fast-neural-style :city_sunrise: :rocket:
This repository contains a pytorch implementation of an algorithm for artistic style transfer. The algorithm can be used to mix the content of an image with the style of another image. For example, here is a photograph of a door arch rendered in the style of a stained glass painting.

The model uses the method described in [Perceptual Losses for Real-Time Style Transfer and Super-Resolution](https://arxiv.org/abs/1603.08155) along with [Instance Normalization](https://arxiv.org/pdf/1607.08022.pdf). The saved-models for examples shown in the README can be downloaded from [here](https://www.dropbox.com/s/lrvwfehqdcxoza8/saved_models.zip?dl=0).

<p align="center">
    <img src="images/style-images/mosaic.jpg" height="200px">
    <img src="images/content-images/amber.jpg" height="200px">
    <img src="images/output-images/amber-mosaic.jpg" height="440px">
</p>

Requirements
The program is written in Python, and uses [pytorch](http://pytorch.org/), [scipy](https://www.scipy.org). A GPU is not necessary, but can provide a significant speed up especially for training a new model. Regular sized images can be styled on a laptop or desktop using saved models.

Usage
Stylize image
```
```

python neural_style/neural_style.py eval --content-image </path/to/content/image> --model </path/to/saved/model> --output-image </path/to/output/image> --cuda 0
```
* `--content-image`: path to content image you want to stylize.
* `--model`: saved model to be used for stylizing the image (eg: `mosaic.pth`)
* `--output-image`: path for saving the output image.
* `--content-scale`: factor for scaling down the content image if memory is an issue (eg: value of 2 will halve the height and width of content-image)
* `--cuda`: set it to 1 for running on GPU, 0 for CPU.

Train model
```bash
python neural_style/neural_style.py train --dataset </path/to/train-dataset> --style-image </path/to/style/image> --save-model-dir </path/to/save-model/folder> --epochs 2 --cuda 1
```

There are several command line arguments, the important ones are listed below
* `--dataset`: path to training dataset, the path should point to a folder containing another folder with all the training images. I used COCO 2014 Training images dataset [80K/13GB] [(download)](http://mscoco.org/dataset/#download).
* `--style-image`: path to style-image.
* `--save-model-dir`: path to folder where trained model will be saved.
* `--cuda`: set it to 1 for running on GPU, 0 for CPU.
```

Refer to ``neural_style/neural_style.py`` for other command line arguments. For training new models you might have to tune the values of `--content-weight` and `--style-weight`. The mosaic style model shown above was trained with `--content-weight 1e5` and `--style-weight 1e10`. The remaining 3 models were also trained with similar order of weight parameters with slight variation in the `--style-weight` (`5e10` or `1e11`).

Models

Models for the examples shown below can be downloaded from [here](https://www.dropbox.com/s/lrvwfehqdcxoza8/saved_models.zip?dl=0) or by running the script ``download_saved_models.py``.

<div align='center'>
  <img src='images/content-images/amber.jpg' height="174px">
</div>

<div align='center'>
  <img src='images/style-images/mosaic.jpg' height="174px">
  <img src='images/output-images/amber-mosaic.jpg' height="174px">
  <img src='images/output-images/amber-candy.jpg' height="174px">
  <img src='images/style-images/candy.jpg' height="174px">

