CMPE 597 Term Project Final Report

Korhan Polat

May 28, 2018

Aim: The aim of my project is to come up with a texture classification scheme which utilizes correlations of feature activations of a pretrained CNN, as described in [1].

I. Introduction

In 2015, Gatys et al. came up with *Neural style transfer* algorithm [1], which utilizes correlation of CNN's feature maps as style representation to match style of an artwork to generated image. They also showed that the correlation features can be used to generate textures [2]. Motivated by their findings, in this project I tried to make use of feature correlations in the context of texture classification.

VGG-19 network [9] that is trained on ImageNet is used as feature extractor. The correlations between intra-layer filter responses are computed as Gram matrices. If the activations for i^{th} filter at k^{th} position of l^{th} layer are denoted as F^l_{ik}, then the Gram matrix that captures the correlations at l^{th} layer is defined as

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk}. \tag{1}$$

Using Gram matrices can be seen as an orderless pooling mechanism on local image descriptors. Orderless means that it discards the spatial configuration of input features so that it becomes invariant to spatial permutations [5].

II. Related Work

Some of the traditional texture classification methods utilize local binary patterns, gray level co-occurrence matrices etc. With the advances in deep learning in image recognition, texture classification methods started to employ CNN structures. Some of the recent approaches are FV-CNN [5], Texture-CNN [7] and Wavelet CNN [8]. FV-CNN combined patch-based classification using the fisher vectors encoding method, on feature extracted with pretrained CNN's, has been able to outperform state-of-the-art methods, such as standard CNNs and SIFT features [3]. Texture-CNN method averages the outputs of each feature map and uses them for classification. Wavelet CNN performs multi-resolution analysis using energy layer based on Haar wavelets, with trainable parameters. Although the later two approaches does not attain state-of-the-art performances, they are remarkable in the sense that they use less number of parameters, compared to VGG driven FV-CNN method.

III. Datasets

I've used two different publicly available datasets for model evaluation.

DTD: (Describable Texture Dataset) This dataset [4] contains 5640 images of 47 classes. This dataset differs from other texture dataset as the classes are not named after the material of the object but rather how it would be described (veined, marbled, zigzagged etc.). The dataset is composed of 10 splits in which all images are divided equally into train, validation and test sets. So for example in a split, there are 1880 training, 1880 validation and 1880 test images. The results are averaged over 10 splits for bench-marking. The state-of-the-art accuracy is 72.3% by [5].

kth-tips-2b: (Textures under varying Illumination, Pose and Scale) [6] contains 11 classes of 432 texture images. Each class consists of four samples and each sample has 108 images. Each sample is used for training once while the remaining three samples are used for testing. The state-of-the-art accuracy is 81.5% by [5].

IV. Experiments

As described in my previous midterm report, I've found that the correlations extracted from 3rd convolutional layer of 5th block yielded the best classification result (58.8%) on DTD's first split with performing PCA and then using SVM as classifier. Therefore I used features extracted from Conv5-3 layer of VGG-19 when I've experimented with different models.

My first attempt was to use Conv5-3 layer to compute Gram matrix and feed it to fully connected layers to perform classification. However, since vectorized upper triangle of 512x512 Gram matrix has 131328 elements, my algorithm raised memory errors. Therefore I tried to come up with solutions which reduce dimensionality of feature correlations. Below are the descriptions of the methods that I've tried.

A. Using 1x1 convolutions to reduce feature dimension

Conv5-3 yields 512x14x14 dimensional tensors. I added another convolutional layer with kernel size 1x1 and output channels of d_{out} to reduce number of feature channels. Then the vectorized Gram matrix using d_{out} channels are fed into fully connected layers, with 4096 and 2048 hidden layers respectively and dropout ratio of 0.3. The diagram of this experiment set up is shown in Fig. 1.

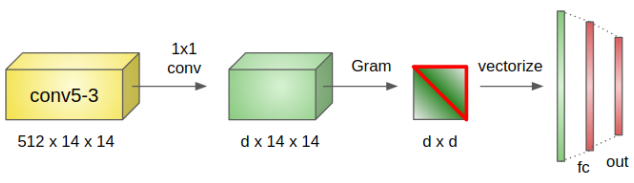

Fig. 1. Diagram of first experiment

I've tried different number of output channels for 1x1 convolution layer. Performance evaluations of these experiments

are shown in Fig. 2 for first split of DTD dataset. It can be seen that the model overfits after fifth epoch and even though using more output channels improves training loss and accuracy, there is not a significant improvement on validation set.

