CMPE - 685

Computer Vision

Final Project

Detecting Building Damage

Authors:
Ryan Malley
Janardhan Chaudhari

Instructor:
Dr.Andreas Savakis

April 30, 2019

Contents

List of Figures

1 Abstract

The goal of this project is to detect small scale damage to infrastructure such as buildings, wall, and roads. While large scale damage is easy to detect from visual inspection, such as a collapsed wall or a large pothole, many of these catastrophic events could be prevented by noticing smaller chips and cracks before they escalate. Two datasets, one a collection of pavement cracks and another a set of wall cracks, were found that match the small-scale requirements of the project.This project used two network architecture VGG16 and GooLeNet for training on the two dataset for binary classification of "Crack" and "No Crack".

2 Introduction

2.1 Problem Statement

Cracks are the common defects that are found in various physical structures,like road pavement, walls of the building. For the maintenance of these Infrastructure detecting cracks is an important thing.More better option is to repair the cracks when they are like hairline. Fixing a crack before its declension can reduce the cost of maintenance.

To detect the hairline crack, we need to automate the process since this crack are hard to detect visually. For the detection, A deep learning based network architecture would solve this easily. The pretrained models like VGG16 and GoogleNet Convolutional Neural Network is used to detect the cracks.

2.2 Related Work

A large number of the recent literature in crack detection and classification of surface with demonstrates an increasing interest with the research area. we went through many research papers, one paper mentioned using the deep crack dataset with a novel end to end convolutional neural network. They fused the multi-feature data and created the fusion map for detection of the

map. The architecture they used was SegNet. Though their Fscore was high.[2]

Another research mentioned the use of the various image processing techniques to detect the defects on the cracks in steel and concrete surfaces. They used convolutional neural network(CNNs) with 8 layers. For edge detection, they passed there dataset through canny edge detector and sobel edge detector and ran through the CNNs.[1]

In the different research the for road crack detection using Deep CNN. As a classified using support vector machine and made it realtime.[3]

For the research of the automatic detector of the cracks, marked the cracks manually and ran that picture through a CNN with 5 classes. Also concluded that the cracks were well detected using there method.[4]

We thought of using the Google Net and VGG16 Network architecture for getting good accuracy and based on the literature survey.

3 Methodology

This project will use two neural network architecture the GoogLeNet CNN model and VGG16 CNN model in a Python environment. Anaconda and Spyder will be used for the Python development environment for their optimization in machine learning. as the main problem will focus on finding continuous dark edges among smooth uniform surfaces. The "GoogLeNet" model is represented by the Inception V3 model in PyTorch's library.

The network will be trained on a subset composed of the first 90% of the images in each category. The other 10% will be reserved as a testing set. By the end of the project, we should have two trained CNN that are able to identify with a high degree of accuracy whether a surface has significant structural damage.

3.1 Datasets

Two datasets were used for this project. Both displayed cracks and structural damage in infrastructure. The first was a set of 40,000 images of pavement and stone walkways. Half of the images in the dataset contained cracks or

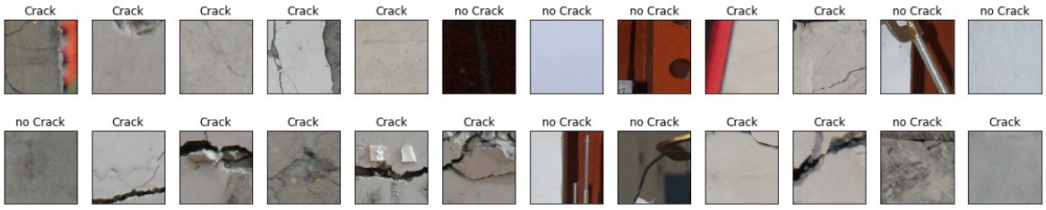

Figure 1: Dataset

other damage to the area. It was the more robust of our two datasets, and provided a strong test set of images upon which to train our models. The quality of the images were strong, though all of the images were hand classified, which draws into question some of the more ambiguous classifications for damage that is extremely small or hard to see.

The second dataset was a smaller compilation of 1,400 images of damage to walls. This dataset not only contained much fewer images, but also had much more varying background and lighting characteristics. While edge detection could have possibly helped us identify damage in the previous dataset, this one caused us to reevaluate its usage, as the walls often had non-crack objects that would have set off the edge detection filter without actually being cracks, like electrical cables, door knobs, and paving caulk. The varying lighting conditions, backgrounds, and extraneous objects made classification a more difficult task due to a larger number of variables that the networks had to ignore to find the actual damage.

3.2 Dependencies

Our solution is dependant on a number of python libraries that would need to be installed on a host machine to run our projects code. We retrieved the network models using PyTorch's implementation. PyTorch also gave us the tools to train and validate our modified networks, as well as structuring the data that it would be trained on. In order to plot sample images from our training data, we used matplotlib. Any other dependencies for our code should be already provided from pythons normal installation.

