A Mini Project report on

Style Transfer Using Pytorch and CNN

submitted in partial fulfillment of the course
CSE-2009: Soft Computing

By

19BCE7467-Ishan Gupta
19BCN7283-Kinshuk Agarwal

Submitted to
Dr. Hari Seetha

VIT® AP

School of Computer Science & Engineering
VIT-AP UNIVERSITY, INAVOLU, AMARAVATI
OCTOBER, 2019

1.0 ABSTRACT:

Convolutional Neural Networks (CNNs) are a category of Neural Network that have proven very effective in areas such as image recognition and classification. CNNs have been successful in computer vision related problems like identifying faces, objects and traffic signs apart from powering vision in robots and self-driving cars.

CNN is shown to be able to well replicate and optimise these key steps in a unified framework and learn hierarchical representations directly from raw images. If we take a convolutional neural network that has already been trained to recognize objects within images then that network will have developed some internal independent representations of the content and style contained within a given image.

Hence This can be violated and used for style transfer.

1. Objectives:

- Use neural style transfer to randomize texture, contrast, and color
- Allow users to create their own artwork using certain content and style images
- Transferring the style of the source image

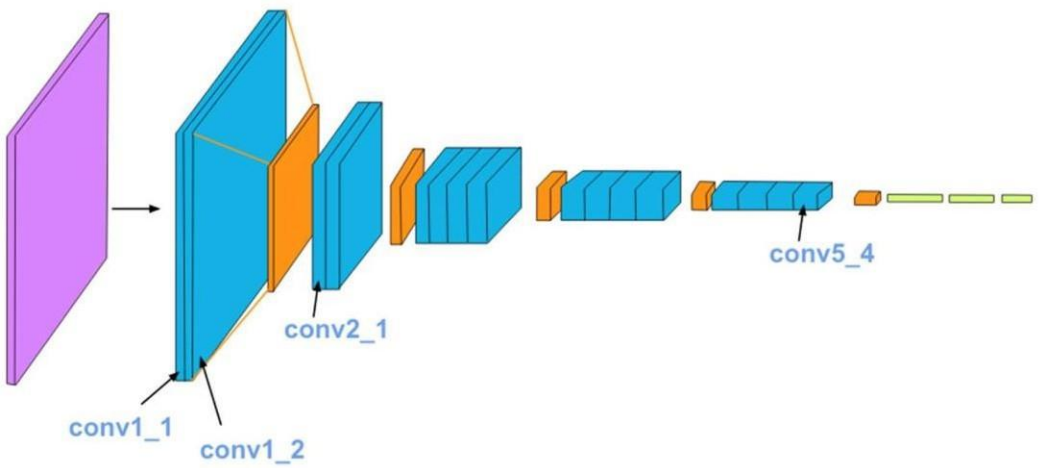
1. Introduction

Style transfer is the technique of recomposing images in the style of other images. It all started when Gatys et al. published an awesome paper on how it was actually possible to transfer artistic style from one painting to another picture using convolutional neural networks

style transfer uses the features found in the 19layer VGG Network, which is comprised of a series of convolutional and pooling layers, and a few fully-connected layers.

In the image below, the convolutional layers are named by stack and their order in the stack.

1. Conv_1_1 is the first convolutional layer that an image is passed through, in the first stack.
2. Conv_2_1 is the first convolutional layer in the *second* stack.
3. The deepest convolutional layer in the network is conv_5_4.



PROPOSED METHDOLOGY:

1. Neural Networks

Neural Network. In a regular Neural Network there are three types of layers:

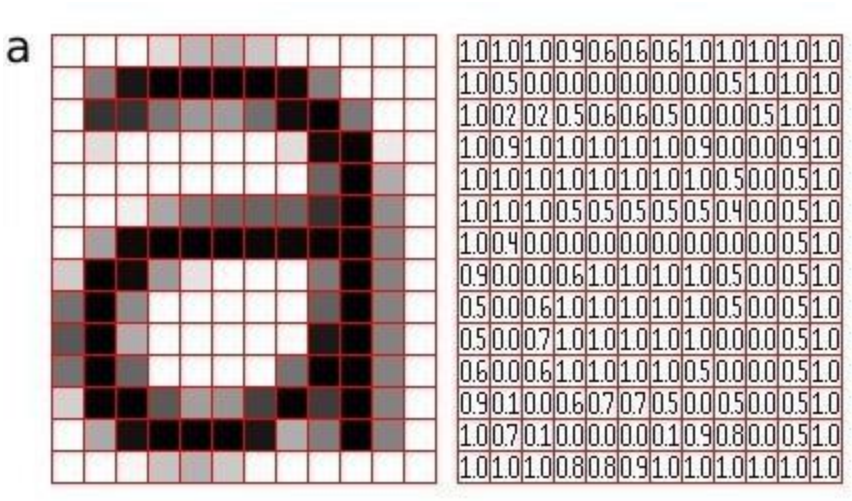
1. Input Layers: It's the layer in which we give input to our model. The number of neurons in this layer is equal to total number of features in our data (number of pixels in case of an image).

2. Hidden Layer: The input from Input layer is then feed into the hidden layer. There can be many hidden layers depending upon our model and data size. Each hidden layer can have different numbers of neurons which are generally greater than the number of features. The output from each layer is computed by matrix multiplication of output of the previous layer with learnable weights of that layer and then by addition of learnable biases followed by activation function which makes the network nonlinear.

