Deep Kernel Mean Embeddings for Generative Modeling and Feedforward Style Transfer

by

Tian Qi Chen

B.Sc., The University of British Columbia, 2015

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2017

Abstract

The generation of data has traditionally been specified using hand-crafted algorithms. However, oftentimes the exact generative process is unknown while only a limited number of samples are observed. One such case is generating images that look visually similar to an exemplar image or as if coming from a distribution of images. We look into learning the generating process by constructing a similarity function that measures how close the generated image is to the target image. We discuss a framework in which the similarity function is specified by a pre-trained neural network without fine-tuning, as is the case for neural texture synthesis, and a framework where the similarity function is learned along with the generative process in an adversarial setting, as is the case for generative adversarial networks. The main point of discussion is the combined use of neural networks and maximum mean discrepancy as a versatile similarity function.

Additionally, we describe an improvement to state-of-the-art style transfer that allows faster computations while maintaining generality of the generating process. The proposed objective has desirable properties such as a simpler optimization landscape, intuitive parameter tuning, and consistent frame-by-frame performance on video. We use 80,000 natural images and 80,000 paintings to train a procedure for artistic style transfer that is efficient but also allows arbitrary content and style images.

Lay Summary

While physical actions generate data in the real world, this thesis discusses the problem of simulating this generation procedure with a computer. The quality of the simulation can be determined by designing a suitable similarity measure between a data set of real examples and the simulated data. We contribute to this line of work by proposing and analyzing the performance of a certain similarity measure that can be arbitrarily complex while still being easy to compute. Furthermore, we contribute to the problem of generating stylistic images by proposing an efficient approximation to the existing state-of-the-art method.

Preface

Chapter 3 of this thesis is original, unpublished, independent work by the author, Tian Qi Chen.

The work described in Chapter 4 of this thesis was performed in collaboration with the author's supervisor Mark Schmidt. All code was written by the author, Tian Qi, while Mark helped analyze the results of the experiments. This work has not been published, but a preprint for the proposed method described in Chapter 4 is available online as "Fast Patch-based Style Transfer of Arbitrary Style". This preprint was written by both Tian Qi and Mark. A shorter version of this preprint was accepted as an oral presentation at the Constructive Machine Learning (CML2016) workshop.

Table of Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I am deeply thankful to my supervisor Mark Schmidt for not just the technical insights and guidance, but also for his unending support and encouragement throughout my graduate studies. Mark has given me ample opportunities to explore research areas I find interesting all the while providing much needed redirections when I hit unforeseen roadblocks.

I would like to express my utmost gratitude to other faculty members who have given me valuable advice before and during the course of my program. Thank you Kevin Leyton-Brown for driving me on the path to research and critical thinking. Thank you Alexandre Bouchard-Côté, Nicholas Harvey, and Michael Gelbart for being awesome professional role models.

I must also thank Issam Laradji, Alireza Shafaei, Mohamed Ahmed, Reza Babanezhad, and everyone in the machine learning lab at UBC for their lighthearted conversations during stressful workdays.

I thank my friends Brendan Shillingford and Yi Pin Cao for their friendship and general existence.

Last but not least, I am also grateful to all the participants of the machine learning and machine learning theory reading groups for spending our time together to learn new topics and explore interesting directions.

Dedication

To my parents for their unconditional love and sacrifices.

Chapter 1

Introduction

Augmenting real data with synthesis data can increase classification results especially when data is scarce [78]. Conditionally generated data can be used in applications such as fraud detection when there are few amounts of customer data, or spam filtering due to the large imbalance of positive and negative samples. It is also possible to simply generate images that are aesthetically pleasing and entertaining. The generation process can be used as an advanced image filter, such as a one that creates haunted buildings and monster faces from normal photographs [92]. This kind of learned or mimicked creativity allows fast production of artistic textures in creative applications.

This thesis discusses methods for the synthesis or generation of data. The act of creating new data requires sufficient knowledge of the underlying properties of the real data. Though perfectly mimicking the data generating process would require complete understanding of the real world, it is possible to approximate the generating process with a simple model and few assumptions. For image data, it is possible to mimic the contours of a face, or the artistic style of a painting. By using domain knowledge about the structure of image patterns, recent methods have been able to synthesize compelling new paintings of existing artistic styles. Methods that can create a seemingly infinite number of new images from just a single exemplar painting can be re-interpreted as mimicking particular patterns of the exemplar painting. By only slightly perturbing the location of these patterns, new structure can be injected into the painting. Existing methods for this task are slow or limited, and in this thesis we propose a method to speedup the generating process without sacrificing generality.

1.1 Generative modeling

A generative process describes how data are sampled and which factors lead to specific changes in the samples. If the real underlying generative process is known, then statistical properties and even causal relations can be extracted. In an ideal setting, discriminative methods will no longer be

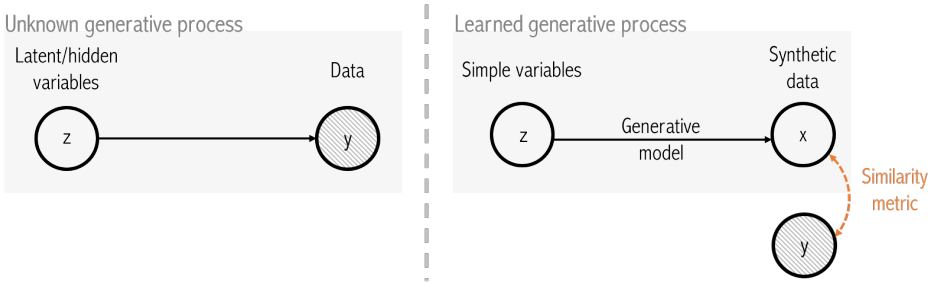


Figure 1.1: (Left) A latent variables model, where the observed data is assumed to be a function of some hidden unobserved variables. (Right) A generative model that approximates the generative process by first sampling z then passing it through a learned function; the synthetic data is then compared with real data to train the generative model.

hindered by limited number of samples, as samples can be generated freely. Many detectors can become more robust to small changes by augmenting the classifier with synthetic data. For instance, face detectors should not be fixated on the specific shape or texture details of facial features. Such overfitting can be avoided by having more data, but while real data is scarce, synthetic data can be produced easily.

Typically the generative process is not known and only a finite number of samples are present. When the generative process is not known or intractible to infer, as is often the case for real world data, it is possible to approximate the underlying process with a generative model. This generative model can be trained using the finite samples that are available. Obviously the model could be imperfect and can only replicate simple aspects of the generative process. However, an approximate model can still be used to increase the effectiveness of discriminative classifiers [78] and perform complex image processing such as inpainting [93] or de-occlusion [96]. The generative model can be further augmented to produce an approximate density estimation [13] or extract cross-domain properties between pairs of datasets [42, 97]. This is by no means an extensive list, and newer applications of generative models are still being discovered.

Constructing a generative model and replicating the real generative process can range from easy to incredibly difficult. Generating the outcome of a coin toss or dice is much simpler than generating an image of a person's face or a piece of artwork. However, it may be possible that a complicated image is composed of simpler components. For example, the structure of a person's face is fixed, with a set of facial features ranging in size and

shape. It may be possible to recreate, or at least approximate, the generative process of a face image by flipping numerous coins. This is the idea that high-dimensional data like images lie in a simpler lower-dimensional manifold. In other words, the data is caused or described by a number of simpler variables. An approximate model can then be constructed by first sampling these simpler variables, then mapping or relating these samples to a point in the high-dimensional space. This kind of generative model is often referred to as a latent variable model, where an assumption is made that the data we want to generate can be represented using a set of latent (hidden; unobservable) random variables.

1.2 Deep neural networks

A model that maps a set of simple random variables to samples from the data distribution can be a very complex function. The state-of-the-art model for learning arbitrary relations is a deep neural network. The family of models described by deep neural networks is extremely large, including simpler models such linear or logistic regression. Recent works have shown neural networks significantly outperforming other models in computer vision, natural language processing, and generative modeling. Researchers have only begun to understand the intricacies of properties that a neural network learns. It is known that neural nets can be used to infer the latent variables of a dataset [6] or be used to extract important properties of how humans classify images [61]. For a history on the development of neural networks, refer to a recent survey by Schmidhuber [80].

In simple terms, a neural network simply refers to a differentiable function. Certain differentiable functions work better than others, or make different assumptions on the input. A neural network $f(\theta)$ often has a fixed structure or architecture f but tunable parameters θ. The output of $f(\theta)$ changes accordingly depending on the values of θ. Training a neural net refers to finding suitable parameters such that $f(\theta)$ has desirable properties. Due to the large number of parameters available to a neural net, this versatility has allowed networks to mimic highly complex functions. For the task of approximating a generative process, a neural network's output should be samples from the data distribution.

In practice, certain neural network modules (functions) work better than others. Certain modules make explicit assumptions on the input structure. For example, a convolutional neural net extracts only translation-invariant information from an image, as it makes the assumption that objects in an

image can appear in any spatial location and have the same meaning. A convolutional layer can be seen as a trainable filter that slides across an image. Prior to convolutional neural networks, computer vision researchers used hand-designed algorithms that extract certain features (e.g. faces) from an image. However, human engineered algorithms often do not perform as well as a trained convolutional network when the true function is very complex. In early research on texture synthesis, researchers defined texture using a set of constraints such as periodicity and spatial homogeneity [17, 18, 72]. More recent methods based on convolutional networks [22] have shown to work with a larger set of images, especially colour images, though implicitly still makes a spatial homogeneity assumption.

1.3 Image synthesis tasks

In this thesis, we discuss two related tasks, both regarding the generation of images. The first task is to simply learn an approximate model of the generative process of images, given a large set of examples. The examples can be a dataset of natural images, faces, etc.

The second task is to learn a generative model of a certain type of artistic style given only a *single* exemplar image. The first task is rather generic and the methods can be applied to other forms of data, but the second task requires more assumptions about the data, such that the painting style is consistent throughout the image. A simpler extension of this task is style transfer, where the artistic style of an image is transferred to the structural content of another image.

Learning such generative processes is difficult. The most popular method for training a neural net is gradient descent on a loss function $L(f(\theta))$. However, what is the correct loss function for these tasks? We look into a very versatile and generic loss function called *maximum mean discrepancy*. In simple terms, maximum mean discrepancy is a measure of difference between two probability distributions, ie. the true data distribution and the generative model distribution. This metric can be used for both tasks, makes minimal assumptions about the data, and is easy to compute.

We combine maximum mean discrepancy with a neural network for both tasks, though the first task is harder and requires an extra regularization trick. We also discuss how a part of a recent popular style transfer algorithm can be seen as minimizing the maximum mean discrepancy. We propose a branch of fast style transfer algorithms by approximating the process with a trained neural network.

The contributions of this thesis include:

- A generalization of generative moment matching networks, a certain type of generative model, to make use of the representational power of neural networks. We develop a regularization term to stabilize training of the model, which shows improvement in generating colored images compared to baseline generative moment matching networks.
- A new take on style transfer where a forced distributional alignment is performed inside the feature space of a neural network, then inverted back into image space. This process provides generalization to artistic images not found in prior feedforward style transfer methods while being faster than methods that are based on optimization.

Chapter 2

Background

2.1 Deep Learning

The works described in this thesis use deep neural networks as modeling tools. These are flexible models that can express any multivariate continuous function as long as the network has enough free parameters and enough data to learn from [80]. Due to their unparalleled expressiveness, deep neural networks are used extensively in computer vision, natural language processing, and probabilistic modeling.

2.1.1 Modularity of neural networks

Neural networks are compositions of linear and non-linear functions, where the non-linearities are typically applied element-wise and are differentiable. The linear functions of the neural network introduce tunable weights, or parameters, that change the output of the function. The composition of functions lends to a high degree of modularity, as larger networks can be created by composing smaller networks. In this context, the small networks are often referred to as *layers* or *modules*. Intermediate outputs of the each layer are called *features* or *activations*. In more complex applications, networks modules can have separate use cases, be trained either independently or altogether in an end-to-end fashion. For example, a variational autoencoder [44] is a neural network where one module encodes data samples to samples from a simple (e.g. Gaussian) prior distribution and another module decodes samples from the prior distribution to data samples. Trained in an end-to-end fashion, this network learns a latent variables model.

An important characteristic of neural nets is the differentiability of the entire network by application of chain rule. In deep learning, the execution of the function is called forward propagation and computing the gradient (first derivatives) with respect to its input and weights is called *backpropagation*, as gradients are propagated from the end of the network to the input layer.

2.1.2 Learning by gradient descent

As the entire network is differentiable, the most popular method of finding the best values for the weights of the network is through gradient descent, a first-order optimization technique. Gradient descent in its simplest form finds local minima of continuous functions

$$\underset{w}{\operatorname{argmin}} L(X, w) \tag{2.1}$$

where X represents the data, w are the weights of the network, and L is a loss function. The loss function can contain a neural network itself, whereby after the optimal w is found, only the neural network is used for inference.

Gradient descent is a simple iterative method where at each iteration the weights w are perturbed slightly to improve the loss function. The update equation is

$$w_{next} = w - \alpha \nabla_w L(X, w) \tag{2.2}$$

where $\nabla_w L(X, w)$ is the gradient, denoting the multivariate first-order derivatives of $L(X, w)$ with respect to w, and α is the step size.

However, when the amount of data is large, even computing $L(X, w)$ can be intractable. As gradient descent is an iterative procedure, a few thousand iterations are needed depending on the dimension of the weight vector w, ie. the degree of freedom in the neural network.

To alleviate the problem of having a runtime being dependent on the amount of data, *stochastic* gradient descent is used in practice. Assuming the loss function can be decomposed as

$$\underset{w}{\operatorname{argmin}} \sum_{i=1}^{N} L_i(X_i, w) \tag{2.3}$$

then stochastic gradient descent looks only at a single sample x_i per iteration.

$$w_{next} = w - \alpha \nabla_w L_i(x_i, w) \tag{2.4}$$

While the number of iterations required to find the best solution to (2.3) may be higher than solving (2.1), stochastic optimization will typically find decent solutions quickly, as the weights can be updated before even observing the entire dataset. In practice, a small number of samples, called a minibatch, is used per iteration.

