from \_\_future\_\_ import print\_function

import numpy as np

import random, sys, os, json

import torch

import torch.nn as nn

import torch.nn.functional as F

import torch.optim as optim

from torch.autograd import Variable

from torchvision import models

from utils import \*

import transforms

import IPython

""" Base model class. """

class BaseModel(nn.Module):

def \_\_init\_\_(self, distribution=transforms.identity, n=1):

super(BaseModel, self).\_\_init\_\_()

if None not in [distribution, n]:

self.distribution, self.n = distribution, n

def forward(self, x):

raise NotImplementedError()

@property

def distribution(self):

return self.\_\_distribution

@distribution.setter

def distribution(self, x):

self.\_\_distribution = x

@property

def n(self):

return self.\_\_n

@n.setter

def n(self, n):

self.\_\_n = n

def set\_distribution(self, distribution=transforms.identity, n=1):

self.distribution, self.n = distribution, n

@classmethod

def load(cls, weights\_file=None, distribution=transforms.identity, n=1):

model = cls(distribution=distribution, n=n)

if weights\_file is not None:

model.load\_state\_dict(torch.load(weights\_file))

return model

def save(self, weights\_file, verbose=False):

if verbose:

print(f"Saving model to {weights\_file}")

torch.save(self.state\_dict(), weights\_file)

"""

DataParallel wrapper for BaseModels that exposes the same methods

(including save and distribution variables) without a .module() call.

"""

class DataParallelModel(BaseModel):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(distribution=None, n=None)

self.parallel\_apply = nn.DataParallel(\*args, \*\*kwargs)

def forward(self, x):

return self.parallel\_apply(x)

@property

def distribution(self):

return self.parallel\_apply.module.distribution

@distribution.setter

def distribution(self, x):

self.parallel\_apply.module.distribution = x

@property

def n(self):

return self.parallel\_apply.module.n

@n.setter

def n(self, n):

self.parallel\_apply.module.n = n

@property

def module(self):

return self.parallel\_apply.module

@classmethod

def load(cls, weights\_file=None, distribution=transforms.identity, n=1):

model = cls(distribution=distribution, n=n)

if weights\_file is not None:

model.parallel\_apply.module.load\_state\_dict(torch.load(weights\_file))

return model

def save(self, weights\_file, verbose=False):

if verbose:

print(f"Saving model to {weights\_file}")

torch.save(self.parallel\_apply.module.state\_dict(), weights\_file)

"""

Simple decoding network with squeezenet features and a

pooling-based linear bit transform.

"""

class DecodingNet(BaseModel):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(DecodingNet, self).\_\_init\_\_(\*args, \*\*kwargs)

self.features = models.squeezenet1\_1(pretrained=True).features

self.classifier = nn.Sequential(nn.Linear(512 \* 8, TARGET\_SIZE \* 2))

# nn.ReLU(inplace=True),

# nn.Linear(4096, TARGET\_SIZE\*2))

self.bn = nn.BatchNorm2d(512)

self.to(DEVICE)

def forward(self, x):

x = torch.cat([self.distribution(x).unsqueeze(1) for i in range(0, self.n)], dim=1)

B, N, C, H, W = x.shape

x = torch.cat(

[

((x[:, :, 0] - 0.485) / (0.229)).unsqueeze(2),

((x[:, :, 1] - 0.456) / (0.224)).unsqueeze(2),

((x[:, :, 2] - 0.406) / (0.225)).unsqueeze(2),

],

dim=2,

)

x = x.view(B \* N, C, H, W)

x = self.features(x)

x = torch.cat([F.avg\_pool2d(x, (x.shape[2] // 2)), F.max\_pool2d(x, (x.shape[2] // 2))], dim=1)

x = x.view(x.size(0), -1)

x = (x - x.mean(dim=1, keepdim=True)) / (x.std(dim=1, keepdim=True))

x = self.classifier(x)

x = x.view(B, N, TARGET\_SIZE, 2) # .mean(dim=0) # reshape and average

return F.softmax(x, dim=3)[:, :, :, 0].clamp(min=0, max=1)

"""

Decoding network with squeezenet features and a

gram-matrix based output that connects to intermediate layers.

"""

class DecodingGramNet(BaseModel):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(DecodingGramNet, self).\_\_init\_\_(\*args, \*\*kwargs)

self.features = models.squeezenet1\_1(pretrained=True).features

# self.gram\_classifiers = nn.ModuleList([

# nn.Linear(256\*\*2, 256),

# nn.Linear(384\*\*2, 256),

# nn.Linear(512\*\*2, 256),

# ])

self.indices = [6, 8, 10, 12]

self.classifier = nn.Linear(1408, TARGET\_SIZE \* 2)

self.to(DEVICE)

def forward(self, x):

x = torch.cat([self.distribution(x).unsqueeze(1) for i in range(0, self.n)], dim=1)

B, N, C, H, W = x.shape

x = torch.cat(

[

((x[:, :, 0] - 0.485) / (0.229)).unsqueeze(2),

((x[:, :, 1] - 0.456) / (0.224)).unsqueeze(2),

((x[:, :, 2] - 0.406) / (0.225)).unsqueeze(2),

],

dim=2,

)

x = x.view(B \* N, C, H, W)

layers = list(self.features.\_modules.values())

gram\_maps = []

for i, layer in enumerate(layers):

x = layer(x)

j = self.indices.index(i) if i in self.indices else None

if j is not None:

y = F.max\_pool2d(x, (x.shape[2], x.shape[3]))

gram\_maps.append(y)