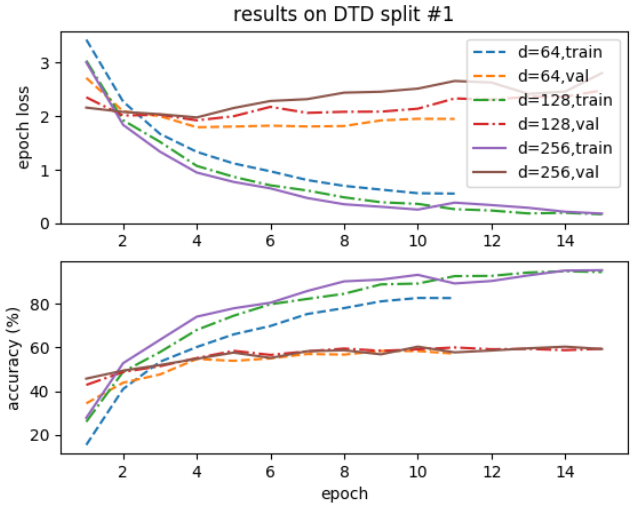

Fig. 2. Training and validation loss and accuracy per epoch for different number of output channels.

I also evaluated the performance of 1x1 convolutions without computing correlations, in order to compare effectiveness of using Gram matrices. The results for reducing to 128 channels with and without computing correlations are given in Fig. 3. Here, it can be observed that using Gram matrices does not yield any performance improvement. This might be due to the some of the classes on DTD clearly show shape of objects and the network without correlations might have learned these shapes instead of textures. Therefore I run the same comparison on a different dataset and again, the results (Fig. 4) are not in favor of using correlations.

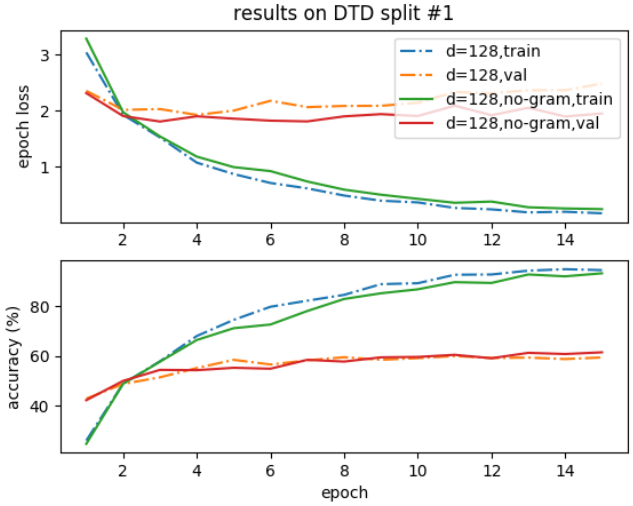

Fig. 3. Comparison of 128d features using Gram matrices and without Gram matrices on DTD dataset.

I could've tried increasing dropout ratio or some other regularization technique but instead I decided to try another design.

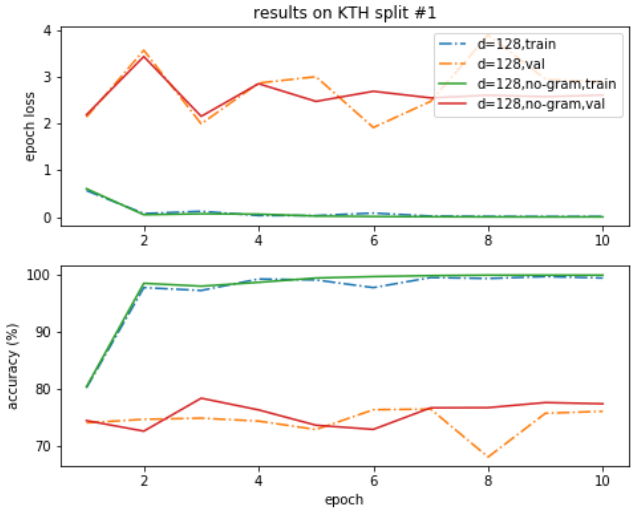

Fig. 4. Comparison of 128d features using Gram matrices and without Gram matrices on KTH dataset.

