



AKADEMIA GÓRNICZO-HUTNICZA W KRAKOWIE

WYDZIAŁ FIZYKI I INFORMATYKI STOSOWANEJ

Style Transfer with Deep Learning

Deep Learning With CUDA: Final Project

*Krzysztof Konieczny
Andrzej Świątek
Jan Kwiatkowski*

Contents

1	Introduction	2
2	Code	2
3	Requirements	2
4	Script Interface	2
5	Overview	3
6	Parameter description and explanation	3
6.1	Number of iterations	4
6.2	Adam vs lbfgs	5
6.3	Content and style weight	6
6.4	Total variation loss	7
6.5	Init method	8
6.5.1	Random init	8
6.5.2	Style init	9
7	Performance and comparison	10
8	Rooms to improvement	10

1. Introduction

This project will explore NST - Neural Style Transfer, a technique that using deep learning, allows one to merge or infuse the artistic style of one image into another. Using convolutional neural networks, we can effectively separate and later recombine both the style and content of images. The overall goal of this project was to implement a neural style transfer system. Then we proceeded with tuning, which allowed us to examine how each of the various hyperparameters - such as no. of iterations, content & style weights - affect the results.

2. Code

The code is available on github: <https://github.com/Andrzej-Swietek/Deep-Learning-With-CUDA-Project/tree/main>. The project contains also GUI, backend services and docker-compose file to run them locally, as well as in CLI using ‘python’ command but this document focuses only on NST transfer script.

3. Requirements

These libraries are needed locally to run the NST python script:

- `torch` and `torchvision`
- `opencv-python-headless`
- `matplotlib`
- `pillow`
- `numpy`

4. Script Interface

```
--content-image-path Specifies the path to the content image, which serves as the base for the style transfer.  
--style-image-path Specifies the path to the style image, which provides the artistic style to be transferred.  
--content-weight The weight for the content loss. A higher value keeps the generated image closer to the content image. The default is 5.  
--style-weight The weight for the style loss. A higher value emphasizes the style of the style image. The default is 1.  
--tv-weight Sets the weight for the total variation loss, which helps reduce noise and smooth out the generated image. The default is 1.0.  
--optimizer Selects the optimizer to use for the style transfer. Supports lbfsgs and adam. The default is lbfsgs.  
--model Chooses the pre-trained model for feature extraction. Supports vgg16 and vgg19. The default is vgg19.  
--init-method Specifies how the initial generated image is created: random for Gaussian noise, content to start with the content image, or style to start with the style image. The default is content.  
--total-iterations Sets the total number of iterations. The default is 1000 for lbfsgs and 3000 for Adam.
```

```
--learning-rate Learning rate if Adam optimizer was chosen. The default is 1.0.
--height Defines the height of the input images. The width is adjusted automatically to maintain
the original aspect ratio. The default is 400.
--output-file Specifies the folder for saving the optimized output image. The default is output_image.
--saving-interval Iteration interval between saving images to the given folder. The default is 100.
```

5. Overview

The following section showcases examples of the neural style transfer process. The figures below demonstrate the results that we've achieved by combining the original (Fig. 3) image with several different styles. More examples are available in our github repository <https://github.com/Andrzej-Swietek/Deep-Learning-With-CUDA-Project/tree/main/data> in output.



Figure 1: Style 1



Figure 2: Effect 1



Figure 3: Original



Figure 4: Style 2



Figure 5: Effect 2



Figure 6: Style 3



Figure 7: Effect 3

6. Parameter description and explanation

All examples in this section were generated with 800 iterations and default parameters unless stated otherwise.

6.1. Number of iterations



Figure 8: original image



Figure 9: 10 iterations



Figure 10: 20 iterations



Figure 11: 40 iterations



Figure 12: 80 iterations



Figure 13: 120 iteration



Figure 14: 200 iterations



Figure 15: 300 iterations

In the figures shown, one can observe the effect of varying the no. of iterations on the output image. We can notice that the most significant stylistic changes occurred within the first few iterations (Fig. 9 through 12). As the iterations increased, especially above 120, the differences became largely unnoticeable.

This suggests, that after a certain point, the process exhibits diminishing returns.

