# BIRLA INSTITUTE OF TECHNOLOGY, MESRA

## SESSION: 2023-25


# CA512: BASICS OF MACHINE LEARNING – LAB PROJECT

# Weather Classification Using CNN

**Prepared by:**

| | |
|---|---|
| **Uttam Kumar** | **MCA/10016/23** |
| **Vikrant Rathi** | **MCA/10005/23** |
| **Shubham Kumar** | **MCA/10014/23** |
| **Aakash Chaudhary** | **MCA/10034/23** |

**Teacher's Signature:**

# INDEX

# Introduction

**Problem Statement**: Developing a weather prediction system using CNN to classify weather images into four categories—sunny, rainy, cloudy, and shiny—is the primary objective of this project. By focusing on image-based weather prediction, we aim to bridge a gap in real-time weather classification, which could prove invaluable in agriculture, public safety, and disaster preparedness.

**Relevance in India**: India is particularly susceptible to weather-related disruptions like floods and droughts, directly affecting the economy and public welfare. Given that a large proportion of India's population depends on agriculture, precise weather forecasting has the potential to improve crop productivity and mitigate risks associated with adverse weather. Real-time forecasts and targeted alerts based on CNN models enable decision-makers and farmers to act more effectively, whether for preparing for heavy rains, optimizing water use during dry spells, or planning for seasonal transitions.

**CNN in Machine Learning**: CNNs are ideal for image analysis tasks, learning features from images such as patterns, textures, and shapes without manual feature extraction. This unique capability allows CNNs to excel in image-based predictions, making them suitable for weather classification tasks, where recognizing cloud formations or rain patterns is crucial.

# Role of CNN in Machine Learning

**Understanding CNNs**: CNNs belong to the broader category of machine learning and deep learning models designed explicitly for visual data. CNNs automate the learning of features from raw images, eliminating the need for manual intervention. A CNN works by scanning each section of an image to detect significant patterns, like edges and textures, before moving to deeper layers for complex features, which allows it to 'understand' objects in the image.

**Applications of CNNs in Image Recognition:**

CNNs have applications beyond weather classification:

- **Medical Imaging:** Identifying tumours, fractures, and diseases.
- **Autonomous Vehicles:** Detecting road elements, pedestrians, and traffic signals.
- **Retail:** Powering visual search engines that allow users to find products by image.

**Automatic Feature Extraction:**

One of CNNs' most powerful features is their ability to extract relevant patterns automatically. This allows CNNs to handle data preprocessing internally, saving time and ensuring consistent results.

**Importance for Weather Classification:**

In weather classification, CNNs detect subtle visual patterns that signify weather types, such as rain or clouds. By understanding image features, CNNs can accurately classify weather conditions, aiding sectors that rely on timely weather insights.

# Data Collection & Description

**Data Collection Process:**

The dataset comprises 1,125 images sourced from team contributions, professional agency images (e.g., Picway Galaxy), and open-source platforms. This data variety ensures that diverse weather patterns are represented, from clear skies to stormy conditions.

**Dataset Categories:**

The dataset is split into four main categories:

- **Cloudy:** Overcast skies with no rain, characterized by different cloud formations.
- **Rainy:** Images showing precipitation, storms, and overcast skies.
- **Shiny:** Clear skies without cloud cover, ideal for capturing sunny weather.
- **Sunrise:** Images of the sun at the horizon with warm colours, symbolizing morning or evening.

**Data Split:**

The dataset is divided as follows:

- **Training Set:** 80% of images used to teach the model.
- **Validation Set:** 10% used to evaluate model tuning.
- **Test Set:** 10% to test model generalization.

# Data Preprocessing

**Resizing Images:** Each image was resized to 224x224 pixels to fit the CNN's input requirements. This step ensures uniformity across data inputs, preventing errors during model training.

**Rescaling Pixel Values:** Each pixel was normalized by dividing its value by 255, converting the values to a 0–1 range. Rescaling aids the model by standardizing data, improving convergence during training.

**Data Augmentation Techniques:** To boost model generalization, various data augmentation methods were applied:

- **Random Flipping:** Images were flipped horizontally and vertically, creating varied perspectives of similar weather conditions.
- **Random Rotation:** Images were rotated randomly up to 50%, making the model robust to orientation variations.

**Importance of Preprocessing:** Each preprocessing step optimizes the data for effective training, enhancing the CNN's ability to recognize distinct weather patterns under various transformations.

# CNN Overview & Structure

**Convolutional Neural Network (CNN):** A CNN is a deep learning architecture tailored for image analysis tasks, where each layer detects specific image features. For weather classification, the CNN first recognizes basic shapes (edges) and later identifies complex structures (cloud patterns, sun positions).

**Layered Approach:**

- **Input Layer:** Accepts a 224x224x3 RGB image, where each channel represents colour depth.
- **Convolutional Layers:** Extract image features like textures and edges.
- **Pooling Layers:** Reduce feature map dimensions, retaining critical information while minimizing data size.

**Pooling Mechanism:** Pooling layers simplify the data, reducing computational load. By retaining only essential features, pooling aids in making the CNN both efficient and effective in learning image representations.

# Flattening, Fully Connected, and Output Layers

**Flattening Layer:** Converts the CNN's 2D output into a 1D vector. Flattening prepares the data for the fully connected layer, translating spatial data into a format suitable for high-level processing.

**Fully Connected Layer:** Connects every neuron to each previous layer neuron, learning complex relationships in the data. This layer synthesizes the extracted features to make final predictions.

**Output Layer with Softmax:** For multi-class classification, the Softmax function in the output layer converts raw scores into probabilities, identifying the most likely category (e.g., Cloudy, Rainy).

**Role in Classification:** These layers ensure accurate classification by leveraging learned features and mapping them to defined categories, finalizing the model's understanding of image content.

# Model Compilation

**Model Compilation Overview:** The CNN model is compiled using the following:

- **Optimizer:** Adam, selected for its efficiency in deep learning.
- **Loss Function:** Sparse Categorical Cross entropy, ideal for multi-class classification.
- **Evaluation Metric:** Accuracy, allowing us to track correct classifications as a percentage of total predictions.

**Purpose of Compilation:** Compilation is essential for transforming the CNN from an architecture into a fully operational model. The optimizer updates weights to minimize error, and the loss function provides feedback on model accuracy.

