

Add libraries

```
In [1]: import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3' # Turn off some unimportant messages
```

```
In [2]: import tensorflow as tf
from keras.applications.vgg19 import VGG19
from keras.applications.resnet import ResNet50
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Flatten, Dense, Dropout
import numpy as np
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.regularizers import l1_l2
from tensorflow.keras.optimizers import Adam
import seaborn as sns
```

WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1739810673.452578 97124 cuda_dnn.cc:8310] Unable to register cuDNN factory: Attempting to register factory for plugin cuDNN when one has already been registered
E0000 00:00:1739810673.459302 97124 cuda_blas.cc:1418] Unable to register cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has already been registered

Path system

```
In [3]: # you need the current working directory NB: works both windows and linux
current_working_directory = os.getcwd()
current_working_directory = os.path.dirname(current_working_directory)

# get the directory where I want to download the dataset
path_of_download = os.path.join(*['..', current_working_directory, 'Assignment2', 'weather_dataset'])
print(f"[DIR] The directory of the current dataset is {path_of_download}")
```

[DIR] The directory of the current dataset is /mnt/c/UTU/2025/Computer Vision and Sensor Fusion/Assignment2/weather_dataset

function for data loading

```
In [4]: # here let's do some functions that we can re-use also for other assignment
def load_the_data_and_the_labels(data_set_path: str, target_size: tuple or None = None):
    try:
        dataset, labels, name_of_the_labels = list(), list(), list()
        # let's loop here and we try to discover how many class we have
        for class_number, class_name in enumerate(os.listdir(data_set_path)):
            full_path_the_data = os.path.join(data_set_path, class_name)
            print(f"[WALK] I am walking into {full_path_the_data}")

            # add the list to name_list
            name_of_the_labels.append(class_name)

            for single_image in os.listdir(f"{full_path_the_data}"):
                full_path_to_image = os.path.join(*[full_path_the_data, single_image])

                # add the class number
                labels.append(class_number)

                if target_size is None:
                    # let's load the image
                    image = tf.keras.utils.load_img(full_path_to_image)
                else:
                    image = tf.keras.utils.load_img(full_path_to_image, target_size=target_size)

                # transform PIL object in image
                image = tf.keras.utils.img_to_array(image)

                # add the image to the ds list
                dataset.append(image)

            return np.array(dataset, dtype='uint8'), np.array(labels, dtype='int'), name_of_the_labels
    except Exception as ex:
        print(f"[EXCEPTION] load the data and the labels throws exceptions {ex}")
```

OHE function

```
In [5]: # here we have to one hot encode the labels
def make_the_one_hot_encoding(labels_to_transform):
    try:
        enc = OneHotEncoder(handle_unknown='ignore')
        # this is a trick to figure the array as 2d array instead of list
        temp = np.reshape(labels_to_transform, (-1, 1))
        labels_to_transform = enc.fit_transform(temp).toarray()
        print(f'[ONE HOT ENCODING] Labels are one-hot-encoded: {(labels_to_transform.sum(axis=1) - np.ones(labels_to_transform.shape[0]))}')
        return labels_to_transform
    except Exception as ex:
        print(f'[EXCEPTION] Make the one hot encoding throws exception {ex}')
```

load the data and labels

```
In [6]: # Resize the image to the correct size for VGG19
target_size = (224, 224, 3) # 224 x 224 pixels, 3 RGB color channels

# Call function to load data
X, y, class_names = load_the_data_and_the_labels(path_of_download, target_size)

# Print data info after loading
print(f'Dataset shape: {X.shape}')
print(f'Labels shape: {y.shape}')
print(f'Classes: {class_names}')
```

```
[WALK] I am walking into /mnt/c/UTU/2025/Computer Vision and Sensor Fusion/Assignment2/weather_dataset/Cloudy
[WALK] I am walking into /mnt/c/UTU/2025/Computer Vision and Sensor Fusion/Assignment2/weather_dataset/Rain
[WALK] I am walking into /mnt/c/UTU/2025/Computer Vision and Sensor Fusion/Assignment2/weather_dataset/Shine
[WALK] I am walking into /mnt/c/UTU/2025/Computer Vision and Sensor Fusion/Assignment2/weather_dataset/Sunrise
Dataset shape: (1125, 224, 224, 3)
Labels shape: (1125,)
Classes: ['Cloudy', 'Rain', 'Shine', 'Sunrise']
```

normalize the data

```
In [7]: X = X / 255.0
```

split the dataset in train and test set (ratio 0.3)

```
In [8]: # Divide data into training set (70%) and validation set (30%)
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3, random_state=42)

# Print train, and test size
print(f"Train size: {X_train.shape}, Validation size: {X_val.shape}")
print(f"Validation size: {X_val.shape}")
```

Train size: (787, 224, 224, 3), Validation size: (338, 224, 224, 3)

Validation size: (338, 224, 224, 3)

create the CNN and set all parameters to trainable

a. Input layer b. As base model use VGG19: i. Weights: imagenet ii. Include_top: False iii. Input_shape the target shape described in point 1. c. Add a flatten layer d. Add a Dense layer with 512 units and a dropout layer with 0.1 unit. e. Add a Dense layer with 256 units and a dropout layer with 0.1 unit. f. Add the final classifier with the correct number of units and the suitable activation.

```
In [9]: baseModel = VGG19(input_shape=target_size, weights='imagenet', include_top=False) # Input model use VGG19 use imagenet weight

# Freeze all layers of VGG19
for layer in baseModel.layers:
    layer.trainable = True

# Flatten layer to convert from 4D tensor -> 1D vector
x = Flatten()(baseModel.output)

# Fully Connected Layers (Dense + Dropout)
x = Dense(512, activation='relu', kernel_regularizer=l1_l2(l1=0, l2=0.01))(x) # Add a Dense Layer with 512 units
x = Dropout(0.1)(x) # Dropout Layer with 0.1 unit
x = Dense(256, activation='relu', kernel_regularizer=l1_l2(l1=0, l2=0.01))(x) # Add a Dense Layer with 256 units
x = Dropout(0.1)(x) # Dropout Layer with 0.1 unit

# Output layer with 4 classes for the final classifier
x = Dense(4, activation='softmax')(x) # 4 classes: ['Cloudy', 'Rain', 'Shine', 'Sunrise']

# Combine base_model and Fully Connected Layers into a final model
model = Model(inputs=baseModel.input, outputs=x)
```

```
model.summary() # Print mode summary
```

```
I0000 00:00:1739810687.520624    97124 gpu_device.cc:2022] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 5520 MB memory: -> device: 0, name: NVIDIA GeForce RTX 4070 Laptop GPU, pci bus id: 0000:01:00.0, compute capability: 8.9
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv4 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv4 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808

block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv4 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 512)	12,845,568
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 256)	131,328
dropout_1 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 4)	1,028

Total params: 33,002,308 (125.89 MB)

Trainable params: 33,002,308 (125.89 MB)

Non-trainable params: 0 (0.00 B)

compile the model with adam

```
In [10]: # Compile model with Adam
model.compile(
    optimizer=Adam(learning_rate=0.00001), # Use low learning rate to avoid overfitting
    loss='categorical_crossentropy', # Use categorical_crossentropy for multi-class classification
    metrics=['accuracy']
)
```

Fit the model with batch size 32 and 15 epochs (This take 15 - 20 minutes with the CPU)

```
In [11]: # Use one_hot_encoding to convert each label into a binary vector for training
y_train = make_the_one_hot_encoding(y_train)
y_val = make_the_one_hot_encoding(y_val)
```

[ONE HOT ENCODING] Labels are one-hot-encoded: True

[ONE HOT ENCODING] Labels are one-hot-encoded: True

