Concrete Crack Detection Using Transfer Larning

Import Libraries

```
import torch
import torch.nn as nn
import torchvision
from torchvision import datasets, transforms, models
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from datetime import datetime
import random
import sys, os
import shutil
from glob import glob
import imageio
```

Load the Crack Dataset

- Link to the crack data set: https://data.mendeley.com/datasets/5y9wdsg2zt/2

```
# Data from: https://md-datasets-cache-zipfiles-prod.s3.eu-west-1.amazona
!wget https://md-datasets-cache-zipfiles-prod.s3.eu-west-1.amazonaws.com,
```

```
--2021-10-07 20:39:01--  https://md-datasets-cache-zipfiles-prod.s3.eu-west-1.amazonaws
Resolving md-datasets-cache-zipfiles-prod.s3.eu-west-1.amazonaws.com (md-datasets-cache-
Connecting to md-datasets-cache-zipfiles-prod.s3.eu-west-1.amazonaws.com (md-datasets-ca
HTTP request sent, awaiting response... 200 OK
Length: 240847944 (230M) [application/octet-stream]
Saving to: ‘5y9wdsg2zt-2.zip’

5y9wdsg2zt-2.zip    100%[===================>] 229.69M  26.5MB/s    in 9.9s

2021-10-07 20:39:12 (23.3 MB/s) - ‘5y9wdsg2zt-2.zip’ saved [240847944/240847944]
```

```
!ls
```

```
5y9wdsg2zt-2.zip  sample_data
```

```
!unzip -qq -o 5y9wdsg2zt-2.zip
```

```
!ls
```

```
5y9wdsg2zt-2.zip  'Concrete Crack Images for Classification.rar'   sample_data
```

```
!unrar x 'Concrete Crack Images for Classification.rar'
!ls
```

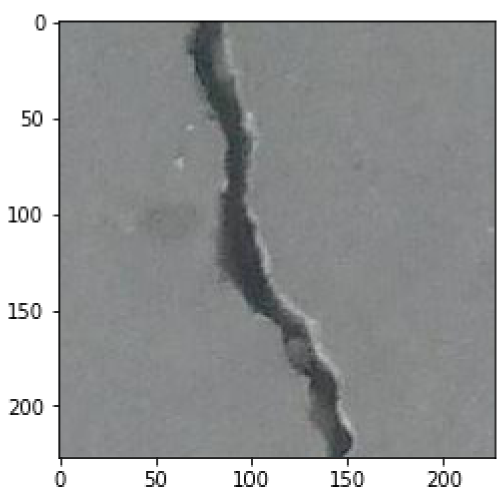
```
Extracting  Negative/19945.jpg                                       OK
Extracting  Negative/19946.jpg                                       OK
Extracting  Negative/19947.jpg                                       OK
Extracting  Negative/19948.jpg                                       OK
Extracting  Negative/19949.jpg                                       OK
Extracting  Negative/19950.jpg                                       OK
Extracting  Negative/19951.jpg                                       OK
Extracting  Negative/19952.jpg                                       OK
Extracting  Negative/19953.jpg                                       OK
Extracting  Negative/19954.jpg                                       OK
Extracting  Negative/19955.jpg                                       OK
Extracting  Negative/19956.jpg                                       OK
Extracting  Negative/19957.jpg                                       OK
Extracting  Negative/19958.jpg                                       OK
Extracting  Negative/19959.jpg                                       OK
Extracting  Negative/19960.jpg                                       OK
Extracting  Negative/19961.jpg                                       OK
Extracting  Negative/19962.jpg                                       OK
Extracting  Negative/19963.jpg                                       OK
Extracting  Negative/19964.jpg                                       OK
Extracting  Negative/19965.jpg                                       OK
Extracting  Negative/19966.jpg                                       OK
Extracting  Negative/19967.jpg                                       OK
Extracting  Negative/19968.jpg                                       OK