3.3 Implementation Decisions

We explored a few different options related to the concepts we've tried in class to help with our classification process, namely edge detection. Our initial though was that because our classes involved cracks and tears in uniform surfaces, an edge detection filter ran before the inputs were sent into the networks hidden layers could help remove extraneous information from the system so it could more easily detect cracks. However, upon closer inspection of our datasets, many of the backgrounds were not uniform at all, with electrical wiring, doors, and textures being included in addition to the possible building damage. After discussion, we decided that it would not be beneficial to add edge detection to the system.

Our implementation used the transfer learning method provided during a previous homework assignment courtesy of PyTorch's tutorials.

Because PyTorch's documentation and code fulfilled all of the needs of our code, few changes were made outside of specific parameters related to the training and validation process were made.

3.4 GoogLeNet

GoogLeNet is based on the Inception CNN model. The main problem that the Inception model sought to solve was finding the salient portion of an image, the parts that are relevant to the classification of the image. For instance, a horse in the background of an image and a horse in the foreground of an image should both be matched by the classifier. However, considering this problem, finding a kernel of the correct size to match the salient portion is difficult. Inception's solution is to include multiple kernel sizes within the same convolutional layer to match different areas. These convolutions are then concatenated and pooled the next layer over. Because the network is very deep, it uses a system of auxillary classifiers that can store preliminary classifications. The loss function of Inception is based not only on the true output classification, but also as a weighted sum of the auxiliary classifications. This also means that performing transfer learning is more difficult, as the auxiliary classifiers can't be frozen due to the weighted sum involved in the loss function.

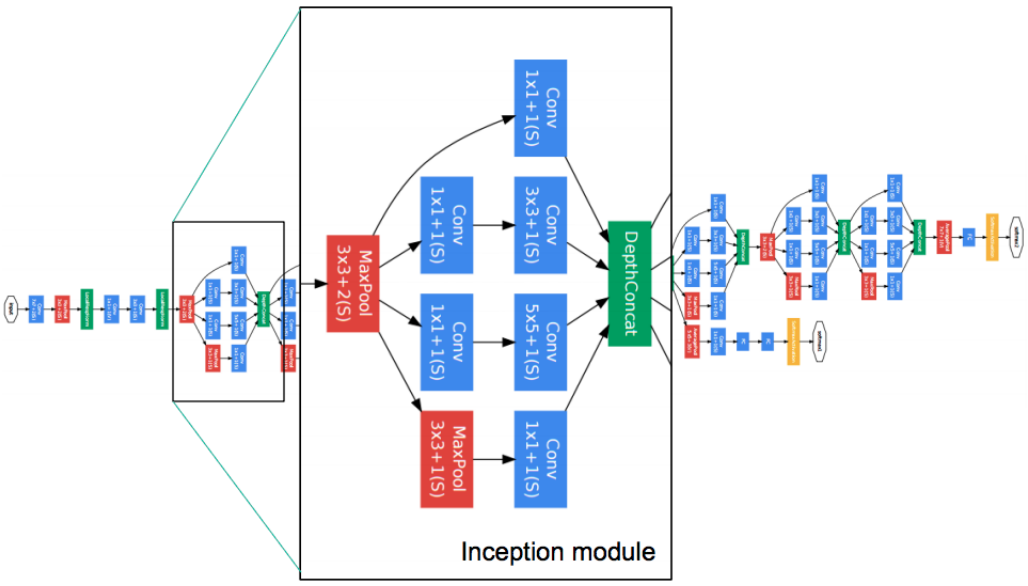

Figure 2: GoogLeNet Architecture

3.5 VGG-16

VGG16 is a convolutional neural network proposed by K.Simonyan and A. Zisserman from the university of Oxford in the paper "Very Deep Convolutional networks for Large Scale Image Recognition". The model achieved around 92 percent accuracy on ImageNet dataset. The model was submitted to ILSVRC-2014.

The Input to Convolution layer 1 is 224 X 224 RGB Image. The image is passed through the stack of convolutional (conv.) layers, where the filters are small receptive field. Convolutional layers are of the size 3X3 and maximum pooling layers with size upto 2X2 size and fully connected layers at the end. So in total it has 16 layers.The three fully connected layers, two have 4096 channels and third layer is the number of the class and softmax layer. All hidden layers are equipped with the rectification (ReLU) non-linearity.It is also noted that none of the networks (except for one) contain Local Response Normalisation (LRN), such normalization does not improve the performance, but leads to increased memory consumption and computation time.