**Code Implementation:**

```
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

# Model Training

**Training Specifications:** The model trained for 50 epochs with a batch size of 32. During training, each epoch represents a complete pass over the dataset, while batches split the dataset into smaller portions, allowing efficient gradient descent.

**Performance Monitoring:** Training accuracy began at 31.23%, with validation accuracy at 56.25%. By the final epoch, training accuracy reached 85.90%, and validation accuracy was 89.58%, indicating substantial improvement.

**Training Process Code:**

python

Copy code

```
history = model.fit(train_data, epochs=50,
batch_size=32, validation_data=val_data,
verbose=1)
```

# Model Evaluation

**Final Evaluation Metrics:**

Model evaluation is critical for understanding how well the CNN has learned to generalize across unseen data. After training, the model is assessed on the training, validation, and test datasets to calculate final metrics such as accuracy, precision, recall, and F1 score. Accuracy alone can be limiting, especially if the data is imbalanced. For example, the CNN might be highly accurate on sunny and cloudy categories but struggle with rainy or sunrise images if these categories are underrepresented. Precision and recall offer additional insight by measuring how well the model identifies true positives and avoids false positives, respectively.

**Metrics Interpretation:**

- **Training Accuracy:** This metric evaluates how well the model has fit the training data. High training accuracy indicates that the model has effectively learned from the labelled data provided during training.
- **Validation Accuracy:** This metric provides insight into how well the model generalizes to new data. The gap between training and validation accuracy helps identify overfitting; a significant drop in validation accuracy indicates the model might be too tuned to the training data.
- **Loss Values:** Training and validation loss values indicate the error magnitude in predictions. Decreasing loss over epochs suggests the model is learning effectively, while a rising validation loss may signal overfitting.

**Code Example for Evaluation:**

```
scores = model.evaluate(test_data)
print("Test Loss:", scores[0])
```

```
print("Test Accuracy:", scores[1])
```

**Significance of Metrics:**

Evaluation metrics guide improvements. High accuracy on both training and validation data suggests good generalization. If validation loss surpasses training loss, techniques like dropout, data augmentation, or early stopping can mitigate overfitting. Understanding these metrics aids in refining the model and optimizing hyperparameters.

# Visualizing Model Performance

**Accuracy Graph:**

The accuracy graph is a key visualization tool for tracking model progress. This graph displays training and validation accuracy trends over epochs, highlighting how the model improves with additional learning. For example, if training accuracy rises steadily but validation accuracy plateaus, it may indicate that the model requires further tuning, such as adjusting the learning rate or incorporating regularization methods.

**Loss Graph:**

The loss graph charts the training and validation loss over time, showing the decrease in prediction error across epochs. A downward trend in loss indicates effective learning, while an upward trend, especially in validation loss, may suggest overfitting. This graph also assists in identifying the optimal number of epochs for training, as excessively high epochs can lead to diminishing returns and even degrade model performance.

**Visual Interpretation and Insights:**

These graphs provide valuable diagnostic insights. Consistent patterns in accuracy and loss suggest that the CNN is learning effectively. Divergent trends, where validation metrics deviate from training metrics, prompt a closer examination of model parameters and architecture.

**Code Example for Visualization:**

```
import matplotlib.pyplot as plt

# Plot training & validation accuracy values
plt.plot(history.history['accuracy'])
```

```
plt.plot(history.history['val_accuracy'])
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(['Train', 'Validation'], loc='upper
left')
plt.show()
```

Visualization adds an intuitive layer to model evaluation, making performance diagnostics straightforward and data-driven.

# Testing and Prediction

**Testing on Test Data:**

Testing evaluates the CNN's robustness on unseen images. A comprehensive test set, ideally encompassing varied weather conditions, offers an accurate reflection of real-world performance. Unlike the validation set, which provides feedback during training, the test set remains untouched until final evaluation, ensuring unbiased performance assessment.

**Prediction and Confidence Levels:**

Each test image receives a predicted class label (e.g., sunny, rainy) and an associated confidence level. Confidence scores reflect the model's certainty in its predictions, with higher scores indicating more reliable classifications. These scores are crucial for applications requiring high accuracy, such as real-time alerts for adverse weather conditions.

**Example Prediction Code:**

