Neural style transfer

Compose image in the style of another image

Reference paper

A Neural Algorithm of Artistic Style by Gatys et. al (https://arxiv.org/abs/1508.06576)

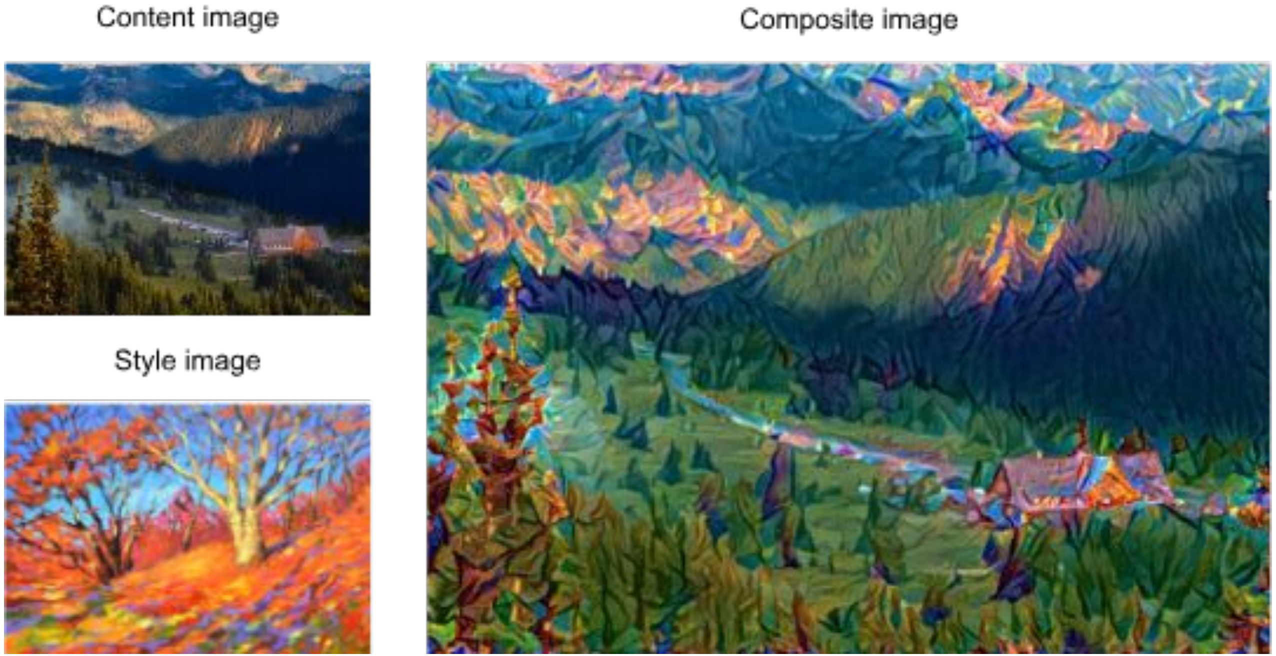
Why style transfer?

Style transfer is a technique to showcase capabilities and internal representations of neural networks

Experiment optimization problem with Convolutional Neural Network design, to make it modifying the style of an image while still preserving its content

It’s an interesting topic :)

Example



https://marvelapp.com/8ej50dg

What is content of image?

The content of the image is what objects are present in this image (person, face, dog, eyes, etc.) and their relationships in space

Higher layers in the network capture the high-level content in terms of objects and their arrangement in the input image but do no constrain the exact pixel values of the reconstruction

What is style of image?

The style of an painting is: the way the painter used brush strokes; how these strokes form objects; texture of objects; color palette used

By including the feature correlations of multiple ayers, we can obtain a stationary, multi-scale representation of the input images, which captures its texture information but not the global arrangement

Principle - optimizing loss

1. How different the content of 2 images are: **LContent**
2. How different the style of 2 images are: **LStyle**

Transform the input image to:

1. Minimize the content distance with content image and
2. Style distance with style image

Minimizing losses with backpropagation

Get content and style representations

They act as feature extractors

1. Looking at intermediate layers
2. Intermediate layers represent feature maps
3. Pre-trained network (We start with VGG19)
4. The combination image starts off as random noise. As the combination image goes through the CNN, it excites some filters in the style layers and some in the content layers.

Feature maps work better for style transfer
Obtained from paper

Loss functions - content and style

Content loss: distance of content from input image (x) and content image (p)

1. Feed content image to network
2. Feed input image to network
3. Calculate euclidean distance between representations of these output images

Content Loss

1. Cnn(X) - Deep convolutional network for X
2. $Flij(x) \in Cnn(x)$ - feature representation of (x) at layer (l)
3. $Plij(p) \in Cnn(p)$ - feature representation of (p) at layer (l)

Content loss at layer (l):

Backpropagation to minimize this loss

$$L^{l}_{content}(p, x) = \sum_{i,j} (F^{l}_{ij}(x) - P^{l}_{ij}(p))^2$$

Style loss - distance between style

1. Feed network input and style images
2. Obtain gram matrices of outputs
3. Obtain distance between gram matrices
4. Obtain correlation between filter responses - style representation
5. *Gli* - inner product of feature map vectors *i* and *j*
6. *Gli* - correlation between feature maps *i* and *j*

Style loss

1. Perform gradient descent from content image to match style of original image
2. Mean squared error of Gram matrices of style image (a) and input image (x)
3. Obtain contribution of style loss at each layer
4. N = number of feature maps
5. M = size of feature map (height * width)

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2$$

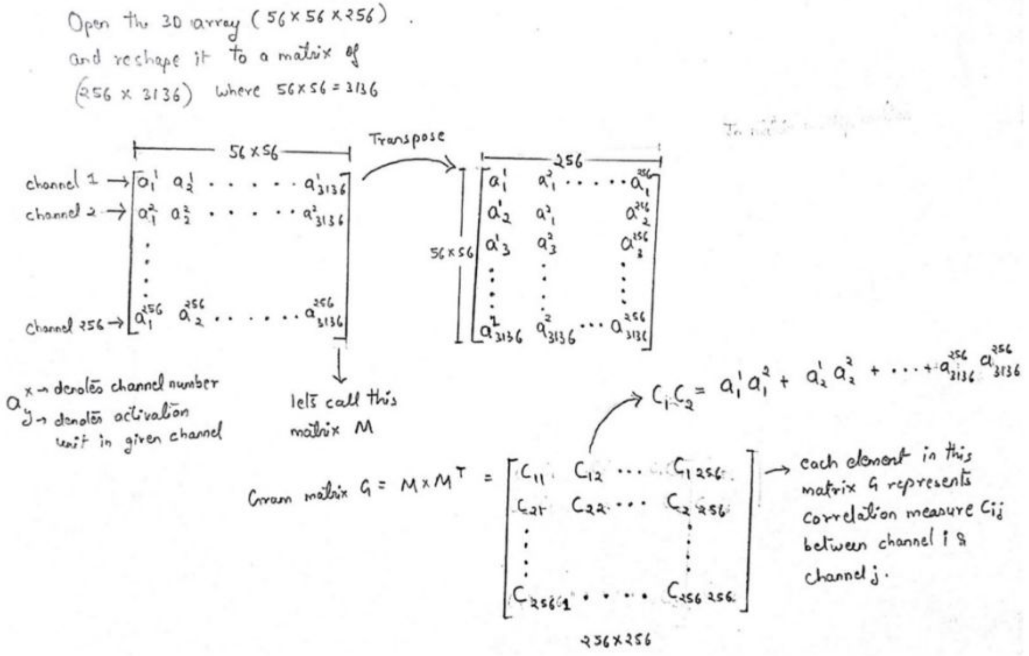
Total style loss across each layer

1. Weight the contribution of each layer’s loss by factor *Wl*
2. Here, we weight each layer equally;

$$L_{style}(a, x) = \sum_{l \in L} w_l E_l \qquad \left(w_l = \frac{1}{|L|}\right)$$

Creating gram matrix

1. Degree of correlation between channels
2. Obtain tensor at particular layer
3. Make image channels
4. Create matrix out of this
5. Create casting



Open the 3D array (56 × 56 × 256) and reshape it to a matrix of (256 × 3136) where 56×56 = 3136

a^x_y x → denotes channel number, y → denotes activation unit in given channel

lets call this matrix M

Gram matrix $G = M \times M^T$

$C_1C_2 = a_1^1 a_1^2 + a_2^1 a_2^2 + \dots + a_{3136}^{256} a_{3136}^{256}$

→ each element in this matrix G represents correlation measure C_{ij} between channel i & channel j.