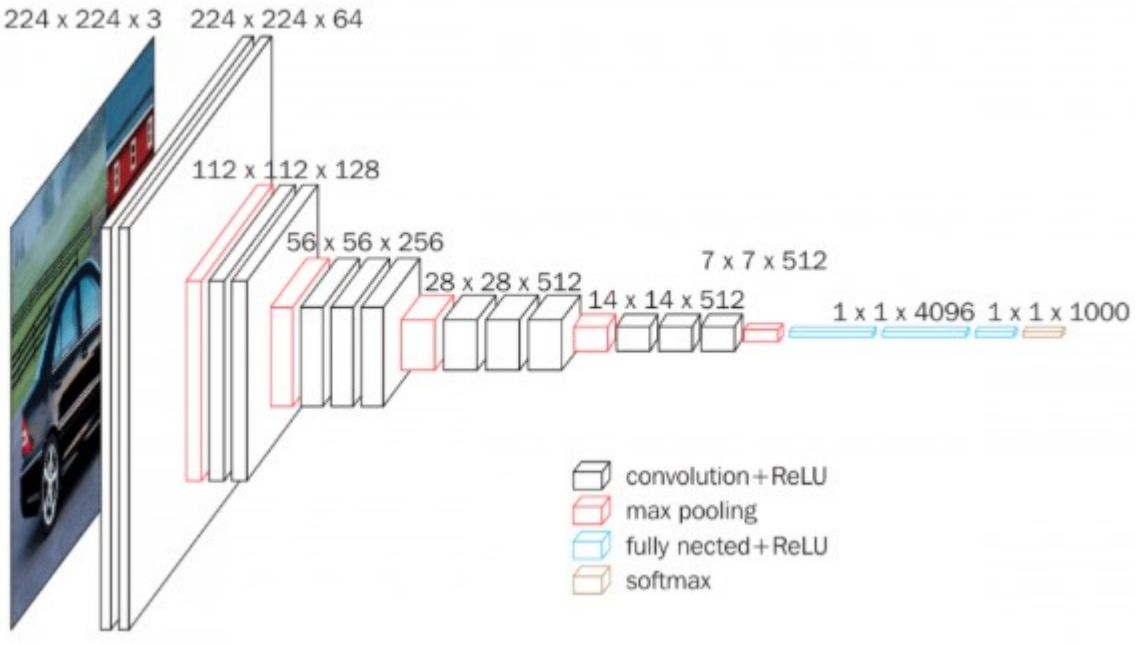

Figure 3: VGG16 Architecture

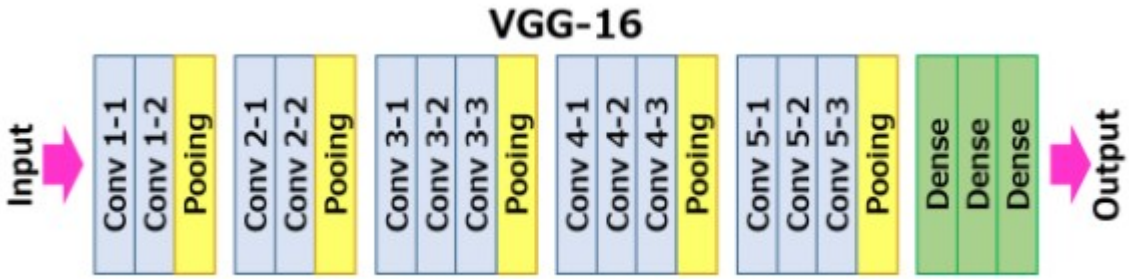

Figure 4: VGG16 Layers

VGG16		
	Dataset 1 (40000)	Dataset 2(1400)
Optimizer	SGD	SGD
Loss function	Cross entropy	Cross entropy
Batch size	20	30
Learning rate	0.001	0.001

Figure 5: VGG16 parameters used

3.6 Parameters

Hyperparameters are the variables that can be changed in the model. Hyperparameter tuning itself is big research area.

- Learning rate: Learning Rate is very vague. It is the hyperparameter which controls the rate at which deep neural network learn. If it is set high, it will overshoot, if it is less then it might not even learn. Definition of slow or fast learning rate is purely experimental.

- Optimizers: Optimizers adjust the weights in every layer according to the difference in loss. Adjusting the weights in each layer basically means learning weights. There many different optimizers in deep neural networks. Two mostly used optimizers are Adam and Stochastic gradient descent.

- Epochs: Epochs define how long one wish to train their model. Longer training in some networks makes the result stable, where as there are network which explode from longer training. Depending upon the loss curve, one can make a smart guess, else it is experimental on how long one should keep their model running

- Loss: Loss functions decides how good is your predicted result. Loss value is then back propagated to every layer according to which optimizers adjust the weight of the layers. Two most used loss functions are cross entropy loss and L1/L2 loss.

