Importing required modules and libraries

```
!pip install torchmetrics torchvision tqdm

import torch
import torch.nn as nn
from torch.nn.functional import softmax
from torch.nn.functional import cross_entropy
from torchmetrics import F1Score
from torchvision import transforms
from torch.utils.data import DataLoader, Dataset
from torch.optim import SGD, Adam
from torch.utils.tensorboard import SummaryWriter
import tqdm
import sklearn.metrics
import numpy as np
import os
from torchmetrics.classification import BinaryF1Score
from torchmetrics.classification import MulticlassF1Score

pwd
```

Uploading datasets from Google Drive

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

!mkdir datasets
!cp /content/drive/MyDrive/surface_crack.zip .
!unzip surface_crack.zip -d datasets
```

Loading tensorboard on Google colab

```
%load_ext tensorboard
```

Multiclass F1 Score Metric (2 Classes)

```
from torchmetrics.classification import MulticlassF1Score
import torch


def metrics(preds, target):
    metr = MulticlassF1Score(num_classes=2)
```

```
    return metr(preds, target)
```

Custom PyTorch Dataset for Multi-Class Image Classification with One-Hot Labels

```
import torch
import torch.nn
from PIL import Image
from torchvision import transforms
import os
import numpy as np
from torch.utils.data import Dataset



class custom_dataset(Dataset):
    def __init__(self, mode = "train", root = "datasets", transforms =
None):
        super().__init__()
        self.mode = mode
        self.root = root
        self.transforms = transforms

        #select split
        self.folder = os.path.join(self.root, self.mode)

        #initialize lists
        self.image_list = []
        self.label_list = []

        #save class lists
        self.class_list = os.listdir(self.folder)
        self.class_list.sort()

        for class_id in range(len(self.class_list)):
            for image in os.listdir(os.path.join(self.folder,
self.class_list[class_id])):
                self.image_list.append(os.path.join(self.folder,
self.class_list[class_id], image))
                label = np.zeros(len(self.class_list))
                label[class_id] = 1.0
                self.label_list.append(label)

    def __getitem__(self, index):
        image_name = self.image_list[index]
        label = self.label_list[index]
```

```
        image = Image.open(image_name).convert("RGB")
        if(self.transforms):
            image = self.transforms(image)

        label = torch.tensor(label)

        return image, label

    def __len__(self):
        return len(self.image_list)
```

Split the data into training, validation and test sets

```
import os
import shutil
from sklearn.model_selection import train_test_split

def split_and_move(source_dir, dest_base, test_size=0.2,
val_size=0.1):
    for class_name in ["Positive", "Negative"]:
        image_paths = [
            os.path.join(source_dir, class_name, fname)
            for fname in os.listdir(os.path.join(source_dir,
class_name))
            if fname.lower().endswith(('.png', '.jpg', '.jpeg'))
        ]

        train_paths, test_paths = train_test_split(image_paths,
test_size=test_size, random_state=42)
        train_paths, val_paths = train_test_split(train_paths,
test_size=val_size / (1 - test_size), random_state=42)

        splits = [("train", train_paths), ("val", val_paths), ("test",
test_paths)]

        for split_name, paths in splits:
            split_folder = os.path.join(dest_base, split_name,
class_name)
            os.makedirs(split_folder, exist_ok=True)
            for src_path in paths:
                dst_path = os.path.join(split_folder,
os.path.basename(src_path))
                shutil.copy2(src_path, dst_path)
```

```
split_and_move("datasets", "datasets")
```

Define 2 CNN architectures ResNet18 and VGG16 using PyTorch

Resnet18

```
import torch
import torch.nn as nn
import torchvision.models as model


class ResNet18(nn.Module):

    def __init__(self, num_classes, pretrained=False):
        super().__init__()

        self.resnet18 = model.resnet18(pretrained=pretrained)
        self.resnet18 =
torch.nn.Sequential(*(list(self.resnet18.children())[:-1]))
        self.classifier = nn.Linear(512, num_classes) # added for the
last code block to work. :)

    def forward(self, image):
        resnet_pred = self.resnet18(image).squeeze()
        out = self.classifier(resnet_pred)

        return out


# if __name__=='__main__':
#     model = model.resnet18(pretrained=False)
#     print(model)
```

VGG

```
import torch
import torch.nn as nn
import torchvision.models as model


class VGG16(nn.Module):

    def __init__(self, num_classes, pretrained=False):
        super().__init__()
```