```
predictions = model.predict(test_images)
for i, pred in enumerate(predictions):
    predicted_class = class_names[np.argmax(pred)]
    confidence = round(np.max(pred) * 100, 2)
    print(f"Image {i}: Predicted {predicted_class}
with {confidence}% confidence")
```

**Significance of Predictions:**

Predictions reveal the model's effectiveness in differentiating weather types. High confidence scores in accurate predictions indicate robustness, while low confidence in misclassifications may highlight areas where additional data or tuning is required.

# Limitations

**Image Quality and Resolution:**

Poor-quality images or low-resolution inputs can significantly impact classification accuracy. For example, images taken in poor lighting or with low contrast might obscure weather features like clouds or rain, making it difficult for the model to identify patterns correctly. Higher-resolution images would likely improve classification accuracy but may increase computational demands.

**Real-Time Prediction Challenges:**

The CNN model's structure demands considerable computational resources, particularly for high-resolution images. Real-time classification poses additional challenges, as processing large volumes of image data quickly requires advanced hardware, including GPUs. The current model may exhibit delays in making predictions if deployed on standard CPUs, limiting its applicability for live weather monitoring and alerts.

**Limited Data Availability:**

While the dataset includes a reasonable variety of weather conditions, its size and diversity could still limit model accuracy, especially for less frequent weather types. Increasing the dataset with additional labelled images across seasons and weather events would enhance model performance.

**Other Challenges:**

In addition to technical limitations, image-based classification may struggle to predict extreme conditions like drought or heatwaves, which lack distinct visual markers. The reliance on cloud patterns further limits the model's scope, as not all-weather phenomena produce identifiable cloud formations.

# Future Scope

**Sensor Integration for Enhanced Accuracy:**

To improve predictions, integrating additional data such as temperature, humidity, and wind speed from meteorological sensors could complement visual analysis. By incorporating these factors, the CNN model could potentially adjust predictions based on contextual environmental data, enhancing accuracy and providing a more holistic forecast.

**Real-Time Prediction and Alerts:**

Advancing the model for real-time application requires optimizing processing speed. Techniques such as model compression or leveraging edge computing could enable quick weather predictions. Real-time prediction capabilities would allow the system to generate alerts for approaching storms, improving public safety and preparedness.

**Mobile App Development:**

Expanding accessibility through a mobile app could allow users to upload sky images and receive weather predictions instantly. Such an app could incorporate both visual and sensor-based predictions, offering users real-time weather insights based on their geographic location and personal observations.

**Integration with Climate Monitoring Systems:**

With further refinement, this model could extend to long-term climate monitoring. Satellite imagery combined with CNN-based weather classification could assist in identifying climate trends and anomalies, providing valuable data for research and policy.

# Societal Impact and Future Implications

**Applications in Agriculture:**

Accurate weather classification could provide valuable insights to the agricultural sector. Farmers could use these predictions to optimize planting schedules, irrigation planning, and crop harvesting. For instance, early forecasts of heavy rain could prevent crop loss, while warnings of drought conditions would allow for pre-emptive measures in water conservation.

**Healthcare and Public Safety:**

Weather predictions play a crucial role in healthcare, as they can help prevent weather-related illnesses and accidents. By integrating weather classification into public health systems, authorities could issue warnings during high-risk conditions, such as intense heat, storms, or flooding, safeguarding vulnerable populations.

**Impact on Autonomous Systems:**

Autonomous vehicles and drones depend heavily on real-time environmental data. Weather classification systems could be integrated into these technologies to provide alerts or adjustments to navigation in adverse conditions, enhancing safety for self-driving cars, delivery drones, and agricultural machinery.

**Future Implications:**

The future of weather classification may involve combining CNN with other AI techniques, such as reinforcement learning or recurrent neural networks, for sequential prediction. CNNs could eventually aid in long-term climate change monitoring by analysing satellite imagery, detecting trends, and helping policymakers implement proactive strategies.

# Weather Classification Using CNN

## Problem Statement :

Weather plays a significant role in various industries, including agriculture, aviation, and outdoor event planning. Accurate weather classification helps in understanding weather patterns and making informed decisions. This task involves classifying weather images into four different categories:

1. Cloudy,
2. Rain,
3. Shiny,
4. Sunrise.

The process involves several steps, including importing libraries, loading and viewing data, splitting the data into training and validation sets, data pre-processing, building the CNN model, training the model, and analyzing the model's performance.

**Step 1:** Importing Libraries and Data: In this step, we import the necessary libraries such as Pandas, NumPy, and TensorFlow. We also load the weather images from the dataset directory using tf.keras.preprocessing.image_dataset_from_directory. This function automatically loads the images from the directory and creates an image dataset along with their corresponding labels.

**Step 2:** Viewing Data Images: We visualize and inspect a few samples of the loaded data to ensure that the images are correctly loaded and to understand the structure of the dataset.

**Step 3:** Splitting the Data: We split the dataset into training and validation sets. The training set is used to train the CNN model, while the validation set is used to evaluate the model's performance on unseen data.

**Step 4:** Data Pre-processing: Data pre-processing involves several steps to prepare the images for training. These steps include resizing and rescaling the images to a consistent size, data augmentation to increase the dataset's diversity and generalization, and other normalization techniques.

**Step 5:** Model Building: In this step, we build the CNN model using TensorFlow's Keras API. The model consists of Convolutional layers (Conv2D), MaxPooling layers to reduce the

spatial dimensions, and other necessary layers. We optimize the model using the Adam optimizer and use Sparse Categorical Crossentropy as the loss function, as we have multiple classes and integer-encoded labels. We also define accuracy metrics to monitor the model's performance during training.

**Step 6:** Model Training and Analysis: We train the CNN model on the training set and evaluate its performance on the validation set. During training, we monitor metrics such as training accuracy, validation accuracy, training loss, and validation loss to analyze the model's behavior and identify potential overfitting.

**Step 7:** Predicting on New Images (Unknown Weather): Finally, we plot and analyze images from a new directory called "Unknown Weather," which contains weather images downloaded from the internet. We use the trained CNN model to predict the class (Cloudy, Rain, Shiny, or Sunrise) of each unknown weather image.

# Step 1 : Importing Libraries and Data

import sys !{sys.executable} -m pip install tensorflow import tensorflow as tf print(tf.**version**)

```
# importing libraries
import numpy as np # for linear algebra
import pandas as pd # data preprocessing

import tensorflow as tf # deep learning
from tensorflow.keras import layers, models # working on layers
import matplotlib.pyplot as plt # data visualization
```

```
# setting image_size and batch_size
IMAGE_SIZE = 224
BATCH_SIZE = 32
```

```
# importing dataset directory
df = tf.keras.preprocessing.image_dataset_from_directory("weather_classification",s
                                                         batch_size= BATCH_SIZE)
```