GoogLeNet		
	Dataset 1 (40000)	Dataset 2(1400)
Optimizer	SGD	SGD
Lost Function	Cross Entropy	Cross Entropy
Batch Size	10	4
Learning Rate	0.005	0.001

Figure 6: GoogLeNet parameters used

4 Results

Due to its representations in PyTorch, the GoogLeNet models were trained using a longer running model of transfer learning, as just freezing the classification layers didn't accomplish the desired outcome. Instead, a pretrained model was used as a starting point, and then the model was retrained during each epoch using back propagation. Though it took much more time for each epoch, being able to retrain each layer improved the outcome of the training.

GoogLeNet had the best outcomes when working with the larger dataset of 40,000 pavement images, with a total accuracy of 99.68 %. The accuracy of the smaller dataset of wall cracks was lower due to the small training set, 85.31 %.

With regards to the first dataset of 1400 images of wall cracks, GoogLeNet did not

The pretrained VGG16 network model is used to train on the dataset.The spilt made was 80% for training dataset and 20% for test dataset. For the 40,000 image dataset, we ran it through several epochs one for 2 epochs it gave an accuracy of around 96% . Other epochs were for 8 epochs, 10 epochs,12 epoch the accuracy was 94%, 92% and 91% respectively. Observation was as we increase the number of epochs the accuracy decreased,since the model was overfitting For the dataset the results were pretty different, 1400 image dataset, for Epochs 8, 10, 15, 20 the accuracy is 69%, 75%, 81% ,83% respectively. Here the results says that the number of epoch will increase the accuracy but for the 30th epoch the accuracy again decreased to

65% that means the model started overfitting.