Extracting  Negative/19969.jpg                                       OK
Extracting  Negative/19970.jpg                                       OK
Extracting  Negative/19971.jpg                                       OK
Extracting  Negative/19972.jpg                                       OK
Extracting  Negative/19973.jpg                                       OK
Extracting  Negative/19974.jpg                                       OK
Extracting  Negative/19975.jpg                                       OK
Extracting  Negative/19976.jpg                                       OK
Extracting  Negative/19977.jpg                                       OK
Extracting  Negative/19978.jpg                                       OK
Extracting  Negative/19979.jpg                                       OK
Extracting  Negative/19980.jpg                                       OK
Extracting  Negative/19981.jpg                                       OK
Extracting  Negative/19982.jpg                                       OK
Extracting  Negative/19983.jpg                                       OK
Extracting  Negative/19984.jpg                                       OK
Extracting  Negative/19985.jpg                                       OK
Extracting  Negative/19986.jpg                                       OK
Extracting  Negative/19987.jpg                                       OK
Extracting  Negative/19988.jpg                                       OK
```

```
Extracting  Negative/19989.jpg                                        OK
Extracting  Negative/19990.jpg                                        OK
Extracting  Negative/19991.jpg                                        OK
Extracting  Negative/19992.jpg                                        OK
Extracting  Negative/19993.jpg                                        OK
Extracting  Negative/19994.jpg                                        OK
Extracting  Negative/19995.jpg                                        OK
Extracting  Negative/19996.jpg                                        OK
Extracting  Negative/19997.jpg                                        OK
Extracting  Negative/19998.jpg                                        OK
Extracting  Negative/19999.jpg                                        OK
Extracting  Negative/20000.jpg                                        OK
All OK
 5y9wdsg2zt-2.zip                                 Negative   sample_data
'Concrete Crack Images for Classification.rar'   Positive
```

```python
crack_images = os.listdir('Positive/')
print("Number of Crack Images: ", len(crack_images))

no_crack_images = os.listdir('Negative/')
print("Number of No Crack Images: ", len(no_crack_images))
```

```
Number of Crack Images:  20000
Number of No Crack Images:  20000
```

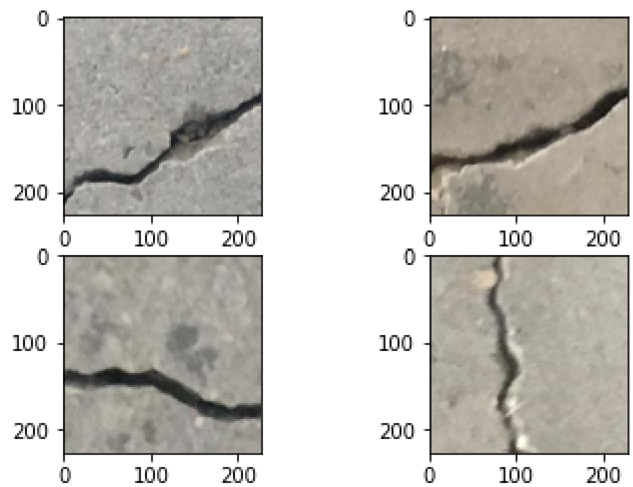
```python
# look at an image for fun
plt.imshow(imageio.imread('Positive/00001.jpg'))
plt.show()
```




```python
## Visualize Random crack images
random_indices = np.random.randint(0, len(crack_images), size=4)
random_images = np.array(crack_images)[random_indices.astype(int)]

f, axarr = plt.subplots(2,2)
```

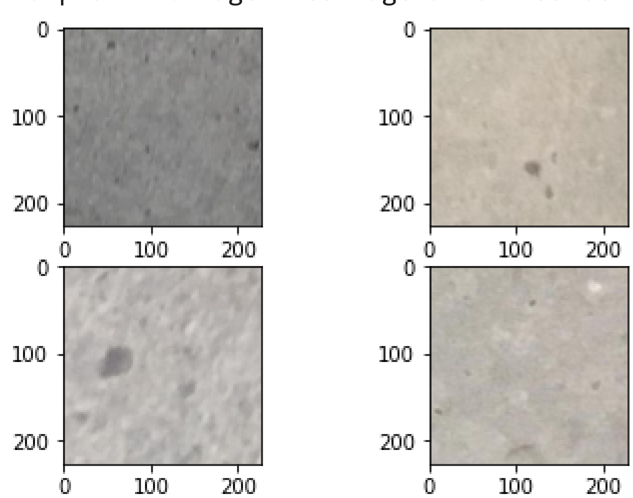
```
