Multimodal Transfer: A Hierarchical Deep Convolutional Neural Network for Fast Artistic Style Transfer

Assignment 1: Marc Schmid Student No 13349752

Github: https://github.com/marcimarc1/ADA_Assignments (https://github.com/marcimarc1/ADA_Assignments)

(Including models and pictures, but not the Coco Dataset)

Content

Introduction

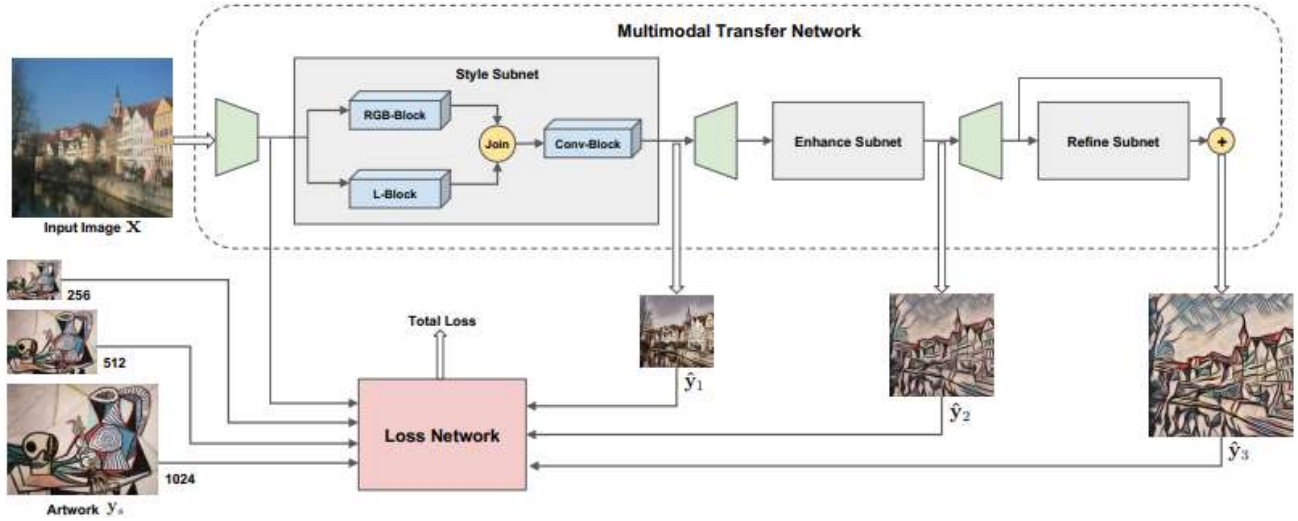
The paper is of the area of deep learning for computer vision. It investigates new generative neural network models to transfer artistic styles onto arbitrary images. There are already online learning algorithms, but they do not represent the style transfer correctly as they fail to display the style in local regions of the picture. The paper is mainly based on the work of L. A. Gatys, A. S. Ecker, and M. Bethge.

(a) Style (b) Gatys *et al.* (c) Johnson *et al.* (d) Ulyanov *et al.* (e) Ours (f) Content

Top row: (a) The style guide is "At the Close of Day" by Tomas King, and (f) is the content image. (b) is the result of Gatys et al's optimization-based method. (Result size is 512 due to memory limitation of the method) (c), (d) and (e) are results generated by different feed-forward networks (all are of size 1024). Bottom row: the zoom-in display of the regions enclosed in the red boxes from the top row. As can be seen, all results are repainted with the colour of the style image. However, a closer examination shows that the brush strokes are not captured well in (c) and (d). The zoom-in region in (b) is a little blurry. Comparing with the others, our multimodal transfer (e) is capable of simulating more closely the brushwork of the original artwork on high-resolution images.(From [1])

Image style transfer uses convolutional neural networks, where a pre-trained deep learning network for visual recognition is used to capture both style and content representations. They reduce the computing time of the neural network by training it offline and introducing a new hierarchical deep convolutional neural network architecture. This helps to fasten the computation of style transfers in software like Adobe Photoshop. Through the fully convolutional network it is guaranteed to process pictures of different sizes, however it has to have a minimal size, otherwise a 1 by 1 convolution will eventually crash somewhere in the neural net. They introduce a new structure in the first neural network they use, where they differentiate in colour scheme and illumination, as visual perception is far more sensitive to changes in illumination than in colour. They prove in experiments that they achieve better results for the style transfer than in traditional CNN-Architectures.

Network Architecture



The overall architecture of the neural network is structured into 3 subnets (MT Network). The Style Subnet, the Enhance Subnet and the Refine Subnet. This network architecture was chosen because its difficult to optimally adjust textures to a variety of styles. The network takes an image as input and generates an output image for each subnet with increasing resolution. These output images are then taken as inputs to the loss network to calculate a stylization loss for each image. Afterwards the total loss is a weighted combination of all stylization losses. The subnet prediction is defined as

$$X_k = f(\omega_k, x)$$

, where X_k is the prediction of the k-th network, ω_k are the parameters of the k-th network, x is the input image and $f()$ is the corresponding neural network architecture.

