Homework 7

Alexandre Olivé Pellicer
aolivepe@purdue.edu

1 Theory Question

In the theory from the lectures we have seen that some of the feature extractors used for image classification are based on neural networks such as VGG or ResNet. The feature map obtained at the output of these neural networks is used to compute the Gram matrix and, after some operations, end up obtaining a feature vector that is used to classify images. As we will see later in the report, neural network based approaches outperform traditional approaches such as LBP. Therefore, our proposed approach will be also based on a neural network.

First a neural network will create a feature map. Later, some operations will be applied on the feature map in order to obtain the feature vector that will be used to classify images. For the neural network we would use a combination of a transformer and a ResNet. A transformer is a deep learning architecture originally designed for natural language processing (NLP) tasks which uses self-attention to focus on relationships between all elements in a sequence simultaneously. In this proposed approach we would use the pre-trained Vision Transformer (ViT) which is adapted to image processing by dividing an image into smaller patches and treating each patch like a sequence in NLP. ViT captures global relationships across an image, making it effective for tasks that benefit from contextual, non-local feature extraction. For the ResNet, we would use the Fine mode which has been introduced in this homework.

The reason of using the combination of ResNet Fine and ViT is the following: the ResNet Fine is effective at capturing fine, local features, while ViT is better at understanding global relationships across an image thanks to it self-attention mechanism. Therefore, the combination of both can yield to a richer feature representation, leveraging the strengths of each model for improved classification performance.

Once obtained the feature map from the ViT and from the ResNet, we would reshape them. For examlpe, from the feature map obtained with the ViT, if it is of shape (c, h, w) we would reshape it to $(c, h \times w)$. Same for the feature map obtained with the ResNet. Then, both matrices would be concatenated in the channel dimension. Now, instead of computing the Gram matrix as the dot product of the resulting matrix from the concatenation with its transpose, we would compute the covariance matrix. Covariance matrices are a powerful alternative to Gram matrices, as they capture not only the relationships between different feature channels but also take into account their variances. This can make covariance matrices more sensitive to texture details, especially in complex textures, as they encode both intensity and interaction between features. The sensitivity to variance allows covariance matrices to represent a broader range of stylistic features. It would be particularly useful for complex images with intricate textures or patterns that might be lost using only the Gram matrix.

Given the matrix A resulting of the concatenation of the 2 reshaped feature maps, the covariance matrix can be computed as:

$$\text{Cov}(A) = \frac{1}{h \times w - 1} \cdot (A - \bar{A})(A - \bar{A})^T, \tag{1}$$

where $\bar{A}$ is the mean of A along each channel.

Since $\text{Cov}(A)$ is a symmetric matrix, we would take the upper triangular part (including the diagonal), flatten it and apply some dimensionality reduction technique to end up getting the feature vector of the image that would be used for classification. For the dimensionality reduction different subsampling techniques or Principal Component Analysis (PCA) can be used.

Overall, we expect a good performance of our approach because we are combining the strengths of ViT and ResNet Fine as feature map generators where the ResNet Fine is effective at capturing fine,

local features, while ViT is better at understanding global relationships. Furthermore, we compute covariance matrices instead of Gram matrices from the concatenation of the feature maps generated by each network because besides considering the relationships between different feature channels, it also takes into account their variances. Algorithm 1 shows the pseudo-code of the proposed approach.

Algorithm 1 Pseudo-code of the propose feature vector extractor method

procedure FEATUREVECTOR
  $F_{resnet} \leftarrow$ Extract feature map using ResNet($image$)
  $F_{vit} \leftarrow$ Extract feature map using ViT($image$)
  $A_{resnet} \leftarrow$ Reshape($F_{resnet}, (c_{resnet}, h \times w)$)
  $A_{vit} \leftarrow$ Reshape($F_{vit}, (c_{vit}, h \times w)$)
  $A \leftarrow$ Concatenate($A_{resnet}, A_{vit}, \text{axis} = 0$)
  $Cov \leftarrow$ ComputeCovarianceMatrix(A)
  $Cov_{tri} \leftarrow$ UpperTriangular(Cov)
  $F_{flat} \leftarrow$ Flatten(Cov_{tri})
  $F_{vector} \leftarrow$ PCA($F_{flat}, n_{components}$)
  return F_{vector}
end procedure

2 Description of Implementations

In this section we describe how we have used 4 different feature vector extractors used for image classification: LBP, VGG, ResNet-fine, and ResNet-coarse. We also describe the approach proposed in the "Extra Credit" section of the instructions about channel normalization. Finally, we describe the classifier used to classify the images according to their class.

2.1 Local Binary Pattern (LBP)

For this approach we obtain a texture feature vector from the hue component of the images. We implement the code to obtain the hue component of the image. We obtain the HSI representation from the RGB channels by:

- **Hue (H):** Computed as the angle that quantifies the relationship between the red component and the mean of the green and blue components. This angle is normalized to fall within the range $[0, 1]$.
- **Saturation (S):** Derived based on its deviation from intensity. It is computed by taking the normalized difference from the lowest RGB channel value.
- **Intensity (I):** Calculated as the mean of RGB values.

The Local Binary Pattern (LBP) descriptor is utilized to extract texture features from images based on their hue component. The process is detailed as follows:

The LBP method is controlled by 2 parameters:

- **Radius (R):** Radius of the circle in which we will set a specific number of points. After experimenting with different parameters, the best performance has been achieved when setting R=1.
- **Number of Points (P):** Number of points in the circle. After experimenting with different parameters, the best performance has been achieved when setting P=8.