```
In [12]: # Train model with epochs = 15, batch_size = 32
history = model.fit(X_train, y_train,
                    epochs=15, batch_size=32,
                    validation_data=(X_val, y_val))
```

Epoch 1/15


WARNING: All log messages before absl::InitializeLog() is called are written to STDERR


I0000 00:00:1739810695.139230 97294 service.cc:148] XLA service 0x7fe7b0003400 initialized for platform CUDA (this does not guarantee that XLA will be used). Devices:


I0000 00:00:1739810695.139397 97294 service.cc:156] StreamExecutor device (0): NVIDIA GeForce RTX 4070 Laptop GPU, Compute Capability 8.9


I0000 00:00:1739810695.800863 97294 cuda_dnn.cc:529] Loaded cuDNN version 90300


I0000 00:00:1739810722.551613 97294 device_compiler.h:188] Compiled cluster using XLA! This line is logged at most once for the lifetime of the process.


25/25  74s 2s/step - accuracy: 0.3720 - loss: 14.6692 - val_accuracy: 0.8107 - val_loss: 13.8367
Epoch 2/15


25/25  9s 359ms/step - accuracy: 0.8618 - loss: 13.5773 - val_accuracy: 0.8462 - val_loss: 13.2153
Epoch 3/15


25/25  9s 354ms/step - accuracy: 0.9307 - loss: 12.9868 - val_accuracy: 0.9379 - val_loss: 12.7895
Epoch 4/15


25/25  9s 355ms/step - accuracy: 0.9641 - loss: 12.6622 - val_accuracy: 0.9586 - val_loss: 12.5217
Epoch 5/15


25/25  9s 358ms/step - accuracy: 0.9892 - loss: 12.3807 - val_accuracy: 0.9497 - val_loss: 12.2903
Epoch 6/15


25/25  9s 356ms/step - accuracy: 0.9851 - loss: 12.1540 - val_accuracy: 0.9586 - val_loss: 12.0590
Epoch 7/15


25/25  9s 361ms/step - accuracy: 0.9932 - loss: 11.9206 - val_accuracy: 0.9556 - val_loss: 11.8442
Epoch 8/15


25/25  9s 365ms/step - accuracy: 0.9935 - loss: 11.6976 - val_accuracy: 0.9556 - val_loss: 11.6143
Epoch 9/15


25/25  9s 358ms/step - accuracy: 1.0000 - loss: 11.4687 - val_accuracy: 0.9586 - val_loss: 11.3955
Epoch 10/15


25/25  9s 359ms/step - accuracy: 1.0000 - loss: 11.2506 - val_accuracy: 0.9586 - val_loss: 11.1829
Epoch 11/15

25/25  10s 358ms/step - accuracy: 1.0000 - loss: 11.0373 - val_accuracy: 0.9645 - val_loss: 10.9808
Epoch 12/15

25/25  9s 357ms/step - accuracy: 1.0000 - loss: 10.8250 - val_accuracy: 0.9645 - val_loss: 10.7670
Epoch 13/15


25/25  9s 357ms/step - accuracy: 1.0000 - loss: 10.6169 - val_accuracy: 0.9556 - val_loss: 10.5649
Epoch 14/15

25/25  9s 358ms/step - accuracy: 1.0000 - loss: 10.4126 - val_accuracy: 0.9586 - val_loss: 10.3771
Epoch 15/15

25/25  9s 356ms/step - accuracy: 0.9993 - loss: 10.2117 - val_accuracy: 0.9497 - val_loss: 10.2903

Evaluate the model

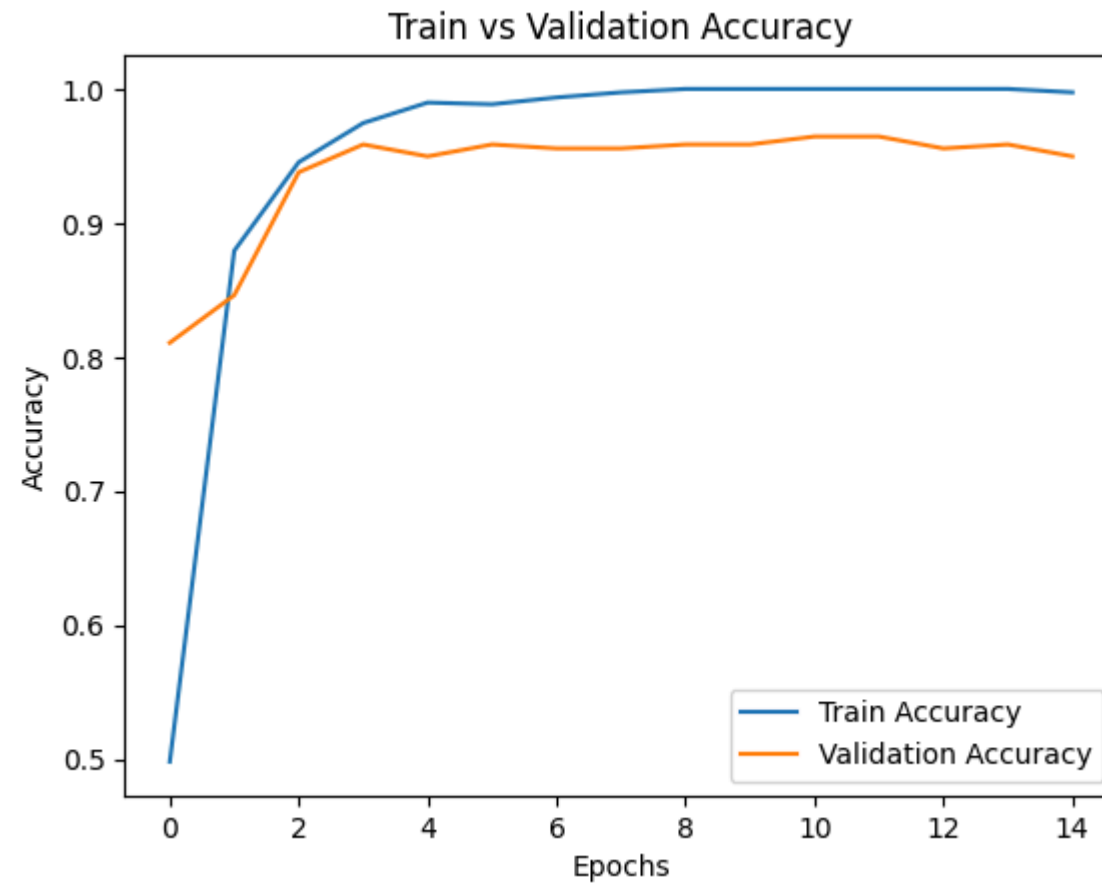
```
In [13]: test_loss, test_acc = model.evaluate(X_val, y_val)
         print(f"Accuracy: {test_acc*100:.2f}%")
```

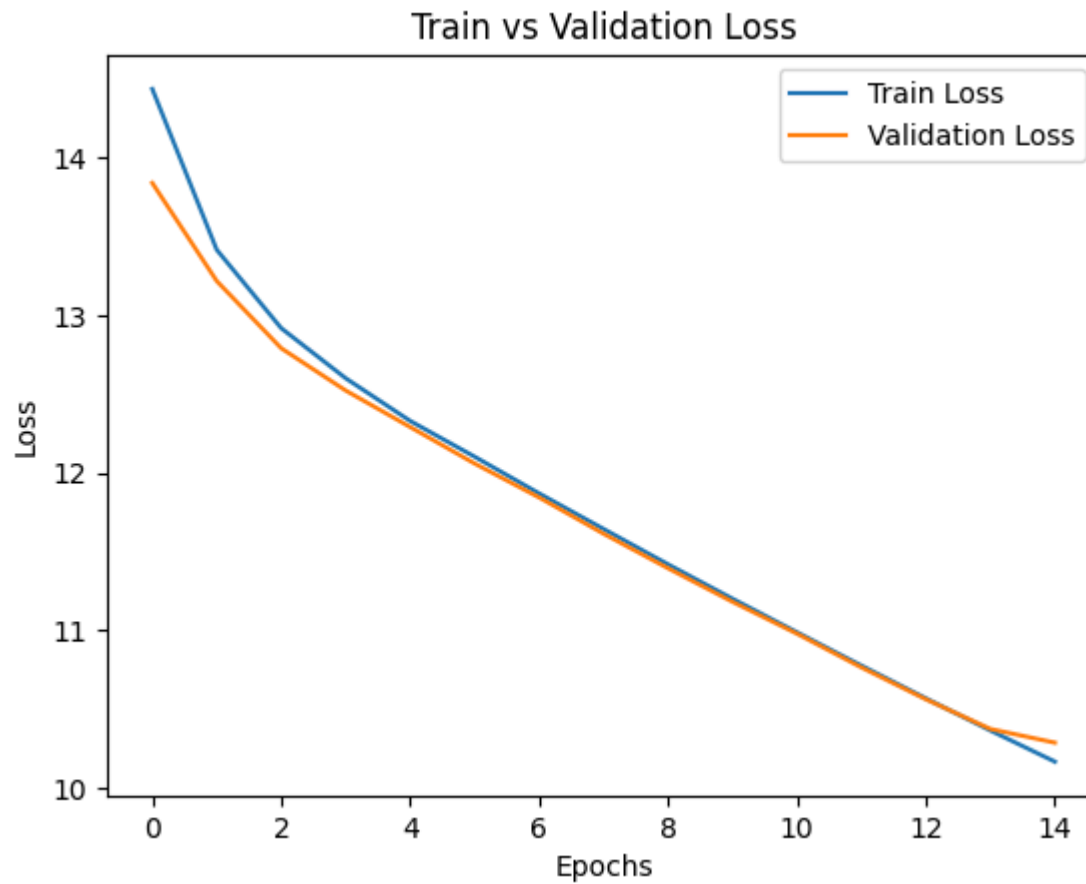
11/11  1s 84ms/step - accuracy: 0.9528 - loss: 10.2374
Accuracy: 94.97%

Draw a chart of the training process to check overfitting/underfitting