axarr[0,0].imshow(mpimg.imread(f'Positive/{random_images[0]}'))
axarr[0,1].imshow(mpimg.imread(f'Positive/{random_images[1]}'))
axarr[1,0].imshow(mpimg.imread(f'Positive/{random_images[2]}'))
axarr[1,1].imshow(mpimg.imread(f'Positive/{random_images[3]}'))
```

```
<matplotlib.image.AxesImage at 0x7f3540fbac90>
```



```
## Visualize Random noncrack images
random_indices = np.random.randint(0, len(no_crack_images), size=4)
random_images = np.array(no_crack_images)[random_indices.astype(int)]

f, axarr = plt.subplots(2,2)
axarr[0,0].imshow(mpimg.imread(f'Negative/{random_images[0]}'))
axarr[0,1].imshow(mpimg.imread(f'Negative/{random_images[1]}'))
axarr[1,0].imshow(mpimg.imread(f'Negative/{random_images[2]}'))
axarr[1,1].imshow(mpimg.imread(f'Negative/{random_images[3]}'))
```

```
<matplotlib.image.AxesImage at 0x7f3540e1de10>
```



Train and Test Datasets

```
# remove data if already exist
!rm -r data
# Make directories to store the data Keras-style
!mkdir data
!mkdir data/train
!mkdir data/test
!mkdir data/train/noncrack
!mkdir data/train/crack
!mkdir data/test/noncrack
!mkdir data/test/crack


# Move the images
# Note: we will consider 'training' to be the train set
#       'validation' folder will be the test set
#       ignore the 'evaluation' set
!mv Negative/* data/train/noncrack
!mv Positive/* data/train/crack
# test set will be generated randomly from train set

crack_train = 'data/train/crack'
crack_test = 'data/test/crack/'
noncrack_train = 'data/train/noncrack/'
noncrack_test = 'data/test/noncrack/'

crack_files = os.listdir(crack_train)
noncrack_files = os.listdir(noncrack_train)

print(f"crack_files:{len(crack_files)}, noncrack_files:{len(noncrack_file

for f in crack_files:
    if random.random() > 0.80:
        shutil.move(f'{crack_train}/{f}', crack_test)

for f in noncrack_files:
    if random.random() > 0.80:
        shutil.move(f'{noncrack_train}/{f}', noncrack_test)

# show the number of train and test images
print(f"Train set: crack={len(os.listdir(crack_train))}, noncrack={len(os
print(f"Test set: crack={len(os.listdir(crack_test))}, noncrack={len(os.l
```

```
crack_files:20000, noncrack_files:20000
Train set: crack=16080, noncrack=16076
Test set: crack=3920, noncrack=3924
```

```
# Note: normalize mmean and std are standardized for ImageNet
# https://github.com/pytorch/examples/blob/97304e232807082c2e7b54c597615d
train_transform = transforms.Compose([
  transforms.RandomResizedCrop(size=256, scale=(0.8, 1.0)),
  transforms.RandomRotation(degrees=15),
  transforms.ColorJitter(),
  transforms.CenterCrop(size=224),
  transforms.RandomHorizontalFlip(),
  transforms.ToTensor(),
  transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_transform = transforms.Compose([
  transforms.Resize(size=256),
  transforms.CenterCrop(size=224),
  transforms.ToTensor(),
  transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])


train_dataset = datasets.ImageFolder(
    'data/train',
    transform=train_transform
)
test_dataset = datasets.ImageFolder(
    'data/test',
    transform=test_transform
)


# Data loader
# Usefull because it automatically generates batches in the training loop
# and takes care of shuffling

batch_size = 128