For the LBP method we resize the images to be of size 64×64. The method consists in iterating over all the pixels of the hue component of the input images. For each pixel, a circle of radius R is set around it and P points are placed in the perimeter of the circle. The value for the P is computed through bilinear interpolation. A binary pattern is created by thresholding the value of each of the P points compared to the value of the actual pixel in the center of the circle. Finally, the BitVector module is used to generate and encode the LBP histograms, which work as the feature vectors used for classification.

2.2 VGG

- We use the pre-trained VGG-19 model to generate a feature map for each image. The activations from the layers of the VGG-19 model have been previously extracted in order to obtain the feature maps.
- The input images to the VGG-19 model have been resized to 256 $\times$ 256.
- The obtained feature map is reshaped to shape $(c, h \times w)$. The Gram matrix is obtained by the dot product of the reshaped feature map and its transpose.
- Since the Gram matrix is symmetric, we take the upper triangular elements and flatten it. We previously resized the Gram matrix to 32 $\times$ 32. The obtained flattened vector is the feature vector used for classification.

2.3 ResNet-50 "Fine" and "Coarse" Mode

- We use the pre-trained ResNet-50 model to generate a feature map for each image. The ResNet model has been truncated so that a feature map will be obtained the deeper layers of the network (fine mode) and another feature map will be obtained from the earlier layers (coarse mode).
- In coarse mode, the feature map captures lower-level, broad details like edges, textures and color patterns.
- In fine mode, the feature map captures higher-level, fine-grained details that are more abstract, like object parts and complex shapes.
- The input images to the ResNet model have been resized to 256 $\times$ 256.
- Once obtained the feature map, we compute the Gram matrix and obtain the feature vector from it as explained in Section 2.2.

2.4 Channel Normalization (Extra Credit)

In the "Extra Credit" section of the instructions an alternative to Gram matrices is proposed to obtain the feature vectors. This consists in channel-wise normalization of feature maps using Adaptive Instance Normalization (AdaIN).

- Feature maps are normalized by adjusting the mean and standard deviation in each channel of the reshaped feature map.
- The feature vector is created by the concatenation of the mean and variance of each channel.
- The classification performance is expected to improve using this method since it enhances resilience to color and lighting variations across different texture classes.

2.5 SVM classifier

We implement the SVM classifier using sklearn.

- We use the $fit()$ function to train the classifier with the feature vectors of the training dataset and the labels
- We use the $predict()$ function to get the predictions of the feature vectors of the testing dataset
- We show a quantitative evaluation of the results in confusion matrices and accuracies

3 Obtained Results

In this section we show the obtained results from the implementations explained in the previous section

3.1 LBP Histograms

Figure 1 shows and example of the LBP histogram computed for an image of each of the 4 classes present in the given dataset.