3. Output Layer: The output from the hidden layer is then fed into a logistic function like sigmoid or SoftMax which converts the output of each class into probability score of each class.

2. Convolutional Neural Network

A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be.



The human brain processes a huge amount of information the second we see an image. Each neuron works in its own receptive field and is connected to other neurons in a way that they cover the entire visual field. Just as each neuron responds to stimuli only in the restricted region of the visual field called the receptive field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well. The layers are arranged in such a way so that they detect simpler patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along.
By using a CNN, one can enable sight to computers.

2.2 Convolutional Neural Network Architecture

A CNN typically has three layers:

1.a convolutional layer,

2. pooling layer,

3.and fully connected layer.

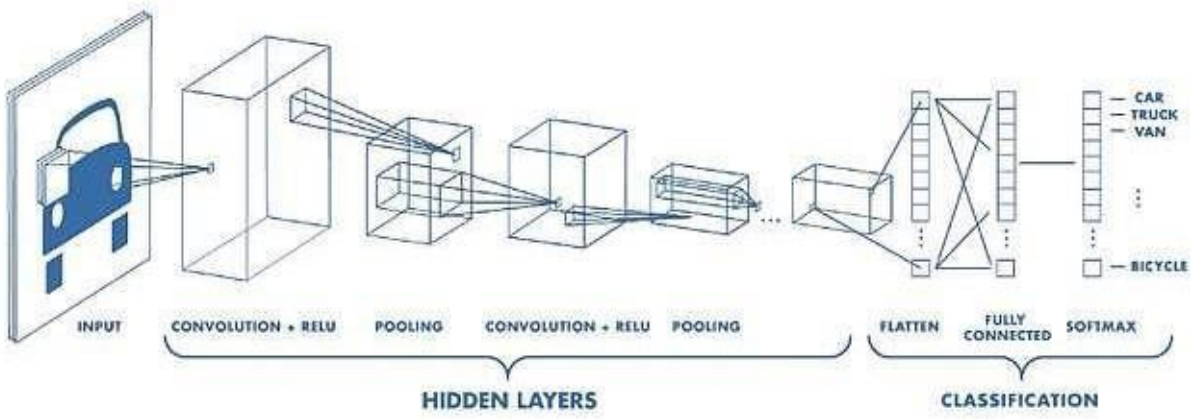


Figure 2: Architecture of a CNN

Convolution Layer

The convolution layer is the core building block of the CNN. It carries the main portion of the network's computational load.

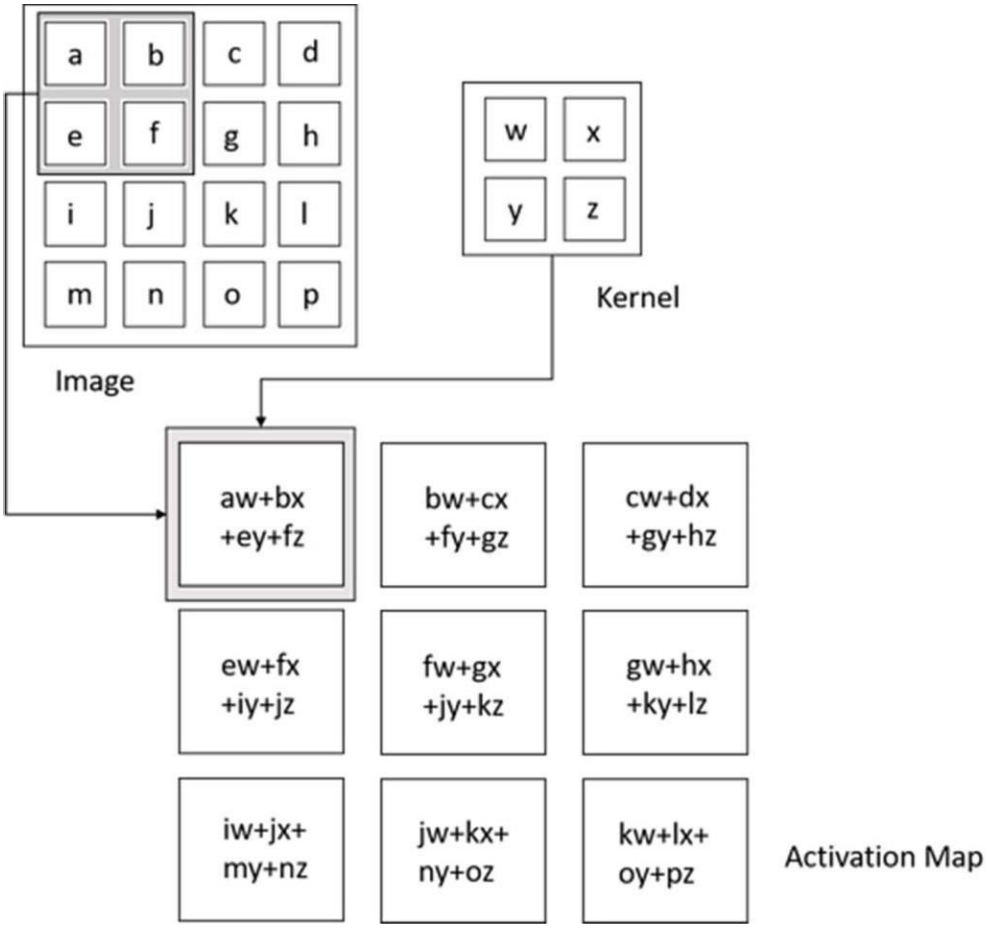
This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field. The kernel is spatially smaller than an image, but is more in-depth. This means that, if the image is composed of three (RGB) channels, the kernel height and width will be spatially small, but the depth extends up to all three channels.