56 × 56
channel 1 →
channel 2 →
Channel 256 →
a_1^1 a_2^1 … a_{3136}^1
a_1^2 a_2^2 … a_{3136}^2
a_1^{256} a_2^{256} … a_{3136}^{256}
Transpose
256
56 × 56
C_{11} C_{12} … $C_{1\,256}$
C_{21} C_{22} … $C_{2\,256}$
$C_{256\,1}$ … $C_{256\,256}$
256×256

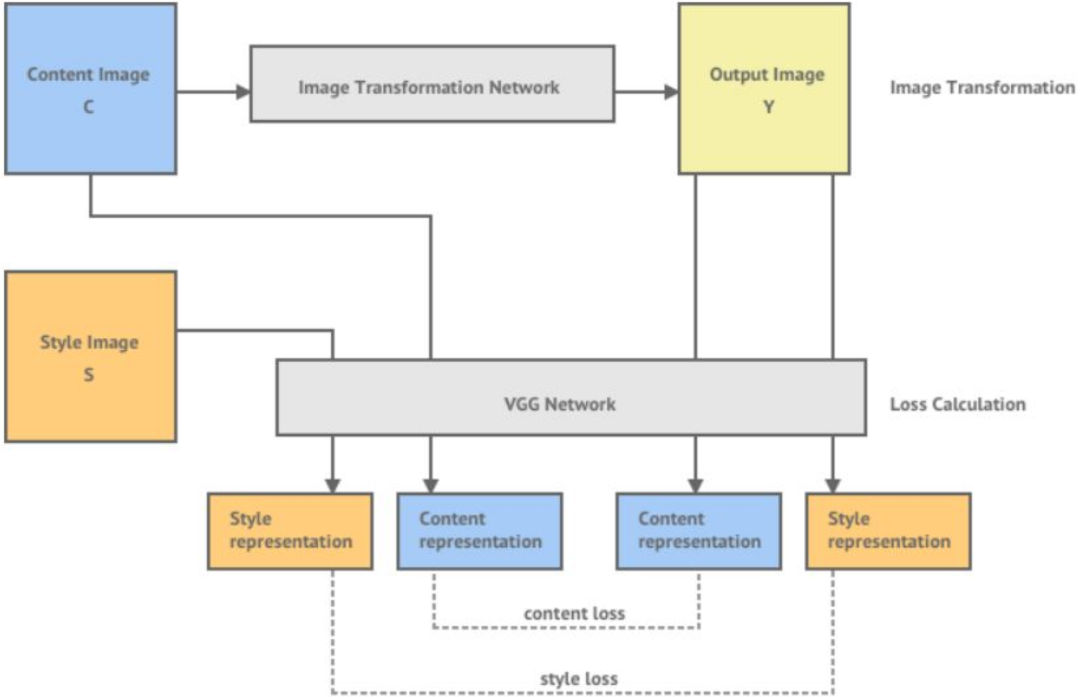
https://towardsdatascience.com/neural-style-transfer-tutorial-part-1-f5cd3315fa7f

Optimizer

1. Adam
2. We don’t update the weights in the network, but we train the input image to minimize the loss
3. Optimizer to minimize loss wrt input parameters

p.s. L-BFGS, which is our recommended. But here, we want to illustrate best practices with eager execution, and, by using Adam, we can demonstrate the autograd/gradient tape functionality with custom training loops

Overview of the Whole Process



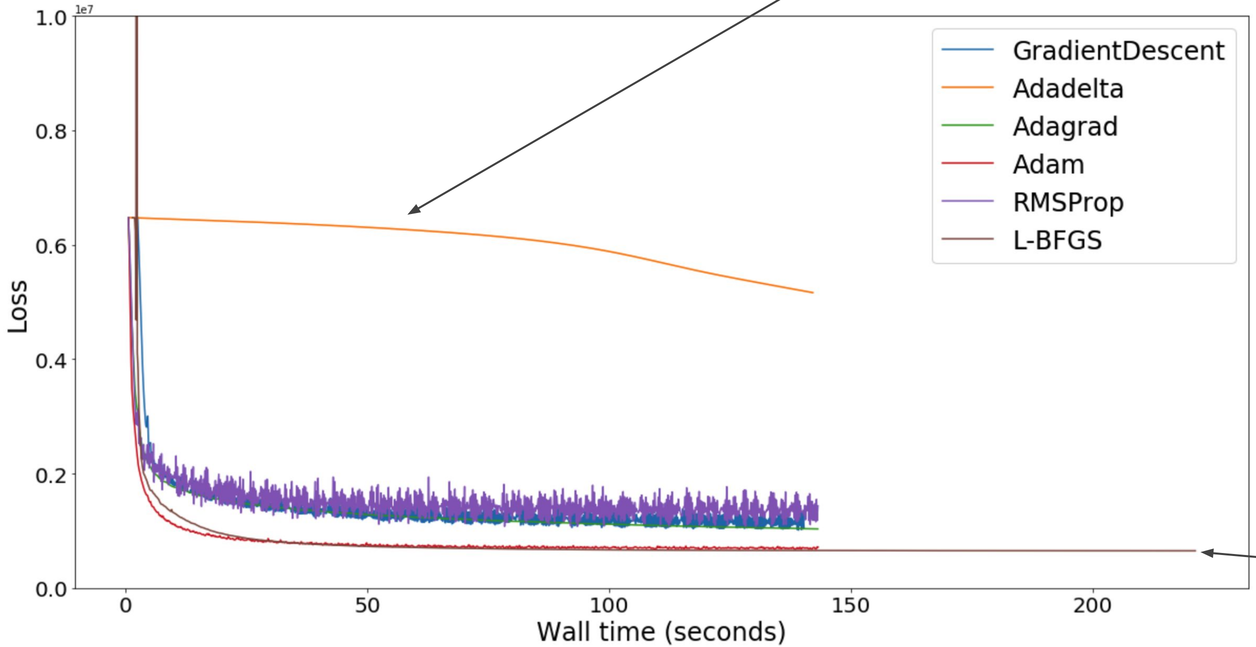
Created model from scratch

- To save the loss results in csv files
- Capture the time taken for training
- Visualize results of all types of optimizers
- Try different parameters like content weight, style weight and learning rate
- Compare the performance when using different content layers and style layers
- Compare the losses when implementing with different models