  <img src='images/style-images/rain-princess-cropped.jpg' height="174px">
  <img src='images/output-images/amber-rain-princess.jpg' height="174px">

```
  <img src='images/output-images/amber-udnie.jpg'
height="174px">
  <img src='images/style-images/udnie.jpg'
height="174px">
</div>
import os
import zipfile

# PyTorch 1.1 moves _download_url_to_file
#   from torch.utils.model_zoo to torch.hub
# PyTorch 1.0 exists another _download_url_to_file
#   2 argument
# TODO: If you remove support PyTorch 1.0 or older,
#       You should remove torch.utils.model_zoo
#       Ref. PyTorch #18758
#         https://github.com/pytorch/pytorch/pull/
18758/commits
try:
    from torch.utils.model_zoo import
_download_url_to_file
except ImportError:
    from torch.hub import _download_url_to_file


def unzip(source_filename, dest_dir):
    with zipfile.ZipFile(source_filename) as zf:
        zf.extractall(path=dest_dir)


if __name__ == '__main__':
    _download_url_to_file('https://www.dropbox.com/
s/lrvwfehqdcxoza8/saved_models.zip?dl=1',
'saved_models.zip', None, True)
    unzip('saved_models.zip', '.')
```

```
import argparse
import os
import sys
import time
import re

import numpy as np
import torch
from torch.optim import Adam
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import transforms
import torch.onnx

import utils
from transformer_net import TransformerNet
from vgg import Vgg16


def check_paths(args):
    try:
        if not os.path.exists(args.save_model_dir):
            os.makedirs(args.save_model_dir)
        if args.checkpoint_model_dir is not None
and not (os.path.exists(args.checkpoint_model_dir)):
            os.makedirs(args.checkpoint_model_dir)
    except OSError as e:
        print(e)
        sys.exit(1)


def train(args):
    device = torch.device("cuda" if args.cuda else
"cpu")

    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
```

```
    transform = transforms.Compose([
        transforms.Resize(args.image_size),
        transforms.CenterCrop(args.image_size),
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.mul(255))
    ])
    train_dataset =
datasets.ImageFolder(args.dataset, transform)
    train_loader = DataLoader(train_dataset,
batch_size=args.batch_size)

    transformer = TransformerNet().to(device)
    optimizer = Adam(transformer.parameters(),
args.lr)
    mse_loss = torch.nn.MSELoss()

    vgg = Vgg16(requires_grad=False).to(device)
    style_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.mul(255))
    ])
    style = utils.load_image(args.style_image,
size=args.style_size)
    style = style_transform(style)
    style = style.repeat(args.batch_size, 1, 1,
1).to(device)

    features_style =
vgg(utils.normalize_batch(style))
    gram_style = [utils.gram_matrix(y) for y in
features_style]

    for e in range(args.epochs):
        transformer.train()
        agg_content_loss = 0.
        agg_style_loss = 0.
```

```
        count = 0
        for batch_id, (x, _) in
enumerate(train_loader):
            n_batch = len(x)
            count += n_batch
            optimizer.zero_grad()

            x = x.to(device)
            y = transformer(x)

            y = utils.normalize_batch(y)
            x = utils.normalize_batch(x)

            features_y = vgg(y)
            features_x = vgg(x)

            content_loss = args.content_weight *
mse_loss(features_y.relu2_2, features_x.relu2_2)

            style_loss = 0.
            for ft_y, gm_s in zip(features_y,
gram_style):
                gm_y = utils.gram_matrix(ft_y)
                style_loss += mse_loss(gm_y,
gm_s[:n_batch, :, :])
            style_loss *= args.style_weight

            total_loss = content_loss + style_loss
            total_loss.backward()
            optimizer.step()

            agg_content_loss += content_loss.item()
            agg_style_loss += style_loss.item()

            if (batch_id + 1) % args.log_interval
== 0:
                mesg = "{}\tEpoch {}:\t[{}/{}]
```

```
\tcontent: {:.6f}\tstyle: {:.6f}\ttotal: {:.
6f}".format(
                    time.ctime(), e + 1, count,
len(train_dataset),

agg_content_loss / (batch_id + 1),
                                  agg_style_loss /
(batch_id + 1),
                                  (agg_content_loss
+ agg_style_loss) / (batch_id + 1)
                )
                print(mesg)

            if args.checkpoint_model_dir is not