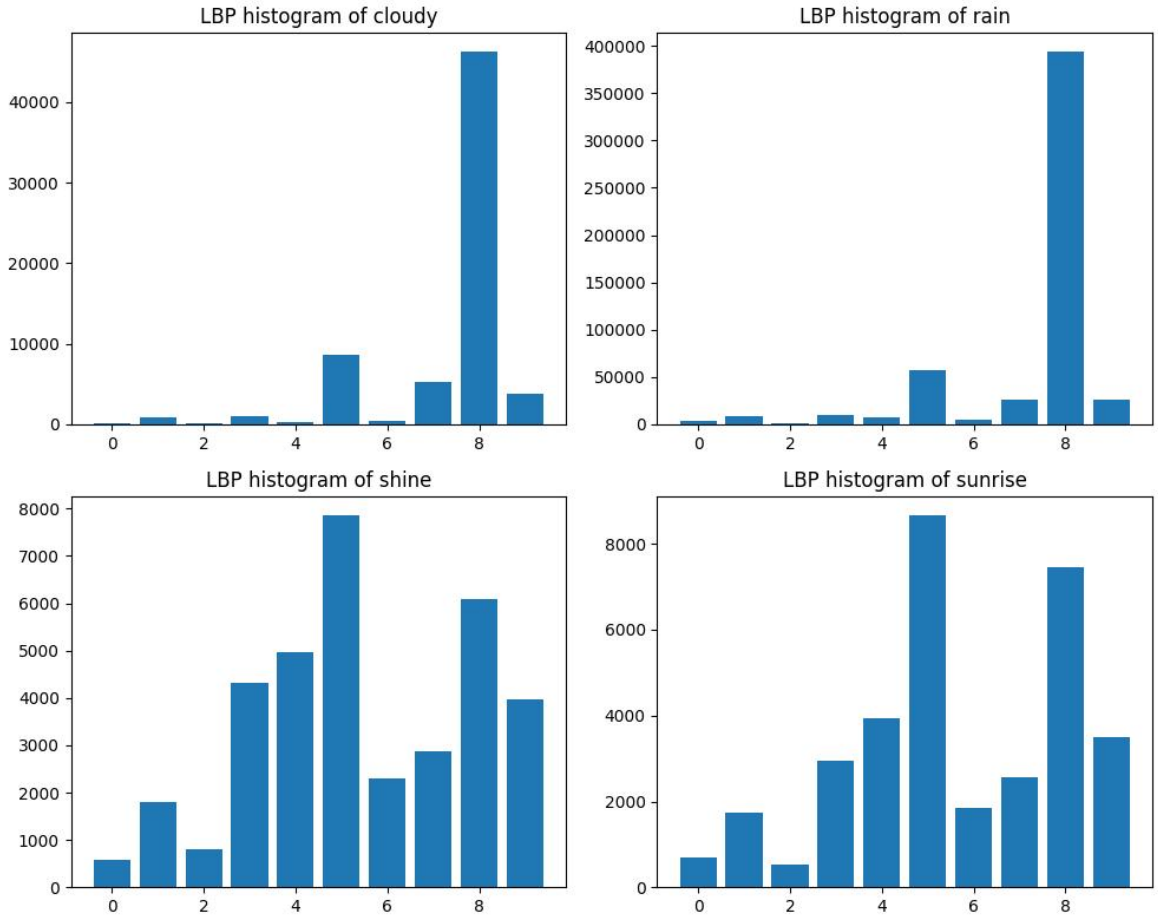


Figure 1: Examples of LBP histograms for each class of the given dataset

3.2 Gram Matrices

Figures 2, 3 and 4 show the Gram matrices obtained with each of the feature vector extractors for each of the four classes of the given dataset. The yellowish, the more correlated are the feature channels. The blueish, the less correlated the channels are. As we can see, the correlation between feature channels is different for each class. This follow our expectations since this difference will be reflected in the feature vectors and therefore, the classifier will learn them and be able to distinguish between classes. We can also see that in all Gram matrices the diagonal is bright. This makes sense since it is saying that each feature channel is highly correlated with itself. Note: in order to visualize the matrices properly, we applied the log scales and adjusted manually the colors for the legend bar. Otherwise, the plots were insignificant since they looked very dark.

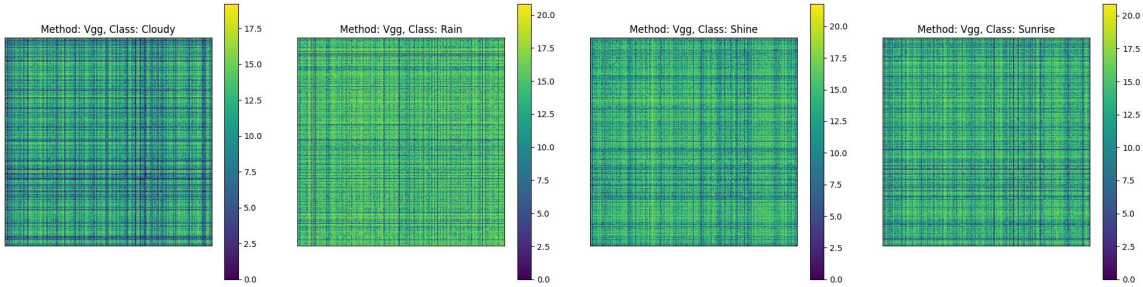


Figure 2: Gram Matrices obtained from the feature map generated by VGG for an image from each of the classes in the dataset.

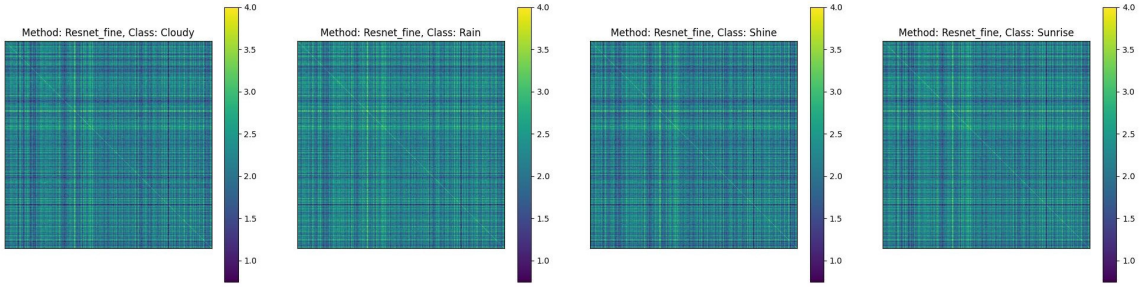


Figure 3: Gram Matrices obtained from the feature map generated by ResNet in fine mode for an image from each of the classes in the dataset.

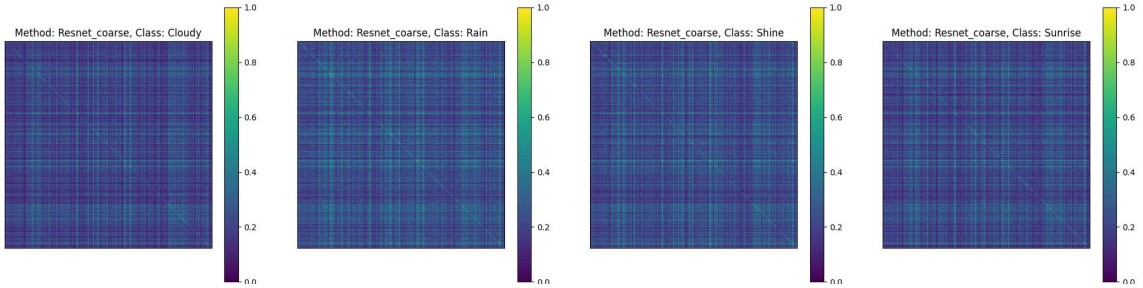
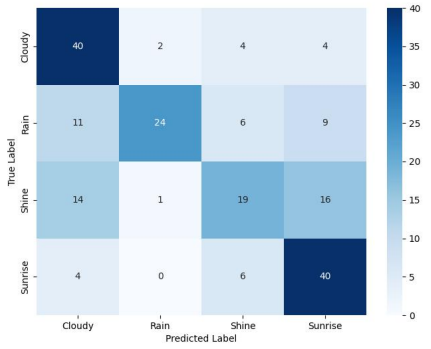


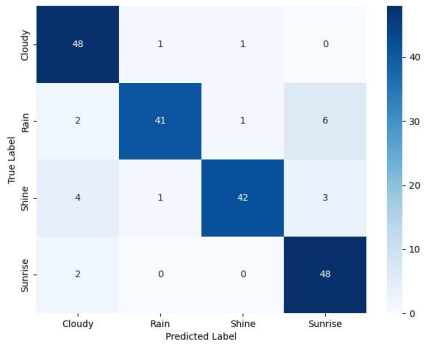
Figure 4: Gram Matrices obtained from the feature map generated by ResNet in coarse mode for an image from each of the classes in the dataset.

