Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

Convolutional Neural Networks for Image Style Transfer

Relatore:
Chiar.mo Prof.
Andrea Asperti

Candidato:
Pietro Battilana

Sessione II
Anno Accademico 2017-2018

Contents

Abstract

In this thesis we will use deep learning tools to tackle an interesting and complex problem of image processing called *style transfer*. Given a content image and a style image as inputs, the aim is to create a new image preserving the global structure of the content image but showing the artistic patterns of the style image. Before the renaissance of Artificial Neural Networks, early work in the field called *texture synthesis*, only transferred limited and repeatitive geometric patterns of textures. Due to the avaibility of large amounts of data and cheap computational resources in the last decade *Convolutional Neural Networks* and Graphics Processing Units have been at the core of a paradigm shift in computer vision research. In the seminal work of *Neural Style Transfer*, Gatys et al. consistently disentangled style and content from different images to combine them in artistic compositions of high perceptual quality. This was done using the image representation derived from Convolutional Neural Networks trained for large-scale object recognition, which make high level image informations explicit. In this thesis, inspired by the work of Li et al., we build an efficient neural style transfer method able to transfer arbitrary styles. Existing optimisation-based methods (Gatys et al.), produce visually pleasing results but are limited because of the time consuming optimisation procedure. More recent feedforward based methods, while enjoying the inference efficiency, are mainly limited by inability of generalizing to unseen styles. The key ingredients of our approach are a Convolutional Autoencoder and a pair of feature transform, Whitening and Coloring, performed in the bottleneck layer. The whitening and coloring transforms reflect a direct matching of feature covariance of the content image to the given style image. The algorithm allows us to produce images of high perceptual quality that combine the content of an arbitrary photograph with the appearance of arbitrary well known artworks. With the intention to understand this unconvential approach, before diving into the architectural and implementational details, we provide an introduction to basic theoretical concepts about machine learning as well as the necessary background notions on artistic style transfer.

Riassunto

In questa tesi verranno utilizzati strumenti di deep learning per affrontare un interessante e complesso problema di processamento delle immagini chiamato *trasferimento di stile*. Dati in input un immagine di contenuto ed un immagine di stile, lo scopo è di creare una nuova immagine che condivida i tratti stilistici dell'immagine di stile preservando la struttura complessiva dell'immagine di contenuto. Prima della rifioritura delle reti neurali artificiali, i primi lavori di ricerca nell'ambito della *sintesi di texture*, erano limitati nel riprodurre texture dagli schemi geometrici e ripetitivi. Grazie alla disponibilità di grandi quantità di dati e risorse computazionali a basso costo nell'ultima decade le reti neurali convoluzionali e le unità di processamento grafico (GPU) sono state al centro di un cambiamento di paradigma in tutti gli ambiti di ricerca legati alla visione artificiale. Il brillante lavoro di Gatys et al. che inaugurò il campo di ricerca chiamato *Neural Style Transfer*, fu il primo in grado di separare la rappresentazione dello stile e del contenuto da due immagini e ricombinarlo creando una composizione artistica di altà qualità. Questi risultati sono divenuti realizzabili grazie alla rappresentazione esplicita di alto livello delle immagini estratta da reti neurali convoluzionali allenate su larga scala per il riconoscimento di oggetti. Questa tesi, inspirata dal lavoro di Li et al., propone un nuovo ed efficiente metodo per il trasferimento di stili arbitrari. I metodi esistenti basati su ottimizzazione (e.g., Gatys et al.) producono risultati visivamente piacevoli ma sono limitati dal dispendio di tempo richiesto dalla procedura iterativa di ottimizzazione. Metodi più recenti basati su reti neurali feedforward godono

invece di un inferenza rapida ma sono limitati nella capacità di generalizzare a stili arbitrari su cui la rete non è stata allenata. Gli ingredienti principali del nostro approccio sono una rete Autoencoder convoluzionale ed un paio di trasformazioni, chiamate *Whitening* e *Coloring* applicate sulla rappresentazione latente. Queste trasformazioni statistiche inducono una corrispondenza diretta tra la covarianza della rappresentazione del contenuto e quella dello stile. Questo algoritmo ci permette di produrre immagini visimente piacevoli che combinino il contenuto di una qualsiasi fotografia con lo stile di una qualsiasi opera d'arte. Per comprendere le motivazioni di questo approccio non convenzionale al trasferimento di stile, prima di addentrarci nei dettagli architetturali ed implementativi, forniremo alcuni concetti teorici sul machine learning ed alcune nozioni sullo stato della ricerca sul tema del trasferimento di stile.

Chapter 1

Introduction

Elmyr de Hory was a Hungarian-born painter and art forger, who is believed to have sold over a thousand forgeries to reputable art galleries all over the world. The forger's skill is a testification to the human talent and intelligence required to reproduce the artistic details of a diverse set of paintings. In computer vision, much work has been invested in teaching computers to likewise capture the artistic style of a painting with the goal of conferring this style to an arbitrary photograph in a convincing manner.

Early work on this topic concetrated on visual texture synthesis using non-parametric models for "growing" textures one pixel at a time. Soon enough, it was found out (Efros *et al.* [5]) that one may transfer a texture to an arbitrary photograph to confer it with the stylism of a drawing. A second line of research focused on building parametric models of visual textures constrained to match the marginal spatial statistics of texture style images. In recent years these spatial image statistics extracted from intermediate feature of state-of-the-art image classifiers proved to be superior in capturing visual textures. Pairing a secondary constraint to preserve the content of an image, as measured by the higher level layers of the same image classification network, extended these ideas into the field of *Artistic Style Transfer* [7].

Convolutional Neural Networks [14] are the aformentioned classifiers. These type of model organize information about an image in a hierarchical and efficient way using concepts like: local receptive field, shared weights and linear downsampling. Nowdays, CNN represent the state-of-the-art for almost all image processing tasks.

Optimizing an image to obey the constraints mentioned above is computationally expensive and yield no learned representation for the artistic style. A line of research addressed this problem by building a secondary network, i.e. *style transfer network*, to explicitly learn the transformation from a photograph to a particular painting style. Although this method provide computational speed, much flexibility is lost. A single style transfer network is learned for a single painting style and a separate style transfer network must be built and trained for each new painting style. This approach avoid the critical ability to learn a *shared representation* across different styles.

In this thesis we proposed a simple yet effective method for *universal style transfer* using Convolutional Neural Networks. The transfer task is formulated as an image reconstruction process, with the content image features being transformed at an intermediate layer with regard to the statistics of the style image features. The signal whitening and coloring transforms (WCT) are used to match the content features to those of the style directly in the deep feature space. Transformed features are reconstructed back to RGB space by a symmetrical convolutional decoder network. The proposed method enjoys *learning freeness at test-time*, at the cost of training a general-purpose image reconstruction decoder in advance.

Chapters Overview

Chapter 1 briefly explains the scientific context and the objectives of this thesis.

Chapter 2 contains theoretical background about machine learning concepts used throughout the rest of the thesis. It also presents the differents research directions in the field of Artistic Style Transfer.

Chapter 3 describes the architecture of the proposed approach in details. Particular techniques like image reconstruction, upsampling, features whitening and coloring are analysed.

Chapter 4 explains the implementational choices made during the development process. This chapter explains the program functionalities and its user interface.

Chapter 5 showcase the stylization obtained with differents hyperparameters configurations. In the end, some performance remarks regarding execution time are given.

Chapter 6 draws concluding remarks on the work done and presents some future work directions.

Chapter 2

Background

This chapter is going to review some of the basic principles and architectures used in Machine Learning, giving notions useful throughout the rest of this thesis. It will also focus on the topic of style transfer reviewing the most important research efforts in the literature and how they contributed to the current state of the art.

2.1 Machine Learning

Machine Learning is a field of Artificial Intelligence suited for problems difficult to address by algorithmic means. Problems of this kind are: spam or fraud detection, recognition of object in an image or of the words in a sound recording. It is essentially a form of applied statistics trying to estimate complicated functions using the knowledge extracted from input data.

The concept of learning in [19] is defined as: "A computer is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at task in T, as measured by P, improves with experience E". Experience is formalized as a set of examples, in turn each composed of a set of features that describe the relevant properties of the example. The implementation consist of a statistical model with a fixed number of parameters, that given an example as input, produce an output value. The perfomance of the model is evaluated by an error measure that compute the distance between the model's output and the correct output. The model's parameters are optimized with respect to the perfomance metric in order to obtain increasingly better results.

This process of automatic learning is useful if our learnt approximation function can perform well not only on already-seen training data but especially on unseen real-world data, thus, achieving the *generalisation* propriety. If the input examples are independent and identically distributed (denoted i.i.d.) the *inductive learning assumption* guarantees us generalisation when the experience data feeded to the model is sufficiently large. This computational exstensive data-driven process is feasible nowdays thanks to the hardware improvement of the last decade.

A distinction between learning algorithms concerns how the experience data, also called *dataset*, encodes the function to be approximated. From that perspective, we have **supervised learning**, when the target function is completely specified by the training data in the form of associated *labels*. In an **unsupervised learning** setting instead, the system will learn to uncover patterns and find groups in the dataset which contain no explicit description of a target concept. Unsupervised learning involves observing several examples of a random vector of feature x and attempting to implicitly or explicitly learn the probability distribution $p(x)$ that generated the dataset; while supervised learning involves observing several examples of a random vector of features x and an associated value or vector y, then learning to predict y from x, usually by estimating the conditioned probability $p(y|x)$.

The process of learning can be seen as a search in a function space $H = h \mid h : X \longrightarrow Y$ for the function h that fits better the not-known target function f according to the error measure. If the model h is too complex and specialized over the peculiarities of the instances of the training set we are in a situation of **overfitting**; instead if the model is too simple and does not

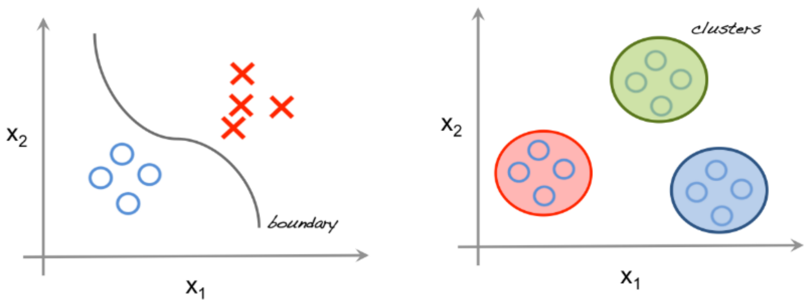


Figure 2.1: Supervised learning (left) find the best fitting line to categorize data points vs Unsupervised learning (right) discover structural properties of the feature space as grouping (also called *clusters*).

allow to express the complexity of the observations (low *capacity* model) we are in a situation of **underfitting**. This is illustrated by Figure 2.2

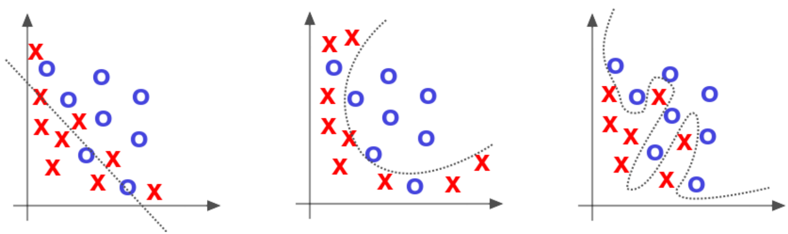


Figure 2.2: Approximation possible outcomes: underfitting (left), desired (center), overfitting (right).