During the forward pass, the kernel slides across the height and width of the image producing the image representation of that receptive region. This produces a two-dimensional representation of the image known as an activation map that gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called a stride.

If we have an input of size W x W x D and Dout number of kernels with a spatial size of F with stride S and amount of padding P, then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F + 2P}{S} + 1$$

This will yield an output volume of size Wout x Wout x Dout.



Pooling Layer

The pooling layer replaces the output of the network at certain locations by deriving a summary statistic of the nearby outputs. This helps in reducing the spatial size of the representation, which decreases the required amount of computation and weights. The pooling operation is processed on every slice of the representation individually.

There are several pooling functions such as the average of the rectangular neighborhood, L2 norm of the rectangular neighborhood, and a weighted average based on the distance from the central pixel. However, the most popular process is max pooling, which reports the maximum output from the neighborhood.

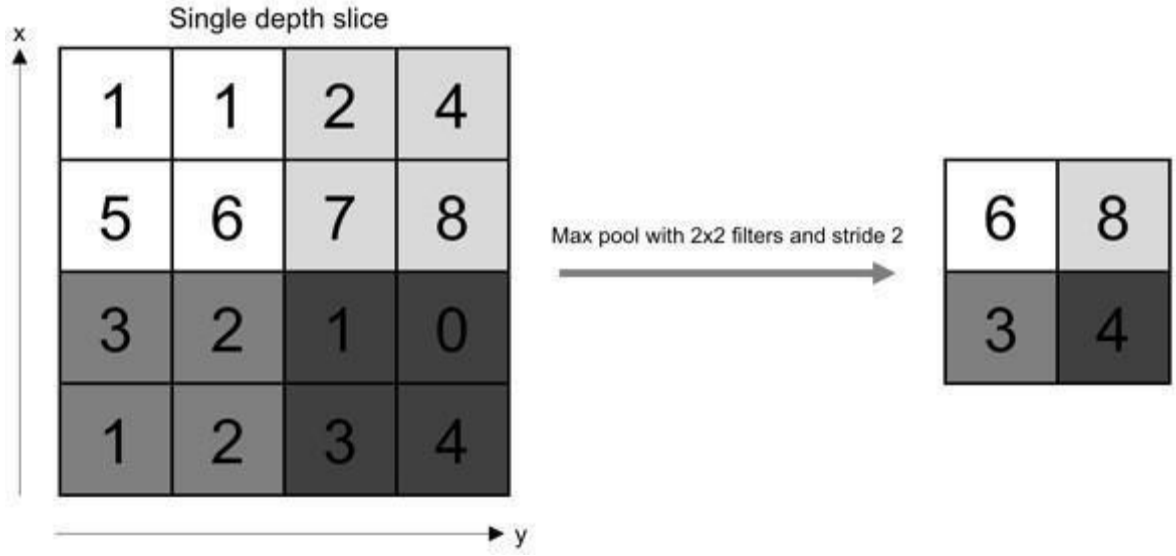


Figure 4: Pooling Operation

If we have an activation map of size W x W x D, a pooling kernel of spatial size F, and stride S, then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F}{S} + 1$$

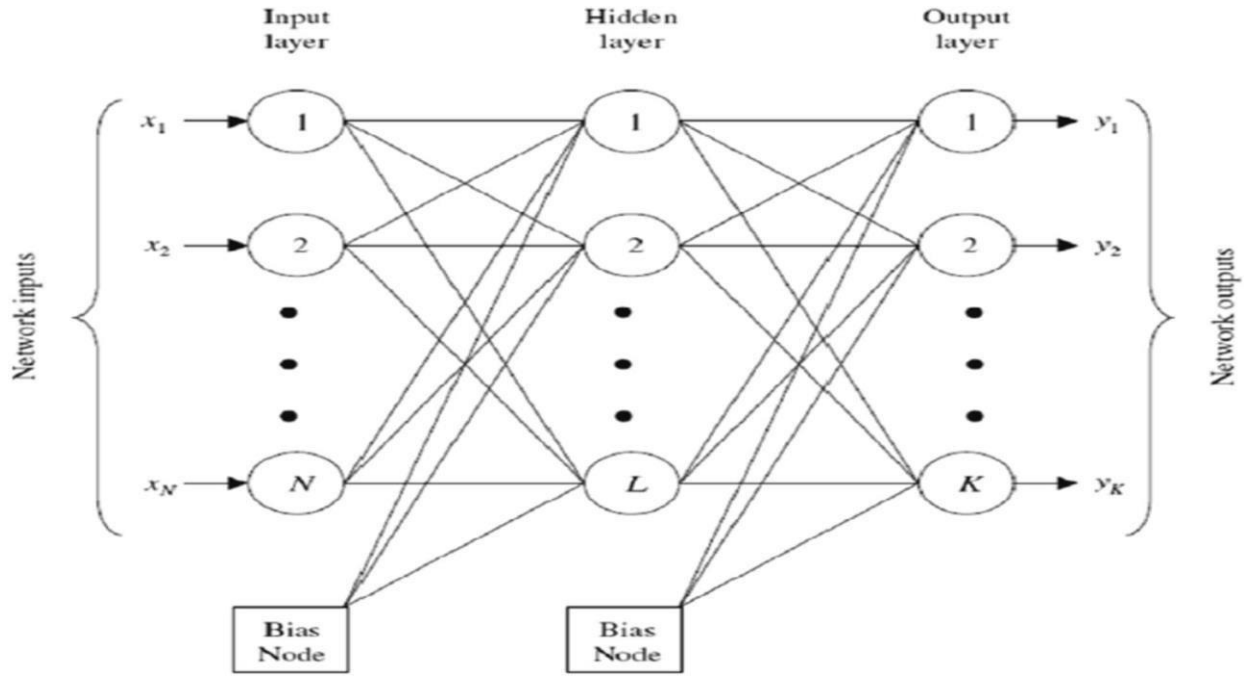
This will yield an output volume of size Wout x Wout x D.