Content and Style Images

Content and Style Images

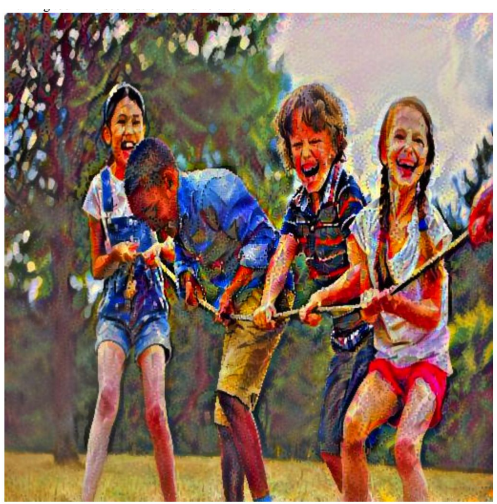
Optimizer results

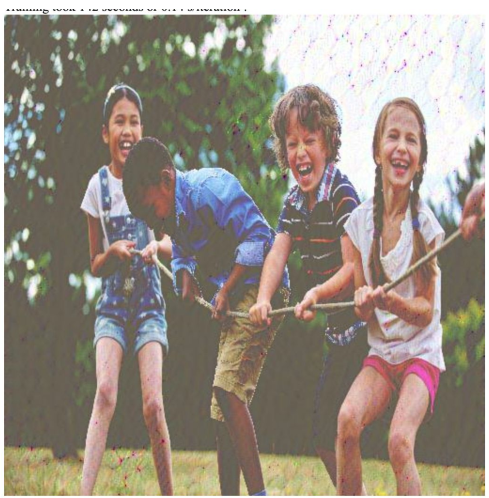


AdaDelta converges slowly, loss is high

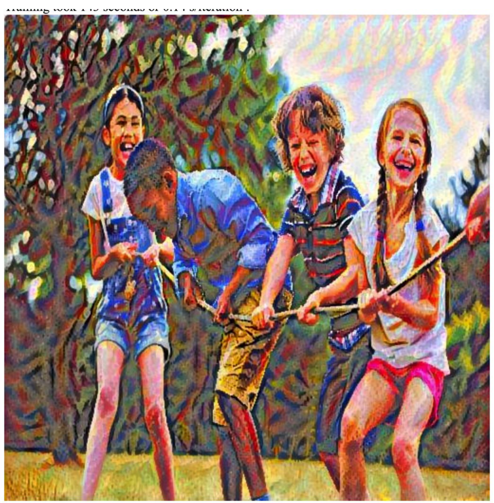
LBFGS takes highest time to converge. Loss is least

Optimizer results





Training took 142 seconds or 0.14 s/iteration .



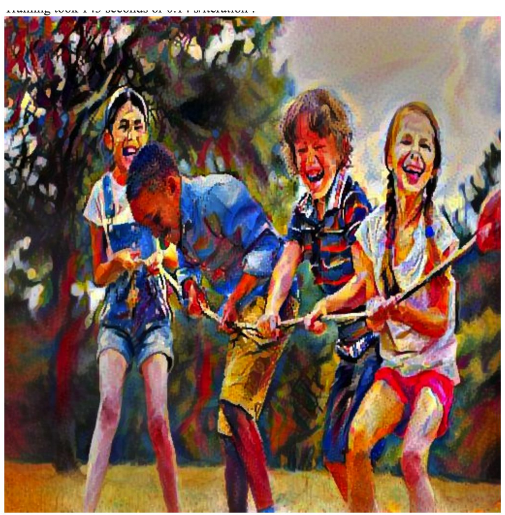
Training took 143 seconds or 0.14 s/iteration .

Gradient descent
Total loss: 1.25 e6

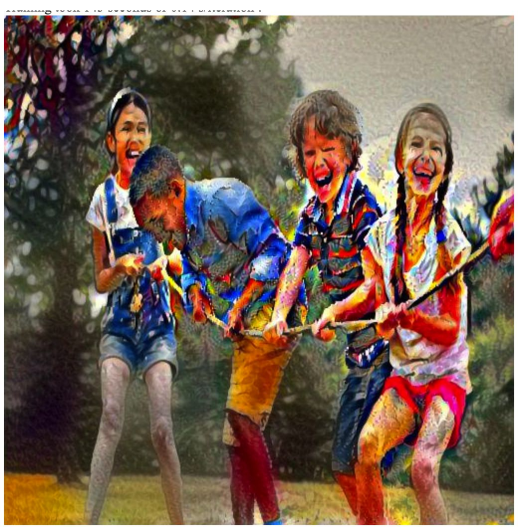
Ada Delta
5.6 e6

Ada Grad
1.08 e6

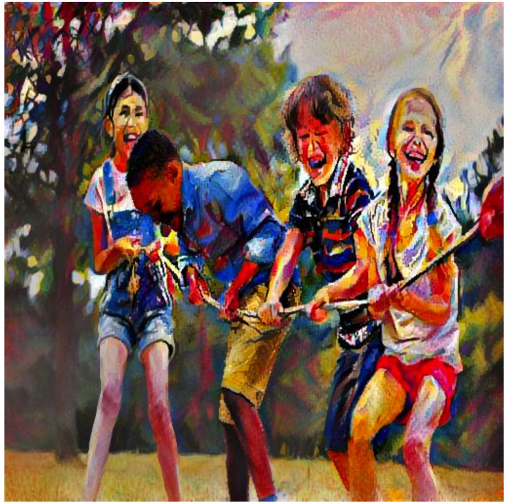
Optimizer results



Adam
Total loss: 7.1 e5



RMSProp
1.3 e6

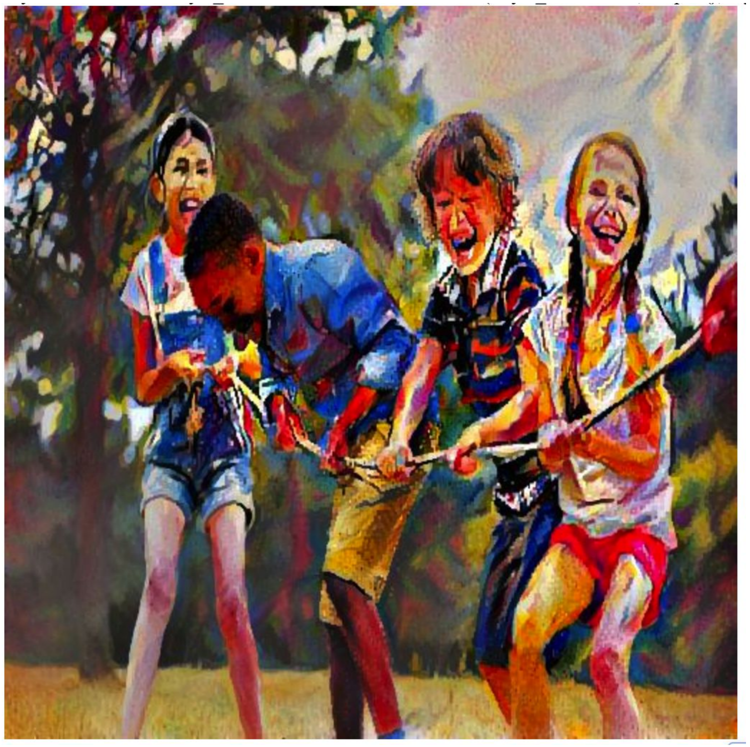


LBFGS
6.5 e5

Block5_conv2

Total loss: 6.5e5

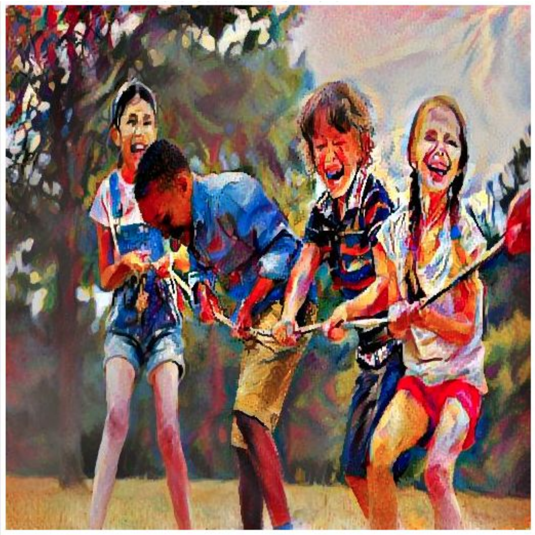
Time: 360s



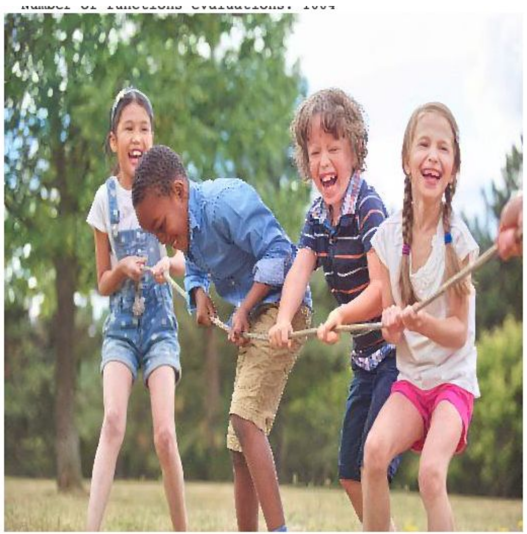
L-BFGS optimizer
Weights (α and β)