One downfall of stochastic gradient descent is its performance near-optimum is quite noisy. As the optimization only sees a handful of samples

each iteration, convergence is reliant of reducing the step size to zero; otherwise, the optimization never converges. It is possible to have the best of both the speed of stochastic gradient descent and the convergence properties of gradient descent by a branch of methods called stochastic average gradient [81]. Other improvements on stochastic gradient descent exist, such as momentum and per-element estimation of curvature. For survey on numerical optimization algorithms in the context of machine learning applications, see [4].

2.1.3 Choice of network modules

Though many functions are differentiable, not all are useful in practice. Certain non-linear functions are more amenable to gradient descent optimization, and linear functions can contain structural information that allow training to progress faster. The problem of vanishing and exploding gradients exists during backpropagation when certain modules significantly decrease or increase the magnitude of the gradient from one module to another. As a neural network becomes deeper (composed of more and more modules) the gradient can entirely vanish to zero or explode to large enough values to cause numerical instability. Architectural changes to neural networks have been proposed to counter this problem, such as the use of batch normalization [38], residual layers [30], and many others. These methods allow deeper neural networks to be trained using just gradient descent.

Imposing structure in neural networks is another interesting direction. The most popular structured linear layer is the *convolutional layer*, which is widely used in computer vision and more recently in natural language processing. Generic convolutional layers impose the constraints of *locality* and *translation-invariance*. Each output neuron of a convolutional layer only depends on a local region of the input, and the same function is applied to each region. This formulation is particularly intuitive for images as the convolutional layer learns filters that slide across the image and computes a cross-correlation score at each region of the input, indicating whether a particular pattern exists in that region of the input. Convolutional neural networks are the standard in image processing [45, 82], where the winning entry in ImageNet competitions [77] use cleverly designed convolutional layers stacked with non-linear activation functions.

Transposed convolutional layers, a variant where the forward propagation and backpropagation algorithms are reversed, are used to upsample image activations. Transposed convolutions are used in applications for semantic segmentation [59], super-resolution [49], image synthesis [74], and other

problems where activations of a small spatial activation must be upsampled to a larger spatial resolution. See [15] for illustrations.

2.2 Loss functions

Similar to the choice of network architecture, choosing an appropriate loss function is essential to solving any problem with gradient descent. In a supervised learning problem setting, each input x_i is paired with an output value y_i. One can train a neural network $f(x, w)$ to learn the relation between each x_i and y_i. A common loss function is the squared L2 norm between the neural network output and desired target y_i

$$L(x_i, y_i, w) = ||f(x_i, w) - y_i||^2. \tag{2.5}$$

Variants include using a different norm or thresholding the loss if y_i is categorical.

The supervised learning setting typically use loss functions defined on real-valued input/output vectors. However, in certain settings of generative modeling, the set of inputs and outputs are not necessarily paired. When the objective is to train the neural net such that $f(x, w)$ follows a specific distribution (where either one or both of x and w may be random), simple loss functions used in supervised learning do not suffice. Instead, one must define loss functions that differentiate between probability distributions rather than real-valued samples.

It should be noted that using the mean squared error (2.5) to train a supervised learning problem can be perceived as assuming elements of the network output $f(x, w)$ follow independent Normal distributions. Other loss functions have similar implications, such as the L1 loss function and Laplace distribution. However, the difference between a loss function defined on real-valued vectors and a loss function defined on probability distributions is that the particular sample of x is assumed to be random as well and there is no corresponding value of y. Instead, $f(x, w)$ is seen as a random vector defined by the particular distribution of x. The neural network should map the distribution of x to the distribution of y, rather than learning the distribution of y conditioned on single samples of x. The two types of modeling are referred to as *generative* and *discriminative*, which model the distributions $p(x, y)$ and $p(y|x)$ respectively.

The latent variables model tries to learn how samples z from a simple distribution distribution can be used to generate samples x from a complex data distribution. This problem definition, contrary to a supervised learning

setting, does not explicitly assume any particular pairing between individual samples z and x but that any reasonable pairing will suffice. Thus in this unpaired setting, a type of unsupervised learning, only loss functions defined on the distributions of z and x can be used.

2.2.1 Divergences

Measuring the difference between probability distributions is not easy. Some measures make the assumption that the distributions are known, some are not proper metrics, and some are simply intractable to compute.

Let p_x and p_y be two continuous distributions while also representing their density functions. One family of consists of f-divergences of the form

$$D_f(p_x||p_y) = \mathbb{E}_{s\sim p_y}\left[f\left(\frac{p_x(s)}{p_y(s)}\right)\right] \tag{2.6}$$

where f must be a convex function such that $f(1) = 0$. Different divergences can be constructed by the choice of the function f [71]. Intuitively, this measure of difference computes the average of the odds ratio p_x/p_y weighted by the function f. When $f(t) = \log t$, (2.6) is the popular Kullback-Leibler divergence, seen in variational inference.

The biggest disadvantage of using f-divergences is p_x/p_y must be computable, and the expectation over p_y must be known. For the latter reason, the reverse divergence $D_f(p_y||p_x)$ is some times used if the expectation under p_x is easier to compute. However, f-divergences are typically asymmetric and the choice of $D_f(p_x||p_y)$ or $D_f(p_y||p_x)$ can have implications that are not fully understood.

2.2.2 Integral probability metrics

We describe a more general class of difference measures on probability distributions called integral probability metrics (IPMs). Given two distributions p_x, p_y on the same support, an IPM can be used to provide a proper metric defined using the supremum over a function class $\mathcal{F}$,

$$\text{IPM}(\mathcal{F}, p_x, p_y) = \sup_{f\in\mathcal{F}} \left|\mathbb{E}_{x\sim p_x}[f(x)] - \mathbb{E}_{y\sim p_y}[f(y)]\right|. \tag{2.7}$$

Intuitively the function f extracts meaningful statistics that can be used to discriminate between the two distributions. Conversely, if p_x and p_y are the same distribution, then the mean of any function under p_x and p_y are

the same. The function class $\mathcal{F}$ must be chosen to be rich enough but also tractable to compute.

Depending on the choice of $\mathcal{F}$, different named metrics can be constructed.

- $\mathcal{F} = \{f : ||f||_\infty \leq 1\}$ results in the *total variation distance*. These functions are bounded between -1 and 1.
- $\mathcal{F} = \{f : ||f||_L \leq 1\}$ results in the *Wasserstein distance*. These functions are smooth 1-Lipschitz.
- $\mathcal{F} = \{f : ||f||_H \leq 1\}$ results in the *maximum mean discrepancy* (MMD). These functions are within the unit ball of a reproducing kernel Hilbert space (RKHS) defined by a well-behaved kernel function.

Note that the constants 1 in the above definitions are merely convention and can be any number. Using a different constant simply scales the metric by the same amount. Note also that this is an non-exhaustive list.

A nice property of integral probability metrics is that they are proper metrics, as opposed to f-divergences which do not satisfy all properties of a metric. Importantly, the definition of IPMs does not require the computation of p_x or p_y, only an expectation which can be approximated using only samples. However, this also comes at the cost of increased computational complexity, as finding the supremum over a class of functions is often intractable. So far the only IPM that can be tractably computed is maximum mean discrepancy.

2.2.3 Maximum mean discrepancy

When the set $\mathcal{F}$ is the unit ball in a reproducing kernel Hilbert space (RKHS) $\mathcal{H}$ with kernel $k(\mathcal{X}, \mathcal{X}) \to \mathbb{R}$, the IPM metric (2.7) is known as maximum mean discrepancy (MMD). For every positive definite kernel, the RKHS is uniquely defined and there exists a feature mapping $\phi : \mathcal{X} \to \mathcal{H}$ such that $k(x, y) = \langle \phi(x), \phi(y) \rangle_{\mathcal{H}}$. This means that while $\mathcal{H}$ may be an infinite dimensional function space, the inner product can still be tractable computed using only the kernel function; this is known as the kernel trick.

The idea of feature maps is extended to probability distributions by defining the kernel mean embedding of p as

$$\phi(p) := \mu_p = \int k(x, \cdot) p(x) dx = \mathbb{E}_{x \sim p} k(x, \cdot) \tag{2.8}$$

An amazing property is the function attaining the supremum is known up to a constant [26] as

$$f^*(\cdot) \propto \mathbb{E}_{x \sim p_x} k(x, \cdot) - \mathbb{E}_{y \sim p_y} k(y, \cdot) \tag{2.9}$$

This is referred to as the witness function because it witnesses the difference in distributions. Intuitively, the witness function puts high values on samples from p_x and low values on samples from p_y. For samples from regions where p_x and p_y have similar densities, the witness function is near zero. By plugging (2.9) into (2.7), the squared MMD is computable in closed form [26],

$$\begin{aligned}\mathrm{MMD}^2(k, p_x, p_y) = E_{x,x' \sim p_x, p_x}[k(x, x')] + \mathbb{E}_{y,y' \sim p_y, p_y}[k(y, y')] \\ - 2\mathbb{E}_{x,y \sim p_x, p_y}[k(x, y)]\end{aligned} \tag{2.10}$$

Maximum mean discrepancy as a measure of difference on probability distributions can be extremely powerful as it is both tractable to compute and satisfies properties of a metric. Specifically, for certain choices of kernels known as *characteristic* kernels, MMD is zero if and only if $p_x = p_y$. However, this often does not hold in practice as only a finite number of sample is observed. Moreover, empirical uses of MMD typically involve manually finding the "right" kernel function that works well for the specific data distribution. In high dimensional spaces, choosing the kernel function often requires heuristics and intuition. (For instance, the popular Gaussian kernel relies on Euclidean distance to be meaningful, but the distance of images represented in RGB do not accurately reflect semantic differences.) As a result, MMD has not shown much progress in modeling complex high dimensional distributions. See the recent survey by Muandet *et al.* [67] for more information.

Instead of manually choosing a kernel function from a handful of functions, it is possible to learn a kernel function that optimizes the discrepancy measure for a specific task. The kernel function can be parameterized by a neural network, then trained to optimize a specific objective. The neural net can also be viewed as a feature extractor that maps the input space to a low-dimensional manifold where a simpler kernel function can be used more effectively. This is discussed further in Chapter 3.

2.3 Generative adversarial networks

Generative adversarial networks in its simplest form consists of a generator network G and an auxiliary discriminator network D. The generator network takes random samples from a prior distribution p_z and outputs samples in a single forward pass. The goal is to train the generator network to be a powerful generative model that can directly learn the generating process of the target data distribution p_x.

Original GAN. The pioneering approach proposed by Goodfellow *et al.* [25] trains the discriminator network to compare the generated samples with real samples from the data distribution in a binary classification task, while the generator network is trained to fool the discriminator in the following objective function:

$$\min_G \max_D \mathbb{E}_{x\sim p_x}[\log D(x)] - \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{2.11}$$

Ironically, in practice if the discriminator is able to perfectly distinguish generated and real samples, then no information is passed to the generator network and training stagnates. This is due in part to D having to squash the output to be in (0,1) with a sigmoid layer, which results in vanishing gradients if the output is close to 0 or 1. When the generator is far from the optimum, the discriminator has a much easier time discriminating between generated and real samples. This leads to the output of D being very close to 0 or 1. This introduces instability during training and is only remedied by careful tuning of network architectures and balancing between training D and G.

Generative Moment Matching Networks. As an alternative to the GAN two-player objective function, Dziugaite *et al.* [16] and Li *et al.* [55] propose to remove the adversarial aspect and use the MMD objective.

$$\begin{aligned}\min_G \mathbb{E}_{x,x'\sim p_x,p_x}[k(x,x')] + \mathbb{E}_{z,z'\sim p_z,p_z}[k(G(z),G(z))] \\ - 2\mathbb{E}_{x,z\sim p_x,p_z}[k(x,G(z))]\end{aligned} \tag{2.12}$$

The biggest advantage of MMD is that the supremum is implicit, so an adversary is not necessary. The proposed algorithm minimizes an empirical estimate of (2.10) with an additive sum of fixed Gaussian kernels. An alternative version proposed by [86] instead trains the discriminator to maximize the power of the statistical test associated with MMD. While they can also fine-tune the specific bandwidths of the Gaussian kernel to critic a trained generative network, existing works are unable to simultaneously train a generative model and optimize the kernel.

Approximate IPM GANs. Recently Arjovsky*et al.* [2] propose to use the Wasserstein distance to train the generative network. This is equivalent to minimizing an IPM with $\mathcal{F}$ the set of all 1-Lipschitz functions. An auxiliary discriminiator network (also referred to as critic) is then used to approximate the supremum over $\mathcal{F}$. Note that if $\forall f \in \mathcal{F} \implies -f \in \mathcal{F}$, then the absolute sign in (2.7) can be removed. This results in the following objective function

$$\min_G \max_{D:||D||_L\leq 1} \mathbb{E}_{x\sim p_x}[D(x)] - \mathbb{E}_{z\sim p_z}[D(G(z))] \tag{2.13}$$

Another recent formulation called mcGAN proposes to use functions f that are finite but multidimensional and match both the mean and covariance of the output of f. This formulation is a special case of our proposed neural MMD with a polynomial kernel of degree 2 and bias 1.

In practice, the resulting GAN algorithms no longer use log and sigmoid functions, so problematic gradients due to asymptotes are gone. Additionally, both WGAN and mcGAN seem to have reduced or even solved the problem of mode collapse often seen in the original GAN formulation. This may be attributed to IPMs being proper metrics rather than divergences.

However, the current IPM-based GAN algorithms do have some downsides. The proposed WGAN algorithm clips the weights of the discriminator to be within a pre-specified interval to enforce a Lipschitz constraint while the mcGAN algorithm does the same to enforce a bounded constraint. Weight clipping is a simple operation but using it can lead to failure in training as existing stochastic optimization methods can conflict with such a constraint [28].

Moreover, mcGAN performs QR decomposition at every update iteration to ensure orthogonality of weights. An improved version of WGAN [28] instead adds a regularization term that encourages the discriminator to be 1-Lipschitz along the path between the generated samples and real samples. Both are additional computation that are more expensive than the original GAN algorithm. Additionally, due to the need to approximate the unknown supremum, the discriminator is trained for more iterations than the generator network. This encourages stability during training but convergence is slower than the original GAN [28].

2.4 Texture synthesis and style transfer

The generation of images started with synthesizing textures. These are images that exhibit a homogeneous property, such that small patches of the image look similar and often display the same pattern. Methods designed for this task [18, 47, 56, 89] often specify an algorithmic approach that generates the image one pixel or small patch at a time. To generate a new pixel, the neighbouring region is typically matched with patches from the exemplar texture image, and the new pixel is picked based on the best matching texture patch. This often works well for simple patterns but does not work as well for complicated artistic images. Even the choice of similarity function between patches is important. For example, simple pointwise differences in color images typically do not have any meaningful interpretaton. That is, it

is difficult to create a similarity function that conveys visual perception.

