Homework 7

Theory Question

Within the neural network, first the image is processed into 4 different scales: the original image, 1/2 resolution, 1/4 resolution and 1/8 resolution. After passing through a VGG-19 like structure (but only the first 16 layers, with modifications to have the resulting sizes not be too small), there are 4 convolutional blocks of varying depths. Each of the image resolutions is passed through these 4 blocks, for the outputs that have a size smaller than 32 by 32, we upsample them to reach size 32 by 32. Then all of these outputs are concatenated together, we pass this result through one last convolutional layer that downsizes the number of channels to 512 but keeps the image resolution the same (32 by 32). We then calculate the channel normalization parameters for these channels, which we then use in a similar way to AdaIN.
This method might work best on images that contain most of the features within the first few scales (1/2, 1/4 and 1/8), as we are explicitly running VGG on these. So there could be some information loss for the textures that are detected at higher scales. Being able to put the images through one more block will allow us to get a little bit more features, and then the concatenation and subsequent convolution allows us to relate information from all scales as well as their features to enhance our classifier and also make sure that there is no redundant information from them as dictated by the weights of this final layer. This would not perform well in images where there are small objects, or where the textures all look similar at the scales we work on.

Task

(a) Example for "cloudy" class.

(b) Example for "rain" class.

(c) Example for "shine" class.

(d) Example for "sunrise" class.

Figure 1: Example images for each class.

LBP

First we convert the image to the HSI representation. We do this by following the algorithm to convert from RGB to HSI from Prof. Kak's book. Additionally, we round up and divide the H channel by 2, this is done to be able to use cv2.resize on the resulting matrix. After this, we loop through the image and use BitVector to encode the pattern. We use $R = 2$ and $P = 12$ for our LBP extraction. We found that this parameters worked a bit better compared to $R = 1$ and $P = 8$. After this, we convert the output to an array and divide by the sum to get values from 0 to 1. These features are what we use to train the SVM on the training dataset and then get the confusion matrix and the accuracy on the testing dataset.

Gram-matrix

For all models we follow the same procedure. First each image is resized to $(256, 256)$ and we get the output for the models and then reshape it, then multiply it by its transpose, which is symmetric so we take the upper triangular part. Since the final matrix is of size

$(512, 512)$ or in the case of Resnet coarse $(1024, 1024)$, we flatten the array, and take the first 2048 values. The reason for 2048 is that since we only took the upper triangular matrix, the dimensionality is half of 2048, 1024. This is to compare how well this does against the AdaIN features in the following section. This feature vector of size 2048 is what is used to train the SVM.

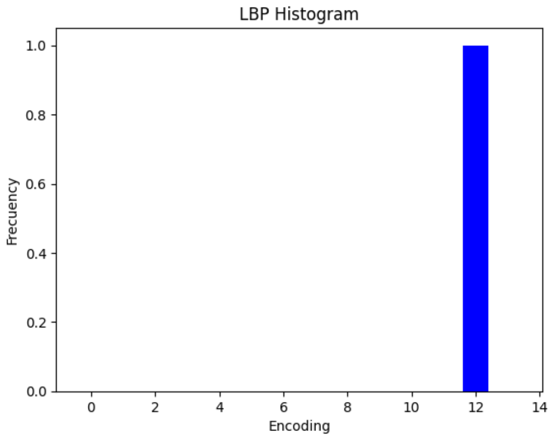
AdaIN features

We use VGG model for this part. We take the output of shape $(512, 32, 32)$ and calculate the mean and standard deviation per channel resulting in 2 arrays of size 512. We concatenate both and use that array of length 1024 to train our SVM.

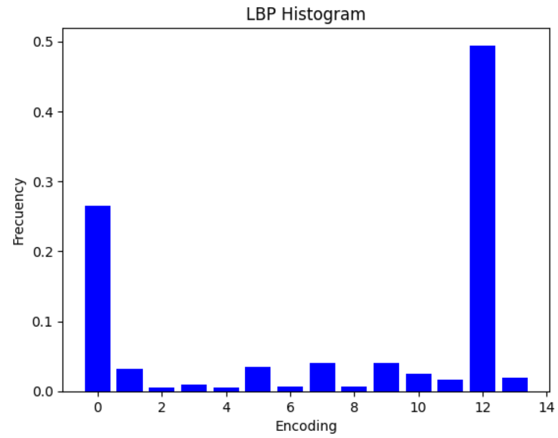
Results

LBP

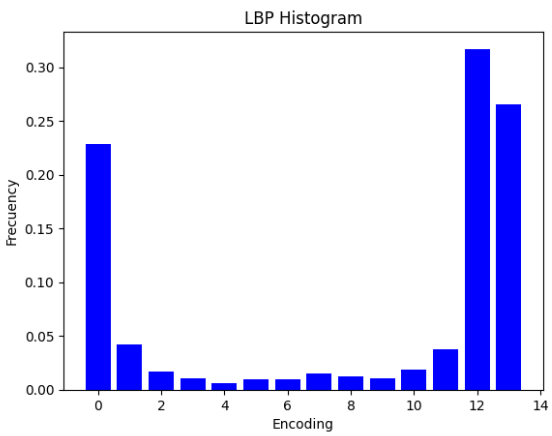
There was one issue we found when looking at the images. Some of the classes, mainly "cloudy" contained grayscale images, the problem with this is that when we get the HSI representation, the hue channel is all 1. This is due to the calculation of the maximum and minimum RGB value, but in grayscale they are all the same. This can be seen in Fig. 2a. This could potentially cause issues when classifying using the SVM.
After training the SVM, it performed poorly. Shown in Fig. 3, we can see that only the "sunrise" class performed well, and that is because it mostly classified everything as "sunrise". This could be explained by the fact that "sunrise" class contains the most amount of images. The class it struggled with the most is "shine", and it mostly classified all of them as "sunrise". There is quite a bit of overlap between "shine" and "sunrise" since they both feature the sun, there are also clouds which explains why part of the "cloudy" images were classified as "sunrise" and there is also some overlap between "rain" and "cloudy" as they both feature clouds. Overall, the accuracy for this classifier on the LBP descriptors was 40% on the testing dataset.



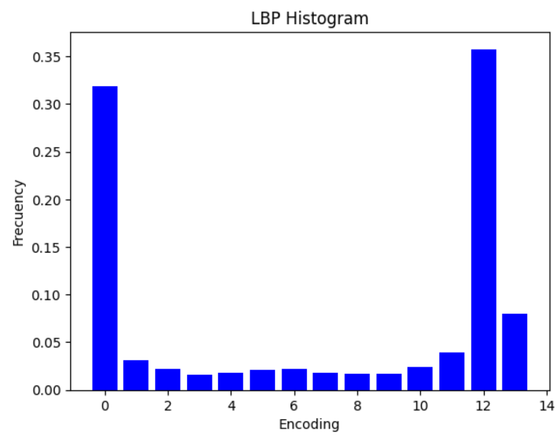
(a) LBP descriptor vector for "cloudy" class.



(b) LBP descriptor vector for "rain" class.



(c) LBP descriptor vector for "shine" class.