2.1.1 Artificial Neural Networks

Under the field of Machine Learning there is a family of models that take their inspiration from the way the brain is configured. The brain is composed by a huge amount (around 10^{11}) of neurons connect together in a very big and intricate and network. Neurons send informations to others neurons through an axon and receive informations through structures called synapses (approximately 10^3 synapses per neuron). Artificial Neural Networks (ANN), also just called Neural Networks, use this connectionist approach for decision making and learning.

The basic unit of a Neural Networks is an artificial neuron. Its main job is to compute a weighted linear combination of the inputs received from other neurons. The result is then passed into a particular nonlinear function called *activation function*. The result of such function is the output of a neuron and gets immediately propagated to others. The idea of an artificial neuron was developed in the 1950s and 1960s by the scientist Frank Rosenblatt [21], inspired by earlier work by Warren McCulloch and Walter Pitts [18].

The first artificial neuron was called *perceptron*. It had binary inputs and a binary output computed by a step function that emitted 1 if the input was larger than some fixed threshold. Networks made of perceptrons were expressive enough to calculate every boolean function. The problem of the perceptron was its mathematical instability when updating the weights due to the step function shape.

Research on the field led to better perfoming activation functions. In particular, the *sigmoid function* has a shape similar to a threshold function but has the benefit of derivability in all its real-valued domain. Derivability plays a major role in the optimization of such neural networks as we will see shortly.

The principal kind of Artificial Neural Networks are **feedforward** models. The neuron's topology of such models is a directed directed acyclic graph (i.e., no loops and backward connections). Among feedforward models, the Multi Layer Perceptron (MLP) is the most basic architecture. We can see an example of it in Figure 2.3. Neurons are grouped in layers, which are stacked one next to each other. The input values flows forward from the input layer to the output layer triggering the computation at each layer. The expressive power of ANN comes from the nonlinearity of the activation function, allowing MLP with hidden layers to approximate any nonlinear function (**universal approximation theorem** [9]).

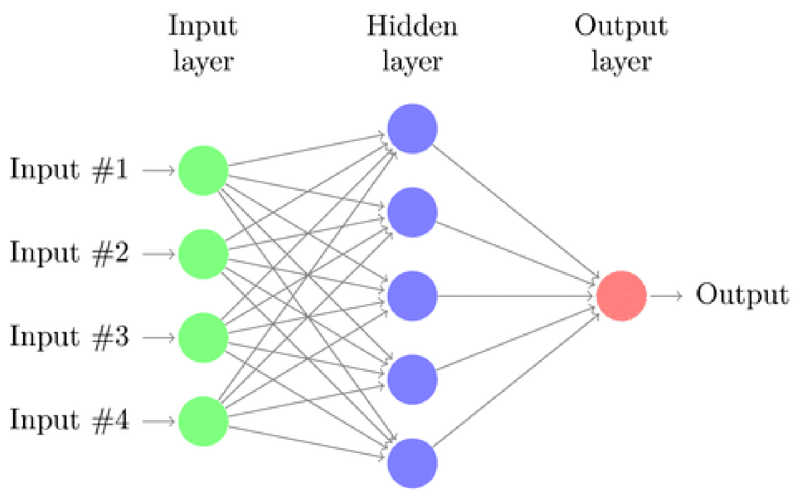


Figure 2.3: Example of a feedforward neural network with 4 neurons in the input layer, 5 neurons in the hidden layer and only one neuron in the output layer. Such small ANN can be trained to perform tasks like *binary classification* or *logistic regression*.

The largest difference between linear models and ANN regards training procedure and the fact that the nonlinearity of neural networks causes the loss function to be nonconvex. This means that neural networks are usually trained by using iterative gradient-based optimizer algorithms that drive the loss function to a very low value. **Stochastic Gradient Descend** is the most used optimization algorithm of this kind. It doesn't have a convergence guarantee and it is sensible to the initial configuration of the model's parameters.

The aformentioned optimization algorithm repeats a two phase cycle: forward propagation and parameters update. After evaluating the model on some input vector, the derivatives of the loss function with respect to the model's parameters are computed at each neuron. The optimizer implement this by calling an internal procedure called **back-propagation** that use the chain rule of calculus to calculate derivatives efficiently. These derivatives, also called *gradients*, are used by the optimizer to adjust the model's parameters in the attempt to find a global minimum of the loss function.

2.1.2 Deep Learning

Very simply, a neural network with more than one hidden layer is called *deep*. Deep models in the recent years outperformed the former state of the art in a lot of tasks. We will briefly investigate the reason of this improvement and later we will consider various deep architectures.

The *curse of dimensionality* is the problem that arise when dealing with high dimensional random variables that can have an exponantial number of configurations. Since the order of possible configurations is computationally intractable, a good generalisation on unseen inputs seems to be very difficult if not an impossible goal to achieve [1].

Deep learning exploits the powerful prior of *compositionality* allowing complex features to be computed in terms of simpler ones in a hierarchical organization. These models are powerful and flexible approximators and are capable, for example, of understanding text or images but they need a lot more training data than "shallow" networks.

Training deep models is not an easy task for various mathematical problems like vanishing gradient, local minimums and overfitting. *Dropout* is a technique that usually helps alleviate these problems by removing with a fixed probability some neuron connections temporarily.

2.1.2.1 Convolutional Neural Networks

In the last decade these kind of deep neural networks gained popularity and established themselves as the gold standard for AI tasks regarding computer vision. The key ingredients of their expressiveness and efficiency are: *loose connectivity* (a neuron is only influenced by a small subset of adjacent neurons) and *shared weights* (every neuron act as a convolutional filter performing the same operation on different areas of its input).

If we were to process an image of size $n \times n$ with a MLP we would have to use one neuron per pixel for a total of n^2 neurons and n^3 parameters in the input layer. Local connectivity drastically reduce the number of parameters improving efficiency and reducing overfitting risk. This approach works really well on images because the relevant characteristics of a photo are usually formed by pixels near to each other.

Performing a convolution, as shown in Figure 2.4, can also be seen as applying a filter to an image. During training we let the network learn the right sequence of filters, adjusting weights during backpropagation phase, to perform better for the task at hand.
A convolutional filter work on all input channels combining them in one feature map. If we consider the output of all the convolutional filters inside a convolutional layer we are dealing with a *feature volume* that can take up a lot of memory for big input images.

It is common to periodically insert a *pooling layer* between successive convolutional layers. Its goal is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. This layer operates by sliding like a convolutional filter and resize the input spatially using a simple max or average operation on a group of adjacent pixels.

CNN for image recognition tasks have a few fully connected layers at the end for making the classification based on the features extracted from the convolutional layers as we see in Figure 2.5. These nets ouput a confidence score for each category and the highest is selected as the predicted class.

2.1.2.2 Autoencoders

An autoencoder is a neural network trained to reconstruct the input data out of a learned internal representation. Internally, it is composed of an encoder function $h = f(x)$ which produce a compact representation of the input,

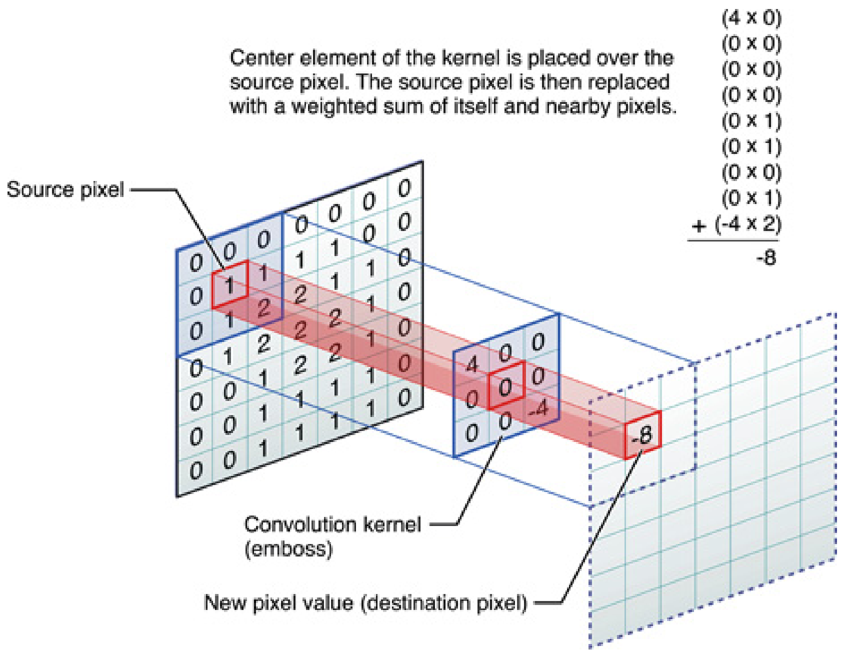


Figure 2.4: Application of a convolutional kernel to an input image. Input pixels are linear combinated with the kernel's parameter. This operation is repeated by sliding the kernel all over the input image. The size of the output image, also called *feature map*, is determined by a few hyperparameters of the convolutional layer. These hyperparameters are: the size of the kernel (**receptive field**), the number of pixels shifted when moving the kernel (**stride**) and the artificial enlargment of the input to allow kernel application on the borders (**padding**).

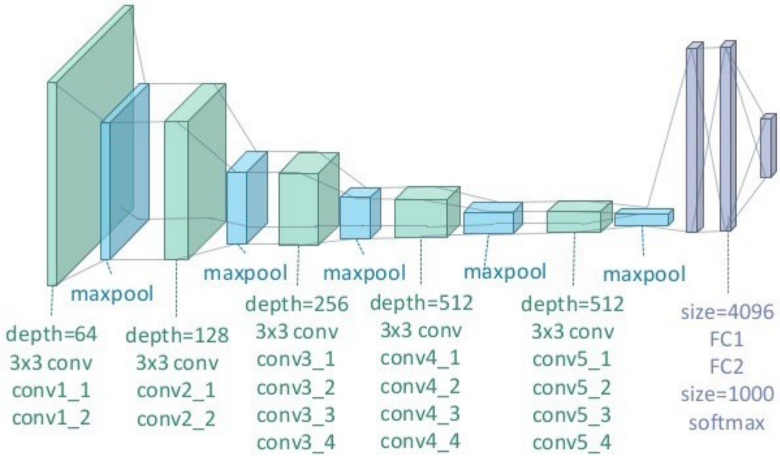


Figure 2.5: Representation of the VGG19 architecture [22] features volume evolution. In green we have convolutional layers, in blue max pooling layers and in purple fully connected layers.

called **code**, and a decoder function $r = g(h)$ that attempts the reconstruction of the input to the original size.

They are trained in a *self-supervised* way, usually using the quadratic reconstruction (input-output difference) error, to perform a form of lossy data compression that works well for data with strong correlations. AE are data-specific, which means that they will only be able to compress well data similar to what they have been trained on.

Reconstructing the input that we already have is not the main goal of the autoencoders. Since they are forced to prioritize which aspects of the input to encode, they often learn useful properties of the data. In fact, the low-dimensional representation learned by an autoencoder is an approximation of the Principal Component Analysis (PCA). Autoencoders for different tasks (e.g. *dimensionality reduction*, *data denoising*) usually have a different architecture.

Since this thesis is about image processing, we will focus on autoencoders composed of a convolutional encoder and decoder (see Figure 2.6). Convolutional Autoencoders (CAE) are state-of-art tools for unsupervised learning of convolutional filters. Features extracted by these filters can be later used to perform any task that requires a compact representation of the input, like style transfer.