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$

Changing weights: outputs



6.00E+04

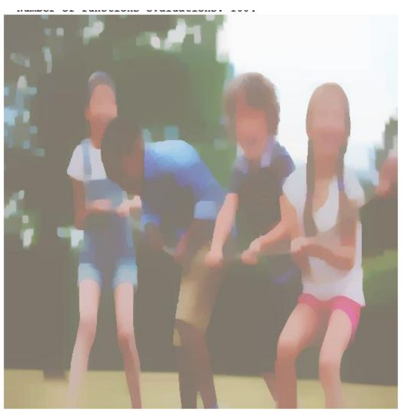


```
Number of functions evaluations: 1004
```

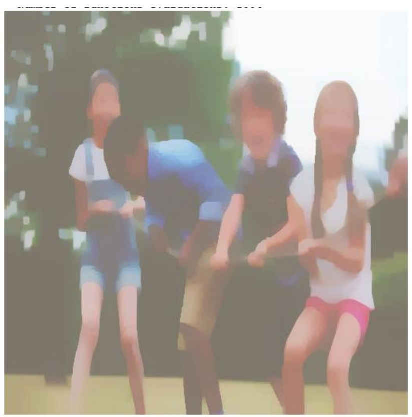
1.40E+05

5.00E+04

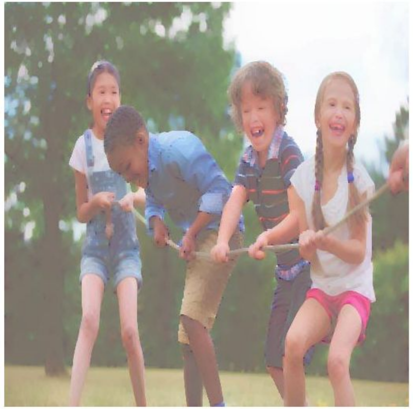
Changing weights: outputs



1.90E+05

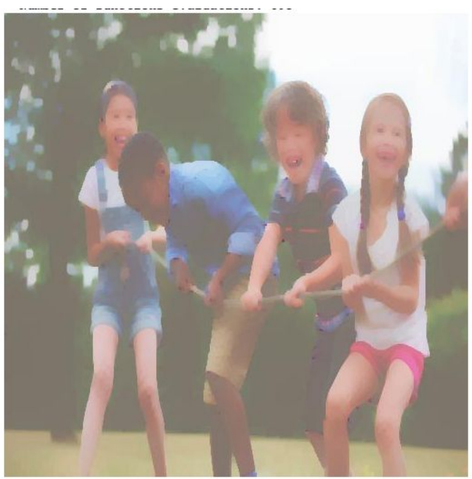


1.90E+04

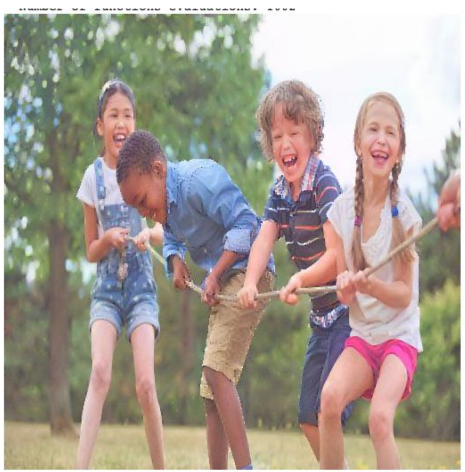


5454

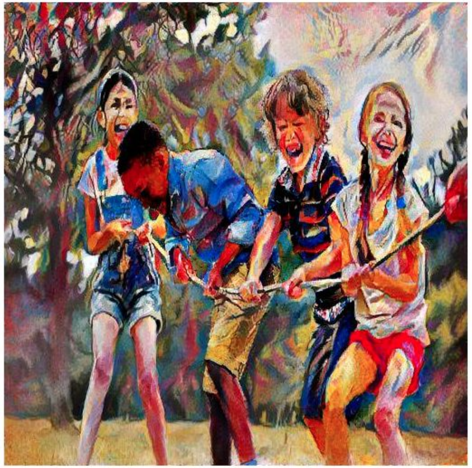
Changing weights: outputs



33095



8435

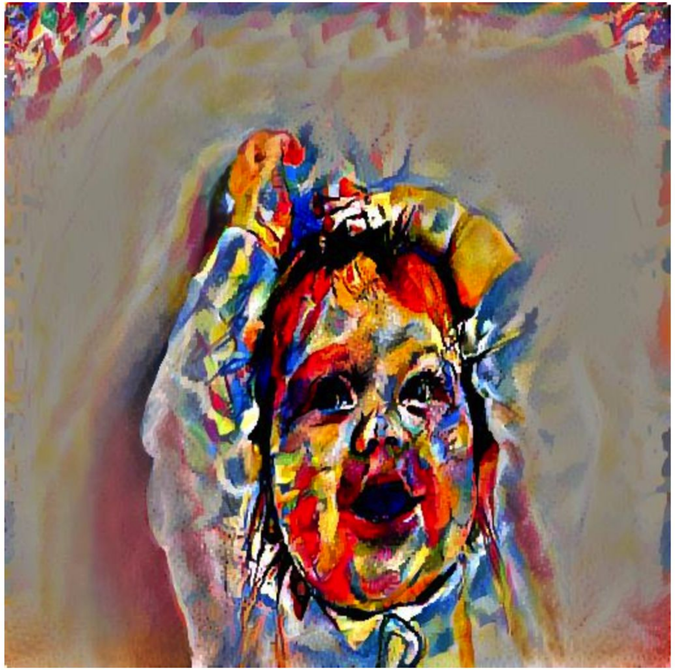


4.80E+05

Faults:

- More weight to style 4e6, less to content 1e3
- Style weight = 40, content weight = 60
- These give bad results

