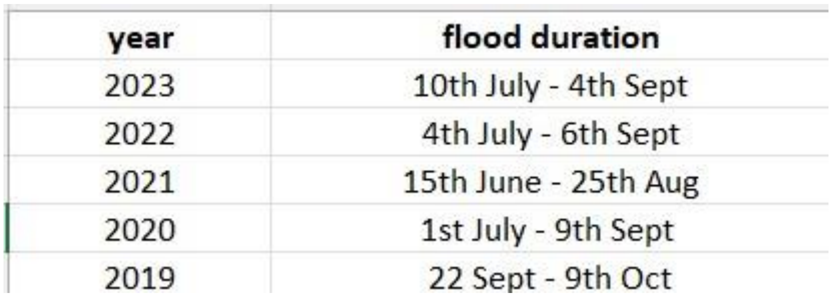
SATELLITE IMAGES METHODOLOGY

DATASET CREATION:

FLOOD DURATIONS SAMPLE:



year	flood duration
2023	10th July - 4th Sept
2022	4th July - 6th Sept
2021	15th June - 25th Aug
2020	1st July - 9th Sept
2019	22 Sept - 9th Oct

Collected from Government sources.

Used Google Earth Engine

Sentinel-1 SAR satellite images used (available from 2014 – present)

PSEUDOCODE FOR FLOOD SATELLITE IMAGES:

```
// Define the geographical bounds for the area of interest (AOI)
var westlimit = 86.195866;
var southlimit = 25.47778;
var eastlimit = 86.703434;
var northlimit = 26.279414;

// Create a rectangle geometry using the bounding box coordinates
var aoi = createRectangle(westlimit, southlimit, eastlimit, northlimit);

// Add the AOI layer to the map
addLayerToMap(aoi, 'kosi area');
```

```
// Load the Sentinel-1 ImageCollection and filter it by AOI and polarization
var collection = loadImageCollection("COPERNICUS/S1_GRD")
  .filterByBounds(aoi)
  .filterByPolarization("VV")
  .selectBand('VV');

// Create a mosaic of images before the flood
var before = collection.filterByDate("2019-04-01", "2019-04-15").createMosaic();
var before_clip = clipImage(before, aoi);
var before_smoothed = applyFocalMedian(before_clip, 30, "circle", "meters");

// Define the flood period start and end dates
var startDate = createDate('2019-09-22');
var endDate = createDate('2019-10-09');

// Create a list of dates during the flood duration
var dateList = generateDateList(startDate, endDate);

// Iterate through each date in the list
for each date in dateList:
   var dateStr = formatDate(date, 'YYYY-MM-dd');
   var dailyCollection = collection.filterByDate(date, date.advance(1, 'day'));

   // Check if the daily collection is not empty
   var count = getImageCount(dailyCollection);
   if count > 0:
      // Create a mosaic of images during the flood
      var during = dailyCollection.createMosaic();
      var during_clip = clipImage(during, aoi);
      var during_smoothed = applyFocalMedian(during_clip, 30, "circle", "meters");
```

```
    // Calculate the difference between before and during images
    var difference = subtractImages(during_smoothed, before_smoothed);
    var flood_extent = applyThreshold(difference, -3);
    var flood = updateMask(flood_extent);

    // Add layers to the map
    addLayerToMap(during_clip, {min: -30, max: 0}, "During Flood " + dateStr);
    addLayerToMap(flood, {}, "Flood affected areas " + dateStr);

    // Export the flood extent image to Google Drive
    exportImageToDrive(flood, "Inception_v3_" + dateStr, 'Inception v3_Flood', 10, 1e13, aoi,
"EPSG:4326");
  else:
    // Log a message if no valid images are found for the date
    logMessage("No valid images for date: " + dateStr);
```

Summary Explanation

1. **Define the AOI**: The script starts by defining the geographical coordinates that form a rectangle representing the Kosi region in Bihar.
2. **Create and Display AOI**: It creates a rectangle geometry using these coordinates and adds this area of interest (AOI) to the map.
3. **Load and Filter Image Collection**: The Sentinel-1 ImageCollection is loaded and filtered to include only images that intersect with the AOI and have VV polarization.
4. **Create Pre-Flood Mosaic**: A mosaic of images from a specific pre-flood period (April 1 to April 15, 2019) is created, clipped to the AOI, and smoothed using a focal median filter.
5. **Define Flood Period**: The flood period is defined with start and end dates (September 22 to October 9, 2019).
6. **Generate Date List**: A list of dates within the flood period is generated.
7. **Process Each Date**: For each date in the flood period:
    - The script filters the image collection to get images for that specific date.

- If images are available, it creates a mosaic, clips it to the AOI, and smooths it.
- The difference between the pre-flood and during-flood images is calculated to identify flood extent.
- The flood-affected areas are masked and added to the map.
- The flood extent image is exported to Google Drive.
- If no images are found for a specific date, a message is logged.

PSEUDOCODE FOR NON-FLOOD SATELLITE IMAGES AND EXPLANATION :

```
// Define the area of interest (AOI)
var aoi = createRectangle(86.053271, 25.506503, 86.75198, 26.307403);

// Define the start and end dates
var startDate = createDate('2021-04-10');
var endDate = createDate('2021-04-25');

// Function to export and map image for each day
function exportAndMapDailyImage(date) {
  // Convert the date to an ee.Date object
  var dayStart = convertToDate(date);
  var dayEnd = advanceDate(dayStart, 1, 'day');

  // Print the date for debugging purposes
  print('Processing date:', dayStart);

  // Filter Sentinel-1 SAR data for the day
  var collection = loadImageCollection('COPERNICUS/S1_GRD')
    .filterByBounds(aoi)
    .filterByDate(dayStart, dayEnd)
    .filterByInstrumentMode('IW')
    .filterByPolarization('VV')
    .selectBand('VV');

  // Check if the collection is empty
  var count = getImageCount(collection);
  if count == 0:
    print('No image for date:', formatDate(dayStart, 'YYYY-MM-dd'));
    return;

  // Create a median composite for the day
  var median = createMedianComposite(collection).clip(aoi);

  // Apply a threshold to detect water
  var waterThreshold = -16;  // Adjust this threshold based on your region
  var water = applyThreshold(median, waterThreshold).selfMask();

  // Visualize the water mask in grayscale
  var dayString = formatDate(dayStart, 'YYYY-MM-dd');
  addLayerToMap(water, {min: 0, max: 1, palette: ['black', 'white']}, 'Water ' + dayString);

  // Export the result as a grayscale image
  exportImageToDrive(water.visualize({min: 0, max: 1, palette: ['black', 'white']}),
          'Inception v3_non_' + dayString,
          50,
          aoi,
          'Inception v3_Non',
          1e13);

  print('Exported and mapped image for date:', dayString);
}
```

```
// Generate a list of dates from start to end date
var days = generateDateList(startDate, endDate);


// Convert the days list to JavaScript dates and process each day
for each date in days:
  exportAndMapDailyImage(date);
```

Summary Explanation

1. **Define AOI**: The script defines the geographical area of interest (AOI) for the Kosi region in Bihar using specific coordinates.
2. **Set Date Range**: The start and end dates for the non-flood period are set (April 10 to April 25, 2021).
3. **Export and Map Function**: A function exportAndMapDailyImage is created to process images for each day:
    - Convert the date to an Earth Engine ee.Date object.
    - Print the date for debugging.
    - Filter the Sentinel-1 SAR data for the specified date, AOI, and polarization ('VV').
    - Check if any images are available for the day. If not, print a message and skip to the next date.
    - Create a median composite image for the day and clip it to the AOI.
    - Apply a threshold to detect water and create a binary water mask.
    - Visualize the water mask on the map.
    - Export the water mask image to Google Drive.
4. **Generate Date List**: A list of dates between the start and end dates is generated.
5. **Process Each Date**: The script loops through each date, calling the exportAndMapDailyImage function to process the images for each day.

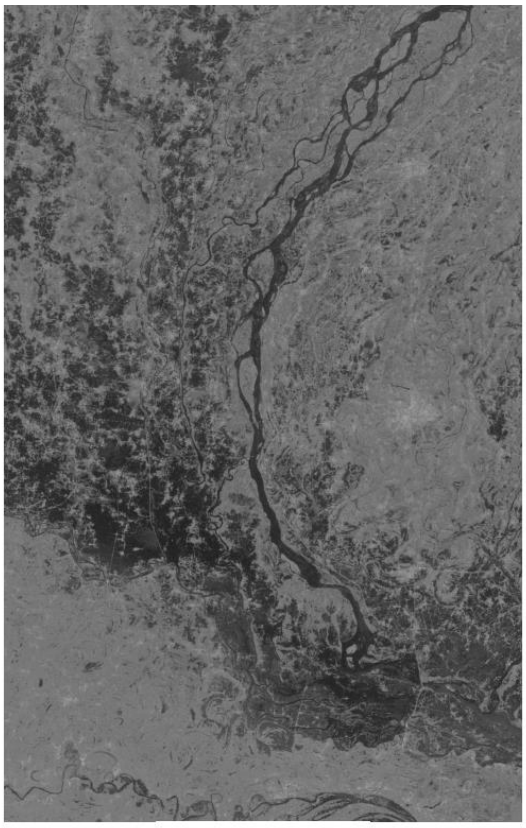
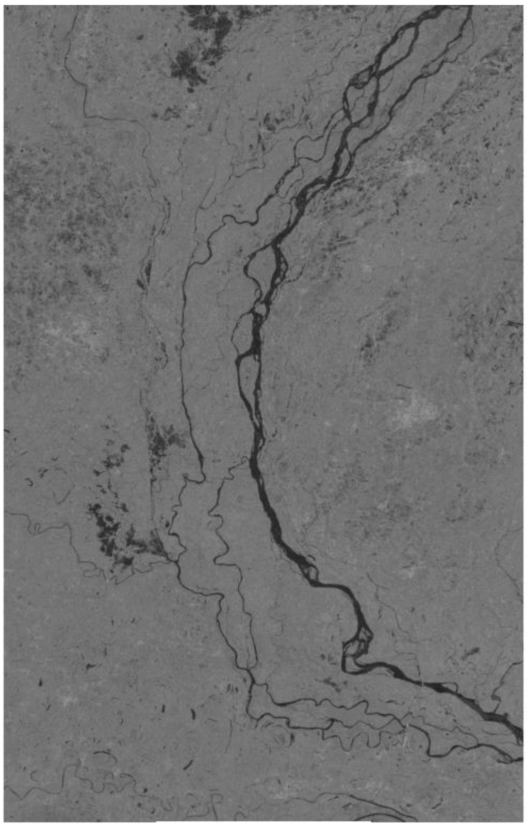
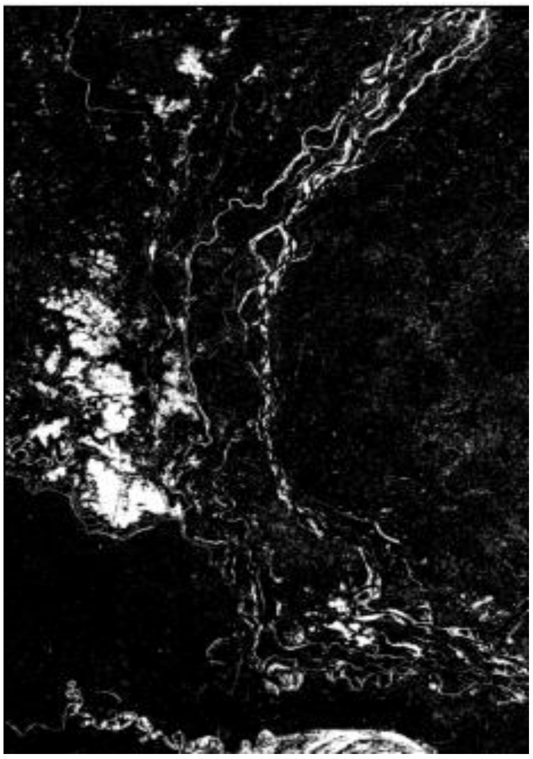
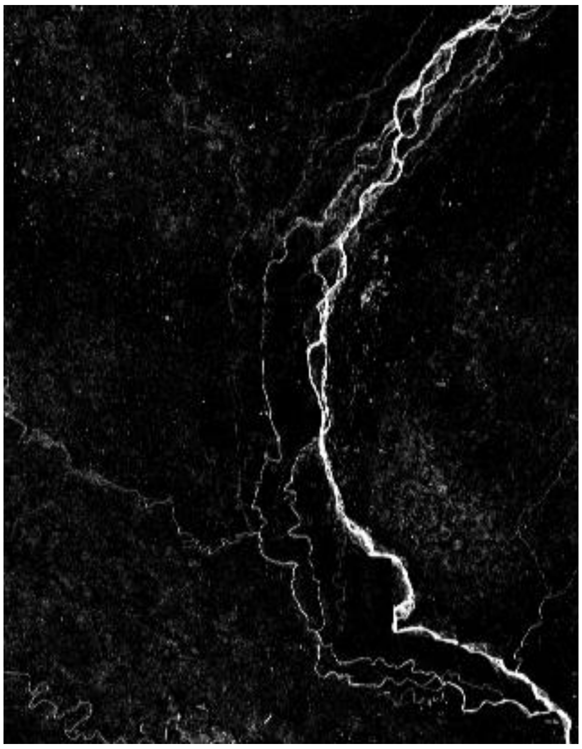
Contrasting Differences from Flood Script

1. **Purpose**:
    - **Flood Script**: Focuses on detecting and exporting flood-affected areas by comparing pre-flood and during-flood images.
    - **Non-Flood Script**: Focuses on detecting and exporting water bodies during a non-flood period using daily images.
2. **Date Range**:
    - **Flood Script**: Uses specific flood period dates (e.g., September 22 to October 9, 2019).
    - **Non-Flood Script**: Uses dates during a non-flood period (e.g., April 10 to April 25, 2021).
3. **Image Processing**:
    - **Flood Script**: Creates a mosaic of images before the flood and compares it with during-flood images to detect flood extents.
    - **Non-Flood Script**: Creates a median composite of daily images and applies a threshold to detect water bodies.
4. **Visualization and Export**:
    - **Flood Script**: Adds both the during-flood image and the detected flood-affected areas to the map.
    - **Non-Flood Script**: Adds only the water mask to the map.
5. **Threshold Application**:
    - **Flood Script**: Uses a difference threshold to detect flood-affected areas.
    - **Non-Flood Script**: Applies a direct threshold to detect water bodies.
6. **Image Export**:
    - **Flood Script**: Exports images of flood-affected areas.
    - **Non-Flood Script**: Exports water body masks during the non-flood period.

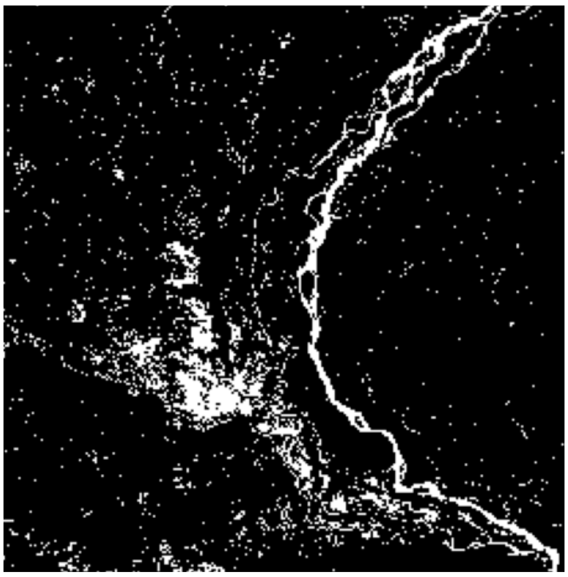
SAMPLE IMAGES:

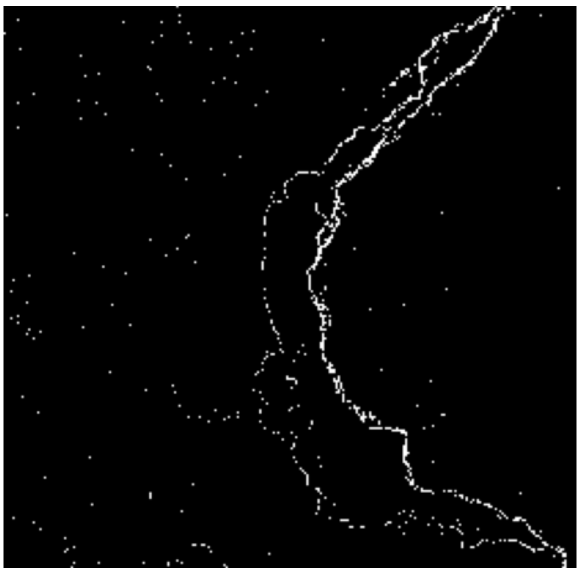
RAW SATELLITE IMAGES GRAYSCALE AND BINARY:

FLOOD **NON-FLOOD**

SAMPLE PRE-PROCESSED BINARY IMAGE MASKS:





After pre-processing the images will be used to give to different Models to classify and predict flood and non-flood maps

Inception v3 CLASSIFICATION:

PSEUDOCODE:

```
# Function to load and preprocess the images
def load_and_preprocess_image(filepath):
  try:
    image = Image.open(filepath).convert('L')  # Convert to grayscale
    image = image.resize((299, 299))  # Resize to 299x299 as required by InceptionV3
    image = np.array(image)
    image = np.stack((image,) * 3, axis=-1)  # Convert to 3 channels
    image = preprocess_input(image)
    return image
  except UnidentifiedImageError:
    print(f"Warning: Could not open {filepath}. Skipping.")
```

```
        return None  # Return None for skipped images


# Load and preprocess all images in a directory
def load_dataset(directory):
    images = []
    labels = []
    for filename in os.listdir(directory):
        if filename.endswith('.tif'):
            filepath = os.path.join(directory, filename)
            image = load_and_preprocess_image(filepath)
            if image is not None:  # Check if image was loaded successfully
                images.append(image)
                # Assuming the filename format is 'flood_001.tif' or 'nonflood_001.tif'
                label = 1 if 'flood' in filename.lower() else 0
                labels.append(label)
    images = np.array(images)
    labels = np.array(labels)
    labels = tf.keras.utils.to_categorical(labels, num_classes=2)
    return images, labels
```

This code contains two main functions: load_and_preprocess_image and load_dataset. These functions are designed to load, preprocess, and prepare a dataset of grayscale .tif images for training a machine learning model, specifically with the InceptionV3 architecture.

Function: load_and_preprocess_image

Purpose: Load and preprocess a single image.

1. **Load and Convert to Grayscale:**
    - Opens the image file and converts it to grayscale.
2. **Resize Image:**
    - Resizes the image to 299x299 pixels, which is required by InceptionV3.
3. **Convert to Array and Add Channels:**
    - Converts the image to a NumPy array and adds extra channels to make it a 3-channel image (from grayscale to RGB).

4. **Preprocess Input:**
    - Preprocesses the image according to InceptionV3 model requirements (scaling pixel values).
5. **Error Handling:**
    - If the image cannot be opened, it prints a warning and skips the image.

Function: load_dataset

Purpose: Load and preprocess all images in a specified directory, and assign labels based on filenames.

1. **Initialize Lists:**
    - Creates empty lists to store images and labels.
2. **Iterate Over Files:**
    - Iterates through all .tif files in the directory, processes each using load_and_preprocess_image, and assigns labels based on the filename (e.g., 'flood' or 'nonflood').
3. **Convert to NumPy Arrays:**
    - Converts the lists of images and labels to NumPy arrays.
4. **One-Hot Encode Labels:**
    - Converts labels to one-hot encoded format for classification tasks.
5. **Return Dataset:**
    - Returns the processed images and their labels for use in model training.

Key Differences from Flood Script

1. **Image Preparation:**
    - The current script focuses on preparing grayscale images for model training, while the flood script was about identifying and exporting flood-affected areas from satellite images.
2. **Resizing and Channel Conversion:**
    - The current script resizes images to 299x299 pixels and converts them to 3-channel format for InceptionV3, while the flood script did not resize images or change their channel format.
3. **Label Assignment:**
    - The current script assigns labels to images based on filenames for supervised learning, whereas the flood script did not handle labeling but focused on exporting processed images.

MODEL ARCHITECTURE:

```
# Load the Inception V3 model
base_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(299, 299, 3))


# Add custom layers on top of the base model
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(2, activation='softmax')(x)


# Define the model
model = Model(inputs=base_model.input, outputs=predictions)


# Freeze the base model layers
for layer in base_model.layers:
  layer.trainable = False


# Compile the model
model.compile(optimizer=Adam(learning_rate=0.0001),
        loss='categorical_crossentropy',
        metrics=['accuracy'])
```

This code demonstrates how to create a custom image classification model using the pre-trained Inception V3 model from Keras, with additional custom layers added on top.

Key Steps and Components

1. **Load Pre-trained Inception V3 Model:**
   - The base Inception V3 model is loaded with pre-trained weights from ImageNet.
   - The top fully connected layers of the original model are excluded (include_top=False).
   - The input shape is set to 299x299 pixels with 3 channels (RGB).
2. **Add Custom Layers:**

   - **Global Average Pooling:** Averages the spatial dimensions of the feature maps to a single vector.
   - **Fully Connected Layer (Dense Layer):** Adds a dense layer with 1024 units and ReLU activation.
   - **Output Layer:** Adds a final dense layer with 2 units (for binary classification) and softmax activation to output class probabilities.

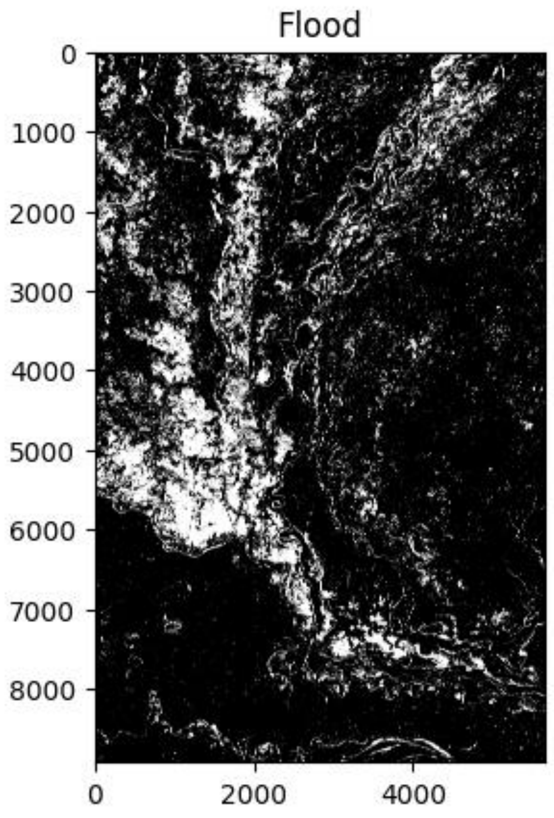
3. **Combine Base and Custom Layers:**
   - The custom layers are appended to the output of the base Inception V3 model.
   - A new model is defined with the combined architecture.
4. **Freeze Base Model Layers:**
   - The layers of the base Inception V3 model are set to non-trainable to preserve the learned features and prevent them from being updated during training.
5. **Compile the Model:**
   - The model is compiled with the Adam optimizer and a learning rate of 0.0001.
   - Categorical cross-entropy is used as the loss function (suitable for multi-class classification).
   - Accuracy is used as the evaluation metric.