More recent methods for texture synthesis operate by specifying a complex similarity function between the generated texture and the real texture. This is often done by taking texture patches as data points, and defining a similarity function based on patches. Methods have been proposed for using principal components [50], wavelet transformation [72], and neural networks [22] that aid in defining such similarity functions by essentially shifting the representation of images to a different domain. The new domain allows conventional distance metrics to become more visually meaningful.

While it is harder to realize the generative process using the second approach, better results are often achieved by these as they only implicitly define the generative process. The key is in choosing a good similarity function, which arguably is more adaptable to different textures and styles, rather than designing the generative process directly which explicitly assumes certain properties of the texture image such as periodicity or size of its patterns.

2.4.1 Neural style transfer

We describe the work of [22] and [24] in more detail, as their method has achieved surprising results and renewed interest in the area. The significant increase in visual quality come from the use of convolutional neural networks (CNN) for feature extraction [19, 21, 23, 51]. The success of these methods has even created a market for mobile applications that can stylize user-provided images on demand [37, 73, 85].

For texture synthesis, we are given a style image S and we want to synthesize an output image Y. These images are passed through a CNN, resulting in intermediate activations for some layer l which we denote by $S_l \in \mathbb{R}^{M_l \times D_l}$ and $Y_l \in \mathbb{R}^{N_l \times D_l}$. Here D_l is the number of feature maps for layer l and N_l, M_l are the sizes (height times width) of each feature map for the output and style images respectively. The style reconstruction loss for layer l as defined by [24] is as follows:

$$\mathcal{L}_{style}^{(l)} = \frac{1}{4N_l M_l D_l^2} ||G(S_l) - G(Y_l)||_F^2 \tag{2.14}$$

where $G(F) := F^T F \in \mathbb{R}^{D_l \times D_l}$ is the Gram matrix. The complete style reconstruction loss is a sum over multiple layers with differing sizes of receptive fields,

$$\mathcal{L}_{style} = \sum_{l \in L_s} \alpha_l \mathcal{L}_{style}^{(l)} \tag{2.15}$$

where L_s is a set of layers and α_l is an additional tuning parameter, though typically a_l is simply set to 1. To adapt this loss for style transfer, an additional content image C is provided. With $C_l \in \mathbb{R}^{N_l \times D_l}$ denoting the content activations, the output image is asked to minimize the following content reconstruction loss best m

$$\mathcal{L}_{content} = \frac{1}{2}||C_l - Y_l||_F^2 \tag{2.16}$$

for some layer l. Note that while the style reconstruction requires the use of multiple layers, in practice only a single layer is used for the content reconstruction. The complete loss for neural style transfer is then formulated as a weighted sum of the style and content reconstruction losses.

$$\mathcal{L}_{total} = \lambda \mathcal{L}_{content} + (1 - \lambda)\mathcal{L}_{style} \tag{2.17}$$

It is clear that the texture synthesis formulation is matching some statistics of the style image, while using a CNN to map images to a more meaningful manifold. However, this had not been rigorously investigated until recently by [54], who showed that the style reconstruction loss is equivalent to a biased estimate of maximum mean discrepancy, a family of (pseudo-)metrics defined on probability distributions.

Patch-based MMD Minimization

The loss function (2.14) may appear unintuitive, but the underlying similarity function is maximum mean discrepancy, as was shown by Li *et al.* [54]. The CNN is applied to patches of the input image, and for each patch a sample vector representation is extracted. The loss function (2.14) assumes these vector samples are from a certain distribution and compares them between the generated image and the exemplar image.

With $k(x, y) = (x^T y)^2$, the squared MMD defined on patch samples

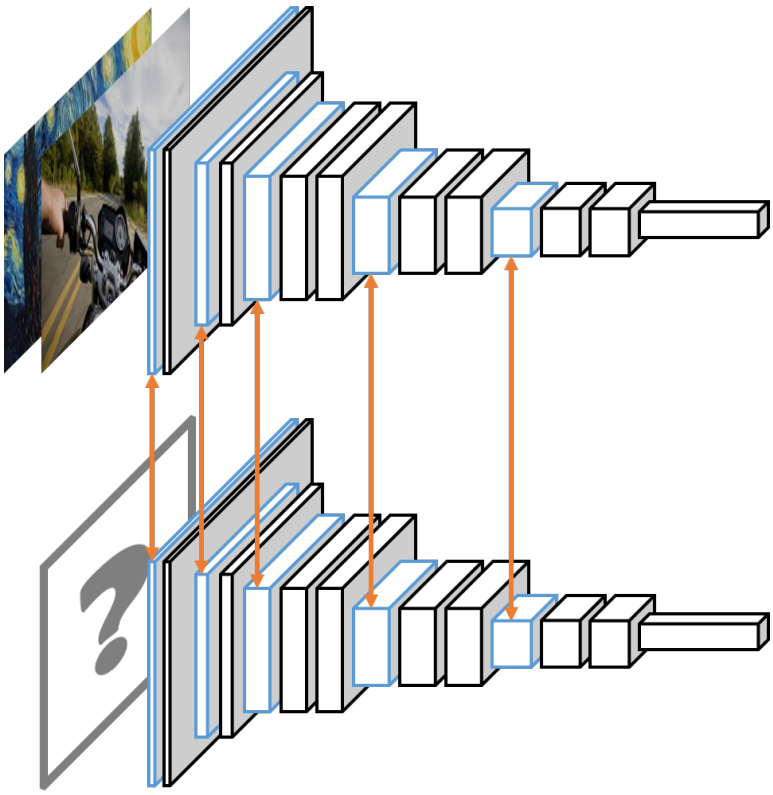


Figure 2.1: Illustration of neural texture synthesis [22] and style transfer [24]. Multiple losses are defined at different layers of a convolutional neural network. The synthetic image must match minimize the L2 norm between its features and those of the content image, while also minimizing the MMD (2.18) between its features and those of the style image.

$S = \{s_i\}$ and $Y = \{y_j\}$ is proportional to the style reconstruction loss.

$$\begin{aligned}
\text{MMD}_b^2[X, Y] &= \frac{1}{N^2}\left[\sum_{i,i'=1}^{N} k(s_i, s_{i'}) + \sum_{j,j'=1}^{N} k(y_j, y_{j'}) - 2\sum_{i,j=1}^{N} k(s_i, y_j)\right] \\
&= \frac{1}{N^2}\left[\sum_{i,i'=1}^{N} (s_i^T s_{i'})^2 + \sum_{j,j'=1}^{N} (y_j^T y_{j'})^2 - 2\sum_{i,j=1}^{N} (s_i^T y_j)^2\right] \\
&= \frac{1}{N^2}\left[||SS^T||_F^2 + ||YY^T||_F^2 - 2||SY^T||_F^2\right] \\
&= \frac{1}{N^2}||S^T S - Y^T Y||_F^2
\end{aligned} \tag{2.18}$$

The last line is proportional to (2.14), indicating that texture synthesis is taking activations from an image and viewing these activations as a dataset. The neural texture synthesis method then creates a new synthetic set of activations by minimizing the MMD between the original and synthesized activations. Note that the activations are not independent, as the convolutional layers of the CNN creates local dependencies in the outputs. In fact, the method works precisely because the activations are dependent on each other due to overlapping regions between the receptive fields. This allows the method to create varying textures while remaining visually similar to the original texture, assuming homogeneity.

It is possible to interpret the neural style transfer algorithm as using a specific kernel on the *pixels* of the image. This kernel is a concatenation of the polynomial kernel of degree 2 and the CNN. Specifically, small patches of the image is passed through the CNN to obtain single activations, which are then passed to a kernel as a similarity function. This interpretation is in line with interpreting neural nets as learnabe complex feature extractors. In this case, the CNN is trained on a fairly challenging dataset, then used as a mapping between arbitrary images and an activation space where the polynomial kernel more semantically meaningful than used directly on the pixel space.

With this interpretation, the effectiveness of neural texture synthesis is easy to understand. It assumes that the image has homogeneous patterns and tries to generate a new image that contains similar homogeneous patterns. Figure 2.2 illustrates this idea, though the actual distribution matching is done at the feature-level inside multiple layers of a convolutional neural network.

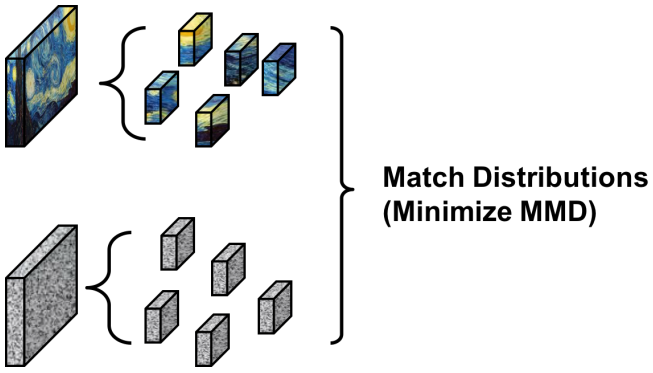


Figure 2.2: Neural texture synthesis views patches of the exemplar texture as samples and synthesizes a new image that minimizes the MMD between the synthetic patches and the exemplar patches.

The constraints of a convolutional neural network acts in an interesting manner for texture synthesis. If the patches of an image can be changed without any constraints, then MMD to zero can be minimized by mimicking the exact patches of the exemplar texture. However, different random noise can result in different textures because the convolutional neural network constrains that patches overlapping with each other must have the same values. This is also due to gradient descent being able to only find local optima. The use of a content reconstruction loss for style transfer acts similarly to constrain specific textures in certain regions on the image such that the generated image looks similar in structure to the content image.

Different choices of kernel and neural network

The authors of [22] recommend using the VGG-19 network architecture [82]. We try out different network architectures and kernel functions and find that this combination can lead to drastic changes in the generated result. Generated samples are shown in Figures 2.3, 2.4, and 2.5. The convolutional neural networks are trained on ImageNet. For each architecture, every downsampling layer (pooling or strided convolution) is used in a style reconstruction loss.

It seems that out of the four architectures tested, VGG19 with the polynomial kernel of degree 2 does perform best. Perhaps the way that the CNN was trained, or the architecture of the CNN itself, inhibits the use of the RBF kernel. A similarity mertic based on the dot product between CNN activations may more semantically align to our perception of visual

features than a similarity metric based on the Euclidean distance between CNN activations. It is yet unclear how to construct the optimal network architecture and kernel function for this method.

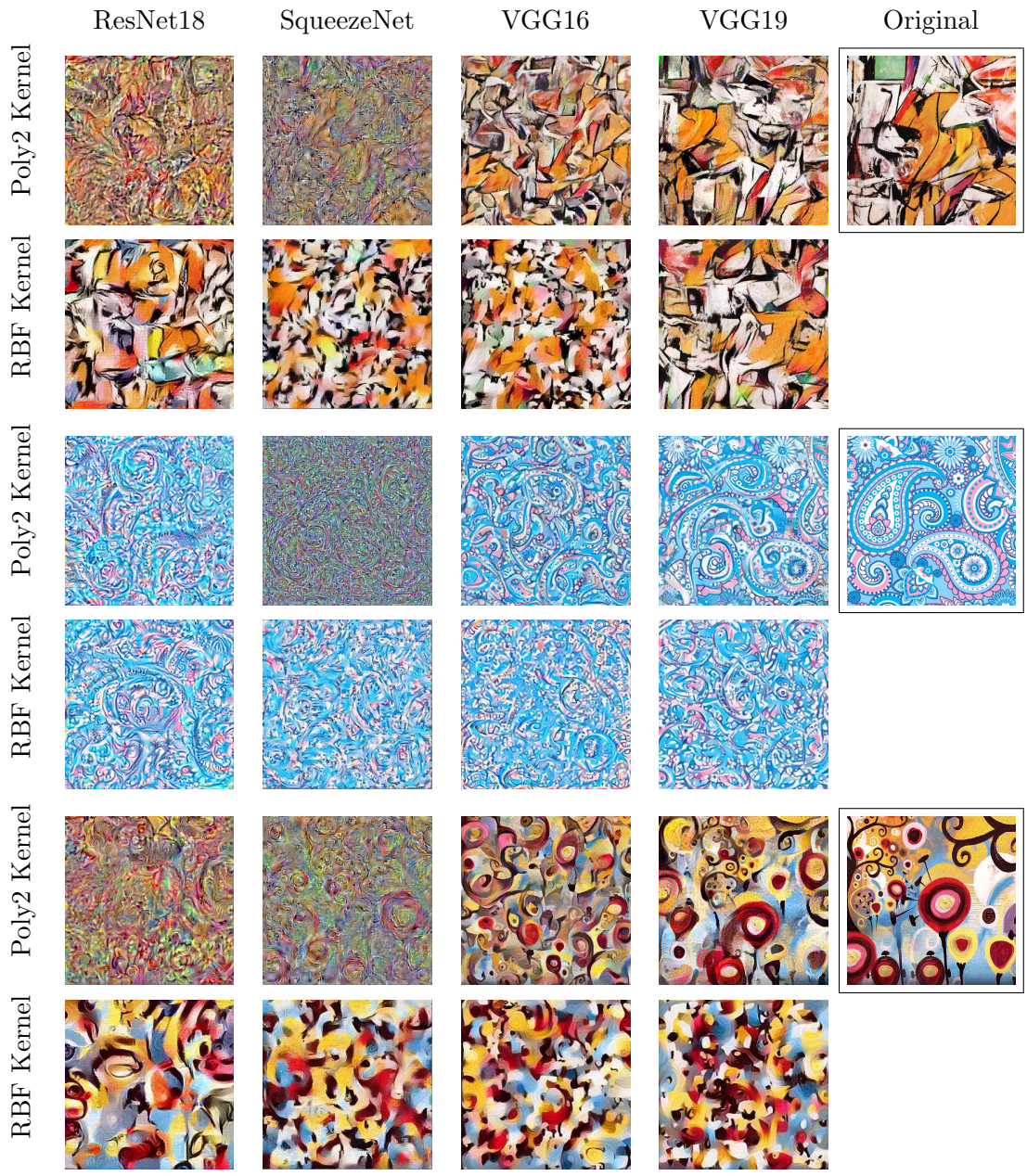


Figure 2.3: Comparison of texture synthesis results using different MMD kernels defined by the composition of a classifical kernel and a convolutional neural network. (Part I)