Adding block5_conv2 as content layer - LBFGS



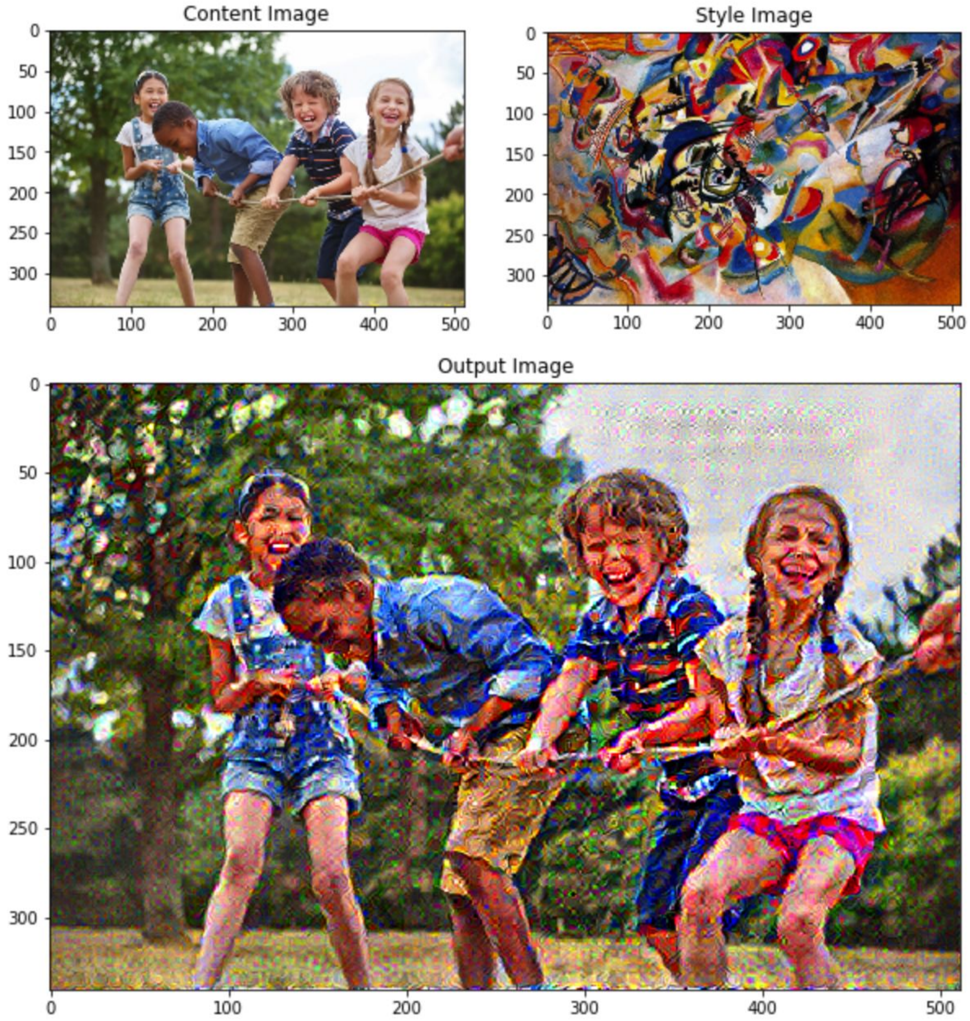
Time:360s Loss:7.7e5

Adding block5_conv2 as content layer - Adam

Adam optimizer

Loss: 9.6e6

Time: 123s



1st inference

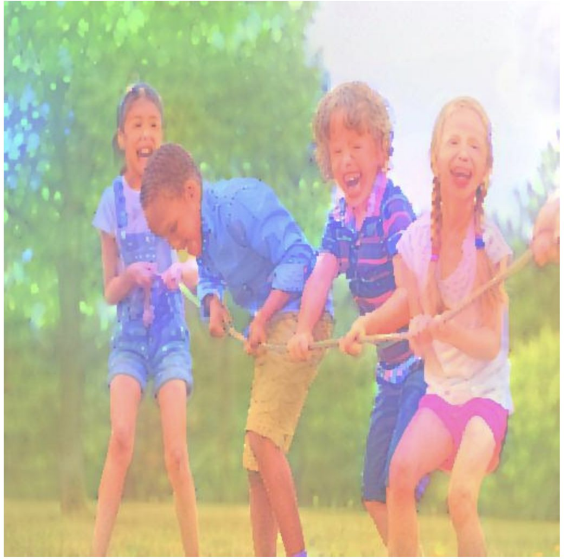
- Adding more layers is increasing model’s overall optimization time
- Keeping the overall loss the same

Adding Total variation loss for smoothing

Total loss: 0.1e5

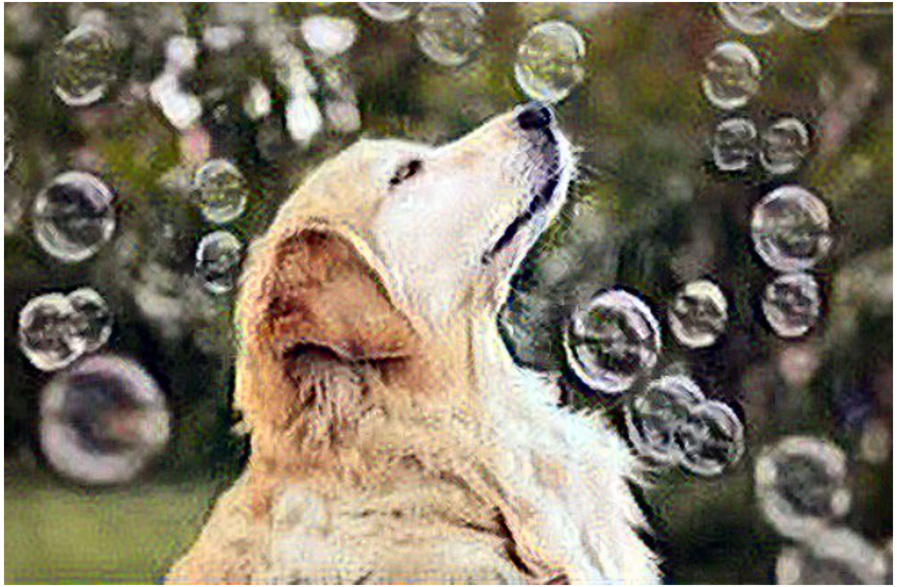
As compared to previous 1.1e6

- Designed to keep image locally coherent
- Supress noise
- Sum of absolute of differences of neighboring pixels



Model 1 VGG_16

- ***Content layers:*** ['block4_conv2']
- ***Style layers:*** ['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1', 'Block5_conv1']

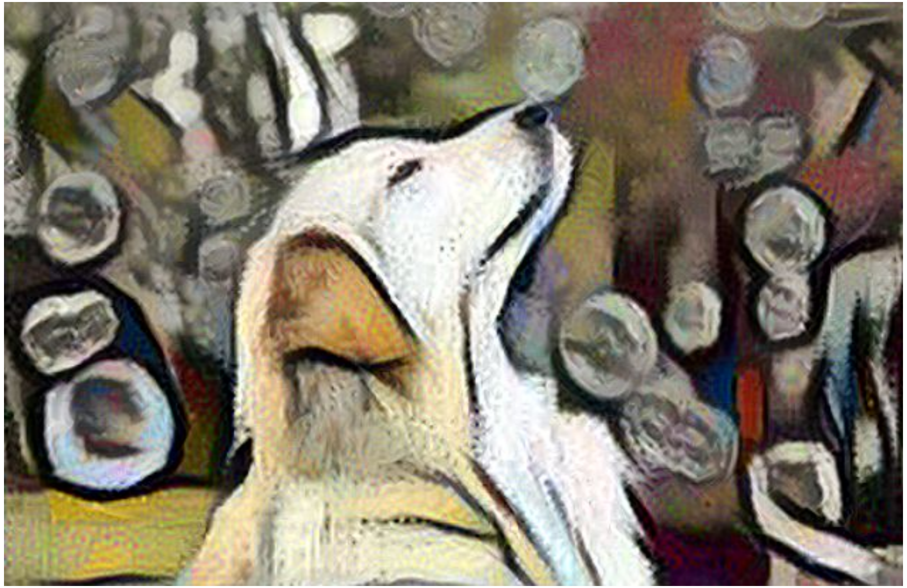


Total loss: 2.5429e+06

Total time: 93.7438s

Model 1 VGG_16

- ***Content layers:***
  ['block5_conv2']
- ***Style layers:***
  ['block1_conv1',
  'block2_conv1',
  'block3_conv1',
  'block4_conv1',
  'Block5_conv1']