Summary

This code builds a custom image classification model leveraging the pre-trained Inception V3 architecture for feature extraction. Custom fully connected layers are added on top for binary classification. The base model's weights are frozen to retain pre-trained features, while the new layers are trained for the specific classification task. The model is compiled with appropriate settings and is ready for training.

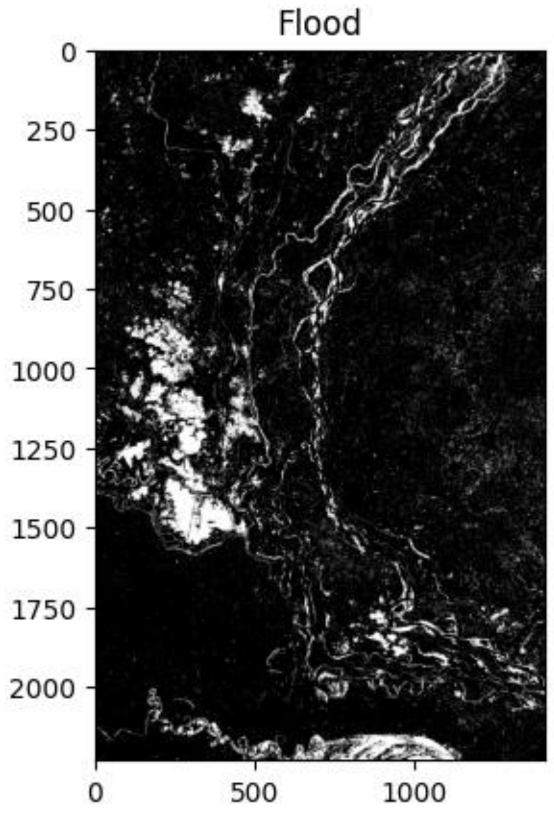
PREDICITONS:

```
1/1 [==============================] - 2s 2s/step
Predictions: [[0.0811001  0.91889995]]
```



91.889995 % ACCURACY

```
1/1 [==============================] - 0s 343ms/step
Predictions: [[0.06725024 0.9327498 ]]
```



93.27498 % ACCURACY

```
1/1 [==============================] - 0s 229ms/step
Predictions: [[0.9671773  0.03282268]]
```

96.71773 % ACCURACY

Vision Transformer (ViT) CLASSIFICATION:

LOADING IMAGES, CLASSIFICATION AND PRE-PROCESSING:

```
# Custom dataset class for grayscale images
class GrayscaleImageDataset(Dataset):
    def __init__(self, image_folder, transform=None):
        self.image_folder = image_folder
        self.transform = transform
        self.image_paths = []
        self.labels = []

        # Load image paths and labels
        for filename in os.listdir(image_folder):
            filepath = os.path.join(image_folder, filename)
            self.image_paths.append(filepath)
            # Assign label based on filename
            if "non_" in filename.lower():
                self.labels.append(0)  # non-flood
            else:
                self.labels.append(1)  # flood

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        # Handle potential errors when opening images
        try:
            image = np.array(Image.open(image_path).convert('L'))  # Load as grayscale
        except Exception as e:
            print(f"Error loading image {image_path}: {e}")
            return None, None  # Return None for both image and label if error occurs

        image = np.stack([image] * 3, axis=-1)  # Convert to 3-channel format

        if self.transform:
            image = self.transform(image)

        label = self.labels[idx]
        return image, label


# Load datasets
train_dataset = GrayscaleImageDataset('/content/drive/MyDrive/extra_ViT/train',
transform=transform)
val_dataset = GrayscaleImageDataset('/content/drive/MyDrive/extra_ViT/valid',
transform=transform)

# Filter out None values from dataset
train_dataset = [(img, lbl) for img, lbl in train_dataset if img is not None]
val_dataset = [(img, lbl) for img, lbl in val_dataset if img is not None]

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
```

The **GrayscaleImageDataset** class is a custom dataset for handling grayscale images in PyTorch. It is designed to work with images stored in a folder and labels them based on filename conventions. Here's a breakdown of its functionality:

1. **Initialization (__init__ Method):**
    - **Inputs:**
        - image_folder: The path to the folder containing the image files.
        - transform: An optional transformation to be applied to the images.
    - **Operations:**
        - Initializes lists for storing image file paths and their corresponding labels.
        - Iterates through each file in the specified folder. Based on the filename, it assigns a label: 0 for "non-flood" (if "non_" is in the filename) and 1 for "flood" (if "non_" is not present).
2. **Length (__len__ Method):**
    - Returns the total number of images in the dataset.
3. **Get Item (__getitem__ Method):**
    - **Inputs:**
        - idx: Index to fetch a specific image and its label.
    - **Operations:**
        - Loads the image at the specified index as a grayscale image.
        - Converts the grayscale image to a 3-channel format by duplicating the single channel three times (to match the input format expected by most deep learning models).
        - Applies any specified transformations.
        - Returns the image and its corresponding label. If there is an error in loading the image, it prints an error message and returns None for both the image and label.

Key Points:

- **Label Assignment:** Labels are based on the presence of "non_" in the filename, distinguishing between flood and non-flood images.
- **Error Handling:** Includes error handling to skip images that cannot be loaded.
- **Transformation:** Supports image transformations through the transform parameter, allowing for preprocessing steps like resizing and normalization.

MODEL ARCHITECTURE:

PSEUDOCODE:

```
# Load pre-trained Vision Transformer model with adjusted output size
model = ViTForImageClassification.from_pretrained(
    'google/vit-base-patch16-224',
    num_labels=2,
    ignore_mismatched_sizes=True
)
model.to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)

# Training loop
num_epochs = 10
model.train()
for epoch in range(num_epochs):
    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images).logits
        loss = criterion(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

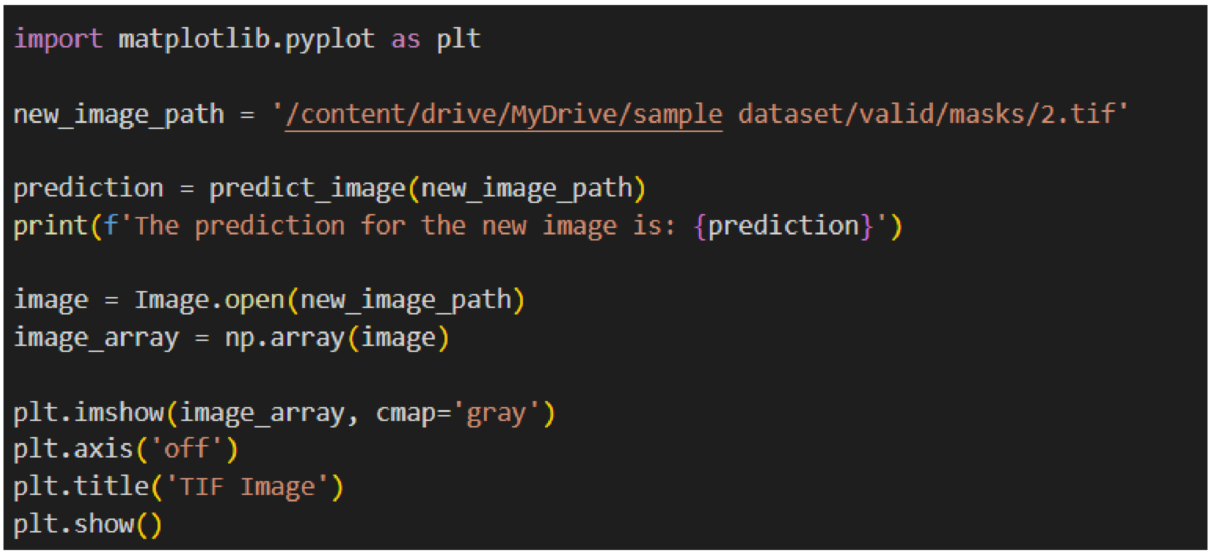
1. **Model Initialization:**
    - **Pre-trained Vision Transformer (ViT):** The ViTForImageClassification model is loaded from the google/vit-base-patch16-224 checkpoint. This model, originally trained on ImageNet, is adapted for the task by specifying num_labels=2 to accommodate binary classification (flood vs. non-flood). The ignore_mismatched_sizes=True argument allows for adaptation of the model's architecture to match the new number of output labels if there are size mismatches.
2. **Device Configuration:**
    - **Device Setup:** The model is moved to the GPU if available, otherwise, it runs on the CPU. This is done using model.to(device), where device is set to 'cuda' if a GPU is available, otherwise 'cpu'.
3. **Loss Function and Optimizer:**
    - **Loss Function:** nn.CrossEntropyLoss() is used to compute the loss between the predicted class scores and the true labels.
    - **Optimizer:** optim.Adam() is chosen with a learning rate of 1e-4 to update model parameters during training.
4. **Training Loop:**
    - **Epochs:** The model is trained for 10 epochs.
    - **Batch Processing:**
        - For each epoch, the model iterates over batches of images and labels from the train_loader.
        - Images and labels are transferred to the appropriate device.
        - The model generates predictions (logits) for the input images.
        - The loss is calculated using the predictions and the true labels.
        - **Backpropagation and Optimization:**
            - Gradients are zeroed using optimizer.zero_grad().
            - Loss gradients are computed with loss.backward().
            - Model parameters are updated with optimizer.step().
    - **Loss Reporting:** After each epoch, the loss value is printed to monitor the training process.

This setup ensures that the ViT model is fine-tuned for the specific task of distinguishing between flood and non-flood images, adjusting the model parameters to improve performance on this binary classification task.

PREDICTIONS:

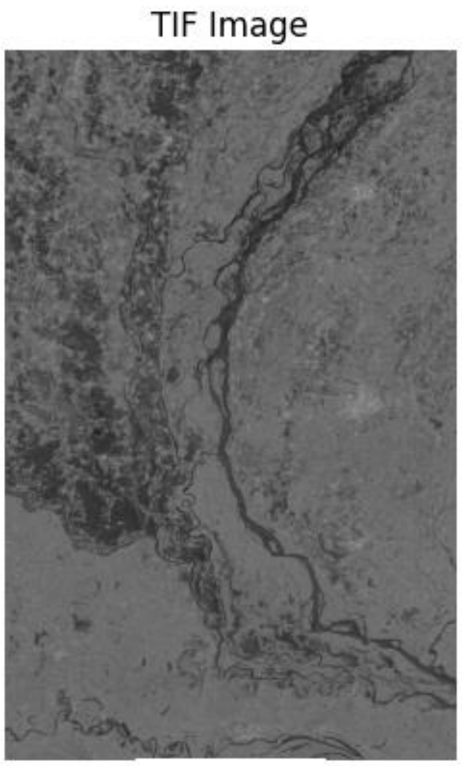
PSEUDOCODE:

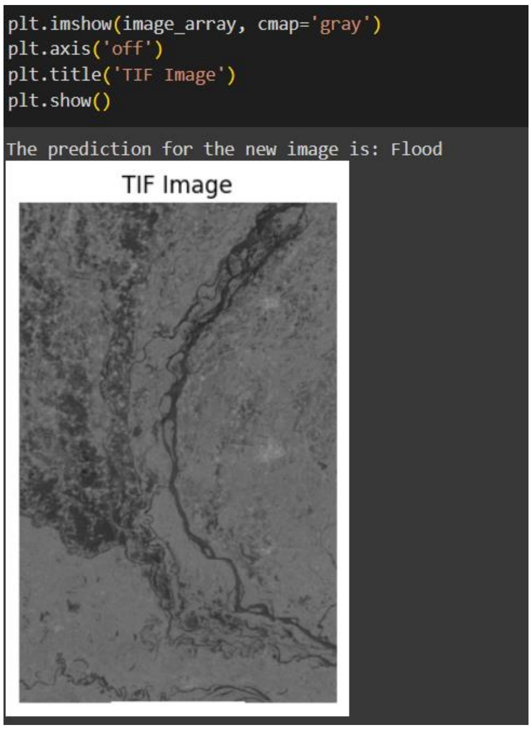
```
import matplotlib.pyplot as plt

new_image_path = '/content/drive/MyDrive/sample dataset/valid/masks/2.tif'

prediction = predict_image(new_image_path)
print(f'The prediction for the new image is: {prediction}')

image = Image.open(new_image_path)
image_array = np.array(image)

plt.imshow(image_array, cmap='gray')
plt.axis('off')
plt.title('TIF Image')
plt.show()
```



```
import matplotlib.pyplot as plt

new_image_path = '/content/drive/MyDrive/sample dataset/valid/masks/2.tif'

prediction = predict_image(new_image_path)
print(f'The prediction for the new image is: {prediction}')

image = Image.open(new_image_path)
image_array = np.array(image)

plt.imshow(image_array, cmap='gray')
plt.axis('off')
plt.title('TIF Image')
plt.show()
```

The prediction for the new image is: Flood

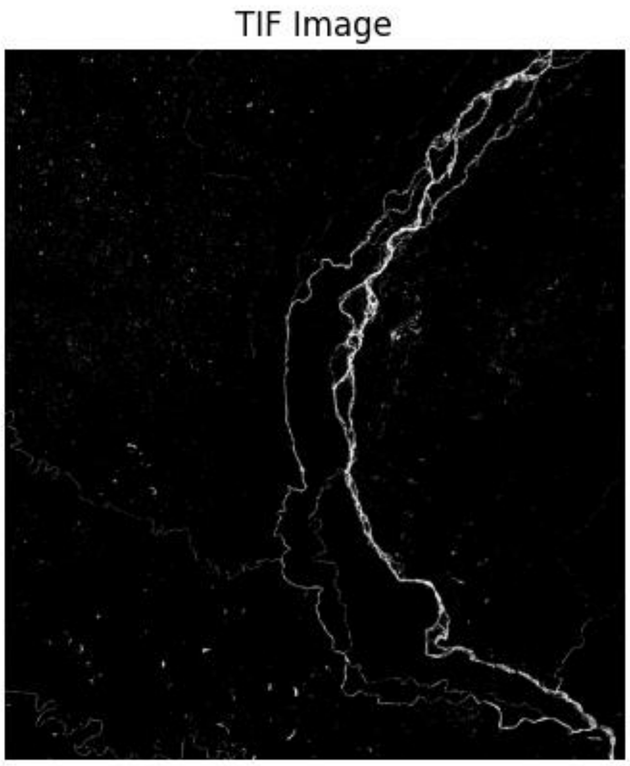


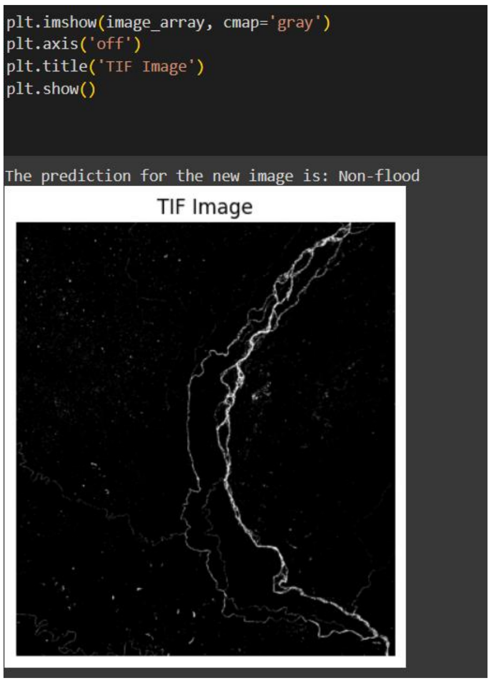


```
plt.imshow(image_array, cmap='gray')
plt.axis('off')
plt.title('TIF Image')
plt.show()
```

```
The prediction for the new image is: Flood
```

The prediction for the new image is: Non-flood

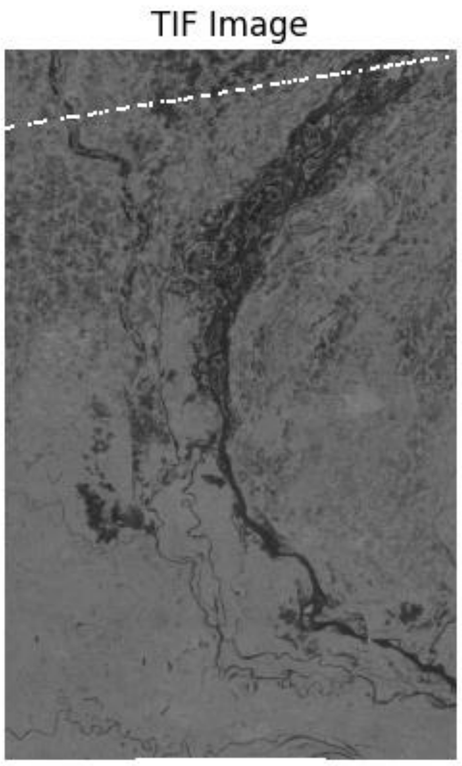


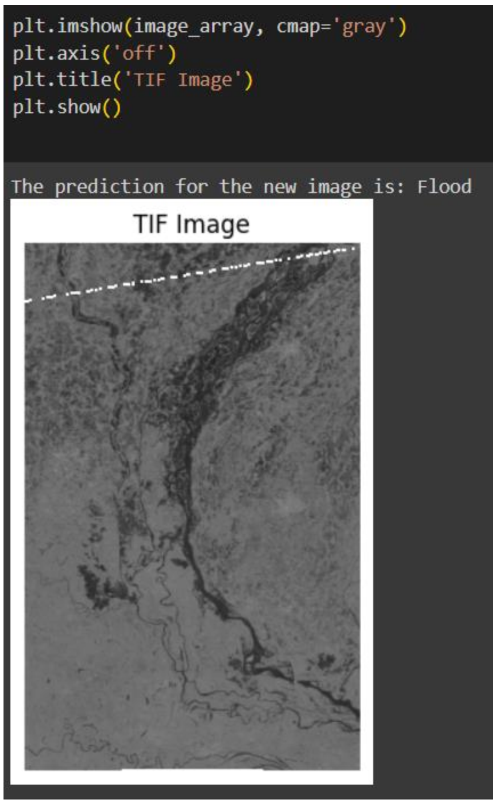


```
plt.imshow(image_array, cmap='gray')
plt.axis('off')
plt.title('TIF Image')
plt.show()
```

```
The prediction for the new image is: Non-flood
```

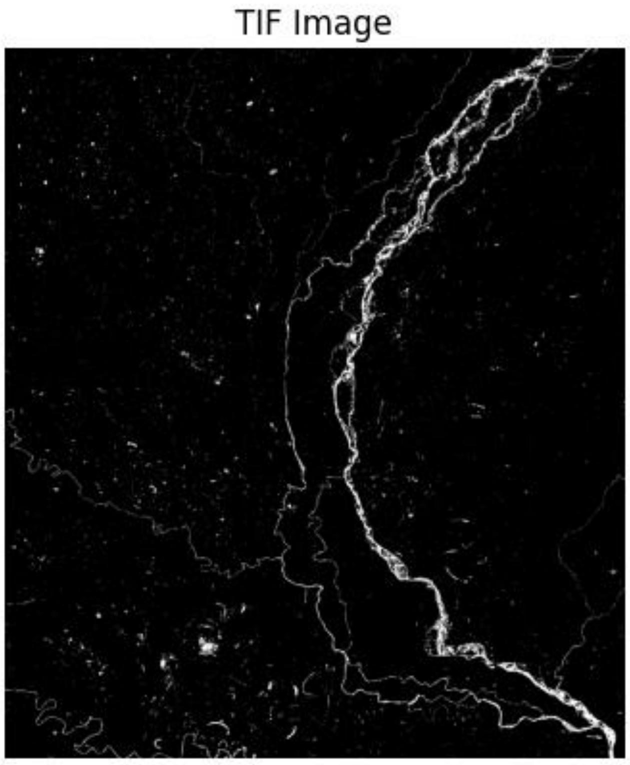
The prediction for the new image is: Flood

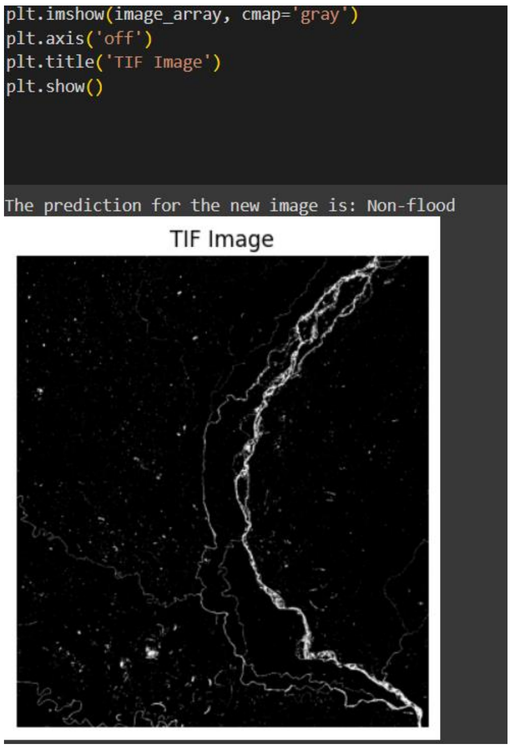



```
plt.imshow(image_array, cmap='gray')
plt.axis('off')
plt.title('TIF Image')
plt.show()
```

```
The prediction for the new image is: Flood
```

The prediction for the new image is: Non-flood




```
plt.imshow(image_array, cmap='gray')
plt.axis('off')
plt.title('TIF Image')
plt.show()
```

```
The prediction for the new image is: Non-flood
```

U-Net Image Prediction & Generation

DATA LOADING AND PRE-PROCESSING:

PSEUDOCODE:

```
# Load and preprocess the dataset
def load_images_from_folder(folder):
    images = []
    for filename in glob.glob(os.path.join(folder, '*.tif')):
        img = tiff.imread(filename)
        if img.ndim == 3:  # Convert to grayscale if the image is in RGB
            img = tf.image.rgb_to_grayscale(img)
        img = tf.image.resize(img, [256, 256])
        img = img / 255.0  # Normalize to [0, 1]
        images.append(img.numpy())
    return np.array(images)

data_path = '/content/drive/MyDrive/ConvLSTM full/'

all_images = []
years = os.listdir(data_path)
for year in years:
  if year == '2023':
    year_path = os.path.join(data_path, year)
    images = load_images_from_folder(year_path)
    all_images.extend(images)

all_images = np.array(all_images)
print(f"Loaded {all_images.shape[0]} images.")
```

```
# Split the dataset

X_train, X_test = train_test_split(all_images, test_size=0.2, random_state=42)


# Reshape for UNet input

X_train = np.expand_dims(X_train, axis=-1)

X_test = np.expand_dims(X_test, axis=-1)
```

Summary of the Code

1. **Loading and Preprocessing Images:**
    - The function load_images_from_folder reads .tif images from a specified folder.
    - It converts RGB images to grayscale, resizes them to 256x256 pixels, and normalizes pixel values to the range [0, 1].
    - Processed images are stored in a list and returned as a NumPy array.
2. **Dataset Loading:**
    - The main script specifies the data path and iterates over the folders within this path.
    - It specifically processes images from the "2023" folder by calling the load_images_from_folder function.
    - All processed images are collected into a list and then converted to a NumPy array.
3. **Dataset Splitting:**
    - The dataset of images is split into training and testing sets using an 80-20 split.
    - X_train and X_test are reshaped to add an additional dimension, making them compatible with the input requirements of a UNet model.
4. **Output:**
    - The total number of loaded images is printed to the console.

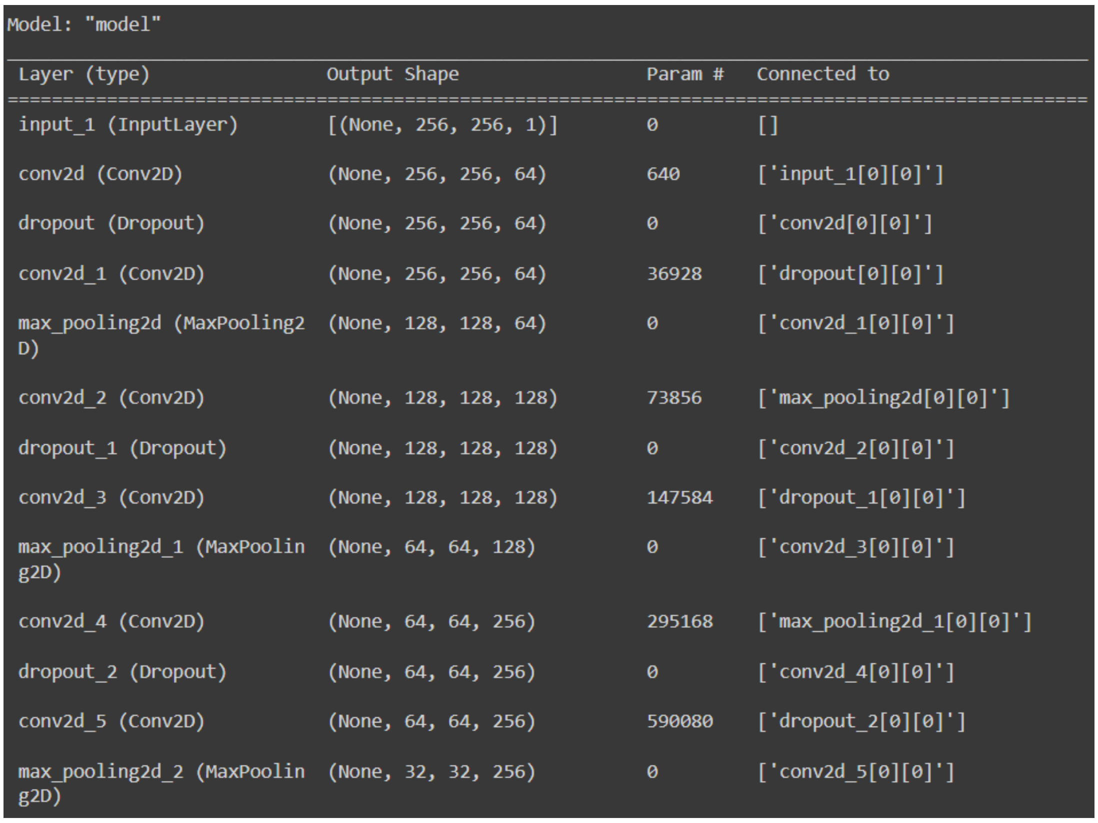
MODEL ARCHITECTURE:

```
# Define the UNet model

def unet_model(input_size=(256, 256, 1)):

  inputs = Input(input_size)


  c1 = Conv2D(64, (3, 3), activation='relu', padding='same')(inputs)

  c1 = Dropout(0.1)(c1)

  c1 = Conv2D(64, (3, 3), activation='relu', padding='same')(c1)

  p1 = MaxPooling2D((2, 2))(c1)


  c2 = Conv2D(128, (3, 3), activation='relu', padding='same')(p1)

  c2 = Dropout(0.1)(c2)

  c2 = Conv2D(128, (3, 3), activation='relu', padding='same')(c2)

  p2 = MaxPooling2D((2, 2))(c2)


  c3 = Conv2D(256, (3, 3), activation='relu', padding='same')(p2)

  c3 = Dropout(0.2)(c3)

  c3 = Conv2D(256, (3, 3), activation='relu', padding='same')(c3)

  p3 = MaxPooling2D((2, 2))(c3)


  c4 = Conv2D(512, (3, 3), activation='relu', padding='same')(p3)

  c4 = Dropout(0.2)(c4)

  c4 = Conv2D(512, (3, 3), activation='relu', padding='same')(c4)

  p4 = MaxPooling2D((2, 2))(c4)


  c5 = Conv2D(1024, (3, 3), activation='relu', padding='same')(p4)

  c5 = Dropout(0.3)(c5)

  c5 = Conv2D(1024, (3, 3), activation='relu', padding='same')(c5)


  u6 = UpSampling2D((2, 2))(c5)
```

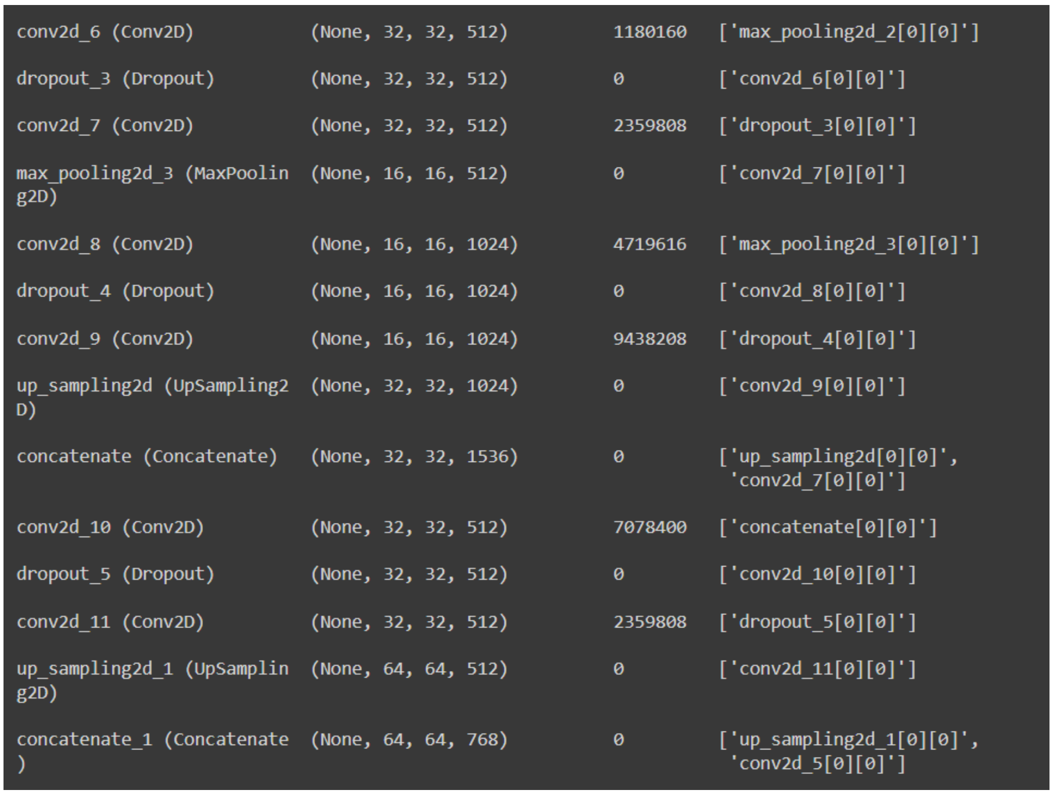
```
    u6 = concatenate([u6, c4])
    c6 = Conv2D(512, (3, 3), activation='relu', padding='same')(u6)
    c6 = Dropout(0.2)(c6)
    c6 = Conv2D(512, (3, 3), activation='relu', padding='same')(c6)

    u7 = UpSampling2D((2, 2))(c6)
    u7 = concatenate([u7, c3])
    c7 = Conv2D(256, (3, 3), activation='relu', padding='same')(u7)
    c7 = Dropout(0.2)(c7)
    c7 = Conv2D(256, (3, 3), activation='relu', padding='same')(c7)

    u8 = UpSampling2D((2, 2))(c7)
    u8 = concatenate([u8, c2])
    c8 = Conv2D(128, (3, 3), activation='relu', padding='same')(u8)
    c8 = Dropout(0.1)(c8)
    c8 = Conv2D(128, (3, 3), activation='relu', padding='same')(c8)

    u9 = UpSampling2D((2, 2))(c8)
    u9 = concatenate([u9, c1])
    c9 = Conv2D(64, (3, 3), activation='relu', padding='same')(u9)
    c9 = Dropout(0.1)(c9)
    c9 = Conv2D(64, (3, 3), activation='relu', padding='same')(c9)

    outputs = Conv2D(1, (1, 1), activation='sigmoid')(c9)

    model = Model(inputs=[inputs], outputs=[outputs])
    model.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])

    return model

# Print model summary
```

model = unet_model()

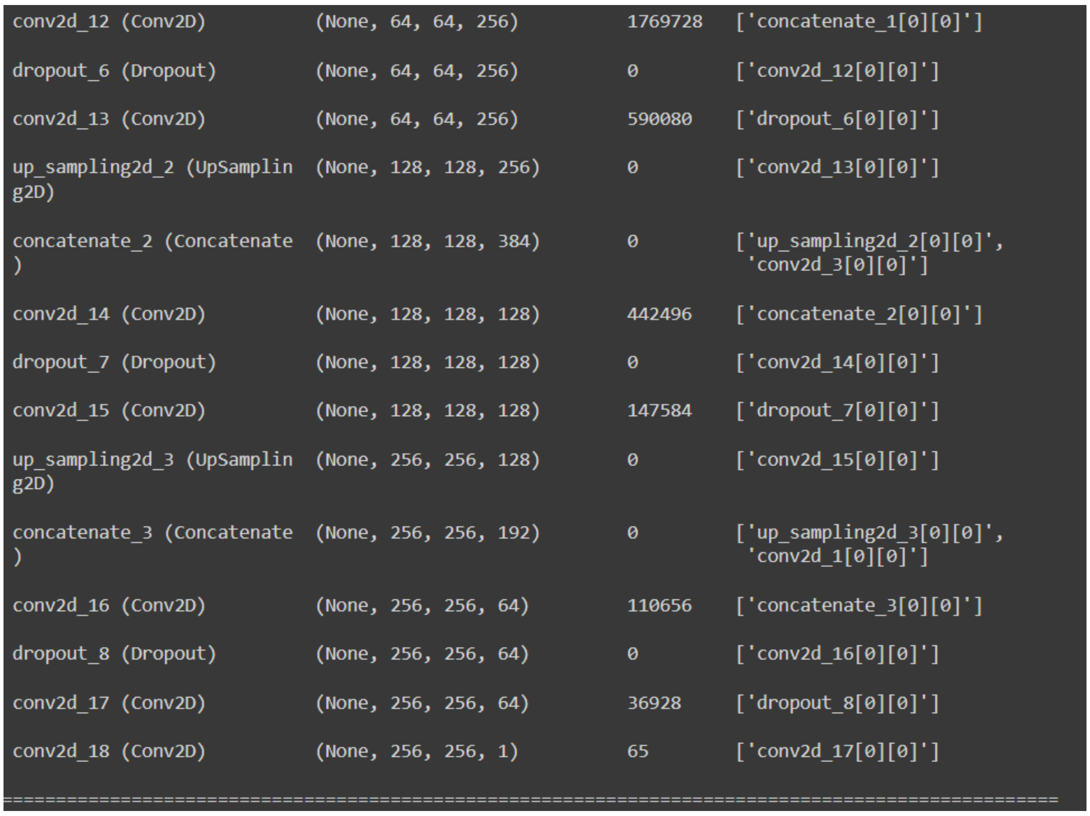
model.summary()



```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 256, 256, 1)]  0         []

 conv2d (Conv2D)                (None, 256, 256, 64)  640        ['input_1[0][0]']

 dropout (Dropout)              (None, 256, 256, 64)  0          ['conv2d[0][0]']

 conv2d_1 (Conv2D)              (None, 256, 256, 64)  36928      ['dropout[0][0]']

 max_pooling2d (MaxPooling2     (None, 128, 128, 64)  0          ['conv2d_1[0][0]']
 D)

 conv2d_2 (Conv2D)              (None, 128, 128, 128)  73856     ['max_pooling2d[0][0]']

 dropout_1 (Dropout)            (None, 128, 128, 128)  0         ['conv2d_2[0][0]']

 conv2d_3 (Conv2D)              (None, 128, 128, 128)  147584    ['dropout_1[0][0]']

 max_pooling2d_1 (MaxPoolin     (None, 64, 64, 128)  0           ['conv2d_3[0][0]']
 g2D)

 conv2d_4 (Conv2D)              (None, 64, 64, 256)  295168      ['max_pooling2d_1[0][0]']

 dropout_2 (Dropout)            (None, 64, 64, 256)  0           ['conv2d_4[0][0]']

 conv2d_5 (Conv2D)              (None, 64, 64, 256)  590080      ['dropout_2[0][0]']

 max_pooling2d_2 (MaxPoolin     (None, 32, 32, 256)  0           ['conv2d_5[0][0]']
 g2D)
```



```
 conv2d_6 (Conv2D)              (None, 32, 32, 512)  1180160     ['max_pooling2d_2[0][0]']

 dropout_3 (Dropout)            (None, 32, 32, 512)  0           ['conv2d_6[0][0]']

 conv2d_7 (Conv2D)              (None, 32, 32, 512)  2359808     ['dropout_3[0][0]']

 max_pooling2d_3 (MaxPoolin     (None, 16, 16, 512)  0           ['conv2d_7[0][0]']
 g2D)

 conv2d_8 (Conv2D)              (None, 16, 16, 1024)  4719616    ['max_pooling2d_3[0][0]']

 dropout_4 (Dropout)            (None, 16, 16, 1024)  0          ['conv2d_8[0][0]']

 conv2d_9 (Conv2D)              (None, 16, 16, 1024)  9438208    ['dropout_4[0][0]']

 up_sampling2d (UpSampling2     (None, 32, 32, 1024)  0          ['conv2d_9[0][0]']
 D)

 concatenate (Concatenate)      (None, 32, 32, 1536)  0          ['up_sampling2d[0][0]',
                                                                  'conv2d_7[0][0]']

 conv2d_10 (Conv2D)             (None, 32, 32, 512)  7078400     ['concatenate[0][0]']

 dropout_5 (Dropout)            (None, 32, 32, 512)  0           ['conv2d_10[0][0]']

 conv2d_11 (Conv2D)             (None, 32, 32, 512)  2359808     ['dropout_5[0][0]']

 up_sampling2d_1 (UpSamplin     (None, 64, 64, 512)  0           ['conv2d_11[0][0]']
 g2D)

 concatenate_1 (Concatenate     (None, 64, 64, 768)  0           ['up_sampling2d_1[0][0]',
 )                                                                'conv2d_5[0][0]']
```



```
conv2d_12 (Conv2D)          (None, 64, 64, 256)         1769728    ['concatenate_1[0][0]']

dropout_6 (Dropout)         (None, 64, 64, 256)         0          ['conv2d_12[0][0]']

conv2d_13 (Conv2D)          (None, 64, 64, 256)         590080     ['dropout_6[0][0]']

up_sampling2d_2 (UpSamplin  (None, 128, 128, 256)       0          ['conv2d_13[0][0]']
g2D)

concatenate_2 (Concatenate  (None, 128, 128, 384)       0          ['up_sampling2d_2[0][0]',
)                                                                   'conv2d_3[0][0]']

conv2d_14 (Conv2D)          (None, 128, 128, 128)       442496     ['concatenate_2[0][0]']

dropout_7 (Dropout)         (None, 128, 128, 128)       0          ['conv2d_14[0][0]']

conv2d_15 (Conv2D)          (None, 128, 128, 128)       147584     ['dropout_7[0][0]']

up_sampling2d_3 (UpSamplin  (None, 256, 256, 128)       0          ['conv2d_15[0][0]']
g2D)

concatenate_3 (Concatenate  (None, 256, 256, 192)       0          ['up_sampling2d_3[0][0]',
)                                                                   'conv2d_1[0][0]']

conv2d_16 (Conv2D)          (None, 256, 256, 64)        110656     ['concatenate_3[0][0]']

dropout_8 (Dropout)         (None, 256, 256, 64)        0          ['conv2d_16[0][0]']

conv2d_17 (Conv2D)          (None, 256, 256, 64)        36928      ['dropout_8[0][0]']

conv2d_18 (Conv2D)          (None, 256, 256, 1)         65         ['conv2d_17[0][0]']