None and (batch_id + 1) % args.checkpoint_interval
== 0:
                transformer.eval().cpu()
                ckpt_model_filename = "ckpt_epoch_"
+ str(e) + "_batch_id_" + str(batch_id + 1) + ".pth"
                ckpt_model_path =
os.path.join(args.checkpoint_model_dir,
ckpt_model_filename)

torch.save(transformer.state_dict(),
ckpt_model_path)
                transformer.to(device).train()

    # save model
    transformer.eval().cpu()
    save_model_filename = "epoch_" +
str(args.epochs) + "_" +
str(time.ctime()).replace(' ', '_') + "_" + str(
        args.content_weight) + "_" +
str(args.style_weight) + ".model"
    save_model_path =
os.path.join(args.save_model_dir,
save_model_filename)
```

```
    torch.save(transformer.state_dict(),
save_model_path)

    print("\nDone, trained model saved at",
save_model_path)


def stylize(args):
    device = torch.device("cuda" if args.cuda else
"cpu")

    content_image =
utils.load_image(args.content_image,
scale=args.content_scale)
    content_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.mul(255))
    ])
    content_image = content_transform(content_image)
    content_image =
content_image.unsqueeze(0).to(device)

    if args.model.endswith(".onnx"):
        output = stylize_onnx_caffe2(content_image,
args)
    else:
        with torch.no_grad():
            style_model = TransformerNet()
            state_dict = torch.load(args.model)
            # remove saved deprecated running_*
keys in InstanceNorm from the checkpoint
            for k in list(state_dict.keys()):
                if re.search(r'in\d+\.running_(mean|
var)$', k):
                    del state_dict[k]
            style_model.load_state_dict(state_dict)
            style_model.to(device)
```

```
            if args.export_onnx:
                assert
args.export_onnx.endswith(".onnx"), "Export model
file should end with .onnx"
                output =
torch.onnx._export(style_model, content_image,
args.export_onnx).cpu()
            else:
                output =
style_model(content_image).cpu()
    utils.save_image(args.output_image, output[0])


def stylize_onnx_caffe2(content_image, args):
    """
    Read ONNX model and run it using Caffe2
    """

    assert not args.export_onnx

    import onnx
    import onnx_caffe2.backend

    model = onnx.load(args.model)

    prepared_backend =
onnx_caffe2.backend.prepare(model, device='CUDA' if
args.cuda else 'CPU')
    inp = {model.graph.input[0].name:
content_image.numpy()}
    c2_out = prepared_backend.run(inp)[0]

    return torch.from_numpy(c2_out)


def main():
    main_arg_parser =
```

```
argparse.ArgumentParser(description="parser for
fast-neural-style")
    subparsers =
main_arg_parser.add_subparsers(title="subcommands",
dest="subcommand")

    train_arg_parser =
subparsers.add_parser("train", help="parser for
training arguments")
    train_arg_parser.add_argument("--epochs",
type=int, default=2,
                                  help="number of
training epochs, default is 2")
    train_arg_parser.add_argument("--batch-size",
type=int, default=4,
                                  help="batch size
for training, default is 4")
    train_arg_parser.add_argument("--dataset",
type=str, required=True,
                                  help="path to
training dataset, the path should point to a folder
"
                                       "containing
another folder with all the training images")
    train_arg_parser.add_argument("--style-image",
type=str, default="images/style-images/mosaic.jpg",
                                  help="path to
style-image")
    train_arg_parser.add_argument("--save-model-
dir", type=str, required=True,
                                  help="path to
folder where trained model will be saved.")
    train_arg_parser.add_argument("--checkpoint-
model-dir", type=str, default=None,
                                  help="path to
folder where checkpoints of trained models will be
saved")
```

```
    train_arg_parser.add_argument("--image-size",
type=int, default=256,
                                  help="size of
training images, default is 256 X 256")
    train_arg_parser.add_argument("--style-size",
type=int, default=None,
                                  help="size of
style-image, default is the original size of style
image")
    train_arg_parser.add_argument("--cuda",
type=int, required=True,