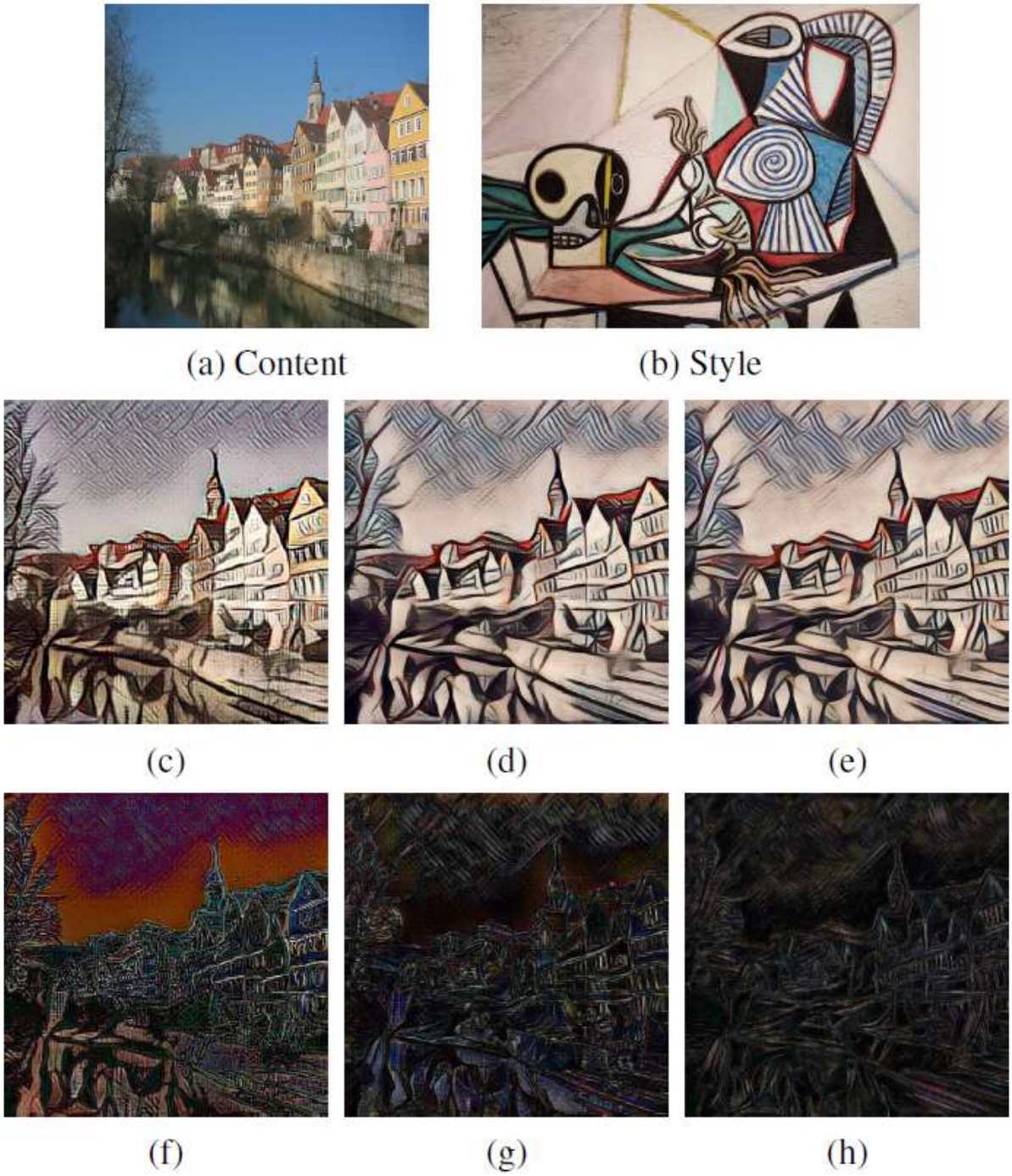
Style Subnet

As visual perception is far more sensitive to illuminance than colour, the Style Subnet addresses the three colour channels (in the RGB-Block) as well as the luminance channels (in the L-Block). The feature maps computed by those both nets are than concatenated in the depth dimension and further processed by the following Convolution Block. The goal of the Style Subnet is to preserve the content and to adjust the textures to the style a lot. The RGB and L Block consist of three convolutional layers followed by three instance normalization layers and ReLU activation functions respectively, followed by three residual blocks. After the concatenation of the two blocks there follows an up-sampling convolutional layer structure to reach the right size of the image.

Enhance Subnet and Refine Subnet

The job of the Enhance Subnet and Refine Subnet is to further change the style of the picture, but less than the Style Subnet. The Enhance Subnet as well as the Refine Subnet are fully convolutional neural networks which up-sample the images to the size of 512 and 1024 respectively. The Enhance Subnet consists of 4 convolutional layers, 6 residual blocks, followed by an up-sampling convolutional structure. The Refine Subnet is embodied similar with 4 convolutional layers, 3 residual blocks as well an up-sampling structure, so that the image matches the target.

Comparison of the Refinement



(a) is the content image and (b) is the style image. (c)(d)(e) show the outputs of the three subnets, $\hat{y}_1$, $\hat{y}_2$ and $\hat{y}_3$, whose sizes are 256, 512 and 1024 respectively. The third row depicts the absolute difference between: (f) the content image and the output image $\hat{y}_1$, (g) the output images $\hat{y}_1$ and $\hat{y}_2$, and (h) the output images $\hat{y}_2$ and $\hat{y}_3$.(From [1])

We can see that the total difference between the pictures is decreasing in each subnet. So that the Style Subnet changes the content picture a lot, while the enhance and refine subnets focus on detail adaption.

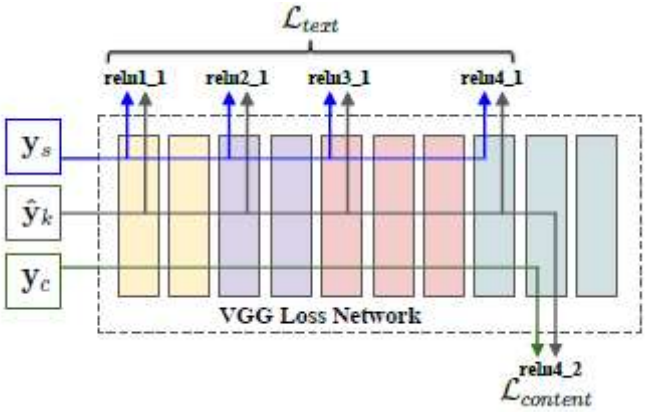
Mathematical Foundations and Learning Schemes

As explained in the network architecture, the multimodal CNN takes an imput image and returns 3 output images. These output images are the put into the loss network which calculates stylization losses. The loss network is the VGG16 (in the implementation I used the pretrained model from torchvision).

Excursion: VGG16-NN Structure and use as Stylization Loss Network

The VGG16 Network was actually build for 224x224x3 pictures as a classification task. It consists of twelve convolutional layers with max pooling layers inbetween which extend to 512 channels in the last convolution , before a few fully connected layers are used for classification. In the MT Network the VGG16-Loss is expanded to a Content Loss and Texture/Style Loss.

Loss Function



Content Loss

Let $\vec{p}$ and X be the original image and the generated image, and P^l and F^l their respective feature representation in layer l. The squared-error loss between the two feature representations is defined as

$$\mathcal{L}_{content}(\vec{p}, X, l) = \frac{1}{2} \sum_{i,j} (F^l_{i,j} - P^l_{i,j})^2$$

Thus, the content loss compares the feature difference for the \textit(l)-th layer of the convolution of the VGG16. It is a good idea to compute the content loss in one of the last layers of the VGG net, as in the first convolutional layers the standard features like vertical and horizontal lines are covered. The last convolution layers are more complex features get covered, so that the content really gets compared, and not some superficial features.

Style Loss

The Style loss consists of the correlations between the different filter responses, where the expectation is taken over the spatial extent of the feature maps. These feature correlations are given by the Gram matrix $G^l \in \mathcal{R}^{N_l \times N_l}$, where G^l_{ij} is the inner product between the vectorised feature maps i and j in layer l:

$$G^l_{i,j} = \sum_k F^l_{ik} F^l_{j,k}$$

which equals:

$$G(x_1, \ldots, x_n) = \begin{pmatrix} \langle x_1, x_1 \rangle & \langle x_1, x_2 \rangle & \ldots & \langle x_1, x_n \rangle \\ \langle x_2, x_1 \rangle & \langle x_2, x_2 \rangle & \ldots & \langle x_2, x_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle x_n, x_1 \rangle & \langle x_n, x_2 \rangle & \ldots & \langle x_n, x_n \rangle \end{pmatrix}.$$

in matrix notation. By including the feature correlations of multiple layers, a stationary, multi-scale representation of the input image is obtained, which captures its texture information but not the global arrangement. Therefore the texture function loss is defined as

$$\mathcal{L}_{texture}(\vec{q}, X) = \sum_{l \in L} w_l (G^l(\vec{q}) - G^l(X))^2$$

with the amount of Layers chosen L, generated picture X and style target $\vec{q}$, and a chosen weight for each layer w_l.

Now it is up to the user to decide which weight of content and texture loss is

$$\mathcal{L}_{style}(\vec{p}, \vec{q}, X) = \alpha \mathcal{L}_{content}(\vec{p}, X, l) + \beta \mathcal{L}_{texture}(\vec{q}, X)$$

where α and β are the weights of the content loss and texture loss, respectively.

