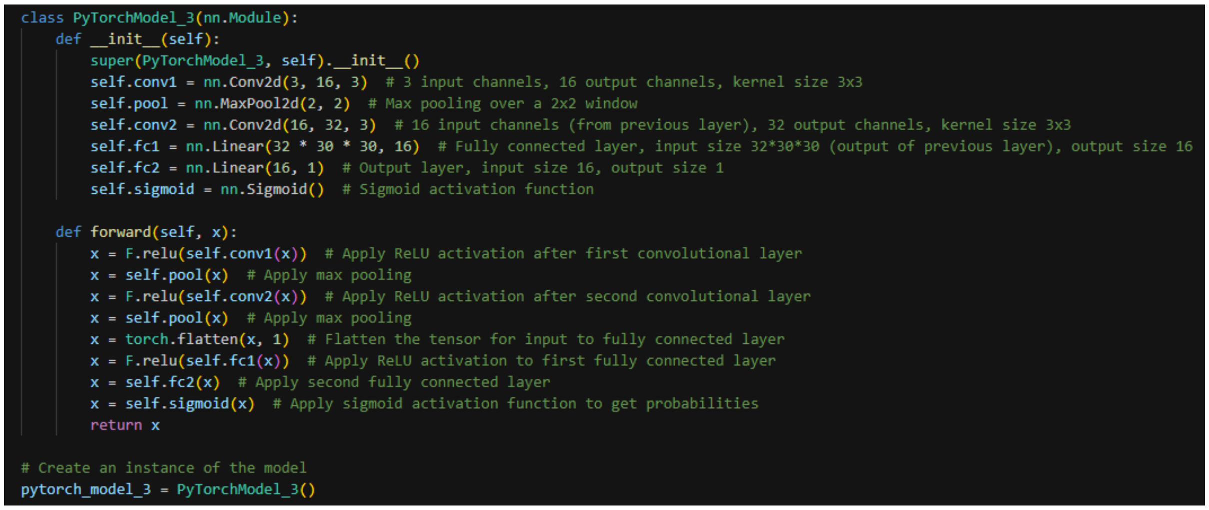
```
model_tf3 = k.models.Sequential([
    k.layers.Conv2D(16, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    k.layers.MaxPooling2D((2, 2)),
    k.layers.Conv2D(32, (3, 3), activation='relu'),
    k.layers.MaxPooling2D((2, 2)),
    k.layers.Flatten(),
    k.layers.Dense(16, activation='relu'),
    k.layers.Dense(1, activation='sigmoid')
])
```

Diagram 2.2.3.1:Tensorflow Model 3



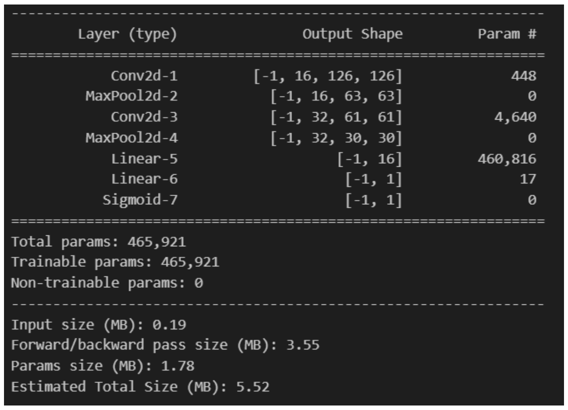
```
Model: "sequential_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_15 (Conv2D)          (None, 126, 126, 16)      448

 max_pooling2d_15 (MaxPooli  (None, 63, 63, 16)        0
 ng2D)

 conv2d_16 (Conv2D)          (None, 61, 61, 32)        4640

 max_pooling2d_16 (MaxPooli  (None, 30, 30, 32)        0
 ng2D)

 flatten_7 (Flatten)         (None, 28800)             0

 dense_16 (Dense)            (None, 16)                460816

 dense_17 (Dense)            (None, 1)                 17

=================================================================
Total params: 465921 (1.78 MB)
Trainable params: 465921 (1.78 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Diagram 2.2.3.2: Tensorflow Model 3 Summary



```python
class PyTorchModel_3(nn.Module):
    def __init__(self):
        super(PyTorchModel_3, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3)  # 3 input channels, 16 output channels, kernel size 3x3
        self.pool = nn.MaxPool2d(2, 2)  # Max pooling over a 2x2 window
        self.conv2 = nn.Conv2d(16, 32, 3)  # 16 input channels (from previous layer), 32 output channels, kernel size 3x3
        self.fc1 = nn.Linear(32 * 30 * 30, 16)  # Fully connected layer, input size 32*30*30 (output of previous layer), output size 16
        self.fc2 = nn.Linear(16, 1)  # Output layer, input size 16, output size 1
        self.sigmoid = nn.Sigmoid()  # Sigmoid activation function

    def forward(self, x):
        x = F.relu(self.conv1(x))  # Apply ReLU activation after first convolutional layer
        x = self.pool(x)  # Apply max pooling
        x = F.relu(self.conv2(x))  # Apply ReLU activation after second convolutional layer
        x = self.pool(x)  # Apply max pooling
        x = torch.flatten(x, 1)  # Flatten the tensor for input to fully connected layer
        x = F.relu(self.fc1(x))  # Apply ReLU activation to first fully connected layer
        x = self.fc2(x)  # Apply second fully connected layer
        x = self.sigmoid(x)  # Apply sigmoid activation function to get probabilities
        return x

# Create an instance of the model
pytorch_model_3 = PyTorchModel_3()
```

Diagram 2.2.3.3: Pytorch Model 3

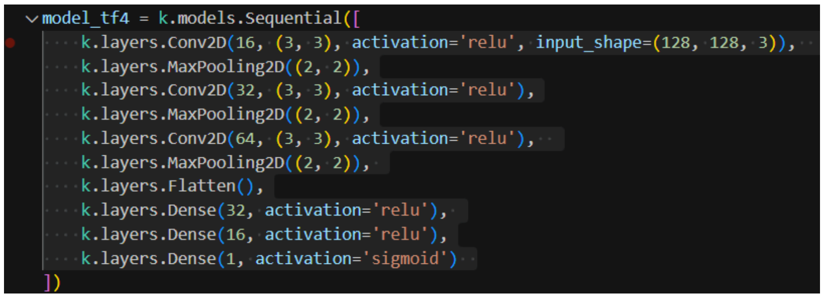


```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 126, 126]             448
         MaxPool2d-2           [-1, 16, 63, 63]               0
            Conv2d-3           [-1, 32, 61, 61]           4,640
         MaxPool2d-4           [-1, 32, 30, 30]               0
            Linear-5                   [-1, 16]         460,816
            Linear-6                    [-1, 1]              17
           Sigmoid-7                    [-1, 1]               0
================================================================
Total params: 465,921
Trainable params: 465,921
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 3.55
Params size (MB): 1.78
Estimated Total Size (MB): 5.52
```

Diagram 2.2.3.4: Pytorch Model 3 Summary

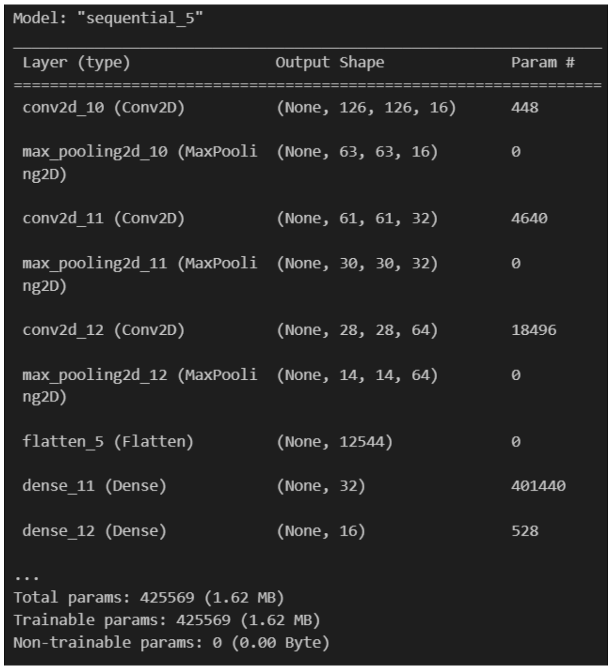
2.2.4 Model Architecture 4

- The double layers of the convolutional layer and max-pooling layer remain the same as in Architecture 3.
- Following the last max-pooling layer, an additional convolutional layer and max-pooling layer with the same configuration are added again and flattened. This further increases the depth of the model, allowing it to learn more abstract features.
- After flattening, an additional fully connected layer with 32 neurons/output functions is added to serve as an additional hidden layer.
- The subsequent double fully connected layers from Architecture 3 remain the same.



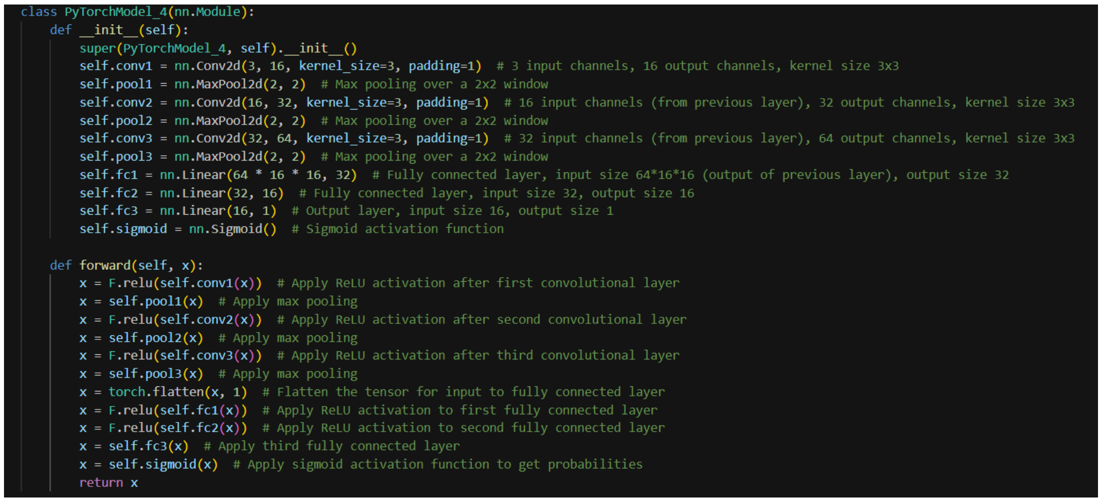
```
model_tf4 = k.models.Sequential([
    k.layers.Conv2D(16, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    k.layers.MaxPooling2D((2, 2)),
    k.layers.Conv2D(32, (3, 3), activation='relu'),
    k.layers.MaxPooling2D((2, 2)),
    k.layers.Conv2D(64, (3, 3), activation='relu'),
    k.layers.MaxPooling2D((2, 2)),
    k.layers.Flatten(),
    k.layers.Dense(32, activation='relu'),
    k.layers.Dense(16, activation='relu'),
    k.layers.Dense(1, activation='sigmoid')
])
```

Diagram 2.2.4.1: Tensorflow Model 4



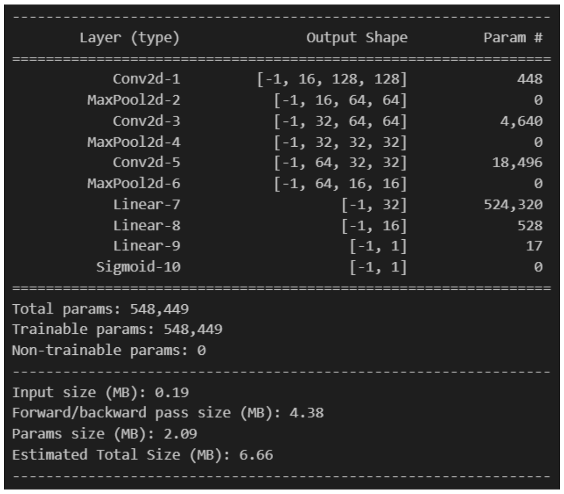
```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 126, 126, 16)      448

 max_pooling2d_10 (MaxPooli  (None, 63, 63, 16)        0
 ng2D)

 conv2d_11 (Conv2D)          (None, 61, 61, 32)        4640

 max_pooling2d_11 (MaxPooli  (None, 30, 30, 32)        0
 ng2D)

 conv2d_12 (Conv2D)          (None, 28, 28, 64)        18496

 max_pooling2d_12 (MaxPooli  (None, 14, 14, 64)        0
 ng2D)

 flatten_5 (Flatten)         (None, 12544)             0

 dense_11 (Dense)            (None, 32)                401440

 dense_12 (Dense)            (None, 16)                528

...
Total params: 425569 (1.62 MB)
Trainable params: 425569 (1.62 MB)
Non-trainable params: 0 (0.00 Byte)
```

Diagram 2.2.4.2: Tensorflow Model 4 Summary