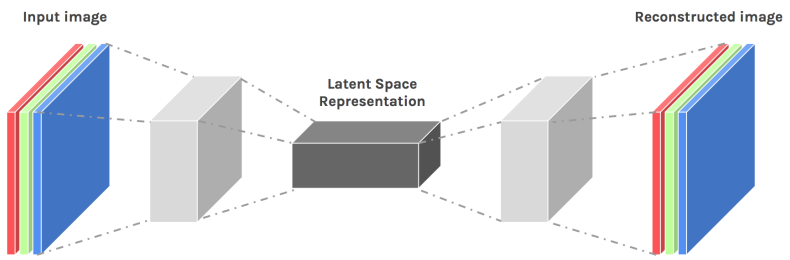


Figure 2.6: Example of a fully convolutional autoencoders working on a RGB image. The convolutional decoder can use deconvolutions (also called *traspose convolutions*) or upsampling layers to upsample the feature maps back to the original input size.

2.2 Artistic Style Transfer

Style transfer is a topic in the field of computer vision researching the interplay between the content and the style of images. The goal of the discipline is to produce an output image that exhibit the desired style (for example, a famous painting) while preserving the *semantic content* of an input image. Artistic style transfer, in the beginning, was considered to be a generalisation of *texture synthesis*, which study how to extract and transfer the texture from a source image to a target image. Results were usually not impressive because only low-level features were captured and transferred.

2.2.1 Neural Style Transfer

Recently, inspired by the power of Convolutional Neural Networks (CNN), Gatys *et al.* [7] in their seminal work, reproduced with great success the style of famous paintings on natural images. They proposed to represent the *content* of a photo as the features responses from a pre-trained CNN, and to model the style of an artwork as the summary features statistics. The second-order (Gram-based) statistics they used were able to model a wide varieties of both natural and non-natural textures. These statistics represent the correlation between filters responses in differents layers of the same pre-trained CNN.

Looking at Figure 2.7 the scientific offsprings following Gatys *et al.* can be distinguished on how the output image is constructed.

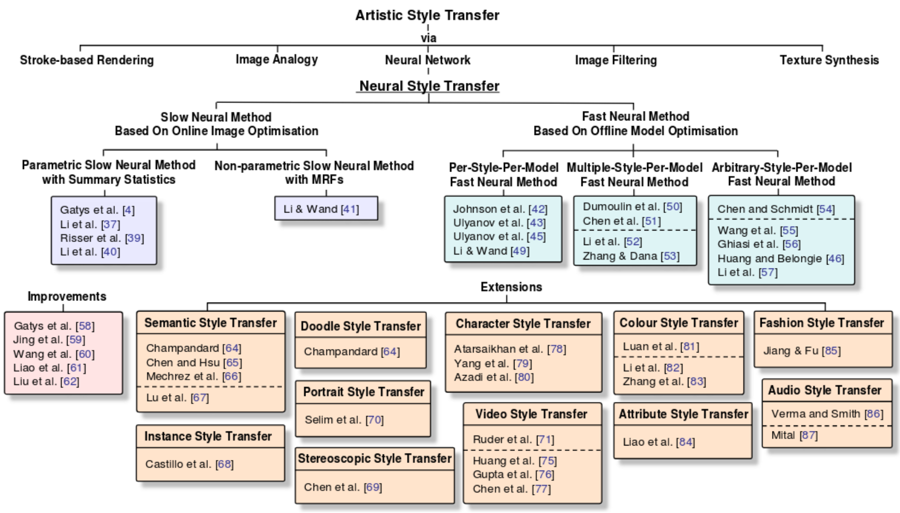


Figure 2.7: A taxonomy of artistic style transfer techniques. For the bibliographic references see [10].

"Slow" methods transfers the style by iteratively optimising an image. Starting from an initial random noise image, these techniques perform gradient-descend optimizing a loss function often composed by a content-related compontent and a style-related compontent. "Faster" methods address the efficiency issue by putting the burden on the training stage. In fact, at testing stage the stylization is obtained quickly by a single feed-forward sweep at the cost of training in advance a model on a large set of content images **for one or more style images**.

Depending on the number of artistic styles a single network can reproduce, these "Fast" methods are further subdivided into three categories: single style, multiple styles and **arbitrary style**. The approach we consider in this thesis belong to the latter category and it is inspired by the recent work of Li *et al.* [16]. It exploits a series of feature transformations in order to transfer an arbitrary style in a learning free manner.

In the proposed approach, the first few layers of a pre-trained CNN network are used as an encoder and the corresponding decoder is trained for image reconstruction. A pair of feature transforms called Whitening and Colouring Transformations (WCT) are applied on the feature maps between the encoder and decoder. Denoting the content image I_c and the style image I_s the stylised output I is computed as follows: $I = Dec(WCT(Enc(I_c), Enc(I_s)))$. The algorithm is built on the observation that the whitening transformation can remove the style related information from the content image while preserving the overall structure.

Therefore, receiving content activations $Enc(I_c)$ from the encoder, whitening transformation can filter the original style out of the content features and return a filtered representation with only content information. Then, by applying colouring transformation, the style patterns contained in $Enc(I_s)$ are incorporated into the filtered content representation, and the stylised result I can be obtained by decoding the transformed features.

Chapter 3

Architecture

This chapter will cover in great details the structure of the model proposed by Li *et al.* [16], explained briefly in the end of the previous chapther. A few sections of this chapter require some notions of statistics and linear algebra.

3.1 Style representation

The key challenge in Style Transfer is how to extract effective representation of the style and then match it to the content image. Convolutional Neural Networks have proved to be very effective at capturing characteristics of images. Thus, after the seminal work of Gatys *et al.* almost all successive approaches used the features extracted by convolutionals filters as the representation of the images.

The representation encoded by the feature maps of a CNN is usually summarized in a statical form of features correlation called *Gram Matrix*. Considering a feature volume as in Figure 3.1, we first reshape the tensor into a $H \times W$ grid of C-dimensional vectors. The outer products between a pair of these vectors gives a $C \times C$ matrix measuring features co-occurence of filters at these two positions. Averaging over all such matrices we get a resulting Gram Matrix of shape $C \times C$.

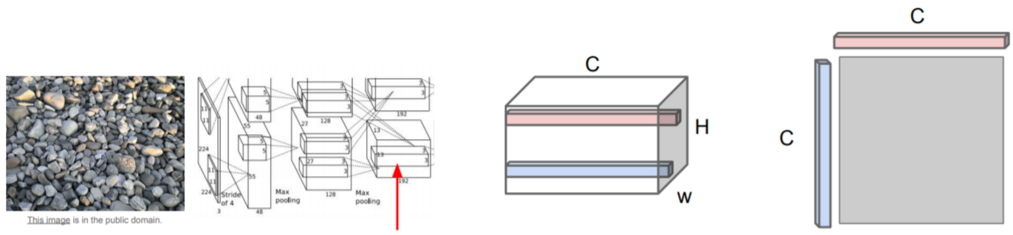


Figure 3.1: The feature volume is a tensor of shape $C \times H \times W$. The Gram Matrix $G \in \mathbb{R}^{C \times C}$ is computed as $G_{ij} = \sum_{k=1}^{C} F_{ik} F_{jk}$ for all filter activations F_i. [6]

3.1.1 Covariance matrix

The Gram matrix in [15] is empirically shown to be sensitive to features scale and poses difficulty in capturing heterogeneous style statistics. Motivated by this observation, we modify the original Gram matrix computation by subtracting $\overline{F}$ before calculating the outer products, where $\overline{F}$ is defined as the mean of all activations in the current layer of the convolutional net. This newly obtained matrix is namely the *normalized covariance matrix* [2], whose elements gives an estimate about how much filters activations in all feature maps share similar behaviour and variation.

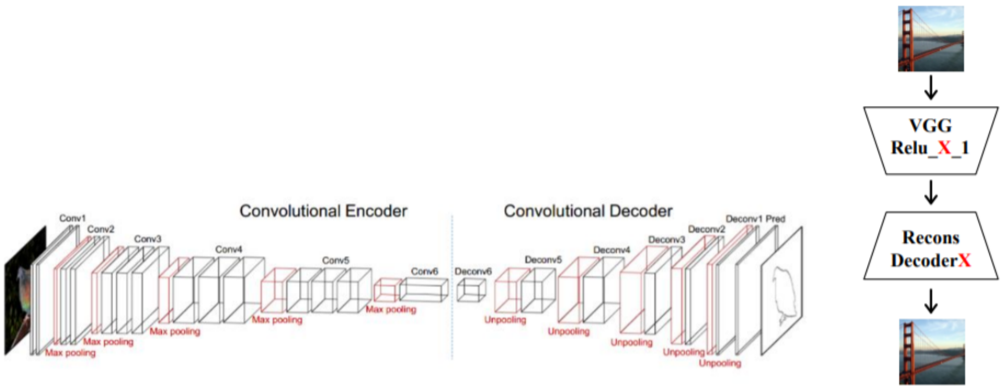
3.2 Image reconstruction

Despite the recent rapid progress in Neural Style Transfer, existing methods often trade off between generalization, quality and efficiency, which means that optimization-based methods can handle arbitrary styles with pleasing visual quality but at the expense of high computational costs, while feed-forward approaches can be executed efficiently but are limited to a fixed number of styles or compromised visual quality.

The proposed approach formulate the task of style transfer as an image reconstruction process, with the content features being transformed, at intermediate layers, with regard of style features in the midst of feed-forward passes. This simple yet effective method enjoys style-agnostic transfer at the cost of marginally compromised visual quality and execution efficiency.

The proposed architecture employ the notorious VGG19 CNN trained for the ImageNet [4] recognition task as the feature extractor. Another net is needed for inverting features back to the RGB space. This net (see Figure 3.2b) is a symmetrical VGG19 decoder responsible for the reconstruction part.

The overall architecture is a general-purpose image reconstruction Convolutional Autoencoders. We can visualize it Figure 3.2a.



(a) The overall VGG19 convolutional autoencoder architecture used.

(b) The input image reconstruction task on which the autoencoder is trained.

3.2.1 Upsampling

To enlarge a input feature map produced by a CNN to a greater spatial extent the natural solution seems to be the use of *traspose convolutional layers* (also called *deconvolutions* or *fractionally-strided convolutions*). These layers learn to do the reverse operation of the convolution. They take each pixel of the input image, multiply it by all the values in a $n \times m$ kernel to get a $n \times m$ weighted kernel to put in the output image. Where these kernels overlaps in the output image the values are simply summed.

The need for transposed convolutions generally arise from the desire to have a convolution's complementary operation, in order to go **from** something that has the shape of the output of some convolution, **to** something that has the shape of its input while maintaining a connectivity pattern compatible with said convolution. In our use case, we would like to employ such operation in the decoder architecture.

Unfortunately, deconvolutions can easily have *uneven overlap*, in particular this happens when the kernel size (the output window size) is not divisible by the stride (the spacing between elements in the input). This leds into putting more output values in some places than others, generating bright colors and **checkerboard artifacts**.

These artifacts are really hard to avoid completely with traspose convolutions and doing so, often involves sacrificing some of the model's capacity by posing restriction on the set of possible filters [20]. In addition, the presence of these artifacts doesn't depend on the type of training done on the deconvolutional layers but it is intrinsic to the method.