Hirarchical Stylization Loss Function

As the previously discussed part is nothing new, as it got already presented at [2] the MT Network provides 3 outputs X_k with $k \in [1, 2, 3]$ for the MT Network. Therefore multiple loss function get computed which have to be combined. Thus, the hierarchical stylization loss function $\mathcal{L}_H$ is introduced. The loss function is a sum of weighted loss function of the stylization losses:

$$\mathcal{L}_H = \sum_k \lambda_k \mathcal{L}_{style}(\vec{p}_k, \vec{q}_k, X_k)$$

where λ_k is the weight of the corresponding stylization loss.

To train the neural network, it is important to compute a gradient for the backward pass. For each subnet (k) feature space ω_k is trained to minimize the parallel weighted stylization losses that are computed from the outputs of the layer after the the layer of which the actuall loss is taken. Thus, they defined the features as

$$\omega_k = argmin_{\omega_k} E_x \left[\sum_{i \geq k} \lambda_i \mathcal{L}^k_{style}(f(\omega_k, x), \vec{p}, \vec{q}) \right]$$

The gradients get computed by

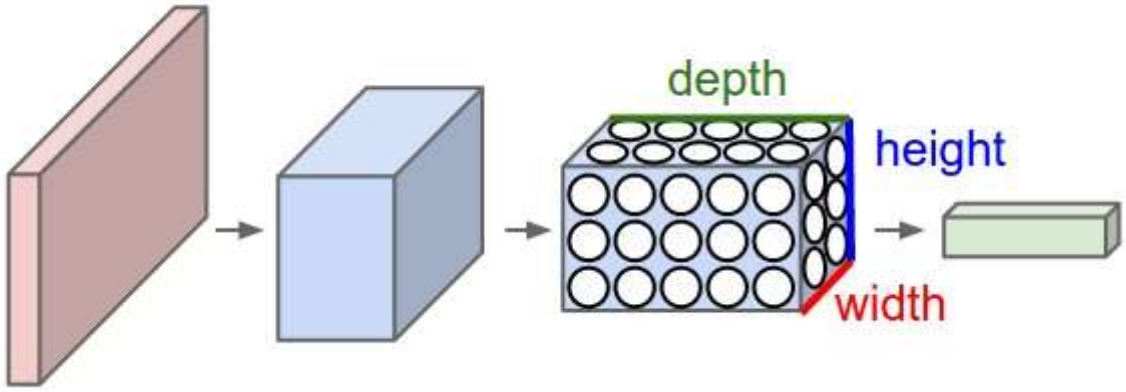
$$\delta\omega_k = \begin{cases} (lambda_k \mathcal{L}^k_{style})' & \text{if } k = K \\ (lambda_k \mathcal{L}^k_{style}, \delta\omega_{k+1})' & \text{if} 1 \leq k \leq K \end{cases}$$

F. Meissen told me to add a regularization loss like in [3] for better training results and to prevent overfitting, as I haven't trained on the full Coco dataset, just on 50% of the dataset.

Used Layers

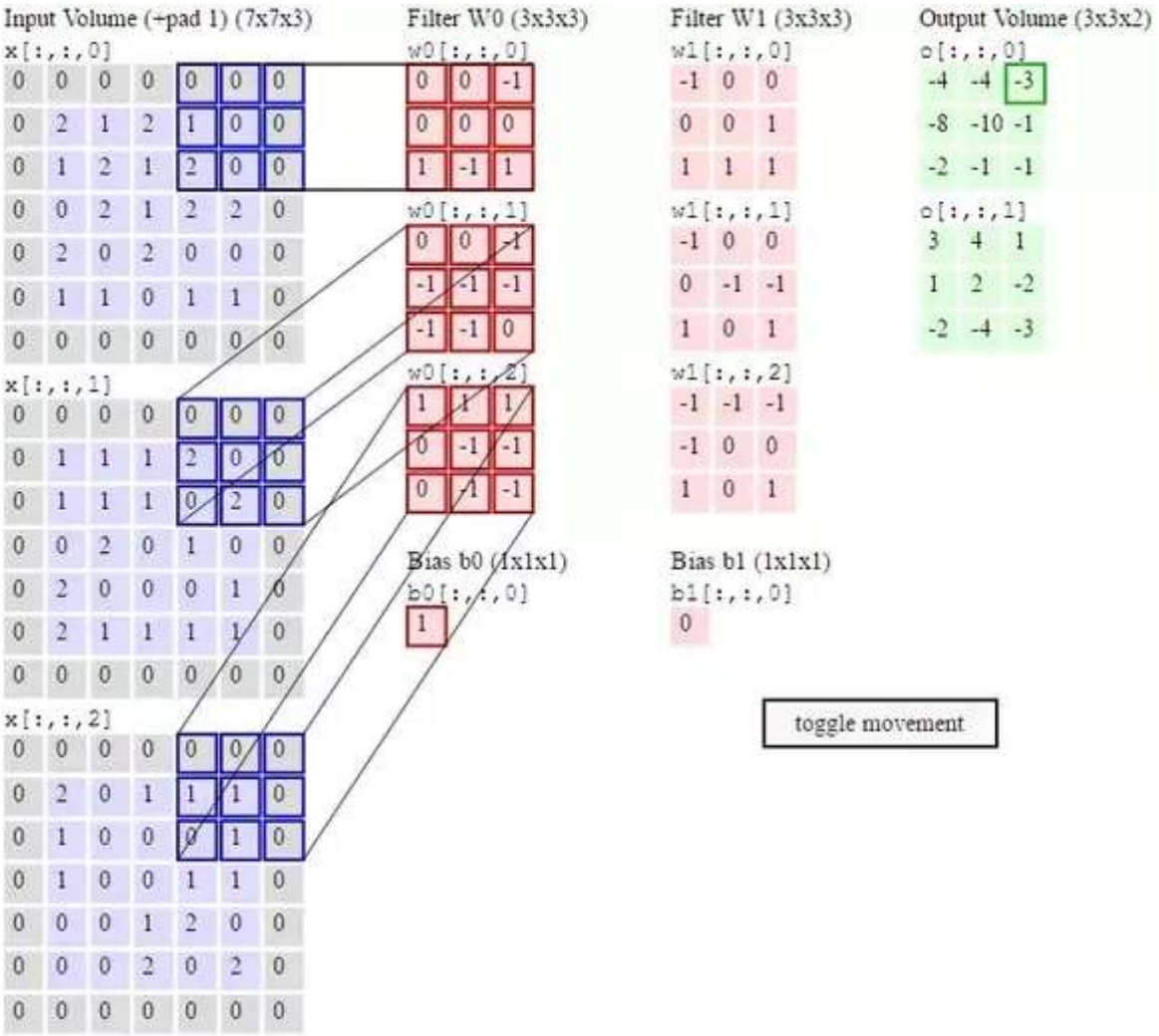
(Just a short mention as they were not explained in the lecture yet)

Convolutional Layer



In this figure we can see a downsampling structure of a convolution just by different filters and strides which decreases the width and height. The amount of filters increases the depth.

In convolutional layers we use so called filters to connect the neurons. The neurons are not all to all connected anymore, either they are dependent on their neighbours. In convolutions we use the fact that pixels next to each other are somehow connected to each other. Through weight sharing in convolutional layers



In this figure we can see how two 3x3 filters/kernels work on an input volume of 7x7x3 with padding 1. You swipe the kernels over the different depth levels. In general you start at the upper left and take one step to the right until you reach the other border. Then you slip one line down until you covered the whole picture.