(d) LBP descriptor vector for "sunrise" class.

Figure 2: LBP descriptor vector for each class.

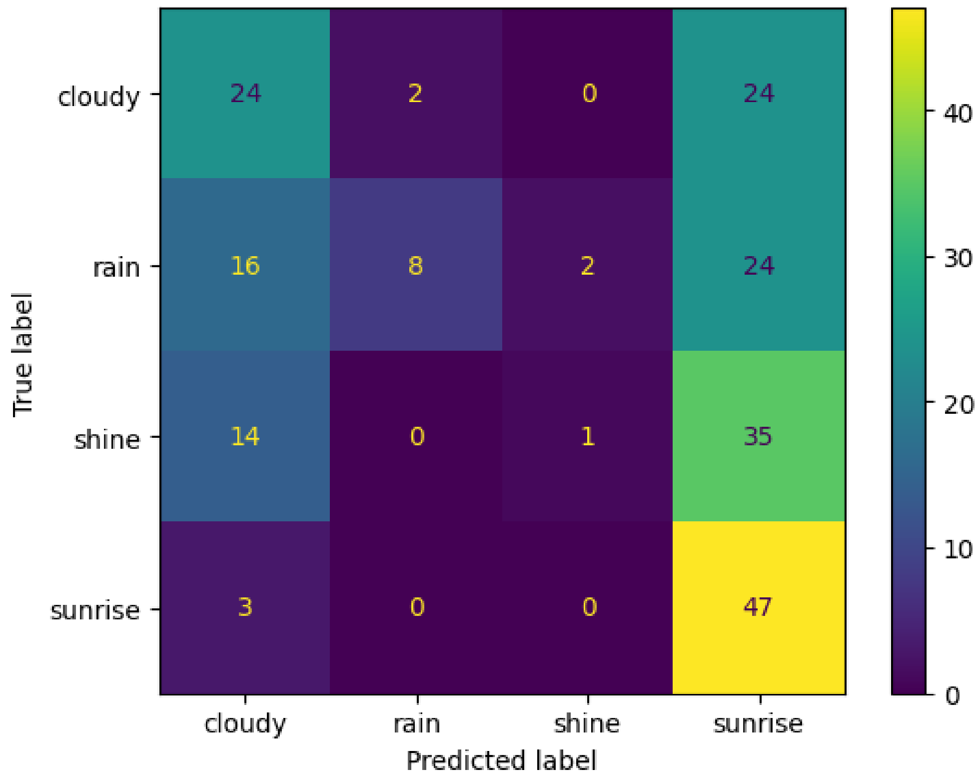


Figure 3: Confusion matrix for the LBP descriptors

(a) Image correctly classified as ”rain” class.

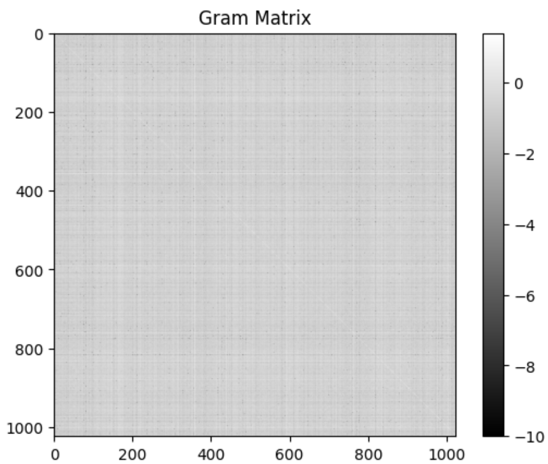
(b) Image incorrectly classified as ”cloudy” class, the true label is ”shine”.

Figure 4: Correct and incorrect classification for LBP.

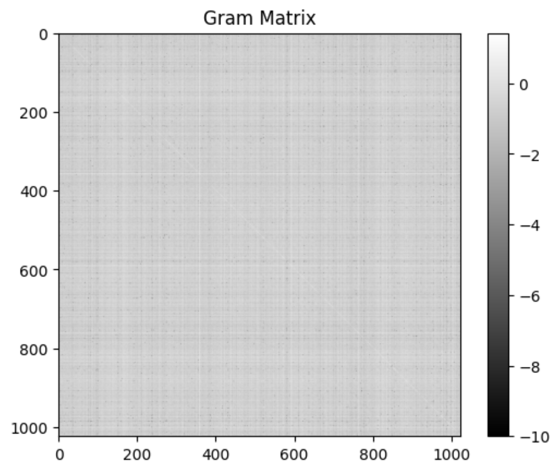
Gram-Matrix

Resnet Coarse For Resnet coarse, it is hard to spot many differences in the Gram matrix representation, as shown in Fig. 5. We no longer have the issue with grayscale images here, as we use the RGB representation instead of HSI.

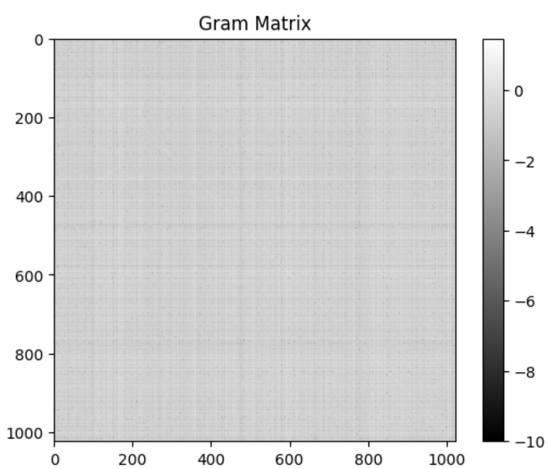
After training the SVM, with a confusion matrix shown in Fig. 6, we notice an improvement in the performance compared to LBP. In this case, we correctly classified all "sunrise" images, and most of the "cloudy" images got classified correctly. The classes that this classifier struggled on the most was "rain" and "shine", with both of them being mostly misclassified as "cloudy", this could be due to the fact that both of those classes have images that have clouds in them. Overall, this classifier achieved an accuracy of 79.5% on the testing set.



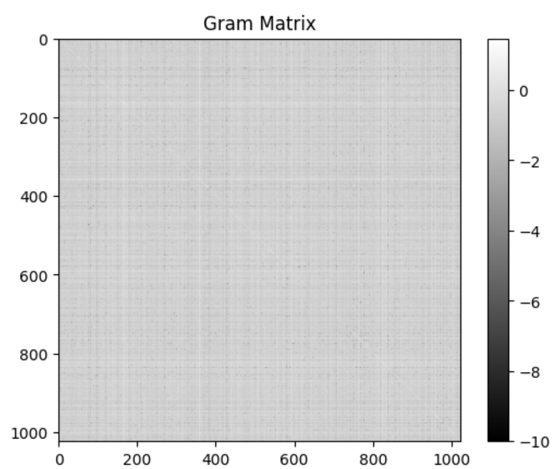
(a) Gram matrix from Resnet coarse for "cloudy" class.



(b) Gram matrix from Resnet coarse for "rain" class.



(c) Gram matrix from Resnet coarse for "shine" class.



