Dobrik Georgiev

Design Choices in Neural Style Transfer

Diploma in Computer Science

Fitzwilliam College

2019

Declaration of Originality

I, Dobrik Georgiev of Fitzwilliam College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

Proforma

Candidate number:	**2338A**
Project Title:	**Design Choices in Neural Style Transfer**
Examination:	**Diploma in Computer Science, July 2019**
Word Count:	**10810**
Final line count	**1887**
Project Originator:	Simeon Spasov
Supervisor:	Simeon Spasov

Original Aims of the Project

First, the project aims to reproduce a state of the art approach (Dumoulin et al., 2016) to arbitrary style transfer. Then, the focus moves to how various changes to the way the style transfer network is trained affects the results obtained. Due to the nature of the problem, the evaluation will be primarily qualitative, but results will be also supported using quantitative empirical evidence whenever suitable, such as the rate of learning of a network. The implementation should be carefully engineered, so that switching between various experiments can be performed with ease.

Work Completed

Overall, the project was a great success, meeting and exceeding all requirements. Various models were trained and evaluated. Results from Dumoulin et al. (2016) were reproduced and a number of design choices were investigated (network architectures, distance metrics, etc.). In addition to the qualitative results, the evaluation also included quantitative measurements, when visual inspection was not enough to draw a conclusion. Employing modularity and sticking to high coding standards led to concise and easy to use/maintain implementation. Additionally, optimisations for faster training on a GPU were included for shorter training times.

Special Difficulties

None.

Contents

Chapter 1

Introduction

A pastiche is a work of visual art which imitates the style of another artwork. Style transfer in the computer vision and machine learning domains is the process of automating pastiche synthesis. Creating a pastiche by hand can take a real artist from a few hours to a few weeks depending on the technique of the art. Recent breakthroughs in Artificial Intelligence, and in particular Deep Learning have managed to reduce this process to a few minutes (or even a few seconds) through measuring and optimising the degree of transfer by using the layer's activations of a trained classifier.

1.1 Motivation

Style transfer has already found its applications in photo editing software products, such as Photoshop[1] or Prisma[2]. Apart from being eye-catching, style transfer can also be used for data augmentation to prevent overfitting, as it preserves the semantic content of the image, whilst altering colour and texture.

My primary goal in this project is to recreate a state of the art arbitrary style transfer network as described in Dumoulin et al. (2016), which can synthesise a pastiche in more than one style. Further to this, since any recent publications on neural style transfer, that have come to my knowledge, use the same trained classifier (VGG) for the training setup, I will evaluate how using different classifiers affects the results obtained. The motivation behind this is that there have not been any published results which have been obtained using a different classifier architecture. On the other hand, there exist neural networks that have achieved better performance than VGG on specific tasks such as object recognition. I believe it is interesting to evaluate the extent to which other architectures are suitable for the task of style transfer.

The rest of my work aims to analyse how other modifications in the training procedure, which have been presented in various publications, but have considered only the transfer of a single style, would influence the multi-style transfer network from Dumoulin et al. (2016).

[1] e.g. `https://petapixel.com/2017/03/29/cornelladobe-show-copy-color-lighting-one-photo-another/`

[2] `https://prisma-ai.com/`

1.2 Related work

In the field of computer vision, colour transfer has existed for more than a decade (Reinhard et al., 2001). When applying the transfer between an artwork and a real image, the generated imitation replicates only the colour, lacking any texture.

A few years ago, a novel approach (Gatys et al., 2015) was presented, which uses the activations of a trained classifier to synthesise a pastiche. A lot of follow-up research has been conducted since then, leading to improvements or extensions of the original algorithm. A (very) high-level classification of the various techniques is as follows:

- **Image optimisation** – These are methods similar to Gatys et al. (2015). Such approaches generate the pastiche by directly optimising the pixel values of the image. The optimisation procedure has to be repeated for every pastiche generated, which makes such procedures quite computationally expensive and less applicable in reality. Nonetheless, the approach is very flexible (i.e. applicable for any style) and the results obtained are usually more appealing in visual quality, compared to other schemes.
- **Model optimisation, Single style** – Such techniques train a model to synthesise a pastiche, given a content image, in one forward pass of the network. Although significantly faster, trained models are fixed to a single style. An example of such method is Johnson et al. (2016).
- **Model optimisation, Many styles** – This is the category where Dumoulin et al. (2016) falls into. Trained style transfer networks posses the speed of the single-style networks and some of the flexibility of synthesising a pastiche directly.

Jing et al. (2017) provides a more comprehensive and in-depth overview of all neural style transfer algorithms.

1.3 Overview of achievements

In this chapter I gave my motivation for this project and presented any related work that has been achieved so far. My achievements/contributions in this project are as follows:

- *Successfully reimplemented a state of art arbitrary style transfer network* (Dumoulin et al., 2016).
- Engineered and implemented a *flexible* learning framework, which can support different trained classifiers, distance metrics, etc.
- Implemented *efficient* methods for calculating learning losses.
- Evaluated a *variety* of design choices in the learning framework both qualitatively and *quantitatively*.
- *Revealed new* results, to the best of my knowledge, on using different trained classifiers as feature extractors.

Chapter 2

Preparation

The aim of this chapter is to present a theoretical overview of the work implemented in this project and then specify the requirements that should be accomplished. Software engineering aspects necessary for the project's success are discussed in the second part of the chapter. Finally, the **starting point** is stated.

2.1 Theory

This section gives a concise and self-contained overview of the theory behind style transfer. It also gives other relevant background, needed for understanding other parts of the project.

2.1.1 Convolutions in deep learning

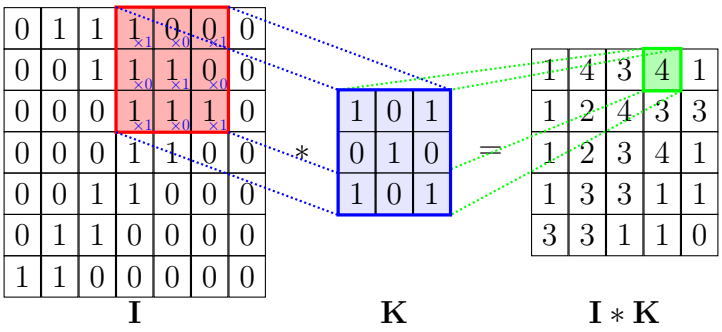


Figure 1: Result of convolving image **I** with kernel **K**

In computer vision, convolution is used as a method for extracting useful local features, such as edges of an image. The operation consists of sliding a kernel (a small, real-valued matrix) over an image and pointwise multiplying the pixel values as shown in Figure 1. Every element in the output image is the sum of the products.

In order to transfer this methodology to the convolutional networks domain, we organise the neurons in convolutional layers of 3 dimensions: depth, width and height. Width and height

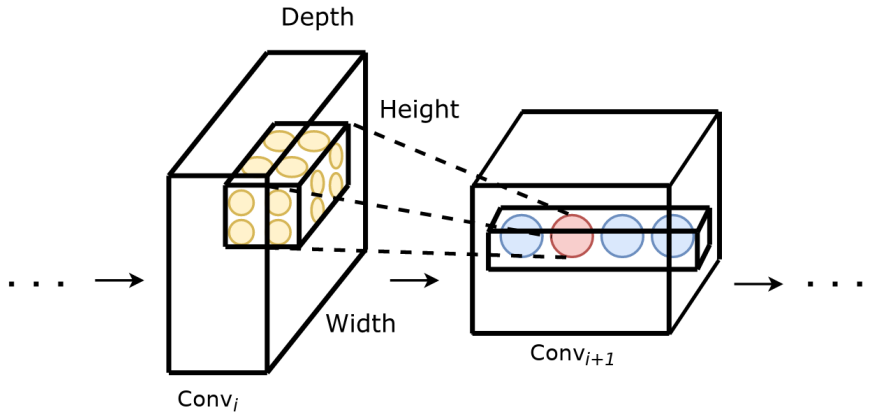


Figure 2: Applying a kernel to a network layer (image courtesy of Momchil Peychev). The neurons, covered by an example kernel are given in yellow. The result of applying the kernel is a single neuron (denoted with red in the example). Every neuron in the $\texttt{Conv}_{i+1}$ layer will therefore be connected to a subset of the $\texttt{Conv}_i$ layer's neurons. Depending on the dimensionality of the kernel, this will span some local region of the Height×Width plane, but it will always cover all channels of $\texttt{Conv}_i$.

can be interpreted as rescaled dimensions of the original image. Depth corresponds to the number of channels that carry some information. E.g. for neural networks operating on RGB images, the first layer would have 3 channels for the red, green and blue colours respectively.

The convolution operation on layers is presented in Figure 2. If the dimensionality of the i-th convolutional layer (denoted with $\texttt{Conv}_i$) is $d_i \times w_i \times h_i$, a kernel can cover only a small area of the $w_i \times h_i$ plane, but it will span *all* of the d_i channels. A different kernel is used for each channel in $\texttt{Conv}_{i+1}$. E.g. if the i-th layer has $d_i = 3$ dimensions and $\texttt{Conv}_{i+1}$ has $d_{i+1} = 8$, then there will be 8 different $3 \times m \times n$[1] kernels, one for each output channel. The values of the neurons in a channel of layer $i+1$ are obtained in the same fashion as in Figure 1 – sliding the kernel, pointwise multiplying the elements and taking the sum.

The values of the output of convolutional layer $\texttt{Conv}_i$ are often referred to as the 'activations' or 'features' of that convolutional layer. Throughout the rest of this and the following chapters, I will adhere to this terminology.

2.1.2 Using deep learning for style transfer

Style transfer can be interpreted as the optimisation problem of finding a pastiche p which has a similar content to that of a content image c and style similar to that of a style image s. In neural style transfer, the following similarity definitions are used:

- The content similarity between two images is the Euclidean distance between the high-level features extracted by a trained classifier. The intuition is that there is a hierarchical separation between layers based on the depth, i.e. the depth of a layer is proportional

[1] m and n – width and height dimensions of the kernel

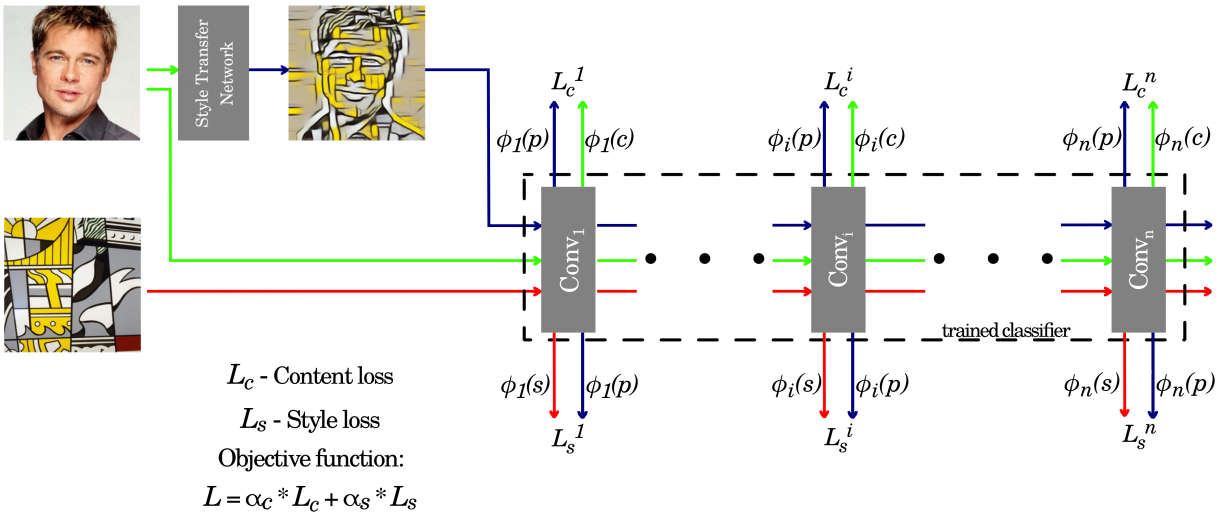


Figure 3: The architecture of the learning framework. The style transfer network synthesizes a pastiche from the content image. The three images (content image, style image and pastiche) are input through the trained classifier. At a layer i a style loss term L_s^i is formed by comparing the Gram matrices of the hidden layer activations of the style image $\phi_i(s)$ and the generated image $\phi_i(p)$. The content loss term L_c^i compares the activations of the content image $\phi_i(c)$ and the pastiche $\phi_i(p)$. For the total style loss L_s the sum of the shallow layers' style loss terms is taken. For the total content loss L_c the sum of deeper layer terms is taken.

to the level of abstraction this layer is capable of. E.g. a deeper layer activations would contain more information about what the object looked like, rather than what were the colours of the pixels.

- Two images are similar in style if the difference between the Gram matrices of the low-level features as extracted by the trained classifier has a low Frobenius norm. Texture can be captured by aligning lower order statistics and the artistic style of a painting can be thought of as a visual texture. Lower order statistics refer to functions which use the first or second power of a sample[2].

Figure 3 presents the structure of the learning framework. The algorithm for optimising a network T to perform style transfer follows Dumoulin et al. (2016): T takes as an input the content image c and outputs a pastiche $p = T(c)$ in the style of s. Then c, p, s are input through the trained classifier to extract the image features. The content and style losses at a layer i are defined as:

$$L_c^i(p) = \frac{1}{U_i} \|\phi_i(p) - \phi_i(c)\|_2^2 \tag{2.1}$$

$$L_s^i(p) = \frac{1}{U_i} \|G(\phi_i(p)) - G(\phi_i(s))\|_F^2 \tag{2.2}$$

where $\phi_i(x)$ are the activations at layer i, U_i are the total number of units at layer i and $G(\phi_i(x))$

[2]e.g. mean or variance

denotes the features' Gram matrix. A Gram matrix of a set of N vectors $\mathcal{X} = \{v_1, v_2, \ldots, v_n\}$ is:

$$\begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ N \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \ldots & N \end{array} \\ \begin{bmatrix} \langle v_1, v_1 \rangle & \langle v_1, v_2 \rangle & \ldots & \langle v_1, v_n \rangle \\ \langle v_1, v_1 \rangle & \langle v_2, v_2 \rangle & \ldots & \langle v_2, v_n \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle v_n, v_1 \rangle & \langle v_n, v_2 \rangle & \ldots & \langle v_n, v_n \rangle \end{bmatrix} \end{array} \tag{2.3}$$

where $\langle v_i, v_j \rangle$ denotes the inner product between v_i and v_j.