Properties

Padding

The padding p puts some numbers around of the border of the picture. With it we can change the size of the output size. To keep the size it is good to use stride one and the formula for the padding is

$$p = \frac{k-1}{2}$$

with k as amount of pixels in one direction.

Spatial Extend

The spatial extend F is the receptive field size of the convolution layer neurons.

Stride

The stride s specifies how many pixels we move the filter in one step. Stride 1 is common. But to reduce size of the convolutional output, stride 2 is also used quit often.

Computation of the Output Size

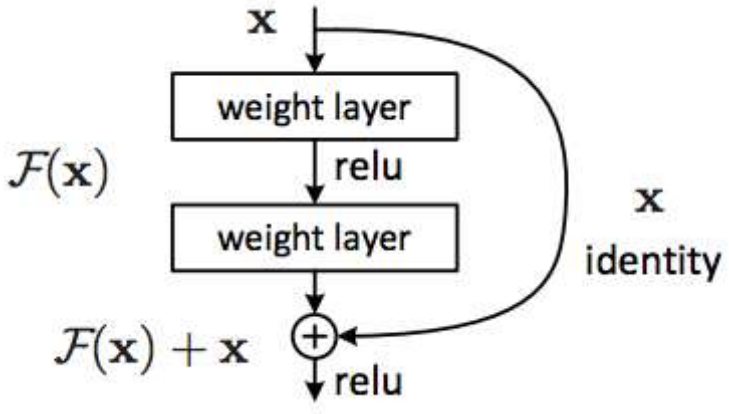
The output size of the new convolutional volume can be computed, if the stride, spatial extend, padding and the input width and hight are known:

$$W_2 = \frac{W_1 - F + 2P}{S} + 1$$
$$H_2 = \frac{H_1 - F + 2P}{S} + 1$$

The Depth is just the amount of kernels used in the layer.

Residual Block

Deep neural networks suffer from vanishing or exploding gradient as well as saturation of accuracy, dependant on the activation functions. Let us consider a shallower architecture and its deeper counterpart that adds more layers onto it. There exists a solution to the deeper model by construction: the layers are copied from the learned shallower model, and the added layers are identity mapping. The existence of this constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart.



Instead of hoping each stack of layers directly fits a desired underlying mapping, we explicitly let these layers fit a residual mapping. The original mapping is recast into $F(x) + x$. We hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers.

Instance Normalization

Normalization is used to keep the values of the input in zero mean, unit variance to improve the performance of neural networks. With sigmoid-like activation functions we tent to saturate quickly for large values in input, which contributes to the vanishing gradient problem. That's why the batch normalization was so successful. Instance normalization is basically batch normalization with the key difference that the latter applies the normalization to a whole batch of images instead of single ones. This prevents instance-specific mean and covariance shift simplifying the learning process, of which is different from batch normalization. Furthermore, the instance normalization layer is applied at test time as well.

Innovation

This paper introduces a new hierarchical deep convolutional neural network architecture for offline training. The results get compared to Gatys Paper and different feed-forward neural networks as well as different singular transfer networks. Those networks are other state-of-the-art networks by Johnson: 'Perceptual losses for real-time style transfer and super-resolution', and Ulyanov: 'Instance normalization: The missing ingredient for fast stylization'. Principally they introduce a new loss function, compared to Johnson, Gatsy and Ulyanov and concatenate three networks with the same structure but different depth.

Nevertheless, they introduce a very interesting comparison of a singular transfer and a multimodal transfer on high resolution images. The setup of the comparison is comprehensible, as they generated a singular transfer neural network with the same amount of weights as the multimodal transfer neural network. While the singular transfer model either had too much or too little style adaption, the multimodal transfer model has a good balance of both. Thus, as can be seen in the implementation, you are able to adapt how much style and content should be learned in every stage and it increases the accuracy of the wanted output.

Technical Quality

I would rate the technical quality of the paper good over all. It is well cited and you can find details easily in the references. Sometimes they mess around with words, as for beginners in neural networks a feed forward neural network does not need to be of convolutional structure. But as this is a more advanced research paper, most of the people reading it are familiar with those expressions. All the figures are well documented and described, which makes it easy to follow the text and the new thoughts. As the paper is highly linked to Johnson, Gatsy and Ulyanov, it is mandatory to read those papers to fully understand the neural network architecture. As they introduced their new Loss function, it was very hard to follow and understand what they meant by the stylization losses are computed from the latter outputs of the actual target. The function itself is self-explanatory, although it is a highly interleaved function with a lot of different indices, which makes it really hard to implement the new loss.

The experiments and comparisons at the end had always the same assumptions. As they introduce this new multimodal transfer network, it is hard to compare it to other multimodal networks, as they were non-existent at the time, so comparing it to the singular transfer network is the logical step. It would have been a good idea to create more than one multimodal network, as the algorithm is scalable and it's easy to introduce maybe 2-5 subnet models. Therefore a higher resolution of the pictures would have been possible and a more detailed scale of weights.

As they compare processing speed and memory usage, they showed that their network is highly advanced (in terms of processing speed and memory usage) to a singular transfer network with the same size, as it has a lower processing time and uses just half of the memory. They also compared it to the Johnson Net you can see that the Johnson net is more efficient in processing speed and memory usage then the MT Net.

Application and X-factor

The newly introduced Network enables artists and designers to adapt new styles to their work. As the style transfer networks need a lot of memory to safe them it would be good to provide them as a website or on a cloud for memory efficiency. Simple machine learning algorithms are not able to alter images in this way, so i think that the method is thoroughly appropriated for this topic. As this is a generative neural network model the scientific value is not outstanding to other introductions, but in combination with arts this paper delivers a high value of innovation (in general the whole three to four papers which developed this art an neural network combination). Through my research in implementation of this paper, they already implemented this method not just for image style transfer, but also for video style transfer. Other applications of this neural network architecture may be in the finance sector for stock market transfer predictions or even in production planning as this neural network may generate new plant designs or even optimize old plants by redesigning them (of course with different training sets and altering of the layer architecture). New research in generative models can be used for graphs. A research group from the TU Munich introduced the first implicit generative model for graphs able to mimic real-world networks in March 2018 (https://arxiv.org/abs/1803.00816) (https://arxiv.org/abs/1803.00816)). I am really looking forward to see more about generative models, as they are not just use supervised learning but also introduce new knowledge to mankind.

Presentation

The paper was easy to understand in general, but there were some difficulties regarding the details of the other papers. While they talked a lot about the loss functions which were already introduced in detail in Gatsy [2], it was hard to implement the instance normalization. As the Johnson [3] paper was really short, they should have picked up the topic again, as they used the instance normalization layer in every subnet. More detailed experiments would be great, but they actually explained their insights well.

References

[1] X. Wang, G. Oxholm, D. Zhang, Y. Wang. Multimodal Transfer: A Hierarchical Deep Convolutional Neural Network for Fast Artistic Style Transfer. Accepted by CVPR 2017

[2] L. A. Gatys, A. S. Ecker, and M. Bethge. Image style transfer using convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 2414–2423, 2016.

[3] J. Johnson, A. Alahi, and L. Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In European Conference on Computer Vision, pages 694–711. Springer, 2016.