6.2. Adam vs lbfgs



Figure 16: Adam 500 iterations



Figure 17: Lbfgs 100 iterations



Figure 18: Adam 1000 iterations



Figure 19: Lbfgs 300 iterations



Figure 20: Adam 2000 iterations



Figure 21: Lbfgs 600 iterations

Comparing the Adam and L-BFGS optimizers, it's clear that L-BFGS achieves similar (if not better) visual results while also requiring fewer iterations than Adam. For example, L-BFGS at 100 iterations (Fig. 17) is comparable to Adam @ 500 (Fig. 16). This demonstrates, that L-BFGS is more efficient and converges faster, while also producing smoother results.

Whether one performed better than the other (artistically) is a topic of subjective nature.

6.3. Content and style weight

Content and style weight are parameters that balance the output image's resemblance to the original content and the desired style. Content weight controls how much the output image retains the original content's structure and details. A higher content weight emphasizes preserving the original image. Style weight determines the degree to which the output image adopts the textures, colors, and patterns of the style image. A higher style weight results in a stronger style effect.

What matters in practice is the difference between those two parameters, so we can decrease the content weight to amplify the style or increase it to preserve the original content.



Figure 22: 10e7 content weight



Figure 23: 10e6 content weight



Figure 24: 10e5 content weight



Figure 25: 10e4 content weight



Figure 26: 10e3 content weight



Figure 27: 100 content weight

High

6.4. Total variation loss

Total variation loss (TV loss) is a hyperparameter that controls the degree of smoothness in the output image. Below is an example of how it affects the output image.



Figure 28: 0 TV loss



Figure 29: 0.01 TV loss



Figure 30: 0.1 TV loss



Figure 31: 0.2 TV loss



Figure 32: 0.5 TV loss



Figure 33: 1 TV loss

Without this loss (Fig. 28), the image appears grainy and noisy. As the loss increases, the images become smoother (Fig. 30 & 31). At high values (Fig. 32 & 33) the images become distorted and lose important details.

After experimentation, the sweet spot appears to be around 0.1 - 0.2.

6.5. Init method

The initialization method determines which image will be used as the initial input. While it is natural to use the content image, other options are also possible. It is recommended to set a high content weight and a low style weight; otherwise, the output may not resemble the content image. The following two examples were generated with a content weight of 10,000 and a style weight of 0.01.