```python
class PyTorchModel_4(nn.Module):
    def __init__(self):
        super(PyTorchModel_4, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)  # 3 input channels, 16 output channels, kernel size 3x3
        self.pool1 = nn.MaxPool2d(2, 2)  # Max pooling over a 2x2 window
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)  # 16 input channels (from previous layer), 32 output channels, kernel size 3x3
        self.pool2 = nn.MaxPool2d(2, 2)  # Max pooling over a 2x2 window
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)  # 32 input channels (from previous layer), 64 output channels, kernel size 3x3
        self.pool3 = nn.MaxPool2d(2, 2)  # Max pooling over a 2x2 window
        self.fc1 = nn.Linear(64 * 16 * 16, 32)  # Fully connected layer, input size 64*16*16 (output of previous layer), output size 32
        self.fc2 = nn.Linear(32, 16)  # Fully connected layer, input size 32, output size 16
        self.fc3 = nn.Linear(16, 1)  # Output layer, input size 16, output size 1
        self.sigmoid = nn.Sigmoid()  # Sigmoid activation function

    def forward(self, x):
        x = F.relu(self.conv1(x))  # Apply ReLU activation after first convolutional layer
        x = self.pool1(x)  # Apply max pooling
        x = F.relu(self.conv2(x))  # Apply ReLU activation after second convolutional layer
        x = self.pool2(x)  # Apply max pooling
        x = F.relu(self.conv3(x))  # Apply ReLU activation after third convolutional layer
        x = self.pool3(x)  # Apply max pooling
        x = torch.flatten(x, 1)  # Flatten the tensor for input to fully connected layer
        x = F.relu(self.fc1(x))  # Apply ReLU activation to first fully connected layer
        x = F.relu(self.fc2(x))  # Apply ReLU activation to second fully connected layer
        x = self.fc3(x)  # Apply third fully connected layer
        x = self.sigmoid(x)  # Apply sigmoid activation function to get probabilities
        return x
```

Diagram 2.2.4.3: Pytorch Model 4

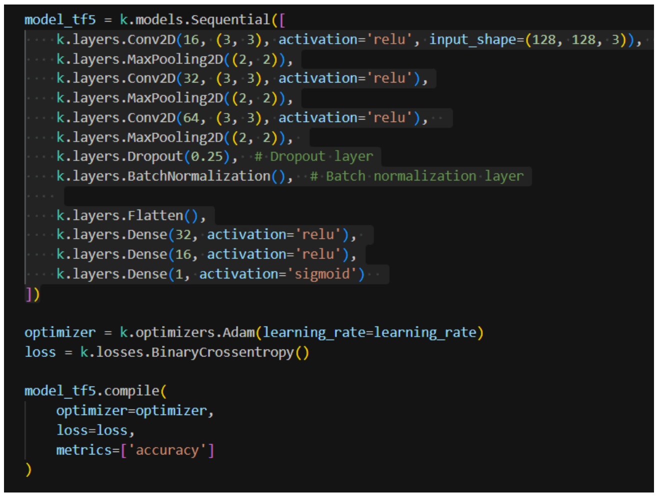


```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 128, 128]             448
         MaxPool2d-2           [-1, 16, 64, 64]               0
            Conv2d-3           [-1, 32, 64, 64]           4,640
         MaxPool2d-4           [-1, 32, 32, 32]               0
            Conv2d-5           [-1, 64, 32, 32]          18,496
         MaxPool2d-6           [-1, 64, 16, 16]               0
            Linear-7                   [-1, 32]         524,320
            Linear-8                   [-1, 16]             528
            Linear-9                    [-1, 1]              17
          Sigmoid-10                    [-1, 1]               0
================================================================
Total params: 548,449
Trainable params: 548,449
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 4.38
Params size (MB): 2.09
Estimated Total Size (MB): 6.66
----------------------------------------------------------------
```

Diagram 2.2.4.4 :Pytorch Model 4 Summary

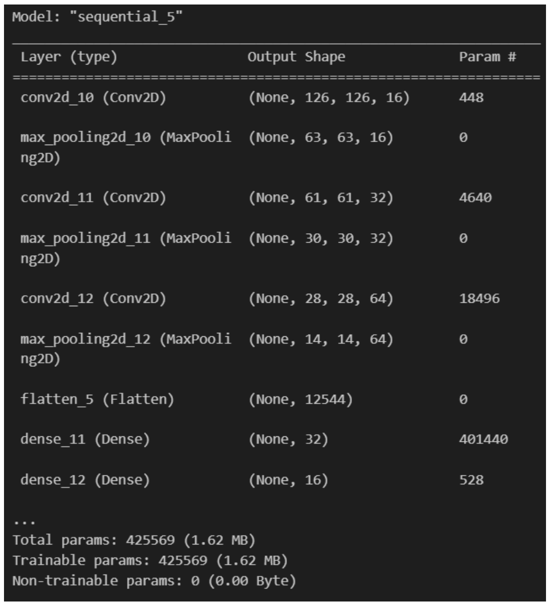
2.2.5 Model Architecture 5

- The triple layers of the convolutional layer and max-pooling layer remain the same as in Architecture 4.
- Following the last max-pooling layer, a dropout layer with a dropout rate of 0.25 and a batch normalization layer is added. Dropout helps in regularizing the model by randomly setting a fraction of input units to zero during training, which helps prevent overfitting; while Batch normalization normalizes the activations of the previous layer at each batch, which can help stabilize and speed up the training process.
- The output is flattened and the triple fully connected layers are added as in Architecture 4.



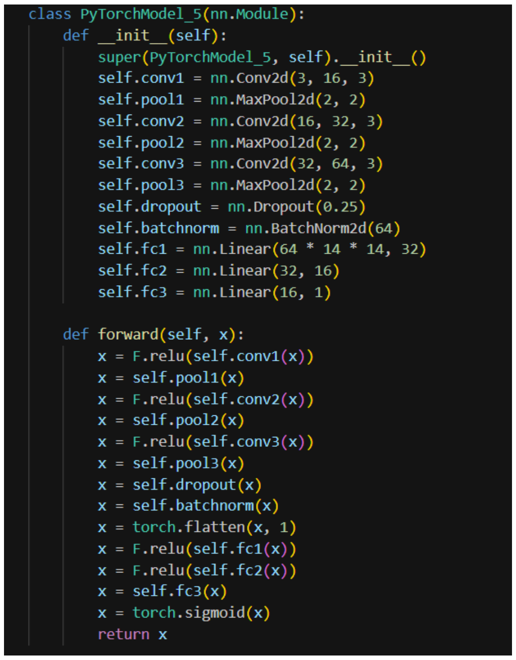
```python
model_tf5 = k.models.Sequential([
    k.layers.Conv2D(16, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    k.layers.MaxPooling2D((2, 2)),
    k.layers.Conv2D(32, (3, 3), activation='relu'),
    k.layers.MaxPooling2D((2, 2)),
    k.layers.Conv2D(64, (3, 3), activation='relu'),
    k.layers.MaxPooling2D((2, 2)),
    k.layers.Dropout(0.25),  # Dropout layer
    k.layers.BatchNormalization(),  # Batch normalization layer
    
    k.layers.Flatten(),
    k.layers.Dense(32, activation='relu'),
    k.layers.Dense(16, activation='relu'),
    k.layers.Dense(1, activation='sigmoid')
])

optimizer = k.optimizers.Adam(learning_rate=learning_rate)
loss = k.losses.BinaryCrossentropy()

model_tf5.compile(
    optimizer=optimizer,
    loss=loss,
    metrics=['accuracy']
)
```

Diagram 2.2.5.1: Tensorflow Model 5



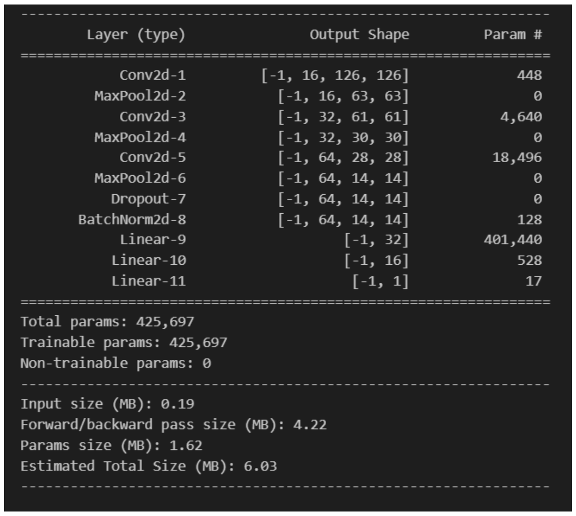
```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 126, 126, 16)      448

 max_pooling2d_10 (MaxPooli  (None, 63, 63, 16)        0
 ng2D)

 conv2d_11 (Conv2D)          (None, 61, 61, 32)        4640

 max_pooling2d_11 (MaxPooli  (None, 30, 30, 32)        0
 ng2D)

 conv2d_12 (Conv2D)          (None, 28, 28, 64)        18496

 max_pooling2d_12 (MaxPooli  (None, 14, 14, 64)        0
 ng2D)

 flatten_5 (Flatten)         (None, 12544)             0

 dense_11 (Dense)            (None, 32)                401440

 dense_12 (Dense)            (None, 16)                528

...
Total params: 425569 (1.62 MB)
Trainable params: 425569 (1.62 MB)
Non-trainable params: 0 (0.00 Byte)
```

Diagram 2.2.5.2: Tensorflow Model 5 Summary