(d) Gram matrix from Resnet coarse for "sunrise" class.

Figure 5: Gram matrix from Resnet coarse for each class.

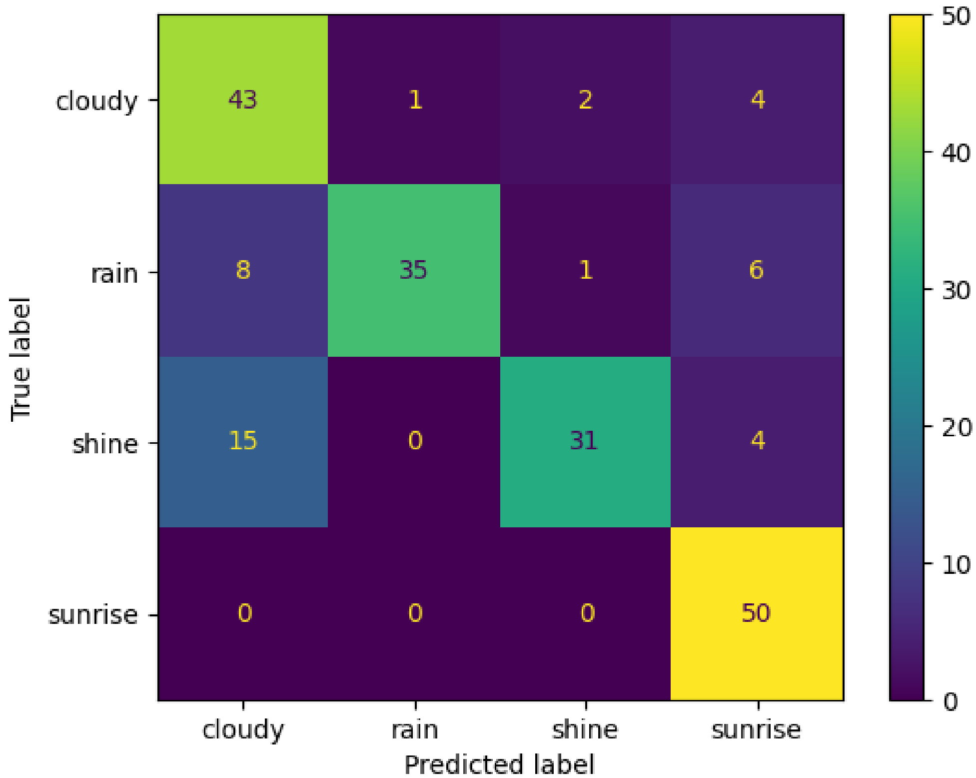
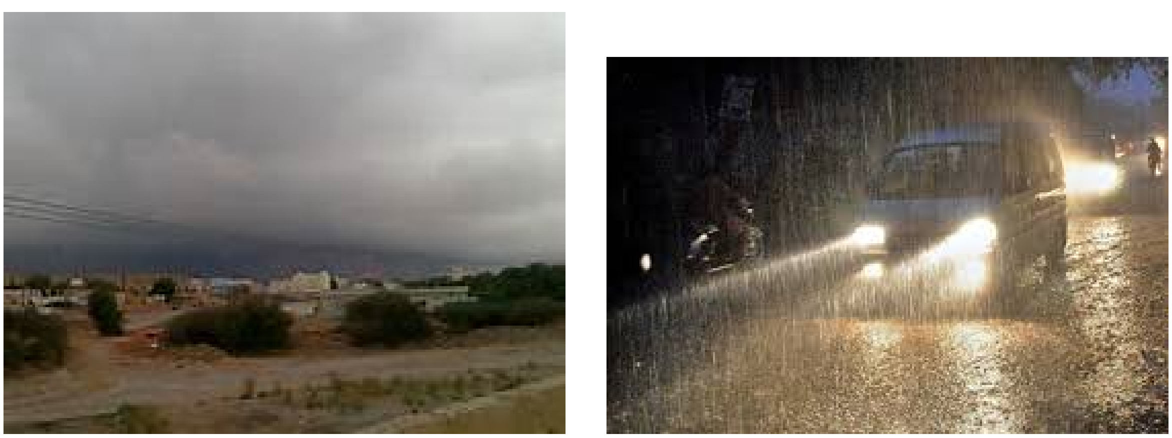


Figure 6: Confusion matrix for the Resnet coarse Gram matrix

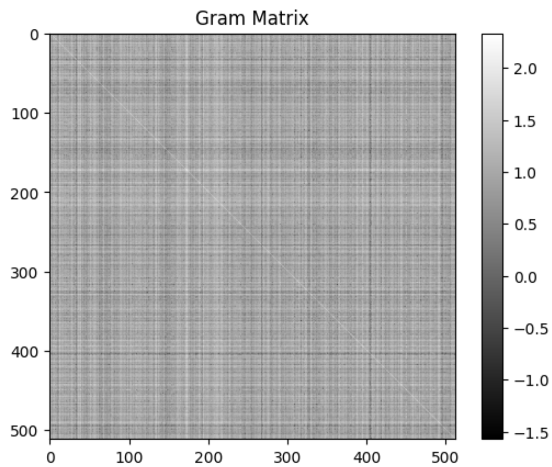


(a) Image correctly classified as "cloudy" class.

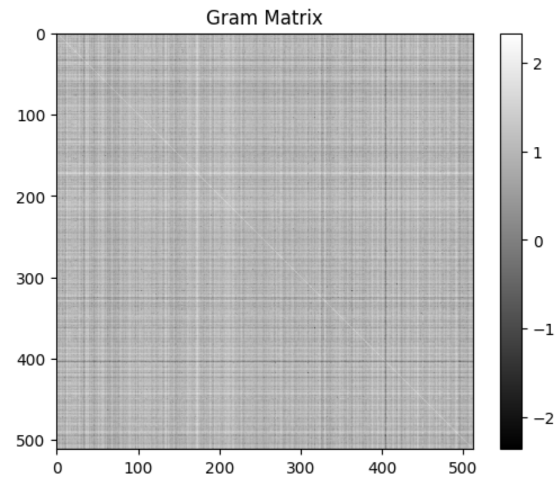
(b) Image incorrectly classified as "sunrise" class, the true label is "rain".

Figure 7: Correct and incorrect classification for Resnet coarse.

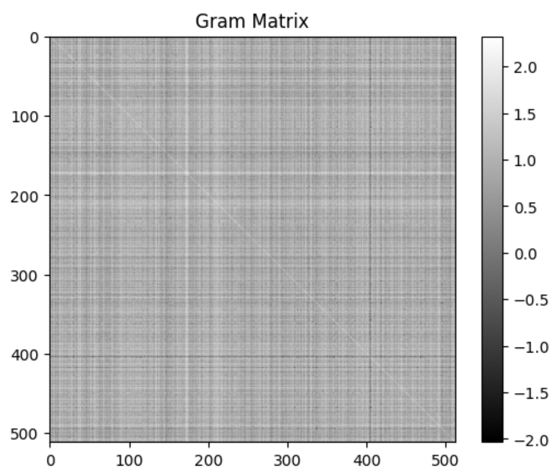
Resnet Fine Now we can see some differences in the Gram matrix representations, as shown in Fig. 8. After training the SVM, notice that the confusion matrix, Fig. 9, shows significant improvements over Resnet coarse. Most of the images have been correctly classified, the only class that shows some struggles is "shine", with some classified as "cloudy" and some classified as ”sunrise”, this could be explained because, in a similar manner to Resnet coarse, all of them contain images with clouds. The accuracy on the testing set for this classifier is 91.5%.