```
Found 1126 files belonging to 4 classes.
```

```
# getting class names
class_names = df.class_names
class_names
```

```
['Cloudy', 'Rain', 'Shiny', 'Sunrise']
```

```
# len of dataset ( Total number of files/ batch size)
len(df)
```

```
36
```

```
36*32 #1152, but number of files are 1125 only
```

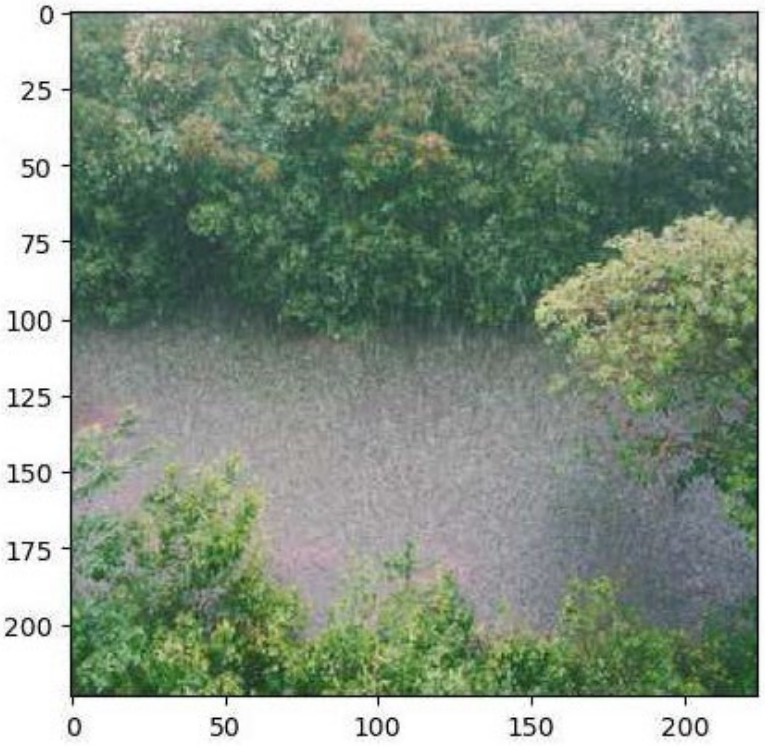
Out[18]: 1152

In [20]:
```
print(df)  # None is the undefined batch size, (256,256) is the image size and 3 =
```

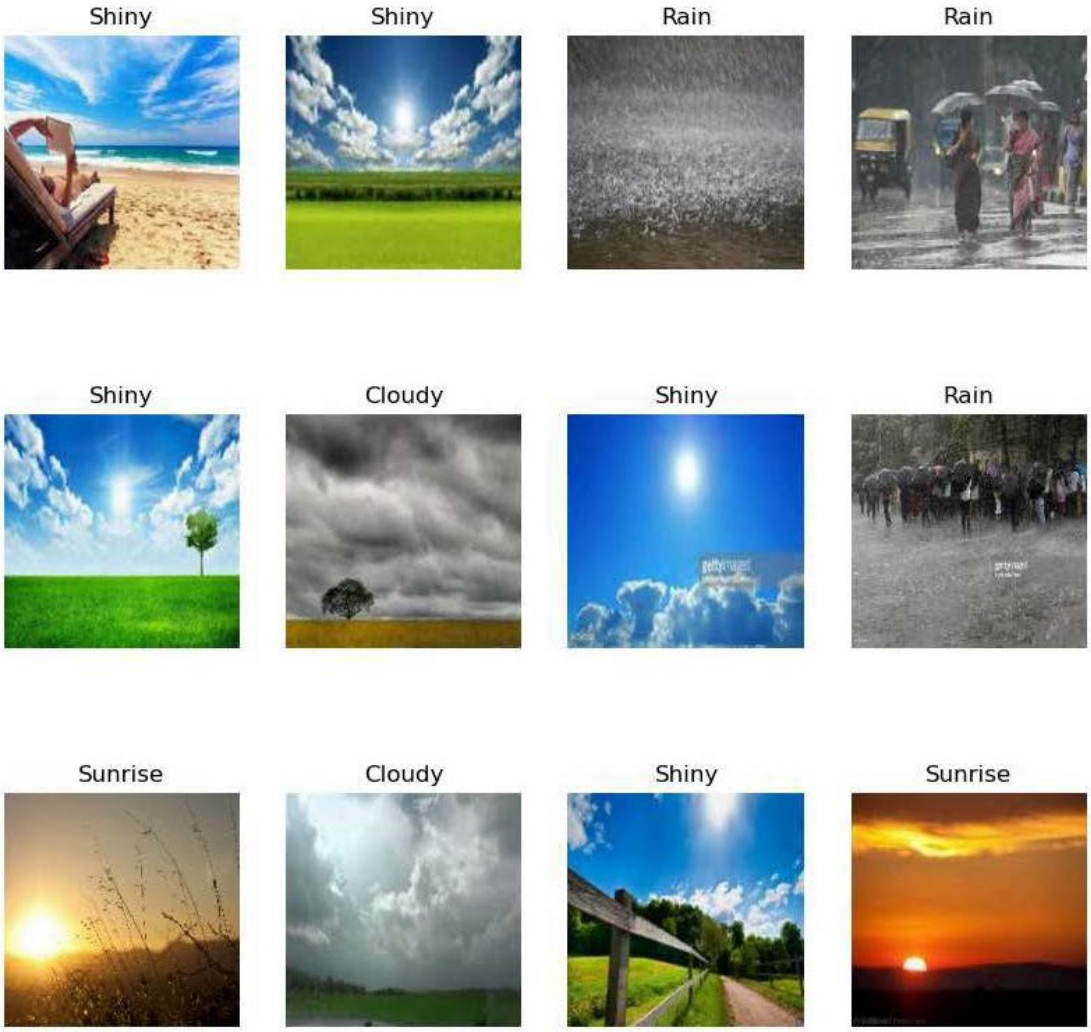
```
<_PrefetchDataset element_spec=(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float
32, name=None), TensorSpec(shape=(None,), dtype=tf.int32, name=None))>
```

## Step 2 : Viewing Data Images

In [23]:
```
# viewing image
for image_batch, label_batch in df.take(1):
    plt.imshow(image_batch[0].numpy().astype("uint8"))

    # plt.axis("off") : used to view image without axis points
# refreshing every time gives another picture and label as given
```




In [25]:
```
# images with the labels
plt.figure(figsize=(10,10))
for image_batch, label_batch in df.take(1):
    for i in range(12):
        plt.subplot(3,4,i+1)
        plt.imshow(image_batch[i].numpy().astype("uint8"))
        plt.title(class_names[label_batch[i]])
        plt.axis('off')

# refreshing every time gives another picture and label as given
```




## Step 3 : Splitting Dataset

80% --> Training

20% --> 10% Validation and 10% Testing

In [29]:
```
# channels (RGB[Red, Green, Blue]), Epochs = 50
CHANNELS = 3
EPOCHS = 50
```

In [31]:
```
# checking the length of training data
train_size = 0.8
len(df) * train_size
```

Out[31]: 28.8

In [33]:
```
# length of validation and test data
val_n_test = len(df) - (len(df) * train_size)
val_n_test
```

Out[33]: 7.199999999999999

In [35]:
```
# splitting the data into training , validation and testing data

def split_datasets(pc,train_split = 0.8,val_split = 0.1, test_split = 0.1, shuffle
    if shuffle:
        pc = pc.shuffle(shuffle_size,seed = 10)

    pc_size = len(pc) # size of weather_data(36)
    train_size = int(train_split*pc_size)
    val_size = int(val_split*pc_size)

    train_pc = pc.take(train_size) # taking first 28 batches(out of 36)
    val_pc = pc.skip(train_size).take(val_size) # leaving first 28 and taking next
    test_pc = pc.skip(train_size).skip(val_size) # skipping first 28(train) batch a
                                                 #taking left 4 batches for test

    return train_pc, val_pc, test_pc
```

In [37]:
```
# getting the training, validation and testing data by 'split_datasets' function
train_data, val_data, test_data = split_datasets(df)
```

In [ ]:
```

```

In [40]:
```
# printing the size of all data splits
print("Size of Data is :{0} \nBatch Size of Training Data is :{1} \nBatch Size of V
      .format(len(df), len(train_data), len(val_data), len(test_data)))
```