6.5.1. Random init

Uses Gaussian noise as an initializer

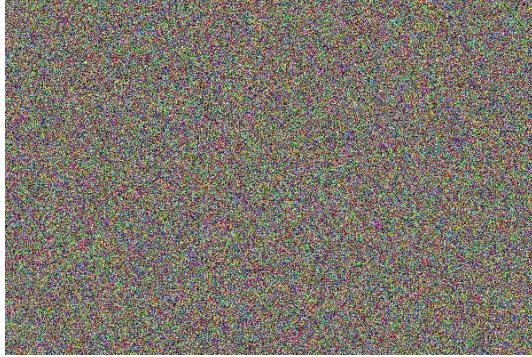


Figure 34: initial image

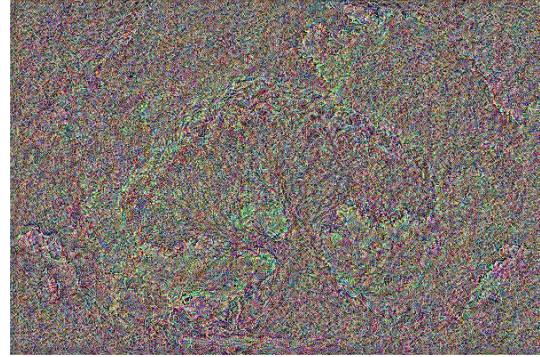


Figure 35: 50 iterations

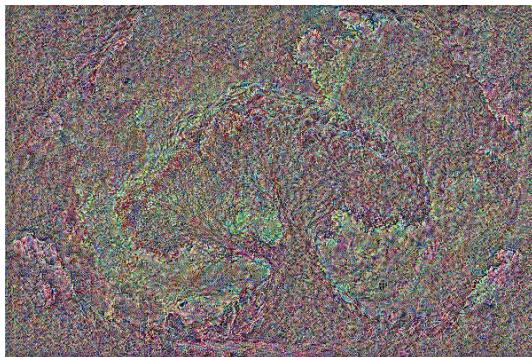


Figure 36: 100 iterations

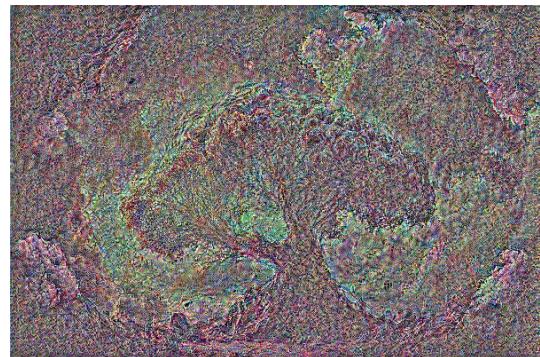


Figure 37: 200 iterations

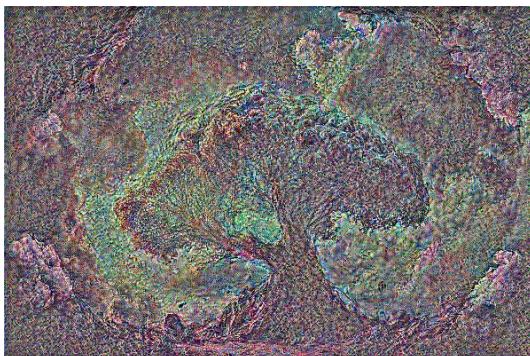


Figure 38: 400 iterations

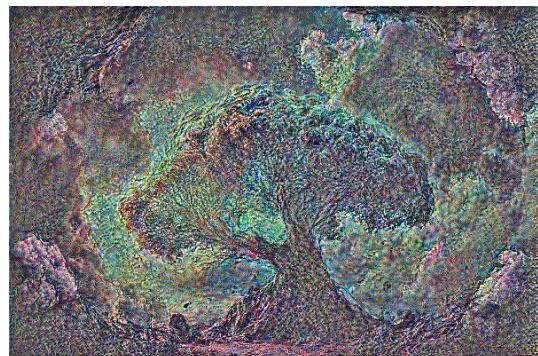


Figure 39: 800 iterations

Using random initialization, the output image begins as a random noise. Initially, it has little to no resemblance to either content or style image. As the process progresses, the random noise gets gradually adjusted to fit the structure of the content image.

While this approach might allow the network to explore a wide range of possibilities, it also leads to slow convergence. This isn't surprising, as the network starts from scratch.

6.5.2. Style init

Uses style image as an initializer.



Figure 40: initial image



Figure 41: 50 iterations



Figure 42: 100 iterations



Figure 43: 200 iterations

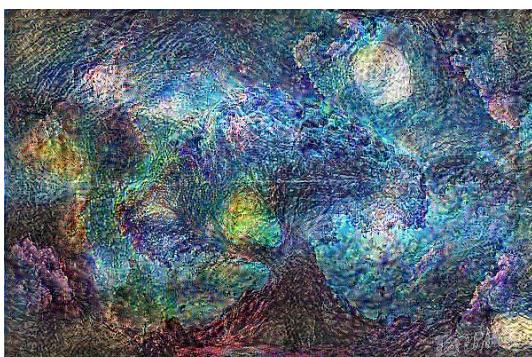


Figure 44: 400 iterations



Figure 45: 800 iterations

Using style initialization, the generated image starts with the *style* image itself. One can notice, that in the beginning there are little changes being made. As the iterations ensue, we see that the *content* image gets gradually overlaid.

Overall, this has led to faster convergence, while also achieving nicer stylistic results.

7. Performance and comparison

We will use [neural-style-transfer](#), an open-source Python implementation of neural style transfer, for comparison. The examples below were generated after 800 iterations. The TV loss was set to 0 for the open-source solution because the default value was causing distortions.