```
        self.vgg16 = model.vgg16(pretrained=pretrained)
        self.vgg16.classifier[-1] = torch.nn.Linear(in_features=4096,
out_features=num_classes, bias=True)

    def forward(self, image):
        out = self.vgg16(image)

        return out


# if __name__=='__main__':
#     model = VGG16(num_classes=2, pretrained=False)
#     print(model)
```

Experiment models with and without transfer learning techniques.Train the model on the training set using the cross-entropy loss function and optimizers named SGD and Adam

```
import torch
import torch.nn as nn
from torch.nn.functional import softmax
from torch.utils.data import DataLoader, Dataset
from torch.optim import SGD, Adam
from torch.utils.tensorboard import SummaryWriter
from torchmetrics import Accuracy, F1Score
import tqdm
import os

save_model_path = "checkpoints/"
os.makedirs(save_model_path, exist_ok=True)
def val(model, data_val, loss_function, writer, epoch, device):
    # Metrics
    f1_macro = F1Score(num_classes=2, average='macro',
task='multiclass').to(device)
    f1_per_class = F1Score(num_classes=2, average=None,
task='multiclass').to(device)
    acc_macro = Accuracy(num_classes=2, average='macro',
task='multiclass').to(device)
    acc_per_class = Accuracy(num_classes=2, average=None,
task='multiclass').to(device)

    model.eval()
    tq = tqdm.tqdm(total=len(data_val))
    tq.set_description('Validation:')
```

```
    total_loss = 0

    with torch.no_grad():
        for _, (image, label) in enumerate(data_val):
            image = image.to(device)
            label = label.to(device)

            pred = model(image)
            loss = loss_function(pred, label)
            total_loss += loss.item()

            pred_probs = pred.softmax(dim=1)

            # Update metrics
            f1_macro.update(pred_probs, label)
            f1_per_class.update(pred_probs, label)
            acc_macro.update(pred_probs, label)
            acc_per_class.update(pred_probs, label)

            tq.update(1)

    tq.close()

    # Compute metrics
    f1_macro_val = f1_macro.compute()
    acc_macro_val = acc_macro.compute()
    f1_per_class_val = f1_per_class.compute()
    acc_per_class_val = acc_per_class.compute()
    avg_loss = total_loss / len(data_val)

    # Logging
    writer.add_scalar("Validation/Loss", avg_loss, epoch)
    writer.add_scalar("Validation/F1_Macro", f1_macro_val, epoch)
    writer.add_scalar("Validation/Accuracy_Macro", acc_macro_val,
epoch)

    for i, (f1, acc) in enumerate(zip(f1_per_class_val,
acc_per_class_val)):
        writer.add_scalar(f"Validation/F1_Class_{i}", f1, epoch)
        writer.add_scalar(f"Validation/Accuracy_Class_{i}", acc,
epoch)

    print(f"\n[Validation] Loss: {avg_loss:.4f}, F1 (macro):
{f1_macro_val:.4f}, Accuracy (macro): {acc_macro_val:.4f}")

    return {
        "loss": avg_loss,
        "f1_macro": f1_macro_val.item(),
        "acc_macro": acc_macro_val.item(),
```

```
        "f1_class": f1_per_class_val.tolist(),
        "acc_class": acc_per_class_val.tolist()
    }

def train(model, train_loader, val_loader, optimizer, loss_fn,
n_epochs, device, log_dir):
    writer = SummaryWriter(log_dir=os.path.join('runs', log_dir))
    model.to(device)

    best_val_f1 = 0
    best_epoch = -1
    best_val_metrics = {}
    final_train_metrics = {}

    for epoch in range(n_epochs):
        model.train()
        running_loss = 0.0
        tq = tqdm.tqdm(total=len(train_loader))
        tq.set_description(f"Epoch {epoch+1}/{n_epochs}")