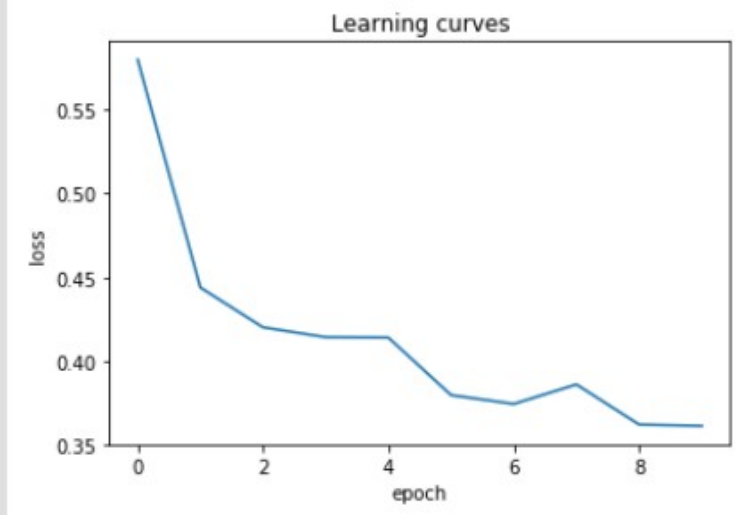

Figure 7: Learning curve for VGG16 with Dataset-2

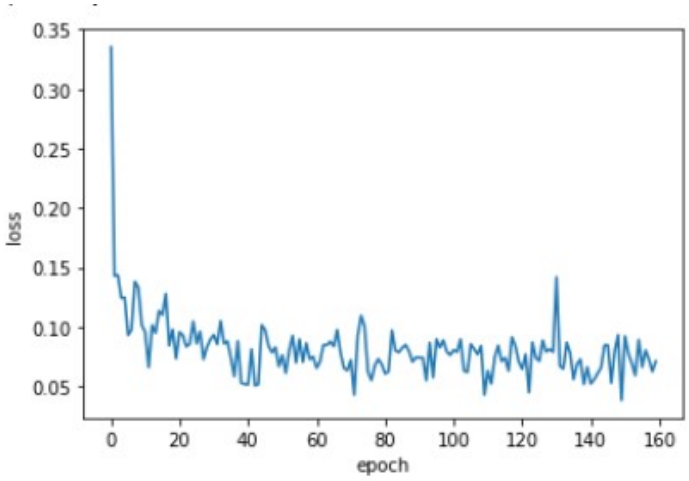

Figure 8: Learning curve for VGG16 with Dataset-1

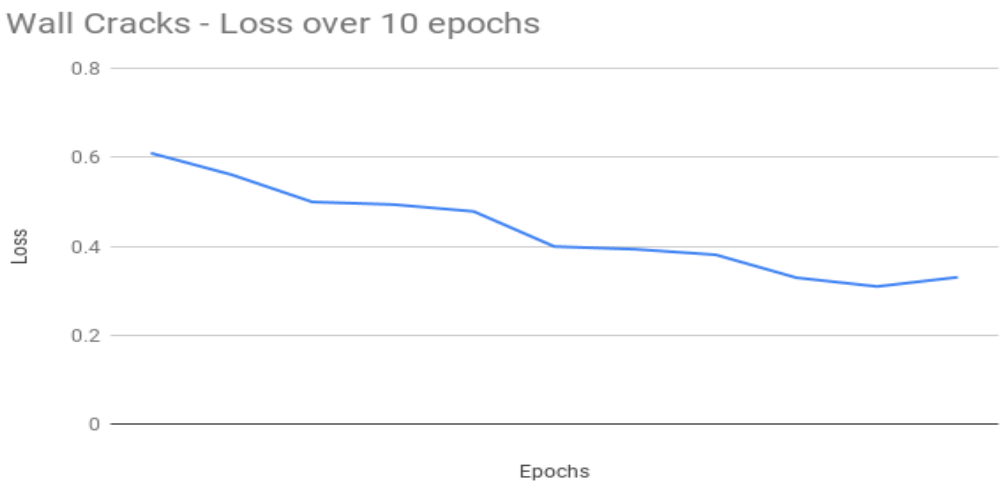

Figure 9: GoogLeNet Losses on Wall Dataset

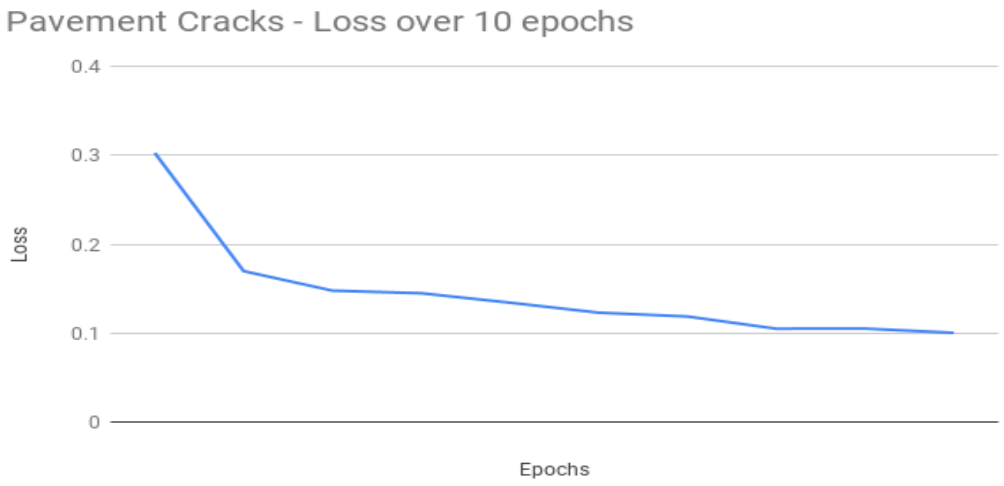

Figure 10: GoogLeNet Losses on Pavement Dataset

5 Discussions and Conclusion

The VGG16 model is painfully slow to train and the network architecture weights themselves are quite large like space and bandwidth. For GoogLeNet,

It has deeper parallel paths, no fully connected layers. Datasets size were uneven, so the 1400 image dataset gave less accuracy. Based on our result the GooLeNet performed well than VGG16.Both models behaved fairly similarly in both datasets. Accuracy for GooLeNet was better than Vgg16.

Our models performed very well on data with more information. We think that a larger scale retraining of the VGG16 network could probably improve its effectiveness instead of just freezing all layers except for the classification. Our largest dataset was also our most homogeneous one, with a nearly uniform background in all images. This allowed us to make a very strong model for ground cracks. However our ability to train on wall cracks was limited by the size and diversity of the dataset. We had good results, but an 85 percent accuracy is still quite low compared to the other dataset. The training process for all of the models was very long, making it difficult to try out many different parameters to try and find the best one. A computer with more processing power working over a longer period of time would go a long way into optimizing parameters for future work. Larger datasets of more uniform images would also help the training process for different kinds of building damage.

Building damage is a diverse set of qualities, many of which have little in common with one another. Flood damage looks very different from earthquake damage, which looks different than natural wear and tear over long spans of time. A generic classifier for damage to a building that was also able to assess the extent of the damage would be an invaluable tool in the efforts of both long term management and emergency responders and their efforts to mitigate catastrophe. As self-driving cars become more common, autonomous fleets could patrol urban areas and find zones of failing infrastructure and automatically notify monitors to the presence of damage. Maps of large areas could be created automatically and targetted repair efforts could be made to more seriously affected areas. While our project focused mainly on crack detection, larger efforts could be made to detect other forms of infrastructure damage, possibly using a single comprehensive model or a series of more specifically trained models designed to assess specific forms of damage.