                                  help="set it to 1
for running on GPU, 0 for CPU")
    train_arg_parser.add_argument("--seed",
type=int, default=42,
                                  help="random seed
for training")
    train_arg_parser.add_argument("--content-
weight", type=float, default=1e5,
                                  help="weight for
content-loss, default is 1e5")
    train_arg_parser.add_argument("--style-weight",
type=float, default=1e10,
                                  help="weight for
style-loss, default is 1e10")
    train_arg_parser.add_argument("--lr",
type=float, default=1e-3,
                                  help="learning
rate, default is 1e-3")
    train_arg_parser.add_argument("--log-interval",
type=int, default=500,
                                  help="number of
images after which the training loss is logged,
default is 500")
    train_arg_parser.add_argument("--checkpoint-
interval", type=int, default=2000,
                                  help="number of
```

```
batches after which a checkpoint of the trained
model will be created")

    eval_arg_parser = subparsers.add_parser("eval",
help="parser for evaluation/stylizing arguments")
    eval_arg_parser.add_argument("--content-image",
type=str, required=True,
                                 help="path to
content image you want to stylize")
    eval_arg_parser.add_argument("--content-scale",
type=float, default=None,
                                 help="factor for
scaling down the content image")
    eval_arg_parser.add_argument("--output-image",
type=str, required=True,
                                 help="path for
saving the output image")
    eval_arg_parser.add_argument("--model",
type=str, required=True,
                                 help="saved model
to be used for stylizing the image. If file ends
in .pth - PyTorch path is used, if in .onnx -
Caffe2 path")
    eval_arg_parser.add_argument("--cuda",
type=int, required=True,
                                 help="set it to 1
for running on GPU, 0 for CPU")
    eval_arg_parser.add_argument("--export_onnx",
type=str,
                                 help="export ONNX
model to a given file")

    args = main_arg_parser.parse_args()

    if args.subcommand is None:
        print("ERROR: specify either train or eval")
        sys.exit(1)
```

```
    if args.cuda and not torch.cuda.is_available():
        print("ERROR: cuda is not available, try
running on CPU")
        sys.exit(1)

    if args.subcommand == "train":
        check_paths(args)
        train(args)
    else:
        stylize(args)


if __name__ == "__main__":
    main()
import torch


class TransformerNet(torch.nn.Module):
    def __init__(self):
        super(TransformerNet, self).__init__()
        # Initial convolution layers
        self.conv1 = ConvLayer(3, 32,
kernel_size=9, stride=1)
        self.in1 = torch.nn.InstanceNorm2d(32,
affine=True)
        self.conv2 = ConvLayer(32, 64,
kernel_size=3, stride=2)
        self.in2 = torch.nn.InstanceNorm2d(64,
affine=True)
        self.conv3 = ConvLayer(64, 128,
kernel_size=3, stride=2)
        self.in3 = torch.nn.InstanceNorm2d(128,
affine=True)
        # Residual layers
        self.res1 = ResidualBlock(128)
        self.res2 = ResidualBlock(128)
        self.res3 = ResidualBlock(128)
```

```
        self.res4 = ResidualBlock(128)
        self.res5 = ResidualBlock(128)
        # Upsampling Layers
        self.deconv1 = UpsampleConvLayer(128, 64,
kernel_size=3, stride=1, upsample=2)
        self.in4 = torch.nn.InstanceNorm2d(64,
affine=True)
        self.deconv2 = UpsampleConvLayer(64, 32,
kernel_size=3, stride=1, upsample=2)
        self.in5 = torch.nn.InstanceNorm2d(32,
affine=True)
        self.deconv3 = ConvLayer(32, 3,
kernel_size=9, stride=1)
        # Non-linearities
        self.relu = torch.nn.ReLU()

    def forward(self, X):
        y = self.relu(self.in1(self.conv1(X)))
        y = self.relu(self.in2(self.conv2(y)))
        y = self.relu(self.in3(self.conv3(y)))
        y = self.res1(y)
        y = self.res2(y)
        y = self.res3(y)
        y = self.res4(y)
        y = self.res5(y)
        y = self.relu(self.in4(self.deconv1(y)))
        y = self.relu(self.in5(self.deconv2(y)))
        y = self.deconv3(y)
        return y


class ConvLayer(torch.nn.Module):
    def __init__(self, in_channels, out_channels,
kernel_size, stride):