3.3 Confusion Matrices and Accuracies

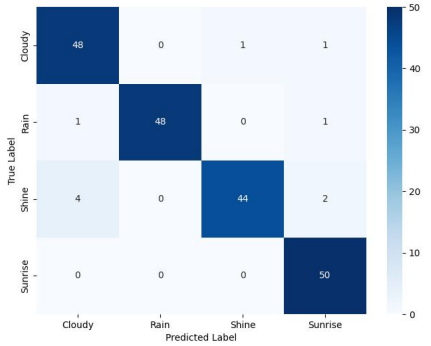
In this section we have used the Gram matrix approach to get the feature vectors of the images given the feature maps generated by each of the neural networks used in this report. For the LBP approach we used the computed histograms as feature vectors. Figure 5 shows the confusion matrices for each of the approaches. Table 1 shows the accuracy results.



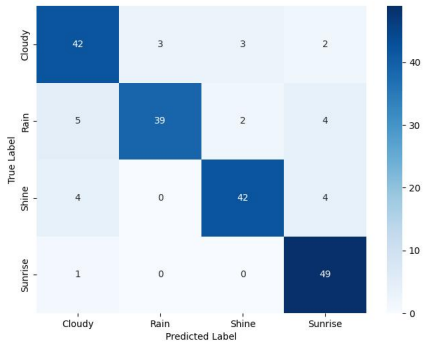
(a) LBP confusion matrix



(b) VGG confusion matrix



(c) ResNet Fine confusion matrix



(d) ResNet Coarse confusion matrix

Figure 5: Confusion matrices for each of the feature vector extractors used

Method	Accuracy
LBP	61.5%
VGG	89.5%
ResNet Fine	95.0%
ResNet Coarse	86.0%

Table 1: Accuracy for each of the methods used

3.4 Classification Results

Figure 6 shows examples of correctly and incorrectly classified images for each feature vector generator approach. Each row corresponds to a different feature vector generator. The first column has examples of correct classification and the second column has examples of incorrect classification.

(a) LBP correct (GT: sunrise, Pred: sunrise)

(b) LBP incorrect (GT: shine, Pred: sunrise)

(c) VGG correct (GT: cloudy, Pred: cloudy)

(d) VGG incorrect (GT: cloudy, Pred: rain)

(e) ResNet Fine correct (GT: sunrise, Pred: sunrise)

(f) ResNet Fine incorrect (GT: rain, Pred: sunrise)

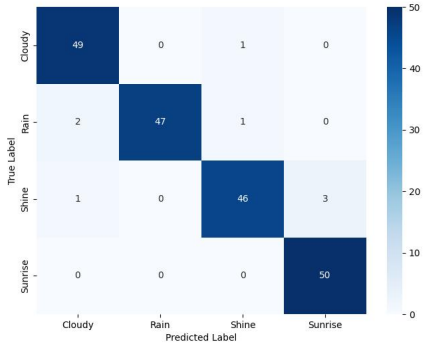
(g) ResNet Coarse correct (GT: rain, Pred: rain)

(h) ResNet Coarse incorrect (GT: shine, Pred: sunrise)

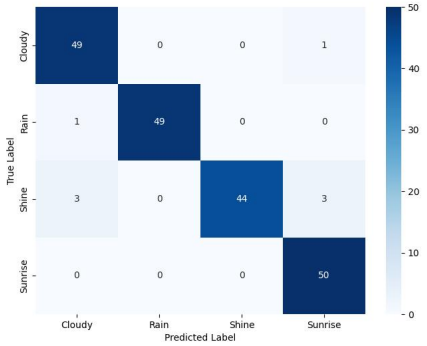
Figure 6: Examples of correctly and incorrectly classified images for each feature vector generator approach. Each row corresponds to a different feature vector generator. The first column has examples of correct classification and the second column has examples of incorrect classification

3.5 Extra Credit - Confusion Matrices and Accuracies

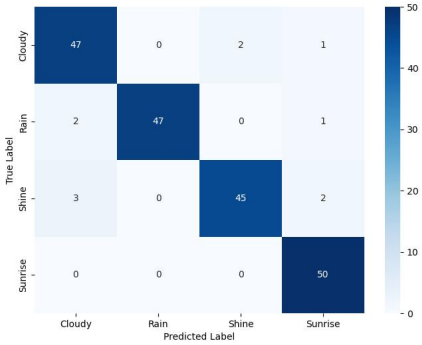
In this section we have used the AdaIN approach described in Section 2.4 to get the feature vectors of the images given the feature maps generated by each of the neural networks used in this report. Figure 7 shows the confusion matrices for each of the approaches. Table 2 shows the accuracy results.



(a) VGG confusion matrix



(b) ResNet Fine confusion matrix



(c) ResNet Coarse confusion matrix

Figure 7: Confusion matrices for each of the feature vector extractors used with the AdaIN approach.

Method	Accuracy
VGG	96%
ResNet Fine	96%
ResNet Coarse	94.5%

Table 2: Accuracy for each of the methods used with the AdaIN approach.