```python
class PyTorchModel_5(nn.Module):
    def __init__(self):
        super(PyTorchModel_5, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, 3)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(32, 64, 3)
        self.pool3 = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.25)
        self.batchnorm = nn.BatchNorm2d(64)
        self.fc1 = nn.Linear(64 * 14 * 14, 32)
        self.fc2 = nn.Linear(32, 16)
        self.fc3 = nn.Linear(16, 1)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = F.relu(self.conv3(x))
        x = self.pool3(x)
        x = self.dropout(x)
        x = self.batchnorm(x)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        x = torch.sigmoid(x)
        return x
```

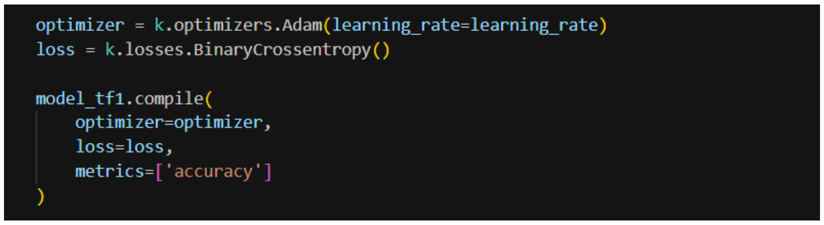
Diagram 2.2.5.3: Pytorch Model 5



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1        [-1, 16, 126, 126]             448
         MaxPool2d-2          [-1, 16, 63, 63]               0
            Conv2d-3          [-1, 32, 61, 61]           4,640
         MaxPool2d-4          [-1, 32, 30, 30]               0
            Conv2d-5          [-1, 64, 28, 28]          18,496
         MaxPool2d-6          [-1, 64, 14, 14]               0
           Dropout-7          [-1, 64, 14, 14]               0
       BatchNorm2d-8          [-1, 64, 14, 14]             128
            Linear-9                  [-1, 32]         401,440
           Linear-10                  [-1, 16]             528
           Linear-11                   [-1, 1]              17
================================================================
Total params: 425,697
Trainable params: 425,697
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 4.22
Params size (MB): 1.62
Estimated Total Size (MB): 6.03
----------------------------------------------------------------
```

Diagram 2.2.5.4: Pytorch Model 5 Summary

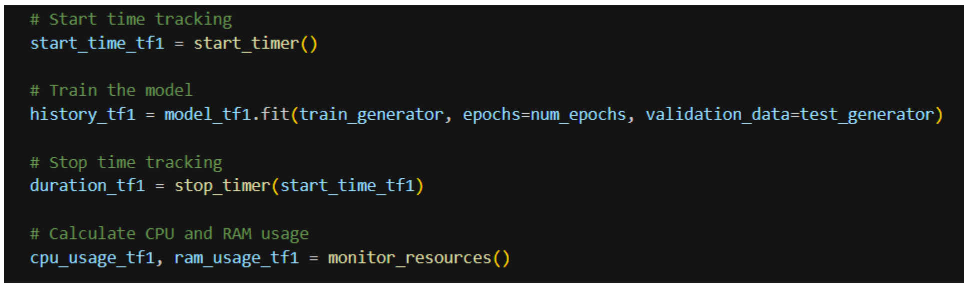
After all the **TensorFlow** architectures are defined, the code in Diagram 2.2.2 is executed to compile the complete model. An Adam optimizer is configured with the global learning rate set earlier which is 0.001, and the Binary cross-entropy is chosen as the loss function. It is commonly used for binary classification problems and measures the difference between the predicted probabilities and the true binary labels. The model is compiled using the optimizer and loss function defined previously, while the accuracy metric is used to evaluate the model performance.



```
optimizer = k.optimizers.Adam(learning_rate=learning_rate)
loss = k.losses.BinaryCrossentropy()

model_tf1.compile(
    optimizer=optimizer,
    loss=loss,
    metrics=['accuracy']
)
```

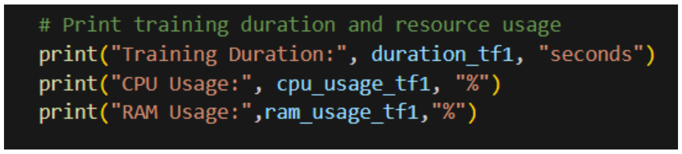
Diagram 2.2.2 : Model Compilation for Tensorflow

The 3 functions defined to track time and monitor system resources during the execution of tasks are applied in Diagram 2.2.3 and the result is printed out with the code in Diagram 2.2.4.



```
# Start time tracking
start_time_tf1 = start_timer()

# Train the model
history_tf1 = model_tf1.fit(train_generator, epochs=num_epochs, validation_data=test_generator)

# Stop time tracking
duration_tf1 = stop_timer(start_time_tf1)

# Calculate CPU and RAM usage
cpu_usage_tf1, ram_usage_tf1 = monitor_resources()
```

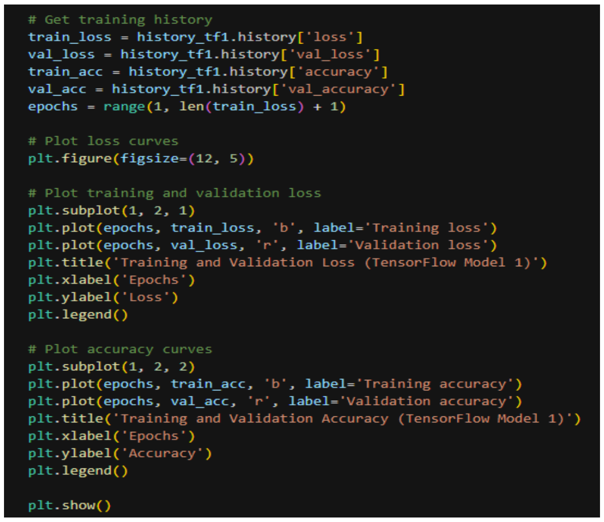
Diagram 2.2.3 : Resource and Time Monitoring and Training of Tensorflow Model



```
# Print training duration and resource usage
print("Training Duration:", duration_tf1, "seconds")
print("CPU Usage:", cpu_usage_tf1, "%")
print("RAM Usage:",ram_usage_tf1,"%")
```

Diagram 2.2.4 : Print Resource and Time Usage

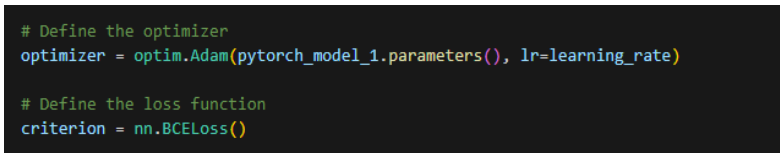
After the model is trained and validated, we visualize 2 subplots: one displaying the training and validation loss over epochs, and the other showing the training and validation accuracy over epochs for each model to evaluate these metrics change during the training process thus providing insights into evaluating the model's performance and convergence.



```
# Get training history
train_loss = history_tf1.history['loss']
val_loss = history_tf1.history['val_loss']
train_acc = history_tf1.history['accuracy']
val_acc = history_tf1.history['val_accuracy']
epochs = range(1, len(train_loss) + 1)

# Plot loss curves
plt.figure(figsize=(12, 5))

# Plot training and validation loss
plt.subplot(1, 2, 1)
plt.plot(epochs, train_loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and Validation Loss (TensorFlow Model 1)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plot accuracy curves
plt.subplot(1, 2, 2)
plt.plot(epochs, train_acc, 'b', label='Training accuracy')
plt.plot(epochs, val_acc, 'r', label='Validation accuracy')
plt.title('Training and Validation Accuracy (TensorFlow Model 1)')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

Diagram 2.2.5 : Visualization of training and validation loss over epochs

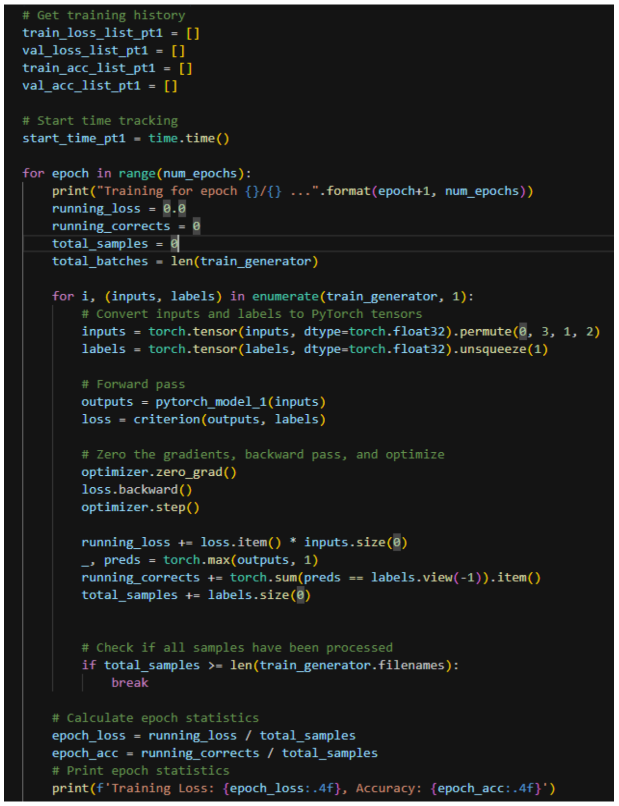
After all the **Pytorch** architectures are defined, we start the model training process with the same Adam optimizer, which is configured with the global learning rate set to 0.001, and the Binary cross-entropy as the loss function shown in Diagram 2.2.6.



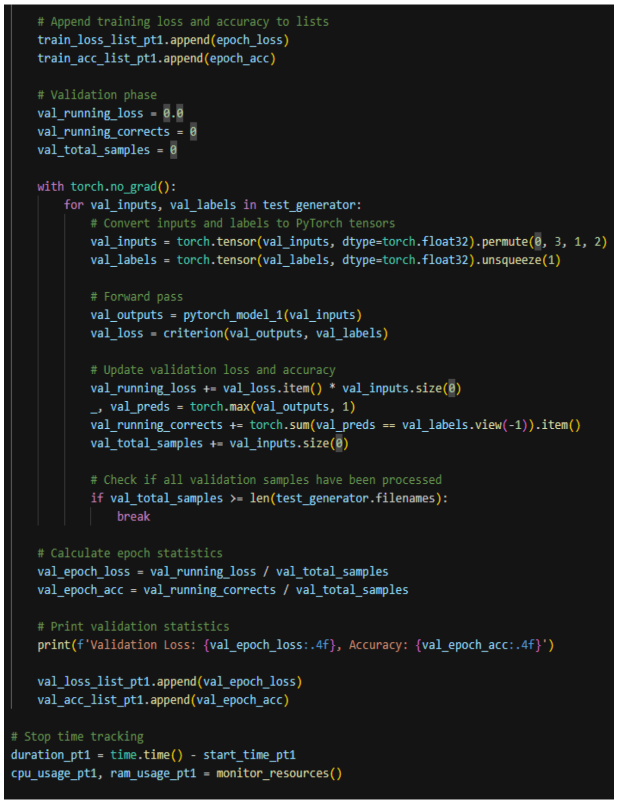
```python
# Define the optimizer
optimizer = optim.Adam(pytorch_model_1.parameters(), lr=learning_rate)