==================================================================================================
```

Explanation and Summary of the UNet Model Architecture

Explanation:

1. **Inputs:**
    - The input layer accepts images of size 256x256 with 1 channel (grayscale).
2. **Contracting Path (Encoder):**
    - **Block 1:** Two convolutional layers with 64 filters each, followed by a dropout layer and max pooling.
    - **Block 2:** Two convolutional layers with 128 filters each, followed by a dropout layer and max pooling.
    - **Block 3:** Two convolutional layers with 256 filters each, followed by a dropout layer and max pooling.
    - **Block 4:** Two convolutional layers with 512 filters each, followed by a dropout layer and max pooling.
    - **Block 5 (Bottom of the U):** Two convolutional layers with 1024 filters each, followed by a dropout layer.
3. **Expanding Path (Decoder):**

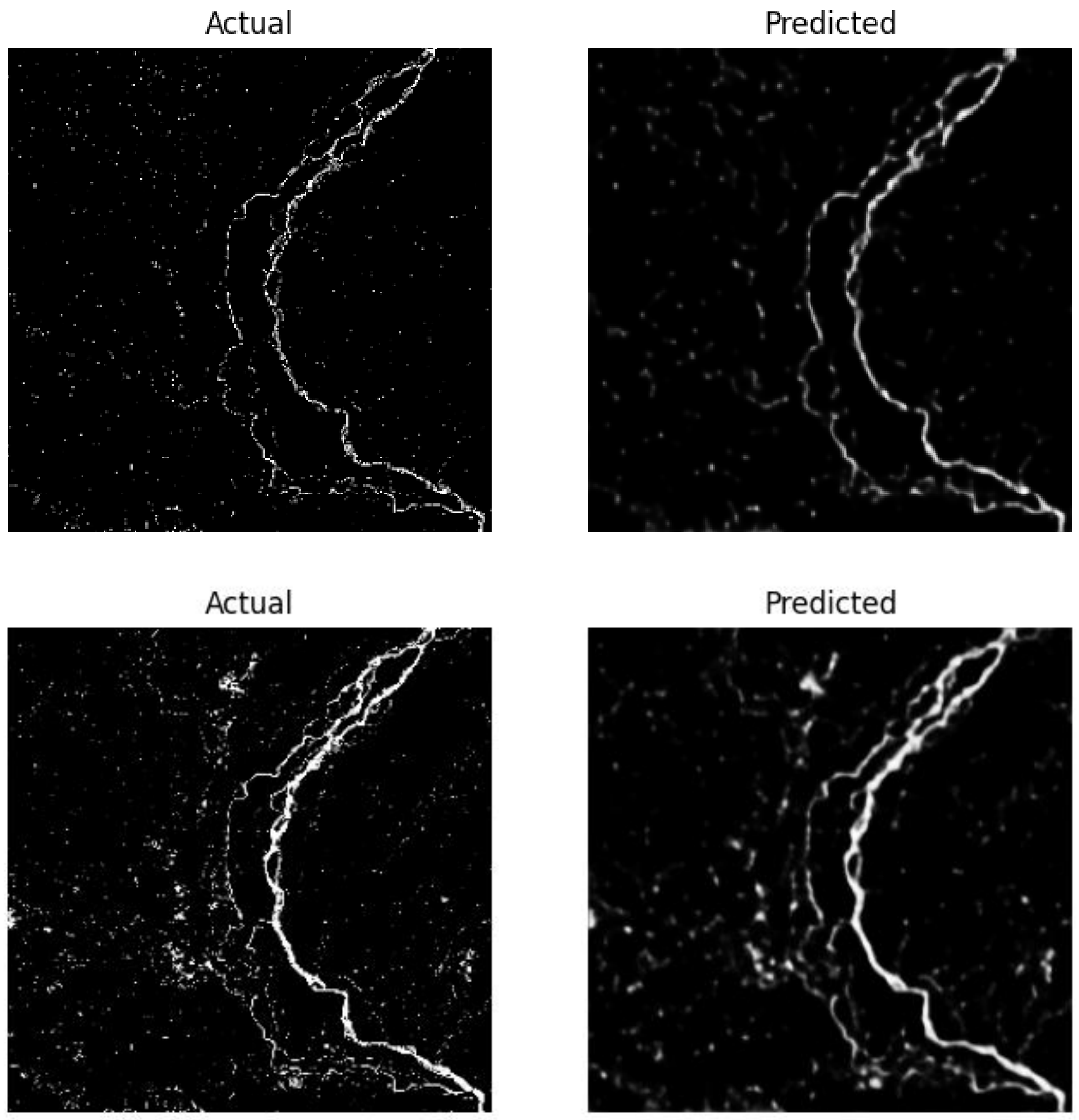
   - **Up Block 1:** Upsampling followed by concatenation with the corresponding feature map from the contracting path, and two convolutional layers with 512 filters each and a dropout layer.
   - **Up Block 2:** Upsampling followed by concatenation, and two convolutional layers with 256 filters each and a dropout layer.
   - **Up Block 3:** Upsampling followed by concatenation, and two convolutional layers with 128 filters each and a dropout layer.
   - **Up Block 4:** Upsampling followed by concatenation, and two convolutional layers with 64 filters each and a dropout layer.
4. **Output:**
   - A final convolutional layer with a single filter and a sigmoid activation function to produce the output mask.
5. **Compilation:**
   - The model is compiled with the Adam optimizer, binary cross-entropy loss, and accuracy as a metric.

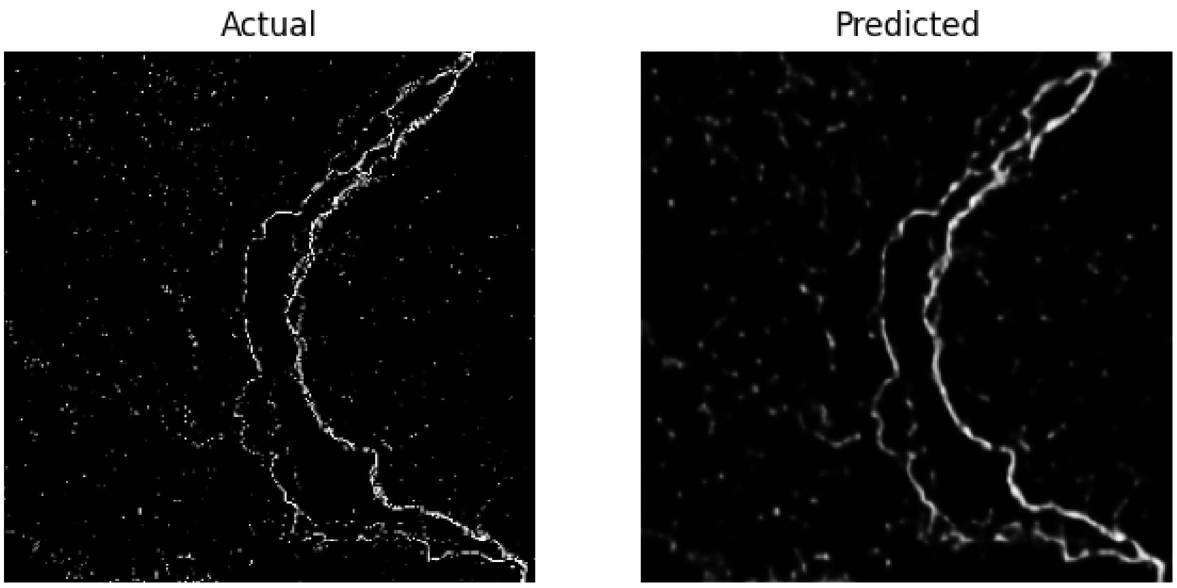
Summary:

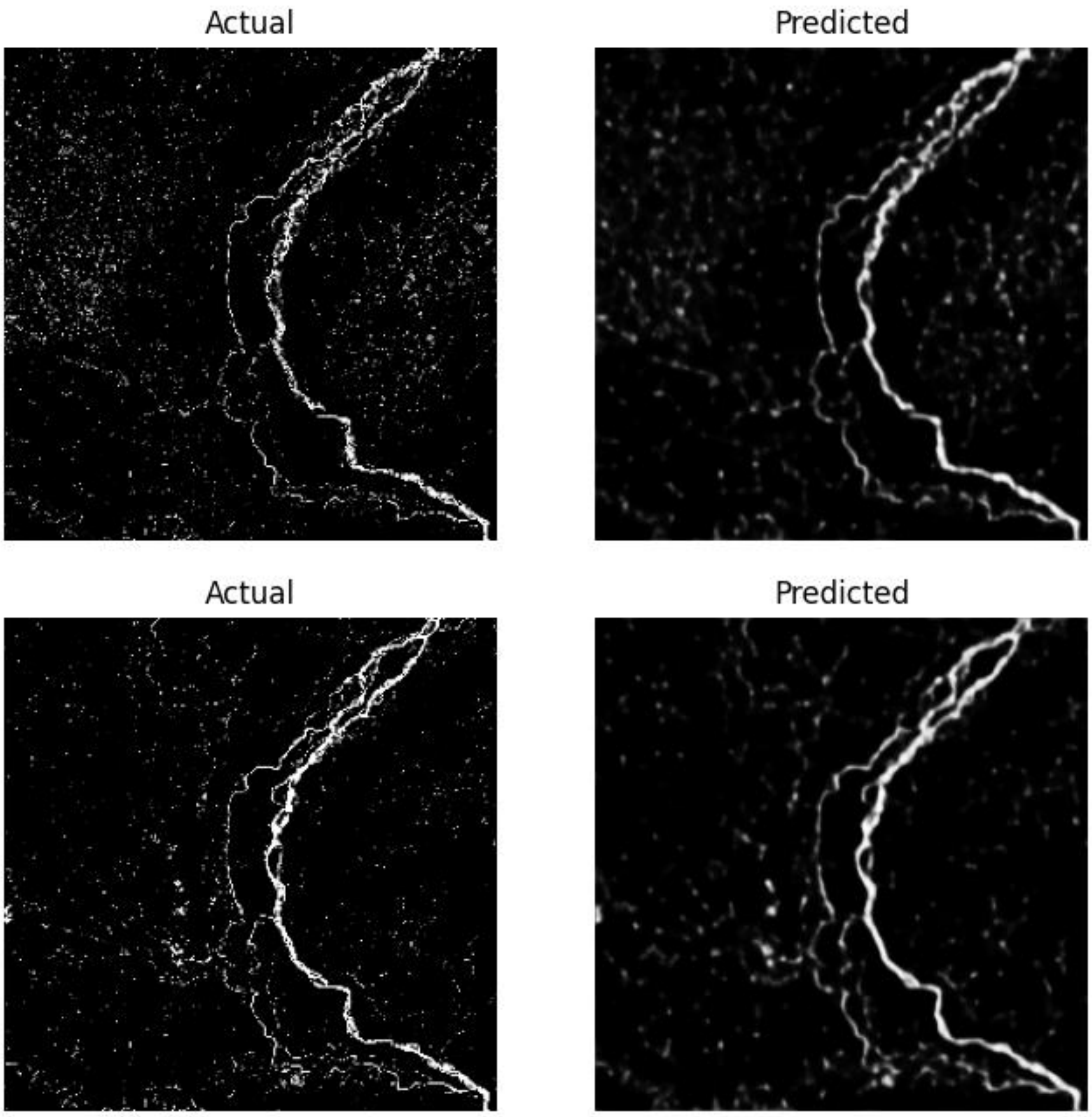
The UNet model is a convolutional neural network designed for image segmentation tasks. It has an encoder-decoder architecture with the following characteristics:

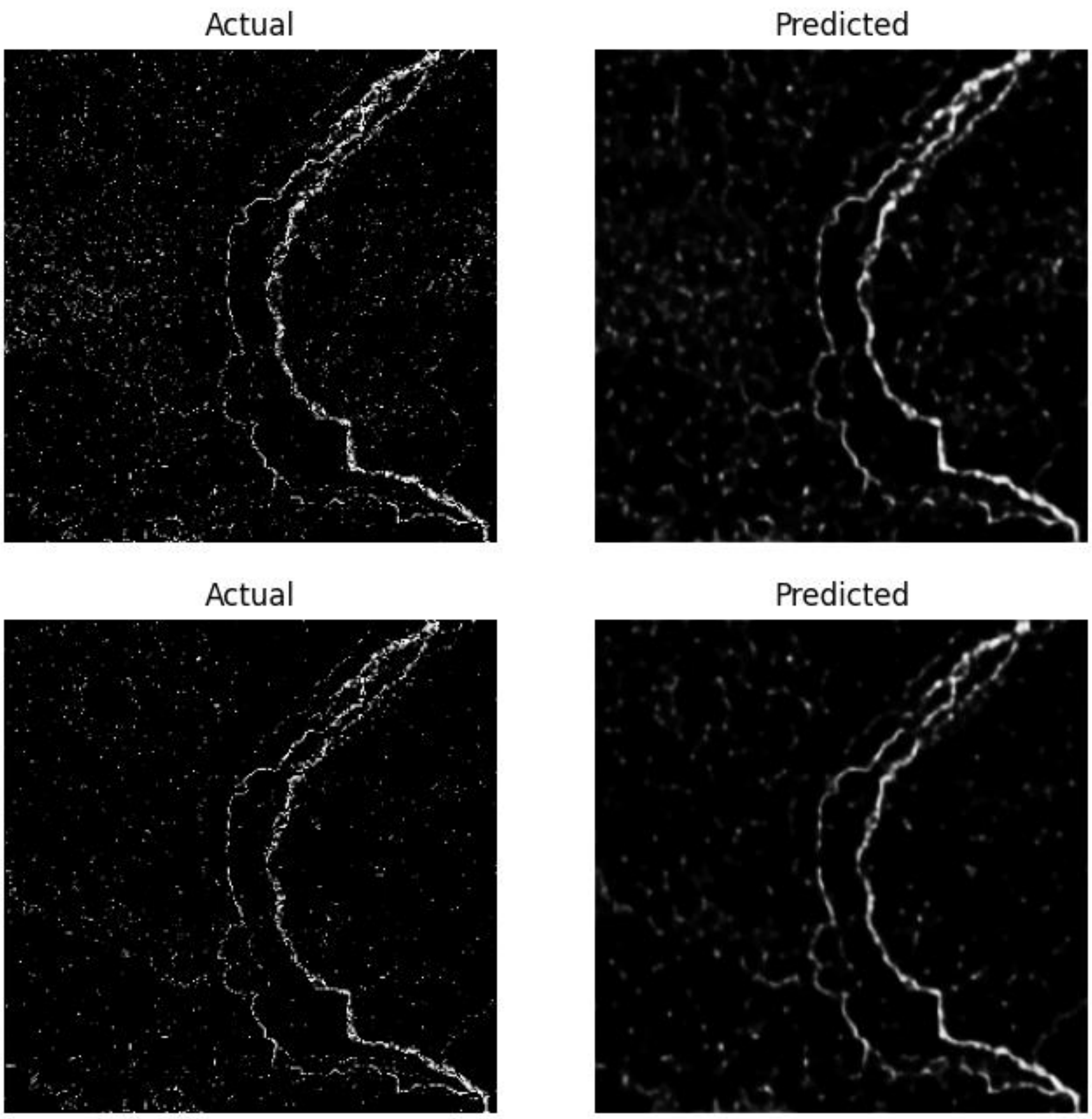
- **Input Size:** 256x256x1 (grayscale images)
- **Encoder (Contracting Path):**
  - Consists of four blocks, each with two convolutional layers, followed by dropout and max pooling.
- **Bottleneck:**
  - Contains two convolutional layers with 1024 filters and dropout.
- **Decoder (Expanding Path):**
  - Contains four upsampling blocks, each followed by concatenation with corresponding encoder layers, two convolutional layers, and dropout.
- **Output:**
  - A single convolutional layer with a sigmoid activation function to generate the output segmentation mask.
- **Compilation:**
  - The model uses the Adam optimizer, binary cross-entropy loss, and accuracy as a performance metric.

PREDICTIONS:




Actual
Predicted


Actual
Predicted
Actual
Predicted


Actual
Predicted
Actual
Predicted

<u>EVALUATION METRICS:</u>

BATCH SIZE = 32

1/1 [==============================] - 12s 12s/step

MSE: 0.014805790036916733, MAE: 0.022873954847455025, RMSE: 0.12167904525995255, R^2: -0.05665513717377002

BATCH SIZE = 16

1/1 [==============================] - 0s 21ms/step

MSE: 0.014636874198913574, MAE: 0.054647572338581085, RMSE: 0.12098295241594315, R^2: -0.0445995653609785

BATCH SIZE = 8

1/1 [==============================] - 0s 26ms/step

MSE: 0.006825014483183622, MAE: 0.02093164063990116, RMSE: 0.08261364698410034, R^2: 0.512914571990879

EPOCHS = 50

(OVERFITTING !!!!)

1/1 [==============================] - 0s 22ms/step

MSE: 0.00044486799743026495, MAE: 0.004357580561190844, RMSE: 0.021091893315315247, **R^2: 0.9682508021572447**

CNN+LSTM IMAGE PREDICTION AND GENERATION:

IMAGE LOADING, PRE-PROCESSING:

PSEUDOCODE:

```
import os

import numpy as np

from datetime import datetime

from tensorflow.keras.preprocessing.image import load_img, img_to_array


# Function to load .tif image, resize, and preprocess it

def load_tif_image(file_path, target_size=(64, 64)):

  img = load_img(file_path, color_mode='grayscale', target_size=target_size)

  img = img_to_array(img)

  img = img / 255.0  # Normalize to [0, 1]

  return img


# Function to load dataset from directory with resizing and additional debug prints

def load_dataset_from_directory(directory, target_size=(64, 64)):

  images = []

  dates = []


  for filename in sorted(os.listdir(directory)):

    if filename.endswith('.tif'):

      date_str = filename.split('.')[0]  # Extract date string from filename

      try:

        date = datetime.strptime(date_str, '%Y-%m-%d')  # Convert to datetime object

        img_path = os.path.join(directory, filename)

        print(f"Loading image: {img_path}")

        img = load_tif_image(img_path, target_size=target_size)

        images.append(img)
```

```
            dates.append(date)
        except ValueError as e:
            print(f"Skipping file {filename}: {e}")

    return np.array(images), np.array(dates)

    # for month in sorted(os.listdir(directory)):
    #     month_dir = os.path.join(directory, month)
    #     if os.path.isdir(month_dir):
    #         print(f"Found month directory: {month_dir}")
    #         for day in sorted(os.listdir(month_dir)):
    #             if day.endswith('.tif'):
    #                 date_str = day.split('.')[0]  # Extract date string
    #                 try:
    #                     date = datetime.strptime(date_str, '%Y-%m-%d')  # Convert to datetime object
    #                     img_path = os.path.join(month_dir, day)
    #                     print(f"Loading image: {img_path}")
    #                     img = load_tif_image(img_path, target_size=target_size)
    #                     images.append(img)
    #                     dates.append(date)
    #                 except ValueError as e:
    #                     print(f"Skipping file {day}: {e}")

    # return np.array(images), np.array(dates)

# Test usage
directory = '/content/drive/MyDrive/ConvLSTM full/2023'
#directory = '/content/drive/MyDrive/ConvLSTM full'
target_size = (64, 64)
images, dates = load_dataset_from_directory(directory, target_size=target_size)
```

```
# Check the shapes of loaded data
print(f"Images shape: {images.shape}")
print(f"Dates shape: {dates.shape}")

# Ensure images array is not empty
if images.size == 0:
    raise ValueError("No images found. Check the directory structure and file paths.")
```

Summary Explanation

This code defines functions for loading and preprocessing grayscale .tif images from a specified directory, while also extracting date information from the filenames. Here's a detailed breakdown:

1. **Import Necessary Libraries:**
   - os: For directory and file path operations.
   - numpy: For array operations.
   - datetime: For date parsing.
   - load_img and img_to_array from tensorflow.keras.preprocessing.image: For image loading and preprocessing.
2. **Function load_tif_image:**
   - Loads a .tif image, resizes it to the target size (default 64x64), and converts it to an array.
   - Converts the image to grayscale.
   - Normalizes the pixel values to the range [0, 1].
3. **Function load_dataset_from_directory:**
   - Iterates over all files in the specified directory.
   - Checks if filenames end with .tif.
   - Extracts the date from the filename (assumes format YYYY-MM-DD.tif).
   - Tries to parse the extracted date string into a datetime object.
   - Loads and preprocesses the image using load_tif_image.
   - Appends the processed image and the corresponding date to lists images and dates.

   - Handles potential ValueError exceptions during date parsing, skipping files with invalid date formats.
   - Returns arrays of images and dates.

4. **Usage Example:**
   - Sets the directory path to load images from.
   - Specifies the target size for image resizing.
   - Calls load_dataset_from_directory to load images and dates.
   - Prints the shapes of the resulting image and date arrays.
   - Checks if the images array is empty and raises an error if no images are found.

Purpose:

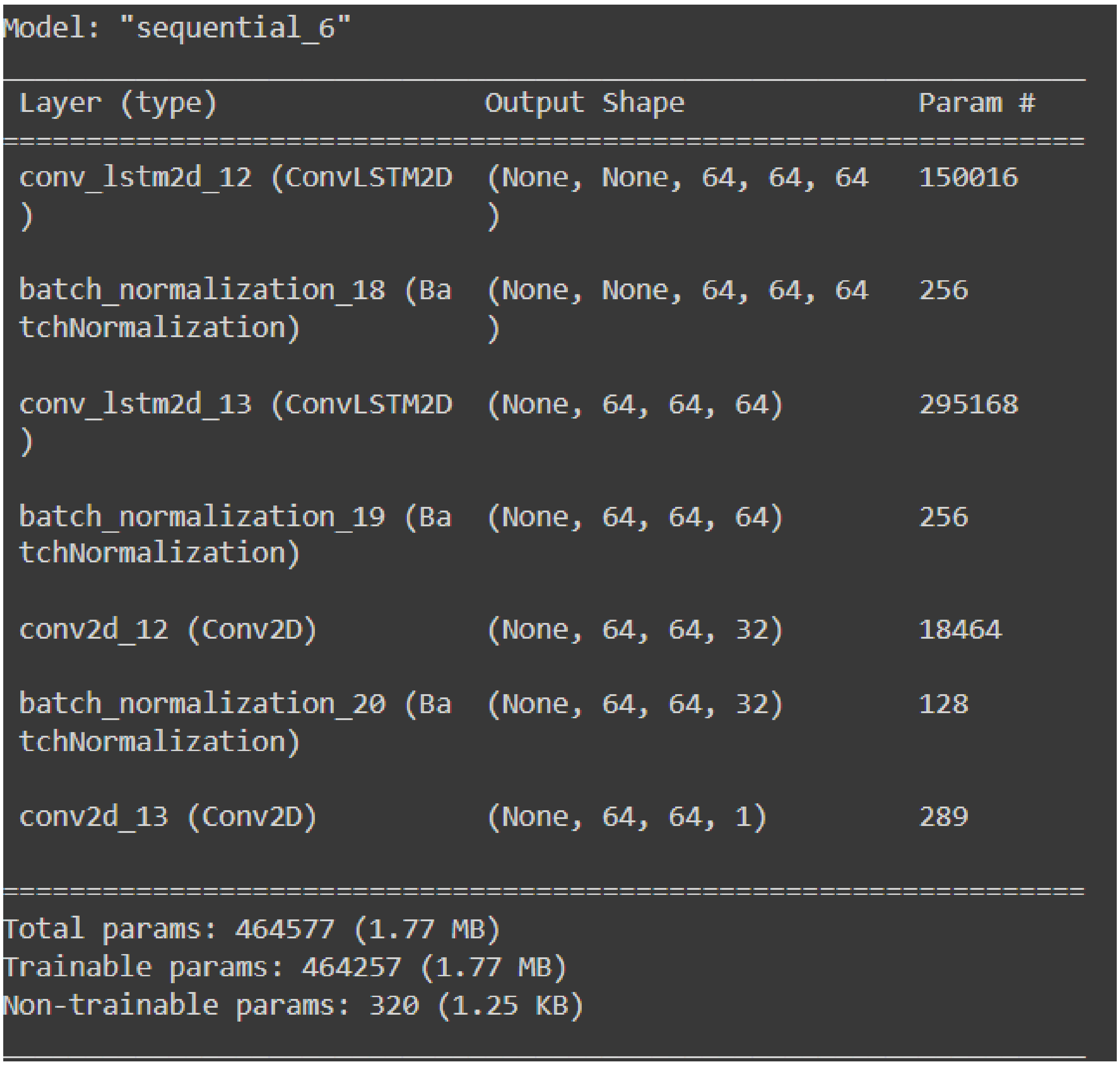
- **Loading and Preprocessing**: The code loads grayscale .tif images, resizes them, and normalizes pixel values.
- **Date Extraction**: Extracts dates from filenames to associate each image with a specific date.
- **Error Handling**: Skips files with invalid date formats and prints debug messages to track the loading process.

This approach is useful for preparing a dataset of time-series images for tasks like sequence prediction or time-based image analysis.

MODEL ARCHITECTURE:

```
def create_convlstm_model_v2(image_shape):
  model = Sequential([
    ConvLSTM2D(filters=64, kernel_size=(3, 3), padding='same', return_sequences=True,
input_shape=(None, *image_shape)),
    BatchNormalization(),
    ConvLSTM2D(filters=64, kernel_size=(3, 3), padding='same'),
    BatchNormalization(),
    Conv2D(filters=32, kernel_size=(3, 3), padding='same'),
    BatchNormalization(),
    Conv2D(filters=1, kernel_size=(3, 3), activation='sigmoid', padding='same')
  ])
```

```
optimizer = Adam(learning_rate=initial_learning_rate)

model.compile(optimizer=optimizer, loss='mean_squared_error')