```
Size of Data is :36
Batch Size of Training Data is :28
Batch Size of Validation Data :3
Batch Size of Test Data :5
```

In [62]:
```
# caching, shuffling and prefetching the data
train_pc = train_data.cache().shuffle(100).prefetch(buffer_size=tf.data.AUTOTUNE)
val_pc = val_data.cache().shuffle(100).prefetch(buffer_size=tf.data.AUTOTUNE)
test_pc = test_data.cache().shuffle(100).prefetch(buffer_size=tf.data.AUTOTUNE)
```

## Step 4 : Data Pre-Processing

In [65]:
```
# Image Preprocessing: Rescaling and Resizing
rescale_n_resize = tf.keras.Sequential([
    layers.Resizing(IMAGE_SIZE, IMAGE_SIZE),
    layers.Rescaling(1.0 / 255)
])

# Data Augmentation by flipping and rotating existing images
data_augmentation = tf.keras.Sequential([
    layers.RandomFlip(mode='horizontal_and_vertical'),
```

```
    layers.RandomRotation(factor=0.5)
])
```

## Step 5 : Model Building

In [68]:
```
# Creating CNN
model = models.Sequential([
    rescale_n_resize,
    data_augmentation,
    layers.Conv2D(filters=16, kernel_size=(3, 3), activation='relu', input_shape=(I
    layers.MaxPool2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPool2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPool2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPool2D((2, 2)),
    layers.Conv2D(128, (3, 3), activation='relu'),
    layers.MaxPool2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPool2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(64, activation='softmax')
])
```

In [79]:
```
# model_summary
model.summary()
```

```
Model: "sequential_7"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| sequential_5 (Sequential) | (None, 224, 224, 3) | 0 |
| sequential_6 (Sequential) | (None, 224, 224, 3) | 0 |
| conv2d_18 (Conv2D) | (None, 222, 222, 16) | 448 |
| max_pooling2d_18 (MaxPooling2D) | (None, 111, 111, 16) | 0 |
| conv2d_19 (Conv2D) | (None, 109, 109, 64) | 9,280 |
| max_pooling2d_19 (MaxPooling2D) | (None, 54, 54, 64) | 0 |
| conv2d_20 (Conv2D) | (None, 52, 52, 128) | 73,856 |
| max_pooling2d_20 (MaxPooling2D) | (None, 26, 26, 128) | 0 |
| conv2d_21 (Conv2D) | (None, 24, 24, 64) | 73,792 |
| max_pooling2d_21 (MaxPooling2D) | (None, 12, 12, 64) | 0 |
| conv2d_22 (Conv2D) | (None, 10, 10, 128) | 73,856 |
| max_pooling2d_22 (MaxPooling2D) | (None, 5, 5, 128) | 0 |
| conv2d_23 (Conv2D) | (None, 3, 3, 64) | 73,792 |
| max_pooling2d_23 (MaxPooling2D) | (None, 1, 1, 64) | 0 |
| flatten_3 (Flatten) | (None, 64) | 0 |
| dense_6 (Dense) | (None, 128) | 8,320 |
| dense_7 (Dense) | (None, 64) | 8,256 |

**Total params:** 964,802 (3.68 MB)

**Trainable params:** 321,600 (1.23 MB)

**Non-trainable params:** 0 (0.00 B)

**Optimizer params:** 643,202 (2.45 MB)

In [72]:
```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',  # Adjust based on your targe
              metrics=['accuracy'])
```

In [74]:
```
# optimizing the model
model.compile(optimizer='adam', loss = tf.keras.losses.SparseCategoricalCrossentrop
              metrics=['accuracy'])
```