Another approach to the problem, is to separate upsampling to a higher resolution from convolution to compute features. For example, you might resize the image using some linear techinque of interpolation and then do a convolutional layer. This seems to discourage high-frequency artifacts really well in a variety of training settings, for example, in Generative Adversarial Networks.

For of this motivations, the decoder designed in this thesis employ *nearest neighbour upsampling* layers for enlarging feature maps followed by standard convolutional layers.

Figure 3.3: The neural style transfer by Johnson *et al.* [11]. Stylization suffer from checkerboard artifacts (i.e. high-frequency repeating patterns) in the stylized images on the bottom.

3.2.2 Decoder training

Since successive layers of a Convolutional Neural Network encode informations about an image at different levels of abstraction, it is also useful to experiment style transfer at different levels in a similar manner. For this reason we divide the VGG19 network in five blocks as denoted in Figure 3.2a. Each successive block is a sequence of convolutional and max pooling layers up to some fixed depth. These blocks are denoted "Relu_X_1" where $X = 1, \ldots,\ 5$ is indicating the depth of the block and "Relu" stands for *Rectified Linear Unit*, which is the activation function employed by VGG19.

Given the features extracted by such blocks, the authors trained accordingly five decoder blocks for image reconstruction. The loss function used for training is composed by a *reconstruction loss* and a *feature loss.* It is defined as follows:

$$L = \|I_o - I_i\|_2^2 + \lambda\|\phi(I_o) - \phi(I_i)\|_2^2 \tag{3.1}$$

where I_i, I_o are the input image and reconstructed output and ϕ is the VGG encoder that extracts the features. In addition $\{\lambda \in \mathbb{R} \mid 0 \leq \lambda \leq 1\}$ is a hyperparameter to balance the two losses.

Another important aspect of the training procedure is the dataset used. In order to guarantee universal style transfer, the model, in addition to have a great approximation *capacity*, needs also to be able to invert features extracted from a wide variety of inputs. To enable such general-purpose image reconstruction task, the COCO (Common Objects in COntext) [17] large-scale dataset was chosen for training. Created in 2014, up today it counts over 330.000 images containing complex everyday scenes with (91 different categories of) objects in their natural context.

3.3 Features transformations

This section will explain the transformations performed in the deep feature space. The input of these transformations are the feature activations of the *bottleneck layer*, located in the middle between the encoder and the decoder part of the convolutional autoencoder. The goal of these transformations is to combine the content with the characteristics extracted from the style in a visually pleasing way.

The type of transformations we will use are composed of two steps and are called Whitening and Colouring Transformations (WCT). These transformations reflect a direct matching of feature covariance of the content image to the style image, transforming the extracted content features such that they exhibit the same statistical characteristics as the style features. They are able to achieve this goal in an almost effortless manner compared to the optimization of the Gram matrix-based cost.

3.3.1 Data whitening

Normalization is a data preprocessing technique useful in many machine learning applications. The reason comes from the fact that important patterns in the data often correspond to the *relative relationships* between the different input dimensions. Therefore, you can make the task of learning and recognizing these patterns easier by removing the constant offset and standardizing the scales. This principle gets more important when dealing with real-world data that usually have a large number of dimensions. There can now be dependencies among the dimensions, which we can think of as patterns within the data. For example, consider an image of a blue ocean (Figure 3.4) that is missing a single pixel, the value of that pixel is not independent of the nearby pixels; it is almost certainly blue.

The degree of linear dependence between the dimensions is captured by the covariance matrix $cov(X) = \Sigma$ of the input data. Σ is a symmetric $D \times D$

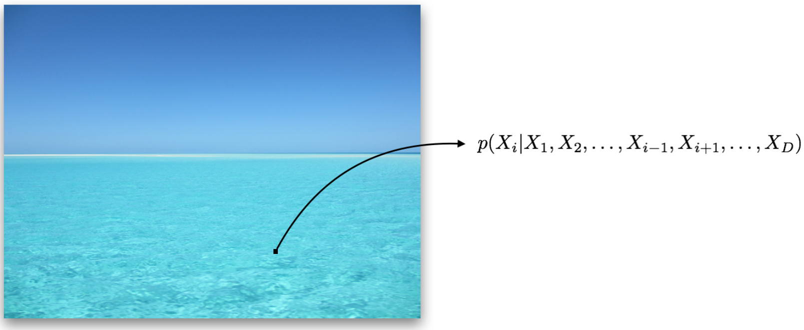


Figure 3.4: In order to determin the value of the missing pixel X_i we have to consider the *joint probability distribution* of all the D input dimensions. Using the chain rule of probability this rewrites as the product of conditional probabilities $p(X) = p(X_1, X_2, \ldots, X_D) = p(X_1)p(X_2|X_1)\ldots p(X_D|X_1, X_2, \ldots, X_{D-1})$.

matrix where $\Sigma_{i,j}$ contains the covariance between dimension i and dimension j. The diagonal entries of this matrix contains the variance of each dimension. Normalization of multi-dimensional variables is called *statistical whitening*. Whitening is a linear transformation that transforms a vector of random variables X with a known covariance matrix Σ into a vector of new variables Z whose covariance is the identity matrix $cov(Z) = I$, meaning that they are uncorrelated and each have variance 1 (see Figure 3.5). The transformation is called "whitening" because it modify the input vector to resemble a white noise vector.

Formally, on input a d-dimensional vector $X = (X_1, \ldots, X_D)$ the result of the whitening Z is computed as:

$$Z = W(x - \mu) \tag{3.2}$$

The input x is centered by subtracting the mean vector μ and it is multiplied by a $D \times D$ whitening matrix W. Such matrix should satisfy the conditions:

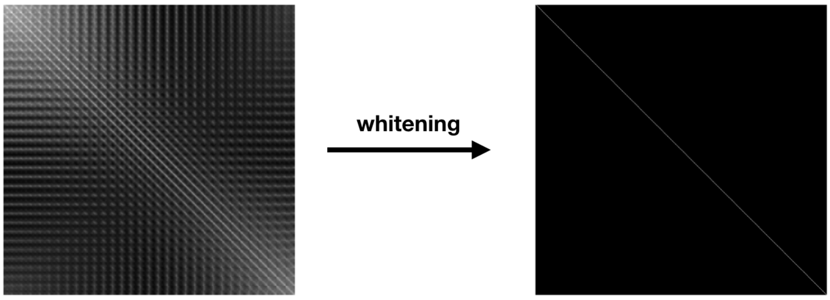


Figure 3.5: Left: the covariance matrix of the 3,072-dimensions (32x32x3) images in the notorious CIFAR10 dataset [13]. Black indicates a low value, white a high value. Dimensions are highly inter-dependent. Right: the covariance matrix after the whitening procedure. Dimensions are now completely uncorrelated.

$$
\begin{aligned}
&cov(Z) = I \\
&ZZ^T = I \\
&(W(X-\mu))(W(X-\mu))^T = I \\
&(W(X-\mu))(X-\mu)^T W^T = I \\
&W\Sigma W^T = I \\
&W\Sigma W^T W = W \\
&W^T W = \Sigma^{-1}
\end{aligned}
\tag{3.3}
$$

However these conditions aren't very restrictive on W and there are actually infinetely many choices of W. This is demonstrated by the fact that taking an orthogonal matrix Q (i.e. $QQ^T = 1$), if we put $W = Q\Sigma^{-1/2}$ we get:

$$
W^T W = (Q\Sigma^{-1/2})^T Q\Sigma^{-1/2} = (\Sigma^{-1/2})^T Q^T Q\Sigma^{-1/2} = (\Sigma^{-1/2})^T I \Sigma^{-1/2} = \Sigma^{-1} \tag{3.4}
$$

that is independent of the choice of Q. From a geometrical standpoint this phenomenon is called *rotational freedom*. Kessy *et al.* in [12] discussed the optimality of various choices for W. Among these, Zero-phase Components Analysis (ZCA) whitening maximize the average cross-covariance between each dimension of the whitened and original data and uniquely produces a symmetric cross-covariance matrix $\phi = cov(Z, X)$. Roughly speaking, this means that it is the method preserving most of the informations in the original data minimizing the total squared distance $||X - Z||^2$.

ZCA whitening choice for W is $W = \Sigma^{-1/2}$. The covariance matrix Σ by definition is assumed to be symmetric and positive semi-definite. Thus, it is possible to obtain an eingendecomposition $\Sigma = EDE^T$ where D is the diagonal matrix containing the eingenvalues of Σ and E is the orthogonal matrix containing its eingenvectors on the columns. The inverse square root matrix of Σ is: $\Sigma^{-1/2} = ED^{-1/2}E^T$ where the exponentation is computed element-wise.

The overall ZCA whitening computation used in this thesis proceeds as follows:

$$\hat{f}_c = E_c D_c^{-1/2} E_c^T f_c \tag{3.5}$$

where (i) f_c are the input content image features, (ii) the multiplication for the eingenvectors matrix E_c^T remove correlation between the components, (iii) the multiplication for the scaled eingvalues $D^{-1/2}$ normalize the components to unit-variance and (iv) the final multiplication for E_c "rotate-back" data from the eingenbasis space to the features space.

The example in Figure 3.6 show a ZCA whitened image. It visually resemble the input image with a "washed out" effect, which remove informations related to colors and style but preserves edges and semantic content of the image. This follows from the optimality of the ZCA whitening method in maximazing the cross-covariance between the original and the whitened features.

Figure 3.6: Reconstructed whitened features. The features from VGG19 Relu_4_1 are whitened and then decoded back to original size. The whitened image maintains global content structure but style is removed (e.g., the stroke patterns of the *Starry Night* are gone in the whitened image).

3.3.2 Coloring

The coloring transformation shares similar spirit with the whitening transformation. In fact, it can be seen as an inverse of the whitening transform. It is applied to confer to the whitened features the patterns of the style image. Coloring transforms the white noise vector $\hat{f}_c$ into a random vector matching the mean and the covariance matrix of the style features f_s.

The first step is always to zero-center f_s by subtracting its mean vector. Then we compute its covariance matrix $\Sigma = f_s f_s^T$ and perform an eigendecomposition yielding $\Sigma = E_s D_s E_s^T$ as in the whitening procedure.
The overall coloring computation used in this thesis proceeds as follows:

$$\hat{f}_{cs} = E_s D_s^{1/2} E_s^T \hat{f}_c \tag{3.6}$$

Then, we re-center $\hat{f}_{cs}$ summing the mean vector. The colored result $\hat{f}_{cs}$ will have $\hat{f}_{cs} \hat{f}_{cs}^{\,T} = f_s f_s^T$ as wanted.

To demonstrate the effectiveness of Whitening and Colouring Transformations, we compare it with *histogram matching*, a commonly used feature adjustament techinque. An image histogram is a chart that acts as a graphical representation of the tonal distribution in a digital image. It plots every tonal value (i.e. RGB value) present in the image on the x-axis and its frequency (i.e. number of times it appears) on the y-axis. Given two images, a reference image and a target image, their histograms are calculated. Then, the *cumulative distribution functions* [3] of the two histograms are computed and used to derive a mapping for every color intesity value. Then, this mapping is applied on each pixel of the reference image.

The channel-wise histogram matching [8] method determines a mapping function such that the mapped f_c has the same cumulative histogram as f_s. As shown in the comparison of Figure 3.7, it is clear that the HM method helps in transferring the global color of the style image but fails to capture salient visual patterns. In fact patterns are broken into pieces and local structures are misrepresented. In contrast, the proposed WCT method captures patterns that reflect the style image better. This can be explained by the fact that the HM method does not consider the correlations between features channels, which are exactly what the covariance matrix is designed for.