References

[1] Zou, Q., Zhang, Z., Li, Q., Qi, X., Wang, Q., & Wang, S. (2019). Deep Crack: Learning Hierarchical Convolutional Features for Crack Detection. IEEE Transactions on Image Processing, 28(3), 1498-1512. doi:10.1109/tip.2018.2878966

[2] Yokoyama, S., & Matsumoto, T. (2017). Development of an Automatic Detector of Cracks in Concrete Using Machine Learning. Procedia Engineering, 171, 1250-1255. doi:10.1016/j.proeng.2017.01.418

[3] Zhang, L., Yang, F., Zhang, Y. D., & Zhu, Y. J. (2016). Road crack detection using deep convolutional neural network. 2016 IEEE International Conference on Image Processing (ICIP). doi:10.1109/icip.2016.7533052

[4] Cha, Y., Choi, W., & Büyüköztürk, O. (2017). Deep Learning-Based Crack Damage Detection Using Convolutional Neural Networks. Computer-Aided Civil and Infrastructure Engineering, 32(5), 361-378. doi:10.1111/mice.12263

[5] Concrete Crack Images for Classification. (2018, January 15). Retrieved from https://data.mendeley.com/datasets/5y9wdsg2zt/1

[6] (n.d.). Retrieved from https://dataturks.com/projects/miaozh17/Crack Classification?fbclid=IwAR1Quhg5Omt_DO0vtX3DL7HdfQYIbaA9qIwMjmJzeX--xV4HGiFbusbwPDI

[7] Transfer Learning Tutorial¶. (n.d.). Retrieved from https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

6 Teamwork

Teamwork Division

Teamwork Division		
	Ryan	Janardan
GoogLeNet	100	0
VGG16	0	100
Dataset 1	50	50
Dataset 2	50	50
Results	50	50
Abstract	80	20
Introduction	0	100
Methodology	60	40
Results	50	50
Discussion	50	50
Appendix	50	50

Figure 11: Teamwork Division

A Appendix

Include your code here!

googlenetmodel_wall.py (GoogLeNet model for the second dataset)

```
from __future__ import print_function, division

import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
```

```
import time
import os
import copy

plt.ion()   # interactive mode

if __name__ == '__main__':
    data_transforms = {
        'train': transforms.Compose([
            transforms.RandomResizedCrop(299),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
        ]),
        'validation': transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
        ]),
    }

    data_dir = 'WallCrack'
    image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x)
                                              data_transforms[x])
                      for x in ['train', 'validation']}
    dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], ba
                                                  shuffle=True, num_work
                   for x in ['train', 'validation']}
    dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'valid
    class_names = image_datasets['train'].classes

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cp

    def imshow(inp, title=None):
        """Imshow for Tensor."""
        inp = inp.numpy().transpose((1, 2, 0))
        mean = np.array([0.485, 0.456, 0.406])
```

```
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # pause a bit so that plots are updated


# Get a batch of training data
inputs, classes = next(iter(dataloaders['train']))

# Make a grid from batch
out = torchvision.utils.make_grid(inputs)

imshow(out, title=[class_names[x] for x in classes])

def train_model(model, criterion, optimizer, scheduler, num_epochs=
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

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

        # Each epoch has a training and validation phase
        for phase in ['train', 'validation']:
            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 inputs, labels in dataloaders[phase]:
                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)

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

            print('{} Loss: {:.4f} Acc: {:.4f}'.format(
                phase, epoch_loss, epoch_acc))

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

        print()

    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

def visualize_model(model, num_images=6):
    was_training = model.training
    model.eval()
    images_so_far = 0
    fig = plt.figure()

    with torch.no_grad():
        for i, (inputs, labels) in enumerate(dataloaders['validation
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            for j in range(inputs.size()[0]):
                images_so_far += 1
                ax = plt.subplot(num_images//2, 2, images_so_far)
                ax.axis('off')
                ax.set_title('predicted: {}'.format(class_names[pre
                imshow(inputs.cpu().data[j])

                if images_so_far == num_images:
                    model.train(mode=was_training)
                    return
        model.train(mode=was_training)

model_ft = models.inception_v3(pretrained=True)
model_ft.aux_logits=False
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, 2)
```

```
model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.005, momentum=

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, ga

model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_sch
                       num_epochs=80
                       )


torch.save(model_ft.state_dict(), "googlenetmodelwall_80")
```

googlenetmodel_floor.py (GoogLeNet model for the first dataset)

```
 data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(299),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
    ]),
    'validation': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
    ]),
}

data_dir = 'FloorCrack'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x)
                                          data_transforms[x])
```

```
                  for x in ['train', 'validation']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], ba
                                              shuffle=True, num_work
               for x in ['train', 'validation']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'valid
class_names = image_datasets['train'].classes

device = torch.device("cuda:0" if torch.cuda.is_available() else "cp

model_ft = models.inception_v3(pretrained=True)
model_ft.aux_logits=False
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, 2)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, ga

model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_sch
                       num_epochs=10
                       )


torch.save(model_ft.state_dict(), "googlenetmodel_80")
```