```
In [14]: # Plot accuracy graph
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Train vs Validation Accuracy')
plt.show()

# Plot Loss graph
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Train vs Validation Loss')
plt.show()
```





Train vs Validation Accuracy

- Trend: Accuracy increases gradually with the number of epochs
- High train accuracy (~100%): The model is learning well on the training set
- High validation accuracy (~95%): The model generates well
- Train-validation distance: The gap is not too far; it illustrates no signs of serious overfitting

Train vs Validation Loss

- Trend: Loss of both training set and validation set are gradually decreasing.
- Loss distance: The gap is quite close to each other, which shows the model is not overfitting.

- Loss reduction: There are no large fluctuations; providing that training is stable

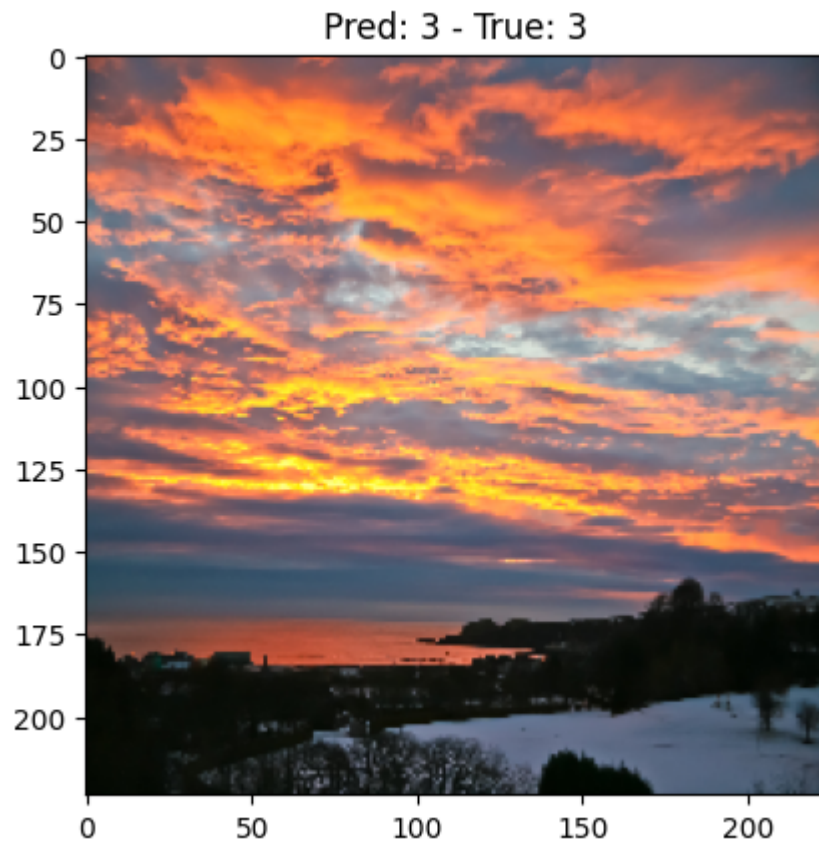
Make and show predictions

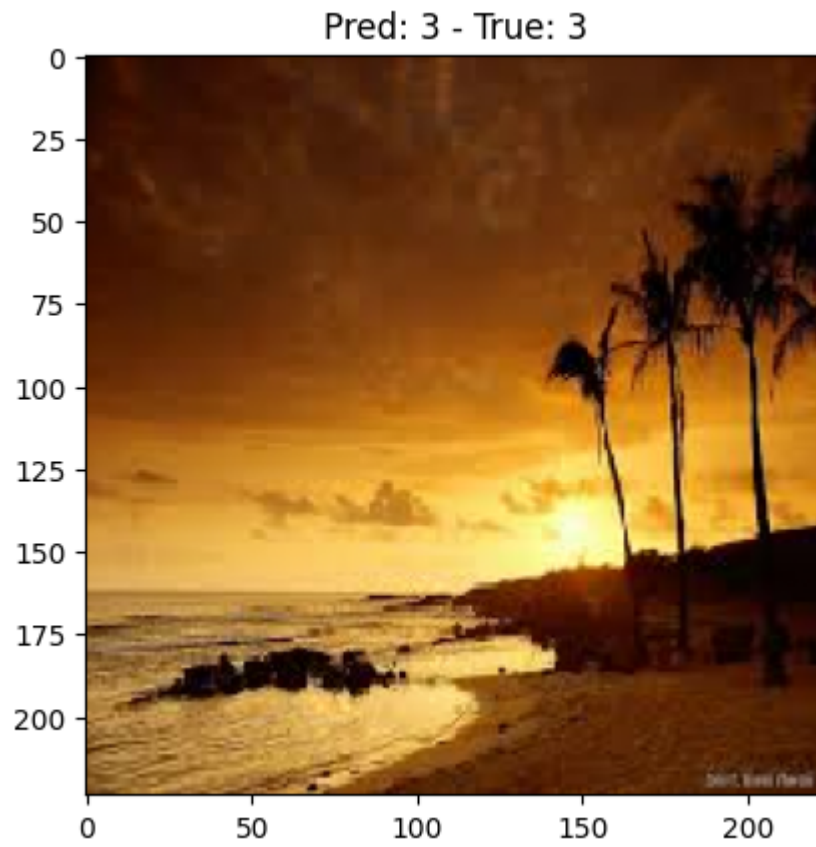
```
In [15]: # Chose random 5 samples from validation
num_samples = 5
indices = np.random.choice(len(X_val), num_samples, replace=False)
sample_images = X_val[indices]
sample_labels = y_val[indices]