train_loader = torch.utils.data.DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True
)
```

```
test_loader = torch.utils.data.DataLoader(
    dataset=test_dataset,
    batch_size=batch_size,
    shuffle=False
)


# Define the model
model = models.vgg16(pretrained=True)

# Freeze VGG weights
for param in model.parameters():
  param.requires_grad = False
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.cache/t
100%                                             528M/528M [00:05<00:00, 88.6MB/s]
```

```
print(model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
      (29): ReLU(inplace=True)
      (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace=True)
      (2): Dropout(p=0.5, inplace=False)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace=True)
      (5): Dropout(p=0.5, inplace=False)
      (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
  )
```

```
# We want to replace the 'classifier'
model.classifier
```

```
Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=1000, bias=True)
)
```

```
n_features = model.classifier[0].in_features
n_features
```

```
25088
```

```
# We're doing binary classification
model.classifier = nn.Linear(n_features, 2)
```

```
# Let's see what the model is now
print(model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
      (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (11): ReLU(inplace=True)
      (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (13): ReLU(inplace=True)
      (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (15): ReLU(inplace=True)
      (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (18): ReLU(inplace=True)
      (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (20): ReLU(inplace=True)
      (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (22): ReLU(inplace=True)
      (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (25): ReLU(inplace=True)
      (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (27): ReLU(inplace=True)
      (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (29): ReLU(inplace=True)
      (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Linear(in_features=25088, out_features=2, bias=True)
  )
```

```
# # Define the model
# class CNN(nn.Module):
#   def __init__(self, K):
#     super(CNN, self).__init__()

#     # define the conv layers
#     self.conv1 = nn.Sequential(
#         nn.Conv2d(3, 32, kernel_size=3, padding=1),
#         nn.ReLU(),
#         nn.BatchNorm2d(32),
#         nn.Conv2d(32, 32, kernel_size=3, padding=1),
#         nn.ReLU(),
#         nn.BatchNorm2d(32),
#         nn.MaxPool2d(2),
#     )

#     self.conv2 = nn.Sequential(
#         nn.Conv2d(32, 64, kernel_size=3, padding=1),
#         nn.ReLU(),
#         nn.BatchNorm2d(64),
#         nn.Conv2d(64, 64, kernel_size=3, padding=1),
#         nn.ReLU(),
#         nn.BatchNorm2d(64),
```

```