## Step 6 : Model Training and Analysis

In [77]:
```
# fitting the model
history = model.fit(train_data, epochs = EPOCHS, batch_size=BATCH_SIZE, validation_
```

```
Epoch 1/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 26s 703ms/step - accuracy: 0.2258 - loss: 2.6053 - val_ac
curacy: 0.4271 - val_loss: 1.2799
Epoch 2/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 580ms/step - accuracy: 0.4926 - loss: 1.0899 - val_ac
curacy: 0.7188 - val_loss: 0.6775
Epoch 3/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 19s 657ms/step - accuracy: 0.6523 - loss: 0.8002 - val_ac
curacy: 0.7500 - val_loss: 0.7214
Epoch 4/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 18s 614ms/step - accuracy: 0.6958 - loss: 0.6879 - val_ac
curacy: 0.7083 - val_loss: 0.7364
Epoch 5/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 18s 615ms/step - accuracy: 0.6994 - loss: 0.6653 - val_ac
curacy: 0.7292 - val_loss: 0.5935
Epoch 6/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 572ms/step - accuracy: 0.6841 - loss: 0.6303 - val_ac
curacy: 0.6979 - val_loss: 0.5646
Epoch 7/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 573ms/step - accuracy: 0.7445 - loss: 0.5981 - val_ac
curacy: 0.7292 - val_loss: 0.5387
Epoch 8/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 18s 626ms/step - accuracy: 0.7576 - loss: 0.5799 - val_ac
curacy: 0.7292 - val_loss: 0.5353
Epoch 9/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 21s 728ms/step - accuracy: 0.7184 - loss: 0.6827 - val_ac
curacy: 0.8646 - val_loss: 0.4380
Epoch 10/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 21s 725ms/step - accuracy: 0.8042 - loss: 0.4917 - val_ac
curacy: 0.7917 - val_loss: 0.5962
Epoch 11/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 19s 645ms/step - accuracy: 0.7292 - loss: 0.6031 - val_ac
curacy: 0.8438 - val_loss: 0.4156
Epoch 12/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 558ms/step - accuracy: 0.8005 - loss: 0.5050 - val_ac
curacy: 0.7917 - val_loss: 0.6420
Epoch 13/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 578ms/step - accuracy: 0.8068 - loss: 0.4987 - val_ac
curacy: 0.7812 - val_loss: 0.4833
Epoch 14/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 566ms/step - accuracy: 0.8052 - loss: 0.4778 - val_ac
curacy: 0.8714 - val_loss: 0.3939
Epoch 15/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 18s 604ms/step - accuracy: 0.8043 - loss: 0.4826 - val_ac
curacy: 0.8333 - val_loss: 0.4232
Epoch 16/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 20s 676ms/step - accuracy: 0.8001 - loss: 0.4604 - val_ac
curacy: 0.7812 - val_loss: 0.5743
Epoch 17/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 20s 678ms/step - accuracy: 0.8153 - loss: 0.4879 - val_ac
curacy: 0.9062 - val_loss: 0.4233
Epoch 18/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 559ms/step - accuracy: 0.8532 - loss: 0.3838 - val_ac
curacy: 0.8542 - val_loss: 0.4088
Epoch 19/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 598ms/step - accuracy: 0.8574 - loss: 0.4121 - val_ac
```

```
curacy: 0.8646 - val_loss: 0.3849
Epoch 20/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 23s 799ms/step - accuracy: 0.8730 - loss: 0.3747 - val_ac
curacy: 0.8750 - val_loss: 0.3221
Epoch 21/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 16s 557ms/step - accuracy: 0.8623 - loss: 0.3664 - val_ac
curacy: 0.9062 - val_loss: 0.2540
Epoch 22/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 20s 696ms/step - accuracy: 0.8500 - loss: 0.3700 - val_ac
curacy: 0.8333 - val_loss: 0.3733
Epoch 23/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 20s 682ms/step - accuracy: 0.8359 - loss: 0.4258 - val_ac
curacy: 0.8438 - val_loss: 0.4026
Epoch 24/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 19s 651ms/step - accuracy: 0.8801 - loss: 0.3398 - val_ac
curacy: 0.9167 - val_loss: 0.2712
Epoch 25/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 19s 633ms/step - accuracy: 0.8713 - loss: 0.3667 - val_ac
curacy: 0.8021 - val_loss: 0.5317
Epoch 26/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 19s 645ms/step - accuracy: 0.8609 - loss: 0.3804 - val_ac
curacy: 0.9062 - val_loss: 0.2028
Epoch 27/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 18s 604ms/step - accuracy: 0.8469 - loss: 0.3709 - val_ac
curacy: 0.8854 - val_loss: 0.3595
Epoch 28/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 21s 706ms/step - accuracy: 0.8566 - loss: 0.3524 - val_ac
curacy: 0.8646 - val_loss: 0.3841
Epoch 29/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 19s 672ms/step - accuracy: 0.9009 - loss: 0.2910 - val_ac
curacy: 0.8958 - val_loss: 0.2505
Epoch 30/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 556ms/step - accuracy: 0.8772 - loss: 0.3251 - val_ac
curacy: 0.9271 - val_loss: 0.2873
Epoch 31/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 16s 544ms/step - accuracy: 0.8472 - loss: 0.3469 - val_ac
curacy: 0.8229 - val_loss: 0.3611
Epoch 32/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 19s 664ms/step - accuracy: 0.8866 - loss: 0.3062 - val_ac
curacy: 0.8750 - val_loss: 0.3508
Epoch 33/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 23s 781ms/step - accuracy: 0.8991 - loss: 0.2871 - val_ac
curacy: 0.9375 - val_loss: 0.2029
Epoch 34/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 22s 754ms/step - accuracy: 0.8780 - loss: 0.3154 - val_ac
curacy: 0.9062 - val_loss: 0.2639
Epoch 35/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 560ms/step - accuracy: 0.8789 - loss: 0.3181 - val_ac
curacy: 0.8958 - val_loss: 0.2582
Epoch 36/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 18s 607ms/step - accuracy: 0.8916 - loss: 0.2817 - val_ac
curacy: 0.8958 - val_loss: 0.3106
Epoch 37/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 16s 555ms/step - accuracy: 0.8591 - loss: 0.3542 - val_ac
curacy: 0.9167 - val_loss: 0.1702
Epoch 38/50
```

```
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 562ms/step - accuracy: 0.9046 - loss: 0.2640 - val_ac
curacy: 0.8333 - val_loss: 0.3653
Epoch 39/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 19s 652ms/step - accuracy: 0.9020 - loss: 0.2503 - val_ac
curacy: 0.8750 - val_loss: 0.3167
Epoch 40/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 573ms/step - accuracy: 0.8937 - loss: 0.2945 - val_ac
curacy: 0.9375 - val_loss: 0.1868
Epoch 41/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 19s 656ms/step - accuracy: 0.8682 - loss: 0.3135 - val_ac
curacy: 0.8854 - val_loss: 0.2653
Epoch 42/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 18s 639ms/step - accuracy: 0.9060 - loss: 0.2413 - val_ac
curacy: 0.8958 - val_loss: 0.2257
Epoch 43/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 18s 612ms/step - accuracy: 0.8826 - loss: 0.3134 - val_ac
curacy: 0.9062 - val_loss: 0.2859
Epoch 44/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 16s 544ms/step - accuracy: 0.9010 - loss: 0.2595 - val_ac
curacy: 0.9062 - val_loss: 0.3719
Epoch 45/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 19s 664ms/step - accuracy: 0.8834 - loss: 0.3103 - val_ac
curacy: 0.8438 - val_loss: 0.4276
Epoch 46/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 561ms/step - accuracy: 0.8813 - loss: 0.3181 - val_ac
curacy: 0.8958 - val_loss: 0.2790
Epoch 47/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 570ms/step - accuracy: 0.9111 - loss: 0.2509 - val_ac
curacy: 0.9375 - val_loss: 0.1442
Epoch 48/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 569ms/step - accuracy: 0.8994 - loss: 0.2571 - val_ac
curacy: 0.9271 - val_loss: 0.2176
Epoch 49/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 17s 563ms/step - accuracy: 0.9220 - loss: 0.2064 - val_ac
curacy: 0.9688 - val_loss: 0.1339
Epoch 50/50
28/28 ━━━━━━━━━━━━━━━━━━━━ 18s 615ms/step - accuracy: 0.9093 - loss: 0.2551 - val_ac
curacy: 0.9167 - val_loss: 0.2368
```

In [82]:
```
# evaluating the scores
scores = model.evaluate(train_data)
scores
```

```
28/28 ━━━━━━━━━━━━━━━━━━━━ 3s 105ms/step - accuracy: 0.9065 - loss: 0.2664
```

Out[82]:
```
[0.2604271471500397, 0.9091954231262207]
```

In [84]:
```
# getting the keys of fitted model "History"
history.params, history.history.keys()
```