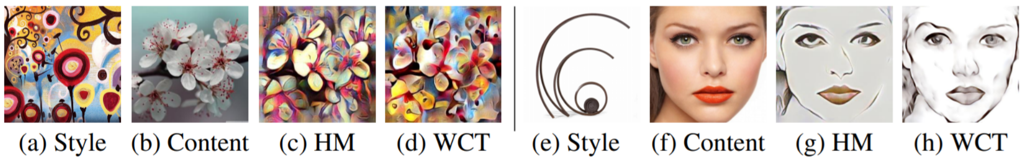


Figure 3.7: Comparison between HM and WCT coloring strategies on two differents content-style image pairs.

3.4 Stylization pipeline

So far we have developed a procedure for stylizing some reference content image to another target style style. We also know that succesive layers of a Convolutional Neural Networks represent the input at an increasingly higher level of abstraction: low-level features like edges and colors are encoded in the first layers and higher-level features like faces and objects are encoded in the last layers. This can be explained by the increasing size of receptive field and feature complexity in the network hierarchy.

A lot of Neural Style Transfer methods leveraged this hierarchical organization to fully capture the characteristics of style resulting in an enhanced stylization. For this purpose, it was developed an additional multi-level stylization pipeline. The workflow is illustrated in Figure 3.8. The content image C is input to the system only one time at the beginning. The style image S instead, is input to the system multiple times at every VGG19 block.

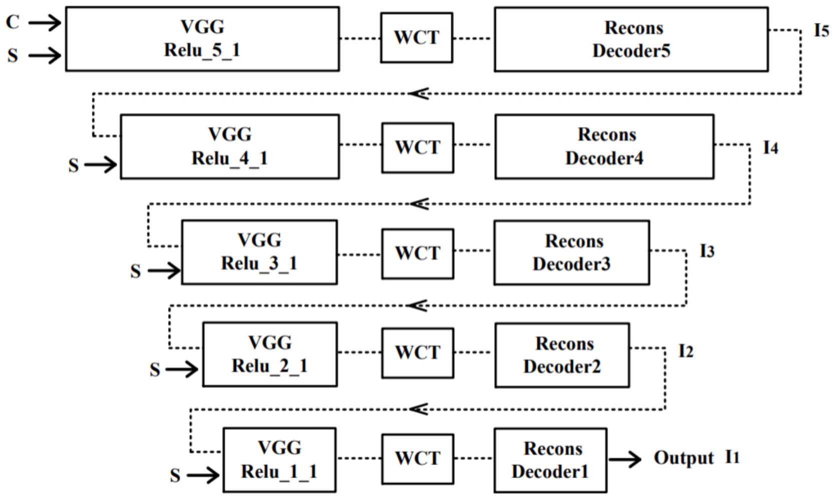
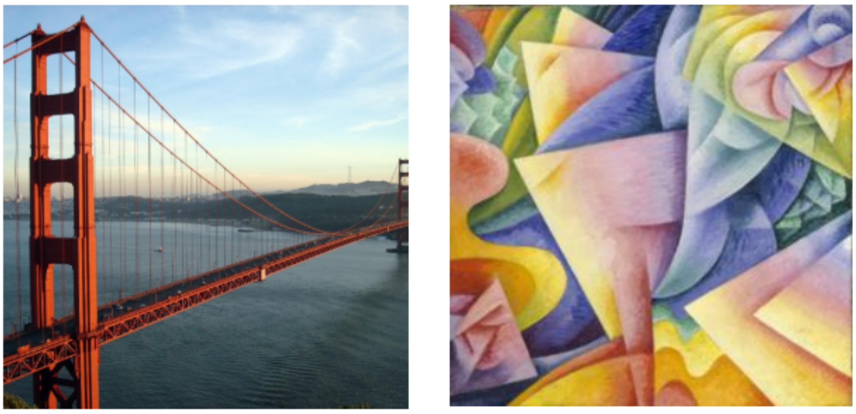
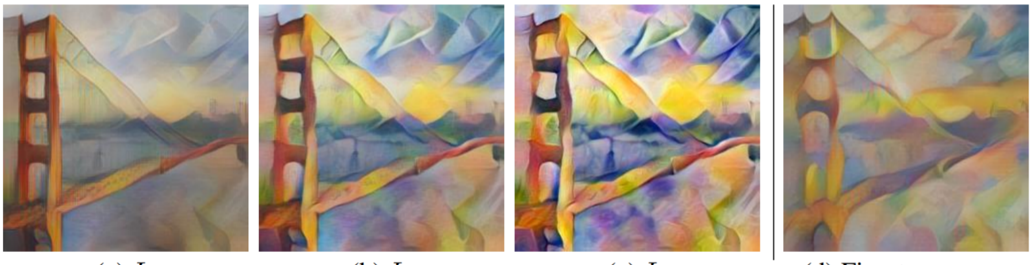


Figure 3.8: Multi-level stylization pipeline schematic. The result obtained by matching higher level statistics of the style is treated as the new content to continue to match lower-level information of the style.

The preffered block-ordering is **descendent**: the first coarse stylization I_5 is obtained from Relu_5_1 and it is regarded as the new input content image to the next stylization level that adjust lower-level features of the style. Intermediate results (I_x $x = 5, 4, 1$) are shown in Figure 3.9. These results have the expected behaviour: higher layer features first capture the main aspects of the style and lower layer features further improve details.



(a) Content image (b) Style image



(a) I_5 (b) I_4 (c) I_1 (d) Fine-to-coarse

Figure 3.9: The first three images show intermediate results of the multi-level *coarse-to-fine* stylization pipeline. From the left respectively: I_5, I_4, I_1. The last image on the right is a *fine-to-coarse* stylization. This approach fails because low-levels information cannot be preserved after manipulating higher-level features.

Chapter 4

Implementation

This chapter will discuss some of the implementational choices made during the developing process of the proposed style transfer techinque. It will also give a practical explanation about the functionalities of the program and the purpose of the user arguments. Therefore, this chapter will contain a few source code snippets with syntax highlighting. All the source code is open-source and available at `https://github.com/pietrocarbo/deep-transfer`.

4.1 Framework

The programming language of choice for the project is **Python** (v3.6). Launched in 1991, it had become the de-facto standard for almost all Data Science related disciplines as machine learning and computer vision. It is a dynamically-typed, object-oriented, interpreted language with several functional features. Python's dynamic nature and simple syntax make it perfect for fast prototyping. In fact, Data Science development process requires fast iterations with focus on data and algorithms. Moreover, Python's open-source libraries ecosystem is mature and solid. It offers packages for almost all math and data-processing needs.

To install the needed software packages we need another tool called a *package*

manager. It is a command line application with a central repository of available packages. The developer can use a package manager to install, update or remove specific packages in his system. For this project, we used the Anaconda Python distribution that offers the largest data-science collection of packages in its repository. This distribution comes with the **conda** package manager, which we used to download and install packages in a project-specific virtual environment.
Maybe the most important choice about the development environment regarded the *machine learning framework* to use. Even sticking only to Python, several options remain available. Keras framework it is essentially a simpler wrapper API over a more complex framework (e.g. Tensorflow). For this reason it doesn't offer a lot of flexibility when implementing low-level operations in-between layers of a neural net. Since performing WCT transformations in the bottleneck layer was a requirement of the project, Keras was discarded from the options.

The choice, in the end, lied on the **PyTorch** framework, developed by the Facebook Artificial Intelligence Research (FAIR) group. At its core PyTorch provides tensor computations with GPU acceleration and deep neural networks with automatic differentiation. Another important aspect is its tight integration with the Python language which make it feel more native most of the times. In fact, it shares a very similar API syntax with the notorius NumPy library for scientific computation. PyTorch is still in beta-version (v0.4.1) but has reached an important level of maturity and it's quickly gaining momentum in the research community.

4.1.1 Dynamic Computational Graph

Another possible framework choice was Tensorflow, developed by Google Brain. It offers many of the functionalties of PyTorch; it became the standard for production environments and it also offers support for mobile applications. Both aformentioned frameworks operate on tensors and view models as a directed acyclic graphs (DAG) but they differ drastically in the construction of such graphs.

In TensorFlow you define graph statically before a model can run. All communication with outer world is performed via special objects as `tf.Session` and `tf.Placeholder` which are tensors substituted by external data at runtime. In PyTorch things are way more imperative and dynamic: you can define, change and execute nodes of a neural net as you go, without session interfaces or placeholders.

Basically, all deep learning frameworks maintain a *computational graph* that describe the exact order and number of operations that need to be performed. Many deep learning framwork (e.g. Tensorflow) follow a "Define and Run" philosophy, building a static data-flow graph in advance and later feed data to it. With this approach, many optimizations regarding memory allocation and data parallelism are straightforward. On the other hand, newer frameworks as PyTorch follow a "Define by Run" philosophy, where the order and number of computations is programmatically defined. This is useful when dealing with inputs of variable size, when building Recurrent Nets used widely for Natual Language Processing tasks and, in general, whenever we want flexibility in the feedforward pass.

In PyTorch this flexibility is obtained through the Dynamic Computational Graph abstraction. In the context of this thesis's project, this enabled to embed control-flow statements in the convolutional autoencoder's computational graph.

4.2 Lua dependencies

We already said that the proposed approach is training-free at test time. This is because the authors of the paper pretrained the five needed decoders for image reconstruction. They implemented the architecture in the Lua using the Torch framework. Therefore, the trained models were exported in the serialization format of Torch which is **`t7`**.

PyTorch share some of the syntax and routine implementation with Torch (i.e. same C and CUDA libraries backends) and, therefore, offers a t7 deserialization procedure called `torch.utils.serialization.load_lua`. However, unlike the PyTorch framework, this routine is not entirely cross-platform. In fact, deserialization fails on Windows machine. Digging further it, the Torch wiki clearly states that Windows is not supported.

In order to regain platform-independence, we decided to use a third-party tool to convert t7 files in another format easier to read. This tool, on input a t7-encoded model, produces two files: a Python file containing the sequence of layer definitions (see next section) and another file with **pth** extension containing the model weights. The pth format is the official PyTorch binary serialization format and pth weigths can be easily loaded into an existing model with `model.load_state_dict` in a cross-platform way.

4.3 Model definition

In this section we will see the actual VGG19 model definition layer by layer. We will show snippets for the encoder and decoder Relu_5_1 block. Shallower blocks will not be shown because they have the same parameters but lesser layers.

4.3.1 Encoder

The encoder model in Listing 1 is an instance of `torch.nn.Sequential` class. The model contains layers in the order in which they are passed to the constructor. Convolutional layers `torch.nn.Conv2d` have the following parameters list (in_channels, out_channels, kernel_size, stride=1, padding=0, ...).

`torch.nn.ReflectionPad2d` layer add a border of one pixel around the image using the specular pixel (i.e. the pixel at the opposite side of the image). `torch.nn.MaxPool2d` layer look at groups of 2×2 pixels and retain only the maximum value.

4.3.2 Decoder

The encoder model in Listing 2 is also an instance of `torch.nn.Sequential` class. Padding is as in the encoder. The main difference are the `torch.nn.UpsamplingNearest2d` layers that act as a reverse of the MaxPooling operation and enlarge the spatial extent of the feature maps. Convolutional layer are used only to compute feature as described in section 3.2.1.