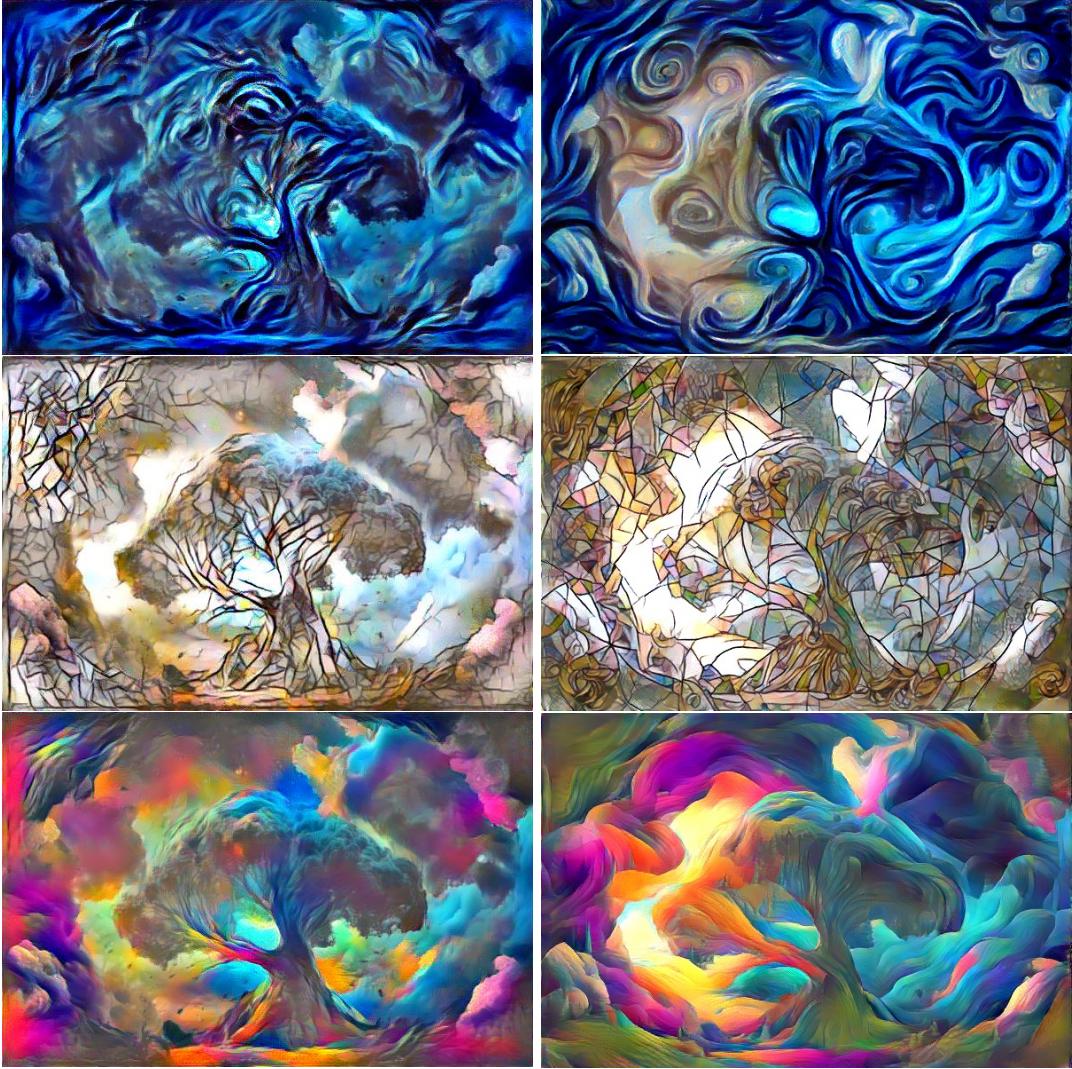


Figure 46: open source NST

Figure 47: our NST

Our NST implementation performs quite well compared to the open-source solution. 800 iterations take 73 seconds for our style transfer, while they take 100 seconds for the Python library.

8. Rooms to improvement

- Content weight and style weight balance – Compared to the open-source library, our model does not handle these parameters effectively; they have too little impact on the output image. In some cases, selecting values that are too high or too low causes the script to crash, depending on the input images.
- Support for custom models – It should be possible to use a custom image recognition network for feature extraction.

- Support for preserving original colors – It should be possible to transfer the style to the content image without altering its original colors.