Style Transfer

This project was an exploration into a couple different ways to capture the style of one image and apply it to other images, i.e. turn different pictures into oil paintings, mosaics, modern art, etc.

The first method we derived was training a machine learning network to minimize losses between an image with a certain style and random training images. The trained network could take any random image, and convert it to the image with that style.

The second technique was a non-ML technique to create specifically a cartoon-like style. This was done by using an edge detection algorithm to find hard-edges of an image, then overlay that with the same image with colors smoothed-out.

Click below to see results of the before-after of images transitioned to different styles:

Project Overview

Problem Setup: We aim to be able to transform any image given to a different style, with the end result both containing the objects from the transformed image, but also undoubtedly carrying the desired style.

Approach: Our approach for the machine learning style-transfer is the same as Johnson et al's in "Perceptual Losses for Real-Time Style Transfer and Super-Resolution" and we aimed to implement the technique they describe: we use a deep neural network, call it the Image Transformer Network, that transforms the training and testing images to a given style. However, unlike Gatys et al in "A Neural Algorithm of Artistic Style" (the first paper to use neural networks for style transfer), we do not attempt to minimize the per pixel loss. Instead, for a faster method, we minimize perceptual loss functions: minimizing the Content Perceptual Loss helps in making the output image look like the content image while minimizing the Style Perceptual Loss helps in transferring the style from the style image to the output image. If we only minimized Content Perceptual Loss, then the output would look very similar to the content image but would have no style. The opposite would happen if we only minimize the Style Perceptual Loss. Thus, we combine the two losses and aim to find the right balance such that the output contains the contents of the content image but style of the style image. Both of these loss functions are based on a select set of features (tend to be higher level features) from a pre-trained neural network, the Loss Network. After training, our Image Transformer Network can transform any given content image to fit a certain style image.

Our approach for the cartoon style-transfer was based on observation on how cartoons look. We notices how cartoons tend to have very few colors compared to reality, and boundaries are often marked with black lines. To replicate this, we first took an image and converted it to black-and-white so that edge detection could be performed. This served as those boundaries. After that, we took the same image and smoothed out the colors with a median blur. Rather than a blur which takes an average, like Gaussian, median blur reduces the number of colors by replacing pixels by the median of surrounding pixels. By overlaying the edge-detection image on top of the smoothed out image, the result is a cartoon-ified version of the original.

Dataset: We want a large dataset of content images to train the Image Transformer Network: this is not just because of the usual reason of improving generalizability and performance on unseen data. More images allow for the network to learn from more of those higher level features in the Loss Network. Because this approach is similar to unsupervised learning, there is no need for labeled data. That is, we can use a large and diverse set of random, unlabeled images as content images. As such, our model uses about 36k images from the MS COCO 2017 dataset. The style images were chosen based on the style images from the paper, though we also looked at online images we liked.

The Loss Network is a pretrained model. We used the VGG16 model that is trained on the ImageNet dataset. The non-ML cartoon-style transfer didn't need any data as it was algorithmic.

Method/Architecture:

Image Transformer Network: the network has down-sampling layers, then residual block layers, and lastly up-sampling layers.

- Down-sampling layer: convolution → instance normalization → ReLU activation.
- Residual block layer: convolution → instance normalization → ReLU activation → convolution → instance normalization
- Up-sampling layer: nearest neighbor interpolation/upsampling → convolution → instance normalization → ReLU activation

This method of down-sampling and upsampling is less computationally expensive than conventional convolutional networks (additional details on the benefits of this architecture are in the paper by Johnson et al and its supplementary material).(Note: the original paper by Johnson et al used batch normalization but a more recent paper specified the benefits of using instance normalization instead. Additional details on instance normalization are in Instance Normalization: The Missing Ingredient for Fast Stylization by Ulyanov et al).