The total content and style losses are defined as:

$$L_c(p) = \sum_{l \in \mathcal{C}} L_c^l(p) \tag{2.4}$$

$$L_s(p) = \sum_{l \in \mathcal{S}} L_s^l(p) \tag{2.5}$$

$\mathcal{C}$ and $\mathcal{S}$ are the sets of 'content layers' and 'style layers', respectively. As mentioned above, $\mathcal{S}$ usually includes shallow layers and $\mathcal{C}$ includes some (quite often a single one) of the deeper layers. For comparability with Dumoulin et al. (2016), throughout my project I have used $\mathcal{C} = \{conv_3\}$ and $\mathcal{S} = \{conv_1, conv_2, conv_3, conv_4\}$ and VGG-19[3], unless stated otherwise.

To train a network to synthesise a pastiche of style s given a content image c the following function is minimised:

$$L(s, c) = \alpha_s L_s(T(c)) + \alpha_c L_c(T(c)) \tag{2.6}$$

and α_s and α_c are scaling parameters. The training procedure is repeated with many different content images. Optimisation is achieved via a variant of gradient descent such as Adam (Kingma and Ba, 2015).

2.1.3 Conditional instance normalisation

Training a network to generate imitations is computationally less expensive than synthesising the pastiche directly. However, there is a trade-off – the network T is tied to a single artwork and a separate network would need to be trained for every style we want to imitate.

In their seminal paper Dumoulin et al. (2016) hypothesised that many styles have a degree of similarity. Paintings of some artistic movement (e.g. impressionism) are similar in their techniques (brushstrokes, materials used, etc.) but differ only in the colour palette used. It is therefore not efficient to treat all impressionists' paintings as separate styles and train a new model for every painting.

The solution proposed by Dumoulin et al. (2016) was to train a single network $T(c, s)$ which takes both a content and a style image to produce a pastiche. This is achieved by employing *conditional instance normalisation* (Figure 4). This normalisation technique transforms a layer's

[3]The initial proposal mentioned VGG-16, but this was changed for comparability.

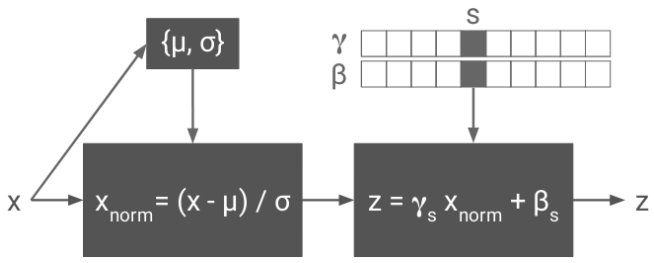


Figure 4: Conditional instance normalisation (Dumoulin et al., 2016). The input activations x are normalised and then scaled and shifted using parameters specific to style s – γ_s and β_s.

activations x by normalising them into style specific ones z:

$$z = \gamma_s(\frac{x - \mu}{\sigma}) + \beta_s \tag{2.7}$$

where μ and σ are the mean and standard deviation of x, and γ_s and β_s are parameters unique to style s.

With this approach all but the scaling and shifting parameters of the normalisations are shared across styles. According to the paper, this comprises only 0.2% of the model parameters which makes the network very memory efficient, whilst still being comparable to single-style networks in quality of transfer. The approach also works for artworks which vary greatly in colour or style used, not just paintings of one field of art.

2.1.4 Kernels

Usually, Gram matrices are constructed by calculating the inner product in the space of the vectors (i.e. calculating the dot product). No modifications or projections to other dimensions are applied to the vectors. As stated in §2.1.2 aligning the Gram matrices captures only the lower order statistics. By transforming the data into another higher dimension, more features could become more separable and more complex correlations and statistics could be learned. To do this, there are two possibilities:

- define an explicit mapping function. For example, if after vectorising the features they are of the form (a, b), a projection function could be:

$$\mathcal{P}(\mathbf{x}) = (a, b, a^2 + b^2) \tag{2.8}$$

- calculate the inner product *implicitly* by using a *kernel*. A kernel is a function $\kappa : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ which takes two vectors x and $y \in \mathcal{X}$ and returns the inner product of their images in a higher dimension:

$$\kappa(\mathbf{x}, \mathbf{y}) = \langle \mathcal{P}(\mathbf{x}), \mathcal{P}(\mathbf{y}) \rangle \tag{2.9}$$

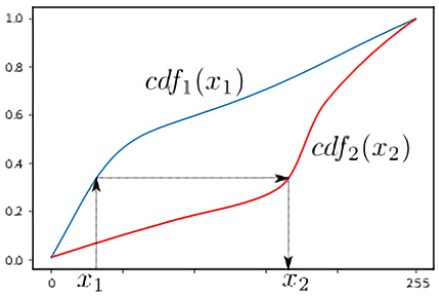


Figure 5: An example of histogram matching between two probability distributions (source: Wikipedia).

Since only the value of the inner product is a real value, the mapping does not need to be explicit[4]. We can use any function for κ as long as it has has the following properties:

1. *Symmetry* – $\kappa(\mathbf{x}, \mathbf{y}) = \kappa(\mathbf{y}, \mathbf{x})$
2. *Positive Semi-Definiteness* – The matrix formed by applying κ to any finite subset of $\mathcal{X}$ is positive and semi-definite

The kernels used in this project are:

- Linear kernel – $\kappa(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\mathbf{y}$. This is equivalent to the inner product in the current vector space and is the kernel used in Dumoulin et al. (2016).
- Polynomial kernel – $\kappa(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T\mathbf{y} + c)^d$
- Radial basis function kernel – $\kappa(\mathbf{x}, \mathbf{y}) = exp(-\frac{\|\mathbf{x}-\mathbf{y}\|_2^2}{2\sigma^2})$. This kernel calculates the inner product of the two vectors in an infinite dimensional space.

2.1.5 Histogram Matching

In image processing, histogram matching is used to transform an image so that its histogram is as close as possible to a specified one. The histogram matching algorithm can be used not only for images, but also for any two probability distributions. It is presented in Figure 5. The values of the two distributions are allocated into histograms and the cumulative distribution functions of these histograms are computed – cdf_1 and cdf_2. Then we map each histogram value x_1 of the first distribution to the histogram value x_2 of the second distribution, such that $cdf_1(x_1) = cdf_2(x_2)$.

In style transfer, histogram matching can be used for matching the distributions of the pastiche and style image in place of aligning Gram matrices.

[4]In fact, even the dimensionality of the other dimension is unnecessary

2.2 Requirements analysis

After a successful review of all the theory needed to understand the concepts presented in the previous section, I could focus on the goals of my project, as defined in the project proposal[5]. The goals of the project are to implement a **pastiche generator neural network** and a **style transfer learning framework** and evaluate how different adjustments (trained classifier, style layer used, kernel used, etc.) affect the generated images. To successfully perform all experiments, the following deliverables are required:

Requirement	**Priority**	**Difficulty**
Implementing Gram matrix calculation	High	High
Content and style loss modules	High	Low
Learning framework	High	Medium
Conditional instance normalisation and style transfer network	High	Low
Histogram matching	Low	Medium

The only reason that the Gram matrix calculation has a high difficulty is that a simple naïve approach to calculating it is too slow to be useful. This is discussed in detail in §3.3, where alternative GPU-optimised methods are presented.

2.3 Development workflow

This section describes the tools/hardware used and the software engineering techniques undertaken for the successful completion of the project.

2.3.1 Hardware

The following hardware was used throughout the project:

- My own ASUS ROG Strix GL703GS (NVidia GeForce GTX 1070, Ubuntu 18.04LTS, 16GB RAM) for the majority of the implementation, part of the testing, unit testing and dissertation writeup.
- NVidia Titan X (Pascal) GPUs (Ubuntu 16.04 LTS, 12 GB RAM), to which I was given access from the Computational Biology group at the Computer Laboratory, for the rest of the testing and evaluation.

2.3.2 Tools and backup strategy

Python was chosen as the main (and only) programming language for the implementation of the project. It is the most common language used by the deep learning

[5]Attached in the end of the document

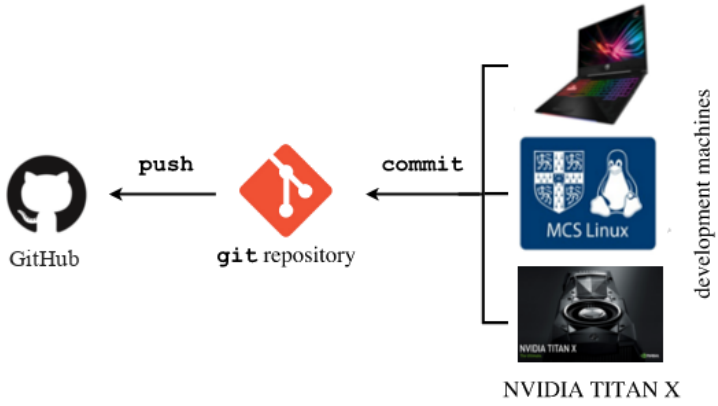


Figure 6: The backup measures taken

research community. It provides a great number of libraries that I could make good use of and its scripting nature allowed for automating the network training and evaluation processes.

PyTorch was used for the implementation of the neural network and everything related to it. PyTorch is more beginner friendly than TensorFlow and more intuitive to learn. Unlike TensorFlow it is completely Pythonic: there is no other computational model. Moreover, PyTorch adopts a *dynamic computational graph*, which means that there is no need to explicitly define how network weights are updated – this is automatically performed at runtime.

PyTorch is widely used in industry and some of the notable companies using it are Apple and Siemens[6]. It is an extremely well documented and stable deep learning framework, making it an ideal choice for this project.

R was used for visualising obtained experimental data. Compared to Python, R allows for faster aggregation of the data and easier displaying of the results.

Git was used for version control to ease development. There were two main reasons to use version control. First, parts of the project were developed independently, e.g. the code for the histogram matching was independent of the code for the Gram matrix. Second, great part of the evaluation was performed on a remote server and occasional corrections were made in the source code residing on the server. Synchronising the source code between various machines was another inconvenience that was easily resolved through version control.

Special care was taken to prevent unexpected hardware and/or software failures. Regular backups to a remote repository hosted by GitHub were made. In addition to that, a copy of the repository was stored (and regularly up-

[6]See `https://discovery.hgdata.com/product/pytorch`

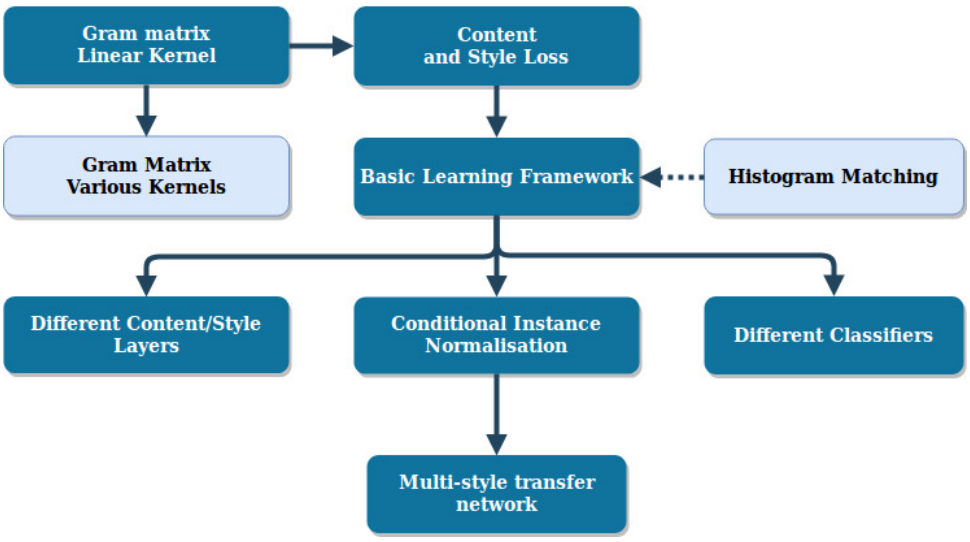


Figure 7: The project organisation and workflow. If there is an arrow between two boxes, then the first box should be implemented fully before any work is started on the second box. The extensions are marked with boxes in light colours. Since the learning framework does not depend truly on histogram matching (but histogram matching is used there), a dashed line is used.

dated) on the university filesystem (MCS Linux). The whole backup strategy is summarised in Figure 6.

LaTeX was used to typeset the dissertation and present the work completed.

2.3.3 Implementation approach

Before any development was done, it was crucial to outline the different dependencies in the project. Figure 7 shows how the different aspects of the implementation rely on each other, which directly stems from the theory described in §2.1 and the deliverables mentioned. As I am new to deep learning I decided to employ the Iterative Development Model. The functionality was gradually developed and evaluated in iterations, analysing the current state of the project on each step. This way, the risks were minimised, allowing for enough time for reaction in case if a theory or implementation poses an unpredicted challenge.

For example, given the dependencies shown above, after implementing core parts up to the basic learning framework (one that uses VGG and a fixed set of style and content layers), I ensured that a (satisfactory) pastiche can be *directly* synthesised through pixel value optimisation, as in the experiments of Gatys et al. (2015).

The code itself was modularised and logically organised, allowing for independent implementation. I adhered to good software practices, following Google's Python Style Guide[7].

[7] `https://google.github.io/styleguide/pyguide.html`

2.4 Artworks used

In the initial proposal, a dataset of images from WikiArt was suggested. It turned out that such a large amount of works was unnecessary, due to the time it takes to train a model. Therefore, for comparability, I focused on two datasets of images, used in Dumoulin et al. (2016). More details on all datasets used is given in the start of the Evaluation chapter.