B. *Summing Gram matrix along an axis*

For this experiment, Gram matrices are extracted from VGG the usual way. After d x d Gram matrix is obtained, column-wise sum is taken so that new dimension is 1 x d. Taking column-wise sum is similar to energy layer in Texture-CNN. Energy layer averages a filter's output across spatial dimensions and gives a value per filter that signifies the filter's activation level. Similarly, summing over correlation matrix signifies how much a filter is activated together with other filters at the same spatial locations, in other words, how much that filter is correlated with other filters. A diagram that explains the model is shown in Fig. 5. Here, 5 feature correlation maps are extracted from last layers of each block of VGG. Number of filters for these layers are 64, 128, 256, 512, 512 respectively. Concatenated sum vectors (d=1472) are fed into fully connected layers with 4096 and 2048 hidden units with ReLU activations and dropout ratio of 0.5. I tested this model on KTH dataset's all splits for 2 runs and acquired average cross-validation accuracy of 72.8% after 5 epochs of training.

C. *Convolving Gram Matrices*

In this experiment, we acquire d x d Gram matrix for a given layer. Then the matrix is convolved with a kernel of size d x 1 so that the output is d x 1 dimensional. For example let's say we have a correlation matrix from fifth layer, which has a size of 512 x 512. Then this matrix is convolved with a kernel of 512 x 1 so that the output layer has 512 activations. Each weight of this convolutional kernel is associated with filter i's correlation with other filters. Therefore if the correlations of i^{th} filter is not important, kernel's i^{th} weight will be small. If the weights of this kernel are all set to 1, the output is equivalent to summing over columns, as in the previous method. The diagram for this experiment is shown in Fig. 6. For this experiment, 5 feature correlation maps are extracted from last layers of each block of VGG as in the previous experiment.

One hyperparameter of this model is number of output channels after Gram matrix is convolved, in other words,

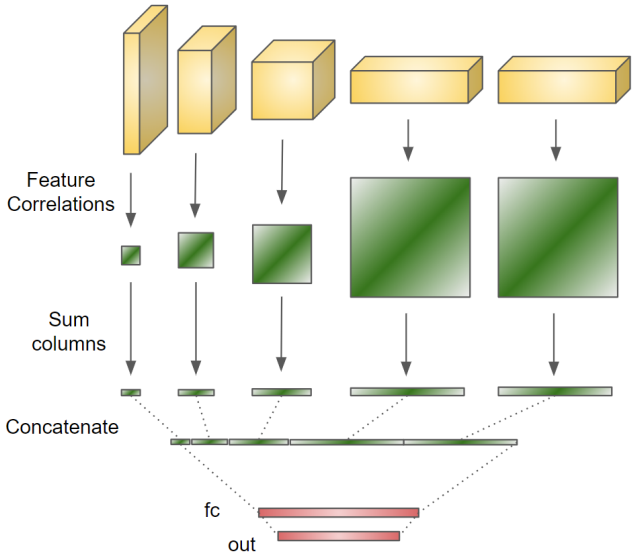

Fig. 5. Diagram of the second experiment, where each correlation matrix is summed along an axis and fed to fully connected layers.

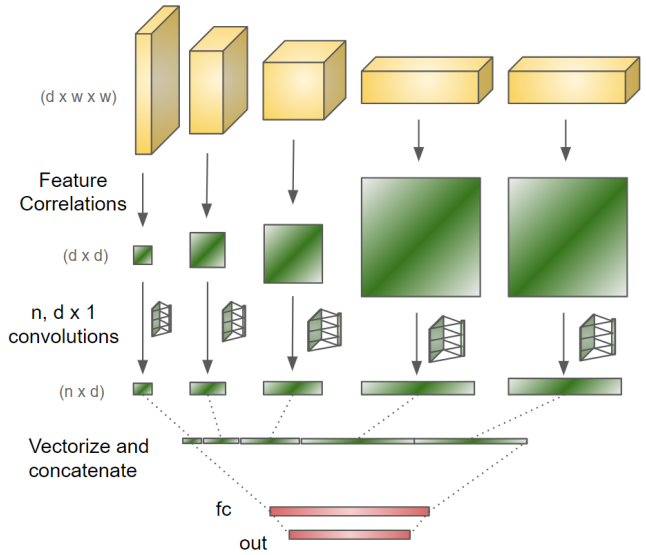

Fig. 6. Diagram of the third experiment, where each correlation matrix is convolved with a kernel of size d x 1, and fed to fully connected layers.

number of different kernels to convolve on a Gram matrix. In order to find optimum number of channels, the model is run on KTH first split for 2 runs. The results are shown in Fig. 7 for validation set only. Unfortunately it is not possible to make a significant decision on optimum number of output channels because the validation scores does not present any trend. Nevertheless, I evaluated this models' performance choosing 2 output filters for Gram convolutions, as shown in Fig. 8.