Loss Network: uses the pretrained weights and features from the VGG16 network. Specifically, we use the "relu1_2," "relu2_2," "relu3_3," and "relu4_3" layers of the VGG 16 network. Using these layers helps us learn and preserve the high level structure of the image while ignoring the less important aspects like shape, texture, etc (additional details on the reasons for this architecture are in the paper by Johnson et al) (Note: in order to understand how to extract information about these layers from the VGG16 network, we looked at the table in Very Deep Convolutional Networks for large-Scale Image Recognition by Simonyan et al that describes VGG16's architecture)

Content Perceptual Loss: we pass in the target input (content) image into the Image Transformer Network and get an output. Then, we pass in the target (content) image and the output into the Loss Network and get their features. The loss is then the squared, normalized Euclidean distance between the two sets of features from the relu2_2 layer. The loss is multiplied by a scalar weight depending on how much content we want to retain.

Style Perceptual Loss: we pass in the target input (content) image into the Image Transformer Network and get an output. Then, we pass in the target (style) image and the output into the Loss Network and get their features. We create gram matrices from both sets of features which represent how similar/different the two images are in terms of the high level features (i.e style) (additional details on the gram matrix in the context of neural style transfer at Neural Network Intuitions). The loss is then the Frobenius norm of the difference between the two matrices. The loss is multiplied by a style weight depending on how much style we want to retain.

Hyper-parameters: we trained using an Adam optimizer with a learning rate of 0.001. We used a content weight of 1e5 for all of our images but varied the style weight from 1e10 to 1e11 depending on how stylized we wanted our output images. We trained for 2 epochs.

Cartoon style-transfer: We utilized OpenCV for image modification. We utilized the canny edge detection algorithm for accurate, quick calculation of the edges of an image. This image resulted in a black background and white edges, so the image was manually reversed for white background and black edges. A median blur was chosen for smoothing out the colors of the image, because it took the median of surrounding pixels rather than the average like Gaussian would, making it merely reduce noise and keep one color overall rather than make a new color.

Components Implemented Ourselves: We used the Johnson et al research paper and the provided supplementary material as references for our code implementation. We fully implemented the Image Transformer Network and the Loss Network by ourselves. For certain aspects that were unfamiliar to us (e.g using up-sampling before convolution and using only certain layers of the pre-trained VGG16 network), we referenced outside code. We also used the code for some utility functions like loading images, etc (with the exception of constructing the gram matrix). We also did not fully implement the styling and training script but did remove things from the original code that were not necessary for our approach. The algorithmic cartoon style transfer was entirely self-done. The only change to dataset was to train model on 1/4 of the dataset; the full dataset had >120k images and would have taken too long to train without any significant improvement.

References:

- https://cs.stanford.edu/people/jcjohns/eccv16/. - Paper followed to create ML models
- https://github.com/pytorch/examples/tree/main/fast_neural_style/neural_style - Sample hyper-parameters for faster tuning and sample style images found here
- https://www.kaggle.com/datasets/awsaf49/coco-2017-dataset - Massive dataset for training images found here
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution - https://arxiv.org/pdf/1603.08155.pdf
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution: Supplementary Material - https://cs.stanford.edu/people/jcjohns/papers/fast-style/fast-style-supp.pdf
- Instance Normalization: The Missing Ingredient for Fast Stylization - https://arxiv.org/pdf/1607.08022.pdf
- Very Deep Convolutional Networks for large-Scale Image Recognition - https://arxiv.org/pdf/1409.1556.pdf
- Neural Network Intuitions - https://towardsdatascience.com/neural-networks-intuitions-2-dot-product-gram-matrix-and-neural-style-transfer-5d39653e7916
- Referenced code - https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py