2.5 Starting point

Before undertaking this project, I was aware of the following theories/concepts/languages or had access to the following codes:

1. Theoretical deep learning and neural networks through the material presented in the Part IB Artificial Intelligence course.

2. Basic experience with Python, but no experience with PyTorch or similar.

3. Access to an open-source implementation of the network presented in Dumoulin et al. (2016)[8]. However, PyTorch and Tensorflow are greatly different in their structure, so I was not able to use the code directly.

2.6 Summary

In this chapter, I have discussed the theory needed to understand the concepts of style transfer. I presented concise and standalone introduction to style transfer as well as an overview of other methods, related to the project. I outlined the deliverables of the project, as well as the tools and software engineering techniques used to achieve these goals.

The next chapter provides an insight of the implementation, what problems arose and how they were tackled.

[8] `https://github.com/tensorflow/magenta/tree/master/magenta/models/arbitrary_image_stylization`

Chapter 3

Implementation

This chapter presents how the requirements from the previous chapter were accomplished. Due to the nature of the PyTorch framework and through employing good software engineering, the final code was concise and easy to maintain. It was well modulated, which allowed for the various experiments to be performed with ease, and it was well optimised, so that the waiting time remains minimised. The focus of this chapter is mainly on how the core parts of the learning framework were implemented, and how the framework itself was designed, but other aspects of the work are also discussed.

3.1 Baseline solution

The idea behind the baseline solution is that humans recognise objects (and faces) by relying primarily on information about the shape of the object and its edges. Therefore, by swapping the gradient of the style image with the gradient of the content image, we may obtain an image which has similar edges to the content image, but preserves the colour for the style image.

Algorithm 1: Performing 'style transfer' using image gradients

input : `content_image`, `style_image`
output: `pastiche`

```
def extract_image_gradient(image): # Uses a Laplacian kernel
    # Uses the OpenCV (Bradski, 2000) package.
    ... skipped ...
grad_content = extract_image_gradient(content_image)
grad_style = extract_image_gradient(style_image)
style_grad_removed = style_image − grad_style
pastiche = 1/2 style_grad_removed + 1/2 grad_content
```

The algorithm for the baseline solution consists of removing the gradient of the style image and adding the gradient of the content on top of what is left. Figure 8 visually shows the workflow of the algorithm and the results obtained. It is quite unlikely that the resulting image will be of good quality, since the gradients of the two images do not match spatially. Simply

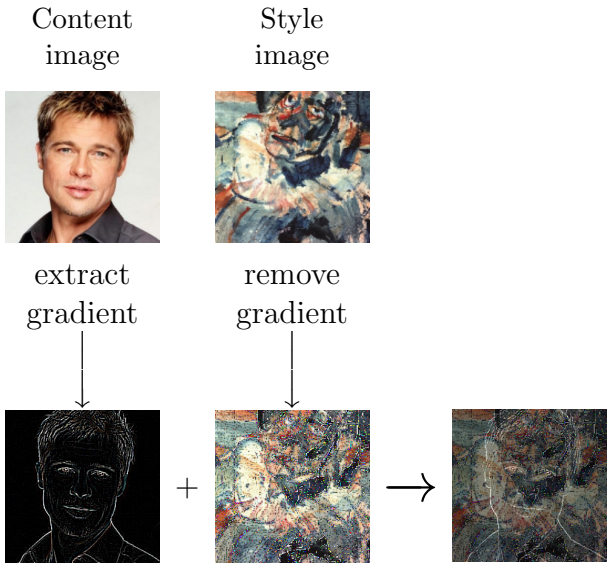


Figure 8: How the gradient 'style transfer' works. The reason that the style transfer failed, is that the gradients are quite unlikely to match perfectly. Hence, in order to achieve desired results, more complicated approaches are needed.

using binary operations (such as addition and subtraction) on the derivatives of the two images will not be successful. The baseline solution shows that achieving style transfer is a non-trivial task, ergo the need for more complicated approaches, such as neural networks, as outlined in §2.1.2.

Alternative approaches exist, e.g. Reinhard et al. (2001), which are based on statistical analysis of the image characteristics. However, such approaches are out of the scope of this project.

3.2 Implementation of the style transfer network architecture

Since PyTorch provides automatic differentiation[1], there was no need to explicitly define how parameters are updated on every iteration based on the losses. This resulted in a concise and easy to maintain implementation of the style transfer network architecture presented in Dumoulin et al. (2016).

The PyTorch framework also provides an implementation of instance normalisation, (described in Ulyanov et al. (2016)), which is the technique from which conditional instance normalisation stems. It is therefore natural to implement the module for conditional instance normalisation as a class which is composed of several of the already implemented and well tested PyTorch classes for instance normalisation. Therefore, this is the implementation I chose to realise.

[1]the `autograd` package – `https://pytorch.org/docs/stable/autograd.html`

	Naïve approach	GPU oriented	approx. speedup
Polynomial kernels	1.669ms	0.032ms	×50
Radial Basis Function kernels	5.072ms	0.174ms	×29

Table 1: Comparison of the two two approaches for calculating the Gram matrix. The number of feature maps of the tensor is 8 and there are 64 elements per feature maps. Results have been averaged over 1000 tests. The benchmarking code is shown in Appendix A.

3.3 Computing Gram matrices

As described in §2.1.2, capturing the style of an image is achieved through matching the Gram matrix of the pastiche to the one of the style image. It has been shown that different distance metrics can be employed for direct pastiche generation (Li et al., 2017), which still yield satisfactory results. I therefore propose to use different kernels such as the polynomial kernel and the radial basis function kernel for constructing the Gram matrix. I will then evaluate how this affects the performance of an arbitrary style transfer neural network. The details of the implementation for supporting different kernels is described below and the results are presented in §4.6.

Given the setup described in Figure 3 (page 5), one approach to computing the features' Gram matrix of $\phi_i(x)$ would be to use nested loops to calculate the inner product of every two vectorised features of $\phi_i(x)$. Although PyTorch has strong GPU acceleration, the interpreter is unable to extract parallelism from the naïve approach of nesting loops. This results in longer running times of the experiments, making the implementation too inefficient to be of any use. An alternative solution was needed.

I used two techniques (one for polynomial kernels and one for the RBF kernel) for implementing the Gram matrix calculation in a way that can be effectively parallelised on the GPU by the interpreter. When compared to the naïve approach, each of the techniques gives speedup of factor of 50 and 29 respectively (see Table 1).

3.3.1 Polynomial kernels

A degree-d polynomial kernel of is defined as:

$$\kappa(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T\mathbf{y} + c)^d \tag{3.1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors in the input space and c is a constant. A linear kernel is just a polynomial kernel of degree $d = 1$ and $c = 0$.

Let the activations at a layer i be denoted $\phi_i(x)$ as shown in Figure 3. By vectorising the feature maps, $\phi_i(x)$ can be transformed into a $C \times N$ matrix M, where C is the number of feature maps of $\phi_i(x)$ and N is the number of elements per map. The Gram matrix will therefore contain $\kappa(M_{i*}, M_{j*})$ for every two rows i and j of M.

The naïve approach of nesting loops to compute the kernel of every two rows cannot be parallelised well on a GPU. To resolve this issue, I have used the standard approach of matrix

multiplication for computing every $M_{i*} \cdot M_{j*}$ 'at once'. This is achieved by multiplying M with a transposed version of itself:

$$G = MM^T \tag{3.2}$$

Every element of G is therefore:

$$G_{ij} = M_{i*} \cdot M_{*j}^T = M_{i*} \cdot M_{j*} \tag{3.3}$$

Adding c to every element of G and then raising all elements to the power of d:

$$G_{ij} = (M_{i*} \cdot M_{j*} + c)^d = \kappa(M_{i*}, M_{j*}) \tag{3.4}$$

which makes G equal to the Gram matrix.

Since matrix multiplication and per element operations can be parallelised well on the GPU by the interpreter, we achieve the speedup shown in the first row of Table 1.

3.3.2 Radial basis function kernel

The definition of the radial basis function (RBF) kernel is:

$$\kappa(\mathbf{x}, \mathbf{y}) = exp(-\frac{\|\mathbf{x} - \mathbf{y}\|_2^2}{2\sigma^2}) \tag{3.5}$$

where $\mathbf{x}$ and $\mathbf{y}$ are vectors, σ is a free parameter and $\|\cdot\|_2^2$ denotes the squared l^2 norm (also known as Euclidean norm).

Again, for computing the Gram matix, the kernel between every pair of features should be considered. Unfortunately, matrix multiplication cannot help here, as the first step towards computing the RBF kernel is not finding the dot product, but calculating the squared l^2 norm between the two vectors. Despite the extensive search I performed, I did not manage to find a suitable implementation[2] for calculating the Gram matrix of RBF kernels. Due to the specificity of my task, it was quite unlikely that such an implementation exists, so an alternative approach was derived.

Let M be the matrix, where every row corresponds to a feature map of $\phi_i(x)$ (just as before). A $C \times C \times N$ tensor (denote it R) is built by stacking iterated copies of M. In other words, each new layer of R is derived from the previous one by shifting all rows down. The first row becomes the second, the second – third, etc. The last row becomes the first. The whole procedure is

[2]One, from which the interpreter can extract good GPU acceleration.

shown below and for every layer the new position of the first row of M is highlighted in red:

$$\begin{array}{c} \begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ C{-}1 \\ C \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \dots & N \end{array} \\ \begin{bmatrix} M_{1,1} & M_{1,2} & \dots & M_{1,N} \\ M_{2,1} & M_{2,2} & \dots & M_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ M_{C-1,1} & M_{C-1,2} & \dots & M_{C-1,N} \\ M_{C,1} & M_{C,2} & \dots & M_{C,N} \end{bmatrix} \\ \textit{Layer 1} \end{array} \begin{array}{c} \begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ C{-}1 \\ C \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \dots & N \end{array} \\ \begin{bmatrix} M_{C,1} & M_{C,2} & \dots & M_{C,N} \\ M_{1,1} & M_{1,2} & \dots & M_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ M_{C-2,1} & M_{C-2,2} & \dots & M_{C-2,N} \\ M_{C-1,1} & M_{C-1,2} & \dots & M_{C-1,N} \end{bmatrix} \\ \textit{Layer 2} \end{array} \dots \begin{array}{c} \begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ C{-}1 \\ C \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \dots & N \end{array} \\ \begin{bmatrix} M_{2,1} & M_{2,2} & \dots & M_{2,N} \\ M_{3,1} & M_{3,2} & \dots & M_{3,N} \\ \vdots & \vdots & \ddots & \vdots \\ M_{C,1} & M_{C,2} & \dots & M_{C,N} \\ M_{1,1} & M_{1,2} & \dots & M_{1,N} \end{bmatrix} \\ \textit{Layer } C \end{array} \end{array} \tag{3.6}$$

A second tensor S of rank 3 is built, with the same number of layers, but all stacked layers are *exact* copies of M. Let $G = (R - S)^2$, i.e. the tensor obtained from subtracting S from R elementwise and squaring every element. The results at a layer i of G are:

$$\begin{array}{c} \begin{array}{c} \\ 1 \\ \vdots \\ i \\ \vdots \\ C{-}1 \\ C \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \dots & N \end{array} \\ \begin{bmatrix} (M_{C-i+2 \bmod C,1} - M_{1,1})^2 & (M_{C-i+2 \bmod C,2} - M_{1,2})^2 & \dots & (M_{C-i+2 \bmod C,N} - M_{1,N})^2 \\ \vdots & \vdots & \ddots & \vdots \\ (M_{1,1} - M_{i,1})^2 & (M_{1,2} - M_{i,2})^2 & \dots & (M_{1,N} - M_{i,N})^2 \\ \vdots & \vdots & \ddots & \vdots \\ (M_{C-i,1} - M_{C-1,1})^2 & (M_{C-i,2} - M_{C-1,2})^2 & \dots & (M_{C-i,N} - M_{C-1,N})^2 \\ (M_{C-i+1,1} - M_{C,1})^2 & (M_{C-i+1,2} - M_{C,2})^2 & \dots & (M_{C-i+1,N} - M_{C,N})^2 \end{bmatrix} \\ \textit{Layer } i \end{array} \tag{3.7}$$

In other words, layer i of G contains a matrix, where every row is the square of the difference between the spatially corresponding row of M and the one offset by i positions backwards.

G is then transformed into a $C \times C$ matrix, by converting every row at layer i to a single element. The value of the element is equal to the sum of all elements in the row. The element at position i, j is thus:

$$G_{i,j} = \sum_{k=1}^{N} (M_{j,k} - M_{(j-i+1+C) \bmod C,\ k})^2 \tag{3.8}$$

$G_{i,j}$ now contains the squared l^2 norm between the j-th row of M and the row offset by distance i. Applying the rest of the operations (dividing by $2\sigma^2$, negating and exponentiating) to every element of G gives us a *shifted* version of the Gram matrix. This is not a problem, since shifting *both*[3] Gram matrices in the *same* way does not change the Frobenius norm of the difference between the two matrices.

Both methods were compared using the same setup as with polynomial kernels. The proposed approach showed improvement (see second row of Table 1), mainly because the squared l^2 norm between every two feature maps is computed by $G = (R - S)^2$. This operation can be well accelerated on the GPU by the interpreter, but this comes at the expense of additional memory. Stacking copies increases memory consumption by a factor of $\mathcal{O}(C)$. Nevertheless, the speedup achieved outweighs the drawback of the memory increase, so I used this approach for calculating the RBF kernel.

[3]For the pastiche and for the style image

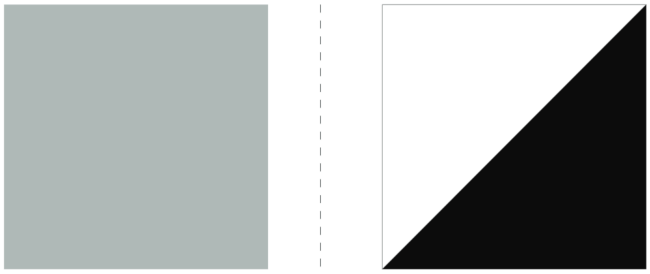


Figure 9: Examples of input images that have equal Gram matrices (Wilmot et al., 2017). The left image is a uniform distribution with mean intensity of $\mu_1 = \frac{1}{\sqrt{2}}$ and standard deviation $\sigma_1 = 0$. The right image is a non-uniform distribution with a mean of $\mu_2 = \frac{1}{2}$ and $\sigma_2 = \frac{1}{2}$. If interpreted as activations of a single feature map, the two activations have equal Gram matrices.