Learning: I've used PyTorch library. VGG-19 trained with ImageNet is used for feature extraction. VGG weights are frozen so that no gradient is computed for those layers. Data augmentation with random horizontal flip is performed during training and random crop of 224 pixels are fed into model during training and center crop images are fed during testing. I've tried Stochastic Gradient Descent and Adam optimizer and found that Adam optimizer converges

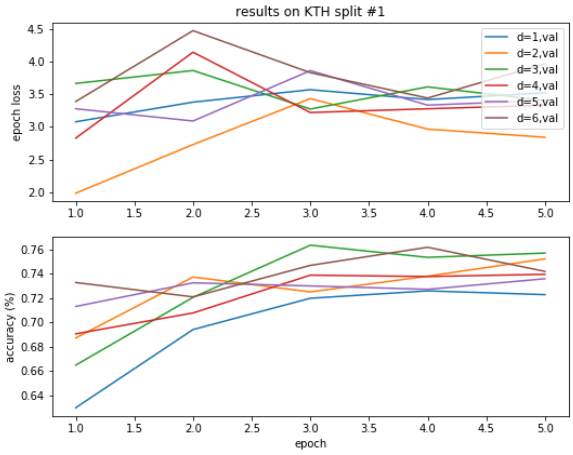

Fig. 7. Validation losses and accuracy for different number of kernels convolved on correlation matrices. Results are averaged over two runs. No significant trend is observed.

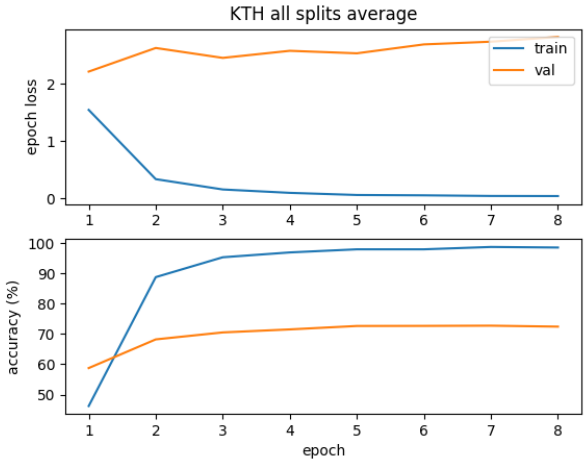

Fig. 8. Validation losses and accuracy for 2 kernels convolved on correlation matrices. Results are averaged over two runs.

much faster, therefore the reported results are obtained using Adam optimizer. I had to manually adjust dropout ratio of fully connected layers, according to dataset size and model complexity.

V. Results

I evaluated the performance of two of my models on DTD and kth-tips-2b datasets. The results are shown in Table V. 'Gram-sum' method refers to my second model where the correlation matrices are summed over an axis and 'Gram-conv' refers to my third model in which correlation matrices are further convolved with kernels of same width but unit height. I did not include the benchmark results for 'Gram-sum' method on DTD dataset because I could not tune the parameters so that the model converges in a reasonable amount of time for all splits.

In comparison to other methods, my models did not achieve significantly better performance in terms of classification accuracy. State-of-the-art performance belongs to FV-CNN [5] method on both datasets. However authors of [8] speculate

TABLE I
ACCURACY (%) COMPARISON TO OTHER METHODS

	T-CNN	Wavelet-CNN	FV-CNN	VGG-19	ResNet-18	Gram-sum	Gram-conv
DTD	55.8	59.8	72.3	62.9	60.3	-	62.8
kth-tips-2b	72.4	74.2	81.8	75.4	71.6	72.8	72.4
Size on disk (MB)	89	44	-	550	45	128	153

that FV-CNN has a significantly larger number of trainable parameters than their models (Wavelet-CNN) and therefore it might have learned non-texture properties too. Therefore it is meaningful to compare model sizes along with accuracy. On that matter, my models occupy 128 MB and 153 MB on disk, smaller than VGG-19 and presumably smaller than FV-CNN which uses VGG-19. Although my models use VGG-19 as feature extractor, the fully connected layers are reduced and therefore occupies less memory.