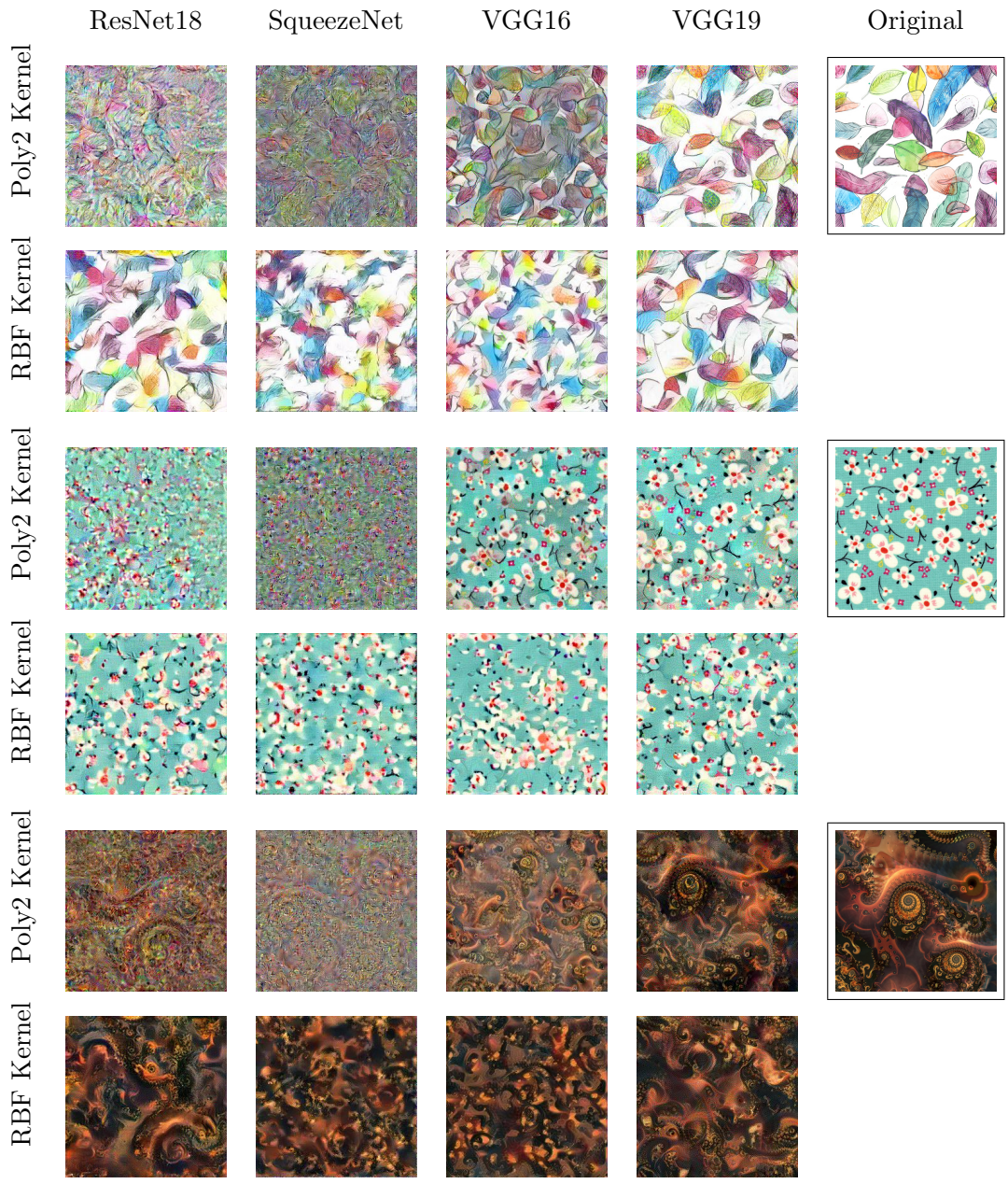


Figure 2.4: Comparison of texture synthesis results using different MMD kernels defined by the composition of a classifical kernel and a convolutional neural network. (Part II)

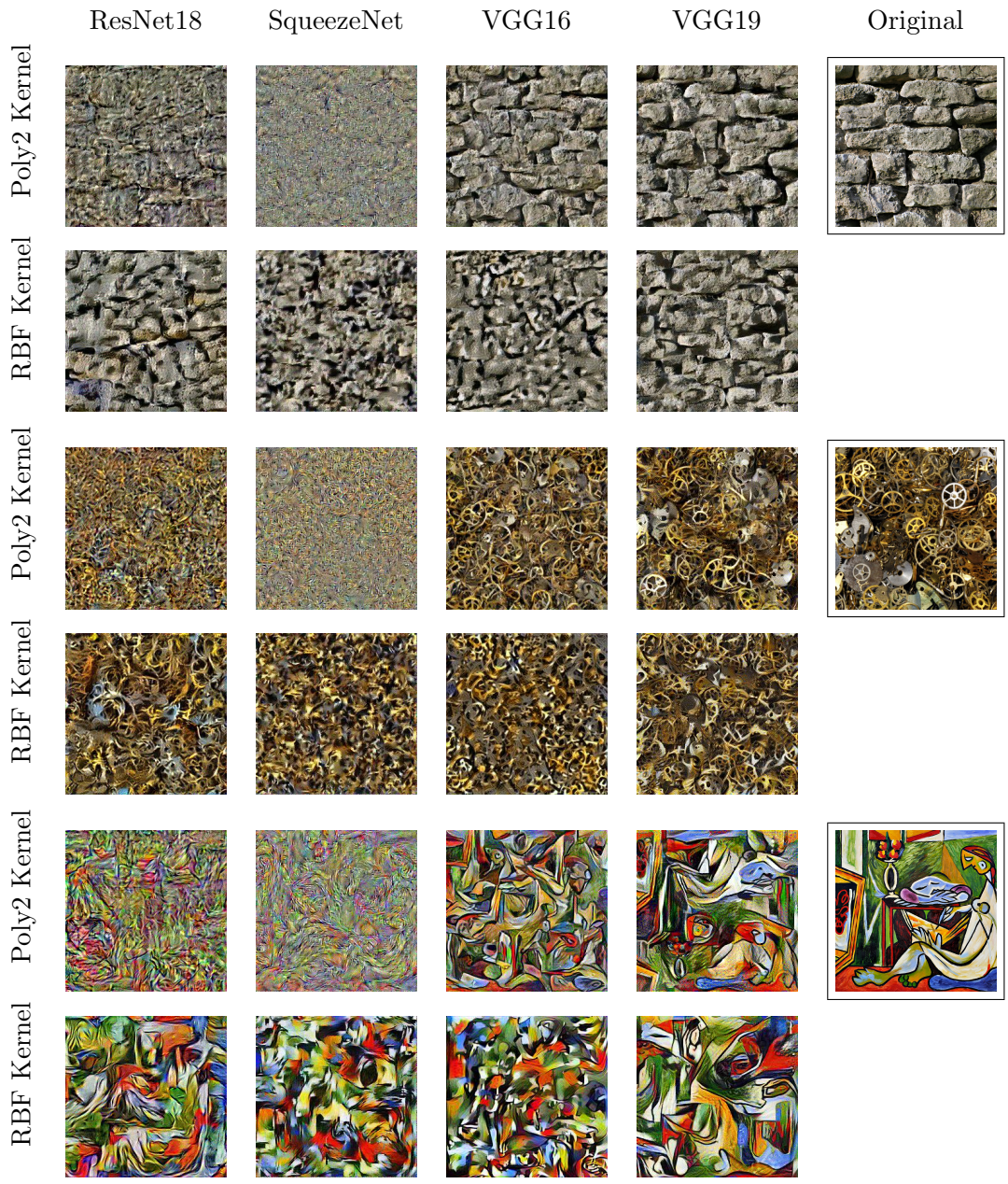


Figure 2.5: Comparison of texture synthesis results using different MMD kernels defined by the composition of a classifical kernel and a convolutional neural network . (Part III)

Chapter 3

Kernelized GAN

This chapter describes the use of maximum mean discrepancy as a criteria for training a generative model. Similar to generative adversarial networks (GANs), the kernelized GAN uses a neural network to discriminate between the real distribution and a distribution of outputs from a simultaneously-trained generator network. Once trained, the distribution of generator samples should be similar to the data generating distribution.

3.1 Motivation

Existing formulations of the GAN adversarial game are theorically equivalent to minimizing some divergence or probability metric between the real distribution (Section 2.3). The generator network minimizes this criterion while a discriminator tries to maximize it. A powerful discriminator should ideally be able to find discriminative features that separate the real data and the generated data, while also providing meaningful gradients to the generator.

This isn't always the case, however, when the discriminator is a fixed-length neural network. The network is usually not be powerful enough to exactly discriminate between the real and generated samples (even for finite samples). We propose the use of kernels to efficiently improve the complexity of the discriminator without requiring extra layers. This approach is theoretically motivated by the use of maximum mean discrepancy with a trained feature extractor, which is referred to as the discriminator in our approach.

Figure 3.3 shows the use of GMMN and our approach with a trained discriminator and Gaussian kernel. GMMN does not use a trained discriminator and already achieves superior performance compared to WGAN. For the 25-Gaussians problem, our approach does not visibly show a significant improvement upon GMMN, but one neat advantage is our approach is much less sensitive to the kernel parameter whereas we had to tune the bandwidth of the Gaussian kernel used in the toy GMMN illustration carefully to obtain good results.

Near the completion of this thesis, [53] independently proposes a similar method to combine MMD and adversarial training. However, their work requires an additional autoencoder whereas we show here that having a regularization term is sufficient to stabilize training. We also emphasize the existence of training instability when this regularization term is omitted.

3.2 MMD with a trained feature extractor

First, we define the deep MMD objective by appending a discriminator to the MMD criteria,

$$\begin{aligned} L(G, D) = \mathbb{E}_{x,x' \sim p_x}[k(D(x), D(x'))] + \mathbb{E}_{z,z' \sim p_y}[k(D(G(z)), D(G(z')))] \\ - 2\mathbb{E}_{x,z \sim p_x, p_z}[k(D(x), D(G(z)))] \end{aligned} \tag{3.1}$$

where k is any kernel function, p_x is the data distribution, and p_z is a fixed prior distribution. However, **naively appending a discriminator does not work**. To see why, first note that while integral probability metrics are typically defined as

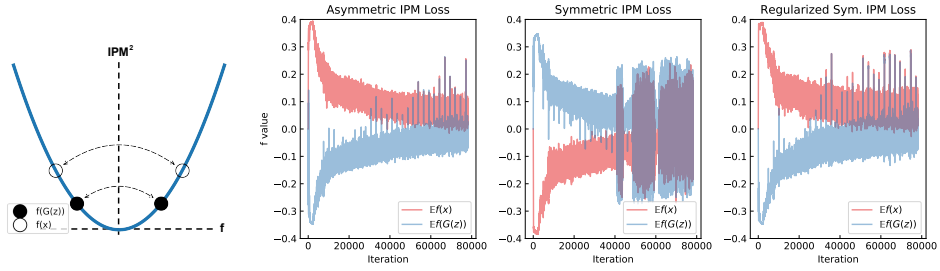
$$\text{IPM}(\mathcal{F}, p_x, p_y) = \sup_{f \in \mathcal{F}} |\mathbb{E}_{x \sim p_x} f(x) - \mathbb{E}_{y \sim p_y} f(y)| \tag{3.2}$$

If $\forall f \in \mathcal{F}, -f \in \mathcal{F}$, then the absolute value sign can be removed. The simplified asymmetric[1] version

$$\text{IPM}(\mathcal{F}, p_x, p_y) = \sup_{f \in \mathcal{F}} \mathbb{E}_{x \sim p_x} f(x) - \mathbb{E}_{y \sim p_y} f(y) \tag{3.3}$$

is used in existing IPM-based GAN frameworks [2, 28, 66]. Intuitively understanding the use of the asymmetric version (3.3) in a two-player GAN framework is straightforward. The discriminator's role is to approximate the supremum, so we refer to it as f in the following. The generator and discriminator both attempt to increase the value of f applied to the generated and real samples, respectively. However, using (3.2) would not lend to such simplification. The value of f for generated samples must "chase" the real samples. This allows the discriminator to skirt around the generator at every iteration, **without necessarily providing any meaningful gradients to the generator**.

[1] We refer to this equation as the asymmetric version because $\text{IPM}(p_x, p_y)$ is not necessarily equal to $\text{IPM}(p_y, p_x)$ for a fixed f.



(a) (b)

Figure 3.1: (a) Depiction of training instability with using the symmetric IPM. (b) Plots of the values of $\mathbb{E}f(x)$ and $\mathbb{E}f(G(z))$ during the course of training WGAN on CIFAR-10. For the rightmost figure, we add an asymmetric regularization described in (3.5) to stabilize training.

We show this instability in Figure 3.1. It is clear that when the generator is nearing optimum, the discriminator simply switches the sign of f and this causes the generator to become unstable. Let $\mathcal{S}_f$ be the subset of $\mathcal{F}$ that achieves a higher value than f in either (3.2) or (3.3) depending on context. We suspect that when the generator network shifts its outputs to minimize the IPM defined by the current discriminator, $|S_f|$ may actually increase instead of decrease. Whereas when the asymmetric IPM is used, $|S_f|$ should clearly decrease if the generator improves.

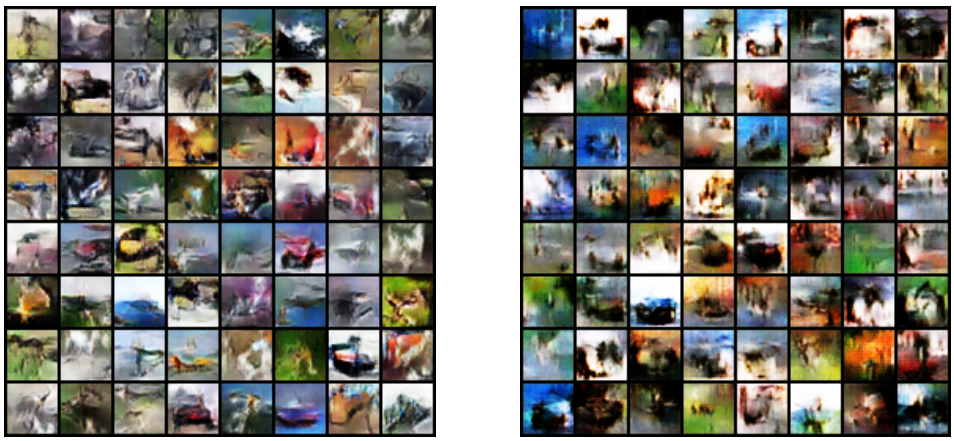
Visual samples are shown in Figure 3.2. While the generator still learns the general shapes and colors of the real data, there are visible large dark spots in the generated samples when either the absolute IPM or squared IPM is used.