4 Observations

4.1 Local Binary Pattern (LBP)

The worst classification performance is achieved when using the Local Binary Pattern (LBP) approach. Is the one that gives lower accuracy. Looking at the LBP histograms plotted in Section 3.1, we see that the LBP histograms for the "cloudy" and "rain" classes are very similar. The same happens with the "shine" and "sunrise" classes. This behavior makes sense since in "cloudy" and "rain" images the predominant colors tend to be black-dark blue while in the "shine" and "sunrise" images the predominant colors tend to be orange-yellow-light blue. The fact that the LBP histograms are similar, decreases the classification performance since it is more difficult to set a difference between classes. In fact, looking at the confusion matrices we can see that the incorrect classifications tend to be for the classes that we have mentioned that are similar. In Section 3.4 we can see how a "shine" image is incorrectly classified as "sunrise".

4.2 VGG

The VGG approach significantly outperforms the LBP approach. From this result and from what we will see in the ResNet approaches, we can conclude that deep learning models can better capture complex information and create feature vectors with more relevant information that will help the classifier to perform better. Specifically, CNNs (VGG and ResNet are CNNs) have a hierarchical architecture that allows them to simple features from the lower layers and more complex features from the deeper layers. Traditional methods like LBP can capture local textures but lock the hierarchical representation, which limits their ability to represent complex patterns.

4.3 ResNet-50 "Fine" and "Coarse" Mode

We can see that the ResNet Fine approach performs better than the ResNet Coarse approach. As we have mentioned in Section 2.3, ResNet Fine operates at a higher resolution. This can make it more sensitive to variations within each class and therefore improve the performance by giving to the classifier a feature vector with more precise data. On the other hand, ResNet Coarse operates at a lower resolution. This means that images have been downsampled and only the main features remain. As we have already mentioned, the images of the classes "cloudy" and "rain", as well as "sunrise" and "shine" can share some of these mean features. For example, in "shine" and "sunrise" images there will be mostly always the sun while in "cloudy" and "rain" images there will be almost always clouds. Therefore, since ResNet Coarse approach just works with main features and they may be shared among classes, it is more difficult for the classifier to distinguish them and, as a consequence, the classification performance decreases.

4.4 Channel Normalization (Extra Credit)

From Tables 1 and 2 we see that when using the AdaIN approach, the classification performance is better than when using the Gram matrix approach. AdaIN is designed to match both the mean and variance of feature maps, making it effective in capturing texture information in a more flexible way. This flexibility allows it to better capture the style of an image. In contrast, the Gram matrix captures style through global correlations between feature map channels, but it does not consider the mean and variance independently, which can make it less adaptable to variations within the dataset. For that reason, the three deep learning based approaches outperform when using the AdaIN approach to create the feature vectors in comparison to the Gram matrix approach.

5 Code

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np
import os
from PIL import Image

def get_confusion_matrix(train_feat, train_lab, test_feat, test_lab, path, method):
    # train svm classifier with training data
    svm_class = SVC(kernel='linear')
    svm_class.fit(train_feat, train_lab)

    # get prediction with testing data
    pred = svm_class.predict(test_feat)

    # create confusion matrix
    cm = confusion_matrix(test_lab, pred)
    class_names = ['Cloudy', 'Rain', 'Shine', 'Sunrise']
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_names, yticklabels=class_names)
    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.savefig(path)

    class_names = ["cloudy", "rain", "shine", "sunrise"]
    test_dir = '/data/aolivepe/HW7-Auxilliary/data/testing'
    image_paths = [os.path.join(test_dir, img) for img in sorted(os.listdir(test_dir))]
    # Iterate over each class and find correct and incorrect classifications
    for cls in range(4):
        class_idx = np.where(test_lab == cls)[0]
        correct_idx = class_idx[test_lab[class_idx] == pred[class_idx]]
        incorrect_idx = class_idx[test_lab[class_idx] != pred[class_idx]]

        # Save one example of correct and incorrect classification
        if len(correct_idx) > 0:
            # Read image from path
            correct_image = Image.open(image_paths[correct_idx[0]])
            save_annotated_image(correct_image, test_lab[correct_idx[0]], pred[correct_idx[0]], True, class_names, correct_idx[0], method)

        if len(incorrect_idx) > 0:
            # Read image from path
            incorrect_image = Image.open(image_paths[incorrect_idx[0]])
            save_annotated_image(incorrect_image, test_lab[incorrect_idx[0]], pred[incorrect_idx[0]], False, class_names, incorrect_idx[0], method)

# Annotate and save images
def save_annotated_image(img, gt, pred, correct, class_name, idx, method):
    img = img.copy()
    text = f"GT: {class_name[gt]}, Pred: {class_name[pred]}, {'Correct' if correct else 'Incorrect'}"
    filename = f"/home/aolivepe/HW7/results/{method}_{text}.jpg"
```

```
    img.save(filename, format="JPEG")