#         nn.MaxPool2d(2),
#     )

#     self.conv3 = nn.Sequential(
#         nn.Conv2d(64, 128, kernel_size=3, padding=1),
#         nn.ReLU(),
#         nn.BatchNorm2d(128),
#         nn.Conv2d(128, 128, kernel_size=3, padding=1),
#         nn.ReLU(),
#         nn.BatchNorm2d(128),
#         nn.MaxPool2d(2),
#     )

#     # Useful: https://pytorch.org/docs/stable/nn.html#torch.nn.MaxPool2
#     # H_out = H_in + 2p - 2 --> p = 1 if H_out = H_in

#     # Easy to calculate output
#     # 32 > 16 >8 > 4

#     # define the linear layers
#     self.fc1 = nn.Linear(128 * 4 * 4, 1024)
#     self.fc2 = nn.Linear(1024, K)

#   def forward(self, x):
#     x = self.conv1(x)
#     x = self.conv2(x)
#     x = self.conv3(x)
#     x = x.view(x.size(0), -1)
#     x = F.dropout(x, p=0.5)
#     x = F.relu(self.fc1(x))
#     x = F.dropout(x, p=0.2)
#     x = self.fc2(x)
#     return x



# # Instantiate the model
# model = CNN(K)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
model.to(device)
```

```
cuda:0
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Linear(in_features=25088, out_features=2, bias=True)
)
```

```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters())
```

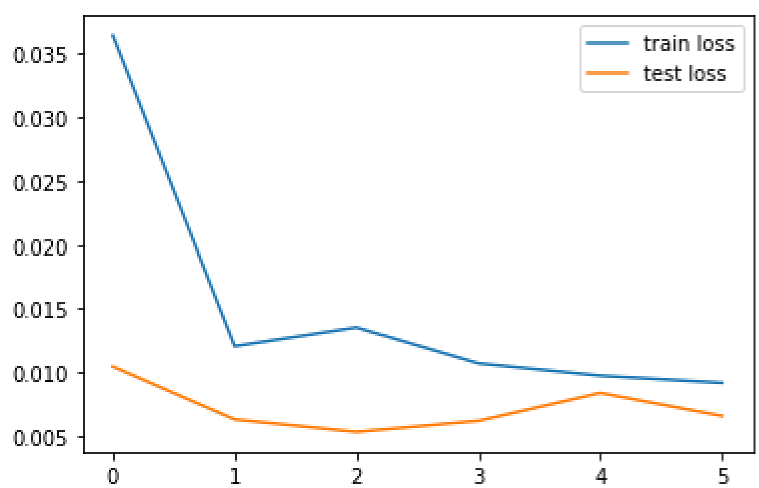
```
# A function to encapsulate the training loop
def batch_gd(model, criterion, optimizer, train_loader, test_loader, epo

  # Stuff to store
  train_losses = np.zeros(epochs)
  test_losses = np.zeros(epochs)

  for it in range(epochs):
```

```
    t0 = datetime.now()
    train_loss = []
    for inputs, targets in train_loader:
      # move data to GPU
      inputs, targets = inputs.to(device), targets.to(device)

      # zero the parameter gradients
      optimizer.zero_grad()

      # Forward pass
      outputs = model(inputs)
      loss = criterion(outputs, targets)

      # Backward and optimize
      loss.backward()
      optimizer.step()

      train_loss.append(loss.item())

    # Get train loss and test loss
    train_loss = np.mean(train_loss) # a little misleading

    test_loss = []
    for inputs, targets in test_loader:
      inputs, targets = inputs.to(device), targets.to(device)
      outputs = model(inputs)
      loss = criterion(outputs, targets)
      test_loss.append(loss.item())
    test_loss = np.mean(test_loss)

    # Save losses
    train_losses[it] = train_loss
    test_losses[it] = test_loss

    dt = datetime.now() - t0
    print(f"Epoch {it+1}/{epochs}, Train Loss: {train_loss:.4f}, Test Los
  return train_losses, test_losses


train_losses, test_losses = batch_gd(
    model,
    criterion,
    optimizer,
```

```
    train_loader,
    test_loader,
    epochs=6
)
```

```
/usr/local/lib/python3.7/dist-packages/torch/nn/functional.py:718: UserWarning: Named te
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
Epoch 1/6, Train Loss: 0.0365, Test Loss: 0.0105, Duration: 0:08:47.185859
Epoch 2/6, Train Loss: 0.0121, Test Loss: 0.0063, Duration: 0:08:41.925517
Epoch 3/6, Train Loss: 0.0135, Test Loss: 0.0053, Duration: 0:08:42.695916
Epoch 4/6, Train Loss: 0.0107, Test Loss: 0.0062, Duration: 0:08:42.016629
Epoch 5/6, Train Loss: 0.0097, Test Loss: 0.0084, Duration: 0:08:42.092295
Epoch 6/6, Train Loss: 0.0092, Test Loss: 0.0066, Duration: 0:08:39.165926
```

```
# Plot the train loss and test loss per iteration
plt.plot(train_losses, label='train loss')
plt.plot(test_losses, label='test loss')
plt.legend()
plt.show()
```




```
# Accuracy

n_correct = 0.
n_total = 0.