3.4 Histogram matching

3.4.1 Why are Gram matrices not enough?

Wilmot et al. (2017) showed that the problem with aligning Gram matrices is that many different distributions can have the same Gram matrix. This is illustrated in Figure 9. Let's assume that the generated pastiche is a grey image (left image) with a mean intensity value of $\mu_1 = \frac{1}{\sqrt{2}}$ and a standard deviation of $\sigma_1 = 0$. Further we assume that the style image is an image with $\mu_2 = \frac{1}{2}$ and $\sigma_2 = \frac{1}{2}$ (the right image of Figure 9). Although the 'style' of the pastiche is clearly different from the one of the style image, their intensities have equal Gram matrices.

In reality, we do not match the Gram matrices of the image intensities, but match the feature activations, as extracted by the trained classifier. If we view the two images from Figure 9 as feature map activations, then different distributions can still have the same Gram matrix. As a result, our network may end up aiming for a different style which has the same Gram matrices of the activations as the desired one.

3.4.2 The solution

Wilmot et al. (2017) proposed to resolve this issue by trying to preserve the histogram of the feature activations as well as aligning the Gram matrices. This is achieved by using an ordinary histogram matching technique to remap the pastiche activations to the style image ones and then comparing the activations of the original pastiche activations and the remapped ones.

My initial proposition was to experiment with histogram matching instead of Gram matrix matching. In addition to this experiment, I suggest to preserve the original style loss term and evaluate the extent to which histogram matching can aid image generation.

Sticking to the notation of Figure 3, (page 5) the histogram loss at layer i is defined as:

$$L_h^i(p) = \frac{1}{U_i} \left\| \phi_i(p) - matched(\phi_i(p), \phi_i(s)) \right\|_2^2 \tag{3.9}$$

where U_i is the number of units at layer i, $\phi_i(p)$ and $\phi_i(s)$ are the activations of the pastiche and the style image respectively. $matched(\phi_i(p), \phi_i(s))$ are the activations obtained by performing histogram matching of $\phi_i(p)$ to $\phi_i(s)$. The total histogram loss is:

$$L_h(p) = \sum_{l \in \mathcal{H}} L_h^l(p) \tag{3.10}$$

where $\mathcal{H}$ is the set of 'histogram layers'. As with the set of style layers $\mathcal{S}$, $\mathcal{H}$ is chosen to contain shallow layers.

3.4.3 Implementation of histogram matching

The histogram matching algorithm is not provided by default in PyTorch, so a custom implementation was needed. The algorithm implemented for histogram matching follows the approach described in Figure 5 (page 8):

Algorithm 2: Matching the histograms of *source* to *target*

input : `source`, `target` – tensors of any shape
output: `matched` – the tensor obtained by histogram matching. Has same dimensions as `source`

```
def compute_histograms(input):
    # Uses a PyTorch function to compute the histogram of a tensor
    ... skipped ...
def compute_cumulative_distribution(input):
    # Calculates the cumulative distribution of input
    ... skipped ...
def nearest(p, cdf):
    # Finds the closest to p value in cumulative distribution function
    # and returns its index, i.e.  its histogram value
    ... skipped ...
source_hist = compute_histograms(source)
target_hist = compute_histograms(target)
source_cdf = compute_cumulative_distribution(source_hist)
target_cdf = compute_cumulative_distribution(target_hist)
matched = [nearest(x, target_cdf) for x in source_cdf]
```

The only drawback of the implementation is the fact that in PyTorch, GPU support for computing the histogram of a tensor has not been been released yet[4], so the `compute_histograms` function must be performed on the CPU. This did increase the time needed to train a network that uses histogram matching but it was still in the acceptable limits and all experiments were performed successfully.

[4] As of April, 2019

3.5 Repository overview

The project is separated into three main modules as shown in Figure 10. One of the core modules is the `loss_calculator`. This contains the implementation of the learning framework: everything related to calculating the losses is organised into this Python module. This comprises the class definitions of the PyTorch modules for the content, style and histogram loss, as well as functions which attach these modules to a given classifier, e.g. the `get_model_and_losses` function. See §3.6 for more details on the design.

The rest of the modules contain the code for the style transfer network itself and a version of the ResNet architecture, but without the batch normalisation layers. This 'copy' was needed as using ResNet directly did not initially work for style transfer and further modifications were necessary (see §4.5.2).

The repository also contains several other scripts. These are not intended to be used as libraries (in contrast to the above modules), but rather to run various processes or to execute simple algorithms. They include:

- The baseline algorithm from §3.1.
- A script to perform the actual training of a style transfer model – this includes loading the training datasets, training the network with using the `loss_calculator` module, saving the training data (such as the running loss on every iteration) and models. A great amount of this training data has become useful at a later stage: correctness of the project was evaluated using this data (as in §4.3) and different configurations were compared (such as in §4.5.1).
- A script to display/save images, given a model – used to speed up generating the pastiches which were later displayed in Evaluation chapter.
- Code with the heuristics used in §4.6 and §4.7.
- A small set of the unit tests described in §3.7.
- An R script to produce most of the graphs displayed in Evaluation chapter.

All the code has been written from scratch. It is standalone and does not rely on any previous repositories.

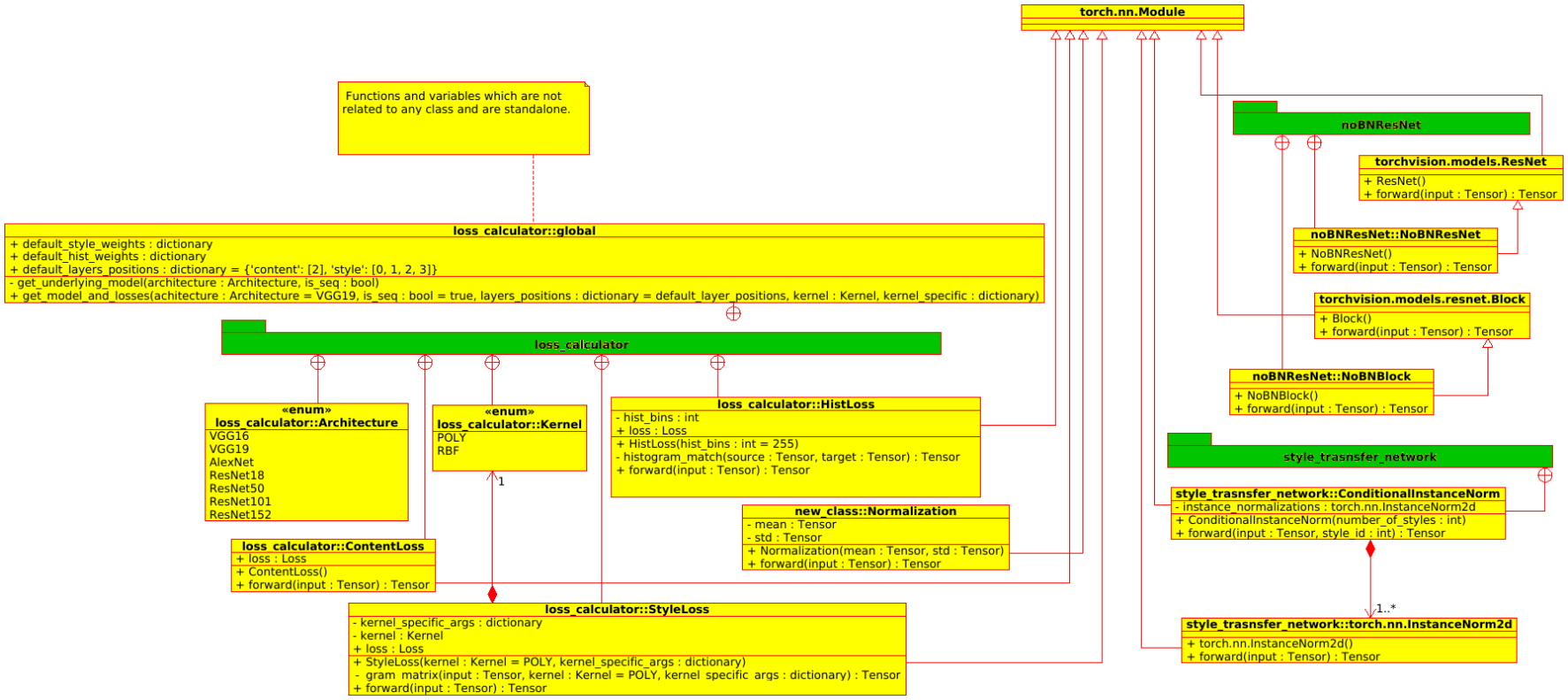


Figure 10: The organisation of the project:

- The learning framework which calculates the losses is contained in `loss_calculator`. This module contains the class definitions of the components for calculating different losses as well as functions that integrate these components into any of the trained classifiers.
- `noBNResNet` implements a batch normalisation-free copy of the ResNet architectures by extending the current PyTorch implementation.
- `style_transfer_network` contains the code for the conditional instance normalisation module as well as the rest of the code for the style transfer network.

3.6 The learning framework

One of the core parts of my project is experimentation with other classifier architectures, beyond VGG (Simonyan and Zisserman, 2014). In theory, this should be feasible, and other architectures, such as ResNet (He et al., 2015), which have achieved higher performance on the ImageNet (Deng et al., 2009) dataset, might generate images of higher quality. However, there have been no official publications on performing style transfer using an architecture other than VGG. It would be interesting to evaluate how suitable different architectures are, in particular ResNet and AlexNet (Krizhevsky et al., 2012). Therefore, the first of the main design goals of the implementation of the learning framework should be able to support the use of other architectures.

The next of the core propositions is to asses how the number of layers used to calculate the style loss impacts the quality of the pastiche and whether using fewer style layers can yield satisfactory results. These two objective require that the implementation is modular and well engineered, so that configurations can easily be changed during experimentation, rather than having to modify tens of lines of code. Not only will the time between successive experiments be reduced, but the chance of error when setting the configuration will be significantly lowered.

The implementation I produced achieved all the design goals. It was possible to select all the configuration details simply through changing the parameters to one function. This *avoids the any need for mental operations*, thus reducing the error rate to the minimum. E.g. if one needs to select the x-th layer for the content loss, the user does not need to remember the exact position of that layer for every architecture, but simply needs to specify the number x in the list for content losses. Due to the high suitability of the chosen tools (the PyTorch framework, in particular), implementing this was not complicated – attaching modules to a CNN architecture at runtime was achieved through a simple in-order traversal[5]. The modularity of the code allowed a new sequential architecture to be added with just *two* simple steps:

1. Add an enumeration for that architecture (usually just an integer).
2. Add the architecture to the list of the supported underlying CNN models.

3.7 Unit testing

The focus of the unit tests was the `loss_calculator` module, as it had a lot of custom elements, in which bugs can arise. There are two sets of unit tests – one for verifying the correctness of calculating different losses and one for verifying that the loss modules have been attached correctly. Most of the unit tests were designed so that they could be manually verified because there is only one implementation possible (i.e. there is no basic version to test against). The only exception, is a unit test that establishes that the improved, GPU-oriented, way to compute the Gram matrices produces same results as the naïve one.

[5]If only sequential architectures were considered, this would have been achieved only with for-loop iteration.

The unit tests do not aim to provide full coverage, as this is impossible to achieve. There are exponential number of configurations that can be used and an infinite number of different inputs that can be fed through the loss modules. Instead, the unit tests aim to identify more major bugs that would affect the training *before the training has taken place*.

Due to the nature of the neural networks, it is impossible to write unit tests that prove the correctness of an arbitrary style transfer network. As a substitute, correctness has been verified empirically, trough qualitative and quantitative experiments (see Evaluation, §4.3).

3.8 Summary

This chapter presented the implementation of the various parts of the project. Optimisations to the running time of the Gram matrix computation were introduced, as well as the trade-offs that come with these optimisations. Then, histogram matching was discussed and implementation was given. Next, the design of the learning framework and interaction with other project parts was studied. Finally, the testing tactics were discussed.

The Evaluation chapter will present the experimental results produced by different models, demonstrating the achievement of the success criteria.

Chapter 4

Evaluation

This chapter starts by revisiting the success criteria and demonstrating that they were all successfully achieved (and even exceeded) with pointers to the corresponding sections. The datasets used are then presented. Each of the sections corresponds to a single experiment.

4.1 Success criteria

The following success criteria were initially defined in the project proposal (Citations from the original proposal are in *italic*):

- *Implement a pastiche generator neural network* and *Implement conditional instance normalization as part of the pastiche generator*

  The implementation has been verified in §4.3 through an experiment (borrowed from Dumoulin et al. (2016)) which empirically proves correctness.

- *Implement methods to construct the Gram matrix ... The code should also be able to accommodate networks with various architectures and number of layers*

  Without the methods for constructing the Gram matrix, no style transfer would be possible at all. The whole chapter proves the correctness of the method.

  Results from architectures with various number of layers are presented and discussed in §4.4.

- *Download AlexNet, VGG-16 and ResNet as pre-trained classifiers and integrate in the system* and *Evaluate how the usage of different pre-trained classifiers affects the quality of the synthesized pastiche ....*

  As mentioned in Preparation, VGG-16 was substituted with VGG-19 for comparability with Dumoulin et al. (2016). The results of using different trained classifiers are given in §4.5. Although ResNet did not work initially, I persisted and showed that adjustments can improve the quality of the generated image.

- *(Extension) Add RBF and polynomial as activation similarity metrics when constructing the Gram matrix*

  I have presented both visual and empirical results on the correlation of the kernel used and the pastiche generated in §4.6.