# Load the feature vectors and labels of the training and testing dataset for each method and compute the confusion matrix
test = np.load('/home/aolivepe/HW7/results/lbp_test.npz')
test_feat = test['features']
test_lab = test['labels']

train = np.load('/home/aolivepe/HW7/results/lbp_train.npz')
train_feat = train['features']
train_lab = train['labels']

get_confusion_matrix(train_feat, train_lab, test_feat, test_lab, "/home/aolivepe/HW7/results/lbp_confusion_matrix.jpg", "lbp")

test = np.load('/home/aolivepe/HW7/results/vgg_test.npz')
test_feat = test['features']
test_lab = test['labels']

train = np.load('/home/aolivepe/HW7/results/vgg_train.npz')
train_feat = train['features']
train_lab = train['labels']

get_confusion_matrix(train_feat, train_lab, test_feat, test_lab, "/home/aolivepe/HW7/results/vgg_confusion_matrix.jpg", "vgg")

test = np.load('/home/aolivepe/HW7/results/resnet_fine_test.npz')
test_feat = test['features']
test_lab = test['labels']

train = np.load('/home/aolivepe/HW7/results/resnet_fine_train.npz')
train_feat = train['features']
train_lab = train['labels']

get_confusion_matrix(train_feat, train_lab, test_feat, test_lab, "/home/aolivepe/HW7/results/resnet_fine_confusion_matrix.jpg", "resnet_fine")

test = np.load('/home/aolivepe/HW7/results/resnet_coarse_test.npz')
test_feat = test['features']
test_lab = test['labels']

train = np.load('/home/aolivepe/HW7/results/resnet_coarse_train.npz')
train_feat = train['features']
train_lab = train['labels']

get_confusion_matrix(train_feat, train_lab, test_feat, test_lab, "/home/aolivepe/HW7/results/resnet_coarse_confusion_matrix.jpg", "resnet_coarse")
```

```
import os
import cv2
import numpy as np
from sklearn.preprocessing import StandardScaler
from vgg_and_resnet import VGG19, CustomResNet


def v_norm(feat_map):
    # Apply channel normalization following the steps from the extra credit instructions
```

```
    feats = feat_map.reshape(feat_map.shape[0], feat_map.shape[1] *
        feat_map.shape[2])
    adain = np.stack((np.mean(feats, axis=1), np.var(feats, axis=1)), axis=1)
    adain = adain / np.max(adain)
    return np.reshape(adain, [1, np.prod(adain.shape)]).squeeze()

def get_data(dir, model, method):
    feat = []
    lab = []

    # Iterate over all images
    for filename in sorted(os.listdir(dir)):
        image_path = os.path.join(dir, filename)

        #Read image, resize it, conevrt it to RGB and scale between 0 and 1
        img = cv2.imread(image_path)
        if img is None:
            print(f"Skip {image_path}")
            continue
        img = cv2.resize(img, (256, 256))
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = img / 255.0

        # Get feature vector from the channel normalization according to the
            method used and append it to the features list
        if method == 'vgg':
            vgg_feat = model(img)
            feat.append(v_norm(vgg_feat))
        elif method == 'resnet_coarse':
            resnet_coarse_feat, _ = model(img)
            feat.append(v_norm(resnet_coarse_feat))
        elif method == 'resnet_fine':
            _, resnet_fine_feat = model(img)
            feat.append(v_norm(resnet_fine_feat))

        # Append the integer corresponding to the label in the labels list
        classes = ["cloudy", "rain", "shine", "sunrise"]
        if 'cloudy' in filename:
            lab.append(classes.index("cloudy"))
        elif 'rain' in filename:
            lab.append(classes.index("rain"))
        elif 'shine' in filename:
            lab.append(classes.index("shine"))
        elif 'sunrise' in filename:
            lab.append(classes.index("sunrise"))
        else:
            continue

    feat = np.array(feat)
    lab = np.array(lab)

    # (feat - mean) / std_dev
    scaler = StandardScaler()
    feat = scaler.fit_transform(feat)
    return feat, lab

train_dir = '/data/aolivepe/HW7-Auxilliary/data/training'
test_dir = '/data/aolivepe/HW7-Auxilliary/data/testing'

# Load pretrained VGG and ResNet
vgg_model = VGG19()
vgg_model.load_weights('/data/aolivepe/HW7-Auxilliary/vgg_normalized.pth')
```

```
resnet_model = CustomResNet(encoder='resnet50')

# Get features and labels from the training and testing dataset and save them. Do this for each feature extractor method used
method = "vgg"
train_feat, train_lab = get_data(train_dir, vgg_model, method)
np.savez_compressed(f'./results/{method}_train_ADAIN.npz', features=train_feat, labels=train_lab)
test_feat, test_lab = get_data(test_dir, vgg_model, method)
np.savez_compressed(f'./results/{method}_test_ADAIN.npz', features=test_feat, labels=test_lab)
print(f"{method} train and test completed")

method = "resnet_coarse"
train_feat, train_lab = get_data(train_dir, resnet_model, method)
np.savez_compressed(f'./results/{method}_train_ADAIN.npz', features=train_feat, labels=train_lab)
test_feat, test_lab = get_data(test_dir, resnet_model, method)
np.savez_compressed(f'./results/{method}_test_ADAIN.npz', features=test_feat, labels=test_lab)
print(f"{method} train and test completed")

method = "resnet_fine"
train_feat, train_lab = get_data(train_dir, resnet_model, method)
np.savez_compressed(f'./results/{method}_train_ADAIN.npz', features=train_feat, labels=train_lab)
test_feat, test_lab = get_data(test_dir, resnet_model, method)
np.savez_compressed(f'./results/{method}_test_ADAIN.npz', features=test_feat, labels=test_lab)
print(f"{method} train and test completed")
```

```
import os
import cv2
import numpy as np
from sklearn.preprocessing import StandardScaler
from vgg_and_resnet import VGG19, CustomResNet

def gram_matrix(feat_map):
    # Compute gram matrix given the feature map
    feats = feat_map.reshape(feat_map.shape[0], feat_map.shape[1] * feat_map.shape[2])
    G = np.dot(feats, feats.T)
    gram = G / (feat_map.shape[0] * feat_map.shape[1] * feat_map.shape[2])