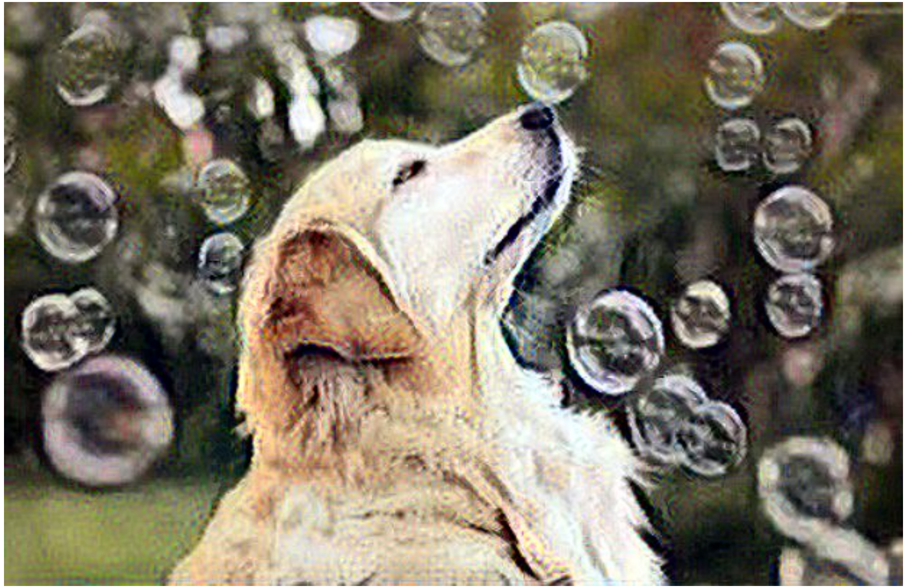


Total loss: 1.2027e+05

Total time: 97.1050s

Model 1 VGG_16

- ***Content layers:*** ['block4_conv2', 'block5_conv2']
- ***Style layers:*** ['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1', 'Block5_conv1']

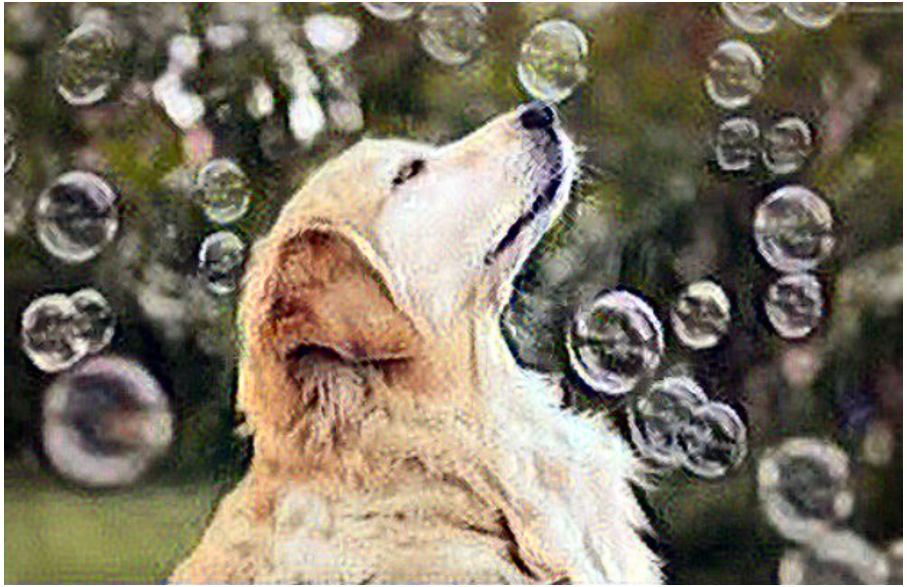


Total loss: 1.9836e+06

Total time: 98.6057s

Model 2 VGG_19

- ***Content layers:*** ['block4_conv2']
- ***Style layers:*** ['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1', 'Block5_conv1']

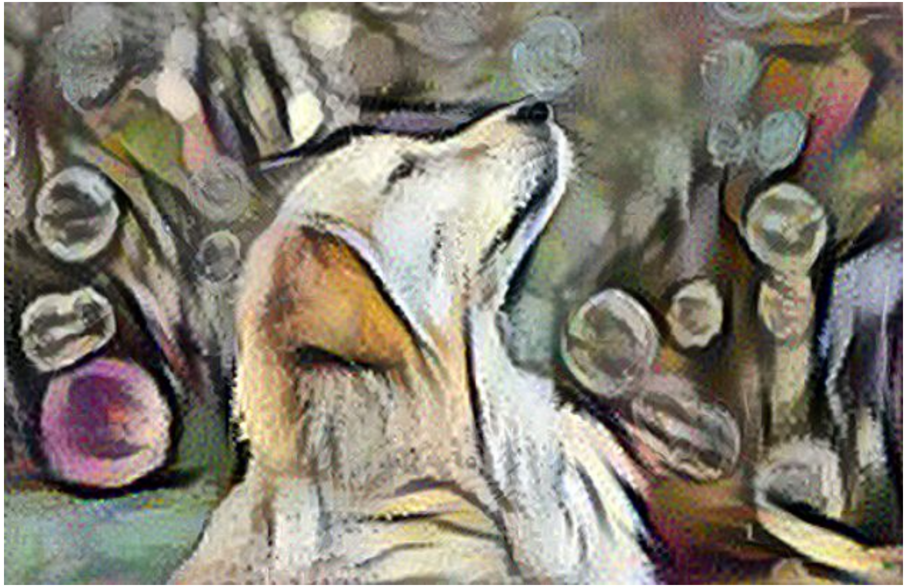


Total loss: 2.5429e+06

Total time: 93.7438s

Model 2 VGG_19

- ***Content layers:***
  `['block5_conv2']`
- ***Style layers:***
  ```
  ['block1_conv1',
  'block2_conv1',
  'block3_conv1',
  'block4_conv1',
  'Block5_conv1']
  ```