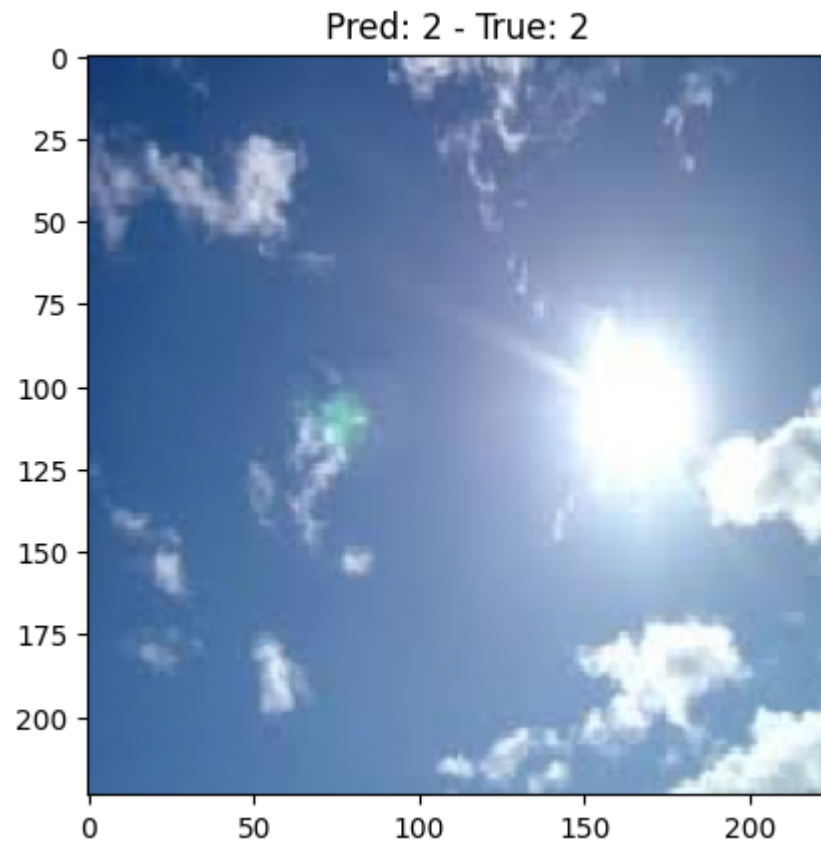
# Prediction
predictions = model.predict(sample_images)

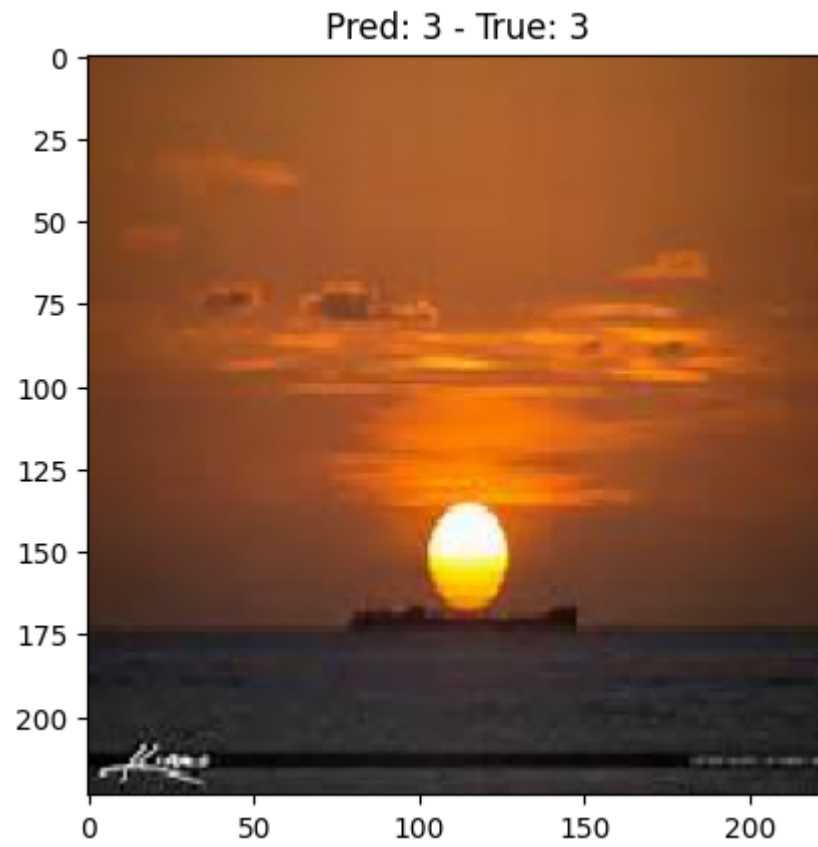
# Plot the result, 4 classes: ['Cloudy', 'Rain', 'Shine', 'Sunrise']
for i in range(num_samples):
    plt.imshow(sample_images[i])
    plt.title(f'Pred: {np.argmax(predictions[i])} - True: {np.argmax(sample_labels[i])}')
    plt.show()
```

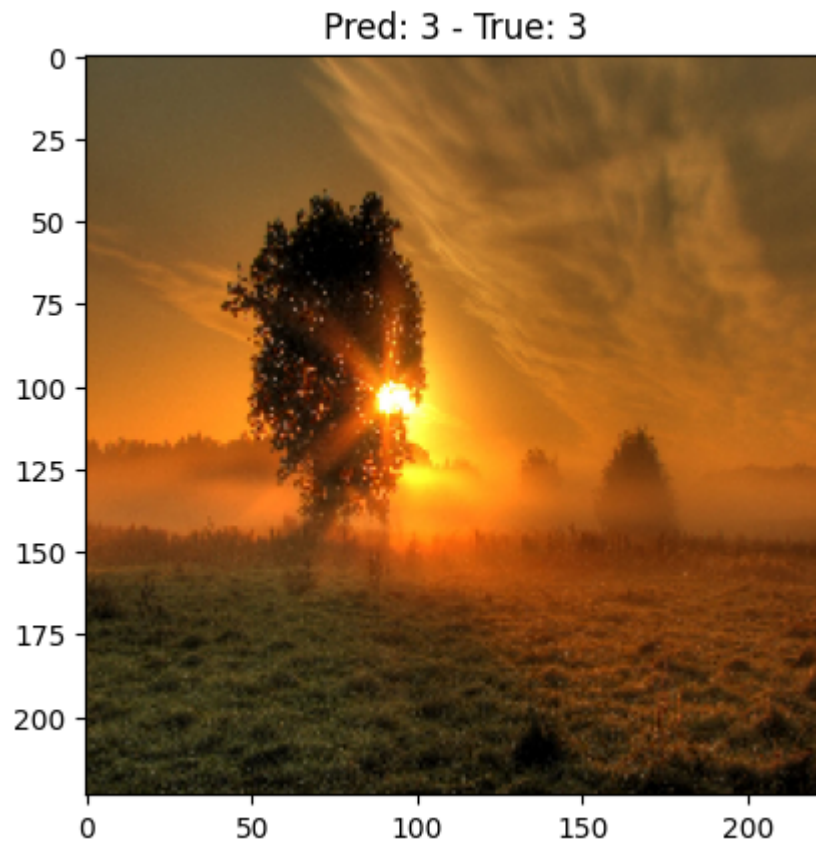
1/1 ————— 3s 3s/step











make confusion matrix

```
In [16]: # Predict the probabilities of classes on the validation set
y_pred_probs = model.predict(X_val)