# model.compile(optimizer='adam', loss='mean_squared_error')

return model
```



```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv_lstm2d_12 (ConvLSTM2D  (None, None, 64, 64, 64   150016
 )                           )

 batch_normalization_18 (Ba  (None, None, 64, 64, 64   256
 tchNormalization)           )

 conv_lstm2d_13 (ConvLSTM2D  (None, 64, 64, 64)        295168
 )

 batch_normalization_19 (Ba  (None, 64, 64, 64)        256
 tchNormalization)

 conv2d_12 (Conv2D)          (None, 64, 64, 32)        18464

 batch_normalization_20 (Ba  (None, 64, 64, 32)        128
 tchNormalization)

 conv2d_13 (Conv2D)          (None, 64, 64, 1)         289

=================================================================
Total params: 464577 (1.77 MB)
Trainable params: 464257 (1.77 MB)
Non-trainable params: 320 (1.25 KB)
_________________________________________________________________
```

Explanation of the ConvLSTM Model Architecture

The given code defines a function create_convlstm_model_v2 which creates and compiles a Convolutional Long Short-Term Memory (ConvLSTM) model using TensorFlow/Keras. This model is designed for tasks that involve spatiotemporal data, such as video prediction or sequence prediction with images. Here's a detailed breakdown of each part of the model:

1. **Model Initialization:**

- Sequential(): Initializes a sequential model, which allows you to stack layers sequentially.

2. **First ConvLSTM2D Layer:**
   - ConvLSTM2D(filters=64, kernel_size=(3, 3), padding='same', return_sequences=True, input_shape=(None, *image_shape))
     - **filters=64**: The layer has 64 output filters.
     - **kernel_size=(3, 3)**: The size of the convolution kernel.
     - **padding='same'**: Ensures the output has the same spatial dimensions as the input.
     - **return_sequences=True**: Returns the full sequence of outputs for each input sequence.
     - **input_shape=(None, *image_shape)**: The input shape includes an unspecified number of time steps (None) and the shape of each image in the sequence (*image_shape).
3. **First BatchNormalization Layer:**
   - BatchNormalization(): Normalizes the outputs of the previous layer, speeding up training and improving stability.
4. **Second ConvLSTM2D Layer:**
   - ConvLSTM2D(filters=64, kernel_size=(3, 3), padding='same')
     - Similar to the first ConvLSTM2D layer but without return_sequences=True, meaning it only returns the output of the last time step.
5. **Second BatchNormalization Layer:**
   - BatchNormalization(): Again, normalizes the outputs of the previous layer.
6. **First Conv2D Layer:**
   - Conv2D(filters=32, kernel_size=(3, 3), padding='same')
     - **filters=32**: The layer has 32 output filters.
     - **kernel_size=(3, 3)**: The size of the convolution kernel.
     - **padding='same'**: Ensures the output has the same spatial dimensions as the input.
7. **Third BatchNormalization Layer:**
   - BatchNormalization(): Normalizes the outputs of the previous layer.
8. **Second Conv2D Layer:**
   - Conv2D(filters=1, kernel_size=(3, 3), activation='sigmoid', padding='same')

        - **filters=1**: The layer has a single output filter, suitable for a binary or grayscale output image.
        - **kernel_size=(3, 3)**: The size of the convolution kernel.
        - **activation='sigmoid'**: The sigmoid activation function, which maps output values to the range [0, 1].
        - **padding='same'**: Ensures the output has the same spatial dimensions as the input.

9. **Model Compilation:**
    - optimizer = Adam(learning_rate=initial_learning_rate): Initializes the Adam optimizer with a specified learning rate.
    - model.compile(optimizer=optimizer, loss='mean_squared_error'): Compiles the model using the Adam optimizer and mean squared error (MSE) loss function, suitable for regression tasks.

Summary

The model consists of the following layers:

1. ConvLSTM2D (with 64 filters and returning sequences)
2. BatchNormalization
3. ConvLSTM2D (with 64 filters)
4. BatchNormalization
5. Conv2D (with 32 filters)
6. BatchNormalization
7. Conv2D (with 1 filter and sigmoid activation)

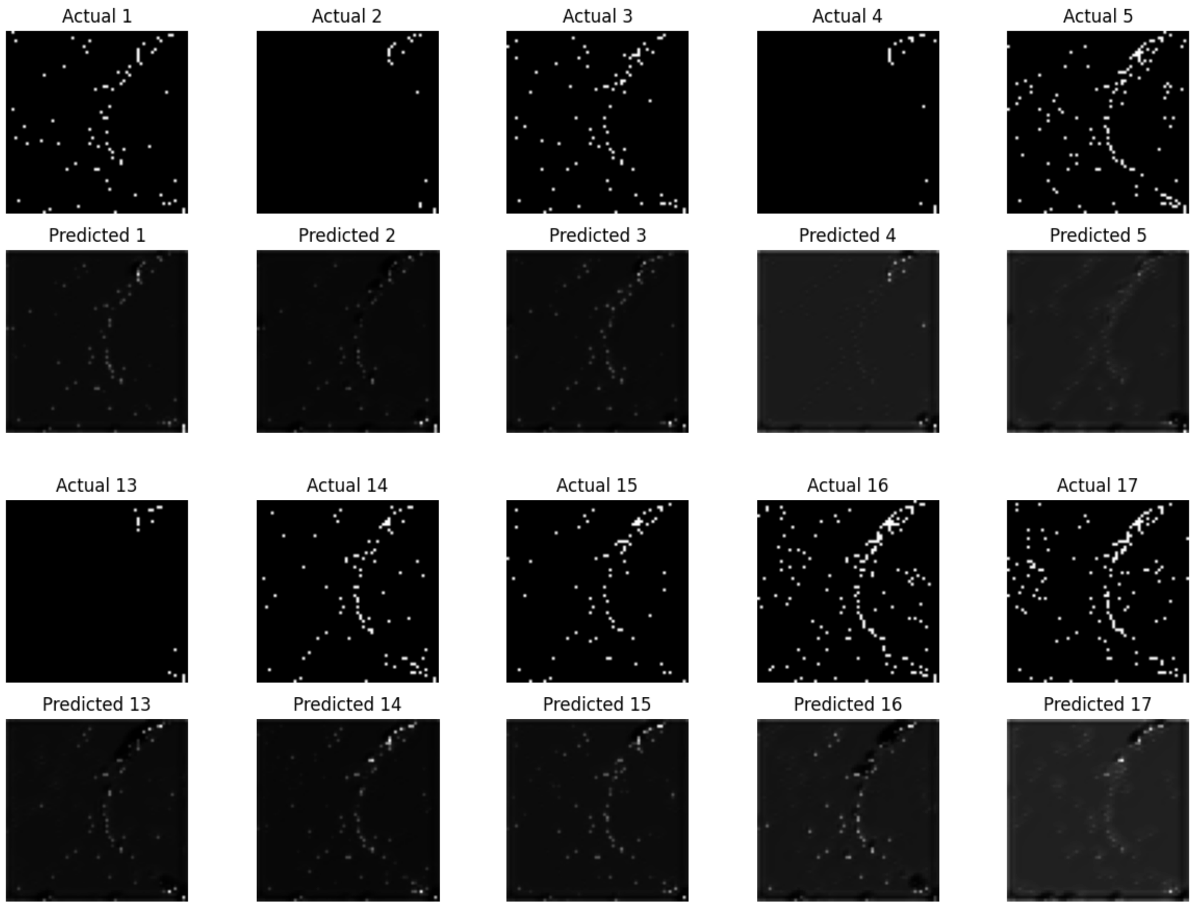
This architecture is designed to capture both spatial and temporal dependencies in the input image sequences. The use of ConvLSTM layers allows the model to process sequences of images, making it suitable for tasks such as video prediction or any other spatiotemporal data analysis. The final Conv2D layer with a sigmoid activation function outputs a single-channel image with pixel values in the range [0, 1]

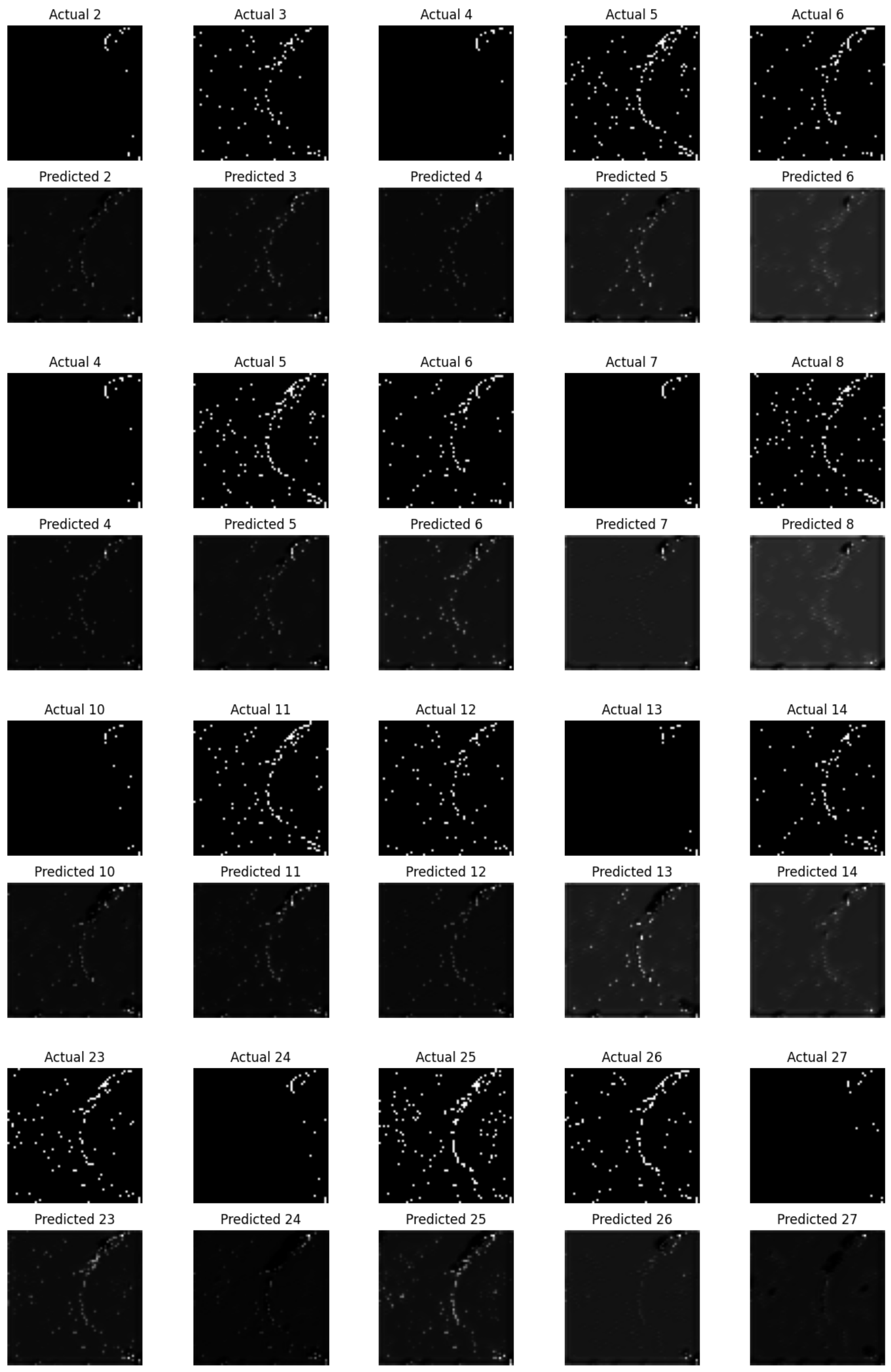
SEQUENCE PREDICITONS:

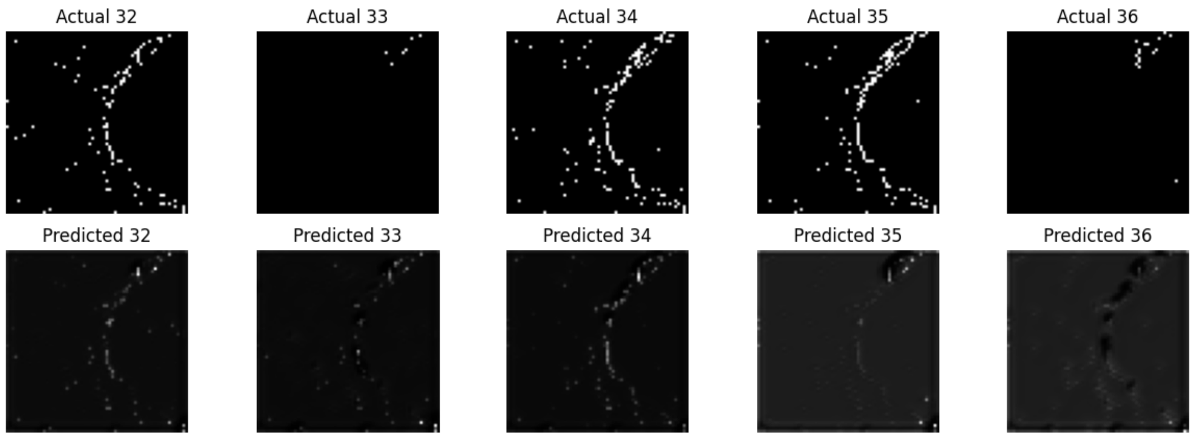
PSEUDODCODE:

```
def prepare_sequences(images, sequence_length):
  sequences = []
  targets = []

  for i in range(len(images) - sequence_length):
    sequence = images[i:i + sequence_length]
    target = images[i + sequence_length]
    sequences.append(sequence)
    targets.append(target)

  return np.array(sequences), np.array(targets)

sequence_length = 5  # Number of past days to use for prediction
X, y = prepare_sequences(images, sequence_length)

# Check the shapes of prepared sequences
print(f"X shape: {X.shape}")
print(f"y shape: {y.shape}")

# Ensure sequences arrays are not empty
if X.size == 0 or y.size == 0:
  raise ValueError("No sequences found. Check the sequence preparation logic.")
```

PREDICTIONS:




Actual 2
Actual 3
Actual 4
Actual 5
Actual 6
Predicted 2
Predicted 3
Predicted 4
Predicted 5
Predicted 6
Actual 4
Actual 5
Actual 6
Actual 7
Actual 8
Predicted 4
Predicted 5
Predicted 6
Predicted 7
Predicted 8
Actual 10
Actual 11
Actual 12
Actual 13
Actual 14
Predicted 10
Predicted 11
Predicted 12
Predicted 13
Predicted 14
Actual 23
Actual 24
Actual 25
Actual 26
Actual 27
Predicted 23
Predicted 24
Predicted 25
Predicted 26
Predicted 27



EVALUATION METRICS:

BATCH SIZE = 32:

```
1/1 [==============================] - 0s 26ms/step

MSE: 0.0231

RMSE: 0.1520

MAE: 0.0381

R^2: 0.0203
```

BATCH SIZE = 64:

```
1/1 [==============================] - 0s 24ms/step

MSE: 0.0216

RMSE: 0.1468

MAE: 0.0375

R^2: 0.0852
```

BATCH SIZE = 128:

```
1/1 [==============================] - 0s 25ms/step

MSE: 0.0201

RMSE: 0.1417
```

MAE: 0.0407

R^2: 0.1477

CNN + TRANSFORMER IMAGE PREDICTION AND GENERATION:

DATASET LOADING, PRE-PROCESSING:

PSEUDOCODE:

```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Parameters
img_height, img_width = 256, 256  # Adjust based on your image size
sequence_length = 10  # Length of sequences for the Transformer
data_dir = '/content/drive/MyDrive/ConvLSTM full'

# Load and preprocess images
def load_images(data_dir):
    images = []
    labels = []
    for year in sorted(os.listdir(data_dir)):
      if year == '2023':
        year_path = os.path.join(data_dir, year)
        if os.path.isdir(year_path):
```

```
        for img_file in sorted(os.listdir(year_path)):
          if img_file.endswith('.tif'):
            img_path = os.path.join(year_path, img_file)
            img = load_img(img_path, target_size=(img_height, img_width), color_mode='grayscale')
            img = img_to_array(img) / 255.0
            images.append(img)
            labels.append(img)  # Assuming the label is the same image for prediction
    return np.array(images), np.array(labels)

images, labels = load_images(data_dir)

# Check the number of loaded images
print(f"Number of images loaded: {len(images)}")

# Create sequences
def create_sequences(images, sequence_length):
    sequences = []
    targets = []
    for i in range(len(images) - sequence_length):
        sequences.append(images[i:i + sequence_length])
        targets.append(images[i + sequence_length])
    return np.array(sequences), np.array(targets)

sequences, targets = create_sequences(images, sequence_length)

# Check the number of created sequences
print(f"Number of sequences created: {len(sequences)}")

# Split data
if len(sequences) > 0:
    train_sequences, test_sequences, train_targets, test_targets = train_test_split(sequences, targets, test_size=0.2, random_state=42)

    train_sequences, val_sequences, train_targets, val_targets = train_test_split(train_sequences, train_targets, test_size=0.2, random_state=42)


    # Check the sizes of the splits

    print(f"Train sequences: {len(train_sequences)}, Validation sequences: {len(val_sequences)}, Test sequences: {len(test_sequences)}")

else:

    print("No sequences created. Please check the data loading and sequence creation steps.")
```

Explanation:

1. **Parameters:**
    - img_height and img_width are set to 256, defining the target size for the images.
    - sequence_length is set to 10, indicating the number of images in each sequence.
    - data_dir specifies the directory where the images are stored.
2. **Load and Preprocess Images:**
    - **Function load_images(data_dir):**
        - Iterates through the files in the specified directory.
        - For each .tif image file in the year 2023, loads the image, resizes it to the target size, converts it to a grayscale array, normalizes it to [0, 1], and appends it to the images list.
        - Assumes the labels are the same as the images for prediction purposes.
    - **Loading images:**
        - Calls the load_images function to load the images and labels.
        - Prints the number of images loaded.
3. **Create Sequences:**
    - **Function create_sequences(images, sequence_length):**
        - Generates sequences of images and their corresponding target images.

        - Each sequence consists of sequence_length consecutive images, and the target is the next image in the sequence.
    - **Creating sequences:**
        - Calls the create_sequences function to generate sequences and targets.
        - Prints the number of sequences created.

4. **Split Data:**
    - Splits the sequences and targets into training, validation, and test sets using train_test_split.
    - Splits the data in two stages:
        - First, into training and test sets (80% training, 20% test).
        - Then, the training set is further split into training and validation sets (80% training, 20% validation).
    - Prints the sizes of the resulting splits.

Summary:

This script loads and preprocesses grayscale .tif images from the year 2023 in a specified directory, creates sequences of images for training a model, and splits the data into training, validation, and test sets.

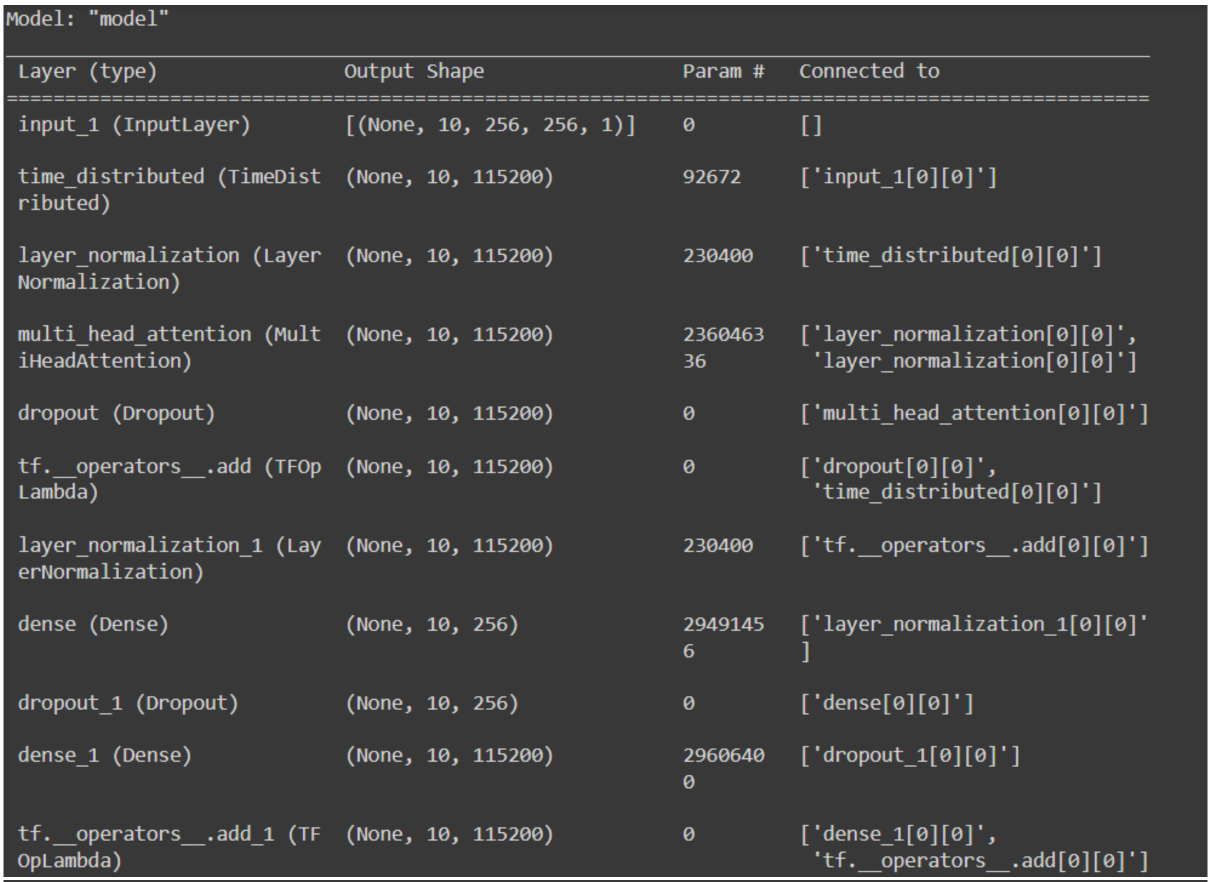
- **Image Loading and Preprocessing:** Images are loaded from the specified directory, resized to 256x256, converted to grayscale, and normalized.
- **Sequence Creation:** Sequences of 10 consecutive images are created, with each sequence's target being the next image.
- **Data Splitting:** The sequences are split into training, validation, and test sets to prepare for model training.

The script provides the necessary data preprocessing steps to train a model that predicts the next image in a sequence, which is useful for tasks like video frame prediction.

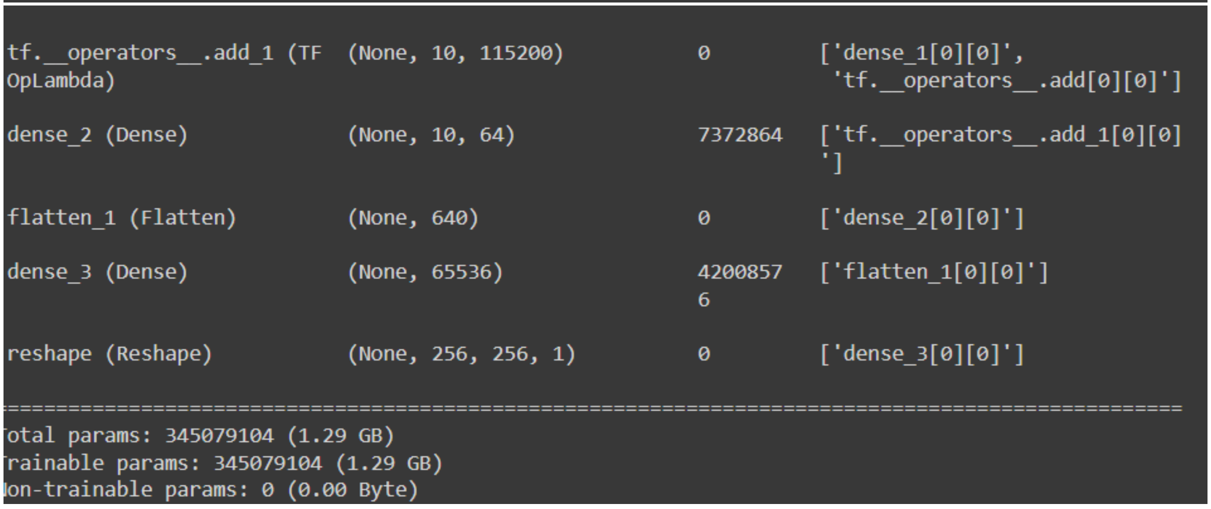
MODEL ARCHITECTURE:

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Input, TimeDistributed, Reshape, UpSampling2D, LayerNormalization, MultiHeadAttention, Dropout

from tensorflow.keras.models import Model

import tensorflow_addons as tfa
```

```
# Custom Transformer Block
def transformer_block(inputs, head_size, num_heads, ff_dim, dropout=0):
    x = LayerNormalization(epsilon=1e-6)(inputs)
    x = MultiHeadAttention(key_dim=head_size, num_heads=num_heads, dropout=dropout)(x, x)
    x = Dropout(dropout)(x)
    res = x + inputs

    x = LayerNormalization(epsilon=1e-6)(res)
    x = Dense(ff_dim, activation='relu')(x)
    x = Dropout(dropout)(x)
    x = Dense(inputs.shape[-1])(x)
    return x + res

# CNN Model to extract features from images
def create_cnn(input_shape):
    model = tf.keras.models.Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', input_shape=input_shape))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    return model

# Transformer Model
def create_transformer_model(input_shape):
    cnn = create_cnn(input_shape)

    input_seq = Input(shape=(sequence_length, *input_shape))
    x = TimeDistributed(cnn)(input_seq)

    x = transformer_block(x, head_size=128, num_heads=4, ff_dim=256, dropout=0.1)

    x = Dense(64, activation='relu')(x)
    x = Flatten()(x)  # Flatten before reshaping
    x = Dense(input_shape[0] * input_shape[1] * input_shape[2], activation='sigmoid')(x)
    output = Reshape((input_shape[0], input_shape[1], input_shape[2]))(x)

    model = Model(inputs=input_seq, outputs=output)
    return model

input_shape = (img_height, img_width, 1)  # Example input shape
model = create_transformer_model(input_shape)
model.compile(optimizer='adam', loss='mse', metrics=[tf.keras.metrics.MeanAbsoluteError(),
tf.keras.metrics.RootMeanSquaredError(), tfa.metrics.RSquare()])

model.summary()
```



```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, 10, 256, 256, 1)]    0         []

 time_distributed (TimeDist  (None, 10, 115200)           92672     ['input_1[0][0]']
 ributed)

 layer_normalization (Layer  (None, 10, 115200)           230400    ['time_distributed[0][0]']
 Normalization)

 multi_head_attention (Mult  (None, 10, 115200)           2360463   ['layer_normalization[0][0]',
 iHeadAttention)                                          36         'layer_normalization[0][0]']

 dropout (Dropout)           (None, 10, 115200)           0         ['multi_head_attention[0][0]']

 tf.__operators__.add (TFOp  (None, 10, 115200)           0         ['dropout[0][0]',
 Lambda)                                                             'time_distributed[0][0]']

 layer_normalization_1 (Lay  (None, 10, 115200)           230400    ['tf.__operators__.add[0][0]']
 erNormalization)

 dense (Dense)               (None, 10, 256)              2949145   ['layer_normalization_1[0][0]'
                                                          6         ]

 dropout_1 (Dropout)         (None, 10, 256)              0         ['dense[0][0]']

 dense_1 (Dense)             (None, 10, 115200)           2960640   ['dropout_1[0][0]']
                                                          0

 tf.__operators__.add_1 (TF  (None, 10, 115200)           0         ['dense_1[0][0]',
 OpLambda)                                                           'tf.__operators__.add[0][0]']
```



```
tf.__operators__.add_1 (TF  (None, 10, 115200)           0         ['dense_1[0][0]',
OpLambda)                                                           'tf.__operators__.add[0][0]']

dense_2 (Dense)             (None, 10, 64)               7372864   ['tf.__operators__.add_1[0][0
                                                                   ']

flatten_1 (Flatten)         (None, 640)                  0         ['dense_2[0][0]']

dense_3 (Dense)             (None, 65536)                4200857   ['flatten_1[0][0]']
                                                         6

reshape (Reshape)           (None, 256, 256, 1)          0         ['dense_3[0][0]']

==================================================================================================
otal params: 345079104 (1.29 GB)
rainable params: 345079104 (1.29 GB)
on-trainable params: 0 (0.00 Byte)
```