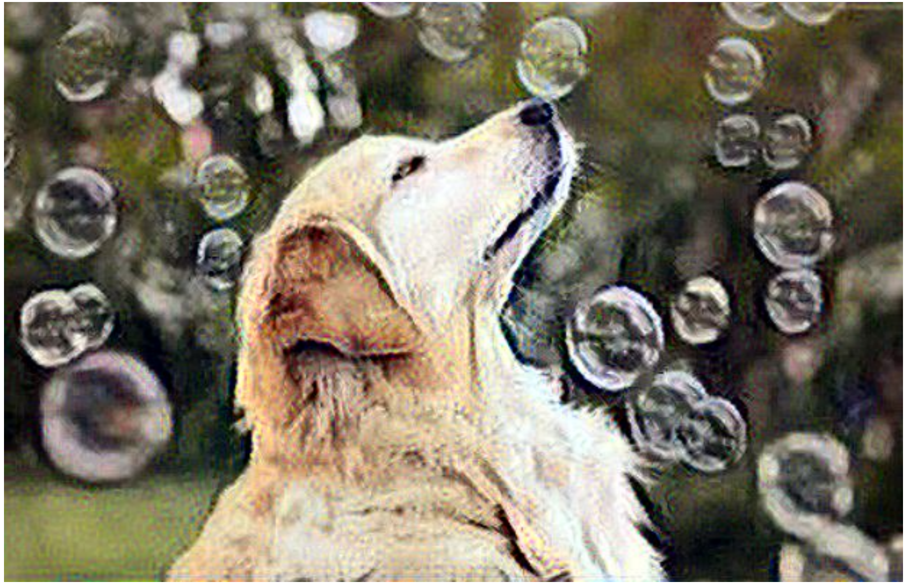
Total loss: 8.8375e+05

Total time: 110.9676s

Model 2 VGG_19

- ***Content layers:***
  `['block4_conv2', 'block5_conv2']`

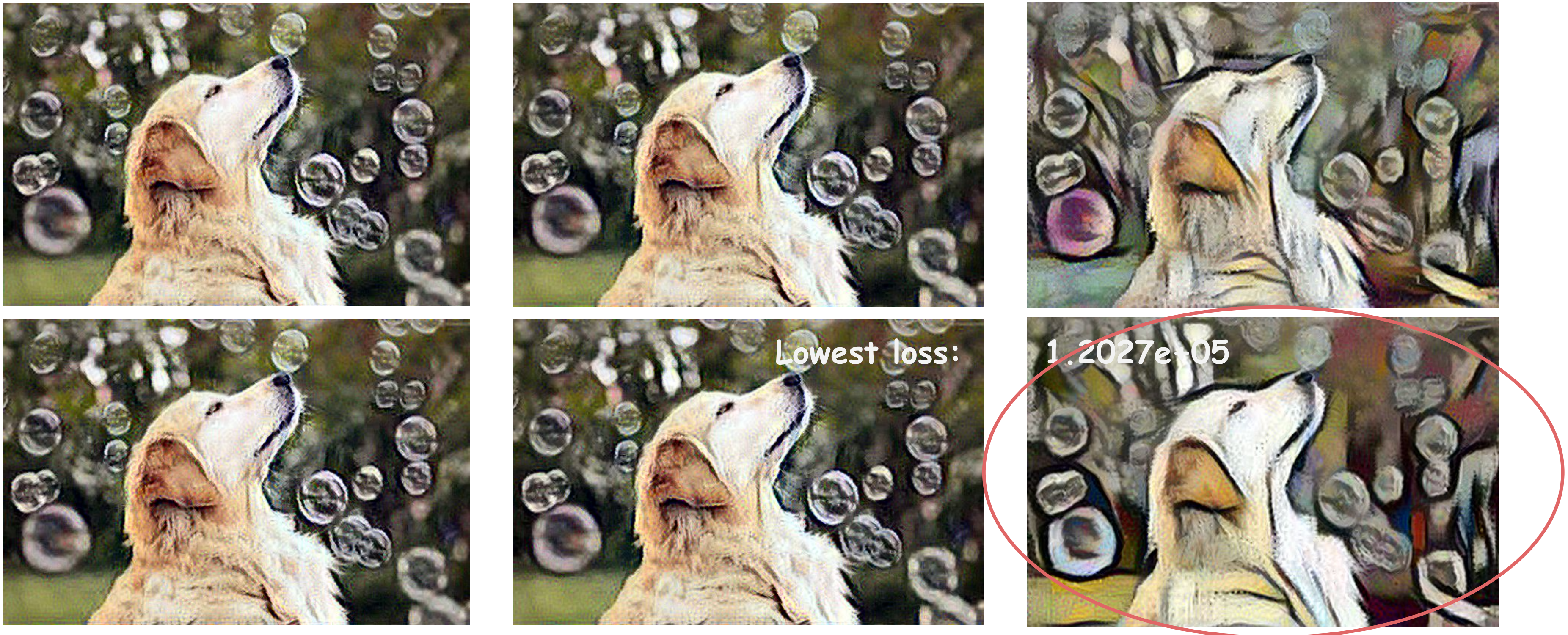
- ***Style layers:***
  `['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1', 'Block5_conv1']`



Total loss: 1.9836e+06

Total time: 98.6057s

Take Away from VGGs



Lowest loss: 1.2027e+05

Take Away from VGGs

VGG's better performance is due to not downsampling aggressively, doing so only after two convolutions and then max pooling.

With VGGs, being relatively shallow and modular and having no residual connections or other special tricks to smuggle raw information up the layers, are forced to create more of a clearcut hierarchical pyramid of abstractions.

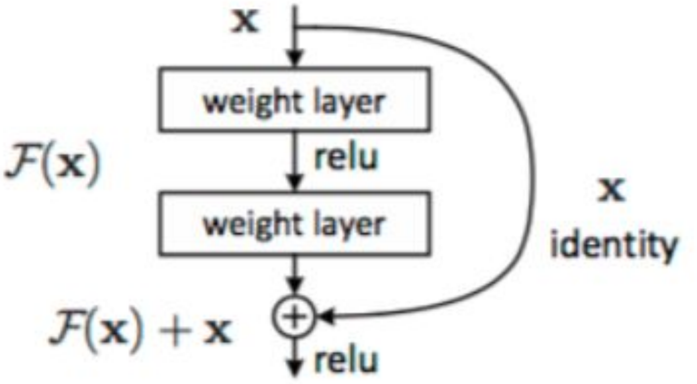
Problems from Fine Tuning VGGs

The problem with training the deeper networks is, performing the optimization on huge parameter space and therefore naively adding the layers leading to higher training error.

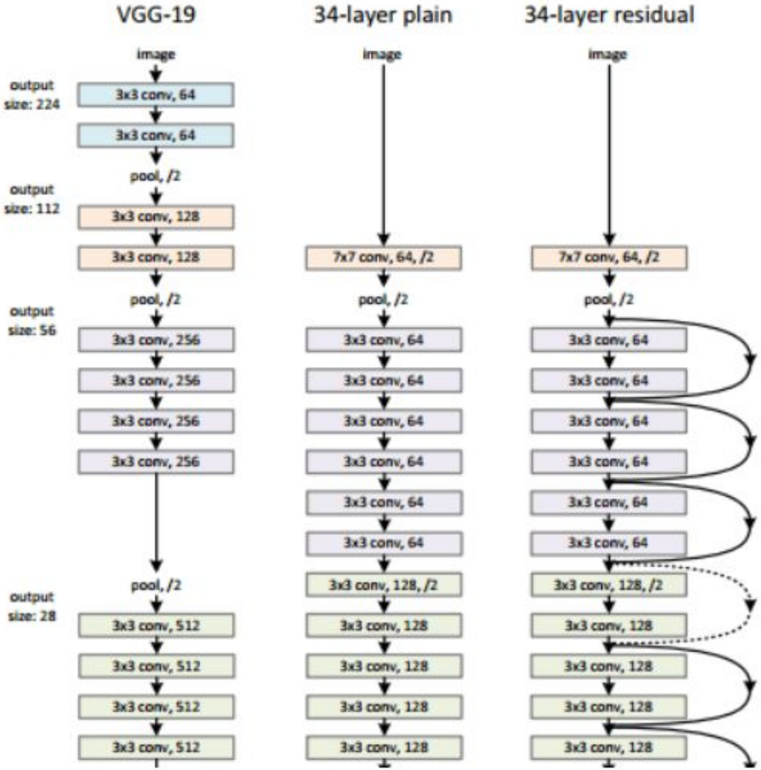
Also the problem with increased depth is that the signal required to change the weights, which arises from the end of the network by comparing ground-truth and prediction becomes very small at the earlier layers, because of increased depth. It essentially means that earlier layers are almost negligible learned. This is called vanishing gradient.

Improvements with Residual Networks

Residual networks allow training of such deep networks by constructing the network through modules called residual models



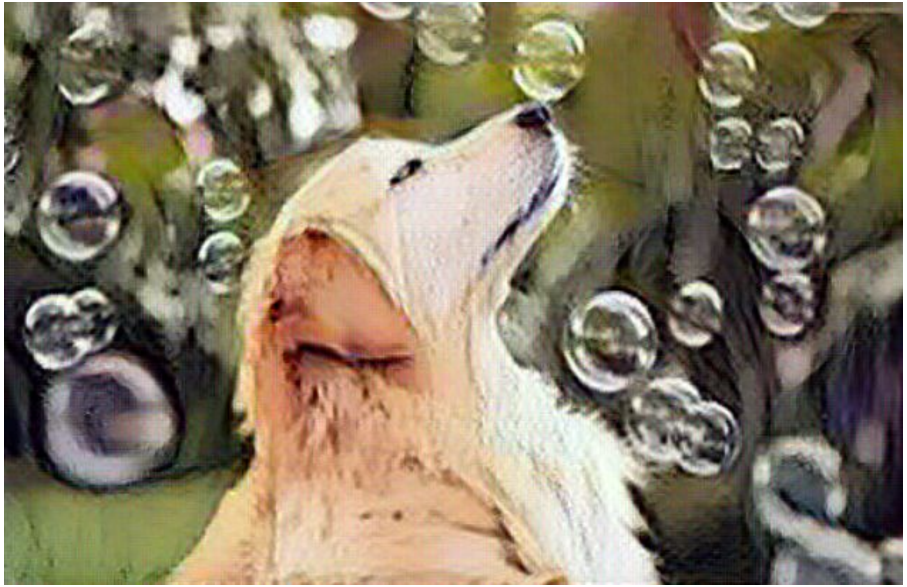
Improvements with Residual Networks



http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006

Model 3 ResNet 50

- ***Content layers:*** `['conv1']`
- ***Style layers:*** `['bn2b_branch2a', 'bn3a_branch2c', 'res3a_branch1', 'bn4e_branch2b', 'conv1']`