# Define the loss function
criterion = nn.BCELoss()
```

Diagram 2.2.6 Model Compilation of PyTorch Model

The code in Diagram 2.2.7 shows a training loop for all PyTorch models. It iterates over the predefined number of epochs, which is 50, wherein each epoch comprises a training phase and a validation phase. The loop breaks if all samples have been processed. Within each epoch, the training data is passed through the model in batches of size 64. Predictions are computed, and the binary cross-entropy loss is calculated, updating the model parameters through the backpropagation process. Average training and validation loss, along with accuracy, are computed and printed for each epoch. Similarly, the validation phase evaluates the model's performance on the validation dataset without altering the model parameters. Additionally, lists are maintained to track the training and validation loss and accuracy across epochs, facilitating the plotting of epoch graphs later. At the end of training, the duration of training and resource usage (CPU and RAM) are recorded, enabling the monitoring and evaluation of the model's performance over multiple training iterations.



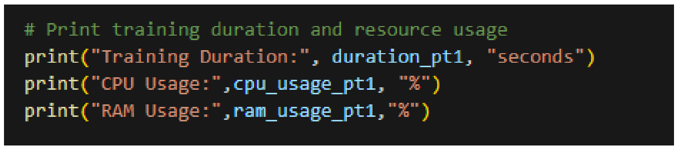
```python
# Get training history
train_loss_list_pt1 = []
val_loss_list_pt1 = []
train_acc_list_pt1 = []
val_acc_list_pt1 = []

# Start time tracking
start_time_pt1 = time.time()

for epoch in range(num_epochs):
    print("Training for epoch {}/{} ...".format(epoch+1, num_epochs))
    running_loss = 0.0
    running_corrects = 0
    total_samples = 0
    total_batches = len(train_generator)

    for i, (inputs, labels) in enumerate(train_generator, 1):
        # Convert inputs and labels to PyTorch tensors
        inputs = torch.tensor(inputs, dtype=torch.float32).permute(0, 3, 1, 2)
        labels = torch.tensor(labels, dtype=torch.float32).unsqueeze(1)

        # Forward pass
        outputs = pytorch_model_1(inputs)
        loss = criterion(outputs, labels)

        # Zero the gradients, backward pass, and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * inputs.size(0)
        _, preds = torch.max(outputs, 1)
        running_corrects += torch.sum(preds == labels.view(-1)).item()
        total_samples += labels.size(0)


        # Check if all samples have been processed
        if total_samples >= len(train_generator.filenames):
            break

    # Calculate epoch statistics
    epoch_loss = running_loss / total_samples
    epoch_acc = running_corrects / total_samples
    # Print epoch statistics
    print(f'Training Loss: {epoch_loss:.4f}, Accuracy: {epoch_acc:.4f}')
```



```python
    # Append training loss and accuracy to lists
    train_loss_list_pt1.append(epoch_loss)
    train_acc_list_pt1.append(epoch_acc)

    # Validation phase
    val_running_loss = 0.0
    val_running_corrects = 0
    val_total_samples = 0

    with torch.no_grad():
        for val_inputs, val_labels in test_generator:
            # Convert inputs and labels to PyTorch tensors
            val_inputs = torch.tensor(val_inputs, dtype=torch.float32).permute(0, 3, 1, 2)
            val_labels = torch.tensor(val_labels, dtype=torch.float32).unsqueeze(1)

            # Forward pass
            val_outputs = pytorch_model_1(val_inputs)
            val_loss = criterion(val_outputs, val_labels)

            # Update validation loss and accuracy
            val_running_loss += val_loss.item() * val_inputs.size(0)
            _, val_preds = torch.max(val_outputs, 1)
            val_running_corrects += torch.sum(val_preds == val_labels.view(-1)).item()
            val_total_samples += val_inputs.size(0)

            # Check if all validation samples have been processed
            if val_total_samples >= len(test_generator.filenames):
                break

    # Calculate epoch statistics
    val_epoch_loss = val_running_loss / val_total_samples
    val_epoch_acc = val_running_corrects / val_total_samples

    # Print validation statistics
    print(f'Validation Loss: {val_epoch_loss:.4f}, Accuracy: {val_epoch_acc:.4f}')

    val_loss_list_pt1.append(val_epoch_loss)
    val_acc_list_pt1.append(val_epoch_acc)

# Stop time tracking
duration_pt1 = time.time() - start_time_pt1
cpu_usage_pt1, ram_usage_pt1 = monitor_resources()
```

Diagram 2.2.7 : Resource and Time Monitoring and training of Pytorch Model

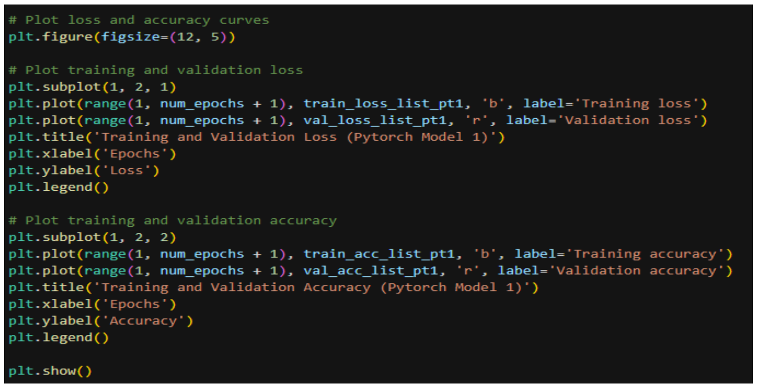
The result of resources monitoring is printed out with code in Diagram 2.2.8.



```
# Print training duration and resource usage
print("Training Duration:", duration_pt1, "seconds")
print("CPU Usage:",cpu_usage_pt1, "%")
print("RAM Usage:",ram_usage_pt1,"%")
```

Diagram 2.2.8 : Print Resource and Time Usage

After the model is trained and validated, the code in Diagram 2.2.9 visualizes the training and validation loss over epochs, as well as the training and validation accuracy over epochs, for each Pytorch model. This visualization is achieved using the loss and accuracy lists collected during the training and validation process for evaluation purposes.

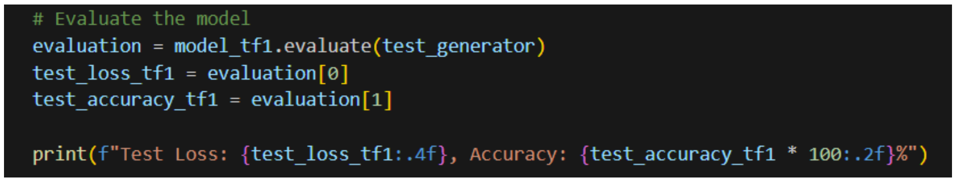


```
# Plot loss and accuracy curves
plt.figure(figsize=(12, 5))

# Plot training and validation loss
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_loss_list_pt1, 'b', label='Training loss')
plt.plot(range(1, num_epochs + 1), val_loss_list_pt1, 'r', label='Validation loss')
plt.title('Training and Validation Loss (Pytorch Model 1)')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

# Plot training and validation accuracy
plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs + 1), train_acc_list_pt1, 'b', label='Training accuracy')
plt.plot(range(1, num_epochs + 1), val_acc_list_pt1, 'r', label='Validation accuracy')
plt.title('Training and Validation Accuracy (Pytorch Model 1)')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.show()
```

Diagram 2.2.9 : Visualization of training and validation loss over epochs

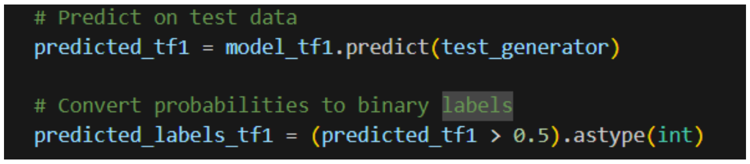
2.3 Model Evaluation

For **TensorFlow** models, we execute the code below to evaluate the trained model on the test data. It computes and prints out the test loss and accuracy metrics which are crucial for assessing how well the model generalized to unseen data. Next, we predict labels for the test data using the trained model.



```
# Evaluate the model
evaluation = model_tf1.evaluate(test_generator)
test_loss_tf1 = evaluation[0]
test_accuracy_tf1 = evaluation[1]

print(f"Test Loss: {test_loss_tf1:.4f}, Accuracy: {test_accuracy_tf1 * 100:.2f}%")
```

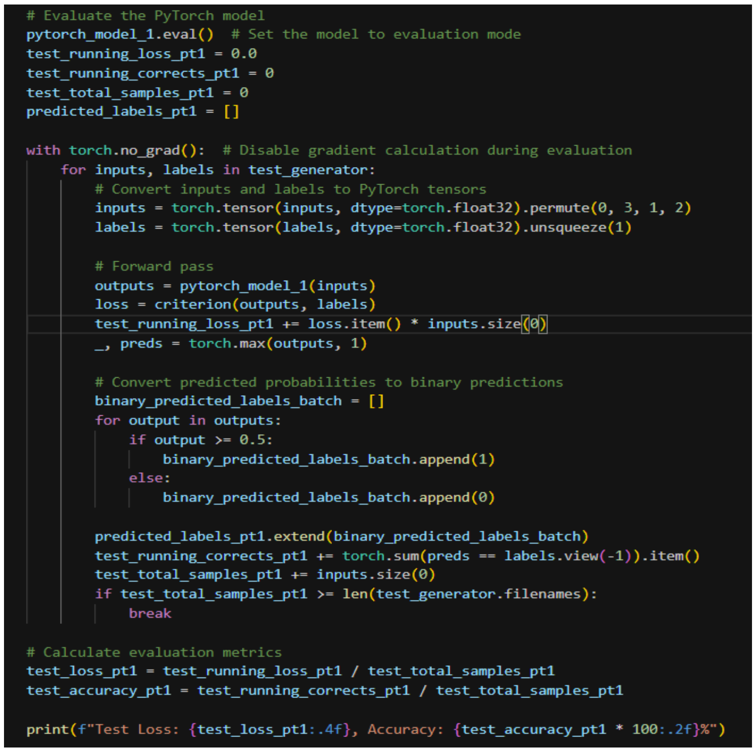
Diagram 2.3.1 : Model Evaluation of TensorFlow model



```
# Predict on test data
predicted_tf1 = model_tf1.predict(test_generator)

# Convert probabilities to binary labels
predicted_labels_tf1 = (predicted_tf1 > 0.5).astype(int)
```

Diagram 2.3.2 : Label prediction using TensorFlow Model

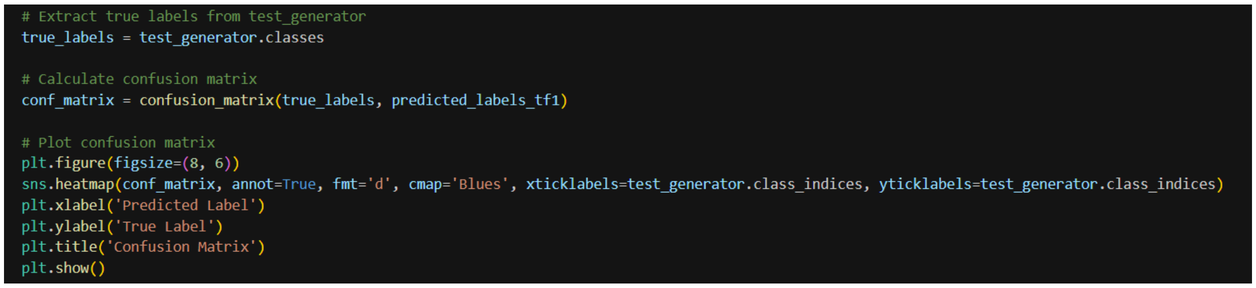
For **PyTorch** models, we set the model to evaluation mode and initialize evaluation variables to keep track of the running loss, correct predictions, and total samples during evaluation. In the evaluation loop, the code converts inputs and labels to PyTorch tensors, performs a forward pass through the model to obtain predictions, calculates the loss between the predictions and the true labels, updates counters for running loss and correct predictions, converts predicted probabilities to binary predictions based on a threshold (0.5), and extends the list of predicted labels for each batch. Finally, the code calculates evaluation metrics such as test loss and accuracy using the aggregated running loss and correct prediction.