For the same reason, naively adding a trained discriminator to the GMMN framework creates instability during training as only the squared MMD is computable in closed form.

$$\begin{aligned}\mathrm{MMD}^2(k, p_x, p_y) =& \mathbb{E}_{x,x'\sim p_x} k(x, x) + \mathbb{E}_{y,y'\sim p_y} k(y, y') \\ &- 2\mathbb{E}_{x,y\sim p_x,p_y} k(x, y).\end{aligned} \tag{3.4}$$

3.2.1 Fixing instability with asymetric constraints

The aforementioned instability problem occurs when the discriminator has multiple directions that can increase the IPM criterion. To fix this, we propose adding an asymmetric regularization term to the criterion that forces the discrimator to move in a single direction. This results in the



(a) $\sup_{f \in \mathcal{F}} \mathbb{E}f(x) - \mathbb{E}f(G(z))$

(b) $\sup_{f \in \mathcal{F}} |\mathbb{E}f(x) - \mathbb{E}f(G(z))|$

(c) $\sup_{f \in \mathcal{F}} (\mathbb{E}f(x) - \mathbb{E}f(G(z)))^2$

(d) $\sup_{f \in \mathcal{F}} (\mathbb{E}f(x) - \mathbb{E}f(G(z)))^2 + \min(\mathbb{E}f(x) - \mathbb{E}f(f), 0)$

Figure 3.2: Random samples after 100 epochs of training WGAN with weight clipping ($\mathcal{F} = \{f : ||f||_L \leq 1\}$) on CIFAR-10. The specific loss function used to train the discriminator is shown below each figure.

following two-player GAN framework,

$$\begin{aligned} &\min_G \; L(G,D) \\ &\max_D \; L(G,D) - \lambda || \min\{\mathbb{E}_{x\sim p_x} D(x) - \mathbb{E}_{z\sim p_z} D(G(z)), 0\}||^2 \end{aligned} \tag{3.5}$$

where $L(G,D)$ is as defined in (3.1). The added regularization term is negative whenever $D(x) < D(G(z))$, and zero otherwise. In order to maximize this expression, D is constrained such that $D(x) \geq D(G(z))$. We use the L2 norm to enact a larger penalty for large deviations so that the discriminator can quickly move back into the constrained region, while only a smaller penalty is given for small deviations so the discriminator does not focus on this regularization term too much. We find that larger values of λ work better, especially at the beginning of optimization as we want to force the discriminator to be in the right direction as early as possible.

3.2.2 Empirical estimates

We use an unbiased estimate of MMD [26]. For each minibatch of real samples $\{x_i\}_{i=1}^n$ and prior samples $\{z_i\}_{i=1}^n$, the empirical loss function is defined as

$$\begin{aligned} \hat{L}(G,D) = &\frac{1}{n(n-1)} \sum_{i=1}^{n} \sum_{j\neq i}^{n} k(D(x_i), D(x_j)) \\ &+ \frac{1}{n(n-1)} \sum_{i=1}^{n} \sum_{j\neq i}^{n} k(D(G(z_i)), D(G(z_j))) \\ &- \frac{2}{nm} \sum_{i=1}^{n} \sum_{j=1}^{n} k(D(x_i), D(G(z_j))) \end{aligned} \tag{3.6}$$

The generator and discriminator networks are then trained using

$$\begin{aligned} &\min_G \; \hat{L}(G,D) \\ &\max_D \; \hat{L}(G,D) - \lambda \left|\left| \min\left\{ \frac{1}{n}\sum_i D(x_i) - \frac{1}{n}\sum_j D(G(z_j)), 0 \right\} \right|\right|^2 \end{aligned} \tag{3.7}$$

3.2.3 Kernels and hyperparameters

We experiment with the linear and Gaussian kernels. The linear kernel is the simplest and can be computed directly as a dot product,

$$k_L(x,y) = x^T y. \tag{3.8}$$

The Gaussian kernel is a popular radial-basis kernel with a tunable parameter γ.

$$k_G(x, y) = \exp(\gamma||x - y||^2) \tag{3.9}$$

Note that one big advantage of combining a kernel with a deep neural network is that the network can adapt to the kernel being used. This implies that the parameter γ need not be manually specified, as the *weights of the neural network can simply learn to scale itself to effectively use any gamma.*

We additionally experiment by approximating the Gaussian kernel with random kitchen sinks (RKS) [75] features. Random kitchen sinks constructs approximate explicit features $\hat{\Phi}(\cdot)$ such that $\hat{\Phi}(x)^T\hat{\Phi}(y) \approx k_G(x, y)$. This approximate computation allows MMD to be computed in linear time, allowing larger minibatch sizes to be used in practice (see Appendix A for details). In experiments, we show that this approximation does not lead to obvious sample degradation. The RKS function $\Phi : \mathbb{R}^d \to \mathbb{R}^s$ is defined as

$$\hat{\Phi}(x) = \sqrt{\frac{2}{s}} \cos\left(\sqrt{2\gamma}Wx + b\right) \tag{3.10}$$

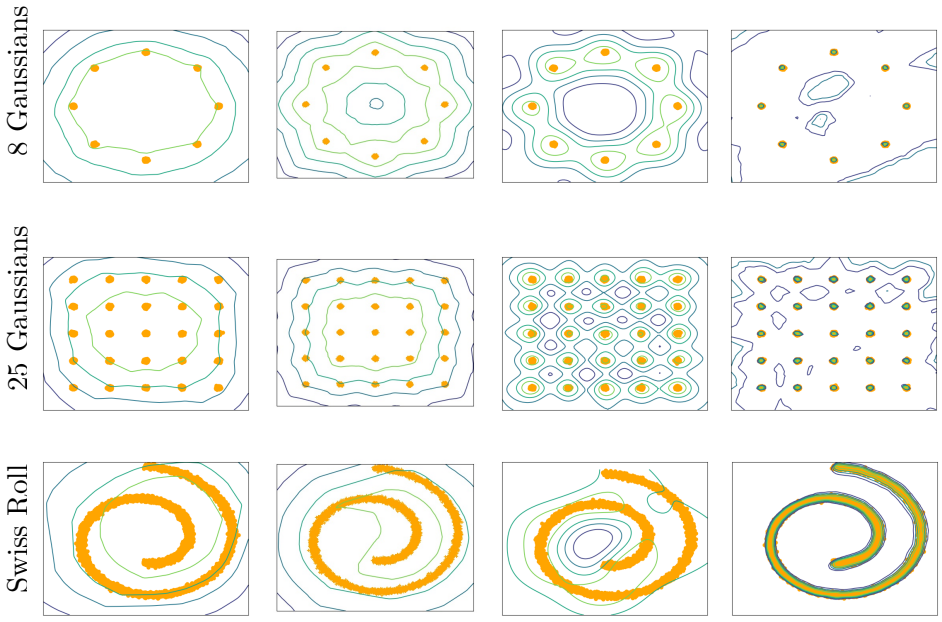
where $W \sim \text{Normal}(0, 1)$, $b \sim \text{Unif}(0, 2\pi)$. The approximation quality increases with s but is independent of d [75]. We fix s at 300 in our experiments.

3.2.4 Intuitions behind the squared MMD objective

Let $k_D(\cdot, \cdot) = k(D(\cdot), D(\cdot))$. A kernel function can be intuitively understood as a similarity measure between its two inputs. The squared MMD objective contains three terms

$$\begin{aligned}\text{MMD}^2(k, p_x, p_y) = &\underbrace{\mathbb{E}_{x,x' \sim p_x, p_x}[k_D(x, x')]}_{k_{xx}} + \underbrace{\mathbb{E}_{y,y' \sim p_y, p_y}[k_D(y, y')]}_{k_{yy}} \\ &- 2\underbrace{\mathbb{E}_{x,y \sim p_x, p_y}[k_D(x, y)]}_{k_{xy}}\end{aligned}$$

The discriminator tries to minimize k_{xx} and k_{yy}. The minimization of these terms ensures that the discriminator places the data from each distribution in a similar area of the activation space. The discriminator tries to maximize k_{xy}. This term contains the similarity between real and generated data, so the discriminator will try to place them in different areas of the activation space. This discriminator can be seen as clustering the two data distributions based on the choice of kernel function.



(a) WGAN (b) WGAN-GP (c) MMD (d) Kernelized GAN

Figure 3.3: IPM contours based on (a) approximate WGAN with weight clipping [2], (b) approximate WGAN with gradient penalty [28], (c) maximum mean discrepancy with Gaussian kernel, and (d) maximum mean discrepancy with trained discriminator and Gaussian kernel. The contour lines for kernelized GAN appear very close to the samples.

On the other hand, the generator tries to minimize k_{xy} while maximizing k_{xx}. The minimization of k_{xy} is the main objective that drives the generator to produce similar samples to real data. The maximization of k_{xx} is interesting as the kernel function can be interpreted as a similarity function, so this implies that the generator should generate dissimilar samples. If the generator always creates the same samples, or a limited number of samples, then k_{xx} will be low, whereas if the generator creates different samples all the time, then k_{xx} will be high. This term explicitly enforces the generator distribution to have high diversity.

3.3 Experiments

3.3.1 Toy distributions

We first show that the use of kernels do in fact help discriminate between distributions when only a finite function approximator is used. Though this property is difficult to observe in higher dimensions, it can be easily shown for synthetic toy distributions.

Figure 3.3 shows the resulting IPM contours based on different IPM estimates. The discriminator, when applicable, is a small multilayer perceptron with 4 ReLU layers and 64 hidden units. The contour lines show the values of $f(\cdot)$, which are either the discriminator output for WGAN methods or the witness function (2.9) for MMD and kernelized GAN. The orange points are samples from the real distribution while the "fake" distribution (not shown) is fixed to be the real distribution plus a small Gaussian noise. This implies that the boundary separating the real and fake distributions need to be extremely tight in order to correctly discriminate between them. However, only the MMD and kernelized GAN methods are able to discriminate between different clusters due to the use of a radial-basis kernel function.

This experiment shows that for the same discriminator architecture, kernelized GAN can provide a much more complex discriminative surface compared to WGAN. This is because kernelized GAN uses an exact maximum mean discrepancy with an optimized kernel, whereas WGAN uses an approximation to the Wasserstein distance. Note that in order to obtain nice contours for MMD, we had to manually tune γ for the Gaussian kernel (3.9). Whereas for kernelized GAN, we kept $\gamma = 1$ and only tuned the discriminator.

3.3.2 Qualitative samples

We show some random qualitative samples from our trained generators. By default, we use a minibatch size of 64 during training and the Gaussian kernel. All generator and discriminators use the DCGAN architecture [74].

MNIST. This dataset contains handwritten digits centered at a resolution of 28×28. Figure 3.4 shows random samples from trained generative moment matching networks (GMMN) and a kernelized GAN. Note that the GMMN algorithm requires using multiple Gaussian kernels with different values of γ and a large number of samples per iteration. In contrast, kernelized GAN discriminator can adapt to any value of *gamma* and so only a single Gaussian kernel is needed. The kernelized GAN can be successfully trained

Real GMMN (64) GMMN (1024) Kernelized GAN

Figure 3.4: Random samples from MNIST dataset. GMMN requires a much higher minibatch size to produce quality samples.

with the typical minibatch size of 64, owing to the effective representational power of the discriminator.

LFW. The Labeled Faces in the Wild (LFW) dataset contains faces of celebrities. We perform a center crop and resize the images to 64×64 resolution. Figure 3.5 shows random samples from the training dataset, along with generators using the GMMN algorithm and kernelized GAN. Generated images are mapped to values in (0, 1). As GMMN does not use a neural network discriminator, it is unable to extract important features of the dataset. For example, some samples have large smudges, a third eye, or a disfigured mouth. The discriminator in kernelized GAN is able to transform the image into a more meaningful space where the Gaussian kernel is able to distinguish samples more easily, leading to a better trained generator.

LSUN Bedrooms. This dataset contains 3 million images of bedrooms, which we resize to a resolution of 64×64. Figure 3.6 shows samples from kernelized GAN with different kernels. We show that the model trains adequately with kernel functions other than the typical Gaussian kernel. While the linear kernel is not characteristic, it still produces a good generative model with no obvious deficiencies. An advantage of using random kitchen sinks to approximate the Gaussian kernel is that it can be computed in linear time whereas Gaussian kernel is quadratic with respect to the minibatch size. This show that kernelized GAN can potentially scale up to large minibatch sizes, unlike GMMN.