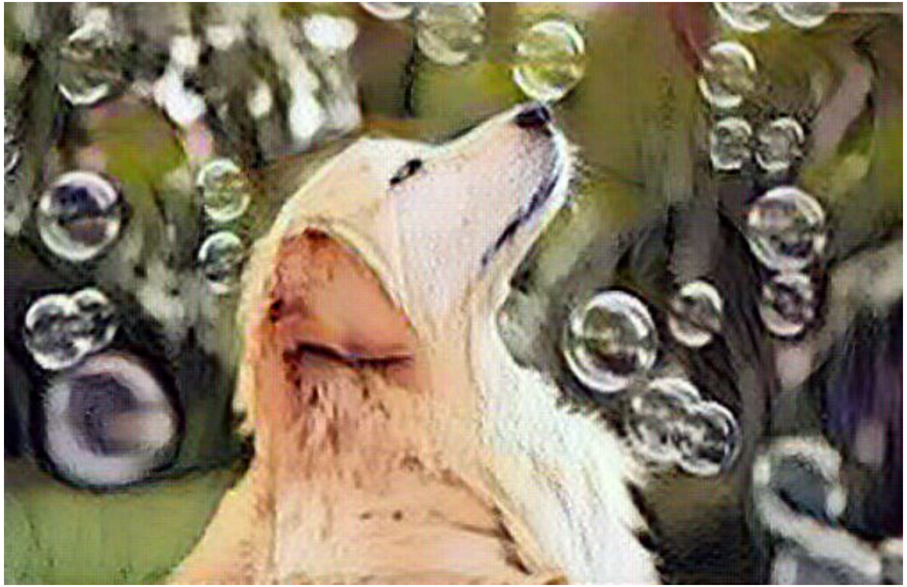


Total loss: 8.0593e+06

Total time: 95.8951s

Model 3 ResNet 50

- ***Content layers:*** ['conv1']
- ***Style layers:*** ['bn2b_branch2a', 'bn3a_branch2c', 'res3a_branch1', 'bn4e_branch2b']

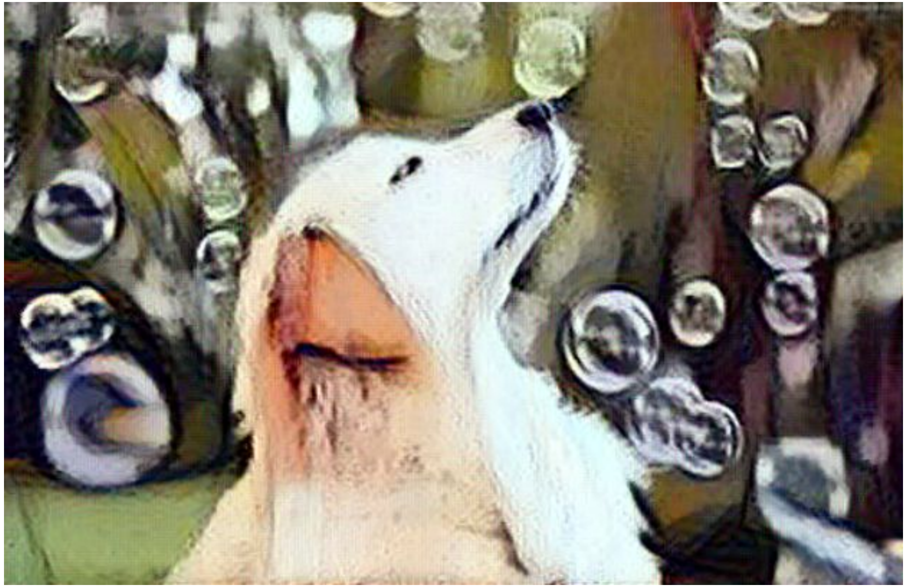


Total loss: 2.4149e+03

Total time: 86.8160s

Model 3 ResNet 50

- ***Content layers:*** `['conv1']`
- ***Style layers:*** `['bn_conv1', 'bn2a_branch2a', 'bn3a_branch2a', 'bn4a_branch2b', 'bn5a_branch2a']`

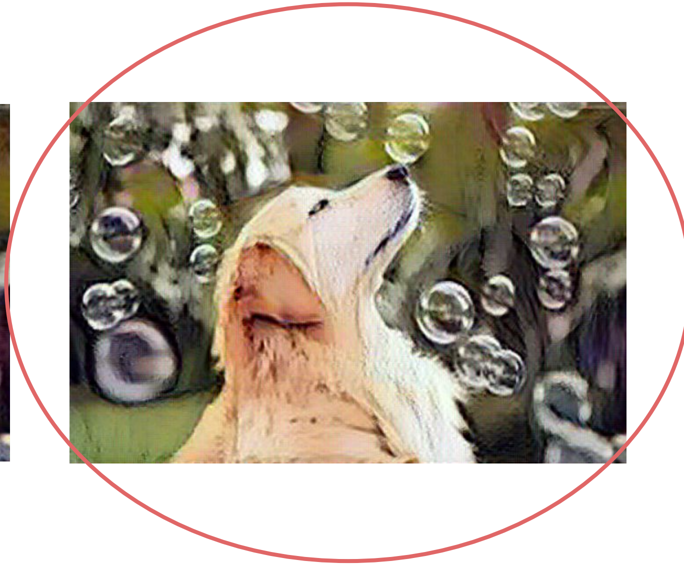


Total loss: 4.4162e+03

Total time: 92.8777s

Take Away from Residual Network

Lowest loss: 2.4149e+03



Take Away from Residual Network

Since our layer doesn’t need to learn how to take our input and generate a new output, but instead it just needs to learn how to adjust our input to produce our output.

So residual layers work for us because we are just trying to learn the difference, or residual, between our input and our desired output.

Take Away from Residual Network

NOT every situation works

Resnets are shallow ensembles: the features do exist but they are too spread out to be pulled out easily and the levels of abstraction are all mixed up, instead of getting a nice balance of features from the bottom and top

Residual connections themselves somehow mess up the optimization procedure by affecting properties like independence of features

Take Away from Residual Network

Our Recommendation

It would be helpful for using residual layers in a network, for certain tasks it makes it easier to train the network because it has less work to do.

For a task like style transfer using residual layers makes sense as we know that our generated output image should be somewhat similar to our input image so we can help the network by only requiring it to learn the residual.

Take Away from Residual Network

Our Recommendation

It would be helpful for using residual layers in a network, for certain tasks it makes it easier to train the network because it has less work to do.

For a task like style transfer using residual layers makes sense as we know that our generated output image should be somewhat similar to our input image so we can help the network by only requiring it to learn the residual.

Future Work

Inspired from Apple Application -- Turi Create

The model is compact and fast and hence can run on mobile devices like an iPhone. The model consists of 3 convolutional layers, 5 residual layers (2 convolutional layers in each) and 3 upsampling layers each followed by a convolutional layer.

- Combined with multiple styles
- Generate fast style transfer
- Play with 3D images
- ......

References

1. https://github.com/tensorflow/docs/blob/master/site/en/r2/tutorials/generative/style_transfer.ipynb
2. https://shafeentejani.github.io/2017-01-03/fast-style-transfer/
3. https://cs.stanford.edu/people/jcjohns/eccv16/

Questions