In all cases, pooling provides some translation invariance which means that an object would be recognizable regardless of where it appears on the frame.

Fully Connected Layer

Neurons in this layer have full connectivity with all neurons in the preceding and succeeding layer as seen in regular FCNN. This is why it can be computed as usual by a matrix multiplication followed by a bias effect.

The FC layer helps map the representation between the input and the output.



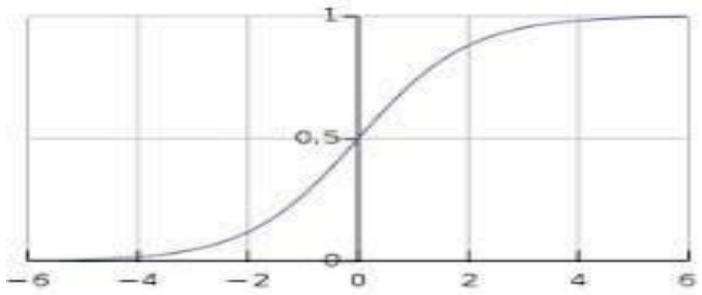
Non-Linearity Layers

Since convolution is a linear operation and images are far from linear, non-linearity layers are often placed directly after the convolutional layer to introduce non-linearity to the activation map.

There are several types of non-linear operations, the popular ones being:

1. Sigmoid

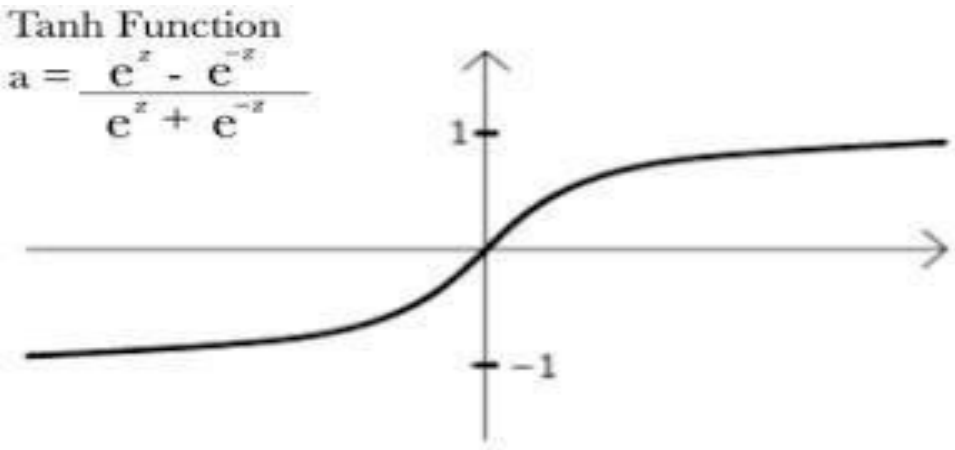
The sigmoid non-linearity has the mathematical form $\sigma(\kappa) = 1/(1+e^{-}\kappa)$. It takes a real-valued number and "squashes" it into a range between 0 and 1.



However, a very undesirable property of sigmoid is that when the activation is at either tail, the gradient becomes almost zero. If the local gradient becomes very small, then in backpropagation it will effectively "kill" the gradient. Also, if the data coming into the neuron is always positive, then the output of sigmoid will be either all positives or all negatives, resulting in a zig-zag dynamic of gradient updates for weight.

2. Tanh

Tanh squashes a real-valued number to the range [-1, 1]. Like sigmoid, the activation saturates, but—unlike the sigmoid neurons—its output is zero centered.

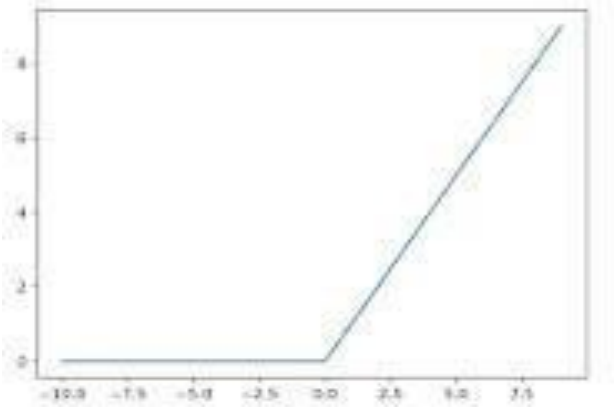


3.ReLU

The Rectified Linear Unit (ReLU) has become very popular in the last few years. It computes the function $f(\kappa)=\max(0,\kappa)$. In other words, the activation is simply threshold at zero.