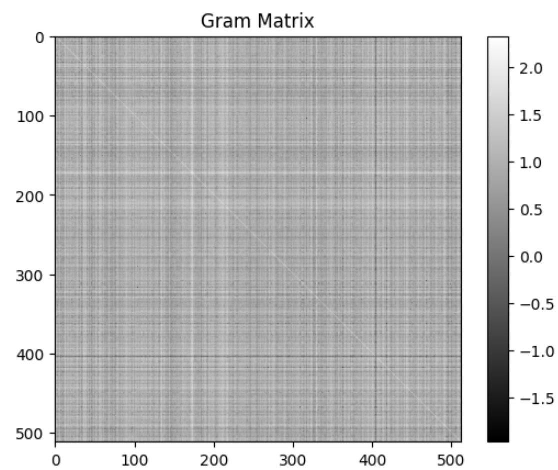
(a) Gram matrix from Resnet fine for ”cloudy” class.



(b) Gram matrix from Resnet fine for ”rain” class.



(c) Gram matrix from Resnet fine for ”shine” class.



(d) Gram matrix from Resnet fine for ”sunrise” class.

Figure 8: Gram matrix from Resnet fine for each class.

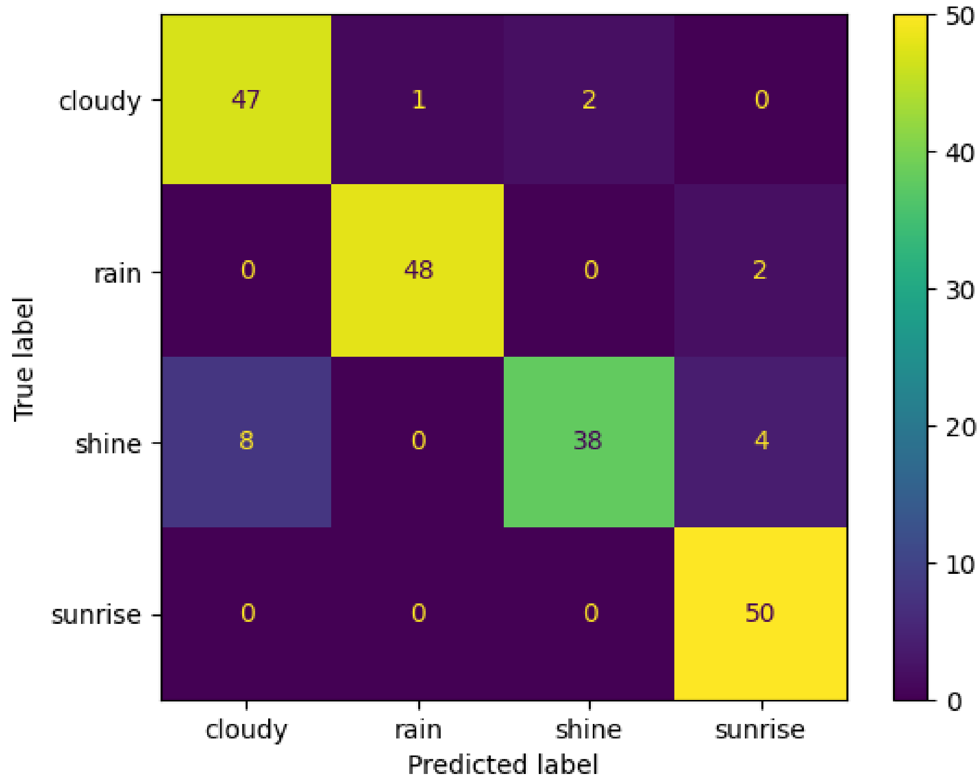


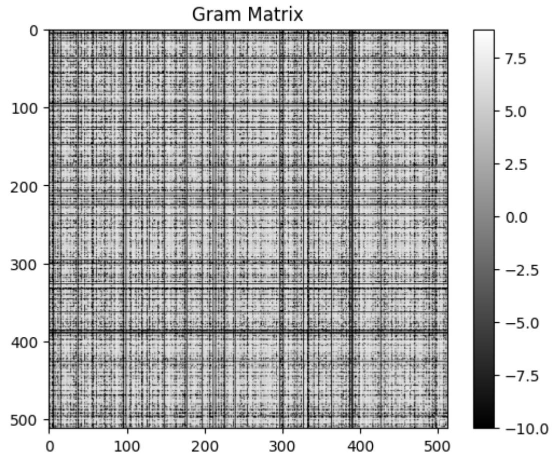
Figure 9: Confusion matrix for the Resnet fine Gram matrix

(a) Image correctly classified as "sunrise" class.

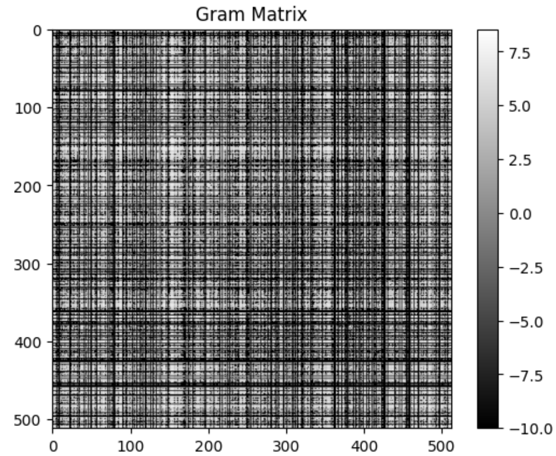
(b) Image incorrectly classified as "sunrise" class, the true label is "rain".

Figure 10: Correct and incorrect classification for Resnet fine.

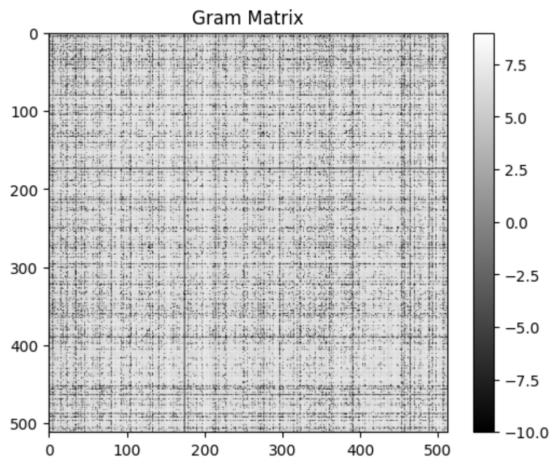
VGG The differences in the Gram matrix representations, as shown in Fig. 11 are very pronounced now. The SVM classifier does well on both "sunrise" and "rain" but is not as accurate on the other classes, as can be seen in Fig. 12. The performance of this model is worse than Resnet fine but better than Resnet coarse. There are many images misclassified as "sunrise", possibly due to the fact that they have the sun. The overall accuracy for this model on the testing set is 89%.



