Image Style Transfer Using Convolutional Neural Networks

Members: Apurva Modi, Maruf Sakib, Lalita Sharkey

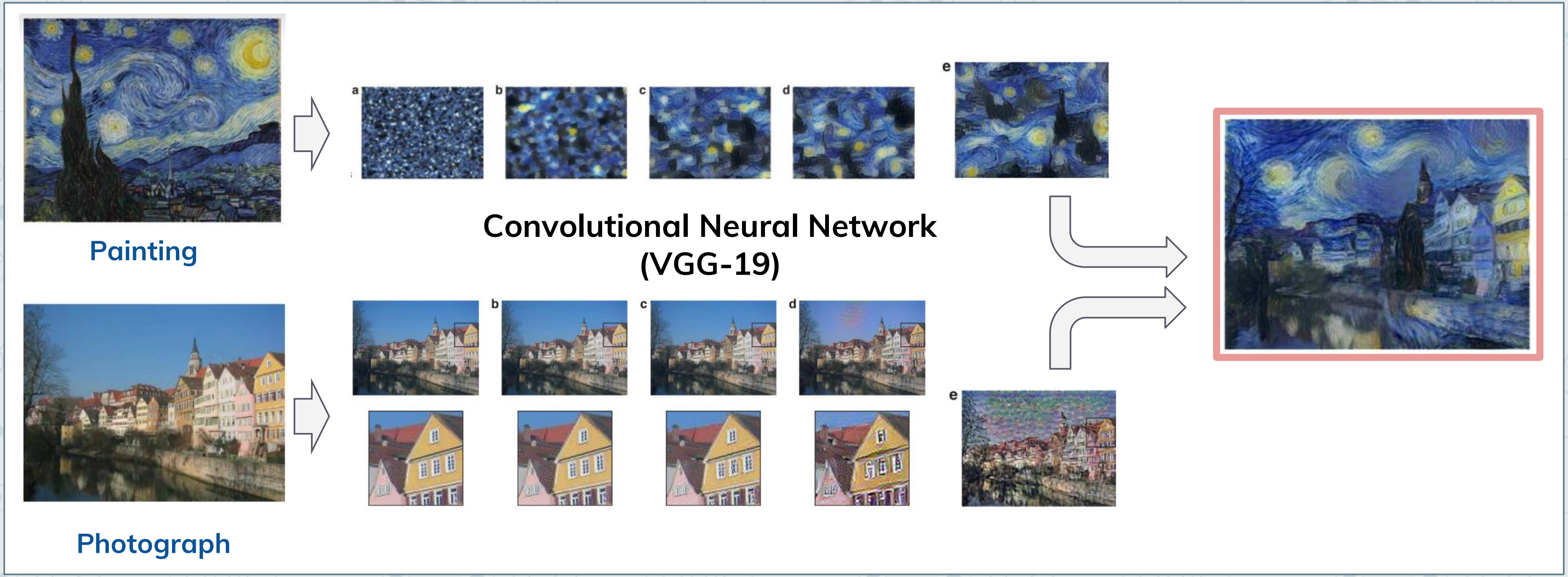
Image Style Transfer

A style transfer is a process of modifying the style or texture of an image while still preserving its content.