4.4 Argument parsing

The program was developed as a command-line application. The user must give the appropriate textual arguments to launch the desired task. The list of the application functionalities is discussed in the following sections.

The command-line interface was written using the `argparse` module of the Python Standard Library. This module provides an `ArgumentParser` object with methods to add arguments that will be parsed from the `sys.argv` list. It also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

In Listing 3 we can see the definition of the argument parser object with its description message. No arguments in our implementation is declared as required. At line 6 we can see the definition of a string arguments that will be stored inside `parser.content`, which will be the content image of the stylization. At line 10 we are defining a boolean argument with a default value of `False` when it is not given. At line 24 we are defining an argument which can only be a real-valued number or else the parser will trow an error. Additional checks on the arguments (i.e. existence of files and folders, floats

```
import torch.nn as nn

vgg_conv5_1 = nn.Sequential( nn.Conv2d(3,3,(1, 1)),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(3,64,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(64,64,(3, 3)),
        nn.ReLU(),
        nn.MaxPool2d((2, 2),(2, 2),(0, 0),ceil_mode=True),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(64,128,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(128,128,(3, 3)),
        nn.ReLU(),
        nn.MaxPool2d((2, 2),(2, 2),(0, 0),ceil_mode=True),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(128,256,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(256,256,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(256,256,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(256,256,(3, 3)),
        nn.ReLU(),
        nn.MaxPool2d((2, 2),(2, 2),(0, 0),ceil_mode=True),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(256,512,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(512,512,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(512,512,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(512,512,(3, 3)),
        nn.ReLU(),
        nn.MaxPool2d((2, 2),(2, 2),(0, 0),ceil_mode=True),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(512,512,(3, 3)),
        nn.ReLU(),
)
```

Listing 1: VGG19 Encoder Relu_5_1 block definition.

```python
import torch.nn as nn

feature_invertor_conv5_1 = nn.Sequential(
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(512,512,(3, 3)),
        nn.ReLU(),
        nn.UpsamplingNearest2d(scale_factor=2),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(512,512,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(512,512,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(512,512,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(512,256,(3, 3)),
        nn.ReLU(),
    nn.UpsamplingNearest2d(scale_factor=2),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(256,256,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(256,256,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(256,256,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(256,128,(3, 3)),
        nn.ReLU(),
    nn.UpsamplingNearest2d(scale_factor=2),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(128,128,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(128,64,(3, 3)),
        nn.ReLU(),
    nn.UpsamplingNearest2d(scale_factor=2),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(64,64,(3, 3)),
        nn.ReLU(),
        nn.ReflectionPad2d((1, 1, 1, 1)),
        nn.Conv2d(64,3,(3, 3)),
)
```

Listing 2: VGG19 Decoder Relu_5_1 block definition.

in the correct interval, etc..) regarding the semantic of the application are done by the `validate_args` function which receives the `parser` object as argument.

4.5 Logging

The application execution is monitored through event logging. For this purpose it was used the Python Standard Library `logging` module. It provides a lot of flexibility and functionalities by means of:

- **loggers** objects expose the interface that application code directly uses.
- **handlers** send the log records (created by a logger) to the appropriate destinations.
- **formatters** are used to specify, for each handler, the layout of the log record.

In our implementation, the logger object was instantiated to output log records to two destinations: in the terminal window and in the `logs.txt` text file. These two handlers have been set on different severity levels: the command line handler is on a more verbose level (DEBUG) than the file handler (INFO). Following the same principle, the two handlers have a different formatting layout. Terminal messages also contain the file and the line number from which they were generated.

4.6 Input Output

In this section we will discuss how the program read and write data, in the form of **images**, from and to the operative system.

```
import argparse

parser = argparse.ArgumentParser(description='Pytorch implementation ' \
    'of arbitrary style transfer via CNN features WCT trasform',
    epilog='Supported image file formats are: jpg, jpeg, png')
parser.add_argument('--content', help='Path of the content image ' \
    '(or a directory containing images) to be trasformed')
parser.add_argument('--style', help='Path of the style image (or ' \
    ' a directory containing images) to use')
parser.add_argument('--synthesis', default=False, action='store_true',
    help='Flag to syntesize a new texture.')
parser.add_argument('--stylePair', help='Path of two style images ' \
    '(separated by ``,'') to use in combination')
parser.add_argument('--mask', help='Path of the binary mask image ' \
    'to trasfer the style pair in the corrisponding areas')
parser.add_argument('--contentSize', type=int, help='Reshape ' \
    'content image to have the new specified maximum size')
parser.add_argument('--styleSize', type=int, help='Reshape ' \
    'style image to have the new specified maximum size')
parser.add_argument('--outDir', default='outputs', help='Path ' \
    'of the directory where stylized results will be saved')
parser.add_argument('--outPrefix', help='Name prefixed in the ' \
    'saved stylized images')
parser.add_argument('--alpha', type=float, default=0.2,
    help='Hyperparameter balancing the blending between ' \
    'original content features and WCT-transformed features')
parser.add_argument('--beta', type=float, default=0.5,
    help='Hyperparameter balancing the interpolation between ' \
    'the two images in the stylePair')
parser.add_argument('--no-cuda', default=False, action='store_true',
    help='Flag to enables GPU (CUDA) accelerated computations')
parser.add_argument('--single-level', default=False, action='store_true',
    help='Flag to switch to single level stylization')

parser.parse_args()
```

Listing 3: Command line arguments parsing configuration.

4.6.1 Dataloader

In most of the use cases, the user indicate as a command line argument the path of a content image and a style image. These paths can be single images or folders containing at least one image. These images should have a **jpg**, **jpeg** or **png** extension. Once the paths are validated, images are read into the program as PyTorch Tensors by the function `load_img` as in Listing 4. The `open` method from the PIL library is used to load the image in the standard 3-channel 8-bit RGB format at its original size. When resizing is needed, in order to keep the aspect-ratio, we need to find the image's longer dimension. This dimension is set to the new size and the shorter dimension is scaled in order to keep the original sizes proportion intact. Lastly, the `torchivision.transforms.to_tensor` method converts the PIL Image with shape $(H \times W \times C)$ with pixel values in the range $[0, 255]$ to a `torch.FloatTensor` of shape $(C \times H \times W)$ in the range $[0.0, 1.0]$.

```python
import PIL
import numpy as np
from PIL import Image
import torchvision.transforms.functional as transforms

def load_img(path, new_size):
    img = Image.open(path).convert(mode='RGB')
    if new_size:
        width, height = img.size
        max_dim_ix = np.argmax(img.size)
        if max_dim_ix == 0:
            new_shape = (int(new_size * (height / width)), new_size)
            img = transforms.resize(img, new_shape, PIL.Image.BICUBIC)
        else:
            new_shape = (new_size, int(new_size * (width / height)))
            img = transforms.resize(img, new_shape, PIL.Image.BICUBIC)
    return transforms.to_tensor(img)
```

Listing 4: Image reading function.

The object responsible for loading in images is an instance of the `ContentStylePairDataset` class. This class is an implementation of the abstract class `torch.utils.data.Dataset`, which is one of the many useful object-oriented abstractions of PyTorch. The methods to implement are: `__len__` that return the total number of elements in the dataset, `__getitem__` that return a single element of the dataset by indexing between 0 and `__len__`. The constructor of the `ContentStylePairDataset` class appends every combination of content-style paths to a list. `__len__` method simply returns the lenght of this list. When requested an image, the `__getitem__` method call the `load_img` function to load and return it.

The dataset object, in the main function of the application, is wrapped inside a `Dataloader` container. This provides a single or multi-process iterator over the dataset used to stylize multiple images in a single `for` loop.

4.6.2 Image saving

The stylized output images are saved using the `torchvision.utils.save_image` function. In order to distinguish multiple outputs very easily the outputs filenames are designed as follows:

`outPrefix_contentName_stylized_by_styleName_alpha_alphaValue.contentExtension`

The command line argument `--outDir` indicates a particular folder to save the ouput images in.

4.7 Functionalities

Apart from style transfer, the application offers other functionalities enabled by giving different combinations of command line arguments (see Listing 3 for the complete list). The functionalities available are:

1. **style transfer**. On input one content image path (`--content`) and one style image path (`--style`), the application stylize the former ac-

cording to the latter. An optional parameter `--alpha` ($0 \leq \alpha \leq 1$) balance the amount of stylization and content preservation.

2. **texture synthesis**. On input one texture image path (`--style`) and the flag `--synthesis`, the application produce a novel texture similar to the texture given.

3. **style transfer interpolation**. On input a content image path (`--content`) and two style image paths (`--stylePair`) separated by a single comma, the application stylize the former according to the the characteristics of both style images. An optional parameter `--beta` ($0 \leq \beta \leq 1$) balance the transferring between the two styles.

4. **texture synthesis interpolation**. On input two texture image paths (`--stylePair`) and the flag `--synthesis`, the application produce a novel texture similar to both textures given.

5. **spatial control**. On input a content image path (`--content`), two style image paths (`--stylePair`) and a binary mask image path (`--mask`), the application stylize the foreground of the content image using the first style image and the background using the second style image.

4.8 Model behavior

The basic PyTorch abstraction for creating neural networks is the class `torch.nn.Module`. The convolutional Autoencoder, the Encoder and Decoder models we created all inherit from `torch.nn.Module`. The Encoder and Decoder model objects as class fields of the Autoencoder. The methods to implement are: `__init__` that is responsible for layers definition and hyperparameter setting and `forward` that describes programmatically the forward pass.

The `__init__` method of the Autoencoder is responsible for creating the Encoder and Decoder VGG19 blocks. The `forward` method of the Autoencoder calls the `stylize` function to get the result. This function passes the input to the Encoder and the Decoder blocks, that are, recalling Section 4.3, `torch.nn.Sequential` containers. The container, in turn, passes the input through all the layers inside it triggering the computation. To better illustrate the process, we present in Listing 5 the code of the multi-level stylization model.

The Autoencoder take as input of the constructor the command line arguments `args`. These are used to set its hyperparameters. The mask image, if present, is loaded into memory straight away. The multi-level pipeline described in Section 3.4 needs five encoder and five decoder blocks at different depths. They are instantieted and appended to the class attribute lists `encoders` and `decoders`. In the forward pass we need to perform five autoencoder sweeps at differents depths. Thus, the `forward` function has a for loop iterating over the indices of the `encoders` list. This index is given, along with others parameters, to the `stylize` function responsible for encoding the images, combining with feature WCT and returning the reconstructed output.

During implementation one of the concern regarded memory saving. Our model hold in memory ten VGG19 blocks along with all their parameters. This can lead to Out of Memory (OOM) error on memory limited machines. In order to do not make the problem worse, intermediate stylization results are not kept in memory. In fact, the variable `content_img` at line 44 of Listing 5 is overwritten at each iteration with the latest stylization output.