# Convert to a Label by taking the value with the highest probability
y_pred = np.argmax(y_pred_probs, axis=1)

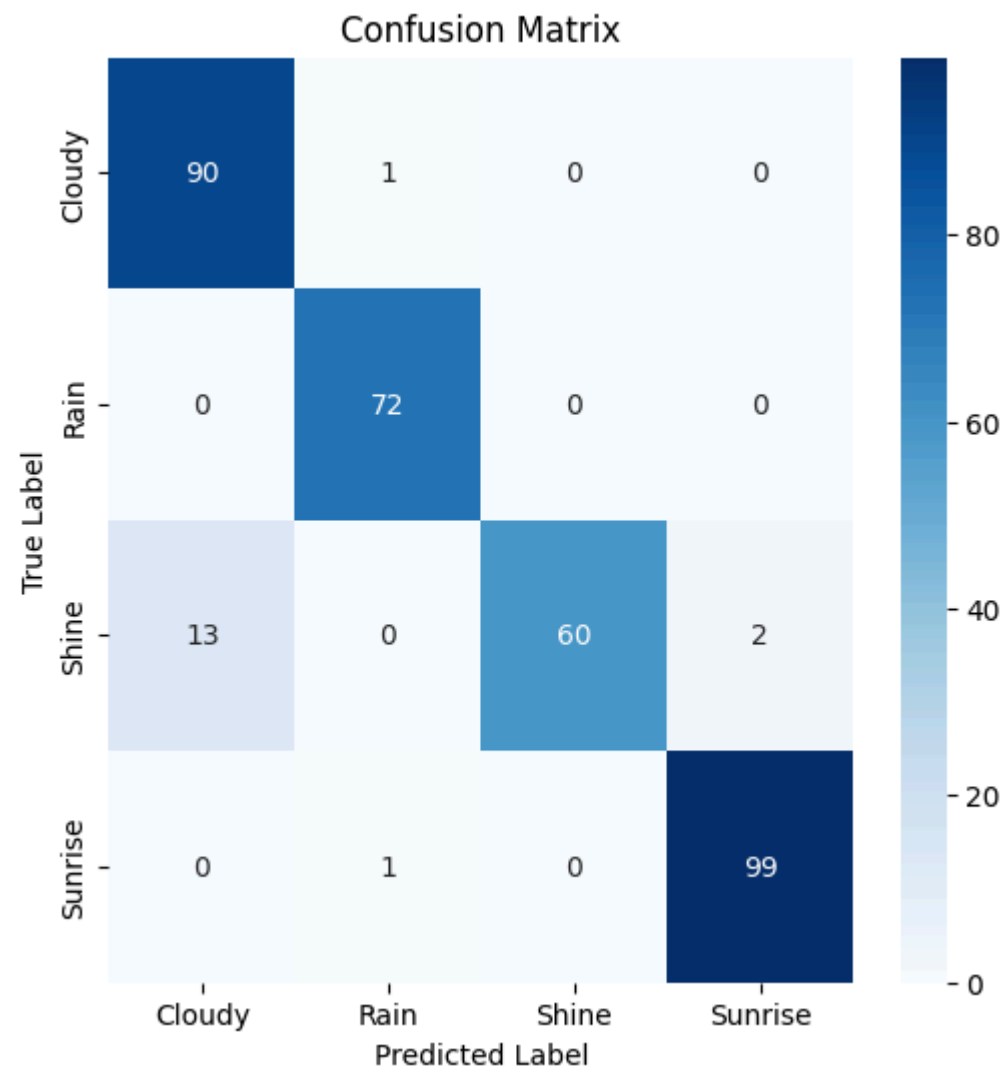
# Convert it back to Label format when y_val is in one-hot encoding
y_true = np.argmax(y_val, axis=1)

# Definition of class Labels
class_labels = ['Cloudy', 'Rain', 'Shine', 'Sunrise']
```

```
# Create confusion matrix
cm = confusion_matrix(y_true, y_pred)

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

11/11 ————— 2s 146ms/step



True Label / Predicted label	Cloudy	Rain	Shine	Sunrise
Cloudy	90	1	0	0
Rain	0	72	0	0
Shine	13	0	60	2

True Label / Predicted label	Cloudy	Rain	Shine	Sunrise
Sunrise	0	1	0	99

Main diagonal (90, 72, 60, 99) → True Positives

- 90 Cloudy photos were correctly predicted.
- 72 Rain photos were correctly.
- 60 Shine photos were correctly predicted.
- 99 Sunrise photos were correctly predicted.

Values off the main diagonal → False Positives

- 1 Cloudy photos were mistaken for Shine.
- 13 photos of Shine were mistaken for Cloudy.
- 2 photos of Shine were mistaken for Sunrise
- 1 Sunrise photos were mistaken for Shine.

Evaluate the model

- The correct prediction rate is high, especially with Rain (100% accurate) and Sunrise (~99% accurate).
- The Shine class has the most errors: 13 Shine photos mistaken for Cloudy → Maybe because Shine photos are sometimes cloudy.
- 2 Shine images were mistaken for Sunrise → Maybe the lighting may be the same between these two layers.
- A photo of Sunrise was mistaken for Rain → Maybe due to lighting conditions and unclear photo background.

Load again the cnn but this time set the parameters to NOT TRAINABLE

```
In [17]: baseModel = VGG19(input_shape=target_size, weights='imagenet', include_top=False) # Input model use VGG19 use imagenet weight

# Freeze all layers of VGG19
for layer in baseModel.layers:
    layer.trainable = False # Keep pre-trained weights intact, train quickly

# Flatten layer to convert from 4D tensor -> 1D vector
x = Flatten()(baseModel.output)
```

```
# Fully Connected Layers (Dense + Dropout)
x = Dense(512, activation='relu', kernel_regularizer=l1_l2(l1=0, l2=0.01))(x) # Add a Dense layer with 512 units
x = Dropout(0.1)(x) # Dropout layer with 0.1 unit
x = Dense(256, activation='relu', kernel_regularizer=l1_l2(l1=0, l2=0.01))(x) # Add a Dense layer with 256 units
x = Dropout(0.1)(x) # Dropout layer with 0.1 unit

# Output layer with 4 classes for the final classifier
x = Dense(4, activation='softmax')(x) # 4 classes: ['Cloudy', 'Rain', 'Shine', 'Sunrise']

# Combine base_model and Fully Connected layers into a final model
model = Model(inputs=baseModel.input, outputs=x)

model.summary() # Print model summary
```

Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv4 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv4 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808

block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv4 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten_1 (Flatten)	(None, 25088)	0
dense_3 (Dense)	(None, 512)	12,845,568
dropout_2 (Dropout)	(None, 512)	0
dense_4 (Dense)	(None, 256)	131,328
dropout_3 (Dropout)	(None, 256)	0
dense_5 (Dense)	(None, 4)	1,028

Total params: 33,002,308 (125.89 MB)

Trainable params: 12,977,924 (49.51 MB)

Non-trainable params: 20,024,384 (76.39 MB)