        super(ConvLayer, self).__init__()
        reflection_padding = kernel_size // 2
        self.reflection_pad =
```

```
torch.nn.ReflectionPad2d(reflection_padding)
        self.conv2d = torch.nn.Conv2d(in_channels,
out_channels, kernel_size, stride)

    def forward(self, x):
        out = self.reflection_pad(x)
        out = self.conv2d(out)
        return out


class ResidualBlock(torch.nn.Module):
    """ResidualBlock
    introduced in: https://arxiv.org/abs/1512.03385
    recommended architecture: http://torch.ch/blog/
2016/02/04/resnets.html
    """

    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = ConvLayer(channels, channels,
kernel_size=3, stride=1)
        self.in1 =
torch.nn.InstanceNorm2d(channels, affine=True)
        self.conv2 = ConvLayer(channels, channels,
kernel_size=3, stride=1)
        self.in2 =
torch.nn.InstanceNorm2d(channels, affine=True)
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        residual = x
        out = self.relu(self.in1(self.conv1(x)))
        out = self.in2(self.conv2(out))
        out = out + residual
        return out
```

```
class UpsampleConvLayer(torch.nn.Module):
    """UpsampleConvLayer
    Upsamples the input and then does a
convolution. This method gives better results
    compared to ConvTranspose2d.
    ref: http://distill.pub/2016/deconv-
checkerboard/
    """

    def __init__(self, in_channels, out_channels,
kernel_size, stride, upsample=None):
        super(UpsampleConvLayer, self).__init__()
        self.upsample = upsample
        reflection_padding = kernel_size // 2
        self.reflection_pad =
torch.nn.ReflectionPad2d(reflection_padding)
        self.conv2d = torch.nn.Conv2d(in_channels,
out_channels, kernel_size, stride)

    def forward(self, x):
        x_in = x
        if self.upsample:
            x_in =
torch.nn.functional.interpolate(x_in,
mode='nearest', scale_factor=self.upsample)
        out = self.reflection_pad(x_in)
        out = self.conv2d(out)
        return out
import torch
from PIL import Image


def load_image(filename, size=None, scale=None):
    img = Image.open(filename)
    if size is not None:
        img = img.resize((size, size),
Image.ANTIALIAS)
```

```
    elif scale is not None:
        img = img.resize((int(img.size[0] / scale),
int(img.size[1] / scale)), Image.ANTIALIAS)
    return img


def save_image(filename, data):
    img = data.clone().clamp(0, 255).numpy()
    img = img.transpose(1, 2, 0).astype("uint8")
    img = Image.fromarray(img)
    img.save(filename)


def gram_matrix(y):
    (b, ch, h, w) = y.size()
    features = y.view(b, ch, w * h)
    features_t = features.transpose(1, 2)
    gram = features.bmm(features_t) / (ch * h * w)
    return gram


def normalize_batch(batch):
    # normalize using imagenet mean and std
    mean = batch.new_tensor([0.485, 0.456,
0.406]).view(-1, 1, 1)
    std = batch.new_tensor([0.229, 0.224,
0.225]).view(-1, 1, 1)
    batch = batch.div_(255.0)
    return (batch - mean) / std
from collections import namedtuple

import torch
from torchvision import models


class Vgg16(torch.nn.Module):
    def __init__(self, requires_grad=False):
```

```
        super(Vgg16, self).__init__()
        vgg_pretrained_features =
models.vgg16(pretrained=True).features
        self.slice1 = torch.nn.Sequential()
        self.slice2 = torch.nn.Sequential()
        self.slice3 = torch.nn.Sequential()
        self.slice4 = torch.nn.Sequential()
        for x in range(4):
            self.slice1.add_module(str(x),
vgg_pretrained_features[x])
        for x in range(4, 9):
            self.slice2.add_module(str(x),
vgg_pretrained_features[x])
        for x in range(9, 16):
            self.slice3.add_module(str(x),
vgg_pretrained_features[x])
        for x in range(16, 23):
            self.slice4.add_module(str(x),
vgg_pretrained_features[x])
        if not requires_grad:
            for param in self.parameters():
                param.requires_grad = False

    def forward(self, X):
        h = self.slice1(X)
        h_relu1_2 = h
        h = self.slice2(h)
        h_relu2_2 = h
        h = self.slice3(h)
        h_relu3_3 = h
        h = self.slice4(h)
        h_relu4_3 = h
        vgg_outputs = namedtuple("VggOutputs",
['relu1_2', 'relu2_2', 'relu3_3', 'relu4_3'])
        out = vgg_outputs(h_relu1_2, h_relu2_2,
h_relu3_3, h_relu4_3)
        return out
```