        # Metrics
        f1_macro = F1Score(num_classes=2, average='macro',
task='multiclass').to(device)
        f1_per_class = F1Score(num_classes=2, average=None,
task='multiclass').to(device)
        acc_macro = Accuracy(num_classes=2, average='macro',
task='multiclass').to(device)
        acc_per_class = Accuracy(num_classes=2, average=None,
task='multiclass').to(device)

        for i, (images, labels) in enumerate(train_loader):
            images = images.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

            outputs = model(images)
            loss = loss_fn(outputs, labels)
            outputs = outputs.softmax(dim=1)

            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Update metrics
            f1_macro.update(outputs, labels)
            f1_per_class.update(outputs, labels)
            acc_macro.update(outputs, labels)
            acc_per_class.update(outputs, labels)
```

```
            tq.set_postfix(loss='%.6f' % loss.item())
            tq.update(1)

        tq.close()

        # Compute metrics
        epoch_loss = running_loss / len(train_loader)
        f1_macro_val = f1_macro.compute()
        acc_macro_val = acc_macro.compute()
        f1_per_class_val = f1_per_class.compute()
        acc_per_class_val = acc_per_class.compute()

        # Logging
        writer.add_scalar("Training/Loss", epoch_loss, epoch)
        writer.add_scalar("Training/F1_Macro", f1_macro_val, epoch)
        writer.add_scalar("Training/Accuracy_Macro", acc_macro_val,
epoch)

        for i, (f1, acc) in enumerate(zip(f1_per_class_val,
acc_per_class_val)):
            writer.add_scalar(f"Training/F1_Class_{i}", f1, epoch)
            writer.add_scalar(f"Training/Accuracy_Class_{i}", acc,
epoch)

        print(f"[Training] Epoch [{epoch+1}/{n_epochs}] Loss:
{epoch_loss:.4f}, F1 (macro): {f1_macro_val:.4f}, Accuracy (macro):
{acc_macro_val:.4f}")

        # Store last epoch training metrics
        final_train_metrics = {
            "loss": epoch_loss,
            "f1_macro": f1_macro_val.item(),
            "acc_macro": acc_macro_val.item(),
            "f1_class": f1_per_class_val.tolist(),
            "acc_class": acc_per_class_val.tolist()
        }

        # Validation step
        val_metrics = val(model, val_loader, loss_fn, writer, epoch,
device)

        # Save best
        if val_metrics["f1_macro"] > best_val_f1:
            best_val_f1 = val_metrics["f1_macro"]
            best_val_metrics = val_metrics
            best_epoch = epoch + 1

        # Save checkpoint
        checkpoint = {
```

```
            'epoch': epoch + 1,
            'state_dict': model.state_dict(),
            'optimizer': optimizer.state_dict()
        }

        torch.save(checkpoint, os.path.join(save_model_path,
log_dir+".pth"))
        print(f"Saved model to {save_model_path}{log_dir}.pth")

    # Summary

    metrics_output = (
        "\n==== Training Complete ====\n"
        f"Final Training Loss: {final_train_metrics['loss']:.4f}, F1
(macro): {final_train_metrics['f1_macro']:.4f}, Accuracy (macro):
{final_train_metrics['acc_macro']:.4f}\n"
        f"Class-wise F1: {final_train_metrics['f1_class']}\n"
        f"Class-wise Accuracy: {final_train_metrics['acc_class']}\n\n"
        f"Best Validation F1: {best_val_f1:.4f} at epoch {best_epoch}\
n"
        f"Validation Loss: {best_val_metrics['loss']:.4f}, Accuracy:
{best_val_metrics['acc_macro']:.4f}\n"
        f"Class-wise F1: {best_val_metrics['f1_class']}\n"
        f"Class-wise Accuracy: {best_val_metrics['acc_class']}\n"
    )

    print(metrics_output)

    output_file_path = os.path.join(save_model_path, log_dir +
"_metrics.txt")
    with open(output_file_path, "w") as f:
        f.write(metrics_output)

    print(f"\nSaved final metrics to {output_file_path}")

!mkdir -p /content/checkpoints



def main(model_name, optimizer_name, lr, pretrained, log_dir):

    device = "cuda"
    #device = "cpu"

    tr_train = transforms.Compose([
    transforms.RandomRotation(45, fill=255),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomAffine(degrees=0, scale=(0.6, 1.2), fill=255),
    transforms.Resize([250, 250]),
    transforms.RandomCrop(224),
    transforms.ToTensor(),
```