Made with Squarespace

```
import cv2
import numpy as np

img_gry = cv2.imread('/Users/varshanmuhunthan/Desktop/woman_face.jpeg', cv2.COLOR_BGR2GRAY)
img_gry = cv2.medianBlur(img_gry, 3)
edges = cv2.Canny(img_gry,140,180)
edges = cv2.bitwise_not(edges)
print(edges.shape)
cv2.imwrite('/Users/varshanmuhunthan/Desktop/woman_face_edge.jpeg', edges)

edges_3 = np.zeros_like(img_gry)
edges_3[:,:,0] = edges
edges_3[:,:,1] = edges
edges_3[:,:,2] = edges

img_clr = cv2.imread('/Users/varshanmuhunthan/Desktop/woman_face.jpeg', cv2.IMREAD_UNCHANGED)
img_blr = cv2.medianBlur(img_clr, 7)
cv2.imwrite('/Users/varshanmuhunthan/Desktop/woman_face_blur.jpeg', img_blr)
print(img_blr.shape)

# img_blnd = cv2.addWeighted(edges_3, 0.7, img_blr, 1, 0)
img_blnd = np.zeros_like(img_clr)
for h in range(len(img_clr)):
    for w in range(len(img_clr[h])):
        if edges[h,w] == 255 :
            img_blnd[h, w, :] = img_blr[h, w, :]
        else:
            img_blnd[h, w, :] = 0

cv2.imwrite('/Users/varshanmuhunthan/Desktop/woman_face_cartoon.jpeg', img_blnd)
```

References:

- https://cs.stanford.edu/people/jcjohns/eccv16/. - Paper followed to create ML models
- https://github.com/pytorch/examples/tree/main/fast_neural_style/neural_style - Sample hyper-parameters for faster tuning and sample style images found here
- https://www.kaggle.com/datasets/awsaf49/coco-2017-dataset - Massive dataset for training images found here
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution - https://arxiv.org/pdf/1603.08155.pdf
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution: Supplementary Material - https://cs.stanford.edu/people/jcjohns/papers/fast-style/fast-style-supp.pdf
- Instance Normalization: The Missing Ingredient for Fast Stylization - https://arxiv.org/pdf/1607.08022.pdf
- Very Deep Convolutional Networks for large-Scale Image Recognition - https://arxiv.org/pdf/1409.1556.pdf
- Neural Network Intuitions - https://towardsdatascience.com/neural-networks-intuitions-2-dot-product-gram-matrix-and-neural-style-transfer-5d39653e7916
- Referenced code - https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py