Explanation:

1. **Custom Transformer Block:**
   - transformer_block(inputs, head_size, num_heads, ff_dim, dropout=0):
     - **Layer Normalization:** Normalizes the inputs.
     - **Multi-Head Attention:** Applies multi-head attention with specified head_size and num_heads.
     - **Dropout:** Applies dropout for regularization.

        - **Residual Connection:** Adds the input to the output (residual connection).
        - **Feed-Forward Network:** Applies a two-layer feed-forward network with ff_dim units in the hidden layer.
        - **Second Residual Connection:** Adds the intermediate result to the feed-forward network's output.
    - The block combines multi-head self-attention and a feed-forward network, which are essential components of the Transformer architecture.
2. **CNN Model to Extract Features:**
    - create_cnn(input_shape):
        - **Conv2D and MaxPooling Layers:**
            - Three convolutional layers with increasing filter sizes (32, 64, 128), each followed by max pooling.
            - **Flatten:** Flattens the output for the next layers.
        - This CNN is used to extract features from each frame in the input sequence.
3. **Transformer Model:**
    - create_transformer_model(input_shape):
        - **Input Sequence:** Takes a sequence of images as input with shape (sequence_length, *input_shape).
        - **TimeDistributed Layer:** Applies the CNN to each image in the sequence.
        - **Transformer Block:** Applies the custom Transformer block to the extracted features.
        - **Dense Layers:** Further processes the features.
        - **Flatten and Reshape:** Flattens the features and then reshapes them to the original image dimensions.
        - **Output:** Outputs the predicted image with the same shape as the input images.
    - This model integrates CNN and Transformer components to process sequences of images and predict the next image.
4. **Model Compilation and Summary:**
    - **Compile:** Uses the Adam optimizer and mean squared error (MSE) loss function. It also tracks mean absolute error (MAE), root mean squared error (RMSE), and R-squared (R^2) metrics.
    - **Summary:** Prints the model summary to visualize the architecture and layers.

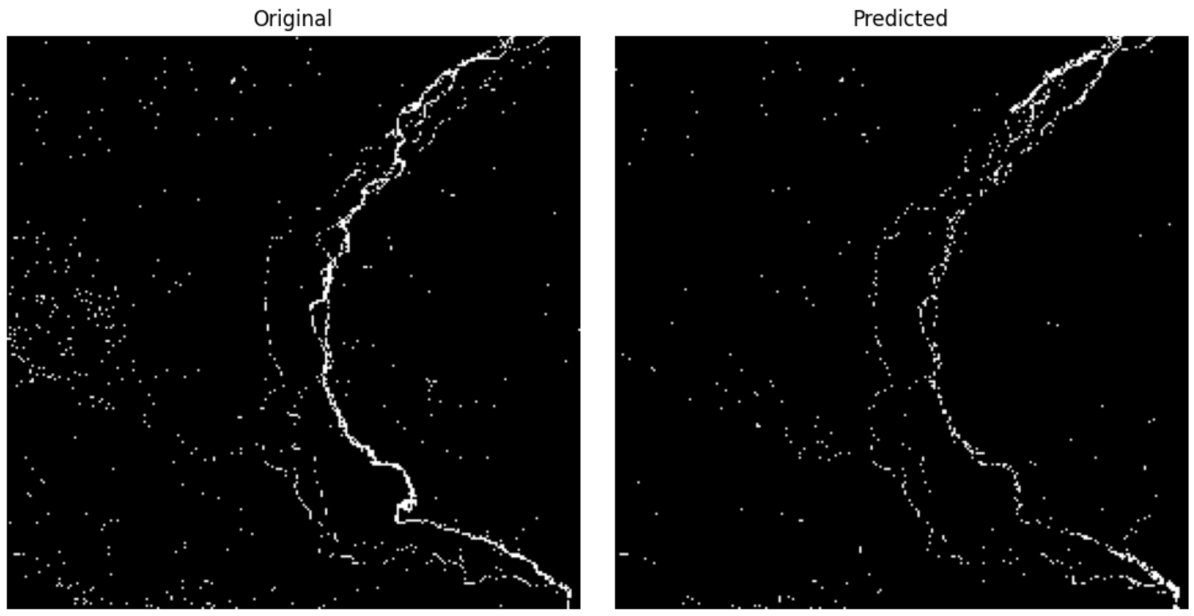
Summary:

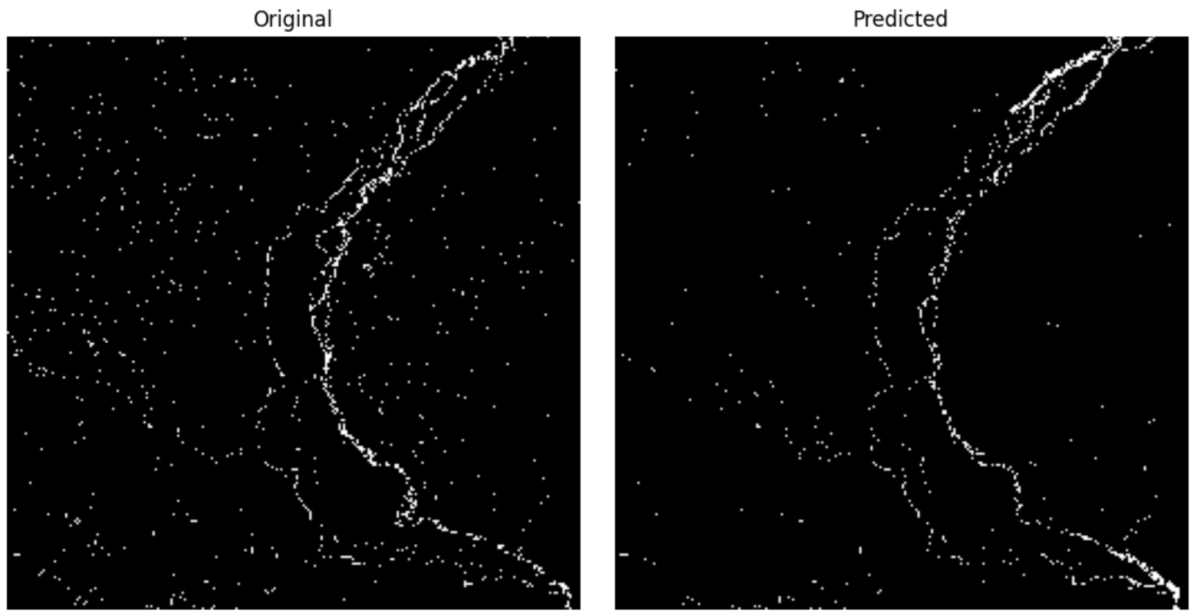
This script defines a deep learning model that combines Convolutional Neural Networks (CNN) and Transformer architectures to process sequences of images and predict the next image in the sequence.

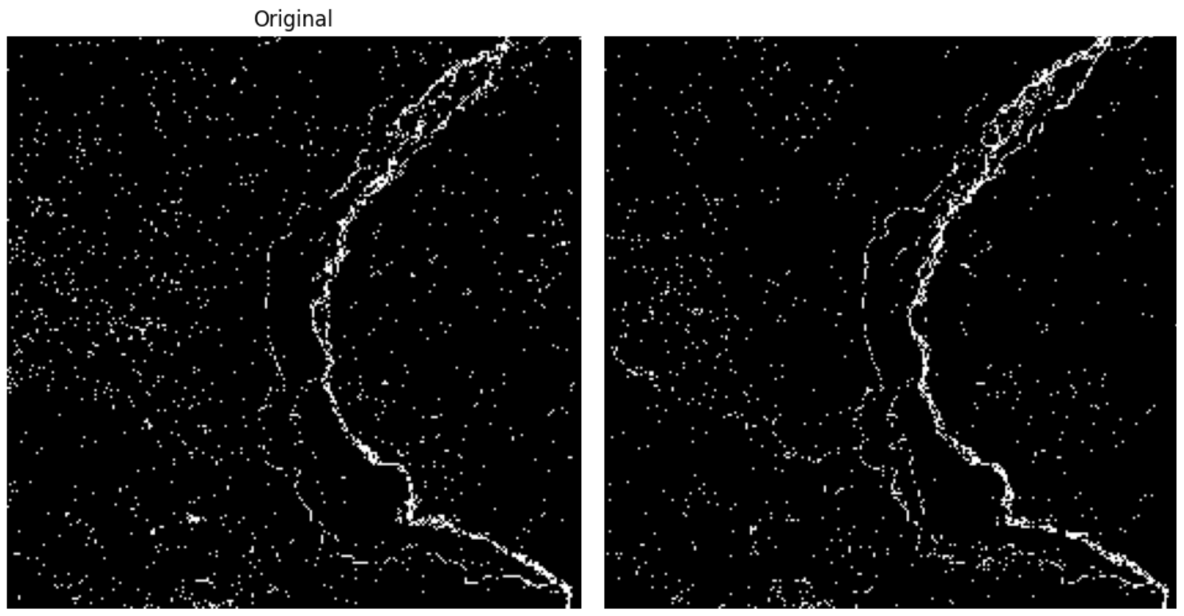
- **CNN Model:** Extracts features from each image in the sequence.
- **Transformer Block:** Applies self-attention mechanisms to capture dependencies between images in the sequence.
- **Overall Model:** Uses a TimeDistributed layer to apply the CNN to each frame, processes the sequence with Transformer blocks, and outputs the next predicted image.

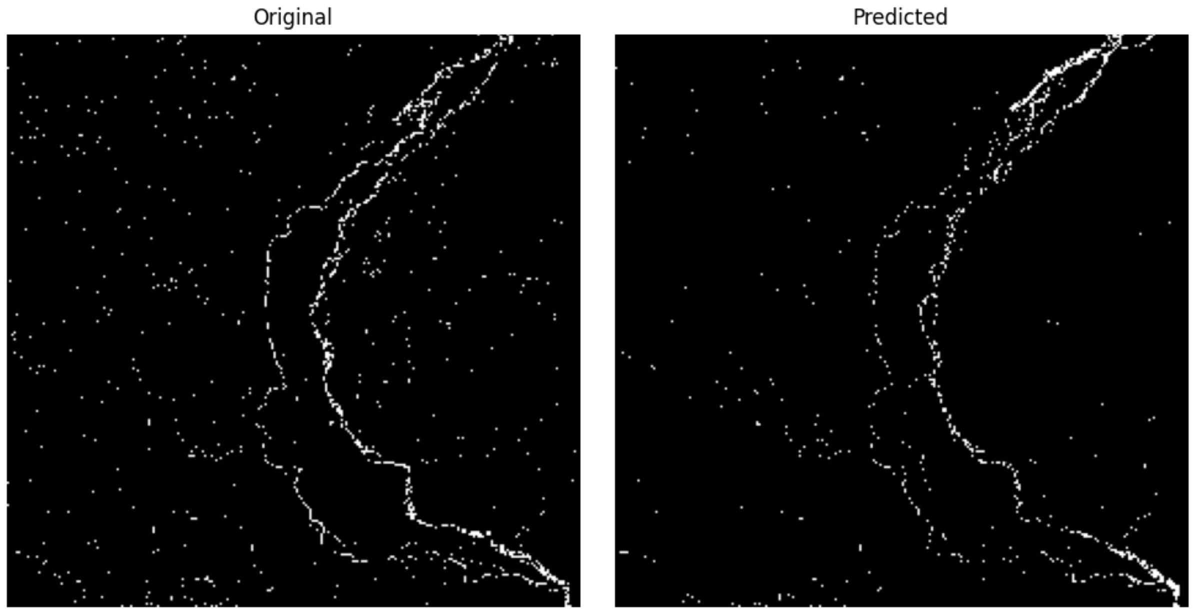
The model is compiled with the Adam optimizer and MSE loss, and it evaluates MAE, RMSE, and R^2 metrics. The summary of the model is printed to review its architecture and layers.

PREDICTIONS:

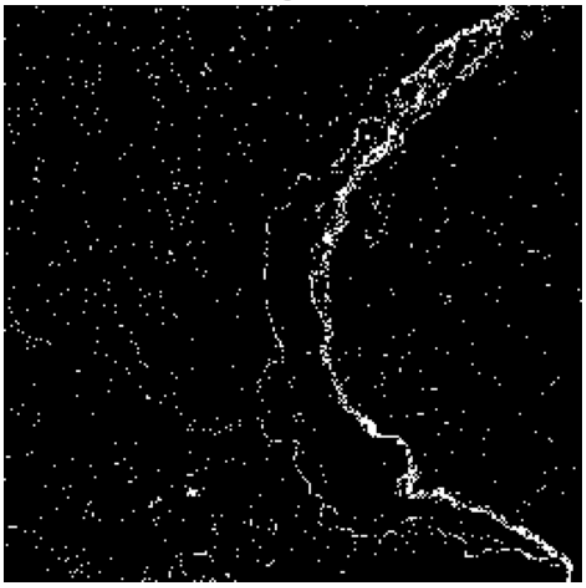


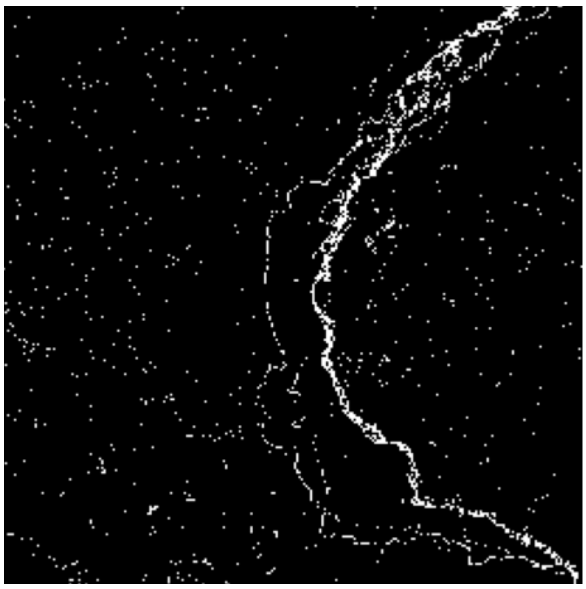



Original

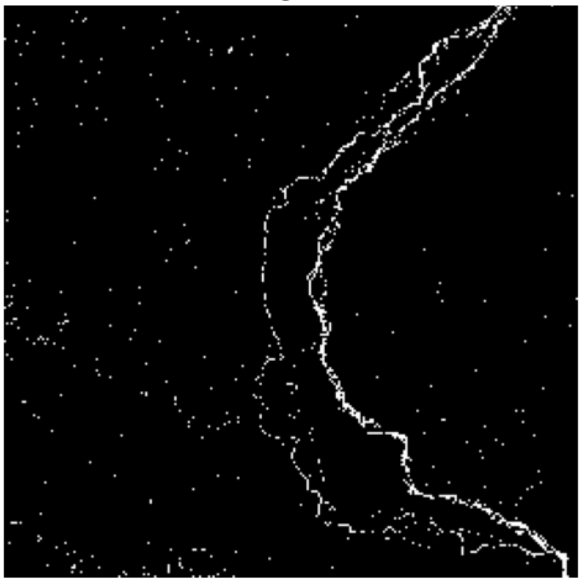

Original
Predicted

Original

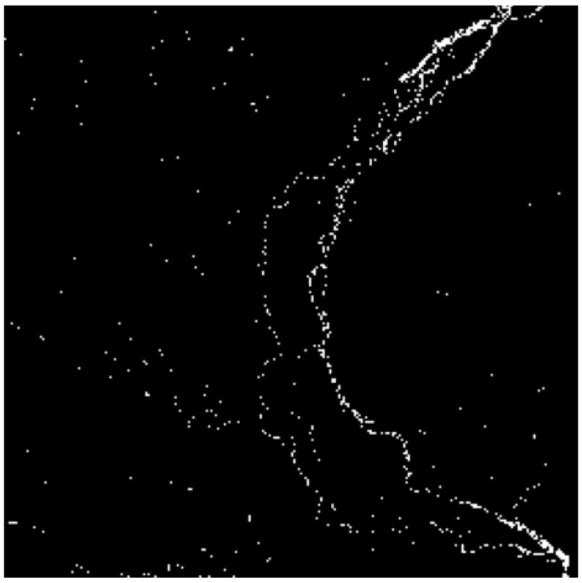




Original



Predicted



EVALUATION METRICS:

BATCH SIZE = 32

```
2/2 [==============================] - 1s 139ms/step
MSE: 0.027248090133070946, MAE: 0.027248090133070946, RMSE: 0.07256683707237244, R-
squared: 0.6780746056501165
```

BATCH SIZE = 16

```
2/2 [==============================] - 1s 109ms/step
MSE: 0.0190887451171875, MAE: 0.0190887451171875, RMSE: 0.03955882787704468, R-
squared: 0.8602270342504718
```

BATCH SIZE = 8

```
2/2 [==============================] - 1s 207ms/step
MSE: 0.030050525441765785, MAE: 0.030050525441765785, RMSE: 0.09400203824043274, R-
squared: 0.5236688263355385
```

COMBINED CNN + TRANSFORMER (SATELLITE IMAGES + CSV PARAMETERS)

CSV DATASET LOADING:

```
# Load and preprocess CSV data

csv_path = '/content/Kosi Rainfall + metrics daily (2014-2023).csv'

data = pd.read_csv(csv_path)
```

PRE-PROCESSING:

```
# Rename columns for consistency

data.rename(columns={'YEAR': 'year', 'MO': 'month', 'DY': 'day'}, inplace=True)


# Create a date column

data['date'] = pd.to_datetime(data[['year', 'month', 'day']])


# Set the date column as index

data.set_index('date', inplace=True)


# Drop unnecessary columns

data.drop(columns=['year', 'month', 'day', 'LAT', 'LON'], inplace=True)


# Fill missing values

data = data.fillna(method='ffill').fillna(method='bfill')


# Normalize the features

scaler = StandardScaler()

data_scaled = pd.DataFrame(scaler.fit_transform(data), index=data.index,
columns=data.columns)
```

DATA.HEAD()

PRECTOTCORR	**WS10M**	**RH2M**	**QV2M**	**T2M_RANGE**	**ALLSKY_SFC_UV_INDEX**	
date						
2014-01-01	0.00	3.70	60.50	7.20	15.83	0.81
2014-01-02	0.00	5.36	67.69	7.84	12.05	0.76
2014-01-03	0.00	3.61	59.84	5.28	14.14	0.58
2014-01-04	0.00	3.07	56.06	4.79	15.85	0.60
2014-01-05	0.01	2.98	56.59	5.58	16.07	0.49
...	...	...	...	...	...	...
2023-12-27	0.00	1.26	70.97	9.28	11.70	0.63
2023-12-28	0.00	1.58	69.50	8.88	11.58	0.61
2023-12-29	0.00	2.65	70.16	8.33	12.25	0.73
2023-12-30	0.00	3.03	67.84	7.45	13.55	0.37
2023-12-31	0.00	2.82	64.56	6.62	13.82	0.37

3652 rows × 6 columns

The provided code is designed to load and preprocess a CSV dataset containing daily rainfall and various metrics for the Kosi region from 2014 to 2023. Here is a detailed summary of each step:

1. Loading the CSV Dataset:

- **Code:**

```
csv_path = '/content/Kosi Rainfall + metrics daily (2014-2023).csv'

data = pd.read_csv(csv_path)
```

  - **Explanation:** The CSV file located at the specified path (csv_path) is read into a Pandas DataFrame called data using the pd.read_csv() function. This function loads the data from the CSV file into a structured DataFrame format for further processing.