for inputs, targets in train_loader:
  # move data to GPU
  inputs, targets = inputs.to(device), targets.to(device)

  # Forward pass
  outputs = model(inputs)

  # Get prediction
```

```
    # torch.max returns both max and argmax
    _, predictions = torch.max(outputs, 1)

    # update counts
    n_correct += (predictions == targets).sum().item()
    n_total += targets.shape[0]

  train_acc = n_correct / n_total

  n_correct = 0.
  n_total = 0.
  for inputs, targets in test_loader:
    # move data to GPU
    inputs, targets = inputs.to(device), targets.to(device)

    # Forward pass
    outputs = model(inputs)

    # Get prediction
    # torch.max returns both max and argmax
    _, predictions = torch.max(outputs, 1)

    # update counts
    n_correct += (predictions == targets).sum().item()
    n_total += targets.shape[0]

  test_acc = n_correct / n_total
  print(f"Train acc: {train_acc:.4f}. Test acc: {test_acc:.4f}")
```

```
Train acc: 0.9978. Test acc: 0.9978
```

Save and Load Model

```
# Look at the state dict
model.state_dict()
```

```
                      ...,

                      [[ 5.1414e-03, -6.4583e-03, -1.4588e-02],
                       [-1.0115e-02, -1.0200e-02, -2.1285e-02],
                       [-6.3856e-03, -7.5624e-03, -2.0476e-02]],

                      [[-1.0839e-02,  6.7214e-03,  5.8918e-03],
                       [-9.5600e-04,  1.6651e-03,  7.1670e-03],
```

```
          [-1.8902e-02, -7.4955e-03, -9.8708e-04]],

         [[-4.4182e-02, -2.7011e-02, -1.5440e-02],
          [-4.1249e-02, -2.9579e-02, -1.0295e-02],
          [-1.9288e-02, -9.9171e-03,  5.1636e-03]]],


        [[[-2.9454e-02, -2.0776e-02, -1.4369e-02],
          [-8.1482e-03, -1.6480e-02, -1.4024e-02],
          [-8.4481e-03, -2.4206e-02, -1.5236e-02]],

         [[ 5.7170e-03,  1.4719e-02,  8.7575e-03],
          [ 7.4073e-03, -8.5165e-03, -1.1737e-02],
          [-2.1247e-03, -1.4506e-02, -1.4657e-02]],

         [[ 3.0560e-02,  4.0275e-02,  5.0865e-02],
          [-2.9477e-03,  2.0548e-02,  4.3352e-02],
          [-1.6921e-02,  4.7982e-03,  2.1169e-02]],

         ...,

         [[-1.9884e-02, -2.8676e-02, -1.7745e-02],
          [-1.8820e-02, -2.7692e-02, -3.7976e-02],
          [-7.1567e-03, -1.6576e-02, -6.9290e-03]],

         [[-3.3155e-03, -8.4667e-03,  4.0157e-03],
          [ 1.9905e-02, -1.0356e-02, -4.5904e-04],
          [ 3.1526e-02,  1.0053e-02,  1.1222e-02]],

         [[-2.6271e-02, -8.1591e-03, -2.9560e-02],
          [-3.3923e-02, -2.4079e-02, -2.2005e-02],
          [-3.4229e-02, -2.6150e-02, -1.4213e-02]]]], device='cuda:0'))
('features.28.bias',
 tensor([ 1.2299e-01,  1.9062e-02,  2.3202e-01,  5.8209e-02,  1.6951e-01
          9.3987e-02,  1.5288e-01,  3.5992e-02, -1.0673e-02,  4.5524e-02
         -3.5247e-02,  2.5516e-02,  3.7165e-01,  7.4319e-02, -5.5337e-02
          1.8086e-01,  9.7370e-03,  2.6187e-01,  1.8216e-01,  8.2105e-02
          1.6739e-01,  1.6799e-02, -7.1030e-02,  2.5600e-01,  1.1929e-01
         -6.0356e-02, -7.9072e-02,  2.1399e-02,  1.5857e-01,  1.0256e-01
          1.5698e-01,  1.0011e-01, -3.1958e-02,  1.2926e-01,  6.9638e-02
         -6.4992e-02, -2.9777e-02,  1.4160e-01,  1.9333e-01, -5.3614e-02
          1.7314e-02,  3.4937e-01,  5.4955e-01,  5.8277e-02,  1.8567e-01
         -5.6476e-02,  5.7856e-02, -3.3354e-01,  1.4938e-01, -4.4627e-02
          2.2324e-01, -1.3016e-01,  6.6589e-03,  1.7766e-01,  6.1969e-02