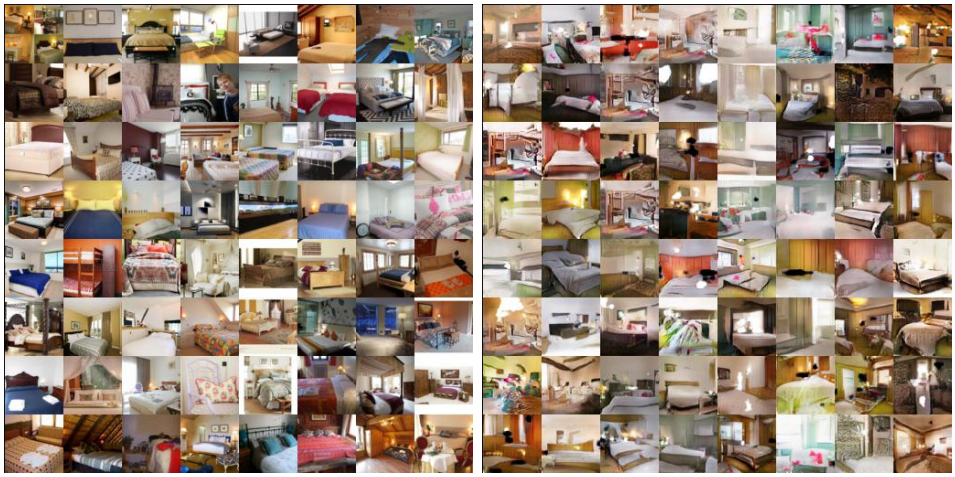
3.3.3 Latent space interpolation

A degenerate case of a trained generative model is to only memorize and only generate samples from the training dataset. For latent variable models, we can visualize how changing the latent state produces diverse samples. In

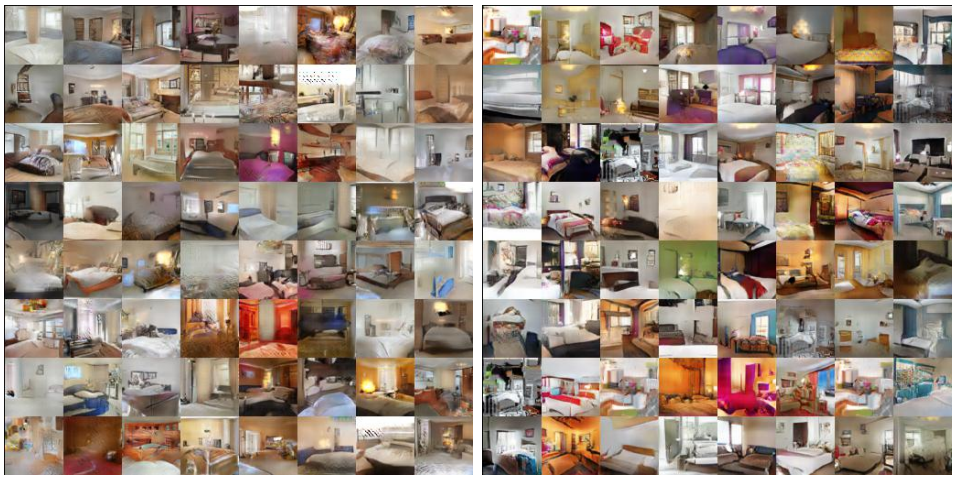
Real GMMN

Kernelized GAN

Figure 3.5: Random samples from the LFW dataset. GMMN is trained using a batchsize of 1024 whereas kernelized GAN uses a batchsize of 64.



Real | Gaussian kernel



RKS kernel | Linear kernel

Figure 3.6: Random samples from the LSUN dataset. Kernelized GAN can be trained using different kernels, with the discriminator appropriately adapting to the specific manifold required to use the kernel.

Figures 3.7 and 3.8, we show random samples on the leftmost and rightmost columns with interpolated samples in-between. We see that in most cases, there is a smooth transition showing that the model learns to generalize instead of memorize. In 3.7, there are signs of a smooth transition from faces with glasses to faces without glasses, and from neutral expressions to happier expressions. In 3.8, we see transitions where windows turn to cabinets or drapes, and chairs turning into beds.

3.3.4 Conditional generation

The simplest version of a conditional GAN [65] involves including image labels to both the generator and discriminator networks. As there is no discriminator network in GMMN, it is possible to implement a conditional generative moment matching network. However, with kernelized GAN, we can augment the generator and discriminator as described in [65] to create conditional samples. Figure 3.9 shows conditional samples from a kernelized GAN. It should also be possible to augment kernelized GAN with more complicated conditional generation such as [6].

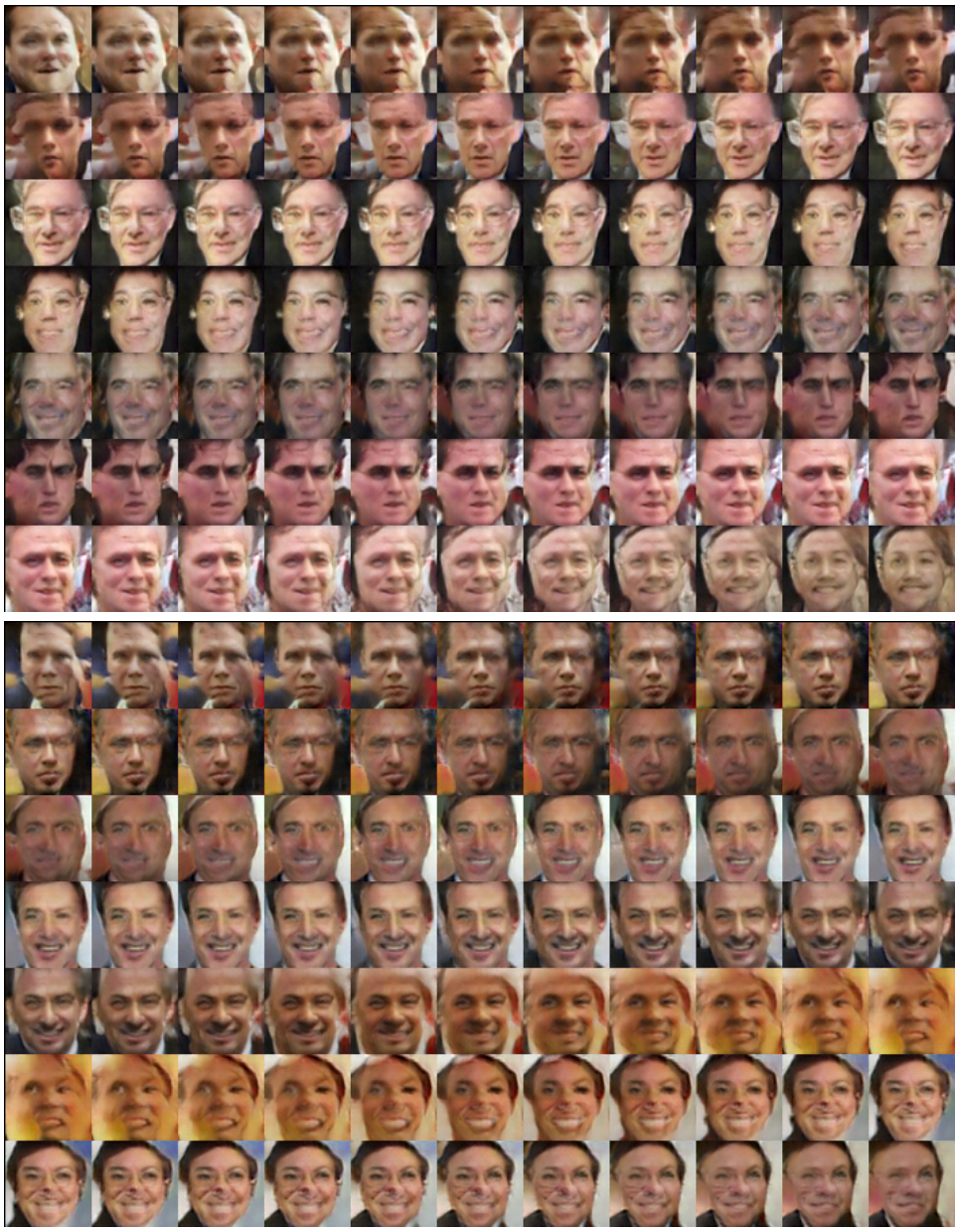


Figure 3.7: Interpolation within latent space for Kernelized GAN trained on LFW.

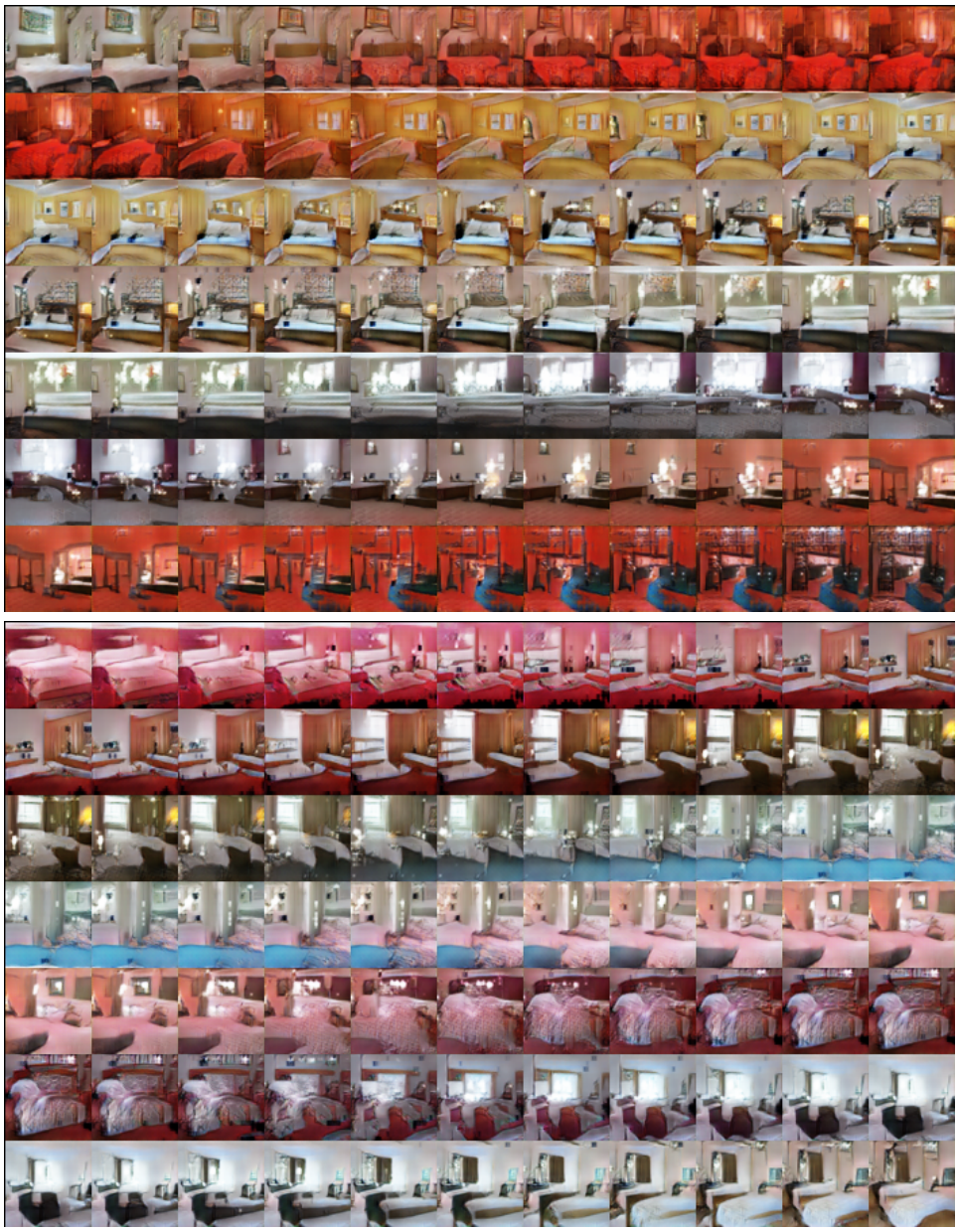


Figure 3.8: Interpolation within latent space for Kernelized GAN trained on LSUN bedrooms.

Figure 3.9: Conditional generation of MNIST digits. Each row corresponds to different label from 0 to 9.

Chapter 4

Feedforward style transfer

Famous artists are typically renowned for a particular artistic style, which takes years to develop. Even once perfected, a single piece of art can take days or even months to create. This motivates us to explore efficient computational strategies for creating artistic images. While there is a large classical literature on texture synthesis methods that create artwork from a blank canvas [18, 47, 56, 89], several recent approaches study the problem of transferring the desired *style* from one image onto the structural *content* of another image. This approach is known as artistic style transfer.

Methods for texture synthesis often create generative processes where carefully chosen insertions of random variables creates diverse textures. The neural texture synthesis algorithm described in Chapter 2 is an example where the generative process involves sampling a noise image, then iteratively refining the image to create a new texture sample. This method can be adapted to style transfer by conditioning on a content image in the generative process. The neural style transfer algorithm simply adds an appropriate content reconstruction loss (2.14) to redirect the iterative process. See Chapter 2 for a more thorough description for neural texture synthesis and style transfer.

4.1 The need for faster algorithms

Despite renewed interest in the domain, the actual process of style transfer is based on solving a complex optimization procedure, which can take minutes on today's hardware. A typical speedup solution is to train another neural network that approximates the optimum of the optimization in a single feed-forward pass [14, 41, 87, 88]. While much faster, existing works that use this approach sacrifice the versatility of being able to perform style transfer with any given style image, as the feed-forward network cannot generalize beyond its trained set of images. Due to this limitation, existing applications are either time-consuming or limited in the number of provided styles, depending on the method of style transfer.

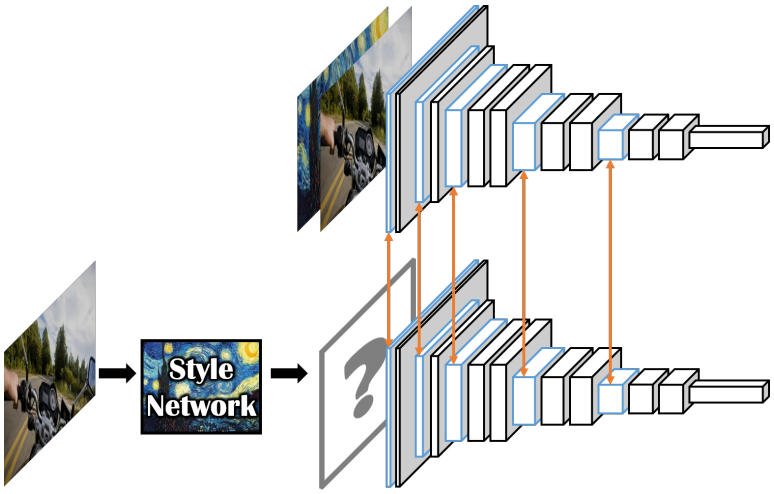


Figure 4.1: Illustration of existing feedforward methods [14, 41, 87, 88] that simply try to minimize (2.17) by training a separate neural network for each style image, or a limited number of style images.

In this chapter we propose a method that addresses these limitations: a new method for artistic style transfer that is efficient but is not limited to a finite set of styles. To accomplish this, we define a new optimization objective for style transfer that notably only depends on one layer of the CNN (as opposed to existing methods that use multiple layers). The new objective leads to visually-appealing results while this simple restriction allows us to use an "inverse network" to deterministically invert the activations from the stylized layer to yield the stylized image.