```python
# Evaluate the PyTorch model
pytorch_model_1.eval()  # Set the model to evaluation mode
test_running_loss_pt1 = 0.0
test_running_corrects_pt1 = 0
test_total_samples_pt1 = 0
predicted_labels_pt1 = []

with torch.no_grad():  # Disable gradient calculation during evaluation
    for inputs, labels in test_generator:
        # Convert inputs and labels to PyTorch tensors
        inputs = torch.tensor(inputs, dtype=torch.float32).permute(0, 3, 1, 2)
        labels = torch.tensor(labels, dtype=torch.float32).unsqueeze(1)

        # Forward pass
        outputs = pytorch_model_1(inputs)
        loss = criterion(outputs, labels)
        test_running_loss_pt1 += loss.item() * inputs.size(0)
        _, preds = torch.max(outputs, 1)

        # Convert predicted probabilities to binary predictions
        binary_predicted_labels_batch = []
        for output in outputs:
            if output >= 0.5:
                binary_predicted_labels_batch.append(1)
            else:
                binary_predicted_labels_batch.append(0)

        predicted_labels_pt1.extend(binary_predicted_labels_batch)
        test_running_corrects_pt1 += torch.sum(preds == labels.view(-1)).item()
        test_total_samples_pt1 += inputs.size(0)
        if test_total_samples_pt1 >= len(test_generator.filenames):
            break

# Calculate evaluation metrics
test_loss_pt1 = test_running_loss_pt1 / test_total_samples_pt1
test_accuracy_pt1 = test_running_corrects_pt1 / test_total_samples_pt1

print(f"Test Loss: {test_loss_pt1:.4f}, Accuracy: {test_accuracy_pt1 * 100:.2f}%")
```

Diagram 2.3.3 : Model Evaluation of Pytorch Model

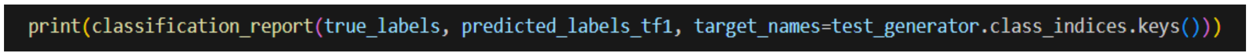
After the evaluation metrics are calculated, we visualize the confusion matrix for evaluating all the TensorFlow and Pytorch models' performance on the test dataset. A heatmap is used to visualize the matrix with annotations, where each cell represents the count of true positives, false positives, true negatives, and false negatives.



```python
# Extract true labels from test_generator
true_labels = test_generator.classes

# Calculate confusion matrix
conf_matrix = confusion_matrix(true_labels, predicted_labels_tf1)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=test_generator.class_indices, yticklabels=test_generator.class_indices)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

Diagram 2.3.4 : Confusion Matrix generation

Finally, a complete classification report is printed out for each model



```python
print(classification_report(true_labels, predicted_labels_tf1, target_names=test_generator.class_indices.keys()))
```

Diagram 2.3.3 : Report generation for the model

2.4 Result

2.4.1 Epoch Graph Visualisation

Model Architecture 1

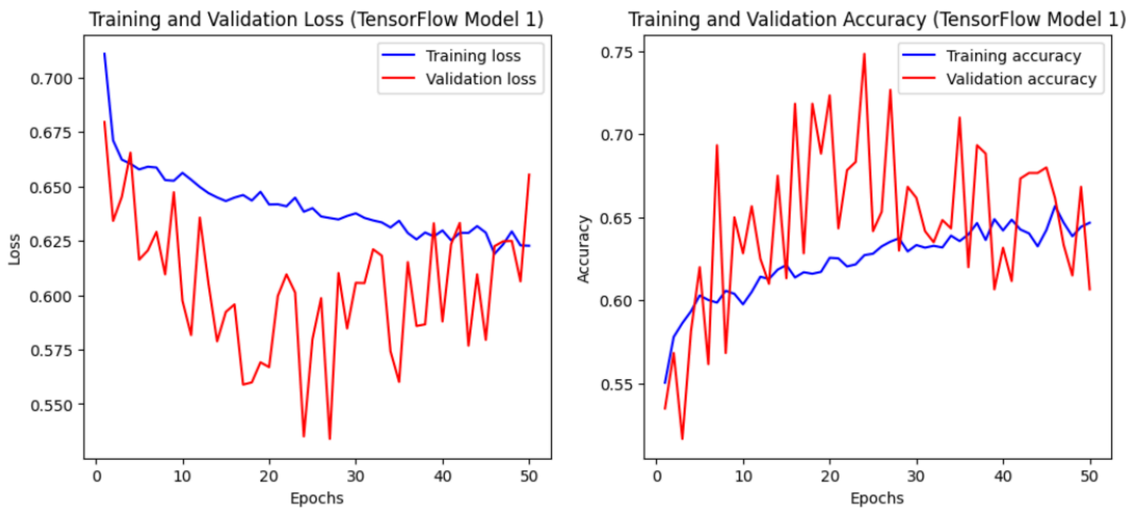


Diagram 2.4.1 : Graph of training and validation loss and accuracy for TensorFlow Model 1

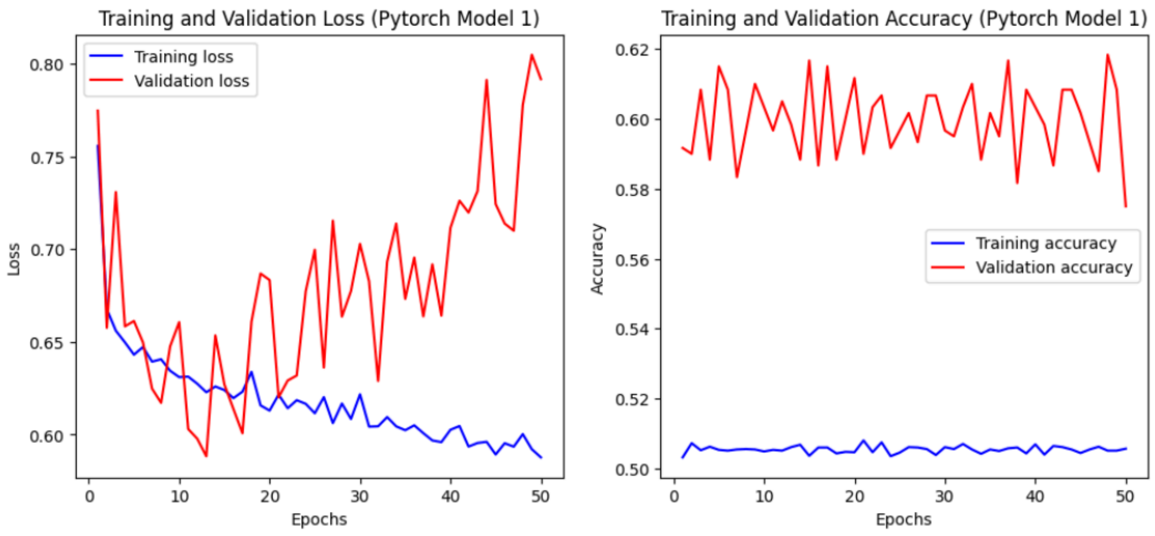
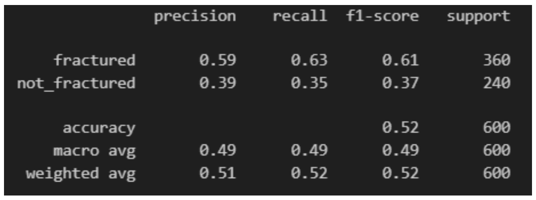
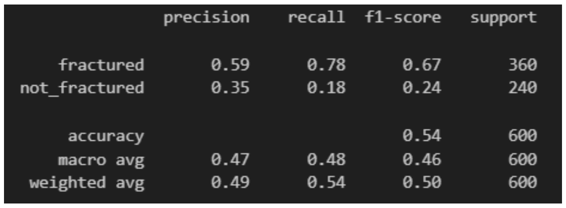


Diagram 2.4.2 : Graph of training and validation loss and accuracy for PyTorch Model 1



```
              precision    recall  f1-score   support

   fractured       0.59      0.63      0.61       360
not_fractured      0.39      0.35      0.37       240

    accuracy                           0.52       600
   macro avg       0.49      0.49      0.49       600
weighted avg       0.51      0.52      0.52       600
```

Diagram 2.4.3 : Classification report of



```
              precision    recall  f1-score   support

   fractured       0.59      0.78      0.67       360
not_fractured      0.35      0.18      0.24       240

    accuracy                           0.54       600
   macro avg       0.47      0.48      0.46       600
weighted avg       0.49      0.54      0.50       600
```

Diagram 2.4.4: Classification report of

TensorFlow Model 1 ***PyTorch Model 1***

Model Architecture 2

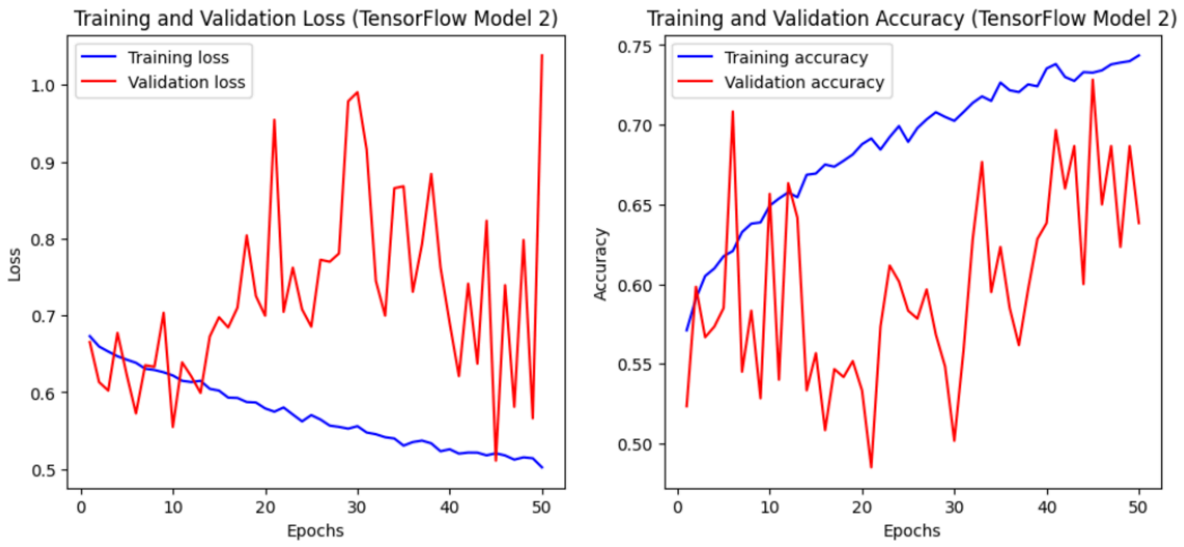


Diagram 2.4.5 : Graph of training and validation loss and accuracy for TensorFlow Model 2