Methods

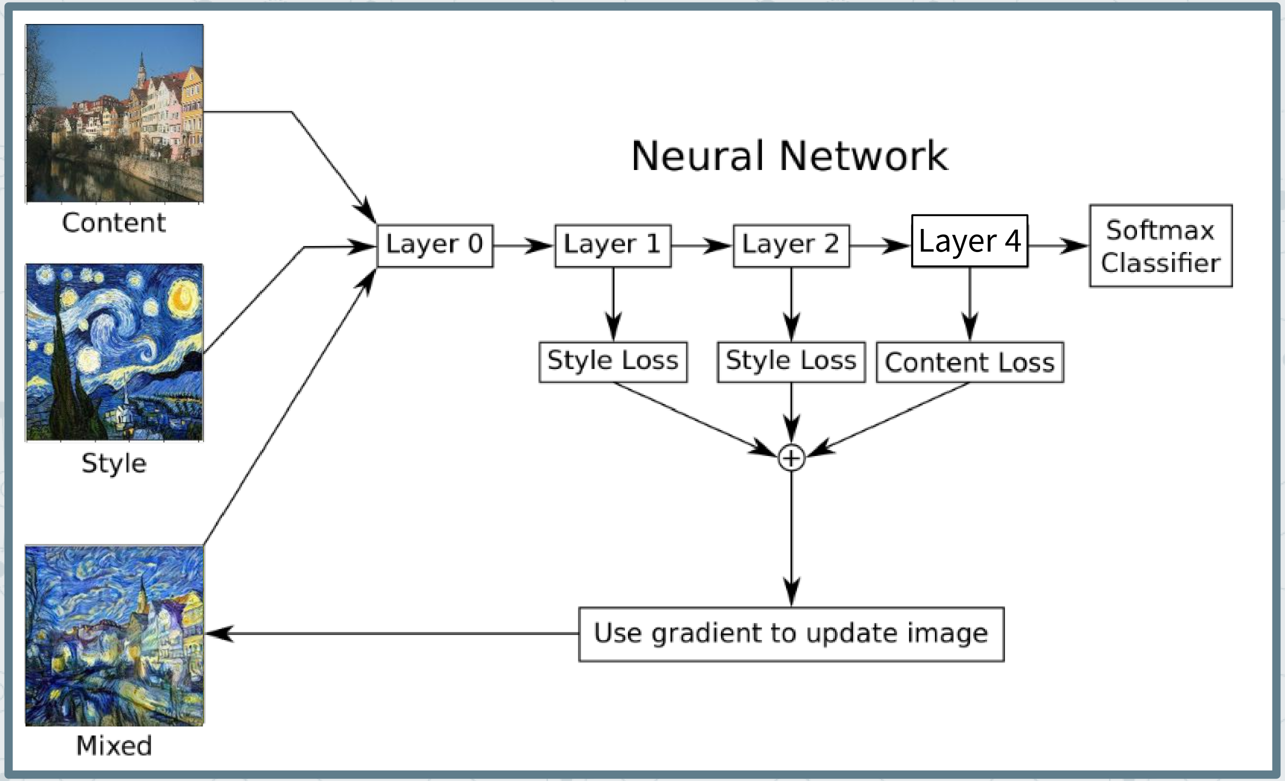
Suggested by the research paper

Style Transfer Algorithm



Our Implementation

The Grand Scheme



Neural Style Transfer Algorithm

1. Importing the necessary packages and the content and the style images
    - TensorFlow 1.x and Enable Eager Execution
    - Tensorflow.keras.applications.vgg19

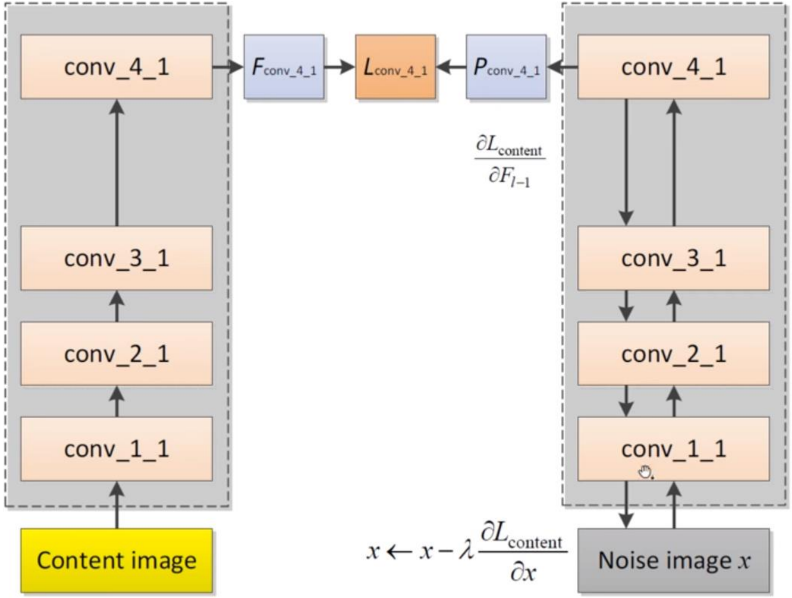
2. VGG networks are trained on image with each channel normalized by mean = [103.939, 116.779, 123.68] and with channels BGR.