Made with Squarespace

CSE455 Final Project - Kevin Kumer & Varshan Muhunthan


```
"""
1)  The original paper's architecture used batch normalization. However, the author later
    mentioned that instance normalization led to better results. We also use instance normalization
    but keep an option to use the original batch normalization.

        More details on instance normalization:
            Instance Normalization: The Missing Ingredient for Fast Stylization
            ref = https://arxiv.org/abs/1607.08022

2)  No tanh function is used after the last layer of the model because pixel value scaling
    and normalization is done during training using the VGG16 mean and std instead of in the model.

3)  For the upsampling layers, we upsample using nearest-neighbor up-sampling followed by a convolution
    instead of a tranpose convolution. This helps avoid the checkerboard artifact.

        More details on this method:
            Deconvolution and Checkerboard Artifacts
            ref = https://distill.pub/2016/deconv-checkerboard/

        Because we had never seen this method before, we referenced outside code on how to
        do the up-sampling followed by a convolution:
            ref = https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/transformer_net.py
"""

import torch.nn as nn


class TransformerNet(nn.Module):
    def __init__(self):
        super().__init__()

        # Each downsample layer: convolution -> normalization -> Relu activation
        self.down_sample = nn.Sequential(
            DownsampleConvLayer(3, 32, 9, 1),  # 32 × 9 × 9 conv, stride 1
            DownsampleConvLayer(32, 64, 3, 2),  # 64 × 3 × 3 conv, stride 2
            DownsampleConvLayer(64, 128, 3, 2)  # 128 × 3 × 3 conv, stride 2
        )

        # Each residual layer:  convolution -> normalization -> Relu activation
        #                       --> convolution -> normalization
        self.residual = nn.Sequential(
            ResidualBlock(128),
            ResidualBlock(128),
            ResidualBlock(128),
            ResidualBlock(128),
            ResidualBlock(128)
        )

        # Each upsample layer: interpolation -> convolution -> normalization -> Relu activation
        self.up_sample = nn.Sequential(
            UpsampleConvLayer(128, 64, 3, 2),
            UpsampleConvLayer(64, 32, 3, 2),
            nn.Conv2d(32, 3, 9, 1, padding=9 // 2, padding_mode='reflect')
        )

    def forward(self, x):
        out = self.down_sample(x)
        out = self.residual(out)
        out = self.up_sample(out)
        return out


class DownsampleConvLayer(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size, stride, instance_norm=True):
        super(DownsampleConvLayer, self).__init__()

        """
        Output dimensions H_out and W_out (both are equal in our case) of a convolution:
            H_out = ⌊(H_in + 2 * padding - dilation * (kernel_size - 1) - 1 / stride) + 1⌋

        Thus, padding of ⌊kernel size / 2⌋ is necessary to maintain the output sizes
        described in the paper's architecture. We decided to keep the default padding
        type of "reflect" as mentioned in the supplementary material of the paper.
        """
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size, stride,
                                padding=kernel_size // 2, padding_mode='reflect')
        if instance_norm:
            self.norm = nn.InstanceNorm2d(out_channels, affine=True)
        else:
            self.norm = nn.BatchNorm2d(out_channels)

        self.relu = nn.ReLU()

    def forward(self, x):
        return self.relu(self.norm(self.conv2d(x)))


class ResidualBlock(nn.Module):
    def __init__(self, channels, instance_norm=True):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(channels, channels, kernel_size=3, padding=3 // 2, padding_mode='reflect')
        if instance_norm:
            self.norm1 = nn.InstanceNorm2d(channels, affine=True)
            self.norm2 = nn.InstanceNorm2d(channels, affine=True)
        else:
            self.norm1 = nn.BatchNorm2d(channels)
            self.norm2 = nn.BatchNorm2d(channels)
        self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, padding=3 // 2, padding_mode='reflect')
        self.relu = nn.ReLU()

    def forward(self, x):
        orig = x
        out = self.relu(self.norm1(self.conv1(x)))
        out = self.norm2(self.conv2(out))
        return out + orig


class UpsampleConvLayer(nn.Module):
    """
        Code is influenced from https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py
        Nearest-neighbor up-sampling followed by a convolution
        Appears to give better results than learned up-sampling aka transposed conv (avoids the checkerboard artifact)

        Initially proposed on distill pub: http://distill.pub/2016/deconv-checkerboard/
    """

    def __init__(self, in_channels, out_channels, kernel_size, upsampling_factor):
        super().__init__()
        self.upsampling_factor = upsampling_factor
        self.conv = DownsampleConvLayer(in_channels, out_channels, kernel_size, 1)

    def forward(self, x):
        if self.upsampling_factor > 1:
            x = nn.functional.interpolate(x, scale_factor=self.upsampling_factor, mode='nearest')
        return self.conv(x)
```


References:

- https://cs.stanford.edu/people/jcjohns/eccv16/. - Paper followed to create ML models
- https://github.com/pytorch/examples/tree/main/fast_neural_style/neural_style - Sample hyper-parameters for faster tuning and sample style images found here
- https://www.kaggle.com/datasets/awsaf49/coco-2017-dataset - Massive dataset for training images found here
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution - https://arxiv.org/pdf/1603.08155.pdf
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution: Supplementary Material - https://cs.stanford.edu/people/jcjohns/papers/fast-style/fast-style-supp.pdf
- Instance Normalization: The Missing Ingredient for Fast Stylization - https://arxiv.org/pdf/1607.08022.pdf
- Very Deep Convolutional Networks for large-Scale Image Recognition - https://arxiv.org/pdf/1409.1556.pdf
- Neural Network Intuitions - https://towardsdatascience.com/neural-networks-intuitions-2-dot-product-gram-matrix-and-neural-style-transfer-5d39653e7916
- Referenced code - https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py