OOM errors can still happen if the input images have a big resolution (i.e. usually above 1920×1080). In this case it is preferable to use the `--contentSize` and/or `--styleSize` command line arguments to resize input images instead of switching to the single-level stylization model using `--single-level` command line argument.

```python
import PIL
import torch
from PIL import Image
import torch.nn as nn
from log_utils import get_logger
from feature_transforms import wct, wct_mask
from encoder_decoder_factory import Encoder, Decoder
import torchvision.transforms.functional as transforms
log = get_logger()

class MultiLevelWCT(nn.Module):
    def __init__(self, args):
        super(MultiLevelWCT, self).__init__()
        self.svd_device = torch.device('cpu')
        self.cnn_device = args.device
        self.alpha = args.alpha
        self.beta = args.beta

        if args.mask:
            self.mask_mode = True
            self.mask = Image.open(args.mask).convert('1')
        else:
            self.mask_mode = False
            self.mask = None

        self.e1 = Encoder(1)
        self.e2 = Encoder(2)
        self.e3 = Encoder(3)
        self.e4 = Encoder(4)
        self.e5 = Encoder(5)
        self.encoders = [self.e5, self.e4, self.e3, self.e2, self.e1]

        self.d1 = Decoder(1)
        self.d2 = Decoder(2)
        self.d3 = Decoder(3)
        self.d4 = Decoder(4)
        self.d5 = Decoder(5)
        self.decoders = [self.d5, self.d4, self.d3, self.d2, self.d1]

    def forward(self, content_img, style_img,
        additional_style_flag=False, style_img1=None):
        for i in range(len(self.encoders)):
            if additional_style_flag:
                content_img = stylize(i, content_img, style_img,
                    self.encoders, self.decoders, self.alpha,
                    self.svd_device, self.cnn_device,interpolation_beta=
                    self.beta, style1=style_img1,
                    mask_mode=self.mask_mode, mask=self.mask)
            else:
                content_img = stylize(i, content_img, style_img,
                self.encoders, self.decoders, self.alpha,
                self.svd_device, self.cnn_device)
        return content_img
```

Listing 5: Multi-level stylization model.

4.9 Features WCT

The source code of the Whitening and Colouring Transformations (WCT) described theoretically in Section 3.3 is given in Listing 6.
First of all, the tensor `cf` containing the features are converted to double-precision decimal format for the following operations. The 3D features volume is reshaped to a 2D matrix using the `view` function. Then, each feature channel is normalized by subtracting its empirical mean before computing the covariance matrix. Since the eigendecomposition seen in 3.3 does not exists for non symmetric positive-semidefinite matrices, the covariance matrix is decomposed using the Singular Value Decomposition (SVD).

The overall whitening Equation 3.5 is calculated on line 21 and the coloring Equation 3.6 have its Python counterpart on line 41. This method can be used to combine multiple feature volumes together. In fact, Listing 6 omits it for brevity but when a second style image (i.e. `s1f`) is given to the application, this gets also colored with the whitened features as the first style. The two colored features are blended together with a simple linear combination balanced by `beta` (see comment on line 47).

```python
import torch

def wct(alpha, cf, sf, s1f=None, beta=None):
    # whitening phase
    cf = cf.double()
    c_channels, c_width, c_height = cf.size(0), cf.size(1), cf.size(2)
    cfv = cf.view(c_channels, -1)  # new shape C × (h * w)
    c_mean = torch.mean(cfv, 1) # calculate means row-wise
    c_mean = c_mean.unsqueeze(1).expand_as(cfv)
    cfv -= c_mean
    # cov(X) = Σ = XX^T/(N-1)
    c_covm = torch.mm(cfv, cfv.t()).div((c_width * c_height) - 1)
    c_u, c_e, c_v = torch.svd(c_covm, some=False) # c_covm = c_u * c_e * c_v^T
    k_c = c_channels
    for i in range(c_channels):
        if c_e[i] < 0.00001:
            k_c = i
            break

    c_d = (c_e[0:k_c]).pow(-0.5)
    whitened = torch.mm(torch.mm(torch.mm(c_v[:, 0:k_c],
        torch.diag(c_d)), (c_v[:, 0:k_c].t())), cfv)

    # coloring phase
    sf = sf.double()
    _, s_width, s_heigth = sf.size(0), sf.size(1), sf.size(2)
    sfv = sf.view(c_channels, -1)
    s_mean = torch.mean(sfv, 1)
    s_mean = s_mean.unsqueeze(1).expand_as(sfv)
    sfv -= s_mean

    s_covm = torch.mm(sfv, sfv.t()).div((s_width * s_heigth) - 1)
    s_u, s_e, s_v = torch.svd(s_covm, some=False)
    s_k = c_channels  # same as content's channels
    for i in range(c_channels):
        if s_e[i] < 0.00001:
            s_k = i
            break

    s_d = (s_e[0:s_k]).pow(0.5)
    colored = torch.mm(torch.mm(s_v[:, 0:s_k],
        torch.diag(s_d)), s_v[:, 0:s_k].t()), whitened)

    cs0_features = colored + s_mean.resize_as_(colored)
    target_features = cs0_features.view_as(cf)
    # if s1f (additional style image): cs1_features = wct(s1f)
    # target_features = β * cs0_features + (1.0 - β) * cs1_features

    ccsf = alpha * target_features + (1.0 - alpha) * cf
    return ccsf.float().unsqueeze(0)
```

Listing 6: Feature Whitening and Coloring transforms.

Chapter 5

Evaluation

In this chapter we will illustrate and analyse the experimental results of the application. We will compare results obtained with differents architectures and user-controlled hyperparameters. In the end, we give some performance-related remarks.

The proposed approach offers a very convenient tradeoff. It is learning-free at test stage and can reproduce arbitrary style without sacrifing too much efficiency in the feedforward pass. In fact, compared to other recent neural style transfer methods, called *Fast Methods based on Offline Model Optimisation* in [10], the proposed approach has a similar per-image execution time.

5.1 Results

To demonstrate the effectiveness of the proposed algorithm we show the outputs obtained for various content, style and texture images in following sections. There is no universally accepted quantitative standard for style transfer evaluation. For this reason, we show our results and relay on a *qualitative evaluation*.

5.1.1 Style transfer

In order to display style transfer results obtained on a variety of content and style images we decided to put them in a grid (see Figure 5.2). The last six images of the first column are the content images. The last eight images of the first row are the style images. The element in position i, j of the grid represent the j-th style transferred to the i-th content image. The content images represent a wide variety of subjects (people, infrastructure, fruit, ...) and contexts (seaside, mountains, train station, ...). The style images are also drawn from a wide array of painting movements (Futurism, Symbolism, Expressionism, ...).

Without learning any style, our method is able to capture visually salient patterns in the style images (e.g., waveform in the third column). Style patterns are not transferred to only relatively smooth regions (e.g. sky background, sea background) of the content images but also to non-smooth regions that usually correspond to key components of the content (e.g. oranges, bridge). It is interesting how the use of contrast and the shadowing of the stylized output resemble the painting (e.g. orages reflection on the table in the fifth row). Small details gets sometimes blurred out (e.g. childrens face in $image_{6,1}$, people in the background at the station in $image_{7,1}$) but the overall message of the artwork (i.e. color palette, edge patterns, rich local structures) is almost always transferred to the content.

The adopted approach allows to blend multiple styles together before transferring it to the content. We decided to allow the transferring of a maximum of two styles simultaneously because more than that usually leads to artifacts and unrecognizable content. In Figure 5.2 we showcase the obtained result. The image in the far left of the grid is the content image. The images on the far right are the two styles images. In the middle there is the stylized result. The features extracted from both styles are transferred in a selective and interesting way. The texture and the shading of the water style is applied to the petals of the flowers. Little sparks of yellow details from the tiger style

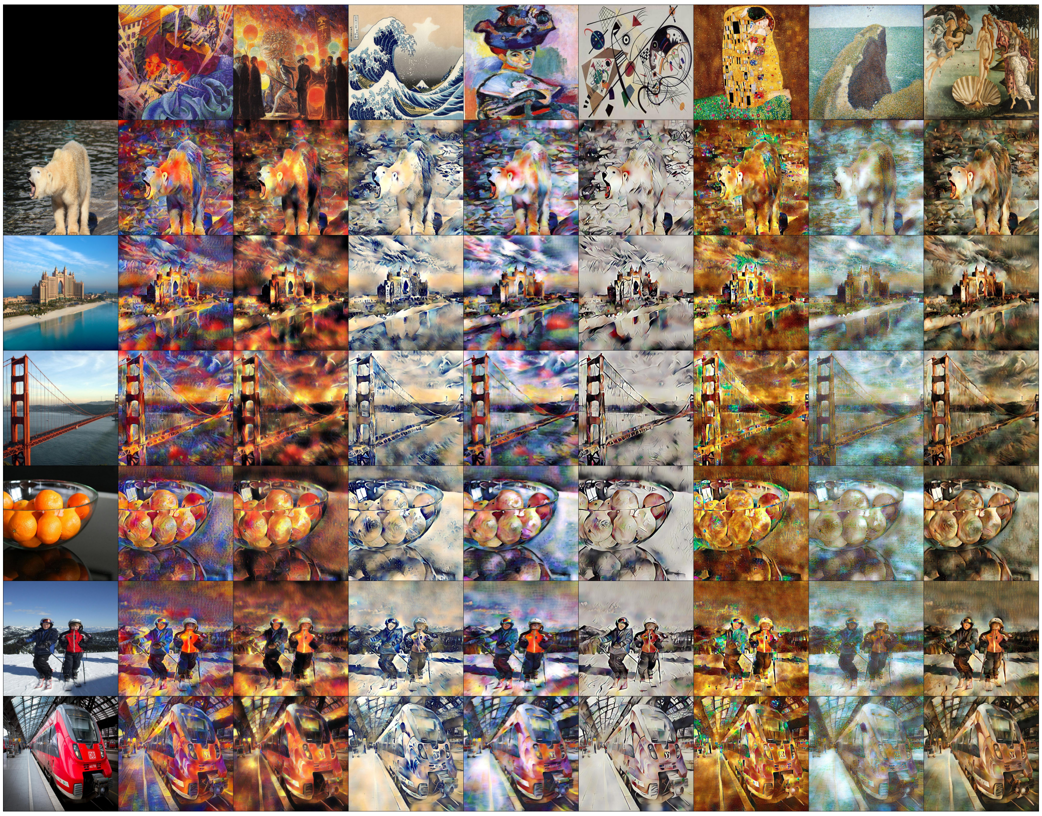


Figure 5.1: Showcase of (48) style transfer results.

are transferred to the flower's goblets.

Figure 5.2: Showcase of two styles simultaneous transfer.

5.1.2 Texture synthesis

By setting random noise as the content image, our stylization framework can be applied to texture synthesis. Empirically, we found out that multiple iteration (3 iteration) of the multi-level stylization pipeline usually yield better looking results. Figure 5.3 showcase the obtained results. In the left column of the grid we have the texture images and in the right colum there are respective synthetized results.

The results show that our method is able to effectively reproduce the reference textural effects with natural variations due to the random noise input. Output textures seems absolutely faithful to the original when there is no geometric pattern in the starting texture image. This is the case of the bubble texture in the middle.

5.2 User control

Given a content image and a style image pair, our approach is flexible enough to accommodate different requirements from the user by providing some control over the stylization, including scale, weight and spatial control.