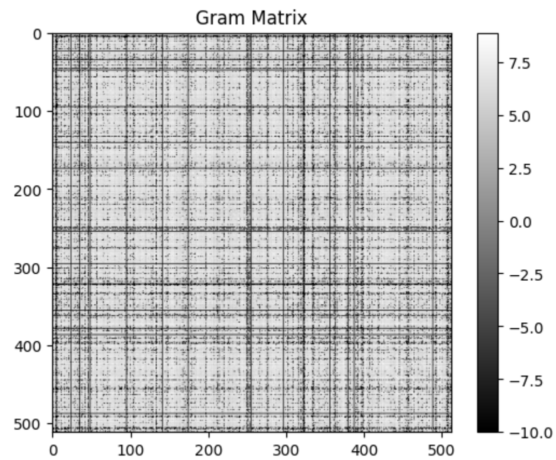
(a) Gram matrix from VGG for "cloudy" class.



(b) Gram matrix from VGG for "rain" class.



(c) Gram matrix from VGG for "shine" class.



(d) Gram matrix from VGG for "sunrise" class.

Figure 11: Gram matrix from Resnet fine for each class.

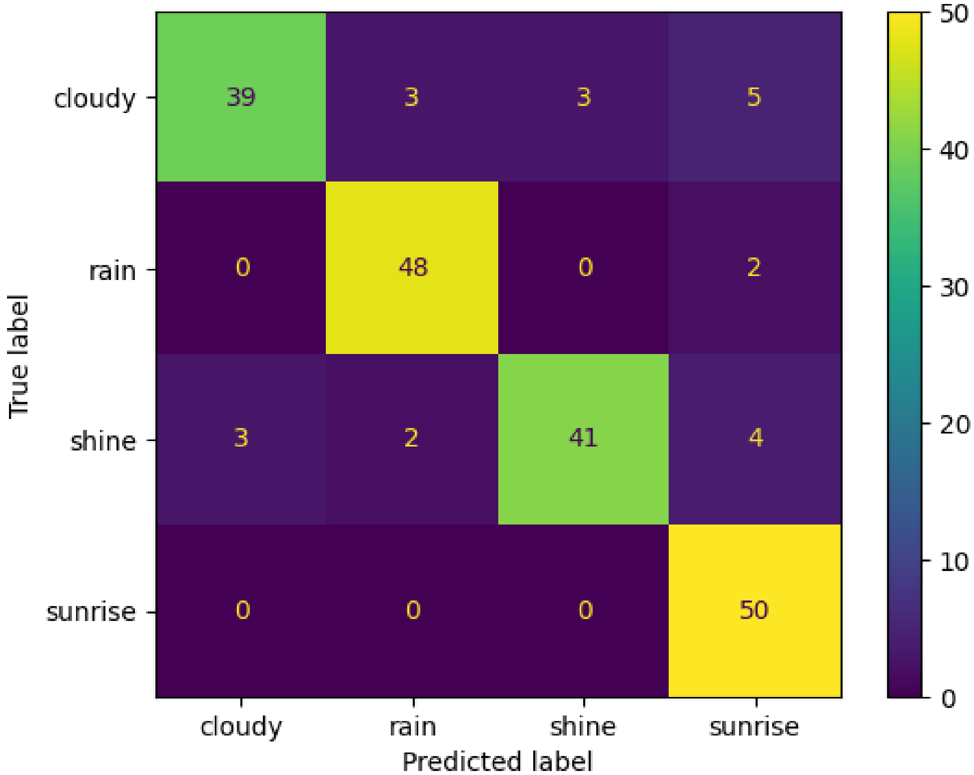


Figure 12: Confusion matrix for the VGG Gram matrix

(a) Image correctly classified as ”shine” class.

(b) Image incorrectly classified as ”sunrise” class, the true label is ”cloudy”.

Figure 13: Correct and incorrect classification for Resnet coarse.

Extra credit: AdaIN

AdaIN features performed the best out of all classifiers. As can be seen in Fig. 14, it performed extremely well on ”cloudy”, ”rain” and ”sunrise”. Only the ”shine” class had 4 images misclassified as both ”sunrise” and ”cloudy” due to the sun and the clouds in the images respectively. The total accuracy on the testing set is 97.5%.

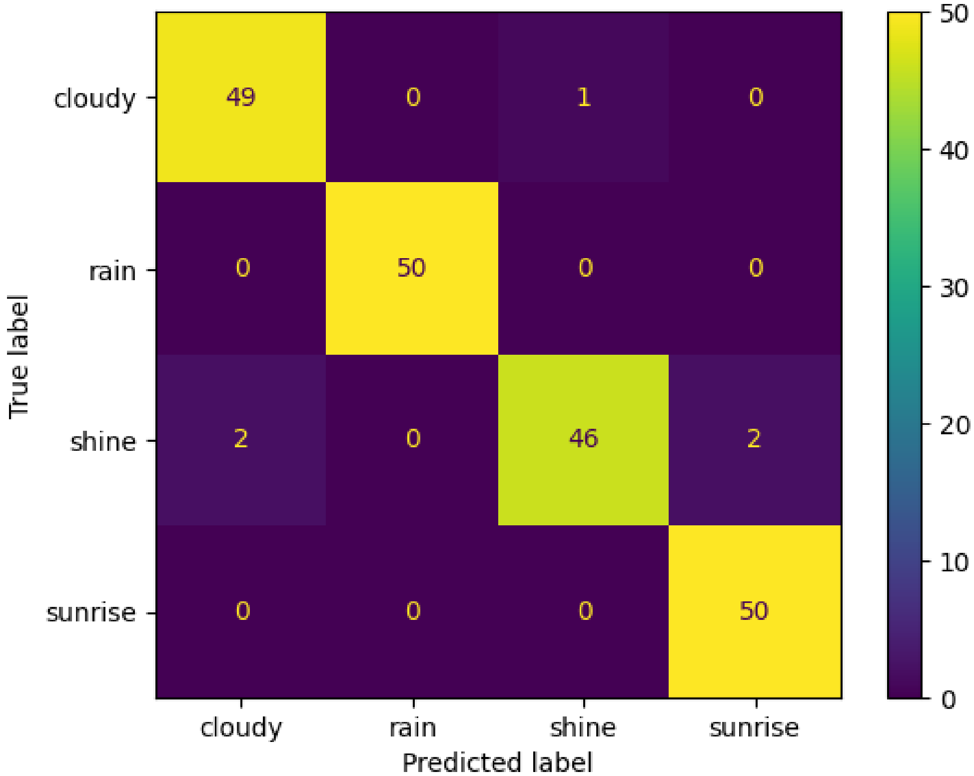


Figure 14: Confusion matrix for the AdaIN features

Source code

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import BitVector
import os
from vgg_and_resnet import *
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn import svm

def get_hsl(image):
    image = image/255.
    # image is bgr, opencv default
    # start by splitting image into blue green red
    blue, green, red = image[...,0], image[...,1], image[...,2]
    # calculate the max and min per position
    Cmax_arg = np.argmax(image,axis=2)
    Cmax = np.max(image,axis=2)
    Cmin = np.min(image,axis=2)
    # get the delta
```