Code for VGG16 Dataset1

```
# -*- coding: utf-8 -*-
"""vgg16(1).ipynb

Automatically generated by Colaboratory.
```

```
Original file is located at
    https://colab.research.google.com/drive/1qgENVU5x178RsZz5uSAdtHK2-f-
"""

from google.colab import drive
drive.mount('/content/drive')

"""! unzip -q '/content/drive/My Drive/CV/final_project/Concrete Crack I

import os
import numpy as np
import torch
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F
# %matplotlib inline

# check if CUDA is available
train_on_gpu = torch.cuda.is_available()

if not train_on_gpu:
    print('CUDA is not available.  Training on CPU ...')
else:
    print('CUDA is available!  Training on GPU ...')

# define training and test data directories
data_dir = 'Concrete Crack Images for Classification/'
train_dir = os.path.join(data_dir, 'train/')
test_dir = os.path.join(data_dir, 'test/')

# classes are folders in each directory with these names
classes = ['Negative', 'Positive']

# load and transform data using ImageFolder

# VGG-16 Takes 224x224 images as input, so we resize all of them
```

```
data_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.ToTensor()])

train_data = datasets.ImageFolder(train_dir, transform=data_transform)
test_data = datasets.ImageFolder(test_dir, transform=data_transform)

# print out some data stats
print('Num training images: ', len(train_data))
print('Num test images: ', len(test_data))

# define dataloader parameters
batch_size = 20
num_workers=4

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch
                                           num_workers=num_workers, shu
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_si
                                          num_workers=num_workers, shuf

# Visualize some sample data

# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy() # convert images to numpy for display

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    plt.imshow(np.transpose(images[idx], (1, 2, 0)))
    ax.set_title(classes[labels[idx]])

# Load the pretrained model from pytorch
vgg16 = models.vgg16(pretrained=True)

# print out the model structure
```

```
# print(vgg16)
# print(vgg16.classifier[6].in_features)
# print(vgg16.classifier[6].out_features)

# Freeze training for all "features" layers
for param in vgg16.features.parameters():
    param.requires_grad = False

import torch.nn as nn

n_inputs = vgg16.classifier[6].in_features

# add last linear layer (n_inputs -> 5 flower classes)
# new layers automatically have requires_grad = True
last_layer = nn.Linear(n_inputs, len(classes))

vgg16.classifier[6] = last_layer

# if GPU is available, move the model to GPU
if train_on_gpu:
    vgg16.cuda()

# check to see that your last layer produces the expected number of outp
print(vgg16.classifier[6].out_features)

import torch.optim as optim

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.
optimizer = optim.SGD(vgg16.classifier.parameters(), lr=0.001, momentum=

# number of epochs to train the model
loss_val = []
#n_epochs = 8,n_epochs = 10,n_epochs = 12
n_epochs = 2
```

```
for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0

    ######################
    # train the model #
    ######################
    # model by default is set to train
    for batch_i, (data, target) in enumerate(train_loader):
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to t
        output = vgg16(data)
        # calculate the batch loss
        loss = criterion(output, target)
        #print(loss)
        # backward pass: compute gradient of the loss with respect to m
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        train_loss += loss.item()

        if batch_i % 20 == 19:    # print training loss every specified
            print('Epoch %d, Batch %d loss: %.16f' %
                  (epoch, batch_i + 1, train_loss / 20))
            loss_val.append(train_loss / 30)
            train_loss = 0.0

plt.plot(loss_val)
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
print('Finished Training')
```

```
# track test loss
# over 5 flower classes
test_loss = 0.0
class_correct = list(0. for i in range(2))
class_total = list(0. for i in range(2))

vgg16.eval() # eval mode

# iterate over test data
for data, target in test_loader:
    # move tensors to GPU if CUDA is available
    if train_on_gpu:
        data, target = data.cuda(), target.cuda()
    # forward pass: compute predicted outputs by passing inputs to the m
    output = vgg16(data)
    # calculate the batch loss
    loss = criterion(output, target)
    # update  test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
    # compare predictions to true label
    correct_tensor = pred.eq(target.data.view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu els
    # calculate test accuracy for each object class
    for i in range(batch_size):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# calculate avg test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))

for i in range(2):
    if class_total[i] > 0:
        print(' Accuracy of %5s: %2d%% (%2d/%2d)' % (
```

```
            classes[i], 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print(' Accuracy of %5s: N/A (no training examples)' % (classes

print('\n Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

Code for vgg16 dataset 2

```
# -*- coding: utf-8 -*-
"""vgg16(2).ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1AUek36hiMLn875Td3RUvjTVeNbq
"""

from google.colab import drive
drive.mount('/content/drive')

"""! unzip -q '/content/drive/My Drive/CV/final_project/crackdataset.zip

import os
import numpy as np
import torch

import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt

# %matplotlib inline

# check if CUDA is available
```

```
train_on_gpu = torch.cuda.is_available()

if not train_on_gpu:
    print('CUDA is not available.  Training on CPU ...')
else:
    print('CUDA is available!  Training on GPU ...')