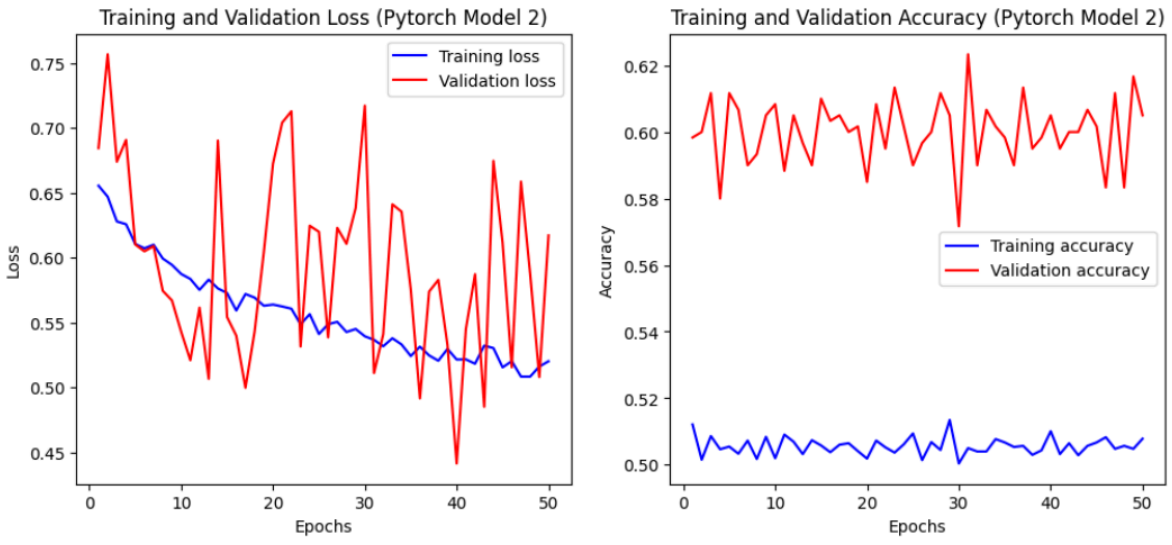
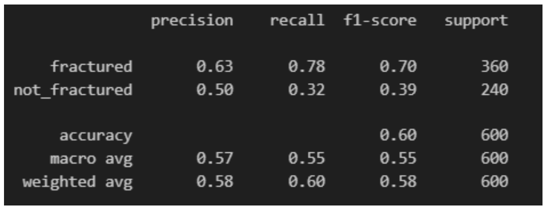
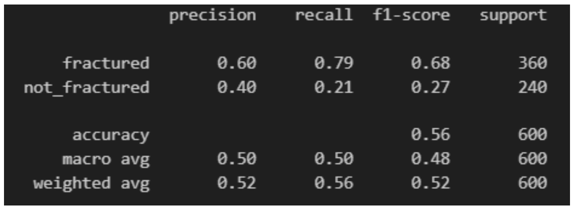


Diagram 2.4.6 : Graph of training and validation loss and accuracy for PyTorch Model 2



```
              precision    recall  f1-score   support

   fractured       0.63      0.78      0.70       360
not_fractured      0.50      0.32      0.39       240

    accuracy                           0.60       600
   macro avg       0.57      0.55      0.55       600
weighted avg       0.58      0.60      0.58       600
```

Diagram 2.4.7 : Classification report of TensorFlow Model 2



```
              precision    recall  f1-score   support

   fractured       0.60      0.79      0.68       360
not_fractured      0.40      0.21      0.27       240

    accuracy                           0.56       600
   macro avg       0.50      0.50      0.48       600
weighted avg       0.52      0.56      0.52       600
```

Diagram 2.4.8: Classification report of PyTorch Model 2

Model Architecture 3

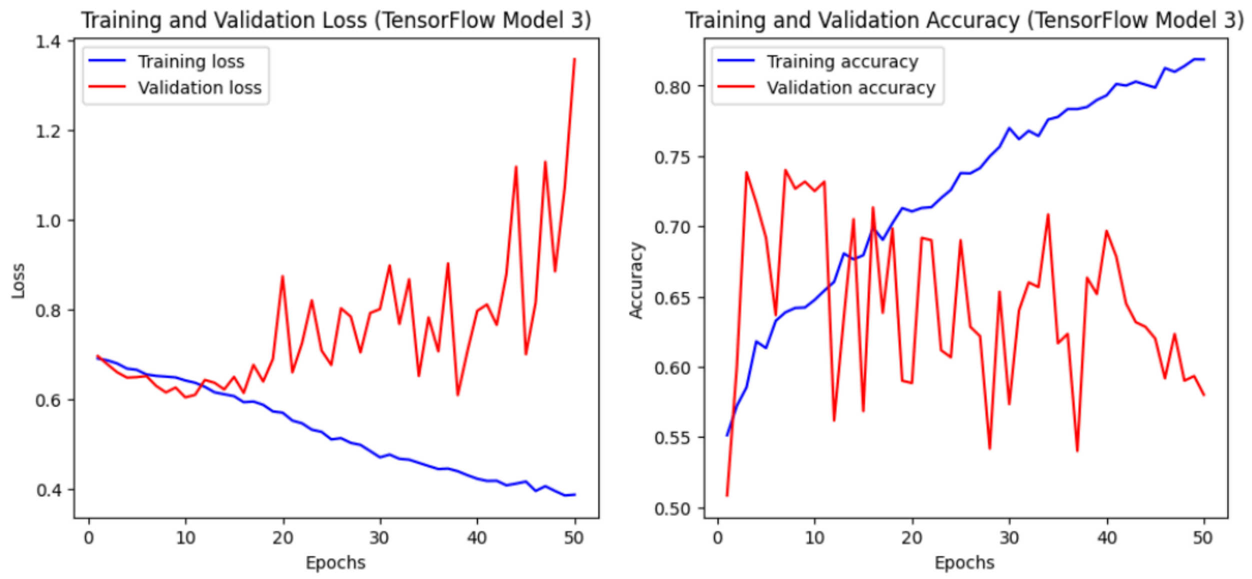


Diagram 2.4.9 : Graph of training and validation loss and accuracy for TensorFlow Model 3

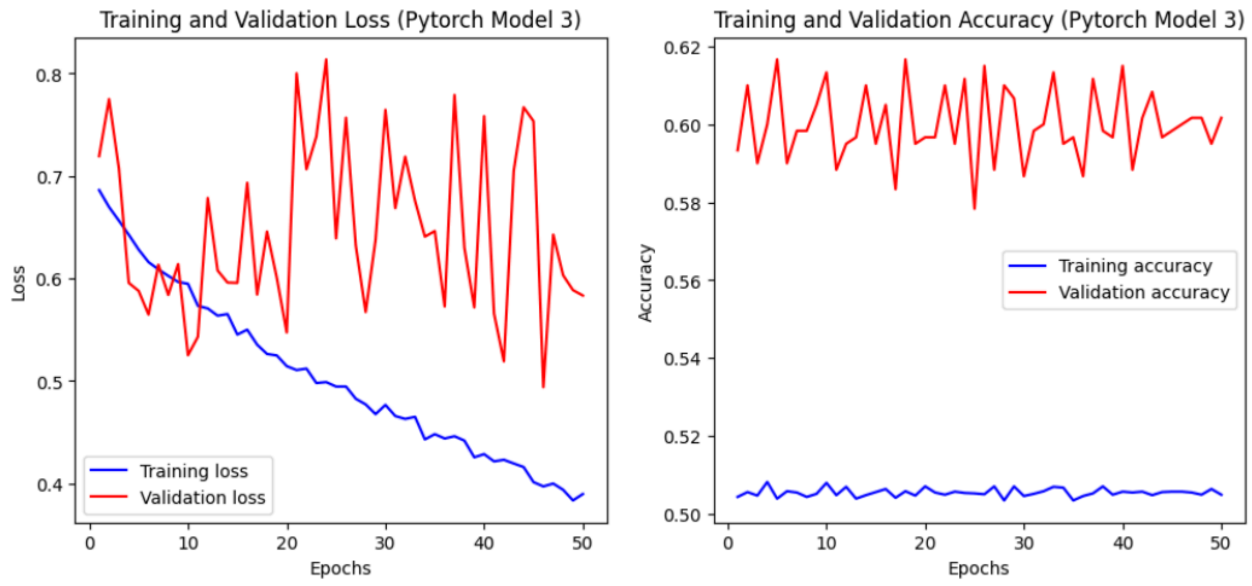
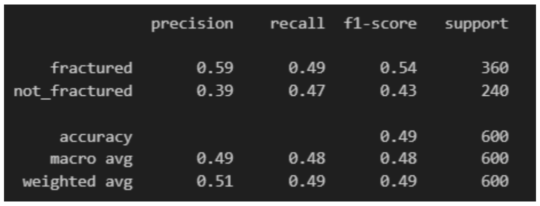
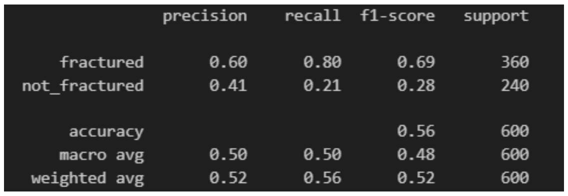


Diagram 2.4.10 : Graph of training and validation loss and accuracy for PyTorch Model 3



```
              precision    recall  f1-score   support

   fractured       0.59      0.49      0.54       360
not_fractured      0.39      0.47      0.43       240

    accuracy                           0.49       600
   macro avg       0.49      0.48      0.48       600
weighted avg       0.51      0.49      0.49       600
```

Diagram 2.4.11 : Classification report of TensorFlow Model 3



```
              precision    recall  f1-score   support

   fractured       0.60      0.80      0.69       360
not_fractured      0.41      0.21      0.28       240

    accuracy                           0.56       600
   macro avg       0.50      0.50      0.48       600
weighted avg       0.52      0.56      0.52       600
```

Diagram 2.4.12: Classification report of PyTorch Model 3

Model Architecture 4

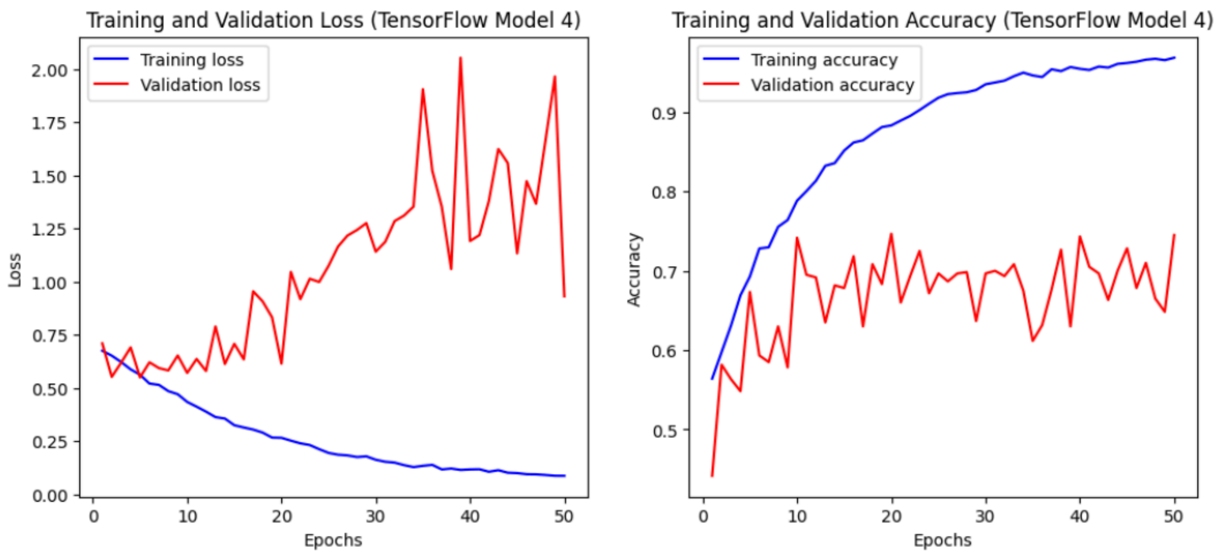


Diagram 2.4.13 : Graph of training and validation loss and accuracy for TensorFlow Model 4