2. Preprocessing Steps:

2.1 Renaming Columns:

```
data.rename(columns={'YEAR': 'year', 'MO': 'month', 'DY': 'day'}, inplace=True)
```

- **Explanation:** The columns 'YEAR', 'MO', and 'DY' are renamed to 'year', 'month', and 'day', respectively. This renaming is done for consistency and to simplify the column names for easier manipulation and readability.

2.2 Creating and Setting Date Column:

```
data['date'] = pd.to_datetime(data[['year', 'month', 'day']])

data.set_index('date', inplace=True)
```

- **Explanation:**
  - A new 'date' column is created by combining the 'year', 'month', and 'day' columns into a single datetime object using the pd.to_datetime() function. This new column represents the date of each observation.
  - The set_index() function then sets this 'date' column as the index of the DataFrame. Setting the date as the index is crucial for time series analysis as it allows for date-based operations and ensures the data is properly aligned chronologically.

2.3 Dropping Unnecessary Columns:

```
data.drop(columns=['year', 'month', 'day', 'LAT', 'LON'], inplace=True)
```

- **Explanation:**
  - After creating the 'date' column, the original 'year', 'month', and 'day' columns are no longer needed, so they are dropped from the DataFrame.

- The 'LAT' and 'LON' columns, representing latitude and longitude, are also dropped. This suggests that the subsequent analysis does not require geographical coordinates, focusing instead on temporal data.

2.4 Handling Missing Values:

```
data = data.fillna(method='ffill').fillna(method='bfill')
```

- **Explanation:**
  - The fillna() function is used to handle missing values in the DataFrame.
  - The first fillna(method='ffill') applies forward fill, which fills the missing values with the last known non-missing value. This propagates the last observed value forward to fill gaps.
  - The second fillna(method='bfill') applies backward fill, filling any remaining missing values with the next known non-missing value. This ensures there are no remaining gaps in the dataset, which is essential for consistent analysis and modeling.

2.5 Normalizing Features:

```
scaler = StandardScaler()

data_scaled = pd.DataFrame(scaler.fit_transform(data), index=data.index,
columns=data.columns)
```

- **Explanation:**
  - The features in the DataFrame are normalized using the StandardScaler() from scikit-learn. Normalization is a common preprocessing step that scales the features to have a mean of 0 and a standard deviation of 1.
  - The fit_transform() method standardizes the features based on the statistical properties of the dataset.
  - The scaled data is then converted back into a DataFrame (data_scaled) with the same index and column names as the original data. This ensures the structure and labeling of the data are preserved, making it easier to work with in subsequent analysis and modeling steps.

SATELLITE IMAGES LOADING AND PRE-PROCESSING:

PSEUDOCODE:

```
# 2. Load and preprocess image data
image_dir = '/content/drive/MyDrive/ConvLSTM full/2023'
image_files = [f for f in os.listdir(image_dir) if f.endswith('.tif')]

def load_image(image_path):
    image = load_img(image_path, target_size=(256, 256), color_mode='grayscale')
    image = img_to_array(image)
    image = image / 255.0  # Normalize
    return image

# Extract dates from filenames and load images
image_dates = []
images = []

for image_file in image_files:
    date_str = image_file.split('.')[0]  # Format: 'YYYY-MM-DD'
    date = pd.to_datetime(date_str, format='%Y-%m-%d')
    image_path = os.path.join(image_dir, image_file)
    image = load_image(image_path)

    image_dates.append(date)
    images.append(image)

# Create a DataFrame for image data
image_df = pd.DataFrame({'date': image_dates, 'image': images})
image_df.set_index('date', inplace=True)
```

```
# 3. Align CSV data with images
# Filter CSV data to include only dates present in image dates
aligned_data = data_scaled.loc[data_scaled.index.intersection(image_df.index)]

# Ensure image_df is aligned with the CSV data
aligned_images = image_df.loc[aligned_data.index]['image'].tolist()
aligned_images = np.array(aligned_images)
```

The provided code performs several steps to load, preprocess, and align image data with a pre-existing CSV dataset. Here's a detailed explanation of each step:

1. Loading and Preprocessing Image Data:

- **Image Directory and File List:**
  - The code specifies the directory containing the images and lists all files in this directory that have a .tif extension. This step is crucial for identifying the relevant image files to be processed.
- **Loading and Normalizing Images:**
  - A function load_image() is defined to load and preprocess each image. This function:
    - Loads the image from the given path, resizes it to 256x256 pixels, and converts it to grayscale.
    - Converts the image to a NumPy array and normalizes the pixel values to the range [0, 1] by dividing by 255. This normalization is important for consistent input to neural networks, which typically perform better with normalized data.
- **Extracting Dates from Filenames and Loading Images:**
  - The code iterates over the list of image files, extracting the date from each filename (which follows the 'YYYY-MM-DD' format) and converting it to a datetime object.
  - For each image file, it calls the load_image() function to load and preprocess the image.

        - The dates and corresponding images are stored in separate lists (image_dates and images).

2. Creating a DataFrame for Image Data:

- A Pandas DataFrame is created with the dates as the index and the images as the data. This DataFrame (image_df) allows for easy manipulation and alignment of image data with other datasets.

3. Aligning CSV Data with Images:

- **Filtering and Aligning Data:**
    - The code aligns the previously preprocessed and normalized CSV data (data_scaled) with the image data based on the dates.
    - It filters the CSV data to include only the dates that are present in the image data, ensuring both datasets cover the same time period.
    - This is done using the intersection of the indexes (dates) of both datasets.
- **Ensuring Consistency:**
    - The image DataFrame is further aligned with the filtered CSV data to ensure that the images correspond to the exact dates in the CSV data.
    - The aligned images are converted to a NumPy array for consistency and ease of use in subsequent analysis or model training.

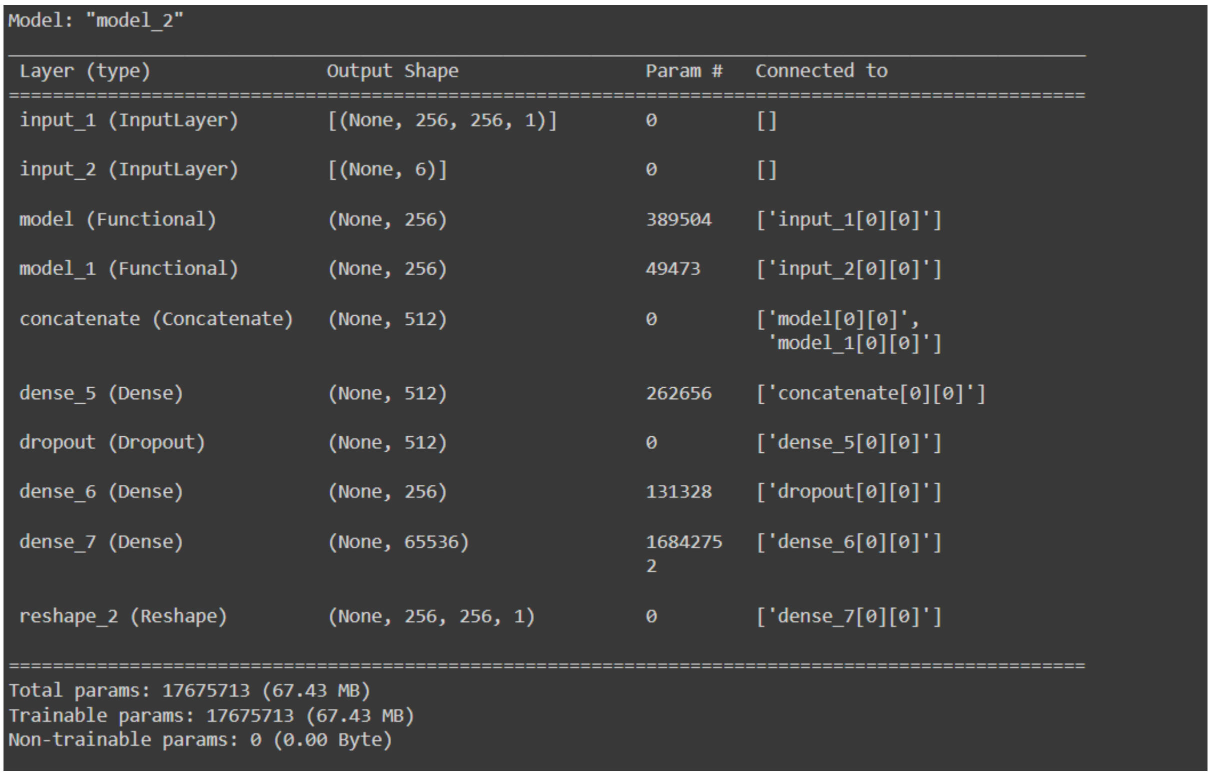
Summary

Overall, the code effectively loads, preprocesses, and aligns image data with an existing CSV dataset based on dates. This process includes loading and normalizing images, extracting dates from filenames, creating a DataFrame to manage image data, and ensuring both the image and CSV datasets are synchronized for further analysis or modeling tasks.

MODEL ARCHITECTURE:

```
def create_cnn_transformer_image_model(image_shape):
    inputs = Input(shape=image_shape)

    # CNN layers
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
    x = MaxPooling2D((2, 2))(x)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2))(x)
    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2))(x)

    # Flatten the output before passing to Transformer
    x = Reshape((x.shape[1] * x.shape[2], x.shape[3]))(x)  # Flattening (32*32, 128) for example
    x = MultiHeadAttention(num_heads=4, key_dim=128)(x, x)
    x = Dense(256, activation='relu')(x)
    x = GlobalAveragePooling1D()(x)

    model = Model(inputs=inputs, outputs=x)
    return model

def create_cnn_transformer_param_model(param_shape):
    inputs = Input(shape=param_shape)

    # Fully connected layers
    x = Dense(64, activation='relu')(inputs)
    x = Dense(128, activation='relu')(x)
    x = Dense(256, activation='relu')(x)

    # Transformer layers
    x = Reshape((256, 1))(x)  # Adjust shape based on your model
    x = MultiHeadAttention(num_heads=4, key_dim=256)(x, x)
    x = Dense(256, activation='relu')(x)
    x = GlobalAveragePooling1D()(x)

    model = Model(inputs=inputs, outputs=x)
    return model
```

```
def create_combined_model(image_shape, param_shape):
    image_input = Input(shape=image_shape)
    param_input = Input(shape=param_shape)

    # CNN+Transformer for Image
    image_model = create_cnn_transformer_image_model(image_shape)
    image_output = image_model(image_input)

    # CNN+Transformer for Parameters
    param_model = create_cnn_transformer_param_model(param_shape)
    param_output = param_model(param_input)

    # Combine Outputs
    combined = Concatenate()([image_output, param_output])
    combined = Dense(512, activation='relu')(combined)
    combined = Dropout(0.5)(combined)
    combined = Dense(256, activation='relu')(combined)
    combined_output = Dense(256*256, activation='sigmoid')(combined)
    combined_output = Reshape((256, 256, 1))(combined_output)

    model = Model(inputs=[image_input, param_input], outputs=combined_output)
    model.compile(optimizer='adam', loss='mean_squared_error')
    return model

# Create and train the combined model
combined_model = create_combined_model((256, 256, 1), (6,))
combined_model.summary()
```



```
Model: "model_2"
__________________________________________________________________________________________________
 Layer (type)                Output Shape                 Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, 256, 256, 1)]        0         []

 input_2 (InputLayer)        [(None, 6)]                  0         []

 model (Functional)          (None, 256)                  389504    ['input_1[0][0]']

 model_1 (Functional)        (None, 256)                  49473     ['input_2[0][0]']

 concatenate (Concatenate)   (None, 512)                  0         ['model[0][0]',
                                                                     'model_1[0][0]']

 dense_5 (Dense)             (None, 512)                  262656    ['concatenate[0][0]']

 dropout (Dropout)           (None, 512)                  0         ['dense_5[0][0]']

 dense_6 (Dense)             (None, 256)                  131328    ['dropout[0][0]']

 dense_7 (Dense)             (None, 65536)                1684275   ['dense_6[0][0]']
                                                          2

 reshape_2 (Reshape)         (None, 256, 256, 1)          0         ['dense_7[0][0]']

==================================================================================================
Total params: 17675713 (67.43 MB)
Trainable params: 17675713 (67.43 MB)
Non-trainable params: 0 (0.00 Byte)
```

1. CNN-Transformer Model for Images

create_cnn_transformer_image_model(image_shape)

This function creates a model that processes image data using a combination of Convolutional Neural Networks (CNNs) and Transformers. Here's the breakdown:

- **Input Layer:**
  - Input(shape=image_shape) defines the input shape of the image data. For example, (256, 256, 1) indicates grayscale images of size 256x256 pixels.
- **CNN Layers:**
  - **Conv2D Layers:**
    - Conv2D(32, (3, 3), activation='relu', padding='same') applies a 2D convolutional layer with 32 filters, a kernel size of 3x3, ReLU activation, and same padding (output size same as input size).
    - MaxPooling2D((2, 2)) performs max pooling with a pool size of 2x2, reducing the spatial dimensions by half.

        - This pattern of Conv2D followed by MaxPooling2D is repeated with increasing filter sizes (64 and 128) to capture more complex features at different levels of abstraction.

- **Flattening and Reshaping:**
    - Reshape((x.shape[1] * x.shape[2], x.shape[3])) reshapes the output from the CNN layers into a sequence format suitable for the Transformer. The shape (32*32, 128) assumes the feature map is 32x32 with 128 channels.
- **Transformer Layers:**
    - MultiHeadAttention(num_heads=4, key_dim=128)(x, x) applies multi-head self-attention to capture long-range dependencies in the feature maps.
    - Dense(256, activation='relu') applies a dense layer to transform the output of the attention layer.
    - GlobalAveragePooling1D() aggregates the feature map across the sequence dimension, reducing it to a single vector.
- **Model Creation:**
    - The Model class combines the input layer and the final output layer to create the CNN-Transformer model.

2. CNN-Transformer Model for Parameters

create_cnn_transformer_param_model(param_shape)

This function processes non-image parameter data using a combination of dense layers and Transformer layers. Here's the breakdown:

- **Input Layer:**
    - Input(shape=param_shape) defines the input shape for parameter data. For example, (6,) could be 6 different parameters.
- **Fully Connected Layers:**
    - Dense(64, activation='relu'), Dense(128, activation='relu'), and Dense(256, activation='relu') apply a series of dense layers with ReLU activation to transform the input data.
- **Transformer Layers:**
    - Reshape((256, 1)) reshapes the output to a sequence format for the Transformer layer. The shape (256, 1) assumes 256 timesteps with a single feature per timestep.
    - MultiHeadAttention(num_heads=4, key_dim=256)(x, x) applies multi-head self-attention.
    - Dense(256, activation='relu') applies a dense layer after the attention layer.
    - GlobalAveragePooling1D() aggregates the feature map across the sequence dimension.

- **Model Creation:**
    - The Model class combines the input layer and the final output layer to create the parameter processing model.

3. Combined Model

create_combined_model(image_shape, param_shape)

This function combines the outputs of the image and parameter models into a single model:

- **Inputs:**
    - Input(shape=image_shape) for image data and Input(shape=param_shape) for parameter data.
- **CNN+Transformer for Image:**
    - image_model processes the image input using the create_cnn_transformer_image_model function.
- **CNN+Transformer for Parameters:**
    - param_model processes the parameter input using the create_cnn_transformer_param_model function.
- **Combine Outputs:**
    - Concatenate()([image_output, param_output]) combines the outputs from both models.
    - Dense(512, activation='relu') and Dense(256, activation='relu') apply dense layers to the combined output.
    - Dropout(0.5) applies dropout to reduce overfitting.
    - Dense(256*256, activation='sigmoid') outputs a flattened image representation.
    - Reshape((256, 256, 1)) reshapes the output back to the original image dimensions.
- **Model Creation:**
    - The Model class combines the input layers and the final output layer to create the combined model.

Summary

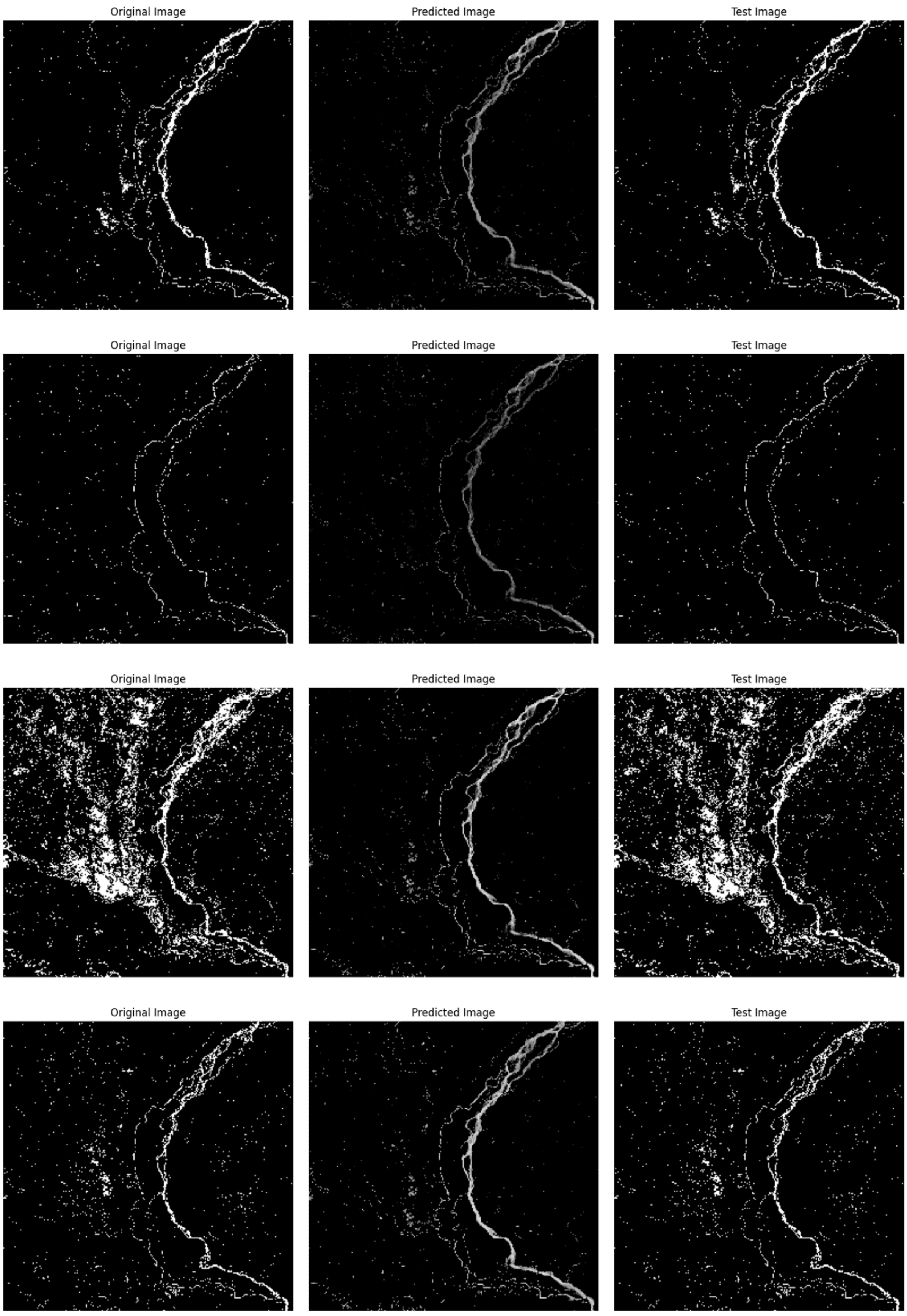
- **Image Model:** Uses CNNs for feature extraction and Transformers for capturing relationships in the feature maps.
- **Parameter Model:** Uses dense layers for feature extraction and Transformers for sequence-based processing.
- **Combined Model:** Merges outputs from both models, applies additional dense layers and dropout, and reshapes the result into an image format.