# define training and test data directories
data_dir = 'crackdataset/'
train_dir = os.path.join(data_dir, 'train/')
test_dir = os.path.join(data_dir, 'validation/')

# classes are folders in each directory with these names
classes = ['Crack', 'no Crack']

# load and transform data using ImageFolder

# VGG-16 Takes 224x224 images as input, so we resize all of them
data_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.ToTensor()])

train_data = datasets.ImageFolder(train_dir, transform=data_transform)
test_data = datasets.ImageFolder(test_dir, transform=data_transform)

# print out some data stats
print('Num training images: ', len(train_data))
print('Num test images: ', len(test_data))

# define dataloader parameters
batch_size = 30
num_workers=4

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch
                                           num_workers=num_workers, shu
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_si
                                          num_workers=num_workers, shuf
```

```
# Visualize some sample data

# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy() # convert images to numpy for display

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(30):
    ax = fig.add_subplot(2, 30/2, idx+1, xticks=[], yticks=[])
    plt.imshow(np.transpose(images[idx], (1, 2, 0)))
    ax.set_title(classes[labels[idx]])

# Load the pretrained model from pytorch
vgg16 = models.vgg16(pretrained=True)

# print out the model structure
print(vgg16)
print(vgg16.classifier[6].in_features)
print(vgg16.classifier[6].out_features)

# Freeze training for all "features" layers
for param in vgg16.features.parameters():
    param.requires_grad = False

import torch.nn as nn

n_inputs = vgg16.classifier[6].in_features

# add last linear layer (n_inputs -> 5 flower classes)
# new layers automatically have requires_grad = True
last_layer = nn.Linear(n_inputs, len(classes))

vgg16.classifier[6] = last_layer

# if GPU is available, move the model to GPU
if train_on_gpu:
```

```
    vgg16.cuda()

# check to see that your last layer produces the expected number of outp
print(vgg16.classifier[6].out_features)

import torch.optim as optim

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.
optimizer = optim.SGD(vgg16.classifier.parameters(), lr=0.001, momentum=

# number of epochs to train the model
loss_val = []
#n_epochs = 8,n_epochs = 15,n_epochs = 20
n_epochs = 10
for epoch in range(1, n_epochs+1):

    # keep track of training and validation loss
    train_loss = 0.0

    ######################
    # train the model #
    ######################
    # model by default is set to train
    for batch_i, (data, target) in enumerate(train_loader):
        # move tensors to GPU if CUDA is available
        if train_on_gpu:
            data, target = data.cuda(), target.cuda()
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to t
        output = vgg16(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to m
        loss.backward()
```

```
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        train_loss += loss.item()

        if batch_i % 30 == 29:    # print training loss every specified
            print('Epoch %d, Batch %d loss: %.16f' %
                  (epoch, batch_i + 1, train_loss / 30))
            loss_val.append(train_loss / 30)
            train_loss = 0.0
plt.plot(loss_val)
plt.title('Learning curves')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
print('Finished Training')

# track test loss
# over 5 flower classes
test_loss = 0.0
class_correct = list(0. for i in range(5))
class_total = list(0. for i in range(5))

vgg16.eval() # eval mode

# iterate over test data
for data, target in test_loader:
    # move tensors to GPU if CUDA is available
    if train_on_gpu:
        data, target = data.cuda(), target.cuda()
    # forward pass: compute predicted outputs by passing inputs to the r
    output = vgg16(data)
    # calculate the batch loss
    loss = criterion(output, target)
    # update  test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
```

```
    # compare predictions to true label
    correct_tensor = pred.eq(target.data.view_as(pred))
    correct = np.squeeze(correct_tensor.numpy()) if not train_on_gpu els
    # calculate test accuracy for each object class
    if(len(target.data) < batch_size):

      batch_size = len(target.data)
    for i in range(batch_size):
        label = target.data[i]

        class_correct[label] += correct[i].item()
        class_total[label] += 1

# calculate avg test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))

for i in range(2):
    if class_total[i] > 0:
        print(' Accuracy of %5s: %2d%% (%2d/%2d)' % (
            classes[i], 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print(' Accuracy of %5s: N/A (no training examples)' % (classes

print('\n Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```