In comparison to sigmoid and tanh, ReLU is more reliable and accelerates the convergence by six times.

Unfortunately, a con is that ReLU can be fragile during training. A large gradient flowing through it can update it in such a way that the neuron will never get further updated. However, we can work with this by setting a proper learning rate.



5.0 PLATFORM:

PyTorch is an open source machine learning library based on the Torch library,[1][2][3] used for applications such as computer vision and natural language processing.[4] It is primarily developed by Facebook's artificial intelligence research group.[5][6][7] It is free and open-source software released under the Modified BSD license. Although the Python interface is more polished and the primary focus of development,

PyTorch also has a C++ frontend.[8] Furthermore, Uber's

Pyro probabilistic programming language software uses PyTorch as a backend.[9]

PyTorch provides two high-level features:[10]

- Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU)
- Deep neural networks built on a tape-based autodiff system

1. RESULT ANALYSIS

```
1.  # import resources
2.  %matplotlib inline
3.
4.  from PIL import Image
5.  from io import BytesIO
6.  import matplotlib.pyplot as plt 7. import numpy as np
8.
9.  import torch
10. import torch.optim as optim
11. import requests
12. from torchvision import transforms, models
13.
14. # getting the "features" portion of VGG19 15. vgg =
models.vgg19(pretrained=True).features
16.
17.         # freezing all VGG parameters since we're only optimizing the target image
18.         for param in vgg.parameters():
19.         param.requires_grad_(False)
    20.
21. ### Load in Content and Style Images
22.
23. # The `load_image` function also converts images to normalized Tensors.
24.
25. def load_image(img_path, max_size=400, shape=None):
26. """ Load in and transform an image, making sure the image  27.       is <= 400 pixels in the x-y
    dims."""
28.             if "http" in img_path:
29.             response = requests.get(img_path)
30.             image = Image.open(BytesIO(response.content)).convert('RGB')
31.             else:
32.             image = Image.open(img_path).convert('RGB')
        33.
34.    # large images will slow down processing   35.     if
max(image.size) > max_size:   36.         size = max_size
37.             else:
38.             size = max(image.size)
```

```
39.
40.    if shape is not None:   41.         size =
 shape
42.
43.                                        in_transform = transforms.Compose([
44.                                        transforms.Resize(size),
45.                                        transforms.ToTensor(),
46.                                       transforms.Normalize((0.485, 0.456, 0.406),    47.
                                           (0.229, 0.224, 0.225))])
48.
49.         # discard the transparent, alpha channel (that's the :3) and add the batch  dimension
50.         image = in_transform(image)[:3,:,:].unsqueeze(0)
    51.
52.    return image
53.
54.         # helper function for un-normalizing an image
55.         # and converting it from a Tensor image to a NumPy image for display
56.         def im_convert(tensor):
57.         """ Display a tensor as an image. """
    58.
59.         image = tensor.to("cpu").clone().detach()
60.         image = image.numpy().squeeze()
61.         image = image.transpose(1,2,0)
62.         image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456,
     0.406))
63.         image = image.clip(0, 1)
64.
65.    return image
66.
67.  # load in content and style image
68.  content = load_image('space_needle.jpg')
69.  # Resize style to match content, makes code easier
70.  style = load_image('delaunay.jpg', shape=content.shape[-2:])
71.
72.  # display the images
73.  fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))
74.  # content and style ims side-by-side
75.  ax1.imshow(im_convert(content)) 76. ax2.imshow(im_convert(style))
77.
78.  ## Run an image forward through a model and get the features for     79.  ##       a set of layers. Default
 layers are for VGGNet matching Gatys et al (2016).
80.
81.                                    def get_features(image, model, layers=None):
82.                                    if layers is None:
83.                                    layers = {'0': 'conv1_1',
84.                                    '5': 'conv2_1',
85.                                    '10': 'conv3_1',
86.                                    '19': 'conv4_1',
87.                                    '21': 'conv4_2',  ## content representation   88.                 '28':
                                       'conv5_1'}
89.
90.         features = {}
```