- *(Extension) Perform histogram matching of the activations in the pre-trained classifier between the pastiche and the style artwork*

  I demonstrated in §4.7 that although histogram matching is not suitable for style transfer on its own, it can be used to augment the current learning method. I verified this not only through visual inspection but also through an additional quantitative experiment.

4.2 Datasets used

The following datasets were used for the experiments:

- MS-COCO 2014 – Common Objects in COntext. The training set of MS-COCO contains 83K images of complex everyday scenes with common objects in their natural context. The dataset also contains object segmentation and labeling, but these were not needed here. All of the dataset's images were used for training each of the models. MS-COCO is freely available on: `http://cocodataset.org/#download`

- 10 Monet paintings from Dumoulin et al. (2016) – These were used to reproduce the experiment from the paper, which proves that using conditional instance normalisation can generalise across styles. Artworks available in the public domain.

- 10 artworks of different styles from Dumoulin et al. (2016) – These were used for the rest of the experiments. This set was chosen for two reasons: firstly, styles vary greatly in colour scheme and painting technique. Secondly, using this datasets allows for results to be compared to the state of the art ones presented in their paper. Artworks available in the public domain.

4.3 Training a network on many styles is comparable to training on a single style

The purpose of reproducing this experiment is to empirically verify the correctness of the style transfer network and more precisely, the conditional instance normalisation module. The style transfer network was trained on the 10 Monet paintings for comparability with Dumoulin et al. (2016). The trained model is able to successfully synthesise a pastiche of a given content image for every style. The images produced retain the **same semantic content** as the original content image, have the **colour scheme and brushstroke texture of the style image** and possess **no artifacts**, such as spots of colours not present in the style image. Several pastiches are presented in Appendix B (page 48), to further support the claim.

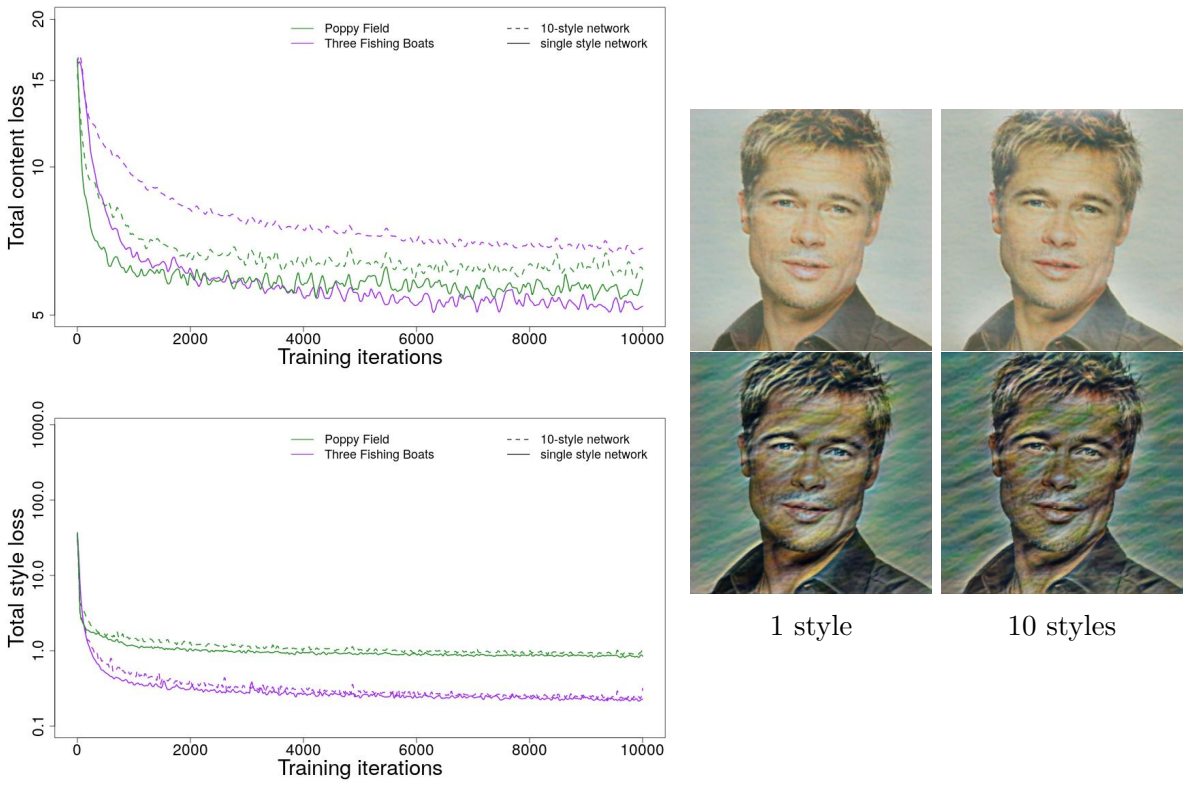
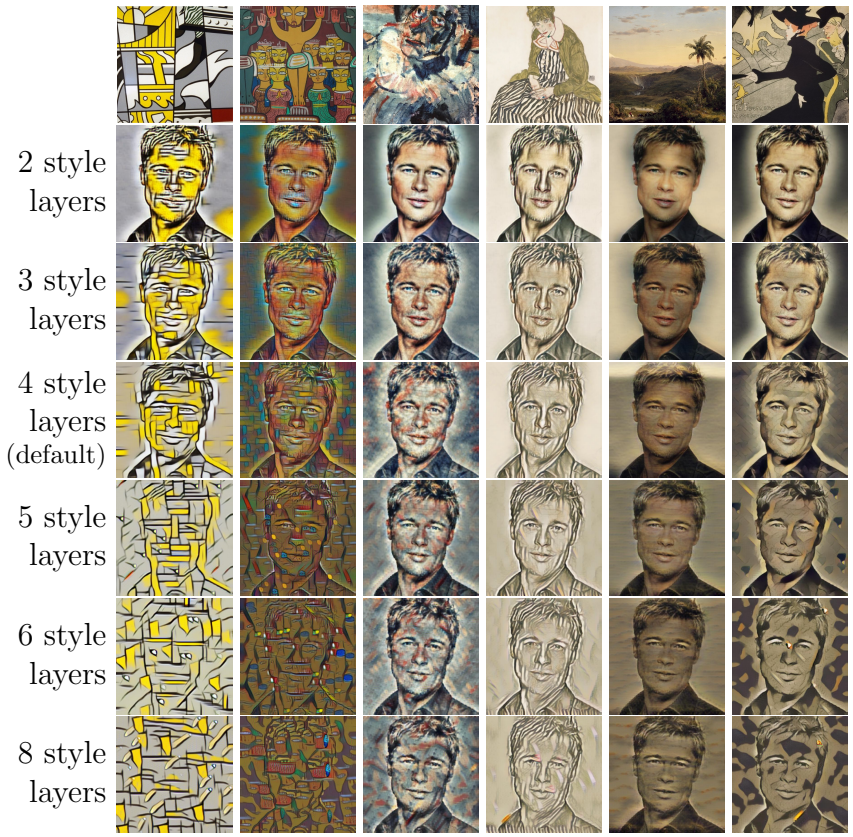


Figure 11: Training on many styles (10 here) produces results as good as when training on a single style. (Left column) Same artworks are represented with the same colour in the graphs. A dashed line is used for the 10-styles model, and a solid for the single style ones. Although the total content loss is always a bit higher for a network trained on many styles, the total style loss is comparable. (Right column) The network trained on many styles produces visually comparable pastiches to the ones generated by a network trained specifically on one of the styles.

Two separate single-style transfer networks were trained on two of the aforementioned Monet paintings. The style and content losses for each painting and the visual results were compared both for the multi-style network and for the single-style ones. The results are presented in Figure 11.

Visually observing the graphs, it can be concluded that the total style loss of a 10-style network converges with a comparable rate to the losses of the single-style networks. The content loss for 10 styles is a bit higher compared to the corresponding losses of single-style networks, but this did not have any major impact on the visual aspect of the images. Pastiches from both types of networks look qualitatively similar, with only minor differences (e.g. the 10-style network used a slightly darker background for the bottom style).

Based on both visual and empirical results, it can be concluded that the style transfer network and the conditional instance normalisation described in §3.2 have been implemented successfully.



Using 8 style layers gives many artifacts.

Figure 12: The granularity of the pastiche is inversely proportional to the number of style layers used. Each row represents a different training configuration. Using fewer than 4 layers (the default case) results in smoother images, but with less texture. Adding more layers has only a negative impact – larger 'patches' from the style image are used for pastiche synthesis, which makes the results less appealing.

4.4 How many style layers should we use for optimal results?

In order to assess how the number of style layers used impacts the quality of the generated pastiche, six different multi-style transfer networks were trained. The 10 artworks of varying style were used for training, in order to compare whether changing the number of style layers would lead to overfitting. A full set of pastiches is presented for the default case of using 4 style layers (Appendix B, page 49)

For computing the style loss, each of the six networks respectively uses the activations of the first 2, 3, 4, 5, 6 and 8 convolution layers of a trained VGG classifier. As we go deeper in the network, by increasing the number of layers, the magnitude of the style loss increases. Therefore, in order to ensure a fair comparison between different configurations, the style weight is manually adjusted so that the initial style loss is comparable to the one when 4 layers are used. With the only exceptions of the style layers and the style loss weight, every other parameter is kept constant within different configurations. Figure 12 shows the effect of varying the number

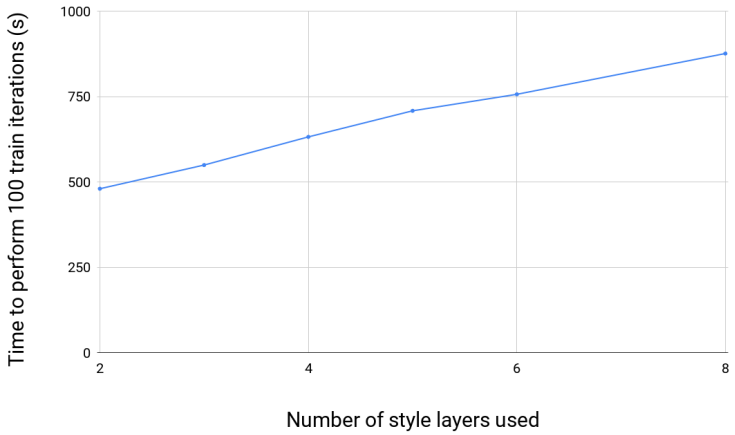


Figure 13: Increasing the number of style layers used gives a linear increase in the time taken to train a model.

of layers. Creating a pastiche using fewer layers is still feasible, but some of the artistic pattern is lost. Style transfer becomes mainly colour transfer.

Increasing the depth of the layers used for calculating the style loss did not increase the quality of pastiches. Instead, the pastiches produced contain spotty artifacts of colours that are not observed in the style images used for training. An additional drawback is that the computational cost to train the network increases, as we increase the number of style layers. This is mainly due to:

- Propagating the images deeper through the trained classifier to extract features.
- Computing the Gram matrix more often (once per style layer).

Style transfer is an ill-posed problem – deciding which option produces 'better' results is mostly a matter of personal preference. One may even want to trade some of the texture granularity for reduced training times. In order to evaluate the correlation between speed and the number of style layers, I measured the time to perform 100 training iterations for each of the configurations from Figure 12. Results (Figure 13) show that the time to train the network is proportional to the number of style layers used. However, the speed gain from reducing the style layers by a factor of 2 (e.g. from 4 style layers to only 2), is approximately a factor 1.3 times.

Based on these observations, the optimal choice of style layers remains the first 4 convolutional layers, since:

1. Preserves enough of the content.
2. Maintains enough texture to make the artwork realistic.
3. Training of a network fast enough to be useful.

Figure 14: Using AlexNet instead of VGG during training of the style transfer network produces suboptimal results. The pastiches produced are of a lower quality and the content is not perceptible for some styles (e.g the abstract style in the first column). Reducing the style weight α_s by a factor of 10 results in better pastiches of the first artwork *only* – other pastiches lose most of their colour and texture.

4.5 Comparing different trained classifiers as feature extractors

The primary motivation for conducting this experiment is that all recent research publications on neural style transfer use VGG as the feature extractor. Moreover, to the best of my knowledge, there is no evidence of training a style transfer network using a different architecture. Therefore, I decided to perform my own investigation, the results of which are presented below.

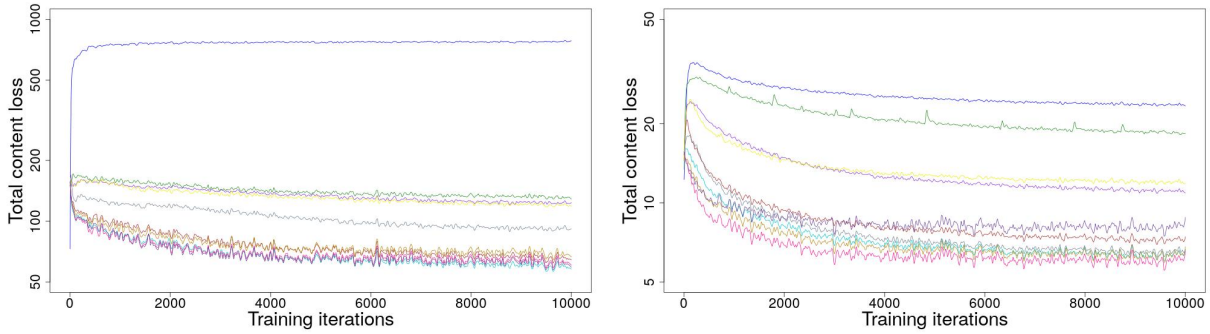
In order to check whether classifiers other than VGG can be used for the training process, several transfer networks were trained. The two main architectures I have focused on are AlexNet and ResNet. Apart from the trained classifier and the style weight parameter (different classifiers have a different magnitude of the activations), the configuration of the learning framework was not changed – first 4 convolutional layers were used for the style loss and the 3rd layer was used for the content loss.