3. Loaded and Preprocessed Images as VGG preprocessed input.

    3.1 We also initiate “x” randomly to produce white noise image with random pixel values.

Neural Style Transfer Algorithm

4. Content extraction and Content Loss



$$\mathcal{L}_{\text{content}}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F_{ij}^{l} - P_{ij}^{l}\right)^2$$

= The activation of the **l th** layer,
i th feature map,
jth position obtained using *the generated image*

= The activation of the **l th** layer,
i th feature map,
jth position obtained using *the noise image*

Content Loss captures the root mean squared error between the activations produced by the generated image and the content image

Neural Style Transfer Algorithm

5. Style Extraction and Style Loss

- Use activation of conv_1_1, conv_2_1, conv_3_1, conv_4_1, and conv_5_1
- On each layer included in the style representation, the element-wise mean squared difference between the noise feature and style representation is computed to give Style Loss
- Calculate the Gram matrix from activations of each layer

$$G_{ij}^{l} = \sum_{k} F_{ik}^{l} F_{jk}^{l}.$$

- Gram matrices allows detecting global repeating patterns-textures in image and be “blind” to local features

Neural Style Transfer Algorithm

6. Total Loss and Termination

$\vec{p} =$

Content image

$\vec{a} =$

Style image

$$\vec{x} := \vec{x} - \lambda \frac{\partial \mathcal{L}_{total}}{\partial \vec{x}}$$

$\vec{x} =$

Result image

$$\mathcal{L}_{total} = \alpha \mathcal{L}_{content} + \beta \mathcal{L}_{style}$$

Note: α, β = some weights

Optimisation: Total_Loss derivative or the **gradient** with respect to the pixel values can be computed using error back-propagation and can be used as input for some numerical optimisation strategy.

They used **L-BFGS** and we used **ADAM**

The gradient is used to iteratively update the image (x) until it simultaneously matches the style features of the style image (a) and the content features of the content image (p).

Problem: Slow runtime

Problems:

- Many *passes* through the Network (both forward and backward)
- Many *Optimization* Steps per new Image
- Some *differences* in the image synthesis are expected due to the optimisation algorithm we use.

Solution:

- Training a neural network to perform the style transfer

Dataset Description

The dataset that was used to evaluate our implementation

Neckarfront in Tubingen, Germany

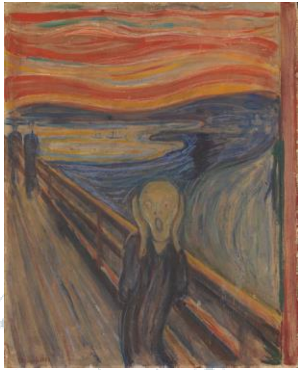
Shipwreck of the Minotaur by J.M.W. Turner, 1805

The Starry Night by Vincent van Gogh, 1889

D Der Schrei by Edvard Munch, 1893

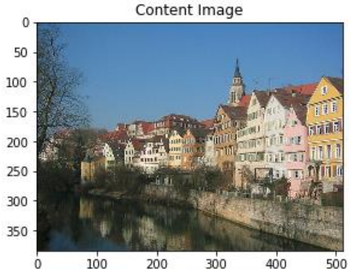
Femme nue assise by Pablo Picasso, 1910

Composition VII by Wassily Kandinsky, 1913



Our Results

Results comparison



Result from our implementation

Result from the research paper

```
best, best_loss = run_style_transfer(content_path, style_path, num_iterations=5000)
```