VI. CONCLUSION

In this project, I've learned how to perform transfer learning and how to modify existing CNN architectures. I observed that choosing the right optimizer and hyper-parameters have huge impact on the model performance. Some trials performed very well due to weight initialization but those good results are not occurring repeatedly.

I could have used ResNet18 for feature extraction but according to some forum posts that I've came across, it is agreed that residual networks do not perform well on image style transfer. Therefore I did not experiment with using ResNet features, even though it would be much faster and lightweight.

One of the weaknesses of my analysis is that I did not examine the classes that are mostly classified wrong. Another weakness is I did not performed my experiments in fully controlled manner; by fully controlled I mean that only one parameter at a time should be changed so that there exists a control group to validate the tried method.

Texture synthesis of Gatys et al. [2] works fine for regenerating textures, but in this project I could not achieve the results that I expected. This is mostly due to the heterogeneous nature of intra-class examples on the dataset. Another reason is I did not analyze how efficiently the weights are trained, it might be the case that some of the layers I introduced did not actually learn any meaningful features except fully connected layers.

REFERENCES

[1] L. Gatys, A. S. Ecker, and M. Bethge. A Neural Algorithm of Artistic Style. *arXiv:1508.06576[cs.CV]*. August 2015

[2] L. Gatys, A. S. Ecker, and M. Bethge. Texture synthesis using convolutional neural networks. In *Advances in Neural Information Processing Systems 28*, pages 262–270. Curran Associates, Inc., 2015

[3] P. Cavalin, L. S. Oliveira. A Review of Texture Classification Methods and Databases. In *30th SIBGRAPI Conference on Graphics, Patterns and Images Tutorials (SIBGRAPI-T)*.p1-8. 2017

[4] M.Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, A. Vedaldi. Describing Textures in the Wild in *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[5] M. Cimpoi, S. Maji, and A. Vedaldi. Deep filter banks for texture recognition and segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015. 2, 6

[6] E. Hayman, B. Caputo, M. Fritz, and J.-O. Eklundh. On the significance of real-world conditions for material classification. In *Computer Vision-ECCV 2004*, pages 253–266. Springer, 2004.

[7] V. Andrearczyk and P. F. Whelan. Using filter banks in convolutional neural networks for texture classification. *Pattern Recognition Letters*, 84:63 – 69, 2016. 2, 5, 6

[8] S. Fujieda, K. Takayama, and T. Hachisuka. Wavelet Convolutional Neural Networks for Texture Classification. *ArXiv e-prints*, Jul. 2017.

[9] K. Simonyan, A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR,* abs/1409.1556, 2014