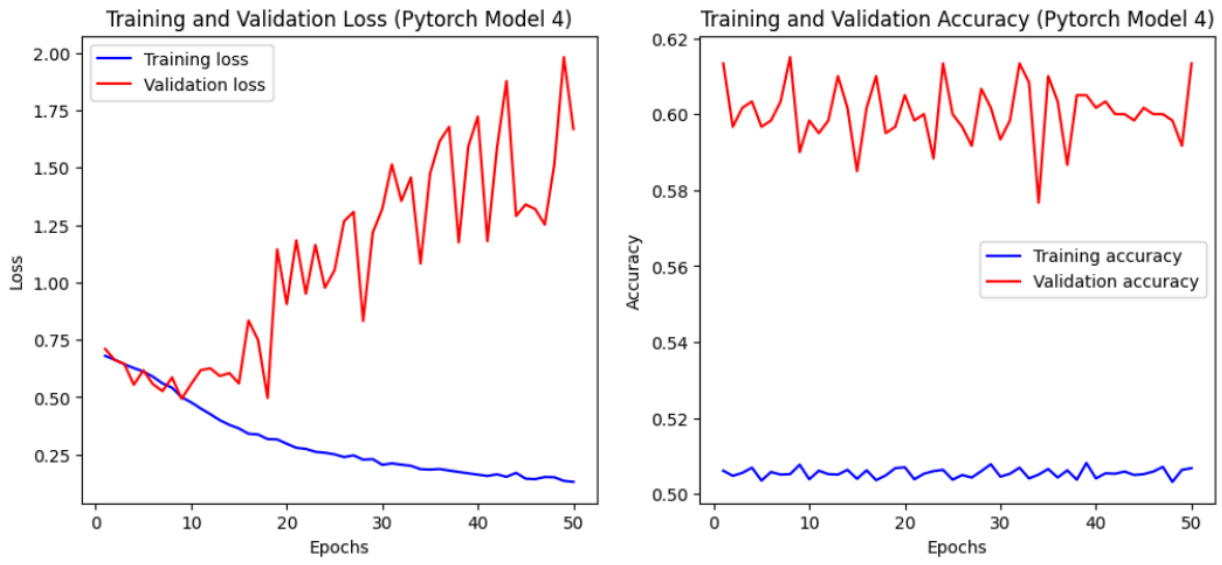
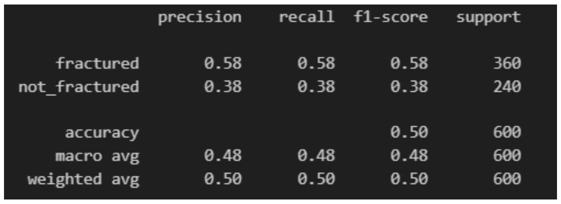
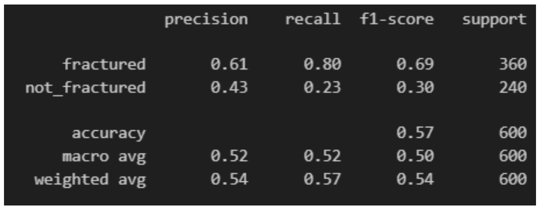


Diagram 2.4.14 : Graph of training and validation loss and accuracy for PyTorch Model 4



```
              precision    recall  f1-score   support

   fractured       0.58      0.58      0.58       360
not_fractured       0.38      0.38      0.38       240

    accuracy                           0.50       600
   macro avg       0.48      0.48      0.48       600
weighted avg       0.50      0.50      0.50       600
```

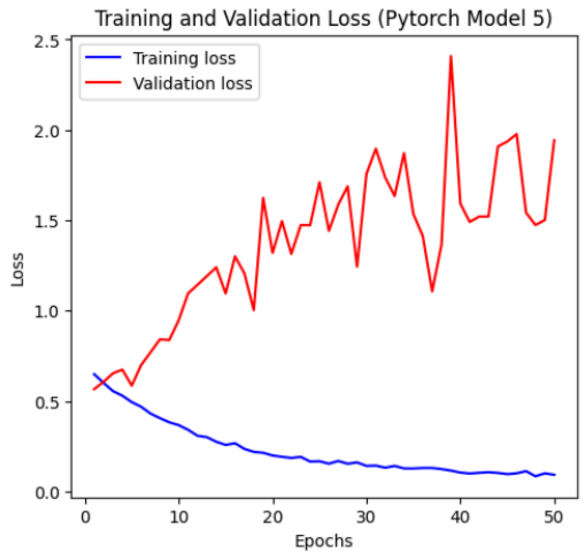
Diagram 2.4.15 : Classification report of TensorFlow Model 4



```
              precision    recall  f1-score   support

   fractured       0.61      0.80      0.69       360
not_fractured       0.43      0.23      0.30       240

    accuracy                           0.57       600
   macro avg       0.52      0.52      0.50       600
weighted avg       0.54      0.57      0.54       600
```

Diagram 2.4.16: Classification report of PyTorch Model 4

Model Architecture 5



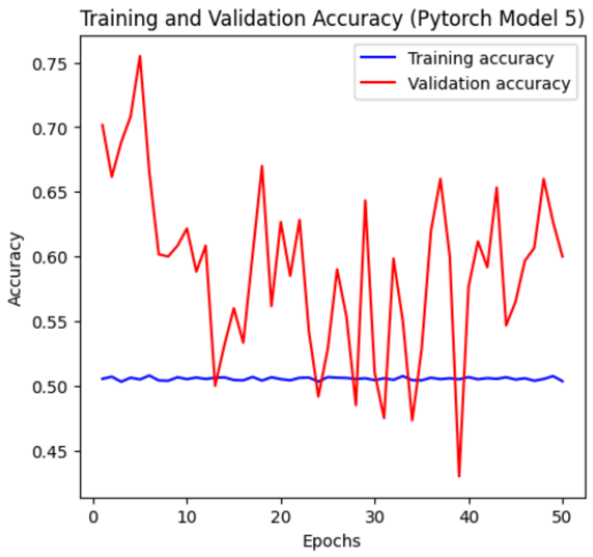
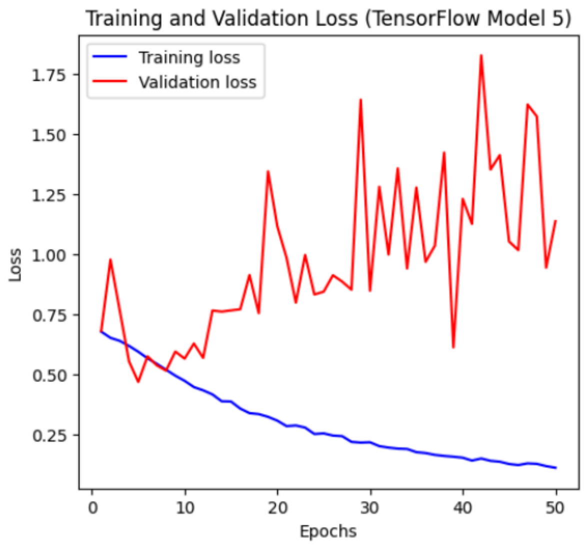


Diagram 2.4.17 : Graph of training and validation loss and accuracy for TensorFlow Model 5



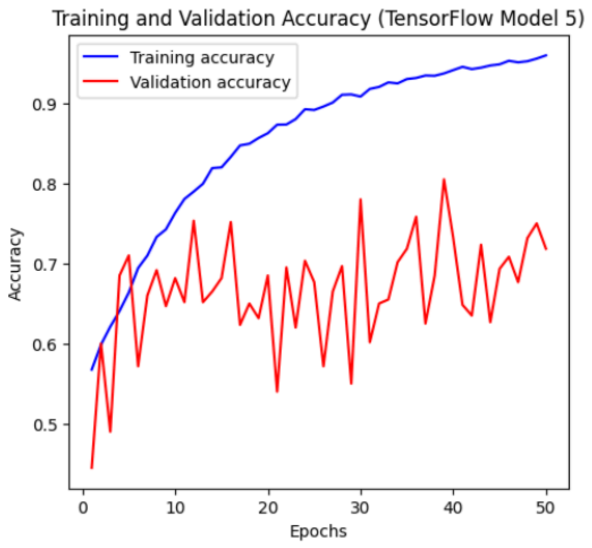
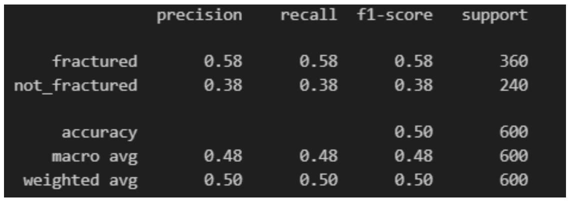
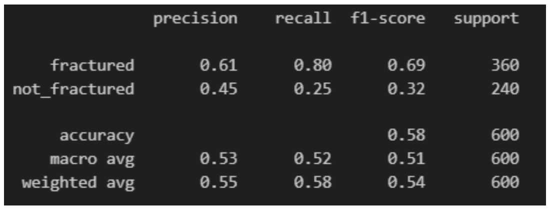


Diagram 2.4.18 : Graph of training and validation loss and accuracy for PyTorch Model 5



```
              precision    recall  f1-score   support

   fractured       0.58      0.58      0.58       360
not_fractured       0.38      0.38      0.38       240

    accuracy                           0.50       600
   macro avg       0.48      0.48      0.48       600
weighted avg       0.50      0.50      0.50       600
```

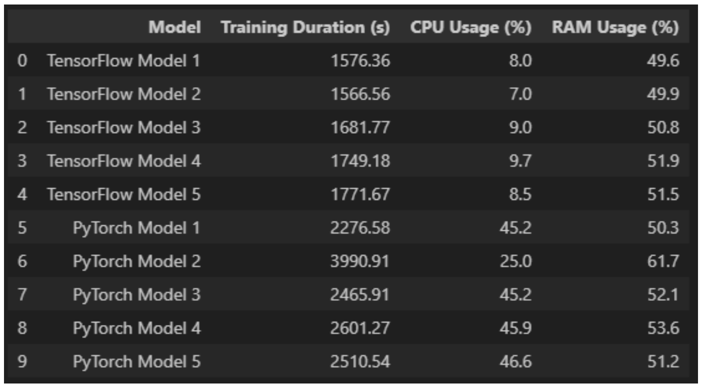
Diagram 2.4.19 : Classification report of TensorFlow Model 5



```
              precision    recall  f1-score   support

   fractured       0.61      0.80      0.69       360
not_fractured       0.45      0.25      0.32       240

    accuracy                           0.58       600
   macro avg       0.53      0.52      0.51       600
weighted avg       0.55      0.58      0.54       600
```