Result from some iterations

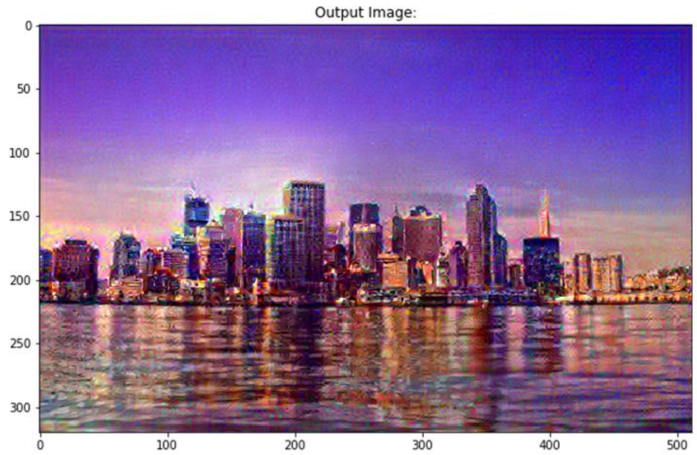
Changes over running the code for 5,000 iteration

More results comparison

Results from the research paper

Results from our implementation

Image Style Transfer



Swapped

Implementation Result Justification

- **Not all parameters was listed** on the research paper
  - Ex: number of iteration
  - We randomly picked the numbers and tried with many different number
- The images used in our implementation were taken from Google as the **images were not provided with the paper**
  - Colors and clarity of the images were different
- We simply need **more computational power** so we can perform more optimization iterations with smaller step-sizes and for higher-resolution images
- We need to use a **more sophisticated optimization** method.

Additional work

Because we love machine learning!!

Comparing other relevant algorithm

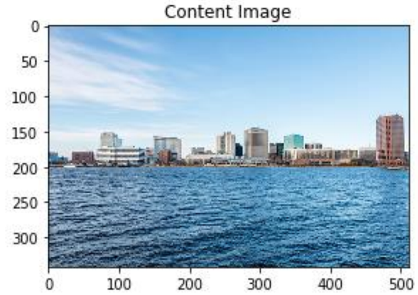
- CNN method method produced beautiful neural style transfer results, the problem was that it was quite **slow**
- Johnson et al. (2016) proposing a neural style transfer algorithm that is up to **three orders of magnitude faster**
- Neural style transfer with OpenCV and Python
- Load a pre-trained neural style transfer model
- The biggest downside is that you cannot arbitrarily select your style images