```
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224,
0.225)),
])

    tr_val = transforms.Compose([
        transforms.Resize([224, 224]),
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224,
0.225))
    ])

    train_data = custom_dataset("train", transforms=tr_train)
    val_data = custom_dataset("val", transforms=tr_val)

    train_loader = DataLoader(
        train_data,
        batch_size=32,
        shuffle=True
    )

    val_loader = DataLoader(
        val_data,
        batch_size=32,
        drop_last=True
    )

    max_epoch = 10

    if model_name=='resnet':
        model = ResNet18(num_classes=2,
pretrained=pretrained).to(device)
    elif model_name=='vgg':
        model = VGG16(num_classes=2, pretrained=pretrained).to(device)

    if optimizer_name=='sgd':
        optimizer = SGD(model.parameters(), lr=lr, momentum = 0.9)
    elif optimizer_name=='adam':
        optimizer = Adam(model.parameters(), lr=lr)

    loss = nn.CrossEntropyLoss()

    train(model, train_loader, val_loader, optimizer, loss, max_epoch,
device, log_dir)



main(model_name='resnet', optimizer_name='sgd', lr=5e-3,
pretrained=False, log_dir='resnet_sgd')

main(model_name='resnet', optimizer_name='adam', lr=1e-4,
pretrained=False, log_dir='resnet_adam')
```

```
main(model_name='resnet', optimizer_name='sgd', lr=5e-4,
pretrained=True, log_dir='resnet_sgd_pretrained')

main(model_name='resnet', optimizer_name='adam', lr=5e-5,
pretrained=True, log_dir='resnet_adam_pretrained')

main(model_name='vgg', optimizer_name='sgd', lr=5e-3,
pretrained=False, log_dir='vgg_sgd')

main(model_name='vgg', optimizer_name='adam', lr=1e-4,
pretrained=False, log_dir='vgg_adam')

main(model_name='vgg', optimizer_name='sgd', lr=5e-4, pretrained=True,
log_dir='vgg_sgd_pretrained')

main(model_name='vgg', optimizer_name='adam', lr=5e-5,
pretrained=True, log_dir='vgg_adam_pretrained')
```

Evaluate the performance of the model on the test set by calculating the accuracy, with additional script.

```
import torch
from torchmetrics import F1Score, Accuracy, ConfusionMatrix
from torchvision import transforms
from torch.utils.data import DataLoader
# from datasets.dataset_retrieval import custom_dataset
from torch.utils.tensorboard import SummaryWriter
import tqdm
import os
# from models.resnet import ResNet18
# from models.vgg import VGG16
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np


def plot_confusion_matrix(cm, log_dir):
    classes = os.listdir('datasets/train')
    classes.sort()

    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
xticklabels=classes, yticklabels=classes)
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Confusion Matrix')
```

```
    plt.tight_layout()
    save_path = os.path.join("plots",
f"{log_dir}_confusion_matrix.png")
    os.makedirs("plots", exist_ok=True)
    plt.savefig(save_path)
    print(f"Confusion matrix saved at: {save_path}")
    plt.show()


def test(model, test_loader, device, log_dir):

    f1 = F1Score(num_classes=2, task='multiclass')
    acc = Accuracy(num_classes=2, task='multiclass')
    cm_metric = ConfusionMatrix(num_classes=2, task='multiclass')

    y_test = []
    y_pred = []

    with torch.no_grad():

        model.eval()

        tq = tqdm.tqdm(total=len(test_loader))
        tq.set_description('Testing:')

        data_iterator = enumerate(test_loader)

        for _, batch in data_iterator:

            image, label = batch
            image = image.to(device)
            label = label.to(device)

            pred = model(image)
            pred = pred.softmax(dim=1)

            y_test.extend(torch.argmax(label, dim=1).tolist())
            y_pred.extend(torch.argmax(pred, dim=1).tolist())

            tq.update(1)

        f1_score = f1(torch.tensor(y_pred), torch.tensor(y_test))
        acc_score = acc(torch.tensor(y_pred), torch.tensor(y_test))
        confusion_matrix = cm_metric(torch.tensor(y_pred),
torch.tensor(y_test))