APPENDIX

```
""" model_loading.py """

import torch.nn as nn
import torchvision
import numpy as np
import torch
from os.path import join,dirname,abspath

class GramMatrix(nn.Module):
    """ compute correlations of a tensor"""
    def forward(self, input):
        b,c,h,w = input.size()
        F = input.view(b, c, h*w)
        G = torch.bmm(F, F.transpose(1,2))
        G.div_(h*w)
        return G


class VectorizeTriu(nn.Module):
    """ vectorize upper triangle of an input matrix """
    def forward(self, x):
        row_idx, col_idx = np.triu_indices(x.shape[2])
        row_idx = torch.LongTensor(row_idx).cuda()
        col_idx = torch.LongTensor(col_idx).cuda()
        x = x[:, row_idx, col_idx]
        return x


class Vgg_FeatExtr(nn.Module):
    """ extract features for Conv5-3 layer """
    def __init__(self,freeze=True):
        super(Vgg_FeatExtr,self).__init__()
        vgg_model = torchvision.models.vgg19(pretrained=True)
        # freeze parameters not to train
        if freeze:
            for i, param in vgg_model.named_parameters():
                param.requires_grad = False

        self.Conv5 = nn.Sequential(
            *list(vgg_model.features.children())[:34])

    def forward(self,x):
        x = self.Conv5(x)
        return x


class FC_layer(nn.Module):
    """ classifier that is used as a template for all my models """
    def __init__(self,d_in,d_out,h1=4096,h2=2048,dropout_p=0.3):
        super(FC_layer,self).__init__()

        fc_features = [nn.Linear(in_features= d_in, out_features=h1, bias=True),
                       nn.ReLU(inplace=True),
                       nn.Dropout(p=dropout_p),
                       nn.Linear(in_features=h1, out_features=h2, bias=True),
                       nn.ReLU(inplace=True),
                       nn.Dropout(p=dropout_p),
                       nn.Linear(in_features=h2, out_features=d_out, bias=True)]

        self.fc = nn.Sequential(*fc_features)

    def forward(self,x):
        pred = self.fc(x)
        return pred


class GramClassifier(nn.Module):
    """ model that is used for experiment #1 """
    def __init__(self,d_ch,n_class,freeze=True):
        super(GramClassifier,self).__init__()
        self.d_ch = d_ch
        self.vgg_feat = Vgg_FeatExtr(freeze=freeze)

        self.conv_channels = nn.Conv2d(512, d_ch, 1)

        self.gram = GramMatrix()
        self.vectorize = VectorizeTriu()

        self.fc = FC_layer(d_in=d_ch*(d_ch-1)/2+d_ch,d_out=n_class,h1=4096,h2=2048)

    def load_trained_conv(self,name):
        self.conv_channels.load_state_dict(torch.load(join(dirname(abspath(__file__)),'saved_models',name)))

    def forward(self,x):
        x = self.vgg_feat(x)
        x = self.conv_channels(x)
        self.x = self.gram.forward(x)
        x = self.vectorize(self.x)
        pred = self.fc(x)

        return pred
```

```
class GramSum5(nn.Module):
    """ model that is used for experiment #2 """
    def __init__(self,n_class,freeze=True):
        super(GramSum5,self).__init__()
        vgg_model = torchvision.models.vgg19(pretrained=True)
        if freeze:
            for i, param in vgg_model.named_parameters(): param.requires_grad = False
        # extract features from 5 different layers
        self.Conv1 = nn.Sequential(*list(vgg_model.features.children())[0:4])
        self.Conv2 = nn.Sequential(*list(vgg_model.features.children())[4:9])
        self.Conv3 = nn.Sequential(*list(vgg_model.features.children())[9:18])
        self.Conv4 = nn.Sequential(*list(vgg_model.features.children())[18:27])
        self.Conv5 = nn.Sequential(*list(vgg_model.features.children())[27:34])

        self.gram = GramMatrix()

        self.fc = FC_layer(d_in=512+512+256+128+64,d_out=n_class,h1=4096,h2=2048,dropout_p=0.5)

    def forward(self,x):
        # extract features
        g1 = self.Conv1(x)
        g2 = self.Conv2(g1)
        g3 = self.Conv3(g2)
        g4 = self.Conv4(g3)
        g5 = self.Conv5(g4)
        # compute correlations
        g1 = self.gram.forward(g1)
        g2 = self.gram.forward(g2)
        g3 = self.gram.forward(g3)
        g4 = self.gram.forward(g4)
        g5 = self.gram.forward(g5)
        # sum along axis
        g1 = torch.sum(g1,dim=1)
        g2 = torch.sum(g2,dim=1)
        g3 = torch.sum(g3,dim=1)
        g4 = torch.sum(g4,dim=1)
        g5 = torch.sum(g5,dim=1)
        # concatenate
        out = torch.cat((g1,g2,g3,g4,g5),1)

        pred = self.fc(out)
        return pred


class GramConv5(nn.Module):
    """ model that is used for experiment #3 """
    def __init__(self,n_class,d_conv=1,freeze=True):
        super(GramConv5,self).__init__()
        vgg_model = torchvision.models.vgg19(pretrained=True)
        if freeze:
            for i, param in vgg_model.named_parameters(): param.requires_grad = False

        self.Conv1 = nn.Sequential(*list(vgg_model.features.children())[0:4])