```python
    delta = Cmax - Cmin
    # now start populating the hsv matrices
    v = Cmax
    h = np.zeros_like(v)
    s = np.zeros_like(v)
    s[v != 0] = delta[v != 0] / v[v != 0]

    # was not entirely sure how to vectorize this so I did it with a loop
    for j in range(h.shape[0]):
        for i in range(h.shape[1]):
            if delta[j,i] != 0:
                # for blue
                if Cmax_arg[j,i] == 0:
                    h[j,i] = ((60 * (red[j,i] - green[j,i]) / delta[j,i]) + 240) % 360

                # for green
                elif Cmax_arg[j,i] == 1:
                    h[j,i] = ((60 * (blue[j,i] - red[j,i]) / delta[j,i]) + 120) % 360

                # and red
                else:
                    h[j,i] = ((60 * (green[j,i] - blue[j,i]) / delta[j,i]) + 360) % 360
    # the hue channel is halved and made to be integer so I can use the opencv resize function
    # this is the same process opencv does to get the hsi value
    return np.round(h/2).astype(np.uint8), s, i

def get_lbp_descriptor(image, P=8, R=1):
    # we use bitvector and professor Kak's implementation from his book
    # get the hsl representation then resize
    img_h, _, _ = get_hsl(image)
    img = cv2.resize(img_h, (64,64))
    eps = 1e-5
    # this is based on professor Kak's implementation
    # since its a square image, height and width are the same
    r_max = 64 - R
    # we start with a dictionary but then convert it to a numpy array with relative values
    lbp_hist = {t:0 for t in range(P+2)}
    # we set the arrays that contain the sin and cos for the neighbors
    pp = np.arange(P)
    pp_x = R*np.cos(2*np.pi*pp/P)
    pp_y = R*np.sin(2*np.pi*pp/P)
    pp_x[pp_x < eps] = 0
    pp_y[pp_y < eps] = 0

    for j in range(R,r_max):
        for i in range(R,r_max):

            pattern = []
            x_ = i + pp_x
```

```
            y_ = j + pp_y
            for p_x, p_y in zip(x_,y_):
                x_base,y_base = int(p_x),int(p_y)
                x_delta, y_delta = p_x - x_base, p_y - y_base
                if (x_delta < eps) and (y_delta < eps):
                    image_p = float(img[x_base][y_base])
                elif (y_delta < eps):
                    image_p = (1 - x_delta) * img[x_base][y_base] + x_delta * img[x_base+1][y_base]
                elif (x_delta < eps):
                    image_p = (1 - y_delta) * img[x_base][y_base] + y_delta * img[x_base][y_base+1]
                else:
                    image_p = (1 - x_delta)*(1 - y_delta)*img[x_base][y_base] + (1-x_delta)*y_delta*img[x_base][y_base + 1] + x_delta*y_delta*img[x_base+1][y_base+1] + x_delta*(1-y_delta)*img[x_base+1][y_base]
                if image_p >= img[j][i]:
                    pattern.append(1)
                else:
                    pattern.append(0)
            bitv = BitVector.BitVector( bitlist = pattern )
            intvals_for_circular_shifts = [int(bitv << 1) for _ in range(P)]
            minbitval = BitVector.BitVector( intVal = min(intvals_for_circular_shifts), size = P )
            bvruns = minbitval.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
    # now we get the numpy array from the values of the dictionary and normalize the array
    lbp_hist = np.array(list(lbp_hist.values()))

    return lbp_hist/lbp_hist.sum()

def get_gram_descriptor(image, model, coarse=None):
    # we use this function to calculate the gram matrix for the image
    # we start by resizing
    image = cv2.resize(image,(256,256))
    # since we have resnet coarse and fine and vgg we use the variable coarse to handle each case
    # coarse = None is for vgg
    # coarse = True is for coarse resnet
    # coarse = False is for fine resnet
    if coarse == None:
        img_features = model(image)
        # flatten the array
        img_features = img_features.reshape((img_features.shape[0], img_features.shape[1]*img_features.shape[2]))
```

```python
        # get the gram matrix
        gmatrix = img_features @ img_features.T
        # normalize it
        g_max = gmatrix.max()
        # we are only interested in the upper triangular
        return np.triu(gmatrix)/g_max, gmatrix
    # we repeat this process for the other 2 models
    elif coarse:
        img_features, _ = model(image)
        img_features = img_features.reshape((img_features.shape[0], img_features.shape[1]*img_features.shape[2]))
        gmatrix = img_features @ img_features.T
        g_max = gmatrix.max()
        return np.triu(gmatrix)/g_max, gmatrix
    elif not coarse:
        _, img_features = model(image)
        img_features = img_features.reshape((img_features.shape[0], img_features.shape[1]*img_features.shape[2]))
        gmatrix = img_features @ img_features.T
        g_max = gmatrix.max()
        return np.triu(gmatrix)/g_max, gmatrix

def get_adain_features(image, model):
    # we start by resizing the image to 256 256
    image = cv2.resize(image,(256,256))
    # get the output
    img_features = model(image)
    # flatten the output
    img_features = img_features.reshape((img_features.shape[0], img_features.shape[1]*img_features.shape[2]))
    # get the mean and standard deviation per channel (axis = 1)
    img_features_mean = img_features.mean(axis=1)
    img_features_std = img_features.std(axis=1)
    # concatenate both mean and standard deviation,
    # this is our feature vector now
    adain_features = np.hstack((img_features_mean,img_features_std))
    return adain_features

def get_class_lbp(path,P=8, R=1, class_name="cloudy"):
    # I use this function to loop through the class images and get a list of all the descriptor vectors corresponding to that class
    file_list = [x for x in os.listdir(path) if (class_name in x and x.endswith(".jpg"))]
    lbp_feat = []
    test_names = []
    for idx in range(len(file_list)):
        # there is a try here since some of the images cannot be opened as they appear to be gifs
        try:
            img = cv2.imread(os.path.join(path,file_list[idx]))
            lbp_desc = get_lbp_descriptor(img, P=P, R=R)
            lbp_feat.append(lbp_desc)
            # I also append the names here to know which are the misclassified images
```