[4] CS231n: Convolutional Neural Networks for Visual Recognition, Stanford 2018

[5] Github Implementation Help: https://github.com/ceshine/fast-neural-style (https://github.com/ceshine/fast-neural-style)

[6] D. Ulyanov, A. Vedaldi, and V. Lempitsky. Instance normalization: The missing ingredient for fast stylization. arXiv preprint arXiv:1607.08022, 2016.

Implementation

For understanding of the topic I like to implement the code. I commented the code, if I used existing implementations. Some code is based on the fast neural style transfer implementation by CeShine Lee and Felix Meissen [https://github.com/ceshine/fast-neural-style (https://github.com/ceshine/fast-neural-style)]

In [ ]:

```
# Imports
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import numpy as np

from PIL import Image
import matplotlib.pyplot as plt

from collections import namedtuple
import time
import os
import sys
import copy
import os
import gc
```

First constructed different neural network modules to keep the code readable. The references are commented in the code.

In [ ]:

```
class ConvLayer(nn.Module):
    """ ConvBlock
    simple convolution with reflection padding
    """
    def __init__(self, in_channels, out_channels, kernel_size, stride):
        super(ConvLayer, self).__init__()
        reflection_padding = int(np.floor(kernel_size / 2))
        self.reflection_pad = nn.ReflectionPad2d(reflection_padding)
        self.conv2d = 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(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 = nn.InstanceNorm2d(channels, affine=True)
        self.conv2 = ConvLayer(channels, channels, kernel_size=3, stride=1)
        self.in2 = nn.InstanceNorm2d(channels, affine=True)
        self.relu = 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 ResizeConvLayer(nn.Module):
    """ ResizeConvLayer
    upsampling with Nearest neighbor interpolation and a conv layer
    to avoid checkerboard artifacts.
    ref: https://distill.pub/2016/deconv-checkerboard/
    """

    def __init__(self, in_channels, out_channels, kernel_size, stride, scale_factor=2):
        super(ResizeConvLayer, self).__init__()
        reflection_padding = int(np.floor(kernel_size / 2))
        self.reflection_pad = nn.ReflectionPad2d(reflection_padding)
        self.nearest_neighbor = nn.Upsample(scale_factor=scale_factor, mode='nearest')
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size, stride)

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

```
class InstanceNormalization(nn.Module):
    """InstanceNormalization
    Improves convergence of neural-style.
    ref: https://arxiv.org/pdf/1607.08022.pdf
    """

    def __init__(self, dim, eps=1e-9):
        super(InstanceNormalization, self).__init__()
        self.scale = nn.Parameter(torch.FloatTensor(dim))
        self.shift = nn.Parameter(torch.FloatTensor(dim))
        self.eps = eps
        self._reset_parameters()

    def _reset_parameters(self):
        self.scale.data.uniform_()
        self.shift.data.zero_()

    def forward(self, x):
        n = x.size(2) * x.size(3)
        t = x.view(x.size(0), x.size(1), n)
        mean = torch.mean(t, 2).unsqueeze(2).unsqueeze(3).expand_as(x)
        # Calculate the biased var. torch.var returns unbiased var
        var = torch.var(t, 2).unsqueeze(2).unsqueeze(3).expand_as(x) * ((n - 1) / float(n))
        scale_broadcast = self.scale.unsqueeze(1).unsqueeze(1).unsqueeze(0)
        scale_broadcast = scale_broadcast.expand_as(x)
        shift_broadcast = self.shift.unsqueeze(1).unsqueeze(1).unsqueeze(0)
        shift_broadcast = shift_broadcast.expand_as(x)
        out = (x - mean) / torch.sqrt(var + self.eps)
        out = out * scale_broadcast + shift_broadcast
        return out
```

For the first neural subnet (Style Subnet) we need the rgb-, the L- and the Conv-Block. The rgb-Block adresses the utilization of color, while the L-Block adresses the luminance, as the visual perception is much more sensitive to change in luminance than color. The features of the different blocks are the joined together along the depth dimension to be further computed by the Conv-Block. The structure of the net is defined in the initialization of the style subnet

In [ ]:

In [ ]:

```
class StyleSubnet(nn.Module):
    def __init__(self):
        super(StyleSubnet, self).__init__()
        # Transform to Grayscale
        self.togray = nn.Conv2d(3, 1, kernel_size=1, stride=1)
        w = torch.nn.Parameter(torch.tensor([[[[0.299]],
                                              [[0.587]],
                                              [[0.114]]]]))
        self.togray.weight = w

        # RGB Block
        self.rgb_conv1 = ConvLayer(3, 16, kernel_size=9, stride=1)
        self.rgb_in1 = InstanceNormalization(16)
        self.rgb_conv2 = ConvLayer(16, 32, kernel_size=3, stride=2)
        self.rgb_in2 = InstanceNormalization(32)
        self.rgb_conv3 = ConvLayer(32, 64, kernel_size=3, stride=2)
        self.rgb_in3 = InstanceNormalization(64)
        self.rgb_res1 = ResidualBlock(64)
        self.rgb_res2 = ResidualBlock(64)
        self.rgb_res3 = ResidualBlock(64)

        # L Block
        self.l_conv1 = ConvLayer(1, 16, kernel_size=9, stride=1)
        self.l_in1 = InstanceNormalization(16)
        self.l_conv2 = ConvLayer(16, 32, kernel_size=3, stride=2)
        self.l_in2 = InstanceNormalization(32)
        self.l_conv3 = ConvLayer(32, 64, kernel_size=3, stride=2)
        self.l_in3 = InstanceNormalization(64)
        self.l_res1 = ResidualBlock(64)
        self.l_res2 = ResidualBlock(64)
        self.l_res3 = ResidualBlock(64)

        # Residual layers
        self.res4 = ResidualBlock(128)
        self.res5 = ResidualBlock(128)
        self.res6 = ResidualBlock(128)

        # Upsampling Layers
        self.rezconv1 = ResizeConvLayer(128, 64, kernel_size=3, stride=1)
        self.in4 = InstanceNormalization(64)
        self.rezconv2 = ResizeConvLayer(64, 32, kernel_size=3, stride=1)
        self.in5 = InstanceNormalization(32)
        self.rezconv3 = ConvLayer(32, 3, kernel_size=3, stride=1)

        # Non-linearities
        self.relu = nn.ReLU()


    def forward(self, x):
        # Resized input image is the content target
        resized_input_img = x.clone()

        # Get RGB and L image
        x_rgb = x
        with torch.no_grad(): x_l = self.togray(x.clone())

        # RGB Block
        y_rgb = self.relu(self.rgb_in1(self.rgb_conv1(x_rgb)))
        y_rgb = self.relu(self.rgb_in2(self.rgb_conv2(y_rgb)))
```

```
        y_rgb = self.relu(self.rgb_in3(self.rgb_conv3(y_rgb)))
        y_rgb = self.rgb_res1(y_rgb)
        y_rgb = self.rgb_res2(y_rgb)
        y_rgb = self.rgb_res3(y_rgb)

        # L Block
        y_l = self.relu(self.l_in1(self.l_conv1(x_l)))
        y_l = self.relu(self.l_in2(self.l_conv2(y_l)))
        y_l = self.relu(self.l_in3(self.l_conv3(y_l)))
        y_l = self.l_res1(y_l)
        y_l = self.l_res2(y_l)
        y_l = self.l_res3(y_l)