    # Downsample to 32x32 as mentioned in instructions
    down_gram = cv2.resize(gram, (32, 32), interpolation=cv2.INTER_LINEAR)
    return down_gram[np.triu_indices(32)]

def get_data(dir, model, method):
    feat = []
    lab = []

    # Iterate over all images
    for filename in sorted(os.listdir(dir)):
        image_path = os.path.join(dir, filename)

        #Read image, resize it, conevrt it to RGB and scale between 0 and 1
        img = cv2.imread(image_path)
        if img is None:
            print(f"Skip {image_path}")
            continue
```

```
            img = cv2.resize(img, (256, 256))
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            img = img / 255.0

            # Get feature vector from the Gram Matrix according to the method
                used and append it to the features list
            if method == 'vgg':
                vgg_feat = model(img)
                feat.append(gram_matrix(vgg_feat))
            elif method == 'resnet_coarse':
                resnet_coarse_feat, _ = model(img)
                feat.append(gram_matrix(resnet_coarse_feat))
            elif method == 'resnet_fine':
                _, resnet_fine_feat = model(img)
                feat.append(gram_matrix(resnet_fine_feat))

            # Append the integer corresponding to the label in the labels list
            classes = ["cloudy", "rain", "shine", "sunrise"]
            if 'cloudy' in filename:
                lab.append(classes.index("cloudy"))
            elif 'rain' in filename:
                lab.append(classes.index("rain"))
            elif 'shine' in filename:
                lab.append(classes.index("shine"))
            elif 'sunrise' in filename:
                lab.append(classes.index("sunrise"))
            else:
                continue

    feat = np.array(feat)
    lab = np.array(lab)

    # (feat - mean) / std_dev
    scaler = StandardScaler()
    feat = scaler.fit_transform(feat)
    return feat, lab

train_dir = '/data/aolivepe/HW7-Auxilliary/data/training'
test_dir = '/data/aolivepe/HW7-Auxilliary/data/testing'

# Load pretrained VGG and ResNet
vgg_model = VGG19()
vgg_model.load_weights('/data/aolivepe/HW7-Auxilliary/vgg_normalized.pth')
resnet_model = CustomResNet(encoder='resnet50')

# Get features and labels from the training and testing dataset and save
    them. Do this for each feature extractor method used
method = "vgg"
train_feat, train_lab = get_data(train_dir, vgg_model, method)
np.savez_compressed(f'./results/{method}_train.npz', features=train_feat,
    labels=train_lab)
test_feat, test_lab = get_data(test_dir, vgg_model, method)
np.savez_compressed(f'./results/{method}_test.npz', features=test_feat,
    labels=test_lab)
print(f"{method} train and test completed")

method = "resnet_coarse"
train_feat, train_lab = get_data(train_dir, resnet_model, method)
np.savez_compressed(f'./results/{method}_train.npz', features=train_feat,
    labels=train_lab)
test_feat, test_lab = get_data(test_dir, resnet_model, method)
np.savez_compressed(f'./results/{method}_test.npz', features=test_feat,
```

```
    labels=test_lab)
print(f"{method} train and test completed")