        self.Conv2 = nn.Sequential(*list(vgg_model.features.children())[4:9])
        self.Conv3 = nn.Sequential(*list(vgg_model.features.children())[9:18])
        self.Conv4 = nn.Sequential(*list(vgg_model.features.children())[18:27])
        self.Conv5 = nn.Sequential(*list(vgg_model.features.children())[27:34])

        self.gram = GramMatrix()
        # define dx1 convolutions
        self.gramconv1 = nn.Conv2d(1,d_conv,kernel_size=(64,1))
        self.gramconv2 = nn.Conv2d(1,d_conv,kernel_size=(128,1))
        self.gramconv3 = nn.Conv2d(1,d_conv,kernel_size=(256,1))
        self.gramconv4 = nn.Conv2d(1,d_conv,kernel_size=(512,1))
        self.gramconv5 = nn.Conv2d(1,d_conv,kernel_size=(512,1))

        self.relu = nn.ReLU()

        self.fc = FC_layer(d_in=d_conv*(512+512+256+128+64),d_out=n_class,h1=4096,h2=2048,dropout_p=0.3)

        self.d_conv = d_conv

    def forward(self,x):

        g1 = self.Conv1(x)
        g2 = self.Conv2(g1)
        g3 = self.Conv3(g2)
        g4 = self.Conv4(g3)
        g5 = self.Conv5(g4)

        g1 = self.gram.forward(g1)
        g2 = self.gram.forward(g2)
        g3 = self.gram.forward(g3)
        g4 = self.gram.forward(g4)
        g5 = self.gram.forward(g5)
        # convolve Gram matrices
        g1 = self.relu(self.gramconv1(g1.unsqueeze_(1)).view(-1,self.d_conv*64))
        g2 = self.relu(self.gramconv2(g2.unsqueeze_(1)).view(-1,self.d_conv*128))
        g3 = self.relu(self.gramconv3(g3.unsqueeze_(1)).view(-1,self.d_conv*256))
        g4 = self.relu(self.gramconv4(g4.unsqueeze_(1)).view(-1,self.d_conv*512))
        g5 = self.relu(self.gramconv4(g5.unsqueeze_(1)).view(-1,self.d_conv*512))
        out = torch.cat((g1,g2,g3,g4,g5),1)
        pred = self.fc(out)
        return pred
```

```
""" model training """

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import time
import os
from torchvision import transforms,datasets
import copy
from os.path import join,dirname,abspath
from model_loading import GramClassifier,GramSum5,GramConv5
from my_utils import MyLogger


def train_model(model, criterion, optimizer, scheduler, num_epochs=25,Logger=None,pretrain=True,previous_epochs=0):
    since = time.time()

    best_acc = 0.0
    phase_list = ['train', 'val']

    for epoch in range(1,num_epochs+1):
        print('Epoch {}/{}'.format(epoch, num_epochs))
        print('-' * 10)
        epoch_start = time.time()

        # Each epoch has a training and validation phase
        for phase in phase_list:
            if phase == 'train':
                scheduler.step()
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for itr, data_inp_lab in enumerate(dataloaders[phase]):
                inputs, labels = data_inp_lab

                inputs = inputs.to(device)
                labels = labels.to(device)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
                if itr%50 ==  0: print(itr,running_loss)

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            print('{} Loss: {:.4f} Acc: {:.4f} Duration: {:.2f}'.format(
                phase, epoch_loss, epoch_acc, time.time()-epoch_start))

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())



    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print('Best val Acc: {:4f}'.format(best_acc))

    # load best model weights
    model.load_state_dict(best_model_wts)


    return model,state


if __name__ == '__main__':

    """ parameters """

    for split_i in range(1,5):
```

```
rootdir = dirname(abspath(__file__))
data_dir = join(rootdir,'kth','sp'+str(split_i))
n_class = 11

model_name = 'gram_sum_5'
exp_name = 'kth_gram_sum_5'

epoch_train = 5
criterion = nn.CrossEntropyLoss()

""" data loading """
batch_size = 32

data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])}

image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x]) for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size,
                                          shuffle=True, num_workers=4) for x in ['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes

""" ===========
load the model
=========== """


if 'gram_conv' in model_name : model_ft = GramConv5(n_class=n_class,d_conv=d_conv)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model_ft = model_ft.to(device)
# set parameters
Learning_rate = 0.002
momentum = .9
# optimizer_ft = optim.SGD(filter(lambda p: p.requires_grad,model_ft.parameters()),lr=Learning_rate,momentum=momentum)
optimizer_ft = optim.Adam(filter(lambda p: p.requires_grad,model_ft.parameters()))
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=1, gamma=0.7)

model_ft,state = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                             num_epochs=epoch_train,Logger=Logger,pretrain=False)

torch.save(model_ft.state_dict(),join(dirname(abspath(__file__)),'saved_models',model_name+'_'+str(split_i)))
```