        # Concatenate blocks along the depth dimension
        y = torch.cat((y_rgb, y_l), 1)

        # Residuals
        y = self.res4(y)
        y = self.res5(y)
        y = self.res6(y)

        # Decoding
        y = self.relu(self.in4(self.rezconv1(y)))
        y = self.relu(self.in5(self.rezconv2(y)))
        y = self.rezconv3(y)


        # Clamp image to be in range [0,1] after denormalization
        y[0][0].clamp_((0-0.485)/0.299, (1-0.485)/0.299)
        y[0][1].clamp_((0-0.456)/0.224, (1-0.456)/0.224)
        y[0][2].clamp_((0-0.406)/0.225, (1-0.406)/0.225)

        return y, resized_input_img
```

The Style Subnet is intended to stylize the input image, but it is hard for the Style Subnet to keep the textures and content of the original image and adjust the image properly to the new style. Therefore, two more nets were introduced with high weights on the style and content respectively to further enhance the stylization. The Enhance Subnet and the Refine Subnet at further details to the image, while doing less in absolute difference.

In [ ]:

In [ ]:

```
class EnhanceSubnet(nn.Module):
    def __init__(self):
        super(EnhanceSubnet, self).__init__()

        self.upsample = nn.Upsample(scale_factor=2, mode='bilinear')

        # Initial convolution layers
        self.conv1 = ConvLayer(3, 32, kernel_size=9, stride=1)   # size = 512
        self.in1 = nn.InstanceNorm2d(32, affine=True)
        self.conv2 = ConvLayer(32, 64, kernel_size=3, stride=2)   # size = 256
        self.in2 = nn.InstanceNorm2d(64, affine=True)
        self.conv3 = ConvLayer(64, 128, kernel_size=3, stride=2)   # size = 128
        self.in3 = nn.InstanceNorm2d(128, affine=True)
        self.conv4 = ConvLayer(128, 256, kernel_size=3, stride=2)   # size = 64
        self.in4 = nn.InstanceNorm2d(256, affine=True)

        # Residual layers
        self.res1 = ResidualBlock(256)
        self.res2 = ResidualBlock(256)
        self.res3 = ResidualBlock(256)
        self.res4 = ResidualBlock(256)
        self.res5 = ResidualBlock(256)
        self.res6 = ResidualBlock(256)

        # Upsampling Layers
        self.rezconv1 = ResizeConvLayer(256, 128, kernel_size=3, stride=1)
        self.in5 = nn.InstanceNorm2d(128, affine=True)
        self.rezconv2 = ResizeConvLayer(128, 64, kernel_size=3, stride=1)
        self.in6 = nn.InstanceNorm2d(64, affine=True)
        self.rezconv3 = ResizeConvLayer(64, 32, kernel_size=3, stride=1)
        self.in7 = nn.InstanceNorm2d(32, affine=True)
        self.rezconv4 = ConvLayer(32, 3, kernel_size=9, stride=1)

        # Non-linearities
        self.relu = nn.ReLU()

    def forward(self, X):
        X = self.upsample(X)
        # resized input image is the content target
        resized_input_img = X.clone()

        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.relu(self.in4(self.conv4(y)))
        y = self.res1(y)
        y = self.res2(y)
        y = self.res3(y)
        y = self.res4(y)
        y = self.res5(y)
        y = self.res6(y)
        y = self.relu(self.in5(self.rezconv1(y)))
        y = self.relu(self.in6(self.rezconv2(y)))
        y = self.relu(self.in7(self.rezconv3(y)))
        y = self.rezconv4(y)

        # Clamp image to be in range [0,1] after denormalization
        y[0][0].clamp_((0-0.485)/0.299, (1-0.485)/0.299)
        y[0][1].clamp_((0-0.456)/0.224, (1-0.456)/0.224)
```

```
        y[0][2].clamp_((0-0.406)/0.225, (1-0.406)/0.225)

        return y, resized_input_img
```

In [ ]:

```

```

In [ ]:

```
class RefineSubnet(nn.Module):
    def __init__(self):
        super(RefineSubnet, self).__init__()

        self.upsample = nn.Upsample(scale_factor=2, mode='bilinear')

        # Initial convolution layers
        self.conv1 = ConvLayer(3, 32, kernel_size=9, stride=1)
        self.in1 = nn.InstanceNorm2d(32, affine=True)
        self.conv2 = ConvLayer(32, 64, kernel_size=3, stride=2)
        self.in2 = nn.InstanceNorm2d(64, affine=True)
        self.conv3 = ConvLayer(64, 128, kernel_size=3, stride=2)
        self.in3 = nn.InstanceNorm2d(128, affine=True)

        # Residual layers
        self.res1 = ResidualBlock(128)
        self.res2 = ResidualBlock(128)
        self.res3 = ResidualBlock(128)

        # Upsampling Layers
        self.rezconv1 = ResizeConvLayer(128, 64, kernel_size=3, stride=1)
        self.in4 = nn.InstanceNorm2d(64, affine=True)
        self.rezconv2 = ResizeConvLayer(64, 32, kernel_size=3, stride=1)
        self.in5 = nn.InstanceNorm2d(32, affine=True)
        self.rezconv3 = ConvLayer(32, 3, kernel_size=3, stride=1)

        # Non-linearities
        self.relu = nn.ReLU()

    def forward(self, X):
        in_X = X
        # resized input image is the content target
        resized_input_img = in_X.clone()

        y = self.relu(self.in1(self.conv1(in_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.relu(self.in4(self.rezconv1(y)))
        y = self.relu(self.in5(self.rezconv2(y)))
        y = self.rezconv3(y)
        y = y + resized_input_img

        # Clamp image to be in range [0,1] after denormalization
        y[0][0].clamp_((0-0.485)/0.299, (1-0.485)/0.299)
        y[0][1].clamp_((0-0.456)/0.224, (1-0.456)/0.224)
        y[0][2].clamp_((0-0.406)/0.225, (1-0.406)/0.225)

        return y, resized_input_img
```

For readable code I add some common utility function of pytorch, those are not made by me, they are copied from the Lecture Notes of 'Introduction to Deep Learning' from Technical University of Munich. I can't reference them properly as they are a part of the assignment of the course. There is no solution on github, but it is quit similar to Stanfords CS231 course, for which there are plenty of solutions on github.

In [ ]:

```
""" Transform tensor back to frame """
def recover_frame(frame):
    frame = frame.cpu().squeeze(0)
    denormalizer = tensor_denormalizer()
    frame = denormalizer(frame)
    frame.data.clamp_(0, 1)
    toPIL = transforms.Compose([transforms.ToPILImage(), transforms.Resize((540, 304))])
    frame = toPIL(frame)
    return frame


""" Image loader, loads image from file using PIL and converts it to torch tensor """
def image_loader(image_name, size=512):
    image = Image.open(image_name).convert('RGB')
    loader = transforms.Compose([transforms.Resize(size),
                                 transforms.ToTensor(),
                                 tensor_normalizer()])
    image = loader(image).unsqueeze(0)   # If only one image, add a fake dimension in front
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    return image.to(device, torch.float)