Made with Squarespace

```
"""
Ref = https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py
"""
from collections import namedtuple

import torch.nn as nn
from torchvision import models


class Vgg16(nn.Module):
    def __init__(self, requires_grad=False):
        super(Vgg16, self).__init__()
        vgg_pretrained_features = models.vgg16(pretrained=True).features
        self.slice1 = nn.Sequential()
        self.slice2 = nn.Sequential()
        self.slice3 = nn.Sequential()
        self.slice4 = nn.Sequential()
        for x in range(4):  # relu1_2
            self.slice1.add_module(str(x), vgg_pretrained_features[x])
        for x in range(4, 9):  # relu2_2
            self.slice2.add_module(str(x), vgg_pretrained_features[x])
        for x in range(9, 16):  # relu3_3
            self.slice3.add_module(str(x), vgg_pretrained_features[x])
        for x in range(16, 23):  # relu4_3
            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(x)
        h_relu2_2 = h
        h = self.slice3(x)
        h_relu3_3 = h
        h = self.slice4(x)
        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
```

References:

- https://cs.stanford.edu/people/jcjohns/eccv16/. - Paper followed to create ML models
- https://github.com/pytorch/examples/tree/main/fast_neural_style/neural_style - Sample hyper-parameters for faster tuning and sample style images found here
- https://www.kaggle.com/datasets/awsaf49/coco-2017-dataset - Massive dataset for training images found here
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution - https://arxiv.org/pdf/1603.08155.pdf
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution: Supplementary Material - https://cs.stanford.edu/people/jcjohns/papers/fast-style/fast-style-supp.pdf
- Instance Normalization: The Missing Ingredient for Fast Stylization - https://arxiv.org/pdf/1607.08022.pdf
- Very Deep Convolutional Networks for large-Scale Image Recognition - https://arxiv.org/pdf/1409.1556.pdf
- Neural Network Intuitions - https://towardsdatascience.com/neural-networks-intuitions-2-dot-product-gram-matrix-and-neural-style-transfer-5d39653e7916
- Referenced code - https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py

Made with Squarespace

```
"""
Ref = https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py
"""

import torch
from torchvision import transforms

import utils
from transformer_net import TransformerNet

CONTENT_IMG_PATH = "trainmini/000000000072.jpg"
MODEL = "checkpoints/ckpt_epoch_0_batch_id_1150.pth"
OUTPUT_IMG_PATH = "outputs/trainimg18.jpg"


def stylize():
    device = torch.device("mps")

    content_image = utils.load_image(CONTENT_IMG_PATH)
    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)
    with torch.no_grad():
        style_model = TransformerNet()
        state_dict = torch.load(MODEL, map_location=device)
        style_model.load_state_dict(state_dict)
        style_model.to(device)
        style_model.eval()
        output = style_model(content_image).cpu()
    utils.save_image(OUTPUT_IMG_PATH, output[0])
    utils.save_image("outputs/van_img.jpg", content_image[0])


stylize()
```

References:

- https://cs.stanford.edu/people/jcjohns/eccv16/. - Paper followed to create ML models
- https://github.com/pytorch/examples/tree/main/fast_neural_style/neural_style - Sample hyper-parameters for faster tuning and sample style images found here
- https://www.kaggle.com/datasets/awsaf49/coco-2017-dataset - Massive dataset for training images found here
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution - https://arxiv.org/pdf/1603.08155.pdf
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution: Supplementary Material - https://cs.stanford.edu/people/jcjohns/papers/fast-style/fast-style-supp.pdf
- Instance Normalization: The Missing Ingredient for Fast Stylization - https://arxiv.org/pdf/1607.08022.pdf
- Very Deep Convolutional Networks for large-Scale Image Recognition - https://arxiv.org/pdf/1409.1556.pdf
- Neural Network Intuitions - https://towardsdatascience.com/neural-networks-intuitions-2-dot-product-gram-matrix-and-neural-style-transfer-5d39653e7916
- Referenced code - https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py