        tq.close()
        print(f"F1: {f1_score} Accuracy: {acc_score}")
        plot_confusion_matrix(confusion_matrix.cpu().numpy(), log_dir)

    return None
```

```
def main(model_name, log_dir):

    device = "cuda"

    tr_test = transforms.Compose([
        transforms.Resize([224, 224]),
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224,
0.225))
    ])

    test_data = custom_dataset("test", transforms=tr_test)

    test_loader = DataLoader(
        test_data,
        batch_size=32,
        drop_last=False
    )

    if model_name == 'resnet':
        model = ResNet18(num_classes=2, pretrained=False).to(device)
    elif model_name == 'vgg':
        model = VGG16(num_classes=2, pretrained=False).to(device)

    checkpoint_path = os.path.join('checkpoints', log_dir+'.pth')
    checkpoint = torch.load(checkpoint_path)
    model.load_state_dict(checkpoint['state_dict'])

    print(f"Loaded model from {checkpoint_path}")

    test(model, test_loader, device, log_dir)


main(model_name='resnet', log_dir='resnet_sgd')

main(model_name='resnet', log_dir='resnet_adam')

main(model_name='resnet', log_dir='resnet_sgd_pretrained')

main(model_name='resnet', log_dir='resnet_adam_pretrained')

main(model_name='vgg', log_dir='vgg_sgd')

main(model_name='vgg', log_dir='vgg_adam')

main(model_name='vgg', log_dir='vgg_sgd_pretrained')

main(model_name='vgg', log_dir='vgg_adam_pretrained')
```

TenserBoard

```
%tensorboard --logdir=runs/resnet_sgd

%tensorboard --logdir=runs/resnet_adam

%tensorboard --logdir=runs/resnet_sgd_pretrained

%tensorboard --logdir=runs/resnet_adam_pretrained

%tensorboard --logdir=runs/vgg_sgd

%tensorboard --logdir=runs/vgg_adam

%tensorboard --logdir=runs/vgg_sgd_pretrained

%tensorboard --logdir=runs/vgg_adam_pretrained

from tensorboard.backend.event_processing.event_accumulator import
EventAccumulator

# Load TensorBoard event file
event_acc = EventAccumulator('runs/resnet_sgd')  # replace with your
actual folder name
event_acc.Reload()
checkpoint = torch.load('checkpoints/resnet_sgdepoch10.pth')
print(checkpoint.keys())


print("Available Scalars:", event_acc.Tags()['scalars'])
```

Manual Testing

```
!mkdir -p datasets/real_images
!cp datasets/Positive/00001.jpg datasets/real_images/test_image_1.jpg
!cp datasets/Negative/00001.jpg datasets/real_images/test_image_2.jpg

import torch
import torchvision.transforms as transforms
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np

def manual_test(model, image_path, device, log_dir):

    tr_test = Compose([
    transforms.RandomRotation(45, fill=255),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomAffine(degrees=0, scale=(0.6, 1.2), fill=255),
    transforms.Resize([250, 250]),
```

```
    transforms.RandomCrop(224),
    transforms.ToTensor(),
    transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224,
0.225)),
])

    image = Image.open(image_path).convert("RGB")
    plt.imshow(image)
    plt.show()
    image = tr_test(image).unsqueeze(0).to(device)
    with torch.no_grad():
        model.eval()
        pred = model(image)
        pred = pred.softmax(dim=0)
        predicted_class = torch.argmax(pred, dim=0).item()
        print(f"Image: {image_path}")
        if predicted_class == 0:
            print("Prediction: Negative (No Crack)")
        else:
            print("Prediction: Positive (Crack)")


# Example usage:
device = "cuda" if torch.cuda.is_available() else "cpu"
model = ResNet18(2, False).to(device)
checkpoint_path = os.path.join('checkpoints', 'resnet_sgd.pth')
checkpoint = torch.load(checkpoint_path)
model.load_state_dict(checkpoint['state_dict'])

manual_test(model, "datasets/real_images/test_image_1.jpg", device,
"resnet_sgd")
manual_test(model, "datasets/real_images/test_image_2.jpg", device,
"resnet_sgd")
```