While it is possible to train a neural network that approximates the optimum of Gatys *et al.*'s loss function (see Section 2.4) for one or more fixed styles [14, 41, 87, 88]. This yields a much faster method, but these methods need to be re-trained for each new style. Figure 4.1 highlights the limitation of this method as a new neural network must be trained for each new style. It should be noted that these works all minimize the objective defined by [24].

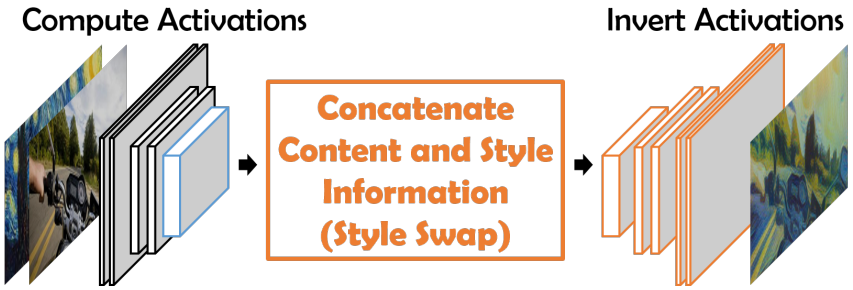


Figure 4.2: We propose a one-shot concatenation method based on a simple nearest neighbour alignment to combine the content and style activations. The combined activations are then inverted back into an image by a trained inverse network.

4.2 Style transfer as one-shot distribution alignment

The main component of our style transfer method is a patch-based operation for constructing the target activations in a single layer, given the style and content images. We refer to this procedure as "swapping the style" of an image, as the content image is replaced patch-by-patch by the style image. We first present this operation at a high level, followed by more details on our implementation.

4.2.1 Style Swap

Let C and S denote the RGB representations of the content and style images (respectively), and let $\Phi(\cdot)$ be the function represented by a fully convolutional part of a pretrained CNN that maps an image from RGB to some intermediate activation space. After computing the activations, $\Phi(C)$ and $\Phi(S)$, the *style swap* procedure is as follows:

1. Extract a set of patches for both content and style activations, denoted by $\{\phi_i(C)\}_{i \in n_c}$ and $\{\phi_j(S)\}_{j \in n_s}$, where n_c and n_s are the number of extracted patches. The extracted patches should have sufficient overlap, and contain all channels of the activations.

2. For each content activation patch, determine a closest-matching style

patch based on the normalized cross-correlation measure,

$$\phi_i^{ss}(C,S) := \underset{\phi_j(S),\, j=1,\ldots,n_s}{\operatorname{argmax}} \frac{\langle \phi_i(C), \phi_j(S) \rangle}{||\phi_i(C)|| \cdot ||\phi_j(S)||}. \tag{4.1}$$

3. Swap each content activation patch $\phi_i(C)$ with its closest-matching style patch $\phi_i^{ss}(C,S)$.

4. Reconstruct the complete content activations, which we denote by $\Phi^{ss}(C,S)$, by averaging overlapping areas that may have different values due to step 3.

This operation results in hidden activations corresponding to a single image with the structure of the content image, but with textures taken from the style image.

4.2.2 Comparison with neural texture synthesis

The neural texture synthesis algorithm minimizes the maximum mean discrepancy between the patches $\{\phi_i(S)\}$ and $\{\phi_j(N)\}$ where S is the style image and N is a white noise image. The MMD kernel used in neural texture synthesis is a polynomial kernel of degree 2. In contrast, style swap takes two sets of patches $\{\phi_i(S)\}$ and $\{\phi_j(N)\}$ and performs a distribution alignment by replacing each patch by their closest neighbour (Figure 4.3), with a similarity metric related to the MMD kernel.

4.2.3 Parallelizable implementation

To give an efficient implementation, we show that the entire style swap operation can be implemented as a network with three operations: (i) a 2D convolutional layer, (ii) a channel-wise argmax, and (iii) a 2D transposed convolutional layer. Implementation of style swap is then as simple as using existing efficient implementations of 2D convolutions and transposed convolutions[2].

To concisely describe the implementation, we re-index the content activation patches to explicitly denote spatial structure. In particular, we'll let d be the number of feature channels of $\Phi(C)$, and let $\phi_{a,b}(C)$ denote the patch $\Phi(C)_{a:a+s,\, b:b+s,\, 1:d}$ where s is the patch size.

[2]The transposed convolution is also often referred to as a "fractionally-strided" convolution, a "backward" convolution, an "upconvolution", or a "deconvolution".

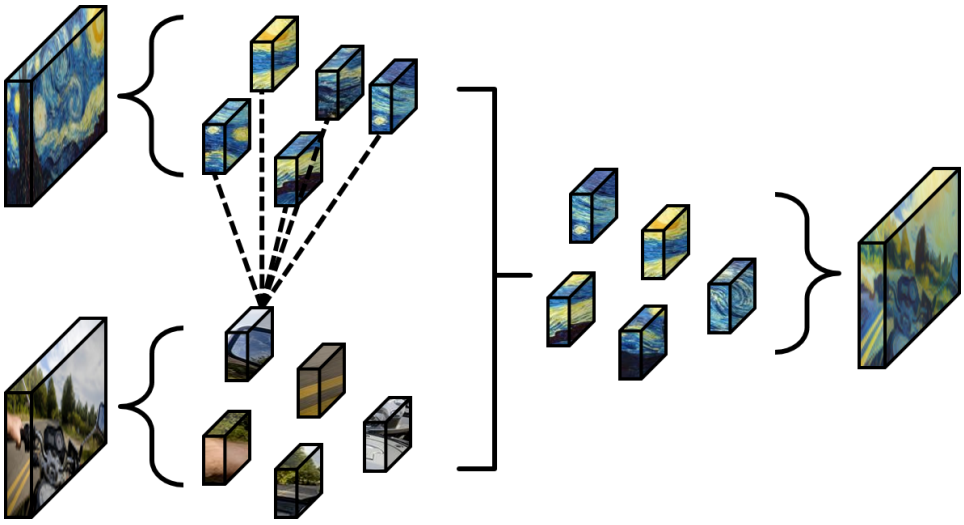


Figure 4.3: Style swap performs a forced distribution alignment by replacing each content patch by its nearest neighbour style patch.

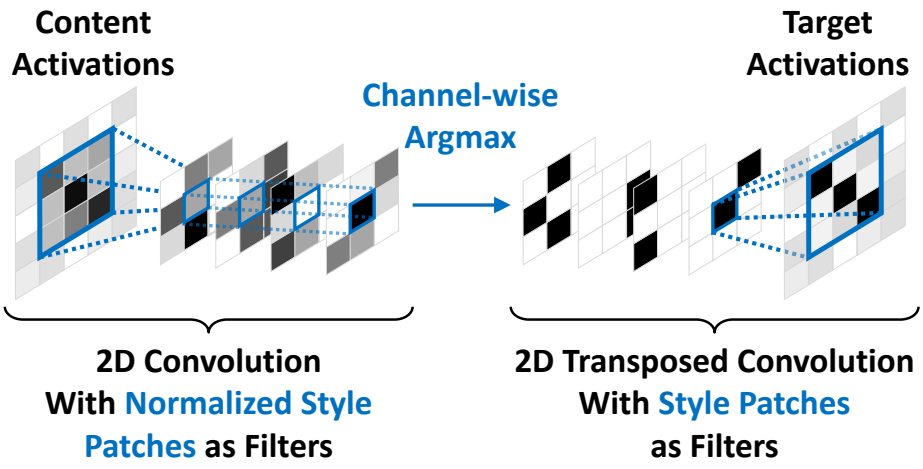


Figure 4.4: Illustration of a style swap operation. The 2D convolution extracts patches of size 3×3 and stride 1, and computes the normalized cross-correlations. There are $n_c = 9$ spatial locations and $n_s = 4$ feature channels immediately before and after the channel-wise argmax operation. The 2D transposed convolution reconstructs the complete activations by placing each best matching style patch at the corresponding spatial location.

Notice that the normalization term for content activation patches $\phi_i(C)$ is constant with respect to the argmax operation, so (4.1) can be rewritten as

$$\begin{aligned} K_{a,b,j} &= \left\langle \phi_{a,b}(C), \frac{\phi_j(S)}{||\phi_j(S)||} \right\rangle \\ \phi^{ss}_{a,b}(C,S) &= \underset{\phi_j(S),\, j \in N_s}{\text{argmax}} \{K_{a,b,j}\} \end{aligned} \tag{4.2}$$

The lack of a normalization for the content activation patches simplifies computation and allows our use of 2D convolutional layers. The following three steps describe our implementation and are illustrated in Figure 4.4:

- The tensor K can be computed by a single 2D convolution by using the normalized style activations patches $\{\phi_j(S)/||\phi_j(S)||\}$ as convolution filters and $\Phi(C)$ as input. The computed K has n_c spatial locations and n_s feature channels. At each spatial location, $K_{a,b}$ is a vector of cross-correlations between a content activation patch and all style activation patches.

- To prepare for the 2D transposed convolution, we replace each vector $K_{a,b}$ by a one-hot vector corresponding to the best matching style activation patch.

$$\overline{K}_{a,b,j} = \begin{cases} 1 & \text{if } j = \text{argmax}_{j'}\{K_{a,b,j'}\} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

- The last operation for constructing $\Phi^{ss}(C,S)$ is a 2D transposed convolution with $\overline{K}$ as input and unnormalized style activation patches $\{\phi_j(S)\}$ as filters. At each spatial location, only the best matching style activation patch is in the output, as the other patches are multiplied by zero.

Note that a transposed convolution will sum up the values from overlapping patches. In order to average these values, we perform an element-wise division on each spatial location of the output by the number of overlapping patches. Consequently, we do not need to impose that the argmax in (4.3) has a unique solution, as multiple argmax solutions can simply be interpreted as adding more overlapping patches.

4.2.4 Optimization formulation

The pixel representation of the stylized image can be computed by placing a loss function on the activation space with target activations $\Phi^{ss}(C, S)$. Similar to prior works on style transfer [23, 51], we use the squared-error loss and define our optimization objective as

$$I_{stylized}(C,S) = \underset{I \in \mathbb{R}^{h \times w \times d}}{\operatorname{argmin}} ||\Phi(I) - \Phi^{ss}(C,S)||_F^2 + \lambda \ell_{TV}(I) \tag{4.4}$$

where we'll say that the synthesized image is of dimension h by w by d, $||\cdot||_F$ is the Frobenius norm, and $\ell_{TV}(\cdot)$ is the total variation regularization term widely used in image generation methods [1, 41, 61]. Because $\Phi(\cdot)$ contains multiple maxpooling operations that downsample the image, we use this regularization as a natural image prior, obtaining spatially smoother results for the re-upsampled image. The total variation regularization is as follows:

$$\begin{aligned} \ell_{TV}(I) = &\sum_{i=1}^{h-1}\sum_{j=1}^{w}\sum_{k=1}^{d}(I_{i+1,j,k} - I_{i,j,k})^2 \\ &+ \sum_{i=1}^{h}\sum_{j=1}^{w-1}\sum_{k=1}^{d}(I_{i,j+1,k} - I_{i,j,k})^2 \end{aligned} \tag{4.5}$$

Since the function $\Phi(\cdot)$ is part of a pretrained CNN and is at least once subdifferentiable, (4.4) can be computed using standard subgradient-based optimization methods.

4.3 Inverse network

Unfortunately, the cost of solving the optimization problem to compute the stylized image might be too high in applications such as video stylization. We can improve optimization speed by approximating the optimum using another neural network. Once trained, this network can then be used to produce stylized images much faster, and we will in particular train this network to have the versatility of being able to use new content and new style images.

The main purpose of our inverse network is to approximate an optimum of the loss function in (4.4) for any target activations. We therefore define

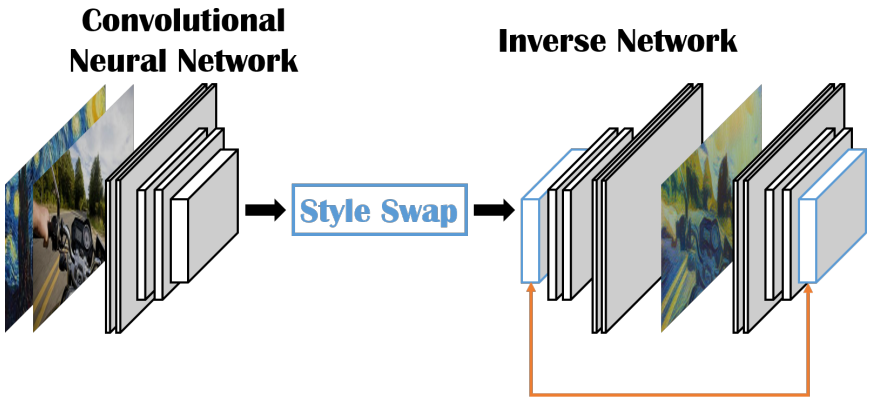


Figure 4.5: The inverse network takes the style swapped activations and produces an image. This network is trained by minimizing the L2 norm (orange line) between the style swapped activations and the activations of the image after being passed through the pretrained network again.

the optimal inverse function as:

$$\underset{f}{\operatorname{arginf}}\ \mathbb{E}_H\left[||\Phi(f(H)) - H||_F^2 + \lambda \ell_{TV}\left(f(H)\right)\right] \tag{4.6}$$

where f represents a deterministic function and H is a random variable representing target activations. The total variation regularization term is added as a natural image prior similar to (4.4).

4.3.1 Training the inverse network

A couple problems arise due to the properties of the pretrained convolutional neural network.

Non-injective. The CNN defining $\Phi(\cdot)$ contains convolutional, max-pooling, and ReLU layers. These functions are many-to-one, and thus do not have well-defined inverse functions. Akin to existing works that use inverse networks [11, 59, 95], we instead train an approximation to the inverse relation by a parametric neural network.