Fit the model with batch size 32 and 15 epochs (This is faster)

```
In [18]: # Compile model with Adam
model.compile(
    optimizer=Adam(learning_rate=0.00001), # Use low learning rate to avoid overfitting
    loss='categorical_crossentropy', # Use categorical_crossentropy for multi-class classification
    metrics=['accuracy']
)

# Train model with epochs = 15, batch_size = 32
history = model.fit(X_train, y_train,
                    epochs=15, batch_size=32,
                    validation_data=(X_val, y_val))
```



```

Epoch 1/15
25/25 ————— 14s 470ms/step - accuracy: 0.3375 - loss: 14.7277 - val_accuracy: 0.5680 - val_loss: 14.2676
Epoch 2/15
25/25 ————— 3s 129ms/step - accuracy: 0.5890 - loss: 14.1418 - val_accuracy: 0.6982 - val_loss: 13.7775
Epoch 3/15
25/25 ————— 3s 128ms/step - accuracy: 0.7248 - loss: 13.6287 - val_accuracy: 0.7515 - val_loss: 13.3407
Epoch 4/15
25/25 ————— 3s 128ms/step - accuracy: 0.7594 - loss: 13.2187 - val_accuracy: 0.7544 - val_loss: 12.9535
Epoch 5/15
25/25 ————— 3s 128ms/step - accuracy: 0.8190 - loss: 12.8080 - val_accuracy: 0.7751 - val_loss: 12.6085
Epoch 6/15
25/25 ————— 3s 127ms/step - accuracy: 0.8824 - loss: 12.4220 - val_accuracy: 0.7781 - val_loss: 12.2931
Epoch 7/15
25/25 ————— 3s 127ms/step - accuracy: 0.8476 - loss: 12.1204 - val_accuracy: 0.8018 - val_loss: 11.9757
Epoch 8/15
25/25 ————— 3s 131ms/step - accuracy: 0.9161 - loss: 11.7431 - val_accuracy: 0.8225 - val_loss: 11.6876
Epoch 9/15
25/25 ————— 3s 129ms/step - accuracy: 0.9012 - loss: 11.5023 - val_accuracy: 0.8254 - val_loss: 11.4137
Epoch 10/15
25/25 ————— 3s 128ms/step - accuracy: 0.9070 - loss: 11.2463 - val_accuracy: 0.8521 - val_loss: 11.1487
Epoch 11/15
25/25 ————— 3s 127ms/step - accuracy: 0.9033 - loss: 10.9664 - val_accuracy: 0.8609 - val_loss: 10.8921
Epoch 12/15
25/25 ————— 3s 126ms/step - accuracy: 0.9365 - loss: 10.7109 - val_accuracy: 0.8491 - val_loss: 10.6579
Epoch 13/15
25/25 ————— 3s 125ms/step - accuracy: 0.9499 - loss: 10.4590 - val_accuracy: 0.8698 - val_loss: 10.4197
Epoch 14/15
25/25 ————— 3s 126ms/step - accuracy: 0.9509 - loss: 10.2308 - val_accuracy: 0.8757 - val_loss: 10.1909
Epoch 15/15
25/25 ————— 3s 125ms/step - accuracy: 0.9570 - loss: 9.9978 - val_accuracy: 0.8757 - val_loss: 9.9787

```

Evaluate the model

```

In [19]: test_loss, test_acc = model.evaluate(X_val, y_val)
         print(f"Accuracy: {test_acc*100:.2f}%")

```

```

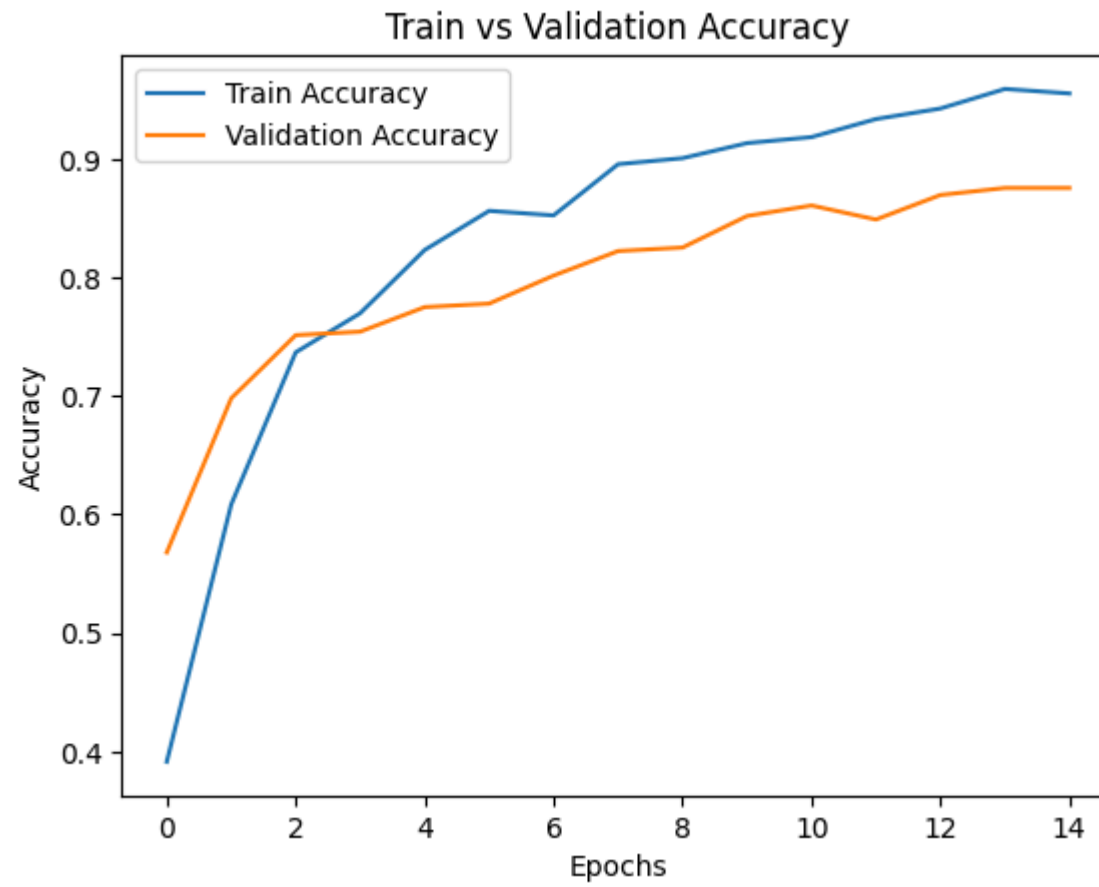
11/11 ————— 1s 84ms/step - accuracy: 0.8814 - loss: 9.9909
Accuracy: 87.57%

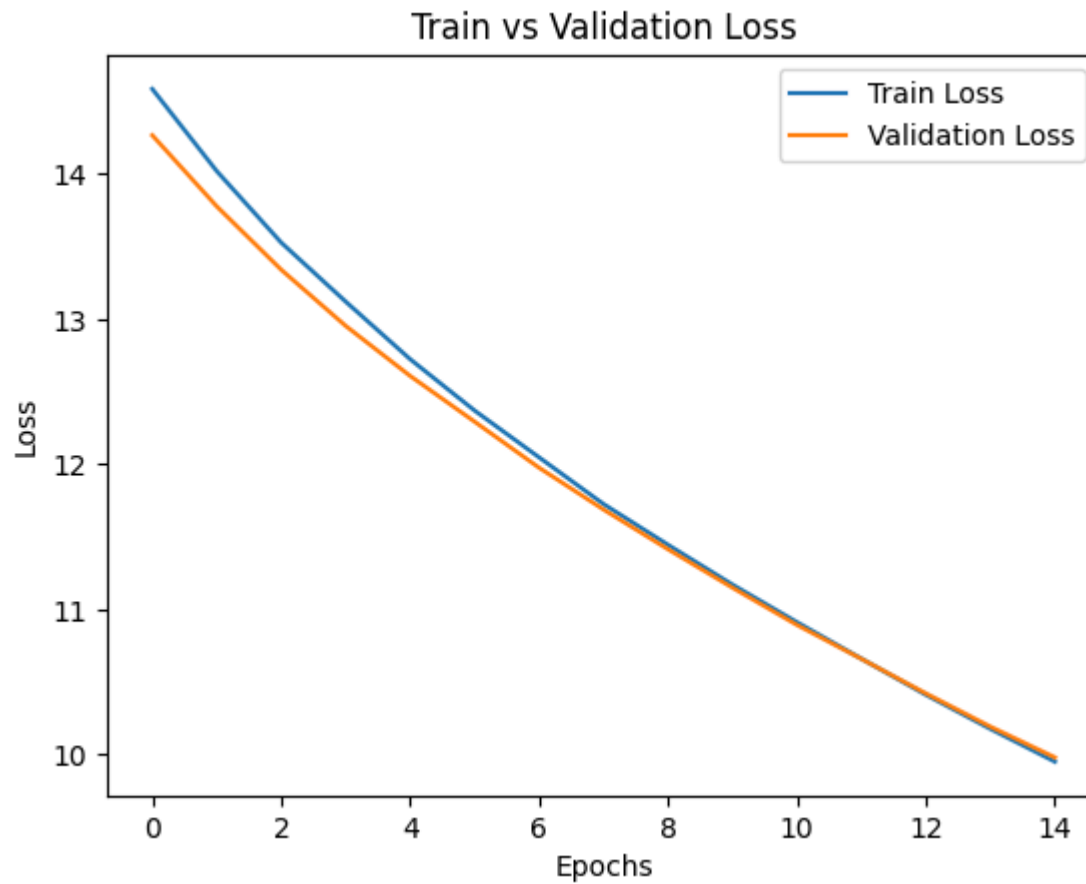
```