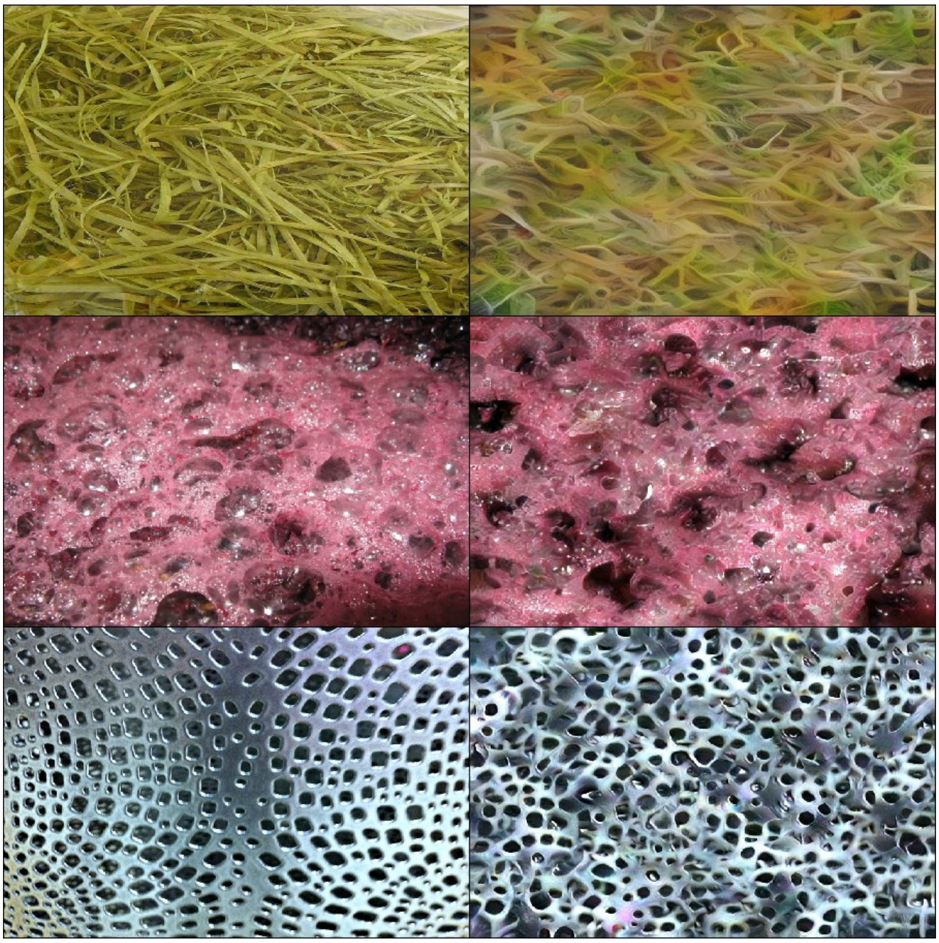


Figure 5.3: Showcase of (3) texture synthesis results.

5.2.1 Spatial control

Spatial control of the stylization allow users to edit an image with different styles on different parts of the image. The grid in Figure 5.4 shows an example of the result obtained by our approach on this task. In addition to the content image (i.e. the women face), a binary mask white-on-black (i.e. the eyes mask in the central column) is required as input, to indicate the spatial correspondence between content regions and styles. The styles used here are on the far right column of the grid. The obtained result is the first column.

Figure 5.4: Showcase of spatial control over the style trasfer.

5.2.2 Hyperparameters

As described in Section 4.4, there a few command line arguments that adjust the style transfer. Varying the style images size will lead to different extracted statistics due to the fixed receptive field of the network. Therefore, the scale control is easily achieved by adjusting the `--styleSize` command line argument.

Another model's hyperparameter to tweak is the weight control `--alpha`, which control the balance between stylization and content preservation. The

proposed method enjoys this flexibility by simply plugging this parameter in Equation 3.6.
An example of usage of these hyperparameters is shown in Figure 5.5. The content image is taken from Figure 5.4. The second an third image from the left illustrate the scale control (different-sized brick patterns), while the last three illustrate the weight control.

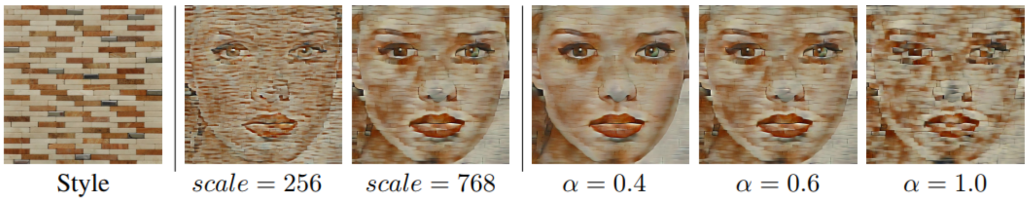


Figure 5.5: Showcase of the hyperparameter control over style transfer.

5.3 Performances

The application is cross-platform and can run on CPU or GPU thanks to PyTorch CUDA accelaration and device-agnostic API. This is done by selecting a PyTorch device type, identified by the string `cpu` or `cuda:0`. The default device is the latter. CPU-only computation is selected by giving the `--no-cuda` command line argument. The model object and input images are moved to the selected device,which is used internally by the framework to perform all the operations.

Table 5.1 show stylization wall-clock timings for various image sizes, computational devices and stylization pipelines.

Our approach is pretty time efficient on GPU. The 48 stylizations of the 768×768-sized images in Figure 5.1 took 8 minutes (whole-process time) on a labtop-grade GPU. The most time consuming task is the eigen decomposition in WCT. It is important to note that the computational cost of this step will not increase along with the image size because the dimension of the

Size \ Type	**SingleCPU**	**SingleGPU**	**MultiCPU**	**MultiGPU**
256 × 256	3.1757	0.8297	9.7269	1.6117
512 × 512	13.2555	1.2131	40.7391	3.6235
768 × 768	25.5276	1.4680	79.8720	5.7486
1080 × 1080	44.7806	2.1954	125.0736	10.4855
1920 × 1920	143.5199	7.1902	417.6532	42.3703

Table 5.1: Stylization timings (in seconds) with different configurations. **SingleCPU** stands for single-level architecture executed on the CPU, **MultiGPU** stands for multi-level architecture executed on the GPU.

covariance matrix to decompose only depends on the fixed number of filters of the final layer (i.e. Relu_5_1 has 512 output channels). This decomposition step is stricly CPU-based because under the hood `torch.svd` implementation has several calls to the LAPACK software library. To evaluate how memory copying back and forth the GPU memory impacted performance, we averaged ten `wct` calls with features of a 512×512 content image and a 591×800 style image. The CPU implementation took 4.604 seconds while the GPU implementation took 5.312 seconds. Thus, in our implementation, the default behaviour is to move features to the CPU memory before calling the `wct` function and then, move them back to GPU memory for decoder reconstruction.

Chapter 6

Conclusion

In fine art, especially painting, humans have mastered the skill to create unique visual experiences through composing a complex interplay between the content and style of an image. Artistic Style Transfer tried to tackle the problem in an algorithmic way but without great sucess. The work by Gatys *et al.* [7] was the first to use *Convolutional Neural Networks*, enabling them to get a hierarchical image representation. Their seminal work effectively created a new research direction called *Neural Style Transfer*.

In this thesis we focused on Neural Style Transfer, adopting a novel approach for **transferring arbitrary styles** in a learning free manner. A convolutional neural network is used to extract a representation of the content image and style image characteristics. A symmetrical Decoder is trained in advance for general-purpose image reconstruction. Then, the **whitening and coloring transforms** (WCT) are applied in the bottleneck layer of the *Convolutional Autoencoder* in order to match the statistical distribution of the content features with the style features. These transformed features are then reconstructed back to RGB space. The stylized result carries informations about the style image but preserves the global structure of the content image. In order to extract all the information from the style image, this process is applied multiple times by inverting the features produced by the CNN at differents depth levels.

The obtained results are visually pleasing and, most of the times, resemble the style artwork patterns. The impressive remark is the flexibility of this approach (applicable also for *texture synthesis*). Content and styles from diverse natural domains and artistic movements can be combined together in a creative way. NST, usually, finds its market in image editing and social media applications. The flexibility of our approach can also support an artist in the creative process by quickly pitching various content-stylization ideas and also allows to spatially control the effect.
On the other hand, since inverting very deep features is a difficult task, it's fair to say that if the user need to preserve detailed structures of the content image a more powerful decoder is necessary.

Bibliography

[1] Yoshua Bengio. "Learning Deep Architectures for AI". In: *Found. Trends Mach. Learn.* 2.1 (Jan. 2009), pp. 1–127. ISSN: 1935-8237. DOI: 10.1561/2200000006. URL: http://dx.doi.org/10.1561/2200000006.

[2] *Covariance*. URL: https://en.wikipedia.org/wiki/Covariance.

[3] *Cumulative Distribution Function*. URL: https://en.wikipedia.org/wiki/Cumulative_distribution_function.

[4] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *In CVPR*. 2009.

[5] Alexei A. Efros and William T. Freeman. "Image Quilting for Texture Synthesis and Transfer". In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 341–346. ISBN: 1-58113-374-X. DOI: 10.1145/383259.383296. URL: http://doi.acm.org/10.1145/383259.383296.

[6] Serena Yeung Fei-Fei Li Justin Johnson. *Lecture 12: Visualizing and Understanding*. URL: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture12.pdf.

[7] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. "A Neural Algorithm of Artistic Style". In: *CoRR* abs/1508.06576 (2015). arXiv: 1508.06576. URL: http://arxiv.org/abs/1508.06576.

[8] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN: 013168728X.

[9] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366.

[10] Yongcheng Jing et al. "Neural Style Transfer: A Review". In: *CoRR* abs/1705.04058 (2017). arXiv: `1705.04058`. URL: `http://arxiv.org/abs/1705.04058`.

[11] Justin Johnson, Alexandre Alahi, and Fei-Fei Li. "Perceptual Losses for Real-Time Style Transfer and Super-Resolution". In: *CoRR* abs/1603.08155 (2016). arXiv: `1603.08155`. URL: `http://arxiv.org/abs/1603.08155`.

[12] Agnan Kessy, Alex Lewin, and Korbinian Strimmer. "Optimal Whitening and Decorrelation". In: (Dec. 2015).

[13] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. "CIFAR-10 (Canadian Institute for Advanced Research)". In: (). URL: `http://www.cs.toronto.edu/~kriz/cifar.html`.

[14] Yann LeCun et al. "Object Recognition with Gradient-Based Learning". In: *Shape, Contour and Grouping in Computer Vision*. London, UK, UK: Springer-Verlag, 1999, pp. 319–. ISBN: 3-540-66722-9. URL: `http://dl.acm.org/citation.cfm?id=646469.691875`.

[15] Yijun Li et al. "Diversified Texture Synthesis with Feed-forward Networks". In: *CoRR* abs/1703.01664 (2017). arXiv: `1703.01664`. URL: `http://arxiv.org/abs/1703.01664`.

[16] Yijun Li et al. "Universal Style Transfer via Feature Transforms". In: *Advances in Neural Information Processing Systems*. 2017.

[17] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *CoRR* abs/1405.0312 (2014). arXiv: `1405.0312`. URL: `http://arxiv.org/abs/1405.0312`.

[18] Warren S. McCulloch and Walter Pitts. "Neurocomputing: Foundations of Research". In: ed. by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Chap. A Logical Calculus of the Ideas Immanent in Nervous Activity, pp. 15–27. ISBN: 0-262-01097-6. URL: `http://dl.acm.org/citation.cfm?id=65669.104377`.

[19] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science, 1997. ISBN: 0070428077.

[20] Augustus Odena, Vincent Dumoulin, and Chris Olah. "Deconvolution and Checkerboard Artifacts". In: *Distill* (2016). DOI: `10.23915/distill.00003`. URL: `http://distill.pub/2016/deconv-checkerboard`.

[21] Frank Rosenblatt. "The perceptron: A probabilistic model for information storage and organization in the brain [J]". In: 65 (Dec. 1958), pp. 386–408.

[22] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). arXiv: `1409.1556`. URL: `http://arxiv.org/abs/1409.1556`.