```
            test_names.append(file_list[idx])
        except:

            print(file_list[idx])

    return np.array(lbp_feat), test_names

def get_class_gram(path,model, class_name="cloudy", coarse=None):
    # I use this function to loop through the class images and get a list of all the gram matrices that correspond to one class
    # similar to the lbp function
    file_list = [x for x in os.listdir(path) if (class_name in x and x.endswith(".jpg"))]
    gram_feat = []
    test_names = []
    for idx in range(len(file_list)):
        try:
            img = cv2.imread(os.path.join(path,file_list[idx]))
            gram_desc,_ = get_gram_descriptor(img, model, coarse=coarse)
            gram_feat.append(gram_desc)
            test_names.append(file_list[idx])
        except:
            print(file_list[idx])
    return np.array(gram_feat), test_names

def get_class_adain(path,model, class_name="cloudy"):
    # this is to get the adain features, similar to the previous lbp and gram matrix features
    file_list = [x for x in os.listdir(path) if (class_name in x and x.endswith(".jpg"))]
    adain_feat = []
    test_names = []
    for idx in range(len(file_list)):
        try:
            img = cv2.imread(os.path.join(path,file_list[idx]))
            adain_desc = get_adain_features(img, model)
            adain_feat.append(adain_desc)
            test_names.append(file_list[idx])
        except:
            print(file_list[idx])
    return np.array(adain_feat), test_names

def create_dataset(train_data, test_data, type="lbp", downsam=2048):
    # we build the dataset for each class in this way

    if type=="lbp":
        # for lbp we just stack the vectors on top of each other
        train_x = np.vstack((train_data["cloudy"],train_data["rain"],train_data["shine"],train_data["sunrise"]))

        test_x = np.vstack((test_data["cloudy"],test_data["rain"],test_data["shine"],test_data["sunrise"]))
        # for the labels we just place as many 0s as images in cloudy class, and so on
```

```python
        train_y = np.hstack((np.array([[0]]).repeat(train_data["cloudy"].shape[0]),
                             np.array([[1]]).repeat(train_data["rain"].shape[0]),
                             np.array([[2]]).repeat(train_data["shine"].shape[0]),
                             np.array([[3]]).repeat(train_data["sunrise"].shape[0])))
        test_y = np.hstack((np.array([[0]]).repeat(test_data["cloudy"].shape[0]),
                             np.array([[1]]).repeat(test_data["rain"].shape[0]),
                             np.array([[2]]).repeat(test_data["shine"].shape[0]),
                             np.array([[3]]).repeat(test_data["sunrise"].shape[0])))
    elif type=="gram":
        # for this we flatten the array, and only sample the first 2048 values to be used as descriptors
        train_x = np.vstack((train_data["cloudy"].reshape(train_data["cloudy"].shape[0],train_data["cloudy"].shape[1]*train_data["cloudy"].shape[2])[:,:downsam],
                              train_data["rain"].reshape(train_data["rain"].shape[0],train_data["rain"].shape[1]*train_data["rain"].shape[2])[:,:downsam],
                              train_data["shine"].reshape(train_data["shine"].shape[0],train_data["shine"].shape[1]*train_data["shine"].shape[2])[:,:downsam],
                              train_data["sunrise"].reshape(train_data["sunrise"].shape[0],train_data["sunrise"].shape[1]*train_data["sunrise"].shape[2])[:,:downsam]))

        test_x = np.vstack((test_data["cloudy"].reshape(test_data["cloudy"].shape[0],test_data["cloudy"].shape[1]*test_data["cloudy"].shape[2])[:,:downsam],
                             test_data["rain"].reshape(test_data["rain"].shape[0],test_data["rain"].shape[1]*test_data["rain"].shape[2])[:,:downsam],
                             test_data["shine"].reshape(test_data["shine"].shape[0],test_data["shine"].shape[1]*test_data["shine"].shape[2])[:,:downsam],
                             test_data["sunrise"].reshape(test_data["sunrise"].shape[0],test_data["sunrise"].shape[1]*test_data["sunrise"].shape[2])[:,:downsam]))

        train_y = np.hstack((np.array([[0]]).repeat(train_data["cloudy"].shape[0]),
                             np.array([[1]]).repeat(train_data["rain"].shape[0]),
                             np.array([[2]]).repeat(train_data["shine"].shape[0]),
                             np.array([[3]]).repeat(train_data["sunrise"].shape[0])))
```

```python
        test_y = np.hstack((np.array([[0]]).repeat(test_data["cloudy"].shape[0]),
                            np.array([[1]]).repeat(test_data["rain"].shape[0]),
                            np.array([[2]]).repeat(test_data["shine"].shape[0]),
                            np.array([[3]]).repeat(test_data["sunrise"].shape[0])))
    elif type=="adain":
        # we just need to stack since its only the mean and std per class
        train_x = np.vstack((train_data["cloudy"],train_data["rain"],train_data["shine"],train_data["sunrise"]))

        test_x = np.vstack((test_data["cloudy"],test_data["rain"],test_data["shine"],test_data["sunrise"]))

        train_y = np.hstack((np.array([[0]]).repeat(train_data["cloudy"].shape[0]),
                            np.array([[1]]).repeat(train_data["rain"].shape[0]),
                            np.array([[2]]).repeat(train_data["shine"].shape[0]),
                            np.array([[3]]).repeat(train_data["sunrise"].shape[0])))
        test_y = np.hstack((np.array([[0]]).repeat(test_data["cloudy"].shape[0]),
                            np.array([[1]]).repeat(test_data["rain"].shape[0]),
                            np.array([[2]]).repeat(test_data["shine"].shape[0]),
                            np.array([[3]]).repeat(test_data["sunrise"].shape[0])))
    return train_x, train_y, test_x, test_y

def get_classified(test_y, test_preds, test_names):
    # I use this to print out correct and incorrect per classifier
    # we get all the indices where each class is
    idx_cloudy = np.argwhere(test_y == 0)[:,0]
    idx_rain = np.argwhere(test_y == 1)[:,0]
    idx_shine = np.argwhere(test_y == 2)[:,0]
    idx_sunrise = np.argwhere(test_y == 3)[:,0]
    # then we loop through each and check if its correct or not, and save it to this dictionary
    # it will only give the final values, but that is ok since we just want one example
    cloudy = {}
    for idx in idx_cloudy:
        if test_y[idx] == test_preds[idx]:
            cloudy["correct"] = test_names[idx]
        else:
            # we have this = to a list since we want the class it was classified as
            cloudy["incorrect"] = [test_names[idx],test_preds[idx]]

```

```
    rain = {}
    for idx in idx_rain:
        if test_y[idx] == test_preds[idx]:
            rain["correct"] = test_names[idx]
        else:
            rain["incorrect"] = [test_names[idx],test_preds[idx]]

    shine = {}
    for idx in idx_shine:
        if test_y[idx] == test_preds[idx]:
            shine["correct"] = test_names[idx]
        else:
            shine["incorrect"] = [test_names[idx],test_preds[idx]]
    sunrise = {}
    for idx in idx_sunrise:
        if test_y[idx] == test_preds[idx]:
            sunrise["correct"] = test_names[idx]
        else:
            sunrise["incorrect"] = [test_names[idx],test_preds[idx]]
    # we calculate the correct number of classifications by summing all
    the True in this array and dividing by the total amount of
    classifications
    correct = (test_y == test_preds).sum()
    accuracy = correct/len(test_y)
    # we just print it
    print("Accuracy: ", accuracy)
    # print correct incorrect pairs
    print(cloudy, rain, shine, sunrise)