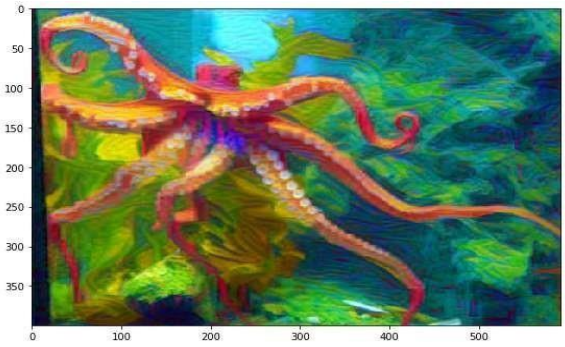
```
        x = image
        for name, layer in model._modules.items():
        x = layer(x)          if name in layers:
          features[layers[name]] = x

    return features

##Calculate the Gram Matrix of a given tensor

                        def gram_matrix(tensor):
                        _, d, h, w = tensor.size()
                        tensor = tensor.view(d, h * w)
                        # the style loss for one layer, weighted appropriately
                        layer_style_loss = style_weights[layer] * torch.mean((target_gram - style_gram)**2)
                        # add to the style loss

                        style_loss += layer_style_loss / (d * h * w)
                # calculate the *total* loss
                total_loss = content_weight * content_loss + style_weight * style_loss

                # update your target image
                optimizer.zero_grad()
                total_loss.backward()
                optimizer.step()

            # display intermediate images and print the loss          if ii %
show_every == 0:
                        print('Total loss: ', total_loss.item())
                        plt.imshow(im_convert(target))

                        plt.show()

        # display content and final, target image
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))

        ax1.imshow(im_convert(content))
        ax2.imshow(im_convert(target))
```