""" Image loader for the style image, returns 3 versions with resolutions in sizes_list """
def style_loader(image_name, device, sizes_list):
    image = Image.open(image_name).convert('RGB')
    out = []
    for size in sizes_list:
        loader = transforms.Compose([transforms.Resize(size),
                                     transforms.CenterCrop(size),
                                     transforms.ToTensor(),
                                     tensor_normalizer()])
        style_img = loader(image).unsqueeze(0)
        out.append(style_img.to(device, torch.float))
    return out


""" Imshow, displays image using matplotlib """
def imshow(tensor, title=None):
    image = tensor.cpu().clone()  # clone the tensor to not do changes on it
    image = image.squeeze(0)      # remove the fake batch dimension
    denormalizer = tensor_denormalizer()
    image = denormalizer(image)
    image.data.clamp_(0, 1)
    toPIL = transforms.ToPILImage()
    image = toPIL(image)
    plt.imshow(image)
    if title is not None:
        plt.title(title)
    plt.pause(0.001) # pause a bit so that plots are updated


""" Saves image in the /output folder with a specified name as .jpg """
def save_image(tensor, title="output"):
    image = tensor.cpu().clone()  # clone the tensor to not do changes on it
    image = image.squeeze(0)      # remove the fake batch dimension
    denormalizer = tensor_denormalizer()
    image = denormalizer(image)
    image.data.clamp_(0, 1)
    toPIL = transforms.ToPILImage()
```

```
    image = toPIL(image)
    scriptDir = os.path.dirname(__file__)
    image.save("{}.jpg".format(title))


""" Returns the gram matrix of a feature map """
def gram_matrix(input):
    b, ch, h, w = input.size()

    features = input.view(b, ch, h * w)  # change input to vectorized feature map K x N
    features_t = features.transpose(1, 2)

    # the gram matrix needs to be normalized because otherwise the early layers with a bigg
    # will result in higher values of the gram matrix.
    gram = features.bmm(features_t) / (ch * h * w)  # compute the gram matrix bmm = batch m

    return gram


""" Transforms to normalize the image while transforming it to a torch tensor """
def tensor_normalizer():
    return transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225])


""" Denormalizes image to save or display it """
def tensor_denormalizer():
    return transforms.Compose([transforms.Normalize(mean = [ 0., 0., 0. ], std = [ 1/0.229,
                               transforms.Normalize(mean = [ -0.485, -0.456, -0.406 ], std
```

Now we need to train the neural network. Therefore,

In [ ]:

```
#train imports
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import torchvision.models as models
from torch.utils.data import DataLoader
from torchvision import datasets

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

First set the variables for the style transfer

In [ ]:

```
IMAGE_SIZE = 256
BATCH_SIZE = 1
STYLE_NAME = "picasso"
LR = 1e-3
NUM_EPOCHS = 1
CONTENT_WEIGHTS = [1, 1, 1]
STYLE_WEIGHTS = [2e4, 1e5, 1e3] # Checkpoint single style
#STYLE_WEIGHTS = [5e4, 8e4, 3e4] # Checkpoint two styles
LAMBDAS = [1., 0.5, 0.25]
REG = 1e-7
LOG_INTERVAL = 400
```

In [ ]:

```
""" Allow PIL to read truncated blocks when loading images """

from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

In [ ]:

```
""" Add a seed to have reproducable results """

SEED = 1080
torch.manual_seed(SEED)
```

In [ ]:

```
""" Configure training with or without cuda """

# if torch.cuda.is_available():
#     device = torch.device("cuda")
#     torch.cuda.manual_seed(SEED)
#     torch.set_default_tensor_type('torch.cuda.FloatTensor')
#     kwargs = {'num_workers': 4, 'pin_memory': True}
# else:
device = torch.device("cpu")
torch.set_default_tensor_type('torch.FloatTensor')
kwargs = {}
```

Load the Coco - Dataset. Download Link-> http://cocodataset.org/#download (http://cocodataset.org/#download)

In [ ]:

```
""" Load coco dataset """

print("Loading dataset..")
DATASET = 'D:/train2014' # <- Change this line to the path of your coco dataset
transform = transforms.Compose([transforms.Resize(IMAGE_SIZE),
                                transforms.CenterCrop(IMAGE_SIZE),
                                transforms.ToTensor(), tensor_normalizer()])
# http://pytorch.org/docs/master/torchvision/datasets.html#imagefolder
train_dataset = datasets.ImageFolder(DATASET, transform)
# http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=False, **kwargs)
print("...Loaded")
```

In [ ]:

```
""" Load Style Image """

style_img_256, style_img_512, style_img_1024 = style_loader(
    "styles/" + STYLE_NAME + ".jpg", device, [256, 512, 1024])
```

In [ ]:

```
imshow(style_img_256)
```

In [ ]:

```
""" Define Loss Network """

StyleOutput = namedtuple("StyleOutput", ["relu1_1", "relu2_1", "relu3_1", "relu4_1"])
ContentOutput = namedtuple("ContentOutput", ["relu2_1"])

# https://discuss.pytorch.org/t/how-to-extract-features-of-an-image-from-a-trained-model/11
class LossNetwork(torch.nn.Module):
    def __init__(self, vgg):
        super(LossNetwork, self).__init__()
        self.vgg = vgg
        self.layer_name_mapping = {
            '1': "relu1_1", '3': "relu1_2",
            '6': "relu2_1", '8': "relu2_2",
            '11': "relu3_1", '13': "relu3_2", '15': "relu3_3", '17': "relu3_4",
            '20': "relu4_1", '22': "relu4_2", '24': "relu4_3", '26': "relu4_4",
            '29': "relu5_1", '31': "relu5_2", '33': "relu5_3", '35': "relu5_4"
        }

    def forward(self, x, mode='style'):
        #return of multiple outputs as dict
        if mode == 'style':
            layers = ['1', '6', '11', '20']
        elif mode == 'content':
            layers = ['6']
        else:
            print("Invalid mode. Select between 'style' and 'content'")
        output = {}
        for name, module in self.vgg._modules.items():
            x = module(x)
            if name in layers:
                output[self.layer_name_mapping[name]] = x
        if mode == 'style':
            return StyleOutput(**output)
        else:
            return ContentOutput(**output)
```

In [ ]:

```
""" Load and extract features from VGG16 """

print("Loading VGG..")
vgg = models.vgg19(pretrained=True).features.to(device).eval()
loss_network = LossNetwork(vgg).to(device).eval()
del vgg
```

In [ ]:

```
""" Before training, compute the features of every resolution of the style image """

print("Computing style features..")
with torch.no_grad():
    style_loss_features_256 = loss_network(Variable(style_img_256), 'style')
    style_loss_features_512 = loss_network(Variable(style_img_512), 'style')
    style_loss_features_1024 = loss_network(Variable(style_img_1024), 'style')
gram_style_256 = [Variable(gram_matrix(y).data, requires_grad=False) for y in style_loss_fe
gram_style_512 = [Variable(gram_matrix(y).data, requires_grad=False) for y in style_loss_fe
gram_style_1024 = [Variable(gram_matrix(y).data, requires_grad=False) for y in style_loss_f
```

In [ ]:

```
""" Init Net and loss """

style_subnet = StyleSubnet().to(device)
enhance_subnet = EnhanceSubnet().to(device)
refine_subnet = RefineSubnet().to(device)
```

In [ ]:

```
""" Prepare Training """

max_iterations = min(10000, len(train_dataset))