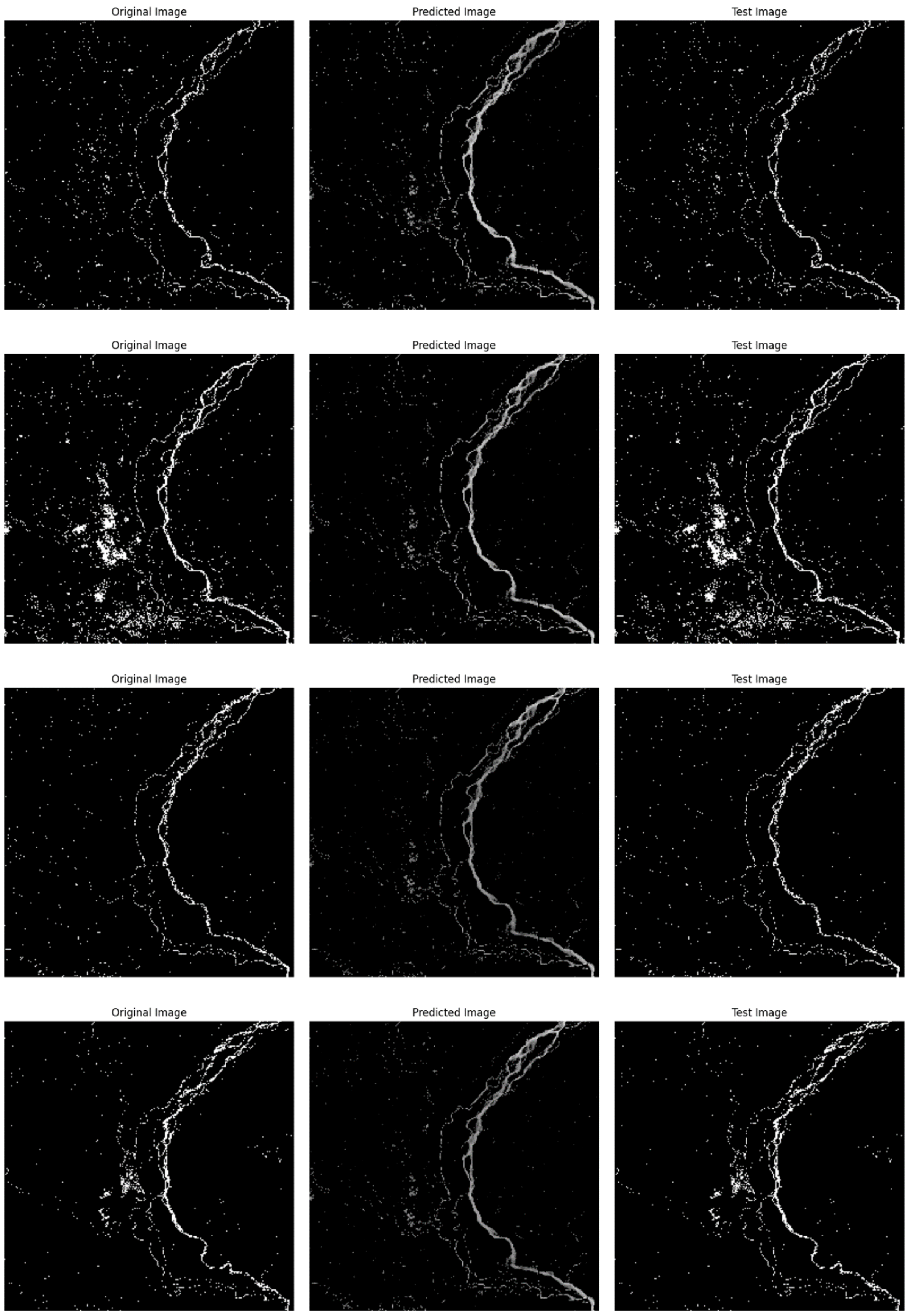
This architecture allows the model to leverage both spatial features from images and additional parameter data for comprehensive predictions.

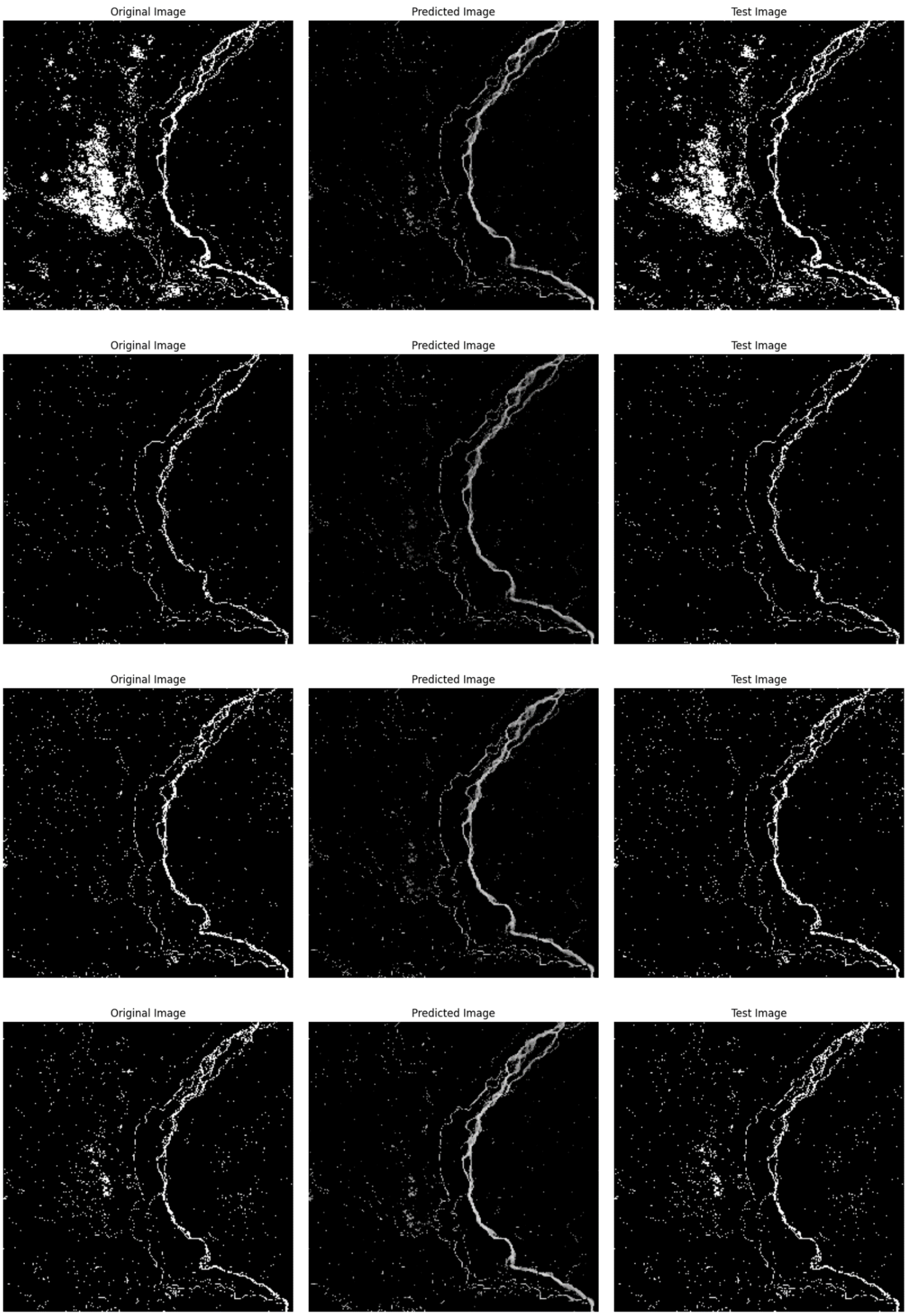
PRE-PROCESSING COMBINED DATASET:

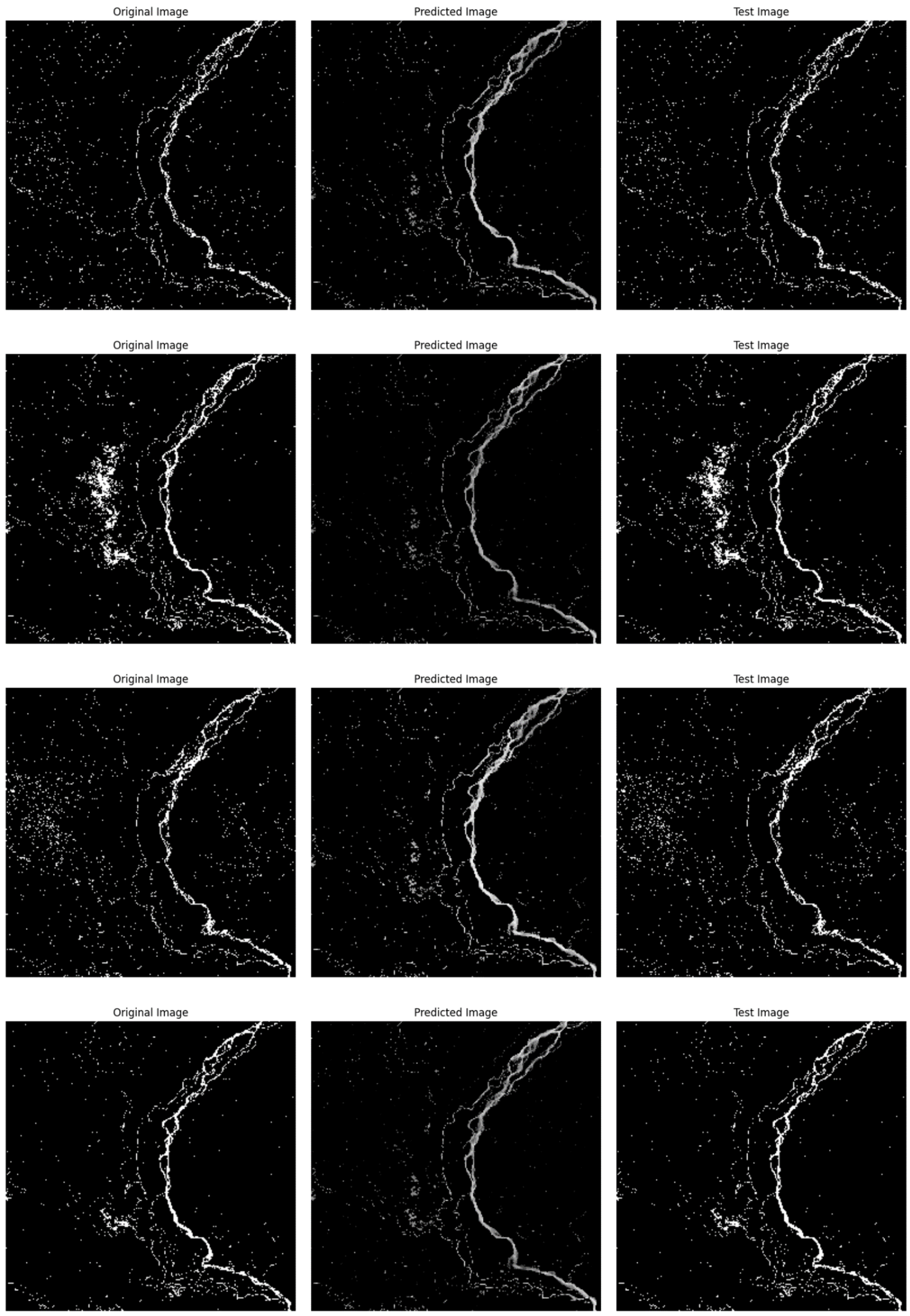
```
from sklearn.model_selection import train_test_split

# Prepare training data
image_shape = (256, 256, 1)
param_shape = len(data_scaled.columns)
X_images = aligned_images
X_params = aligned_data.values
y = aligned_images  # Assuming you are predicting the images

# Split the data
X_train_img, X_test_img, X_train_params, X_test_params, y_train, y_test =
train_test_split(X_images, X_params, y, test_size=0.2, random_state=42)
```

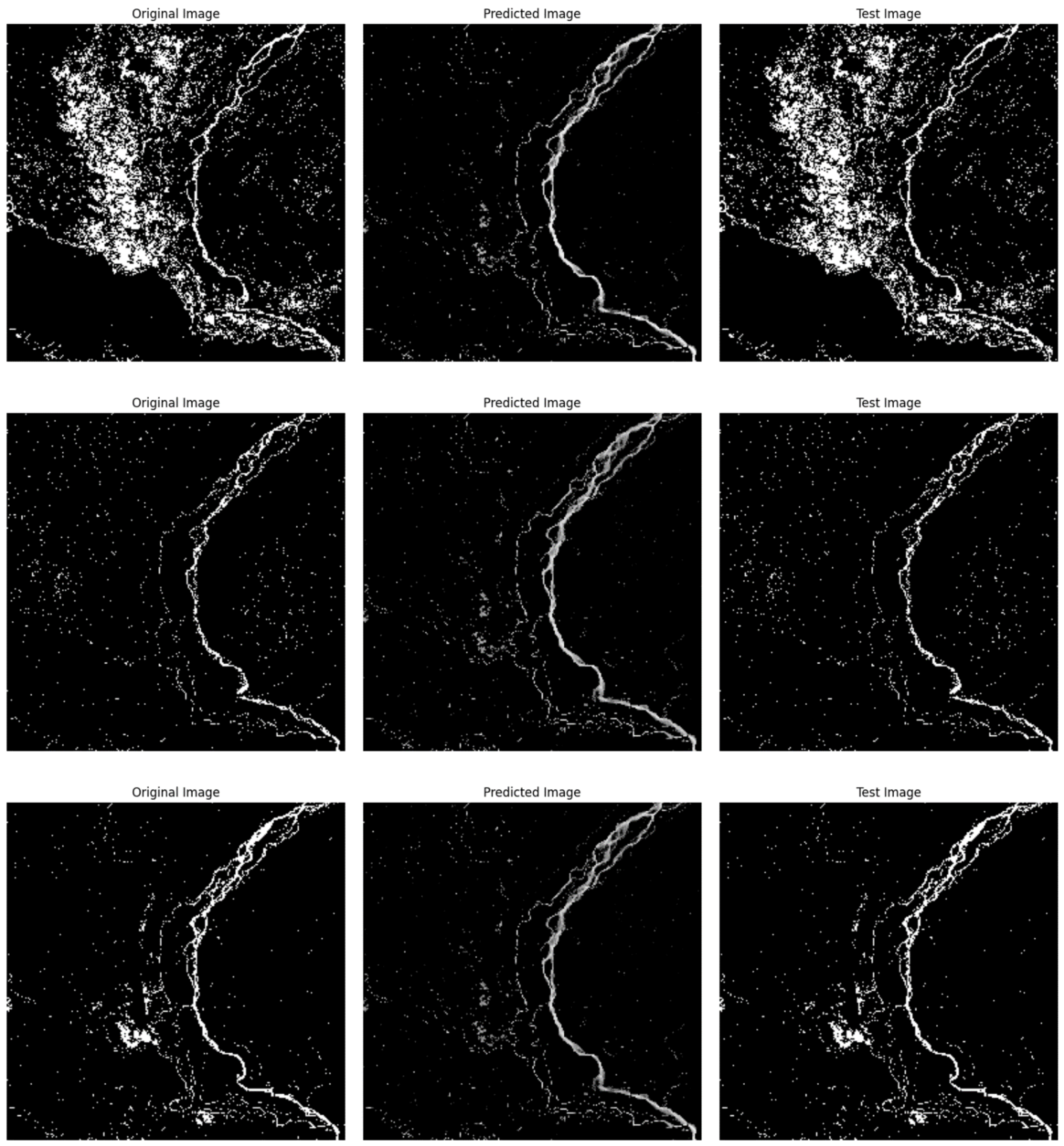
PREDICTIONS:


Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image


Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image


Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image


Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image


Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image
Original Image
Predicted Image
Test Image

PARAMETER PREDICTIONS:

Parameter Predictions for the Test Set:

```
[[[[2.7745140e-05]
  [1.4068257e-05]
  [3.2726755e-05]
  ...
  [3.0458801e-05]
  [2.8348173e-05]
  [2.0270165e-05]]

 [[4.0775925e-05]
  [2.8679888e-05]
  [1.6406328e-05]
  ...
  [1.5945498e-05]
  [2.5873625e-05]
  [3.9134236e-05]]

 [[1.0543097e-04]
  [2.0146210e-05]
  [4.5180757e-05]
  ...
  [2.0018355e-05]
  [2.8298826e-05]
  [4.5966415e-05]]

 ...
```

```
[[2.3469256e-05]
 [2.4511317e-05]
 [1.4641407e-05]
 ...
 [2.1935764e-05]
 [4.0366122e-05]
 [6.9203023e-05]]

[[4.7626432e-05]
 [2.2307049e-05]
 [5.7875510e-05]
 ...
 [3.3763685e-05]
 [4.7775684e-05]
 [5.1054860e-05]]

[[2.7322509e-05]
 [2.9465516e-05]
 [4.3688808e-05]
 ...
 [8.3666288e-05]
 [2.5436042e-05]
 [2.1456282e-05]]]


[[[5.4115546e-04]
 [4.7844026e-04]
 [7.9310773e-04]
 ...
 [7.1769487e-04]
 [6.8242446e-04]
 [4.3775878e-04]]

[[7.6136464e-04]
 [7.2436762e-04]
 [4.4753787e-04]
 ...
 [3.6893538e-04]
 [4.4140356e-04]
 [9.8283915e-04]]

[[1.2843186e-03]
 [4.9177720e-04]
 [6.3294475e-04]
 ...
 [5.6888355e-04]
 [7.7799265e-04]
 [1.1005161e-03]]

...

[[6.2678149e-04]
 [4.8222466e-04]
 [3.6721950e-04]
 ...
 [6.6833314e-04]
 [8.3815790e-04]
 [1.0245679e-03]]

[[7.2167040e-04]
 [6.0479128e-04]
 [7.7935721e-04]
 ...
 [8.1674237e-04]
 [8.3312747e-04]
 [7.7636359e-04]]

[[6.1007723e-04]
 [5.2048307e-04]
 [5.9683825e-04]
 ...
 [1.5904694e-03]
 [4.4437574e-04]
 [4.2481415e-04]]]


[[[2.0674616e-03]
 [1.0710831e-03]
 [9.9856651e-04]
 ...
 [1.2671027e-03]
 [1.7077145e-03]
 [9.2161720e-04]]

[[2.0377026e-03]
 [2.1599559e-03]
 [1.2066105e-03]
 ...
 [8.6697465e-04]
 [1.9088528e-03]
 [1.7331848e-03]]

[[2.2971588e-03]
 [1.1000087e-03]
 [2.2110089e-03]
 ...
 [1.8772073e-03]
 [1.7951989e-03]
 [2.0434007e-03]]

...

[[7.2122668e-04]
 [1.6553159e-03]
 [1.1298530e-03]
 ...
 [1.4035467e-03]
 [1.3911786e-03]
 [1.4433725e-03]]

[[1.7606072e-03]
 [1.0313670e-03]
 [1.4691765e-03]
 ...
 [7.8661472e-04]
 [1.3344218e-03]
 [1.7758040e-03]]

[[1.4094569e-03]
 [1.7727838e-03]
 [1.5966828e-03]
 ...
 [1.9344462e-03]
 [1.3889156e-03]
 [1.1715234e-03]]]


[[[3.9226623e-07]
 [1.2453586e-07]
 [4.1163014e-07]
 ...
 [3.7594117e-07]
 [3.3861065e-07]
 [2.5874581e-07]]

[[6.6959967e-07]
 [3.5860347e-07]
 [1.6121440e-07]
 ...
 [1.7803978e-07]
 [4.0040794e-07]
 [4.8461101e-07]]

[[2.6278415e-06]
```

```
 [2.2676080e-07]
 [9.2463955e-07]
 ...
 [2.0209902e-07]
 [3.0136650e-07]
 [6.4595287e-07]]

...

[[2.6270845e-07]
 [3.3992751e-07]
 [1.4953199e-07]
 ...
 [2.1571201e-07]
 [5.9698505e-07]
 [1.4174415e-06]]

[[8.7603928e-07]
 [2.4710616e-07]
 [1.2314524e-06]
 ...
 [4.0854638e-07]
 [8.1958939e-07]
 [9.5696601e-07]]

[[3.6770879e-07]
 [4.5316202e-07]
 [8.4824467e-07]
 ...
 [1.4416480e-06]
 [3.7497011e-07]
 [2.7425608e-07]]]


[[[5.7964805e-03]
 [3.5178177e-03]
 [4.7708191e-03]
 ...
 [4.5772195e-03]
 [4.3044901e-03]
 [3.5928038e-03]]

 [[5.9966599e-03]
 [4.7303061e-03]
 [3.6042074e-03]
 ...
 [3.2361727e-03]
 [5.3661331e-03]
 [4.9715349e-03]]

 [[9.3117235e-03]
 [3.3018247e-03]
 [6.7277169e-03]
 ...
 [4.7939979e-03]
 [4.4624656e-03]
 [6.9924630e-03]]
```

```
...

[[3.5808675e-03]
 [4.4187619e-03]
 [4.0194131e-03]
 ...
 [4.5377770e-03]
 [5.6230775e-03]
 [6.0294406e-03]]

[[5.6544715e-03]
 [3.9271992e-03]
 [5.8279526e-03]
 ...
 [3.6752529e-03]
 [6.0678488e-03]
 [5.7316730e-03]]

[[4.8677456e-03]
 [5.9076380e-03]
 [5.6793680e-03]
 ...
 [7.0798201e-03]
 [5.6174444e-03]
 [3.8102455e-03]]]]
```

1. Structure of the Predictions

Each array in your output is a 4D tensor with the shape [num_samples, height, width, 1], where:

- num_samples is the number of test samples in your batch.
- height and width are the dimensions of each image (in your case, 256x256).
- 1 indicates that the output is a single-channel (grayscale) image.

2. Understanding the Values

The values in these arrays are floating-point numbers, typically in the range [0, 1] due to normalization during training. These values represent the predicted pixel intensities or parameters for the corresponding test images.

Example Explanation

Consider this specific output snippet:

plaintext

Copy code

```
[[[[2.7745140e-05]
  [1.4068257e-05]
  [3.2726755e-05]
  ...
  [3.0458801e-05]
  [2.8348173e-05]
  [2.0270165e-05]]
```

- **2.7745140e-05** is the predicted value for a pixel in the image. This small value indicates a low intensity (near black) in the predicted image.
- **1.4068257e-05** is another pixel value, also quite small.

3. Interpreting Predictions

For Image Predictions:

- **Normalized Values:** Since you likely normalized your image data to the range [0, 1], these predictions are in the same range. The values close to 0 represent darker regions in the predicted images, and values close to 1 represent brighter regions.

For Parameter Predictions:

If the output is not an image but a set of parameter predictions (like PRECTOTCORR, WS10M, etc.), the structure would still be similar:

plaintext

Copy code

```
[[[5.4115546e-04]
  [4.7844026e-04]
  [7.9310773e-04]
  ...
  [7.1769487e-04]
  [6.8242446e-04]
  [4.3775878e-04]]
```

Here, each value represents a normalized prediction for a parameter. You should map these back to the original scale if you normalized the data before training.

4. Potential Issues

If all predictions are identical or very similar, it might indicate a problem with the model or training. Here are some common issues:

- **Model Overfitting:** The model might be overfitting to the training data and not generalizing well to the test data.
- **Learning Rate Issues:** If the learning rate is too high or too low, it can affect how well the model learns.
- **Data Imbalance:** If your training data is not representative, the model might not learn meaningful patterns.

EVALUATION METRICS:

BATCH SIZE = 128

5/5 [===============================] - 15s 3s/step

MSE: 0.023583538830280304

MAE: 0.03489188104867935

RMSE: 0.1535693258047104

R^2: 0.13255018552164769

BATCH SIZE = 64

5/5 [===============================] - 7s 1s/step

MSE: 0.022882582619786263

MAE: 0.03287021443247795

RMSE: 0.15126989781856537

R^2: -0.008572827100427663

BATCH SIZE = 32

1/1 [===============================] - 3s 3s/step

MSE: 0.01596750319004059

MAE: 0.016038144007325172

RMSE: 0.12636259198188782

R2 Score: 0.8854514189832527

BATCH SIZE = 16

1/1 [==============================] - 4s 4s/step

MSE: 0.01108498964458704

MAE: 0.024387391284108162

RMSE: 0.10528527945280075

R^2: -0.008976259284156216