4.5.1 Comparison with AlexNet

A sample of the results produced by a network trained using AlexNet are shown in Figure 14. Compared to using VGG, results obtained from training with AlexNet are of a lower quality. Pastiches generated have more noise (as in Figure 15), making the results less appealing. The network is also unable to create a pastiche of more abstract paintings (e.g. *Bicentennial Print*, the leftmost artwork in Figure 14), as any visual information about the content is lost. In order to confirm that this was not just a case of putting more emphasis on the style, the experiment was repeated with the weight of the style loss reduced by a factor of 10 (Figure 14, third row). With the new setup, transferring the style of *Bicentennial Print* whilst preserving the content is successful. However, the rest of the generated images lose their artistry. Unlike using VGG,

AlexNet VGG

Figure 15: Results produced by an AlexNet-trained network are more noisy than the ones produced by a network trained using VGG.



AlexNet VGG

Figure 16: A model, trained using AlexNet, fails in preserving the content for some of the styles. In the graphs, same colour represents same artwork. The content loss loss for *Bicentennial Print* (indicated in blue) ends up much higher when compared to that of other styles' content losses. By contrast, when VGG is used, the gap between the final loss of *Bicentennial Print* and other styles' losses is noticeably smaller.

with AlexNet, different styles need to have different style weights in order to achieve same results. This suggests that using AlexNet for training cannot produce a model that has the same ability to generalise between different styles.

To verify this empirically, the convergence of the total content loss of AlexNet was compared to that of VGG for every style. As it can be seen from Figure 16, *Bicentennial Print*'s content loss is not being optimised when AlexNet is used and remains several times higher than that of other styles. By comparison, with VGG there is still an initial increase in the content loss for that style, but ultimately the difference in content loss between this artwork and the next ones is much lower.

From the visual and empirical evidence, it can be conjectured that the model obtained through training with AlexNet generalises between styles less well than VGG. It can be hypothesised that this is due to AlexNet having worse performance than VGG on specific tasks, such as object detection (see Canziani et al. (2016) for quantitative comparison between different networks).

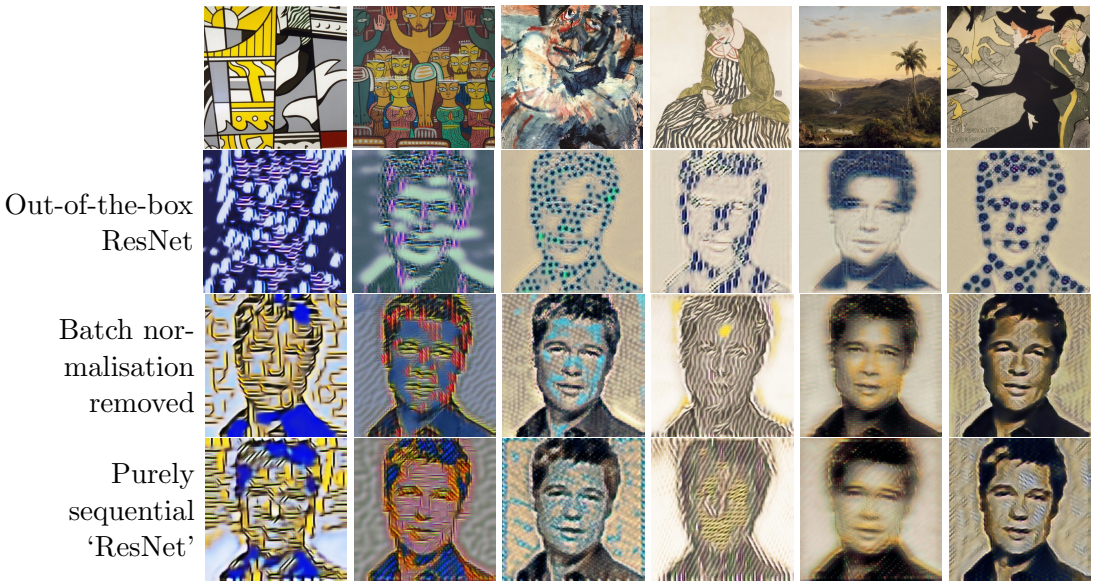


Figure 17: ResNet does not work out of the box at all, but adjustments can improve the performance. Removing batch normalisation improves on transferring some of the styles. Additionally removing the residual connections slightly improves the performance for the first two styles.

4.5.2 Comparison with ResNet

Based on the previous experiment, common sense suggests that ResNet would be more useful than the other two options as it has superior performance to the other two classifiers on the ImageNet dataset. Surprisingly, it produced very poor results when used for training a style transfer network. The first row of Figure 17 shows that using ResNet *without any adjustments*, could be said to have failed in training a style transfer network because:

- Different colours are used most of the time, e.g. in the leftmost artwork yellow, white, gray and black are the colours used most, but the output of the network has completely different colours.
- Some of the images produced have many artifacts in the form of dark spots.

Two of the differences between ResNet and VGG which might explain this behaviour are:

1. ResNet (as the name suggests) has residual connections between layers – this mixing of features from different layers might affect the process of optimisation of the style transfer network.
2. ResNet employs batch normalisation, while VGG does not have any normalisation layers. This could be a potential problem – in the implementation, in order to calculate the content and style losses, pastiches, content images and an artwork are input through the network *together*. These three types of images could be of very different colours and while normalisation has a positive effect on specific tasks (e.g. object detection/segmentation), it can potentially have a negative effect on style transfer.

Therefore, in order to check these conjectures, the following adjustments were made:

- The first adjustment to be made was deleting the batch normalisations. The results in Figure 17 (third row) showed improvement, especially in matching the correct colour, but not for every style.

- Then, the residual connections were removed[1] too. This resulted in a slight improvement in the quality of the pastiches of the first two artworks.

Unfortunately, even with these two modifications, the results are much less satisfactory, compared to employing AlexNet or VGG for the training procedure. Further improvements could be attempted, e.g. training a new sequential-like version of the ResNet, as the current sequential version uses the same weights as the residual one. Another (quite brute force) possibility, could be to try different combinations of style and content layers, as ResNet features are not hierarchical, but rather spread out a lot. However, as each training is time consuming and the second proposal could take an exponential number of trials and errors, further experiments were not conducted.

4.6 (Extension) Correlation between smoothness and kernel degree

It has been shown that style transfer can be performed by using different distance metrics (Li et al., 2017). Therefore, I experimented with *constructing the Gram matrix using different kernels*, in order to find out whether this affects the training of multi-style transfer networks. Apart from this modification (and again adjusting the style loss weight as in §4.4), the default VGG configuration with 4 style layers and 1 content layer was used.

Mapping the features to a higher dimension allows for more complex correlations to be learned. Visual results (Figure 18) showed that increasing the degree of the kernel makes pastiches more fine-grained even for very abstract images, such as the leftmost artwork. As with using fewer style layers, the patterns, that give the feel of a real piece of art are lost in the generated images. In the 'extreme' case of using a Radial Basis Function (RBF) kernel, the synthesised pastiches only have the colours swapped and almost no texture.

In order to evaluate the extent to which these artistic patterns are lost (which is also equivalent to how much the content is preserved), I used the approach, outlined in the following

[1] The rest of the weights are not changed!

Figure 18: As we increase the order of the kernel used for constructing the Gram matrix the, images produced become finer and the differences between styles are eventually lost. However, this comes at the expense of some of their artistic qualities as the texture of the artwork is lost. In the case of mapping the features to an infinite dimensional space using a Radial Basis Function (RBF) kernel, the content is extremely well preserved, even for very abstract artworks (leftmost two images), and no images show signs of repeating patterns/brushstrokes.

algorithm:

Algorithm 3: Heuristic for calculating content preservation (*GradientLoss*)

input : `pastiche`, `content_image`
output: `grad_loss` – how much of the content is lost

```
# Remove any colour information from both images, by converting them to
# grayscale space
pastiche.to_grayscale()
artwork.to_grayscale()
# Extract the gradients as in the baseline solution
grad_content = extract_image_gradient(content_image)
grad_pastiche = extract_image_gradient(pastiche)
# Calculate the euclidean distance between poxels of the two images
grad_loss = euclidean_distance(grad_content, grad_pastiche)
```

The idea is to dispose all information about colour and then compare the gradients of the content images and their corresponding pastiches. The patterns from the artwork will result in a gradient where none exists in the content image. I did not compare the content loss directly, because the deeper layer activations ignore the patterns we seek to quantify.

The algorithm was applied to 1000 images from the MS-COCO validation set, and the results were averaged. Figure 19 shows that the degree of the kernel is correlated to how much of the gradient is preserved (which is similar to how much of the artistic texture is lost). In the extreme case of a kernel with infinite degree (the RBF kernel), the difference in the gradients is reduced by at least a quarter.

From the experiments performed, it can be concluded, that changing the degree of the kernel

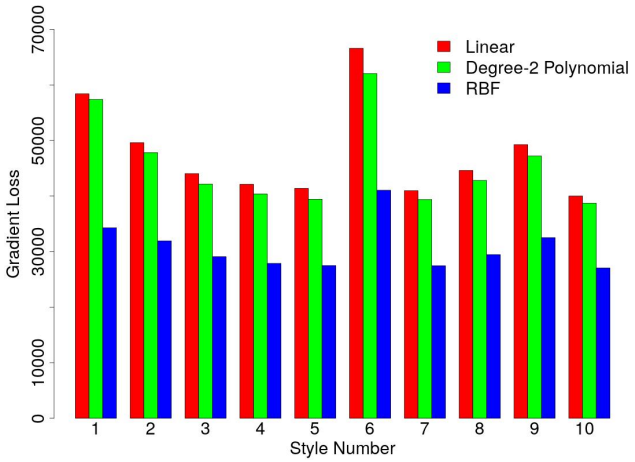


Figure 19: The Euclidean distance between the gradients of the content image and the pastiche (i.e. the 'gradient loss') decreases as the degree of the kernel is increased. When a Radial Basis Function kernel is used, the decrease is more than 25% for every style.

Original image | Style image | RBF kernel (4 style layers)

Figure 20: Using a Radial Basis Function kernel can lead to undesired results. Two objects with contrasting colours (in this case the face and the shirt), can have similar colours in the pastiche.

is another way (in addition to varying the number of style layers) to control the granularity of the pastiche.

However, using an RBF kernel may lead to anomalies, as shown in Figure 20. The style image has a sufficient number of different colours to represent each of the shirt, face and background (three objects with originally contrasting colours), yet the shirt and the face have very similar colours in the pastiche. This makes it difficult to distinguish them from one another.

4.7 (Extension) Improving style transfer with histogram matching

The first experiment related to histogram matching was to completely replace the default computation of the style loss (aligning Gram matrices). In the histogram matching implementation I used 256 bins. As Figure 21 shows, results produced by a network trained using histogram

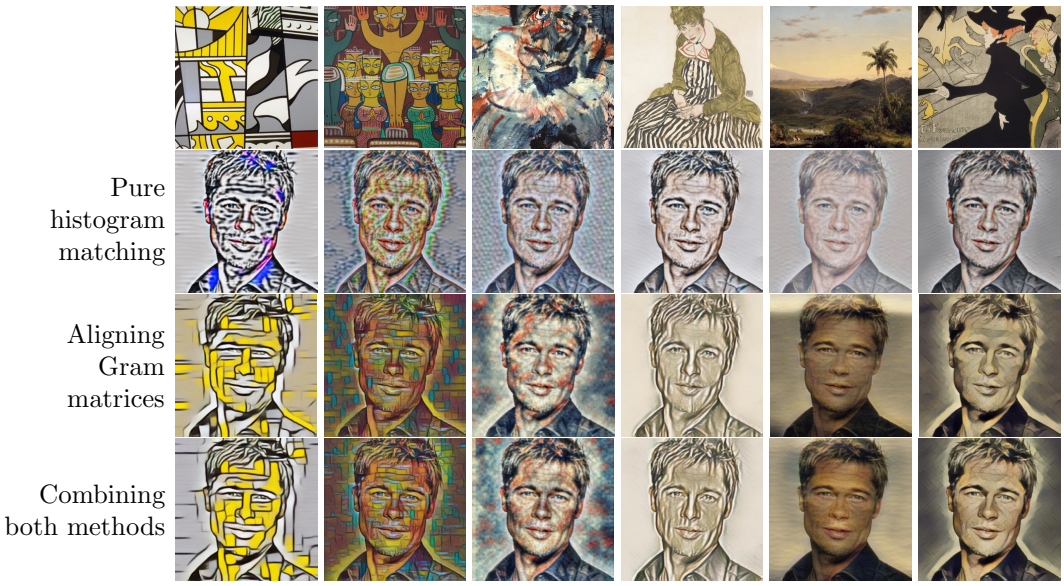


Figure 21: Using histogram matching alone is not enough to reproduce the style or colour of the artwork. Nonetheless, histogram matching can be used to 'augment' the process of simply aligning Gram matrices and lead to better results – the number of pixels with colours that *do not* appear in the style image is reduced (see Figure 22).

matching do not preserve the colour palette of the style image well enough.

One possible explanation for this is that the histogram matching algorithm is not exact and can only approximate the desired histogram (in this case the histogram of the features of the style image). Methods for exact histogram matching exist, e.g. Coltuc et al. (2006), but they are more computationally expensive.

Another issue, possible explanation for the colour mismatch is that the activations of an image are no longer discrete 8-bit pixel values but continuous ones. This implies that ideally an infinite number of histogram bins should be used. Unfortunately, this is not possible in real-world applications, since increasing the number of bins would increase the computational cost of the matching algorithm.

Histogram matching is more useful, when it is used *in addition to* the classical approach of aligning Gram matrices. After adding a histogram loss term to the total loss, the quality of the colours in the generated images improves (especially for the leftmost abstract artwork of Figure 21). Zooming into the pastiches of this abstract artwork (see Figure 22) reveals that not using histogram matching generates colours which are not part of the style image. The background colour is a 'muddy' mixture of the gray and yellow. This result visually shows the problem with only aligning Gram matrices: the Gram matrix of the activations of these colours happens to be close to the ones of the style image.