method = "resnet_fine"
train_feat, train_lab = get_data(train_dir, resnet_model, method)
np.savez_compressed(f'./results/{method}_train.npz', features=train_feat,
    labels=train_lab)
test_feat, test_lab = get_data(test_dir, resnet_model, method)
np.savez_compressed(f'./results/{method}_test.npz', features=test_feat,
    labels=test_lab)
print(f"{method} train and test completed")
```

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
import random
from vgg_and_resnet import VGG19, CustomResNet

def gram_matrix(feat_map):
    # Compute gram matrix given the feature map
    feats = feat_map.reshape(feat_map.shape[0], feat_map.shape[1] *
        feat_map.shape[2])
    G = np.dot(feats, feats.T)
    return np.log1p(G)

test_dir = '/data/aolivepe/HW7-Auxilliary/data/testing'
vgg_model = VGG19()
vgg_model.load_weights('/data/aolivepe/HW7-Auxilliary/vgg_normalized.pth')
resnet_model = CustomResNet(encoder='resnet50')

fig, axs = plt.subplots(3, 4, figsize=(20, 15))
classes = ['cloudy', 'rain', 'shine', 'sunrise']

# Iterate over the 4 classes and randomly select one image for each class to
    plot the Gram matrix with each of the approaches used
for i, class_l in enumerate(classes):
    img_filenames = [im for im in os.listdir(test_dir) if
        im.lower().startswith(class_l)]
    if not img_filenames:
        continue
    img = cv2.imread(os.path.join(test_dir, random.choice(img_filenames)))
    img = cv2.resize(img, (256, 256))

    vgg_feat = vgg_model(img)
    G = gram_matrix(vgg_feat)
    cax = axs[0, i].imshow(G, cmap='viridis')
    fig.colorbar(cax, ax=axs[0, i])
    axs[0, i].set_title(f"Method: Vgg, Class: {class_l.capitalize()}")

    _, resnet_fine_feat = resnet_model(img)
    G = gram_matrix(resnet_fine_feat)
    cax = axs[1, i].imshow(G, cmap='viridis', vmin=0.75, vmax=4)
    fig.colorbar(cax, ax=axs[1, i])
    axs[1, i].set_title(f"Method: Resnet_fine, Class: {class_l.capitalize()}")

    resnet_coarse_feat, _ = resnet_model(img)
    G = gram_matrix(resnet_coarse_feat)
    cax = axs[2, i].imshow(G, cmap='viridis', vmin=0.0, vmax=1)
    fig.colorbar(cax, ax=axs[2, i])
    axs[2, i].set_title(f"Method: Resnet_coarse, Class:
```

```
        {class_l.capitalize()}")

plt.tight_layout()
# Save the result
plt.savefig("./results/gram_matrices.jpg")
```

```
import cv2
import numpy as np
from BitVector import BitVector
import math
import os
from sklearn.preprocessing import StandardScaler

# Function to convert RGB image to HSI
def rgb_to_hsi(image):
    # Convert image to float for precision
    image = image.astype(np.float32) / 255.0  # Normalize to [0,1]
    R, G, B = image[:, :, 0], image[:, :, 1], image[:, :, 2]

    # Intensity calculation
    I = (R + G + B) / 3

    # Saturation calculation
    min_rgb = np.minimum(np.minimum(R, G), B)
    S = 1 - (3 * min_rgb / (R + G + B + 1e-6))

    # Hue calculation
    numerator = 0.5 * ((R - G) + (R - B))
    denominator = np.sqrt((R - G)**2 + (R - B) * (G - B)) + 1e-6
    theta = np.arccos(numerator / denominator)
    H = np.where(B <= G, theta, 2 * np.pi - theta)
    H = H / (2 * np.pi)  # Normalize hue to [0,1]

    return np.dstack((H, S, I))

# Function to compute Local Binary Pattern (LBP)
def computeLbp(img, R=1, P=8):
    # Create feature vector full of zeros
    lbp_hist = [0]*(P+2)

    # Iterate over all the pixels of the image except of the ones at distance
        R of the border to avoid problems with LBP implementation
    for h in range(R, img.shape[0] - R):
        for w in range(R, img.shape[1] - R):
            pattern = []
            # Get the value of each of the p points arround the center
            for p in range(P):
                k, l = h + R*math.cos(2*math.pi*p/P), w -
                    R*math.sin(2*math.pi*p/P)
                k_base, l_base = int(k), int(l)
                if abs(k - k_base) < 1e-8 and abs(l - l_base) < 1e-8:
                    img_val_at_p = img[k_base, l_base]
                else:
                    delta_k = k - k_base
                    delta_l = l - l_base

                    if delta_l < 1e-8:
                        img_val_at_p = (1 - delta_k) * img[k_base, l_base] +
                            delta_k * img[k_base + 1, l_base]
                    elif delta_k < 1e-8:
                        img_val_at_p = (1 - delta_l) * img[k_base, l_base] +
```

```
                            delta_l * img[k_base, l_base + 1]
                    else:
                        img_val_at_p = (
                            delta_k * (1 - delta_l) * img[k_base + 1, l_base] +
                            delta_k * delta_l * img[k_base + 1, l_base + 1] +
                            (1 - delta_k) * (1 - delta_l) * img[k_base, l_base] +
                            (1 - delta_k) * delta_l * img[k_base, l_base + 1]
                        )

                    # Threshold value at p compared to the value of the pixel in the center
                    if img_val_at_p >= img[h, w]:
                        pattern.append(1)
                    else:
                        pattern.append(0)

                # Similar to Prof. Kak's tutorial
                bv = BitVector(bitlist=pattern)
                int_circular_shift = [int(bv << i) for i in range(P)]
                minbv = BitVector(intVal=min(int_circular_shift), size=P)
                bvruns = minbv.runs()

                if len(bvruns) > 2:
                    lbp_hist[P + 1] += 1
                elif len(bvruns) == 1 and bvruns[0][0] == '1':
                    lbp_hist[P] += 1
                elif len(bvruns) == 1 and bvruns[0][0] == '0':
                    lbp_hist[0] += 1
                else:
                    lbp_hist[len(bvruns[1])] += 1

    return lbp_hist

def get_data(dir):
    feat = []
    lab = []

    # Iterate over all images
    for filename in sorted(os.listdir(dir)):
        image_path = os.path.join(dir, filename)

        #Read image, resize it, get hue
        img = cv2.imread(image_path)
        if img is None:
            print(f"Skip {image_path}")
            continue
        img = cv2.resize(img, (64, 64))
        hsi_image = rgb_to_hsi(img)
        hue_channel = (hsi_image[:, :, 0] * 255).astype(np.uint8)

        #Get LBP feature vector and append it to the features list
        feat.append(computeLbp(hue_channel, R=1, P=8))

        # Append the integer corresponding to the label in the labels list
        classes = ["cloudy", "rain", "shine", "sunrise"]
        if 'cloudy' in filename:
            lab.append(classes.index("cloudy"))
        elif 'rain' in filename:
            lab.append(classes.index("rain"))
        elif 'shine' in filename:
```

```
            lab.append(classes.index("shine"))
        elif 'sunrise' in filename:
            lab.append(classes.index("sunrise"))
        else:
            continue

    feat = np.array(feat)
    lab = np.array(lab)

    # (feat - mean) / std_dev
    scaler = StandardScaler()
    feat = scaler.fit_transform(feat)
    return feat, lab

train_dir = '/data/aolivepe/HW7-Auxilliary/data/training'
test_dir = '/data/aolivepe/HW7-Auxilliary/data/testing'

# Get features and labels from the training and testing dataset and save them
train_feat, train_lab = get_data(train_dir)
np.savez_compressed(f'./results/lbp_train.npz', features=train_feat,
    labels=train_lab)

test_feat, test_lab = get_data(test_dir)
np.savez_compressed(f'./results/lbp_test.npz', features=test_feat,
    labels=test_lab)
```