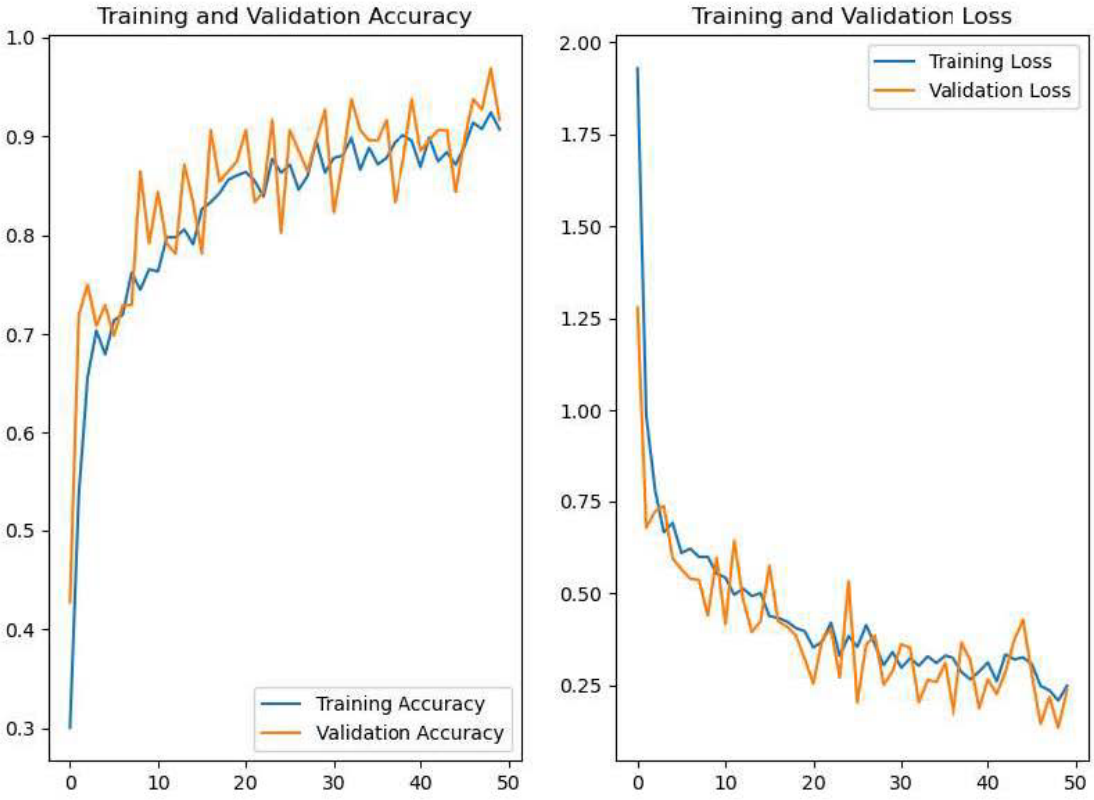
Out[84]:
```
({'verbose': 1, 'epochs': 50, 'steps': 28},
 dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss']))
```

In [88]:
```
# assigning names to the keys
train_loss = history.history['loss']
train_acc = history.history['accuracy']
```

```
val_loss = history.history['val_loss']
val_acc = history.history['val_accuracy']
```

In [90]:
```
# plotting the comparison graphs
plt.figure(figsize = (15,15))

# Accuracy Table
plt.subplot(2,3,1)
plt.plot(range(EPOCHS),train_acc,label='Training Accuracy')
plt.plot(range(EPOCHS),val_acc,label = 'Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

# Loss Table
plt.subplot(2,3,2)
plt.plot(range(EPOCHS),train_loss,label='Training Loss')
plt.plot(range(EPOCHS),val_loss,label = 'Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
```