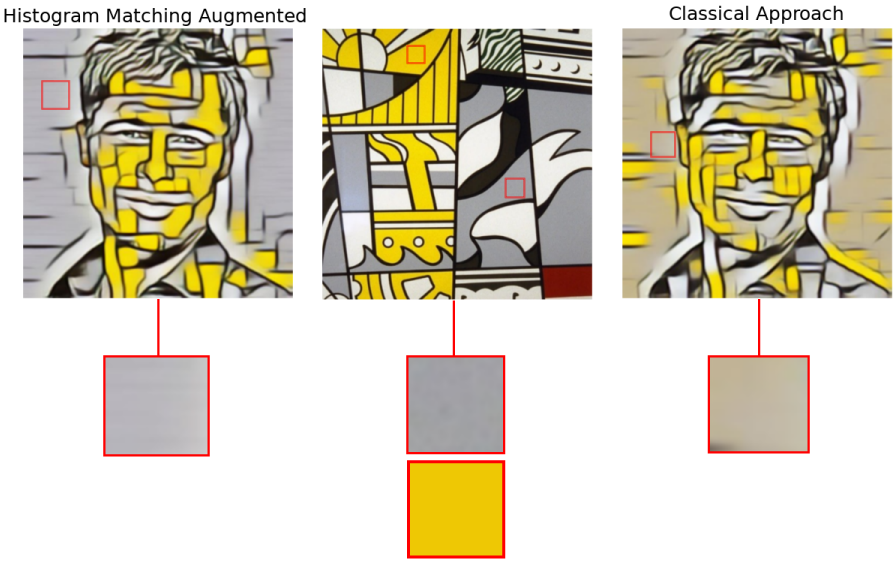


Figure 22: Without histogram matching, the generated images use colours (e.g. for the background) which are not part of the style image. When augmenting the style transfer learning framework with histogram matching, the colours used are much closer to ones in the original artworks.

To further quantify my observation, I developed the following heuristic for calculating the number of 'colour-mismatched' pixels:

Algorithm 4: Heuristic for calculating the number of pixels with different colours

input : `pastiche`, `artwork`
output: `mismatched` – the number of pixels, which have colour not present in the artwork, i.e. a mismatched colour

```
# Convert every image of both images from RGB to Hue, Saturation, Value
pastiche.to_HSV()
artwork.to_HSV()
mismatched = 0
for pixel in pastiche:
    if There is no value in artwork, which is closer than 18 units in Hue:
        mismatched++
```

The two main motivations behind this heuristic are:

- Since content images can be different with various shades, etc. it is acceptable for the saturation (how vibrant a colour is) or the value (how bright/dark a colour is) to be different.
- The number of unique hue values is 360. A difference of up to 5% ($5\% \times 360 = 18$) may be sometimes recognisable but would not alter the overall colour of the pastiche. This it is acceptable, given that the network will have to apply the colour scheme of the style image to any content image possible (e.g. if the style image consists only of the colour blue, whereas the content image has many different objects of different colours).

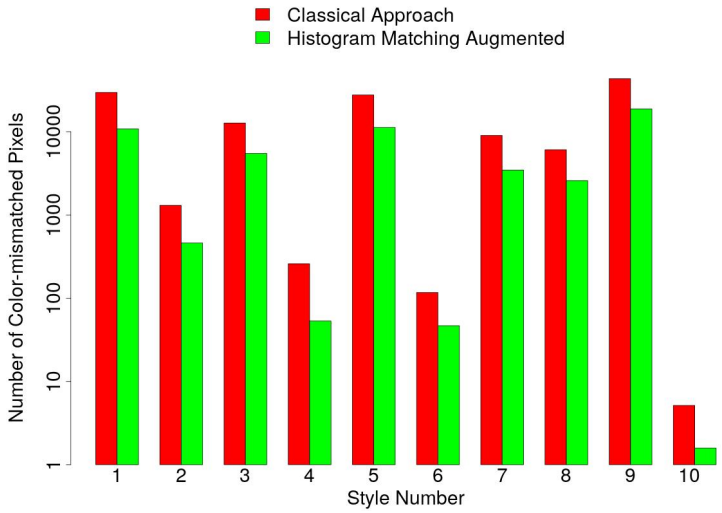


Figure 23: Augmenting the learning process with histogram matching reduces the number of colour mismatched pixels by at least a factor of two for all styles. The y axis is scaled logarithmically.

The performance of the two networks (one trained using the classical approach and one where histogram matching was added) was compared on 1000 images from the MS-COCO validation set. Experimental results (see Figure 23) confirmed the visual observations that histogram matching is beneficial for style transfer – the number of mismatched pixels was reduced by a factor of two.

4.8 Summary

In this chapter I demonstrated that the project meets and exceeds its original success criteria. I not only showcased how using different classifiers affects the pastiche, but also presented how adjustments can improve on the quality of the image. Although the results were not comparable to the ones obtained with VGG, the findings suggest that other classifiers could be used, but they are much harder to get working.

Other experiments showed that number of style layers or the kernel, used for constructing the Gram matrix, can be a tool for modifying the resulting pastiche until personal satisfaction is achieved. Finally, augmenting the learning process with histogram matching improves the results, but I did not find that *solely* using histogram matching leads to satisfactory pastiches.

Chapter 5

Conclusion

5.1 Achievements

Overall, the project was a great success, achieving all milestones in the success criteria (see §4.1), that is all the core objectives and extension goals. Recent scientific papers (Dumoulin et al., 2016) were successfully reproduced, demonstrating both visual and empirical evidence of correctness (see §4.3). A range of experiments related to the design choices in the neural style transfer were performed. Despite the qualitative nature of the project, quantitative evaluation (as in §4.7) was used to minimise the subjective factor of personal preference. I successfully applied ideas and techniques from other research to the style transfer methodology of Dumoulin et al. (2016). Flexibility of the learning framework was demonstrated, with the variety of experiments conducted.

I revealed what are the consequences of using a feature extractor other than VGG. I showed that although architectures like ResNet produced poor results, small modifications could lead to some improvements. Despite the improvements, I achieved best results when using VGG, without the need for any purpose-specific adjustments (as with ResNet). This gives an insight as to why VGG is currently the most popular (and only) choice for style transfer in academia – it produces appealing results and it is fairly easy to get it working, allowing to focus on other objectives.

5.2 Lessons learnt

I found the overall project extremely enjoyable and I am extremely satisfied with the results obtained. Throughout the work done, I had to familiarise myself with a lot of theoretical knowledge and I gained a substantial set of practical skills. If I were to redo this project, I would have approached large scale experimenting with more confidence. To save precious time, I would have considered some of the adjustments at a much earlier stage, e.g. applying the ResNet modifications, instead of spending time on smaller details.

5.3 Further work

The results presented in Evaluation provide some interesting questions which are worth examining further:

1. *What makes a classifier suitable to use for style transfer?* In §4.5 I showed that the suitability of different classifiers is not necessarily correlated to their performance on specific tasks, such as object detection and/or classification. AlexNet and ResNet are just two examples from the vast range of different convolutional network architectures that exist. Yet, to the best of my knowledge, all scientific research uses VGG for style transfer.

2. *Can adjustments transform an 'unsuitable' classifier into a suitable one?* This question comes partly from the previous one and partly from the fact that I showed that it is possible to improve the performance of models trained with ResNet through small adjustments to the underlying classifier used.

3. *Will increasing the number of histogram bins, when performing histogram matching, result in successful style transfer?* It would be interesting to see if, when using this approach on its own, increasing the number of histogram bins used can improve on transferring the style – some of the pastiches of Figure 21 slightly hint what was the style of the artwork they imitate. Once PyTorch releases GPU support for computing a histogram of a tensor, the time to train a network trough the use of histogram matching will be significantly reduced, which will allow for more experiments to be conducted using this technique.

Closing remarks

This project has been a great opportunity to expand my theoretical knowledge and gain practical skills in the field of deep learning. The final product maintains high coding standards and is well-documented. Since, to the best of my knowledge, there is no open-source implementation of a style transfer framework with so many degrees of freedom (e.g. choosing different underlying classifier, distance metric, which layers to use for style loss, etc.), I plan to make my implementation repository openly available under the MIT license, so that anyone can benefit from the work presented here.

Bibliography

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Canziani, A., Paszke, A., and Culurciello, E. (2016). An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678.

Coltuc, D., Bolon, P., and Chassery, J. . (2006). Exact histogram specification. *IEEE Transactions on Image Processing*, 15(5):1143–1152.

Deng, J., Dong, W., Socher, R., Li, L., and and (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255.

Dumoulin, V., Shlens, J., and Kudlur, M. (2016). A learned representation for artistic style. *CoRR*, abs/1610.07629.

Gatys, L. A., Ecker, A. S., and Bethge, M. (2015). A neural algorithm of artistic style. *CoRR*, abs/1508.06576.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

Jing, Y., Yang, Y., Feng, Z., Ye, J., and Song, M. (2017). Neural style transfer: A review. *CoRR*, abs/1705.04058.

Johnson, J., Alahi, A., and Li, F. (2016). Perceptual losses for real-time style transfer and super-resolution. *CoRR*, abs/1603.08155.

Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

Li, Y., Wang, N., Liu, J., and Hou, X. (2017). Demystifying neural style transfer. *CoRR*, abs/1701.01036.

Reinhard, E., Adhikhmin, M., Gooch, B., and Shirley, P. (2001). Color transfer between images. *IEEE Computer Graphics and Applications*, 21(5):34–41.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

Ulyanov, D., Vedaldi, A., and Lempitsky, V. S. (2016). Instance normalization: The missing ingredient for fast stylization. *CoRR*, abs/1607.08022.

Wilmot, P., Risser, E., and Barnes, C. (2017). Stable and controllable neural texture synthesis and style transfer using histogram losses. *CoRR*, abs/1701.08893.

Appendices

Appendix A

Benchmarking Scripts

Comparing[1] different ways to calculate the Gram matrix constructed by. :

- polynomial kernels

```
import torch

class Timer(object):

    def __enter__(self):
        self.start = time.clock()
        return self

    def __exit__(self, typ, value, traceback):
        self.duration = time.clock() - self.start
        print(self.duration)

# Nesting for loops
torch.cuda.synchronize()
with Timer() as t:
    for rep in range(1000):
        x = torch.rand(8, 8, 8).cuda()
        x = x.view(8, -1)
        for i in range(8):
            for j in range(8):
                torch.dot(x[i], x[j])


# Using matrix multiplication
torch.cuda.synchronize()
with Timer() as t:
    for rep in range(1000):
        x = torch.rand(8, 8, 8).cuda()
        x = x.view(8, -1)
        torch.mm(x, x.t())
```

[1] On my NVidia GeForce GTX 1070 8192MB

- Radial Basis Function kernels

```
import torch

class Timer(object):

    def __enter__(self):
        self.start = time.clock()
        return self

    def __exit__(self, typ, value, traceback):
        self.duration = time.clock() - self.start
        print(self.duration)

sigma = 1
# Nesting for loops
torch.cuda.synchronize()
with Timer() as t:
    for rep in range(1000):
        x = torch.rand(8, 8, 8).cuda()
        x = x.view(8, -1)
        for i in range(8):
            for j in range(8):
                torch.exp(-torch.dot((x[i]-x[j]), (x[i]-x[j])) / (2 * sigma**2))


# Building 3D tensors
torch.cuda.synchronize()
with Timer() as t:
    for rep in range(1000):
        x = torch.rand(8, 8, 8).cuda()
        x = x.view(8, -1)
        rolled = torch.stack(list(x.roll(i, 0) for i in range(x.shape[0])))
        rolled -= x
        rolled = rolled.pow(2)
        rolled = rolled.sum(dim=len(rolled.shape)-1)
        rolled = torch.exp(-rolled / (2. * sigma**2))
```

Appendix B

Pastiches

Claude Monet Artworks Used

Grainstacks at Giverny; the Evening Sun

Plum Trees in Blossom

Poppy Field

Rouen Cathedral, West Façade

Sunrise (Marine)

The Road to Vétheuil

Three Fishing Boats

Vétheuil (1879)

Vétheuil (1902)