# gram\_maps = []

# for layer, clf in zip(layers[-3:], self.gram\_classifiers):

# x = layer(x)

# y = gram(x).view(x.shape[0], -1)

# print (x.shape, y.shape)

# print (clf)

# #gram\_maps.append(clf(y))

x = torch.cat(gram\_maps, dim=1)

x = x.view(x.size(0), -1)

x = (x - x.mean(dim=1, keepdim=True)) / (x.std(dim=1, keepdim=True))

x = self.classifier(x)

x = x.view(B, N, TARGET\_SIZE, 2) # .mean(dim=0) # reshape and average

return F.softmax(x, dim=3)[:, :, :, 0].clamp(min=0, max=1)

"""

Tiny un-pretrained decoding network.

"""

class TinyDecodingNet(BaseModel):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.conv1 = nn.Conv2d(3, 128, (3, 3), padding=1)

self.conv2 = nn.Conv2d(128, 128, (3, 3), padding=1)

self.conv3 = nn.Conv2d(128, 128, (3, 3), padding=1)

self.conv4 = nn.Conv2d(128, 2 \* TARGET\_SIZE, (3, 3), padding=1)

self.to(DEVICE)

def forward(self, x):

x = torch.cat([self.distribution(x).unsqueeze(1) for i in range(0, self.n)], dim=1)

B, N, C, H, W = x.shape

x = torch.cat(

[

((x[:, :, 0] - 0.485) / (0.229)).unsqueeze(2),

((x[:, :, 1] - 0.456) / (0.224)).unsqueeze(2),

((x[:, :, 2] - 0.406) / (0.225)).unsqueeze(2),

],

dim=2,

)

x = x.view(B \* N, C, H, W).contiguous()

# print (x.shape)

# x = F.relu(self.conv1(x))

# x = F.max\_pool2d(x, 2)

# print (x.shape)

# x = F.relu(self.conv2(x))

# x = F.max\_pool2d(x, 2)

# print (x.shape)

# x = F.relu(self.conv3(x))

# x = F.max\_pool2d(x, 2)

# print (x.shape)

# x = F.relu(self.conv4(x))

# x = F.max\_pool2d(x, 2)

# print (x.shape)

x = F.avg\_pool2d(x, (x.shape[2], x.shape[3]))

x = x.view(B, N, TARGET\_SIZE, 2) # .mean(dim=0) # reshape and average

return F.softmax(x, dim=3)[:, :, :, 0].clamp(min=0, max=1)

"""Decoding network that tries to predict on images using a dilated DCNN,

which should theoretically be invariant to any scale of input. """

"Understanding GitHub Code Search syntax."

Merge branch 'master' of github.com:nikcheerla/neuralhash

<https://github.com/nikcheerla/neuralhash>

**building the next-gen watermark with deep learning**.

class DilatedDecodingNet(BaseModel):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(DilatedDecodingNet, self).\_\_init\_\_(\*args, \*\*kwargs)

self.features = models.vgg11(pretrained=True)

self.features.eval()

self.classifier = nn.Linear(512 \*\* 2, TARGET\_SIZE \* 2)

self.gram = GramMatrix()

if USE\_CUDA:

self.cuda()

def forward(self, x, verbose=False, distribution=transforms.identity, n=1, return\_variance=False):

# make sure to center the image and divide by standard deviation

x = torch.cat(

[

((x[0] - 0.485) / (0.229)).unsqueeze(0),

((x[1] - 0.456) / (0.224)).unsqueeze(0),

((x[2] - 0.406) / (0.225)).unsqueeze(0),

],

dim=0,

)

x = torch.cat([distribution(x).unsqueeze(0) for i in range(0, n)], dim=0)

# vgg layers

dilation\_factor = 1

for layer in list(self.features.features.\_modules.values()):

if isinstance(layer, nn.Conv2d):

x = F.conv2d(

x,

layer.weight,

bias=layer.bias,

stride=layer.stride,

padding=tuple(layer.padding \* np.array(dilation\_factor)),

dilation=dilation\_factor,

)

elif isinstance(layer, nn.MaxPool2d):

if dilation\_factor == 1:

x = F.max\_pool2d(x, 2, stride=1, dilation=1)

x = F.pad(x, (1, 0, 1, 0))

else:

x = F.max\_pool2d(x, 2, stride=1, dilation=dilation\_factor)

x = F.pad(x, [dilation\_factor // 2] \* 4)

dilation\_factor \*= 2

else:

x = layer(x)

x = self.gram(x)

x = x.view(x.size(0), -1)

x = (x - x.mean(dim=1, keepdim=True)) / (x.std(dim=1, keepdim=True))

x = self.classifier(x)

x = x.view(x.size(0), TARGET\_SIZE, 2) # .mean(dim=0) # reshape and average

predictions = F.softmax(x, dim=2)[:, :, 0]

return predictions

DecodingModel = eval(MODEL\_TYPE)

if \_\_name\_\_ == "\_\_main\_\_":

model = nn.DataParallel(TinyDecodingNet(n=16, distribution=transforms.identity))

images = torch.randn(4, 3, 224, 224).float().to(DEVICE)

x = model.forward(images)

print(x.shape)