Made with Squarespace

CSE455 Final Project - Kevin Kumer & Varshan Muhunthan

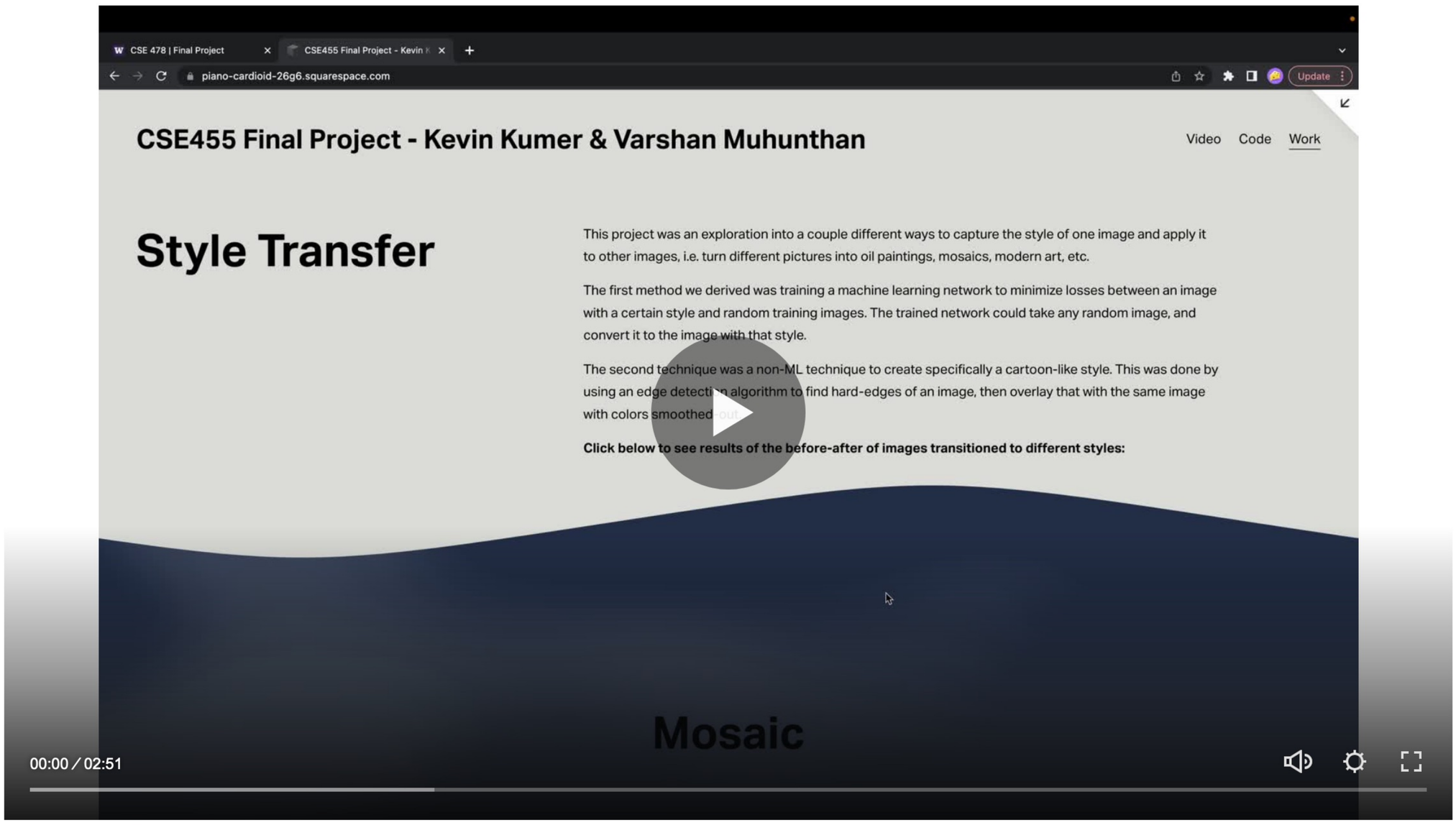

```
"""
Ref = https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py
"""

import torch
from torch.optim import Adam
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import transforms
import time
import os
import utils
from transformer_net import TransformerNet
from vgg import Vgg16

IMG_SIZE = 256
BATCH_SIZE = 4
LR = 1e-3
DATASET_PATH = "../style_transfer/code/images/train"
STYLE_IMG_PATH = "./style-images/tsunami.png"
EPOCHS = 2
CONTENT_WEIGHT = 1e5
STYLE_WEIGHT = 1e10
LOG_INTERVAL = 10
CHECKPOINT_MODEL_DIR = "./checkpoints"
CHECKPOINT_INTERVAL = 1000
SAVE_MODEL_DIR = "./models"


def train():
    device = torch.device("mps")

    transform = transforms.Compose([
        transforms.Resize(IMG_SIZE),
        transforms.CenterCrop(IMG_SIZE),
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.mul(255))
    ])
    train_dataset = datasets.ImageFolder(DATASET_PATH, transform)
    train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE)

    transformer = TransformerNet().to(device)
    optimizer = Adam(transformer.parameters(), lr=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(STYLE_IMG_PATH)
    style = style_transform(style)
    style = style.repeat(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(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 = 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 *= 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) % 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 CHECKPOINT_MODEL_DIR is not None and (batch_id + 1) % 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(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(EPOCHS) + "_" + str(time.ctime()).replace(' ', '_') + "_" + str(
        CONTENT_WEIGHT) + "_" + str(STYLE_WEIGHT) + ".model"
    save_model_path = os.path.join(SAVE_MODEL_DIR, save_model_filename)
    torch.save(transformer.state_dict(), save_model_path)

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


train()
```