# init loss
mse_loss = torch.nn.MSELoss()
# init optimizer
optimizer = torch.optim.Adam(list(style_subnet.parameters()) +
                             list(enhance_subnet.parameters()) +
                             list(refine_subnet.parameters()), lr=LR)

def getLosses(generated_img, resized_input_img, content_weight, style_weight, mse_loss, gra

    # Compute features
    generated_style_features = loss_network(generated_img, 'style')
    generated_content_features = loss_network(generated_img, 'content')
    target_content_features = loss_network(resized_input_img, 'content')

    # Content loss
    target_content_features = Variable(target_content_features[0].data, requires_grad=False
    content_loss = content_weight * mse_loss(generated_content_features[0], target_content_

    # Style loss
    style_loss = 0.
    for m in range(len(generated_style_features)):
        gram_s = gram_style[m]
        gram_y = gram_matrix(generated_style_features[m])
        style_loss += style_weight * mse_loss(gram_y, gram_s.expand_as(gram_y))

    # Regularization loss
    reg_loss = REG * (
        torch.sum(torch.abs(generated_img[:, :, :, :-1] - generated_img[:, :, :, 1:])) +
        torch.sum(torch.abs(generated_img[:, :, :-1, :] - generated_img[:, :, 1:, :])))

    return content_loss, style_loss, reg_loss
```

In [ ]:

```
""" Perform Training """

style_subnet.train()
enhance_subnet.train()
refine_subnet.train()
start = time.time()
print("Start training on {}...".format(device))
for epoch in range(NUM_EPOCHS):
    agg_content_loss, agg_style_loss, agg_reg_loss = 0., 0., 0.
    log_counter = 0
    for i, (x, _) in enumerate(train_loader):


        # update learning rate every 2000 iterations
        if i % 2000 == 0 and i != 0:
            LR = LR * 0.8
            optimizer = torch.optim.Adam(list(style_subnet.parameters()) +
                                         list(enhance_subnet.parameters()) +
                                         list(refine_subnet.parameters()), lr=LR)


        optimizer.zero_grad()
        x_in = x.clone()

        """ Style Subnet """
        x_in = Variable(x_in).to(device)

        # Generate image
        generated_img_256, resized_input_img_256 = style_subnet(x_in)
        resized_input_img_256 = Variable(resized_input_img_256.data)

        # Compute Losses
        style_subnet_content_loss, style_subnet_style_loss, style_subnet_reg_loss = getLoss
            generated_img_256,
            resized_input_img_256,
            CONTENT_WEIGHTS[0],
            STYLE_WEIGHTS[0],
            mse_loss, gram_style_256)

        """ Enhance Subnet """
        x_in = Variable(generated_img_256)

        # Generate image
        generated_img_512, resized_input_img_512 = enhance_subnet(x_in)
        resized_input_img_512 = Variable(resized_input_img_512.data)

        # Compute Losses
        enhance_subnet_content_loss, enhance_subnet_style_loss, enhance_subnet_reg_loss = g
            generated_img_512,
            resized_input_img_512,
            CONTENT_WEIGHTS[1],
            STYLE_WEIGHTS[1],
            mse_loss, gram_style_512)

        """ Refine Subnet """
        x_in = Variable(generated_img_512)

        # Generate image
        generated_img_1024, resized_input_img_1024 = refine_subnet(x_in)
```

```
        resized_input_img_1024 = Variable(resized_input_img_1024.data)

        # Compute Losses
        refine_subnet_content_loss, refine_subnet_style_loss, refine_subnet_reg_loss = getL
            generated_img_1024,
            resized_input_img_1024,
            CONTENT_WEIGHTS[2],
            STYLE_WEIGHTS[2],
            mse_loss, gram_style_1024)

        # Total Loss
        total_loss = LAMBDAS[0] * (style_subnet_content_loss + style_subnet_style_loss + st
                     LAMBDAS[1] * (enhance_subnet_content_loss + enhance_subnet_style_loss
                     LAMBDAS[2] * (refine_subnet_content_loss + refine_subnet_style_loss +
        total_loss.backward()
        optimizer.step()

        # Aggregated Loss
        agg_content_loss += style_subnet_content_loss.data[0] + \
                            enhance_subnet_content_loss.data[0] + \
                            refine_subnet_content_loss.data[0]
        agg_style_loss += style_subnet_style_loss.data[0] + \
                          enhance_subnet_style_loss.data[0] + \
                          refine_subnet_style_loss.data[0]

        agg_reg_loss += style_subnet_reg_loss.data[0] + \
                        enhance_subnet_reg_loss.data[0] + \
                        refine_subnet_reg_loss.data[0]

        # Log training process
        if (i + 1) % LOG_INTERVAL == 0:
            log_counter += 1
            hlp = log_counter * LOG_INTERVAL
            time_per_pass = (time.time() - start) / hlp
            estimated_time_left = (time_per_pass * (max_iterations - i))/3600
            print("{} [{}/{}] time per pass: {:.2f}s  total time: {:.2f}s  estimated time l
                        time.ctime(), i+1, max_iterations,
                        (time.time() - start) / hlp,
                        time.time() - start,
                        estimated_time_left,
                        agg_content_loss / LOG_INTERVAL,
                        agg_style_loss / LOG_INTERVAL,
                        agg_reg_loss / LOG_INTERVAL,
                        (agg_content_loss + agg_style_loss + agg_reg_loss) / LOG_INTERVAL))
            agg_content_loss, agg_style_loss, agg_reg_loss = 0., 0., 0.
            imshow(x, title="input image")
            imshow(generated_img_256, title="generated_img_256")
            imshow(generated_img_512, title="generated_img_512")
            imshow(generated_img_1024, title="generated_img_1024")

        # Stop training after max iterations
        if (i + 1) == max_iterations: break

""" Save model """
torch.save(style_subnet, 'models/style_subnet_picasso.pt')
torch.save(enhance_subnet, 'models/enhance_subnet_picasso.pt')
torch.save(refine_subnet, 'models/refine_subnet_picasso.pt')
```

Pull the git repository and just load the models, its more effective than training it for hours.

In [ ]:

```
MODEL = "picasso"
input_img = image_loader("maja.jpg", size=256)
imshow(input_img)
```

In [ ]:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
style_subnet = torch.load('models/style_subnet_picasso.pt', map_location='cpu').eval().to(d
enhance_subnet = torch.load('models/enhance_subnet_picasso.pt', map_location='cpu').eval().
refine_subnet = torch.load('models/refine_subnet_picasso.pt', map_location='cpu').eval().to
```

In [ ]:

```
print("Start transforming on {}..".format(device))
start = time.time()
with torch.no_grad():
    generated_img_256, resized_input_img_256 = style_subnet(input_img)
    generated_img_512, resized_input_img_512 = enhance_subnet(generated_img_256)
    generated_img_1024, resized_input_img_1024 = refine_subnet(generated_img_512)
print("Image transformed. Time for pass: {:.2f}s".format(time.time() - start))

imshow(generated_img_256)
imshow(generated_img_512)
imshow(generated_img_1024)
save_image(generated_img_256, title="generated_images/multimodal_" + MODEL + "_256")
save_image(generated_img_512, title="generated_images/multimodal_" + MODEL + "_512")
save_image(generated_img_1024, title="generated_images/multimodal_" + MODEL + "_1024")
```

In [ ]:

```

```