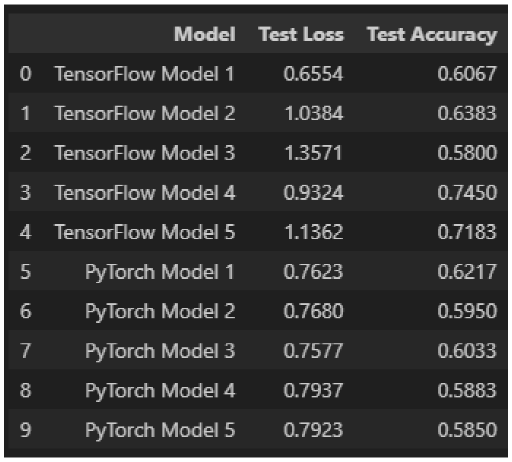
Diagram 2.4.20: Classification report of PyTorch Model 5

Summary



	Model	Training Duration (s)	CPU Usage (%)	RAM Usage (%)
0	TensorFlow Model 1	1576.36	8.0	49.6
1	TensorFlow Model 2	1566.56	7.0	49.9
2	TensorFlow Model 3	1681.77	9.0	50.8
3	TensorFlow Model 4	1749.18	9.7	51.9
4	TensorFlow Model 5	1771.67	8.5	51.5
5	PyTorch Model 1	2276.58	45.2	50.3
6	PyTorch Model 2	3990.91	25.0	61.7
7	PyTorch Model 3	2465.91	45.2	52.1
8	PyTorch Model 4	2601.27	45.9	53.6
9	PyTorch Model 5	2510.54	46.6	51.2

Diagram 2.4.1: Training Duration, CPU Usage, and RAM Usage of each model.



	Model	Test Loss	Test Accuracy
0	TensorFlow Model 1	0.6554	0.6067
1	TensorFlow Model 2	1.0384	0.6383
2	TensorFlow Model 3	1.3571	0.5800
3	TensorFlow Model 4	0.9324	0.7450
4	TensorFlow Model 5	1.1362	0.7183
5	PyTorch Model 1	0.7623	0.6217
6	PyTorch Model 2	0.7680	0.5950
7	PyTorch Model 3	0.7577	0.6033
8	PyTorch Model 4	0.7937	0.5883
9	PyTorch Model 5	0.7923	0.5850

Diagram 2.4.2: Test Loss and Test Accuracy of Each Model

2.4.2 Comparision of 5 Architecture

Model		**Accuracy**	**Test Loss**	**Test Accuracy**	**Training Duration**	**CPU Usage**	**RAM Usage**
1	TensorFlow	0.54	0.6554	0.6067	1576.36	8.0	49.6
	PyTorch	0.52	0.7623	0.7217	2276.58	45.2	50.3
2	TensorFlow	0.60	1.0384	0.6383	1566.56	7.0	49.9
	PyTorch	0.56	0.7680	0.5950	3990.91	25.0	61.7
3	TensorFlow	0.49	1.3571	0.5800	1681.77	9.0	50.8
	PyTorch	0.56	0.7577	0.6033	2467.91	45.2	52.1
4	TensorFlow	0.50	0.9324	0.7450	1749.18	9.7	51.9

	PyTorch	0.57	0.7937	0.5883	2601.27	45.9	53.6
5	TensorFlow	0.50	1.1362	0.7183	1771.67	8.5	51.5
	PyTorch	0.58	0.7923	0.5850	2510.54	46.6	51.2

Table 2.4.1: Summary Table of All Models Based on Evaluation Metrics

Based on the summary table of each model's evaluation metrics, some insights are extracted as shown below:

Model	Layers	Performance
1	Convolutional Layer: 1 Max-Pooling Layer: 1 Fully Connected Layers: 1	Pros: ● Simple architecture with fewer layers, leading to faster training and inference. ● Lower computational complexity, making it suitable for resource-constrained environments. Cons: ● Limited capacity to learn complex patterns and features due to the shallow network architecture. ● Struggle to achieve high accuracy on more challenging datasets.
2	Convolutional Layers: 1 Max-Pooling Layers: 1 Fully Connected Layers: 2	Pros: ● Adds depth to the network with an additional fully connected layer, allowing for better feature extraction and representation. ● Increased capacity to learn complex patterns compared to Architecture 1. Cons: ● Still relatively simple compared to deeper architectures, potentially limiting its ability to capture intricate features.
3	Convolutional Layers: 2 Max-Pooling Layers: 2 Fully Connected Layers: 2	Pros: ● Further increases the depth of the network with additional convolutional and max-pooling layers, enhancing feature learning capabilities.

		● Balanced architecture with moderate complexity, suitable for a wide range of tasks. Cons: ● Increased computational complexity compared to shallower architectures, leading to longer training times and higher resource usage.
4	Convolutional Layers: 2 Max-Pooling Layers: 2 Fully Connected Layers: 3	Pros: ● Introduces an additional fully connected layer, allowing for more complex feature combinations and abstraction. ● Achieves the highest test accuracy among all architectures, indicating superior performance. Cons: ● Higher computational complexity and resource requirements compared to shallower architectures. ● Longer training times may be a limitation in time-sensitive applications.
5	Convolutional Layers: 3 Max-Pooling Layers: 3 Dropout Layers: 1 Batch Normalization Layers: 1 Fully Connected Layers: 3	Pros: ● Adds even more depth to the network with additional convolutional and max-pooling layers, along with dropout and batch normalization layers for regularization and stability. ● Offers the most comprehensive feature learning capabilities among all architectures. Cons: ● Highest computational complexity and resource usage among all architectures, potentially requiring significant computational resources for training and inference. ● Longer training times compared to shallower architectures. ● Might result in overfitting

Overall, TensorFlow models use a shorter time for processing and consume less resources (CPU, RAM) than PyTorch models. Among the TensorFlow models, Model

Architecture 4 has the highest test accuracy, relatively low test loss and moderate accuracy. Hence, the TensorFlow model 4 is the best-performed model.

3. Discussions and Findings

3.1 Pros and Cons of Model Complexity in relation to Model Performance

Model Complexity	**Simple Model Architecture**	**Complex Model Architecture**
Pros	● **Require fewer computational resources and less training time** compared to complex models as shown in result, when complexity increases, resources which can be seen from the increase in training duration across each TensorFlow model. ● Simple models are **less prone to overfitting** since they have fewer parameters and are less likely to learn noise or irrelevant patterns from the training data.	● Complex models have a **larger number of parameters**, allowing them to capture intricate patterns and relationships in the data more effectively which will increase the leading to **higher model accuracy and better performance** on complex tasks. This is demonstrated from the slightly increasing test accuracy from TensorFlow Model 1 to Model 4.
Cons	● Have limited capacity to capture complex relationships and patterns in the data, which may **lead to lower performance** on tasks that require advanced feature representations.	● Complex models require **more computational resources** including running time and resources for training which makes them slower and more resource consuming. ● Complex models have a bigger chance to face **overfitting** compared to simple models , where they are too used to the training data which leads to poor generalization on unseen data. This can be seen from the drop of Test Accuracy from Model 4 to Model 5.

3.2 Model Complexity vs Model Performance

Initially, as the complexity of the models increased from Model 1 to Model 4, there was a corresponding improvement in performance metrics such as training accuracy and validation accuracy. This improvement can be attributed to the ability of more complex models to capture intricate patterns and relationships in the data, leading to better representation learning.

However, as we progressed to Model 5, which represented the peak of complexity among our models, we noticed a decline in performance metrics despite the increased complexity. This phenomenon indicates that the model became overly specialized to the training data, exhibiting signs of overfitting. Overfitting occurs when the model learns to capture noise or irrelevant patterns present in the training data, which do not generalize well to unseen data.

Therefore, we discovered the importance of striking a balance between model complexity and performance. While increasing model complexity can lead to improved performance up to a certain point, it's crucial to monitor for signs of overfitting. Finding the optimal level of complexity involves careful experimentation and validation, ensuring that the model generalizes well to unseen data while effectively capturing the underlying patterns in the dataset.

4. Conclusion

In summary, we designed five different architectures using two different libraries, TensorFlow and PyTorch, which made up to ten models in total for the classification of bone fractures using X-ray images. Based on the results, the best-performing model identified is TensorFlow Model 4, which achieved the highest test accuracy of 0.7450 among all the models. This indicates that it can classify fractured bones more effectively compared to other models when presented with unseen images. Additionally, it is noteworthy that the training duration (seconds) and computational resources (RAM and CPU) used by all PyTorch models are higher than those of the TensorFlow models.

Besides determining the best model for the project, we also identified the pros and cons of model complexity in relation to model performance. While complex models with larger parameters may show better performance on complex tasks, we also need to consider factors such as training time and the computational resources required to train a complex model. Additionally, complex models may generalize too well on training data, leading to overfitting. Therefore, it is important to find a balance between model complexity and model performance when developing a model.

One limitation of this project is the potential impact on system resources, particularly RAM and CPU usage when performing other tasks concurrently. Since the models utilized significant computational resources during training and inference, running them alongside other resource-intensive applications or processes may lead to performance degradation or system slowdown. Besides, One notable limitation is that the actual performance of the models on the dataset may differ from our expectations. Despite carefully designing and training the models with specific architectures and parameters, the real-world performance may deviate due to various factors inherent to the dataset itself. These factors could include data quality issues, label inaccuracies, class imbalances, or inherent complexities within the dataset that the models may struggle to capture effectively.

For future improvements, one of the mitigation methods that can be done is by implementing model optimization techniques such as model pruning, quantization, or compression to reduce the model's size and computational requirements without compromising performance. Additionally, exploring transfer learning approaches could enhance overall model performance with limited computational resources, leveraging pre-trained models and fine-tuning them on smaller, domain-specific datasets. These methods would also address the discrepancy between expected and actual performance by ensuring that the models are appropriately sized and efficient, leading to more accurate predictions.

5. References

JALIL, O. (2023). *Bone Fracture Dataset*. https://www.kaggle.com/datasets/osamajalilhassan/bone-fracture-dataset

Sharma, D. (2021, January 11). *Image Classification Using CNN | Step-wise Tutorial*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/01/image-classification-using-convolutional-neural-networks-a-step-by-step-guide/#:~:text=Image%20classification%20using%20CNN%20involves

Symbl, T. (2022, September 16). *Building the Same Neural Network in TensorFlow and PyTorch*. Symbl.ai. https://symbl.ai/developers/blog/building-the-same-neural-network-in-tensorflow-and-pytorch/

Thian, Y. L., Li, Y., Jagmohan, P., Sia, D., Chan, V. E. Y., & Tan, R. T. (2019). Convolutional neural networks for automated fracture detection and localization on wrist radiographs. Radiology: Artificial Intelligence, 1(1). https://doi.org/10.1148/ryai.2019180001