TESTCASE -1

CONTENT IMAGE

STYLE IMAGE

OUTPUT



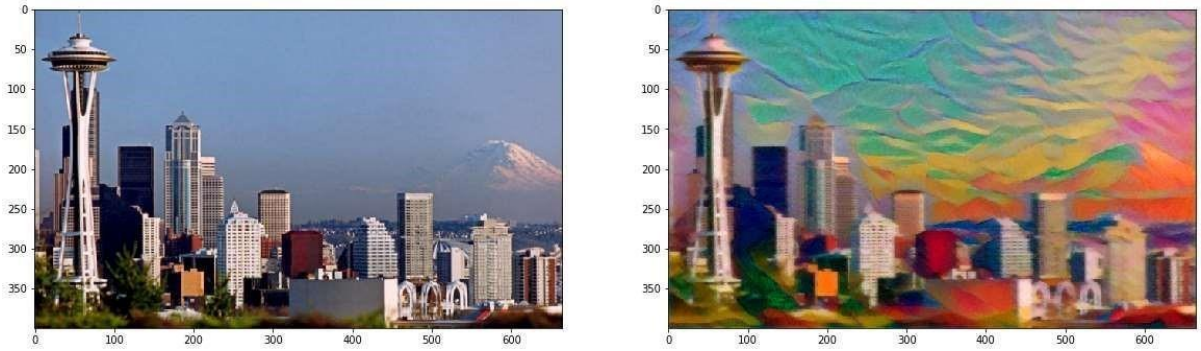
TARGET IMAGE

TESTCASE 2

CONTENT IMAGE

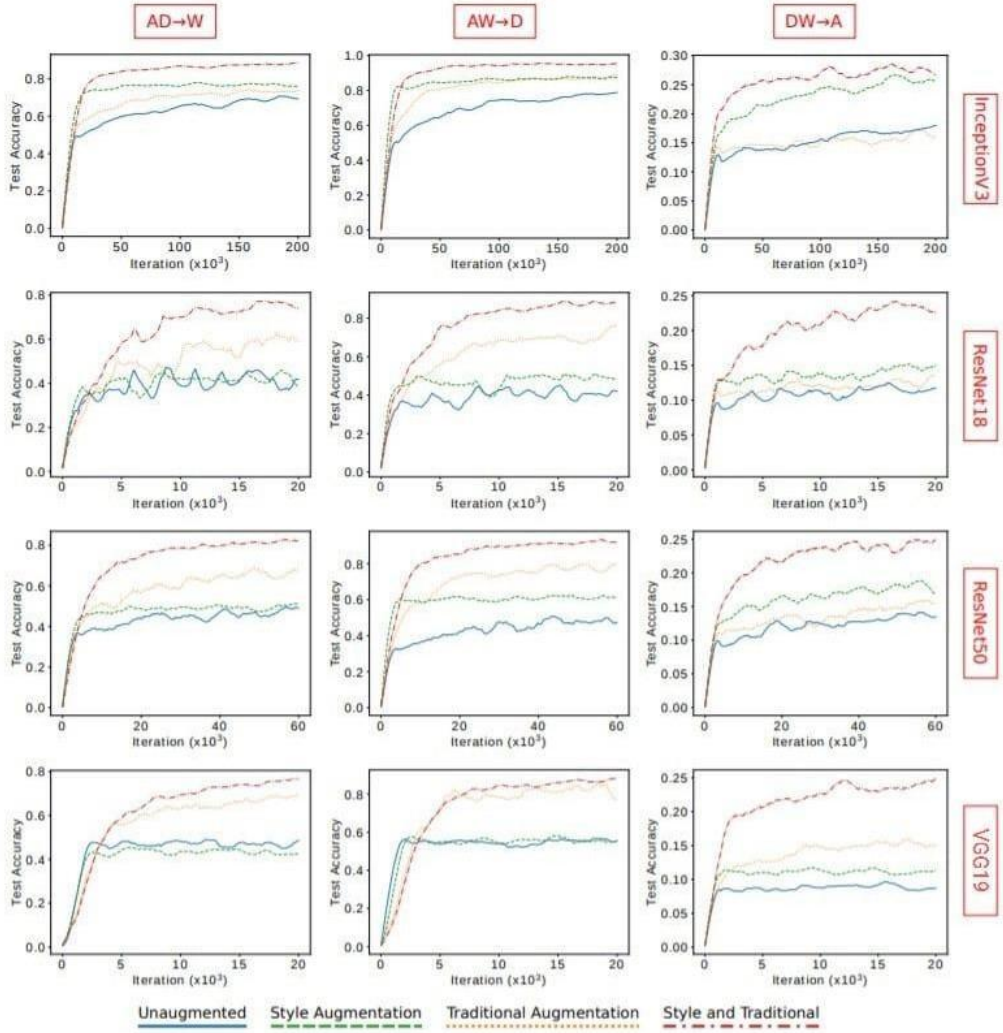
STYLE IMAGE

OUTPUT

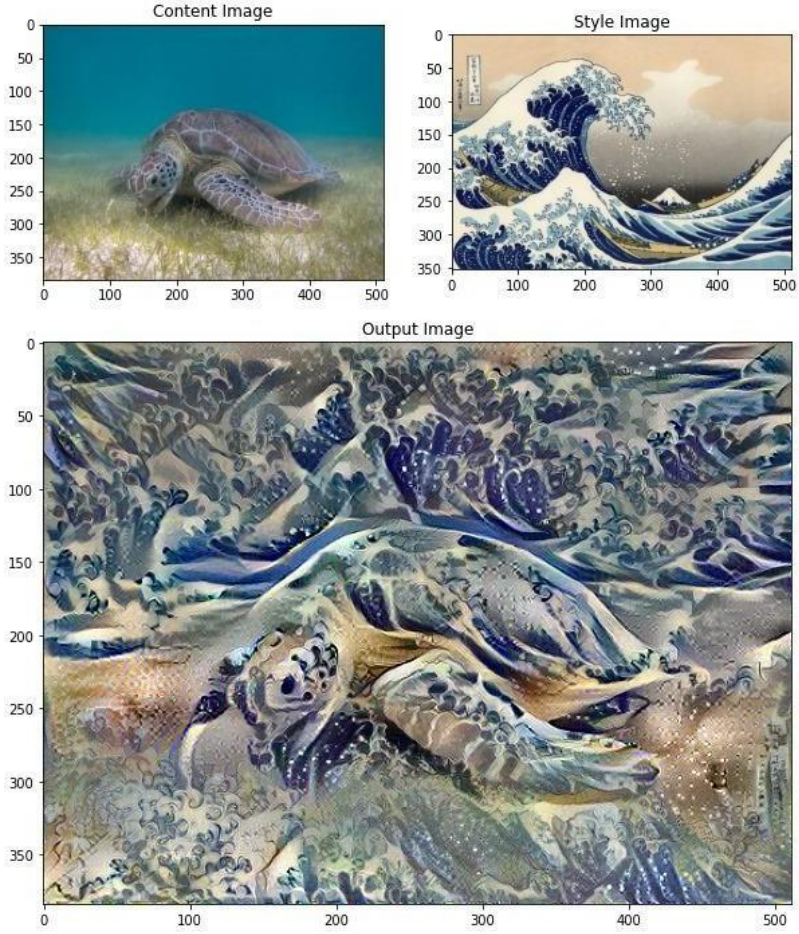


TARGET IMAGE

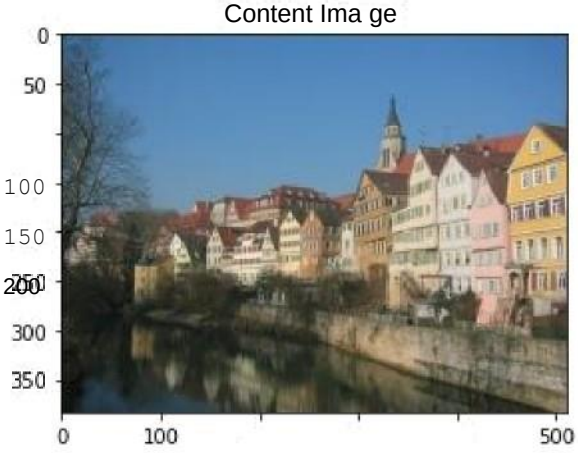
GRPAH SHOWING ACCURACY OF DIFFERENT MODELS USED



6.0 RESULT USING SIMPLE GRADIENT DESCENT OPTIMIZER



Content Ima ge



Style Image

20d 300 40g

0 I00 500
2o0

Output Image

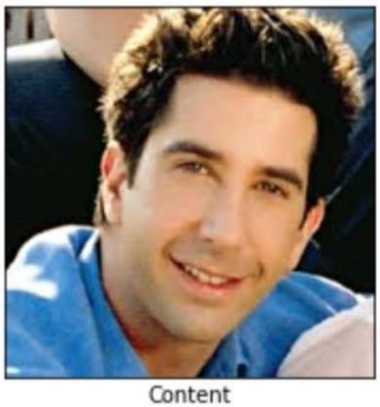
I0d 20d 300

7.0 RESULT USING DIFFERENT NET(VGG16)

```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import PIL.Image
```

```
from vgg16 import vgg16
```

```
#This is the main optimization algorithm for the Style-Transfer algorithm. It is basically just gradient descent
#on the loss-functions defined above.
def style_transfer(content_image, style_image,
                   content_layer_ids, style_layer_ids,
                   weight_content=1.5, weight_style=10.0,
                   weight_denoise=0.3,
                   num_iterations=120, step_size=10.0):
    """
    Use gradient descent to find an image that minimizes the
    loss-functions of the content-layers and style-layers. This
    should result in a mixed-image that resembles the contours
    of the content-image, and resembles the colours and textures
    of the style-image.

    Parameters:
    content_image: Numpy 3-dim float-array with the content-image.
    style_image: Numpy 3-dim float-array with the style-image.
    content_layer_ids: List of integers identifying the content-layers.
    style_layer_ids: List of integers identifying the style-layers.
    weight_content: Weight for the content-loss-function.
    weight_style: Weight for the style-loss-function.
    weight_denoise: Weight for the denoising-loss-function.
    num_iterations: Number of optimization iterations to perform.
    step_size: Step-size for the gradient in each iteration.
    """

    # Create an instance of the VGG16-model. This is done
    # in each call of this function, because we will add
    # operations to the graph so it can grow very large
    # and run out of RAM if we keep using the same instance.
    model = vgg16.VGG16()

    # Create a TensorFlow-session.
    session = tf.InteractiveSession(graph=model.graph)
```



Content

Mixed

Style

8.0 RESULT AND DISCUSSION

Using neural style transfer to randomize texture, contrast, and color, while preserving shape and semantic content the sample images have been transformed into the target images using CNN.

The images are transformed in such a manner that the contents of the sample are preserved to the maximum possible limits thereby helping in better augmented services, forensic analysis and perspective change in various industrial applications.

We iteratively updated our image by applying our optimizers update rules. The optimizer minimized the given losses with respect to our input image.

The core innovation of NST is the use of deep learning to disentangle the representation of the content (structure) of an image, from the appearance (style) in which it is depicted.

CONCLUSION

Thus we conclude the Neural Style Transfer to be an advanced and futuristic technique to render images and generate results from ordinary input images into artistic impressions which previously could only be obtained by an artist. The future holds a lot of improvements in the process but for time being the techniques can be extensively used to ease any problems including customized beautifications, forensics and augmented reality basics.