def plot_confusion_matrix(svm, test_y, preds, name, classes=["cloudy","
    rain","shine","sunrise"]):
    cm = confusion_matrix(test_y, preds, labels=svm.classes_)
    plt.cla()
    plt.clf()
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                                  display_labels=classes)
    disp.plot()
    plt.savefig(name+"_cm.png", bbox_inches='tight')

def plot_lbp(lbp_descriptor, name):
    # this plots the lbp histogram
    vals = np.arange(len(lbp_descriptor))
    plt.bar(vals, lbp_descriptor, color ='blue',
        width = 0.8)

    plt.xlabel("Encoding")
    plt.ylabel("Frecuency")
    plt.title("LBP Histogram")
    new_name = name + "_lbp.png"
    plt.savefig(new_name, bbox_inches='tight')
    plt.clf()
    plt.cla()

def plot_gram(gram_matrix, name, model):
```

```
    # plots the 512 by 512 gram matrix
    # we add this 1e-10 to prevent having a log(0)
    # we want log scale since the values could be big
    # we are using the unnormalized gram matrix
    gram_matrix += 1e-10
    gram_matrix = np.log10(gram_matrix)
    plt.clf()
    plt.cla()
    plt.imshow(gram_matrix, cmap="gray")
    plt.colorbar()
    plt.title("Gram Matrix")
    new_name = name + "_gram_" + model + ".png"
    plt.savefig(new_name, bbox_inches='tight')
    plt.clf()
    plt.cla()

img_path = "data/training"
test_path = "data/testing"
classes = ["cloudy","rain","shine","sunrise"]
class_dict = {"cloudy": 0, "rain": 1, "shine": 2, "sunrise": 3}

# LBP
lbp_cfeat = {}
lbp_test = {}
P = 12
R = 2
names = {}
for cls in classes:
    lbp_cfeat[cls], _ = get_class_lbp(img_path, P=P, R=R, class_name=cls)
    lbp_test[cls], names[cls] = get_class_lbp(test_path, P=P, R=R,
   class_name=cls)
test_filenames = names["cloudy"] + names["rain"] + names["shine"] + names[
   "sunrise"]

lbp_train_x, lbp_train_y, lbp_test_x, lbp_test_y = create_dataset(
   lbp_cfeat, lbp_test, type="lbp")

svm_lbp = svm.SVC()
svm_lbp.fit(lbp_train_x, lbp_train_y);

lbp_preds = svm_lbp.predict(lbp_test_x)

get_classified(lbp_test_y, lbp_preds, test_filenames)

plot_confusion_matrix(svm_lbp, lbp_test_y, lbp_preds, "lbp");

# GRAM MATRIX
vgg = VGG19()
vgg.load_weights('vgg_normalized.pth')
encoder_name='resnet50'
resnet = CustomResNet(encoder=encoder_name)

gram_cfeat_resnet_coarse = {}
gram_test_resnet_coarse = {}
```

```python
names_resnetc = {}

gram_cfeat_resnet_fine = {}
gram_test_resnet_fine = {}
names_resnetf = {}

gram_cfeat_vgg = {}
gram_test_vgg = {}
names_vgg = {}

for cls in classes:
    gram_cfeat_resnet_coarse[cls],_ = get_class_gram(img_path,resnet, class_name=cls, coarse=True)
    gram_test_resnet_coarse[cls], names_resnetc[cls] = get_class_gram(test_path,resnet, class_name=cls, coarse=True)

    gram_cfeat_resnet_fine[cls], _ = get_class_gram(img_path,resnet, class_name=cls, coarse=False)
    gram_test_resnet_fine[cls], names_resnetf[cls] = get_class_gram(test_path,resnet, class_name=cls, coarse=False)

    gram_cfeat_vgg[cls], _ = get_class_gram(img_path,vgg, class_name=cls, coarse=None)
    gram_test_vgg[cls], names_vgg[cls] = get_class_gram(test_path, vgg, class_name=cls, coarse=None)

resnetc_train_x, resnetc_train_y, resnetc_test_x, resnetc_test_y = create_dataset(gram_cfeat_resnet_coarse, gram_test_resnet_coarse, type="gram")
resnetf_train_x, resnetf_train_y, resnetf_test_x, resnetf_test_y = create_dataset(gram_cfeat_resnet_fine, gram_test_resnet_fine, type="gram")
vgg_train_x, vgg_train_y, vgg_test_x, vgg_test_y = create_dataset(gram_cfeat_vgg, gram_test_vgg, type="gram")

svm_resnetc = svm.SVC()
svm_resnetc.fit(resnetc_train_x, resnetc_train_y);
resnetc_preds = svm_resnetc.predict(resnetc_test_x)

get_classified(resnetc_test_y, resnetc_preds, test_filenames)

plot_confusion_matrix(svm_lbp, lbp_test_y, resnetc_preds, "resnetc");

svm_resnetf = svm.SVC()
svm_resnetf.fit(resnetf_train_x, resnetf_train_y);
resnetf_preds = svm_resnetf.predict(resnetf_test_x)

get_classified(lbp_test_y, resnetf_preds, test_filenames)

plot_confusion_matrix(svm_lbp, lbp_test_y, resnetf_preds, "resnetf");

svm_vgg = svm.SVC()
svm_vgg.fit(vgg_train_x, vgg_train_y);
vgg_preds = svm_vgg.predict(vgg_test_x)
```

```

get_classified(lbp_test_y, vgg_preds, test_filenames)

plot_confusion_matrix(svm_lbp, lbp_test_y, vgg_preds, "vgg");

adain_train_vgg = {}
adain_test_vgg = {}
names_vgg = {}

for cls in classes:
    adain_train_vgg[cls], _ = get_class_adain(img_path,vgg, class_name=cls
    )
    adain_test_vgg[cls], names_vgg[cls] = get_class_adain(test_path, vgg,
    class_name=cls)

adain_train_x, adain_train_y, adain_test_x, adain_test_y = create_dataset(
    adain_train_vgg, adain_test_vgg, type="adain")

svm_adain= svm.SVC()
svm_adain.fit(adain_train_x, adain_train_y);
adain_preds = svm_adain.predict(adain_test_x)

get_classified(lbp_test_y, adain_preds, test_filenames)

plot_confusion_matrix(svm_lbp, lbp_test_y, adain_preds, "adain");

ex_path = "data/training"
examples = ["cloudy1.jpg","rain1.jpg","shine1.jpg","sunrise1.jpg"]

for example in examples:
    basename = example[:-4]
    img = cv2.imread(os.path.join(ex_path, example))
    # plot lbp
    lbp_desc = get_lbp_descriptor(img, P=P, R=R)
    plot_lbp(lbp_desc, basename);

    # plot gram
    _, gram_matrix = get_gram_descriptor(img, vgg, coarse=None)
    plot_gram(gram_matrix, basename, "vgg");

    _, gram_matrix = get_gram_descriptor(img, resnet, coarse=False)
    plot_gram(gram_matrix, basename, "resnet_fine");

    _, gram_matrix = get_gram_descriptor(img, resnet, coarse=True)
    plot_gram(gram_matrix, basename, "resnet_coarse");
```

Listing 1: Source code