Comparing other relevant algorithm

Result from OpenCV version

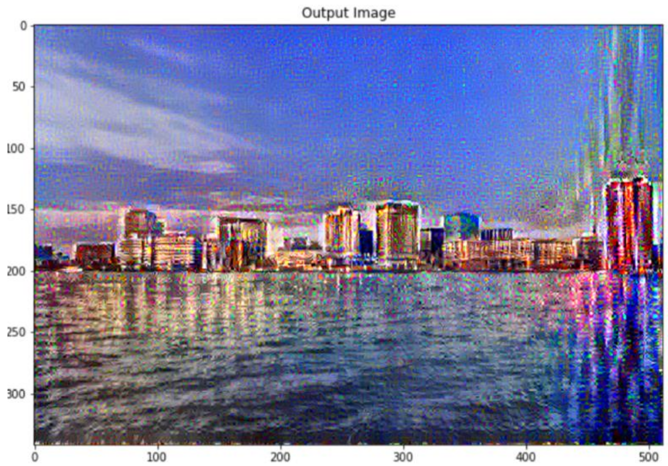
Result from the research paper

Apply the algorithm to additional Images



Norfolk by day

Norfolk by night

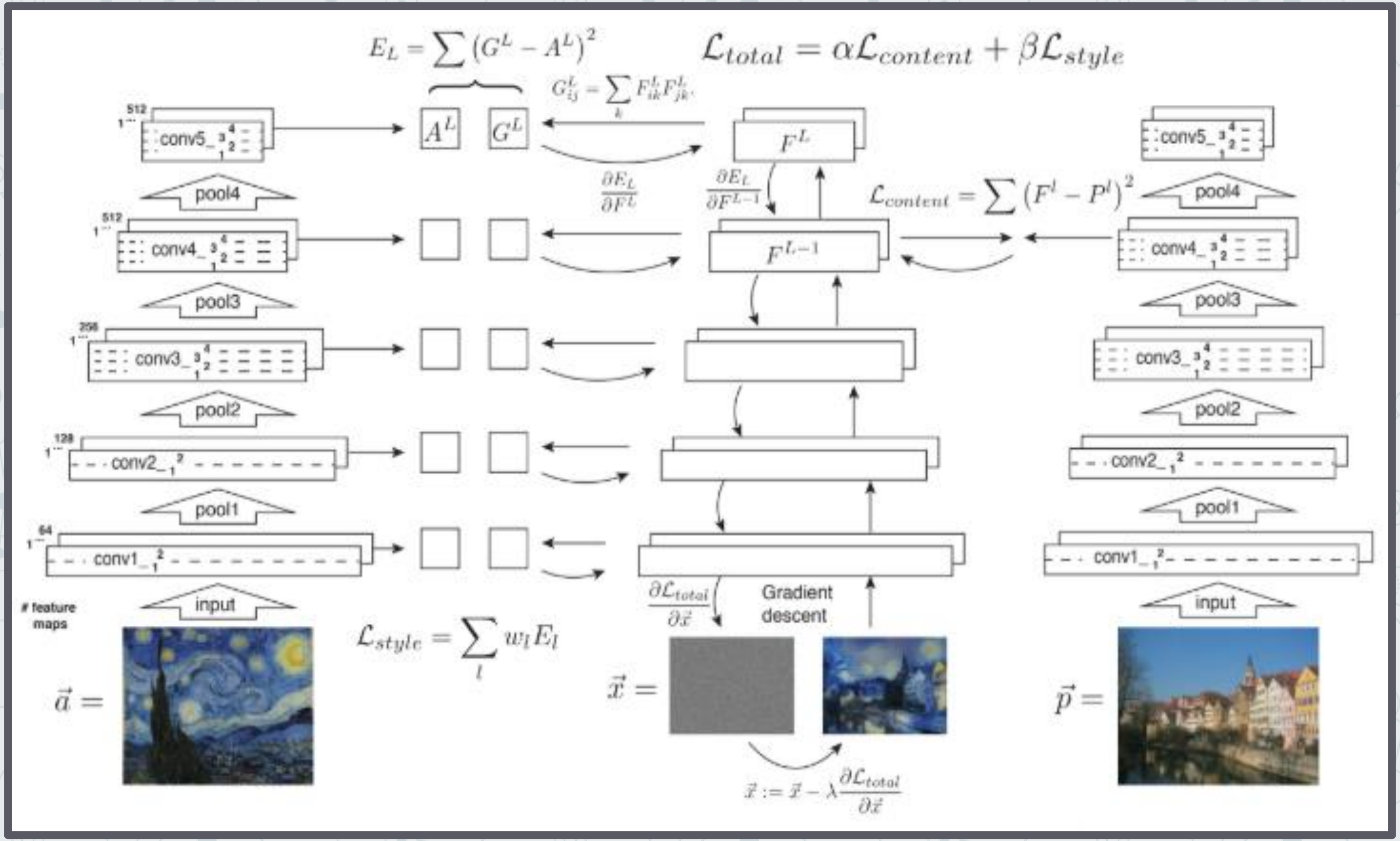


Thanks!

Any questions?