References:

- https://cs.stanford.edu/people/jcjohns/eccv16/. - Paper followed to create ML models
- https://github.com/pytorch/examples/tree/main/fast_neural_style/neural_style - Sample hyper-parameters for faster tuning and sample style images found here
- https://www.kaggle.com/datasets/awsaf49/coco-2017-dataset - Massive dataset for training images found here
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution - https://arxiv.org/pdf/1603.08155.pdf
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution: Supplementary Material - https://cs.stanford.edu/people/jcjohns/papers/fast-style/fast-style-supp.pdf
- Instance Normalization: The Missing Ingredient for Fast Stylization - https://arxiv.org/pdf/1607.08022.pdf
- Very Deep Convolutional Networks for large-Scale Image Recognition - https://arxiv.org/pdf/1409.1556.pdf
- Neural Network Intuitions - https://towardsdatascience.com/neural-networks-intuitions-2-dot-product-gram-matrix-and-neural-style-transfer-5d39653e7916
- Referenced code - https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py

Made with Squarespace

Demonstration



References:

- https://cs.stanford.edu/people/jcjohns/eccv16/. - Paper followed to create ML models
- https://github.com/pytorch/examples/tree/main/fast_neural_style/neural_style - Sample hyper-parameters for faster tuning and sample style images found here
- https://www.kaggle.com/datasets/awsaf49/coco-2017-dataset - Massive dataset for training images found here
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution - https://arxiv.org/pdf/1603.08155.pdf
- Perceptual Losses for Real-Time Style Transfer and Super-Resolution: Supplementary Material - https://cs.stanford.edu/people/jcjohns/papers/fast-style/fast-style-supp.pdf
- Instance Normalization: The Missing Ingredient for Fast Stylization - https://arxiv.org/pdf/1607.08022.pdf
- Very Deep Convolutional Networks for large-Scale Image Recognition - https://arxiv.org/pdf/1409.1556.pdf
- Neural Network Intuitions - https://towardsdatascience.com/neural-networks-intuitions-2-dot-product-gram-matrix-and-neural-style-transfer-5d39653e7916
- Referenced code - https://github.com/pytorch/examples/blob/main/fast_neural_style/neural_style/utils.py

Made with Squarespace