9.0 FUTURE SCOPE

Neural Style Transfer has also been extended to videos.

Most recently, feature transform based Neural Style Transfer methods have been explored for fast stylization that are not coupled to single specific style and enable user-
controllable blending of styles, for example the Whitening and Coloring Transform (WCT).

One of the most intriguing recent ideas is the CycleGAN, which shows how to train image transformation algorithms from collections of before images and after images without requiring correspondences. That is, you can train to convert images of southern France to Monet paintings, without ever needing a photograph to go with each of the Monet training paintings.

10. REFERENCES

1. https://www.cvfoundation.org/openaccess/content_cvpr_2016/papers/Gatys_Image_Style_Transfer_CVPR_2016_paper.pdf
2. Bryson, Arthur Earl; Ho, Yu-Chi (1969). Applied optimal control: optimization, estimation, and control. Blaisdell Publishing Company or Xerox College Publishing. p. 481
3. https://colab.research.google.com/github/jakevdp/ PythonDataScien ceHandbook/blob/master/notebooks/01.01-Help-AndDocumentation.ipynb 4. https://pytorch.org/docs/stable/ _modules/torchvision/models/vgg. html
5. Deep Learning; Ian Goodfellow, YoshuaBengio, Aaaron Courville; MIT Press; 2016, p 196
6. "What is Backpropagation?". deepai.org.
7. Nielsen, Michael A. (2015). "Chapter 6". Neural Networks and Deep Learning.

8. "Deep Networks: Overview - Ufldl". ufldl.stanford.edu. Retrieved 2017-08-04.
9. Schmidhuber, Jürgen (2015-01-01). "Deep learning in neural networks: An overview". Neural Networks. 61: 85–117. arXiv:1404.7828. doi: 10.1016/j.neunet.2014.09.003. ISSN 08936080. PMID 25462637.
10. Griewank, Andreas (2012). Who Invented the Reverse Mode of Differentiation?. Optimization Stories, DocumentaMatematica, Extra Volume ISMP (2012), 389400.
11. Seppo Linnainmaa (1970). The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 6–7.
12. Linnainmaa, Seppo (1976). "Taylor expansion of the accumulated rounding error". BIT Numerical Mathematics. 16 (2): 146–160. doi:10.1007/bf01931367.
13. Stuart Russell and Peter Norvig. Artificial Intelligence A Modern Approach. p. 578. The most popular method for learning in multilayer networks is called Backpropagation.
14. Bryson, Arthur Earl; Ho, Yu-Chi (1969). Applied optimal control: optimization, estimation, and control. Blaisdell Publishing Company or Xerox College Publishing. p. 481
15. L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs. arXiv:1412.7062 [cs], Dec. 2014. arXiv: 1412.7062. 2
16. M. Cimpoi, S. Maji, and A. Vedaldi. Deep filter banks for texture recognition and segmentation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 3828–3836, 2015. 2

17. J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. DeCAF: A Deep Convolutional Activation Feature for
18. A. Efros and T. K. Leung. Texture synthesis by nonparametric sampling. In Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on, volume 2, pages 1033–1038. IEEE, 1999.
19. A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pages 341–346. ACM, 2001. 1
20. D. Eigen and R. Fergus. Predicting Depth, Surface Normals and Semantic Labels With a Common Multi-Scale Convolutional Architecture. pages 2650–2658, 2015.