Draw a chart of the training process to check overfitting/underfitting

```
In [20]: # Plot accuracy graph
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Train vs Validation Accuracy')
plt.show()

# Plot Loss graph
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Train vs Validation Loss')
plt.show()
```





Train vs Validation Accuracy

- Trend: Accuracy increases gradually with the number of epochs
- High train accuracy (~98%): The model is learning well on the training set
- High validation accuracy (~85%): The model generates well
- Train-validation distance: The gap is not too far; it illustrates no signs of serious overfitting

Train vs Validation Loss

- Trend: Loss of both training set and validation set are gradually decreasing.
- Loss distance: The gap is quite close to each other, which shows the model is not overfitting.

- Loss reduction: There are no large fluctuations; providing that training is stable

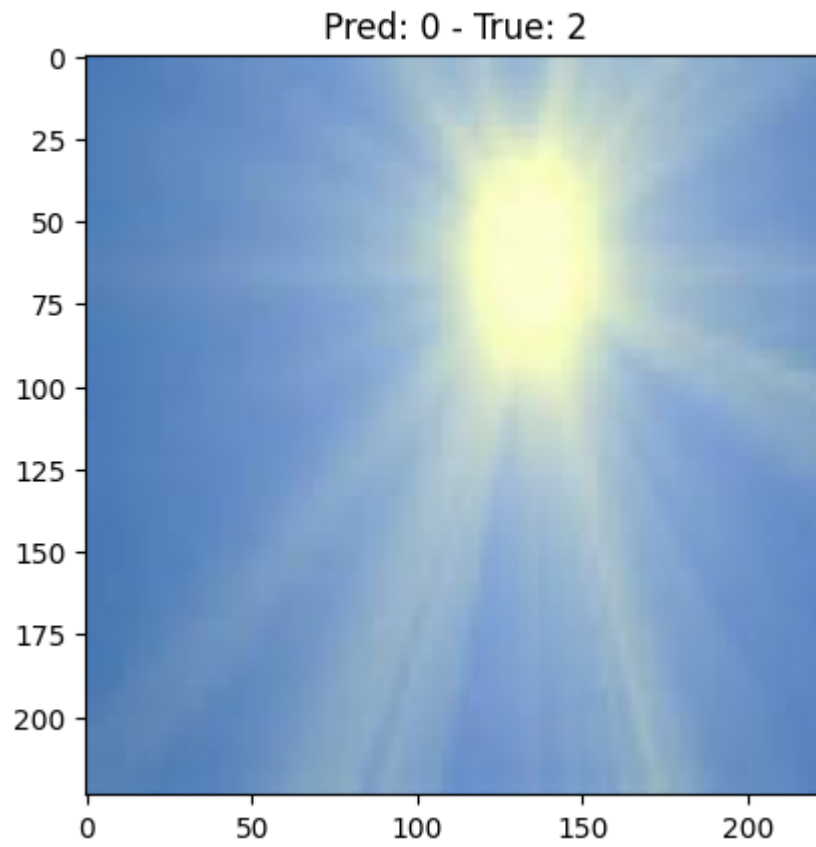
Make and show some predictions

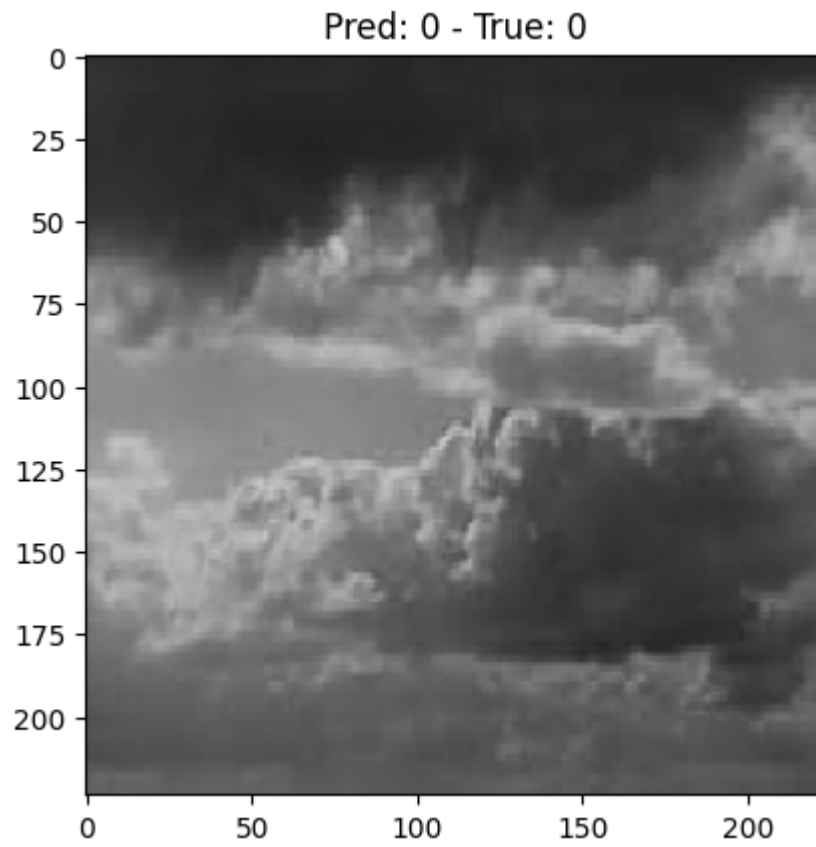
```
In [21]: # Chose random 5 samples from validation
num_samples = 5
indices = np.random.choice(len(X_val), num_samples, replace=False)
sample_images = X_val[indices]
sample_labels = y_val[indices]

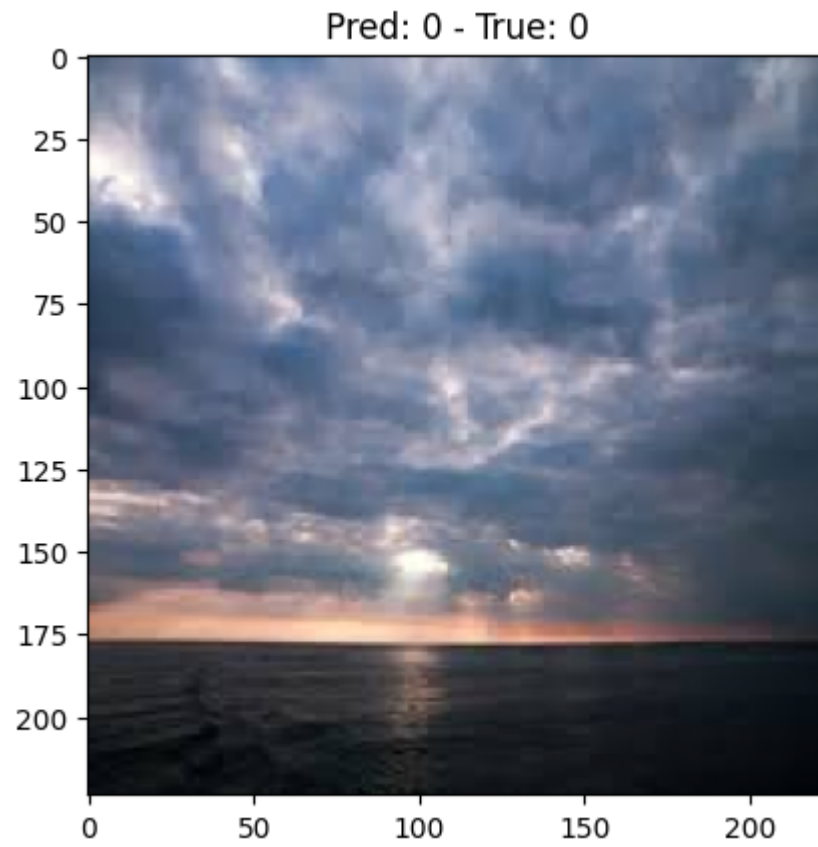
# Prediction
predictions = model.predict(sample_images)

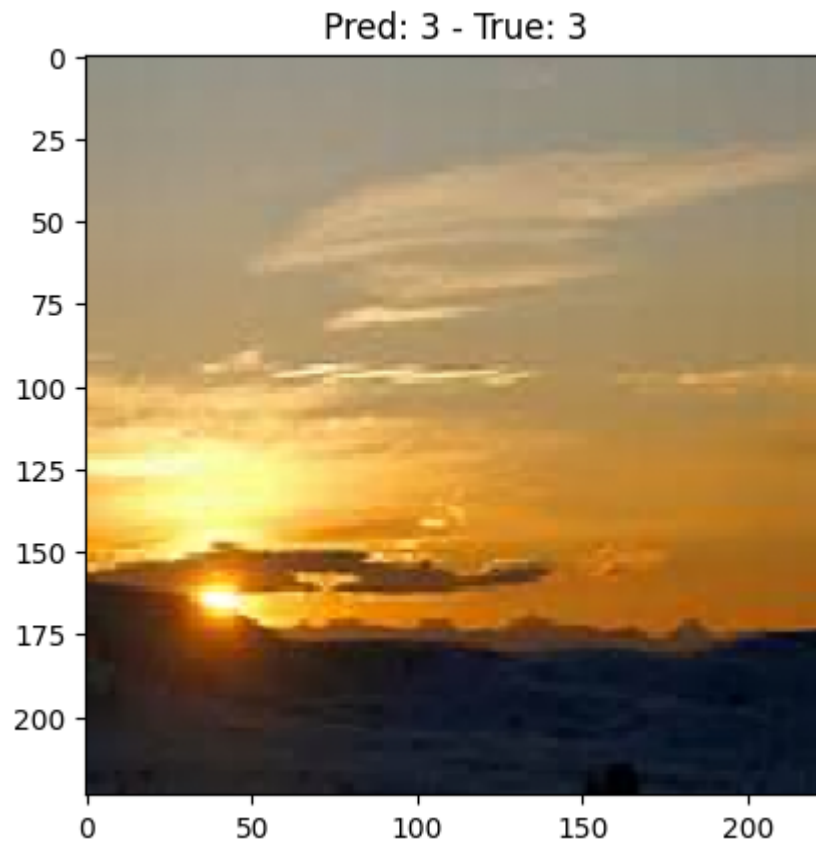
# Plot the result, 4 classes: ['Cloudy', 'Rain', 'Shine', 'Sunrise']
for i in range(num_samples):
    plt.imshow(sample_images[i])
    plt.title(f'Pred: {np.argmax(predictions[i])} - True: {np.argmax(sample_labels[i])}')
    plt.show()
```

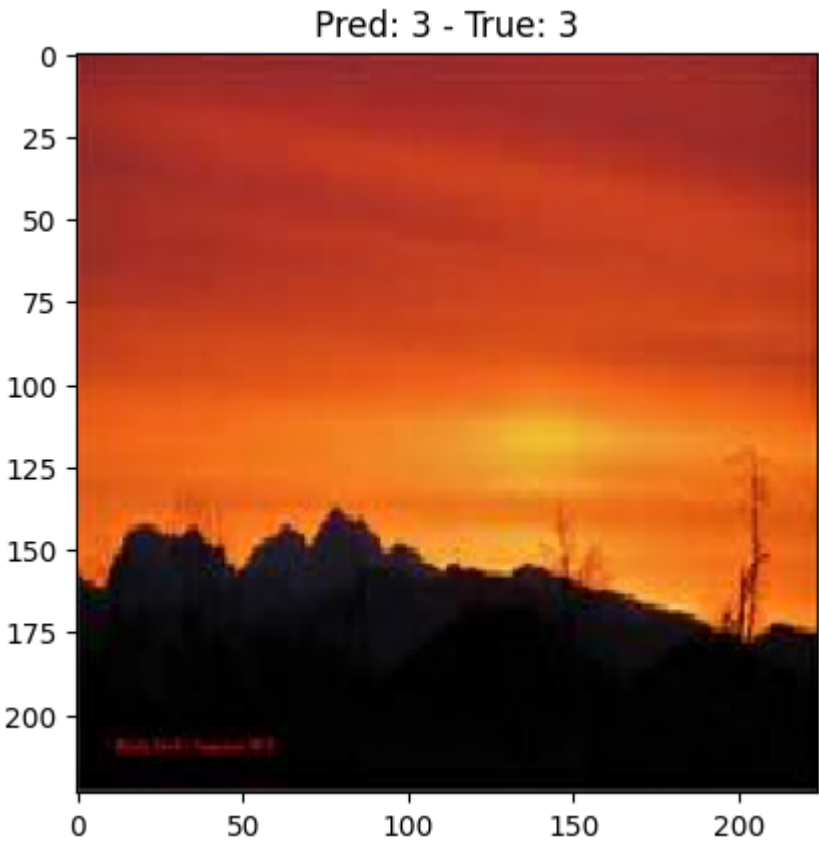
1/1 ————— 1s 547ms/step











Compare two models

Method	Train Accuracy	Validation Accuracy	Test Accuracy	Overfitting