         -2.4899e-01,  7.3821e-02,  9.5302e-02,  1.0045e-02, -5.1411e-02
          1.2429e-01,  1.0792e-01, -4.1483e-02,  9.6371e-02, -1.0660e-01
          7.5465e-02,  2.5620e-01,  1.4411e-01, -2.2501e-02,  1.0363e-01
          7.0449e-02,  1.1813e-01,  3.5013e-02, -1.6152e-01,  7.5051e-02
          1.4182e-01,  2.1379e-01,  4.5386e-02,  2.3022e-01,  1.0673e-01
         -2.7411e-01, -8.0965e-03,  1.5953e-01,  1.0124e-01,  1.3451e-01
          6.2137e-02, -6.4873e-02,  5.2526e-02,  7.6597e-02,  4.9359e-02
```

```
# Save the model
saved_model_file = 'crack_detection_transfer_learning.pt'
torch.save(model.state_dict(), saved_model_file)
```

```
!ls
```

```
 5y9wdsg2zt-2.zip                                   data       sample_data
'Concrete Crack Images for Classification.rar'   Negative
 crack_detection_transfer_learning.pt             Positive
```

```
# Load the model
# Note: this makes more sense and is more compact when
# your model is a big class, as we will be seeing later.

# Define the model
model2 = models.vgg16(pretrained=True)

# Freeze VGG weights
for param in model2.parameters():
  param.requires_grad = False


# We want to replace the 'classifier'
model2.classifier

n_features = model2.classifier[0].in_features
n_features

# We're doing binary classification
model2.classifier = nn.Linear(n_features, 2)

# Let's see what the model is now
print(model2)
model2.load_state_dict(torch.load(saved_model_file))
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Linear(in_features=25088, out_features=2, bias=True)
)
<All keys matched successfully>
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(device)
model2.to(device)
```

```
cuda:0
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
      (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (25): ReLU(inplace=True)
      (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (27): ReLU(inplace=True)
      (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (29): ReLU(inplace=True)
      (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Linear(in_features=25088, out_features=2, bias=True)
  )
```

```
# Evaluate the new model
# Resuls should be the same!

n_correct = 0.
n_total = 0.
for inputs, targets in train_loader:
  # move data to GPU
  inputs, targets = inputs.to(device), targets.to(device)

  # Forward pass
  outputs = model2(inputs)

  # Get prediction
  # torch.max returns both max and argmax
  _, predictions = torch.max(outputs, 1)

  # update counts
  n_correct += (predictions == targets).sum().item()
  n_total += targets.shape[0]

train_acc = n_correct / n_total

n_correct = 0.
n_total = 0.
for inputs, targets in test_loader:
  # move data to GPU
  inputs, targets = inputs.to(device), targets.to(device)

  # Forward pass
  outputs = model2(inputs)

  # Get prediction
  # torch.max returns both max and argmax
```

```
    _, predictions = torch.max(outputs, 1)

    # update counts
    n_correct += (predictions == targets).sum().item()
    n_total += targets.shape[0]

  test_acc = n_correct / n_total
  print(f"Train acc: {train_acc:.4f}. Test acc: {test_acc:.4f}")
```

```
Train acc: 0.9977. Test acc: 0.9978
```

```
# Download the model
from google.colab import files
files.download(saved_model_file)
```

Related Work

- [Concrete-Crack-Detection](#) The model acheived 98% accuracy on the validation set. (Same dataset)

✓ 0s completed at 4:22 AM