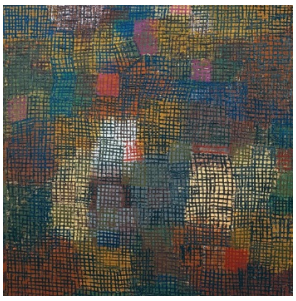
Water Lilies

The 10 varied artworks used

Roy Lichtenstein, *Bicentennial Print*

Ernst Ludwig Kirchner, *Boy with Sweets*

Paul Signac, *Cassis, Cap Lombard, Opus 196*



Paul Klee, *colours from a Distance*

Frederic Edwin Church, *Cotopaxi*

Jamini Roy, *Crucifixion*

Henri de Toulouse-Lautrec, *Divan Japonais*

Egon Schiele, *Edith with Striped Dress, Sitting*

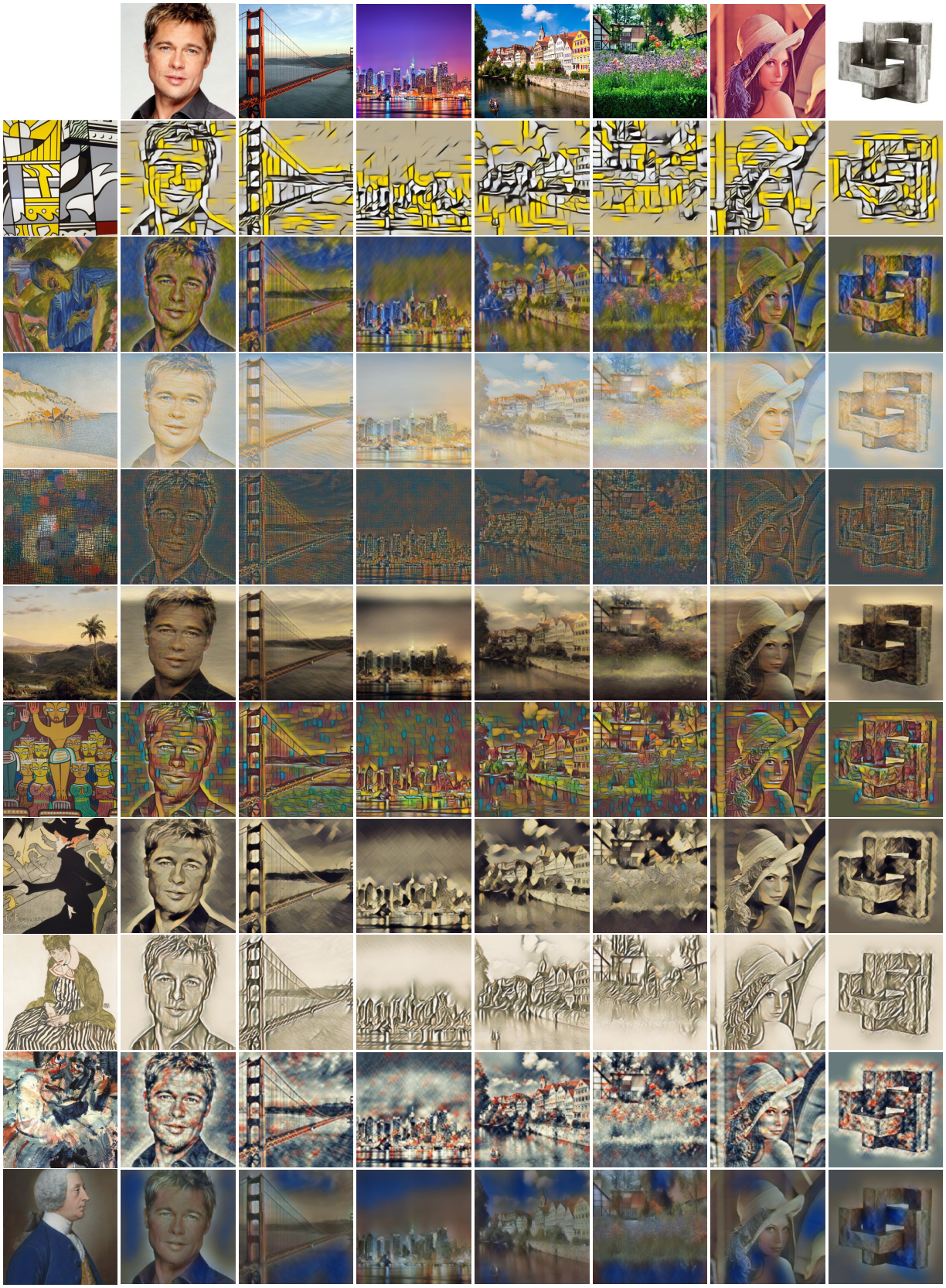
Georges Rouault, *Head of a Clown*

William Hoare, *Henry Hoare, "The Magnificent", of Stourhead*

Pastiches of all 10 Monet artworks

The row corresponds to the style used, the column to the content image used.

Pastiches of 10 artworks of varying style



The row corresponds to the style used, the column to the content image used.

Since typesetting a palette of pastiches is tedious work, other configurations were skipped.

2338A
Hidden
Hidden

Diploma in Computer Science Project Proposal

Design Choices in Neural Style Transfer

October 8, 2018

Project Originator: Hidden

Resources Required: See Resources Declaration

Project Supervisor: Hidden

Signature:

Director of Studies: Hidden

Signature:

Overseers: Dr Robert Mullins and Prof Pietro Lio'

Signatures:

1 Introduction and Description of the Work

A pastiche is a work of visual art which imitates the style of another artwork. Style transfer in computer vision and machine learning domains is the process of automating pastiche synthesis by means of image processing, statistical or machine learning methods. Style transfer has found its application not only in photo editing applications, such as Photoshop, but has also been used in data augmentation, as it preserves shape and semantic content, while altering texture and color.

A style transfer neural network is used to synthesize a pastiche given a content image and a stylistic artwork. The degree of transferring the style of the artwork onto the content image can be measured by inputting the synthesized pastiche, the content and style images through a pre-trained neural network and comparing the layers' activations.

The core part of my project will therefore be to recreate a neural network for arbitrary style transfer as described in Dumoulin et al. (ICLR, 2017). Then I will experiment with different classifier architectures and different style loss structures.

2 Starting Point

The only theoretical knowledge I have of Deep Learning and Neural networks is through the very introductory material presented in the Part IB Artificial Intelligence course. I have basic experience in Python, the programming language I will use for the project, and no experience whatsoever with PyTorch, the library I will be using to create the neural networks, or Tensorflow.

Dumoulin et al. (ICLR, 2017) have the code to their Tensorflow implementation open-sourced. But since PyTorch and Tensorflow differ greatly in their framework structure, I will not be able to use their code directly.

3 Substance and Structure of the Project

The key aspects of the project are its implementation and its evaluation.

3.1 Implementation

The first part of the implementation will be obtaining the MS-COCO dataset which consists of content images and a dataset with works of art taken from WikiArt which is freely available at Kaggle.com. Then I will perform any preprocessing as necessary to ensure that the transformed data is ready to be used.

I will then need to develop a pastiche generator neural network which will take the

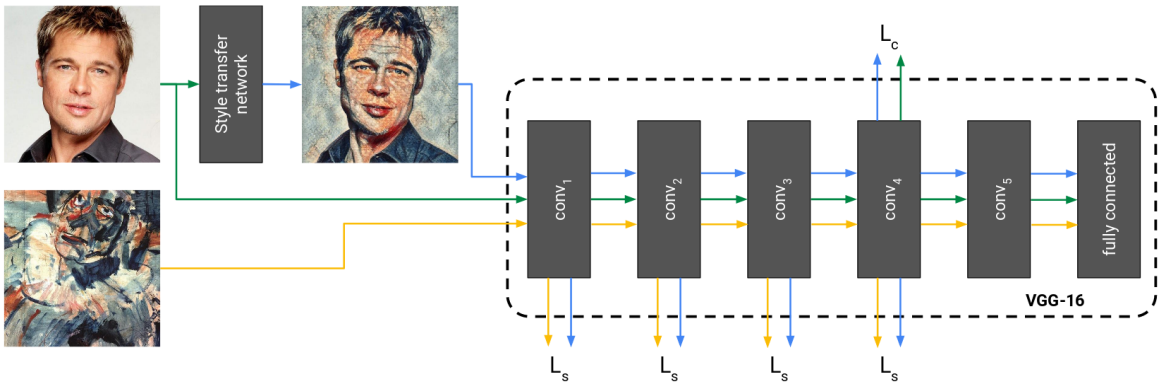


Figure 1: The overall architecture of the deep learning style transfer framework. The generator network (style transfer network) processes the content image and outputs a pastiche. We input both the stylistic artwork and the synthesised pastiche through the pre-trained classifier and compare the Gram matrices of hidden layer activations in the initial shallow layers in order to form the style loss (L_s). The content loss (L_c) compares the pre-trained classifier's deeper layer activations when we input the synthesised pastiche and the content image.

content image as an input and synthesise a pastiche (see fig. 1). The Dumoulin et al. (ICLR, 2017) implementation uses instance batch normalization presented in Ulyanov et al. (2016) to condition on the style used for generating the pastiche (i.e. the specific stylistic artwork). This conditional style transfer methodology gives our algorithm the capacity to learn to transfer multiple styles in a single training procedure. In order to reproduce this functionality I will implement instance normalization as part of the generator network. After that I will download the pre-trained on the ImageNet (Deng et al., 2009) dataset architectures AlexNet (Krizhevsky et al., 2012), VGG-16 (Simonyan & Zisserman, 2015) and ResNet (He et al., 2015) and integrate them into the style transfer system. Finally, I will implement a method which constructs the Gram matrices from the activations in the hidden layers of a pre-trained classifier.

Finally, I will evaluate how the usage of different pre-trained classifiers affects the quality of the synthesized pastiche. My expectation would be that the usefulness of the learned latent representations of these networks to style transfer is correlated with their performance on the ImageNet dataset (Deng et al., 2009). Hence, I hypothesize that ResNet should result in the highest quality pastiche, followed by VGG-16 and finally AlexNet.

3.2 Extensions

The Gram matrix is constructed using the inner product between all activation pairs in a given layer in the pre-trained classifier. My proposition is to use other distance metrics,

such as the radial basis kernel (RBF) function and the polynomial kernel to compare activation similarity. (Li et al., 2017) I will additionally, experiment with histogram matching instead of Gram matrix matching when constructing the style loss (Wilmot et al., 2017). Finally, I will evaluate the efficacy of these modifications as compared to the original formulation in Dumoulin et al. (ICLR, 2017).

3.3 Evaluation

I will use both qualitative and quantitative metrics to evaluate my style transfer framework. I plan on performing the following assessment procedures:

1. Compare the style and content losses as a function of the training iterations for different pre-trained classifiers. This will allow me to assess both the rate of learning and the loss values at the end of training.

2. I will assess how the number of layers used to calculate the style loss impacts the quality of the generated pastiche for different classifier architectures. The purpose of this experiment is to test whether we can achieve similar results with less computation.

3. I will test how different similarity functions, i.e. the RBF or polynomial kernel, cross-entropy measures and histogram matching, used when constructing the Gram matrix impact the rate of learning and the quality of the generated pastiche.

4. I will create a baseline style transfer solution based on simple image processing, such as finding the gradient of images and performing binary operations. I will compare pastiches from my deep learning framework to this baseline.

3.4 Professional Practice

I plan to use version control both for the implementation of my work and for writing up the text of the dissertation. The repository will be backed up on a remote server. In case of a technical issue I would be able to continue working from another setup.

I will endeavour to maintain high coding standard, including but not limited to adhering to design principles such as writing clean and self-documenting code, modularity and ensuring test coverage of key areas of code.

4 Success Criteria

The core part of the project will be considered successful if I manage to:

- Implement a pastiche generator neural network;

- Implement conditional instance normalization as part of the pastiche generator;
- Implement methods to construct the Gram matrix of the activations of each layer in a pre-trained classifier. The code should be modular such that the type of activation similarity can be easily changed during experimentation. The code should also be able to accommodate networks with various architectures and number of layers;
- Download AlexNet, VGG-16 and ResNet as pre-trained classifiers and integrate in the system;
- Evaluate how the usage of different pre-trained classifiers affects the quality of the synthesized pastiche using the evaluation criteria in 3.3.

The extension will be considered successful if I manage to:

- Add RBF and polynomial as activation similarity metrics when constructing the Gram matrix;
- Perform histogram matching of the activations in the pre-trained classifier between the pastiche and the style artwork.

5 Timetable and Milestones

5.1 Michaelmas Weeks 3-4

- Research style transfer in more detail understanding how different architectures and loss metrics work by reading relevant publications in deep learning / style transfer;
- Set up repository for the dissertation;
- Set up structure of the dissertation document.

5.2 Michaelmas Weeks 5-6

- Become more familiar with Python and PyTorch, play with toy examples such as writing a classifier to run on the canonical MNIST data set of handwritten numbers;
- Set up repository of the implementation.

5.3 Michaelmas Weeks 7-8

- Acquire datasets;
- Evaluate the quality of the data; if needed change dataset;
- Perform any necessary preprocessing;
- Implement baseline solution.

5.4 Christmas Vacation Weeks 1-2

- Begin working on the implementation of the neural network of the ICLR paper — since this is one of the core parts of the project, it is expected to take a substantial amount of time.

5.5 Christmas Vacation Weeks 3-4

- Finish and test the implementation of the neural network;
- **Milestone** Fully implemented neural network for style transfer.

5.6 Christmas Vacation Weeks 5-6

- Implement style transfer with different classifier network architectures — once the main implementation is ready and assuming it allows modularity, it would take less time to integrate different changes;
- Begin drafting the dissertation — prepare the core structure and start writing the introductory parts;
- **Milestone** Different architectures implemented;
- Build framework for test and evaluation of the different architectures.

5.7 Christmas Vacation Week 7

- This week is left blank as a buffer period, in case I have other arrangements during the holidays - family, celebrations, etc.

5.8 Lent Weeks 1-2

- Write the progress report;
- Prepare a presentation;
- Discuss with the supervisor successes and failures of the project so far.

5.9 Lent Weeks 3-4

- Perform more extensive testing; obtain results;
- Evaluate the results;
- If ahead of plan, start working on extensions.

5.10 Lent Weeks 5-6

- Catch-up on work — Lent term will have a module of examination with practical sessions.

5.11 Lent Weeks 7-8

- Continue working on the dissertation;
- Discuss how the present the work done with the supervisor.

5.12 Easter Vacation Weeks 1-4

- Finalise first draft of the dissertation;
- Receive feedback from supervisor, DoS & others.

5.13 Easter Vacation Weeks 3-7

- If time allows, work on extensions;
- Reiterate over the dissertation, add additional content if extensions were implemented;
- Catch-up on revision.

5.14 Easter Weeks 1-2

- Receive final feedback;
- Prepare the final version of the dissertation;
- Submit the dissertation.

5.15 Remaining days to 17th of May

The last week and a bit, are not supposed to be used for any work on the dissertation, but rather for revision. These days will also work as another small buffer in case I fall behind with any work.

6 Resource declaration

I will make use of the following resources:

- My own laptop computer (Asus ROG Strix GL703GS-E5011T) for code development and dissertation writing. The exact specifications are:
  - Processor: 2.2GHz Intel Core i7 8750H
  - RAM: 16GB DDR4 SDRAM
  - Graphics: NVIDIA GeForce GTX 1070 8192MB
  - Persistent storage: 800GB HDD
  - OS: Ubuntu 18.04
- The PyTorch library which is open-sourced and freely available;
- The GPUs of the Computational Biology Group in order to train my models and run any experiments.

7 References

V. Dumoulin, J. Shlens, and M. Kudlur. A learned representation for artistic style. *In ICLR*, 2017

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

Y. Li, N. Wang, J. Liu, and X. Hou. Demystifying neural style transfer. *arXiv preprint arXiv:1701.01036*, 2017.

A. Krizhevsky, I. Sutskever, G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks, 2012

K. He, X. Zhang, S. Ren, J. Sun. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385*,2015

K. Simonyan, A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014

J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248255, June 2009

P. Wilmot, E. Risser, and C. Barnes. Stable and controllable neural texture synthesis and style transfer using histogram losses. *arXiv preprint arXiv:1701.08893*, 2017.