Train all layers (trainable = True)	100%	94.97%	94.97%	There are signs of overfitting (Train is 5% higher than Validation)
Freeze VGG19 (trainable = False)	95.7%	87.57%	87.57%	Less overfitting, but less learning ability

Trainable model = True (100% train - 94.97% validation)

- When setting `layer.trainable = True`, the entire VGG19 is retrained from scratch, including the weights of the convolutional layers.
- It helps the model learn better on the training set, helping to increase the highest accuracy, but there is also a higher risk of overfitting if the data is not diverse enough.
- The model learns very well on the training set (reaches 100% accuracy). High validation accuracy (94.97%), good generalization model.
- Problem: Slight overfitting, train is too high (100%) while validation is only 94.97%. It takes a long time to train, because all the weights of the model have to be updated, requiring more GPU resources.

Trainable = False (95.7% train - 87.57% validation)

- When freezing the entire VGG19, only the fully connected layers behind are trained.
- It helps avoid overfitting, but limits learning because the model only uses existing features from ImageNet, without further refinement.
- Train is faster, because it only trains fully connected layers. Validation accuracy is stable (87.57%), although lower than `trainable = True`.
- Problem: Mild underfitting, because the model doesn't learn the best for the data. Accuracy on test is lower (~87.57%), because the model only uses pre-trained features.

Conclusion

Train Method	When to be used?
Trainable = True	When there is a large data set, it helps the model learn more optimally. Needs a more powerful GPU to train.
Trainable = False	When the dataset is small, avoid overfitting but may suffer from underfitting. Train faster.