Out[90]:
```
Text(0.5, 1.0, 'Training and Validation Loss')
```

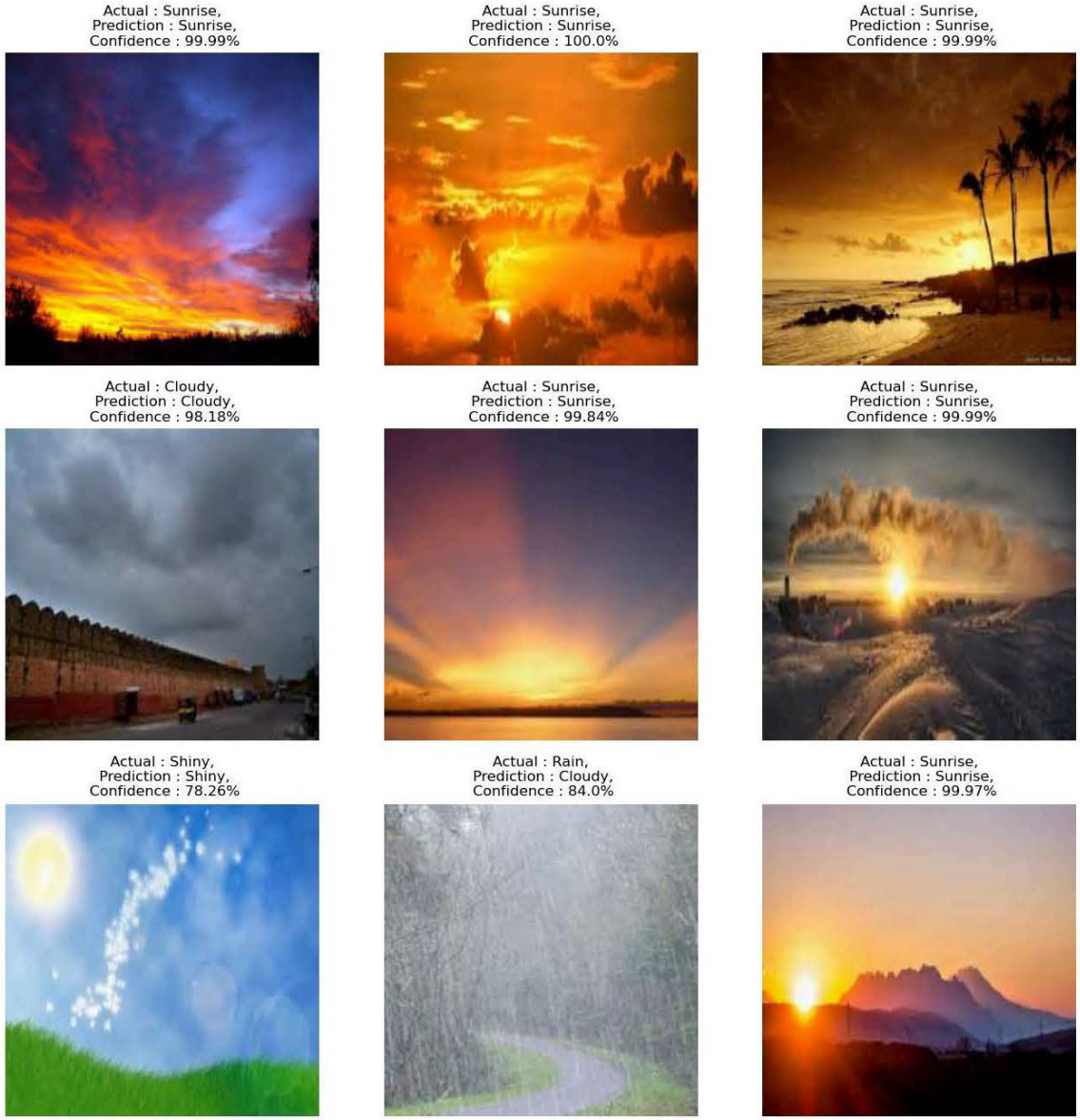



# Validation on Test Data

In [92]:
```
# plotting batch of test images with its actual label, predicted label and confiden
plt.figure(figsize = (16,16))
for batch_image, batch_label in test_pc.take(1):
```

```python
for i in range(9):
    ax = plt.subplot(3,3,i+1)
    image = batch_image[i].numpy().astype('uint8')
    label = class_names[batch_label[i]]

    plt.imshow(image)

    batch_prediction = model.predict(batch_image)
    predicted_class = class_names[np.argmax(batch_prediction[i])]
    confidence = round(np.max(batch_prediction[i]) * 100, 2)

    plt.title(f'Actual : {label},\n Prediction : {predicted_class},\n Confidenc

    plt.axis('off')
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 216ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 107ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 104ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 110ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 105ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 103ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 115ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 102ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 104ms/step
```

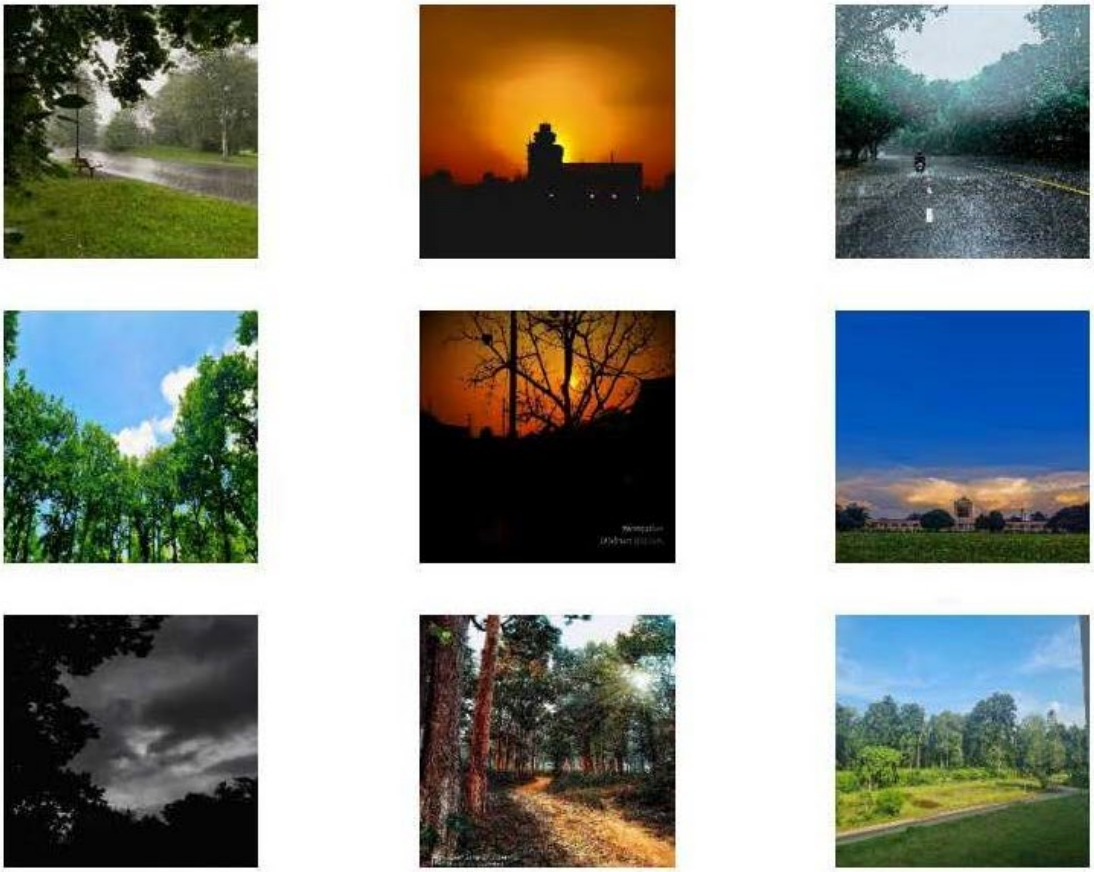



## Step 7 : Predicting on New Images

In [125…
```
# importing new dataset for new predictions
unk_lf = tf.keras.preprocessing.image_dataset_from_directory("New_images",image_siz
```

```
Found 9 files belonging to 1 classes.
```

In [127…
```
for image_batch, label_batch in unk_lf.take(2):
    total_images = image_batch.shape[0]  # Get the total number of images in the ba
    plt.figure(figsize=(10, 10))  # Adjust the figure size if needed
    for i in range(9):
        plt.subplot((9 // 3) + 1, 3, i + 1)  # Adjust the grid size dynamically
        plt.imshow(image_batch[i].numpy().astype("uint8"))
        plt.axis("off")
    plt.show()
```



In [129…

```
plt.figure(figsize=(15, 10))
for image_batch, label_batch in unk_lf.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i+1)  # Changed to a 3x3 grid
        image = image_batch[i].numpy().astype("uint8")
        label = class_names[label_batch[i]]

        plt.imshow(image)

        # Predict the class
        unk_lf_pred = model.predict(image_batch)
        pred_class = class_names[np.argmax(unk_lf_pred[i])]
        confidence = round(np.max(unk_lf_pred[i]) * 100, 2)

        plt.title(f'Predicted Class: {pred_class}, \nConfidence: {confidence}%')
        plt.axis("off")

plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 53ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step
```

Predicted Class: Sunrise, Confidence: 55.7%

Predicted Class: Sunrise, Confidence: 100.0%

Predicted Class: Rain, Confidence: 92.17%

Predicted Class: Sunrise, Confidence: 89.4%

Predicted Class: Rain, Confidence: 99.79%

Predicted Class: Shiny, Confidence: 98.52%

Predicted Class: Cloudy, Confidence: 98.09%

Predicted Class: Rain, Confidence: 99.09%

Predicted Class: Shiny, Confidence: 79.71%

# Thank You!