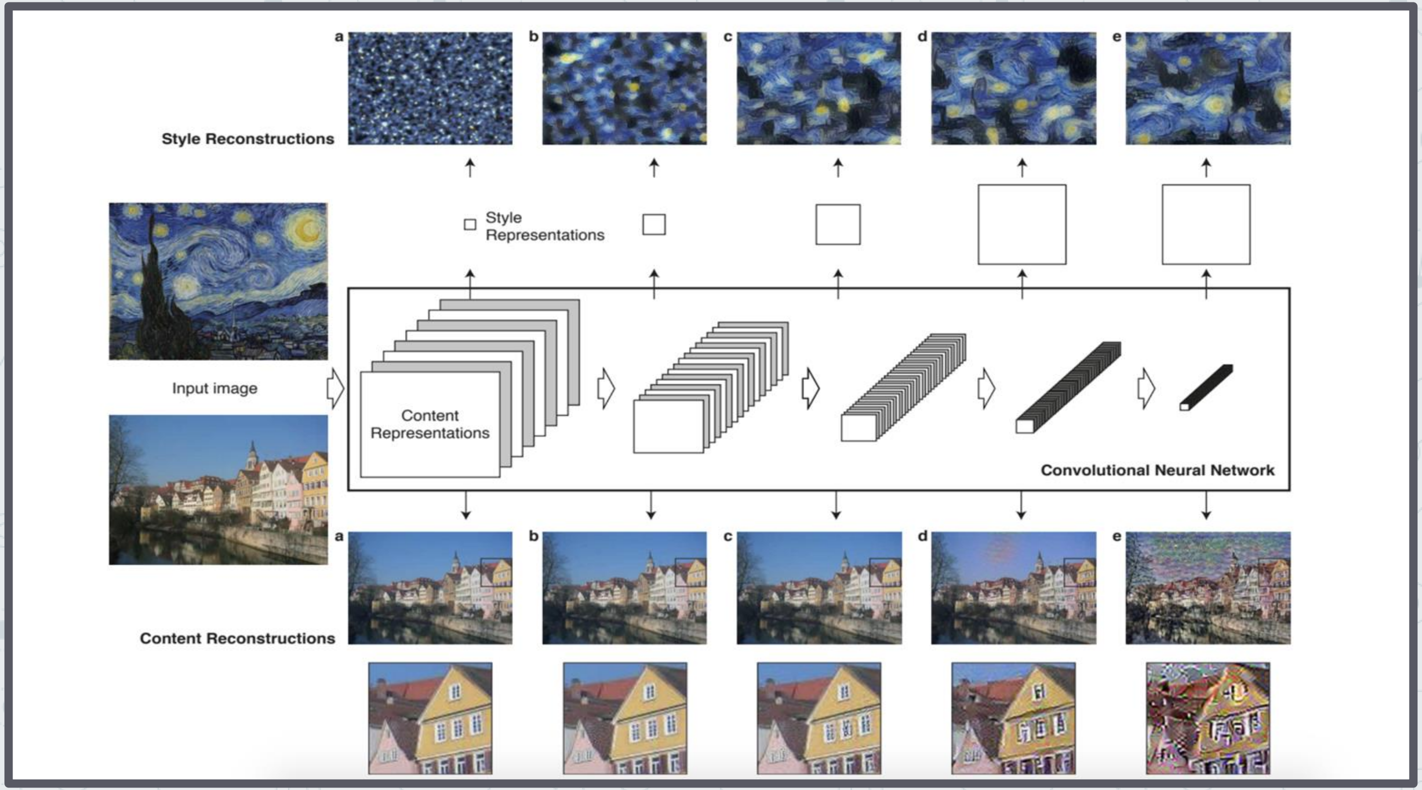
Appendix

Style Transfer Algorithm

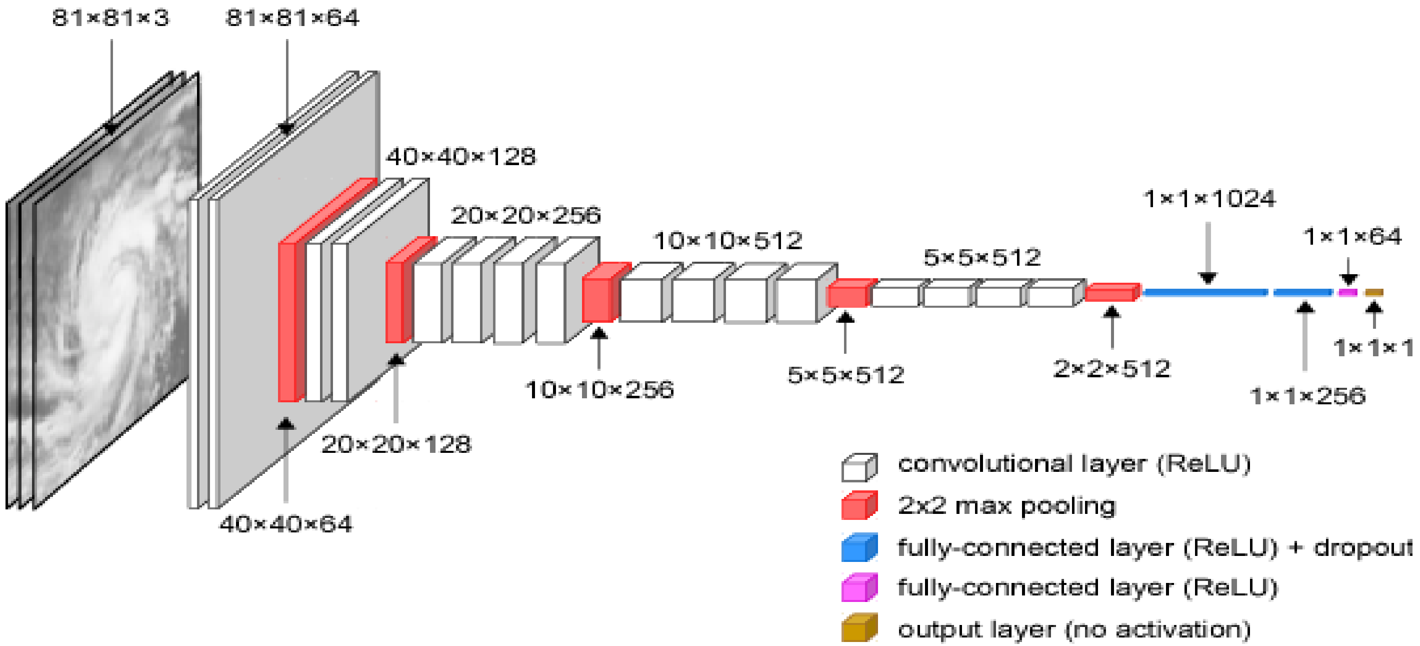


a = artwork
A = Style representation
alpha = content weight
beta =style weight
E = Sum(G - A)^2
F = content feature x Alpha beta losses
G = Style feature x
l = layer
L = loss
p = photograph
P = Content representation
W = weighing factor
x = noise (combination)

Convolutional Neural Network (CNN)



VGG - 19 Architecture



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, None, None, 3)     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, None, None, 64)    1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, None, None, 64)    36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, None, None, 64)    0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, None, None, 128)   73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, None, None, 128)   147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, None, None, 128)   0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, None, None, 256)   295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv4 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, None, None, 256)   0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, None, None, 512)   1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv4 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, None, None, 512)   0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv4 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, None, None, 512)   0
=================================================================
Total params: 20,024,384
Trainable params: 20,024,384
Non-trainable params: 0
_________________________________________________________________
```

Neural Style Transfer with OpenCV

- Load a pre-trained neural style transfer model into memory
- Load the input image and resize it
- Construct a blob by performing mean subtraction
  - cv2.dnn.blobFromImage
- Perform a forward pass to obtain an output image
- Reshape the output tensor
- Add back in the mean subtraction, and then swap the channel ordering
- Show the output of the neural style transfer process to the screen

Role of Optimizers

Optimizers update the weight parameters to minimize the loss function. Loss function acts as guides to the terrain telling optimizer if it is moving in the right direction to reach the bottom of the valley, the global minimum.

L-BFGS vs ADAM

- **L-BFGS** solver is is an optimization algorithm in the family of true quasi-Newton method that it estimates the curvature of the parameter space via an approximation of the Hessian. It has the downside of additional costs in performing a rank-two update to the Hessian approximation at every step.

- **ADAM** is a first order method that attempts to compensate for the fact that it doesn't estimate the curvature by adapting the step-size in every dimension. In some sense, this is similar to constructing a diagonal Hessian at every step, but they do it cleverly by simply using past gradients. In this way it is still a first order method, though it has the benefit of acting as though it is second order.