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} ||\Phi(f(H_i;\theta)) - H_i||_F^2 + \lambda \ell_{TV}\left(f(H_i;\theta)\right) \tag{4.7}$$

where θ denotes the parameters of the neural network f and H_i are activation features from a dataset of size n. This objective function leads to unsupervised

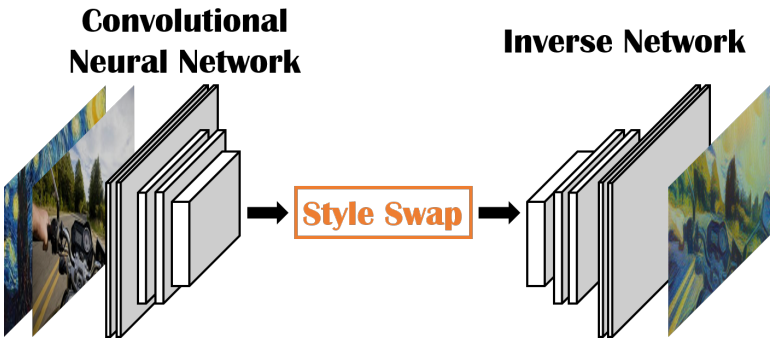


Figure 4.6: We propose the first feedforward method for style transfer that can be used for arbitrary style images. We formulate style transfer using a constructive procedure (Style Swap) and train an inverse network to generate the image.

training of the neural network as the optimum of (4.4) does not need to be known. We place the description of our inverse network architecture in the appendix.

Non-surjective. The style swap operation produces target activations that may be outside the range of $\Phi(\cdot)$ due to the interpolation. This would mean that if the inverse network is only trained with real images then the inverse network may only be able to invert activations in the range of $\Phi(\cdot)$. Since we would like the inverse network to invert style swapped activations, we augment the training set to include these activations. More precisely, given a set of training images (and their corresponding activations), we augment this training set with style-swapped activations based on pairs of images.

4.3.2 Feedforward style transfer procedure

Once trained, the inverse network can be used to replace the optimization procedure. Thus our proposed feedforward procedure consists of the following steps:

1. Compute $\Phi(C)$ and $\Phi(S)$.
2. Obtain $\Phi^{ss}(C, S)$ by style swapping.
3. Feed $\Phi^{ss}(C, S)$ into a trained inverse network.

This procedure is illustrated in Figure 4.6. As described in Section 4.2.3, style swapping can be implemented as a (non-differentiable) convolutional neural

network. As such, the entire feedforward procedure can be seen as a neural net with individually trained parts. Compared to existing feedforward approaches [14, 41, 87, 88], the biggest advantage of our feedforward procedure is the ability to use new style images with only a single trained inverse network.

4.4 Experiments

In this section, we analyze properties of the proposed style transfer and inversion methods. We use the Torch7 framework [7] to implement our method[3], and use existing open source implementations of prior works [40, 51, 76] for comparison.

4.4.1 Style swap results

Target Layer. The effects of style swapping in different layers of the VGG-19 network are shown in Figure 4.7. In this figure the RGB images are computed by optimization as described in Section 4.2. We see that while we can style swap directly in the RGB space, the result is nothing more than a recolor. As we choose a target layer that is deeper in the network, textures of the style image are more pronounced. We find that style swapping on the "relu3_1" layer provides the most visually pleasing results, while staying structurally consistent with the content. We restrict our method to the "relu3_1" layer in the following experiments and in the inverse network training. Qualitative results are shown in Figure 4.13.

Consistency. Our style swapping approach concatenates the content and style information into a single target feature vector, resulting in an easier optimization formulation compared to other approaches. As a result, we find that the optimization algorithm is able to reach the optimum of our formulation in less iterations than existing formulations while consistently reaching the same optimum. Figures 4.8 and 4.9 show the difference in optimization between our formulation and existing works under random initializations. Here we see that random initializations have almost no effect on the stylized result, indicating that we have far fewer local optima than other style transfer objectives.

Straightforward Adaptation to Video. This consistency property is advantageous when stylizing videos frame by frame. Frames that are the same will result in the same stylized result, while consecutive frames will be stylized in similar ways. As a result, our method is able to adapt to video

[3] Code available at https://github.com/rtqichen/style-swap

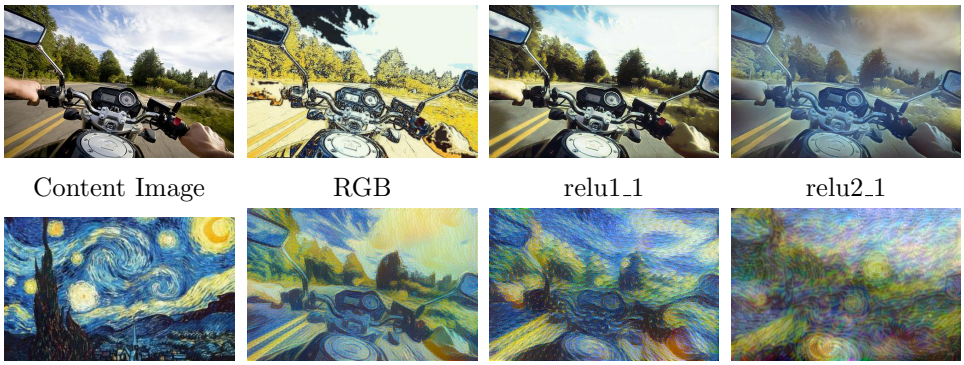


Figure 4.7: The effect of style swapping in different layers of VGG-19 [82], and also in RGB space. Due to the naming convention of VGG-19, "relu$X$_1" refers to the first ReLU layer after the $(X-1)$-th maxpooling layer. The style swap operation uses patches of size 3×3 and stride 1, and then the RGB image is constructed using optimization.

Figure 4.8: Our method achieves consistent results compared to existing optimization formulations. We see that Gatys *et al.*'s formulation [23] has multiple local optima while we are able to consistently achieve the same style transfer effect with random initializations. Figure 4.9 shows this quantitatively.

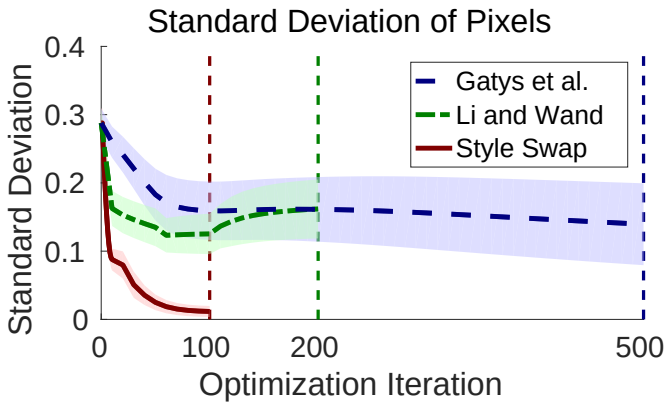


Figure 4.9: Standard deviation of the RGB pixels over the course of optimization is shown for 40 random initializations. The lines show the mean value and the shaded regions are within one standard deviation of the mean. The vertical dashed lines indicate the end of optimization. Figure 4.8 shows examples of optimization results.

without any explicit gluing procedure like optical flow [76]. We place stylized videos in the code repository.

Simple Intuitive Tuning. A natural way to tune the degree of stylization (compared to preserving the content) in the proposed approach is to modify the patch size. Figure 4.10 qualitatively shows the relationship between patch size and the style-swapped result. As the patch size increases, more of the structure of the content image is lost and replaced by textures in the style image.

4.4.2 CNN inversion

Here we describe our training of an inverse network that computes an approximate inverse function of the pretrained VGG-19 network [82]. More specifically, we invert the truncated network from the input layer up to layer "relu3_1". The network architecture is placed in the appendix.

Dataset. We train using the Microsoft COCO (MSCOCO) dataset [57] and a dataset of paintings sourced from wikiart.org and hosted by Kaggle [12]. Each dataset has roughly $80,000$ natural images and paintings, respectively. Since typically the content images are natural images and style images are paintings, we combine the two datasets so that the network can learn to recreate the structure and texture of both categories of images. Additionally,

patch size 3×3 patch size 7×7 patch size 12×12

Figure 4.10: We can tradeoff between content structure and style texture by tuning the patch size. The style images, *Starry Night* (top) and *Small Worlds I* (bottom), are shown in Figure 4.13.

the explicit categorization of natural image and painting gives respective content and style candidates for the augmentation described in Section 4.3.1.

Training. We resize each image to 256×256 pixels (corresponding to activations of size 64×64) and train for approximately 2 epochs over each dataset. Note that even though we restrict the size of our training images (and corresponding activations), the inverse network is fully convolutional and can be applied to arbitrary-sized activations after training.

We construct each minibatch by taking 2 activation samples from natural images and 2 samples from paintings. We augment the minibatch with 4 style-swapped activations using all pairs of natural images and paintings in the minibatch. We calculate subgradients using backpropagation on (4.7) with the total variance regularization coefficient $\lambda = 10^{-6}$ (the method is not particularly sensitive to this choice), and we update parameters of the network using the Adam optimizer [43] with a fixed learning rate of 10^{-3}.

Result. Figure 4.11 shows quantitative approximation results using 2000 full-sized validation images from MSCOCO and 6 full-sized style images. Though only trained on images of size 256×256, we achieve reasonable results for arbitrary full-sized images. We additionally compare against an inverse network that has the same architecture but was not trained with the augmentation. As expected, the network that never sees style-swapped activations during training performs worse than the network with

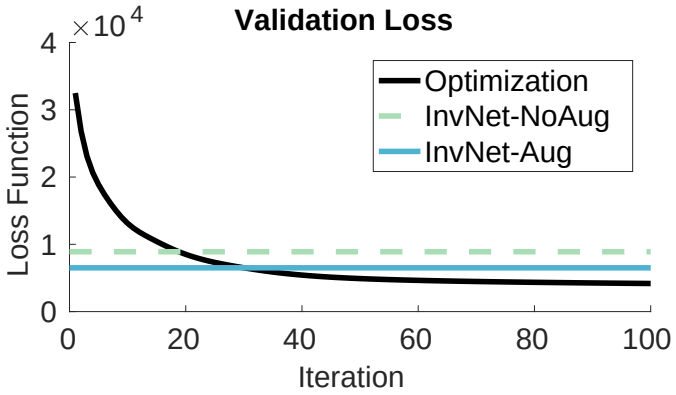


Figure 4.11: We compare the average loss (4.7) achieved by optimization and our inverse networks on 2000 variable-sized validation images and 6 variable-sized style images, using patch sizes of 3×3. Style images that appear in the paintings dataset were removed during training.

the augmented training set.

4.4.3 Computation time

Computation times of existing style transfer methods are listed in Table 4.1. Compared to optimization-based methods, our optimization formula is easier to solve and requires less time per iteration, likely due to only using one layer of the pretrained VGG-19 network. Other methods use multiple layers and also deeper layers than we do.

We show the percentage of computation time spent by different parts of our feedforward procedure in Figures 4.12a and 4.12c. For any nontrivial image sizes, the style swap procedure requires much more time than the other neural networks. This is due to the style swap procedure containing two convolutional layers where the number of filters is the number of style patches. The number of patches increases linearly with the number of pixels of the image, with a constant that depends on the number of pooling layers and the stride at which the patches are extracted. Therefore, it is no surprise that style image size has the most effect on computation time (as shown in Figures 4.12a and 4.12b).

Interestingly, it seems that the computation time stops increasing at some point even when the content image size increases (Figure 4.12d), likely due to parallelism afforded by the implementation. This suggests that our procedure can handle large image sizes as long as the number of style patches

Method	N. Iters.	Time/Iter. (s)	Total (s)
Gatys *et al.*[23]	500	0.1004	50.20
Li and Wand [51]	200	0.6293	125.86
Style Swap (Optim)	100	0.0466	4.66
Style Swap (InvNet)	1	1.2483	1.25

Table 4.1: Mean computation times of style transfer methods that can handle arbitary style images. Times are taken for images of resolution 300×500 on a GeForce GTX 980 Ti. Note that the number of iterations for optimization-based approaches should only be viewed as a very rough estimate.

is kept manageable. It may be desirable to perform clustering on the style patches to reduce the number of patches, or use alternative implementations such as fast approximate nearest neighbour search methods [29, 68].

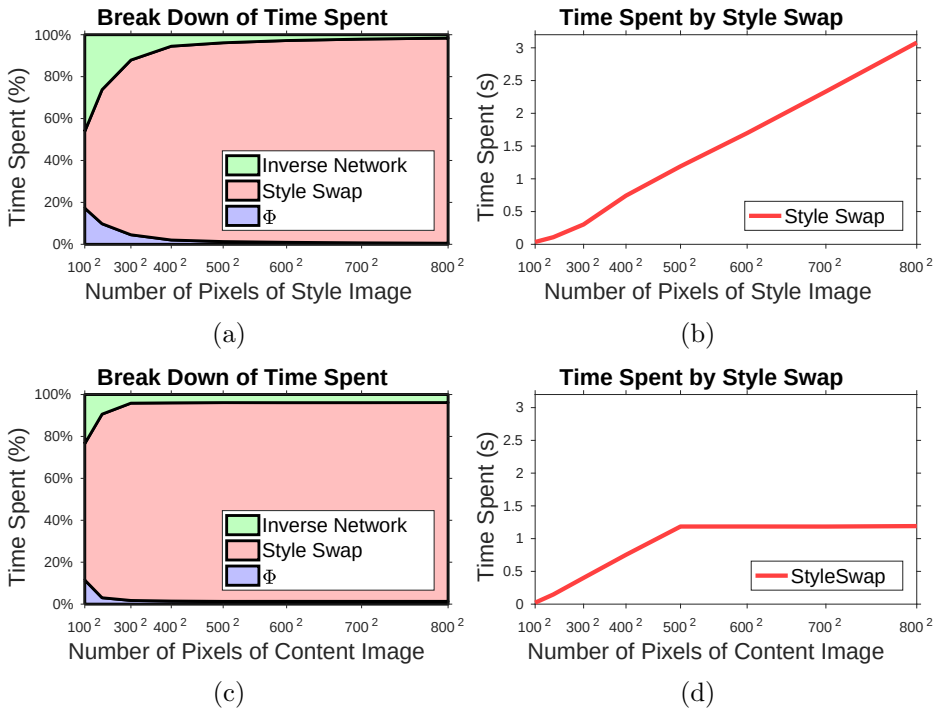


Figure 4.12: Compute times as (a,b) style image size increases and (c,d) as content image size increases. The non-variable image size is kept at 500×500. As shown in (a,c), most of the computation is spent in the style swap procedure.

Figure 4.13: Qualitative examples of our method compared with Gatys *et al.*'s formulation of artistic style transfer.

Chapter 5

Conclusion

We have discussed the use of maximum mean discrepancy in learning generative models in two different settings. The first setting covers a general use case where the generative process for a training dataset is approximated. The second setting covers a more specific use case of combining content and style images in a generative process that produces a new artistic image. We show generalization of our style transfer method to arbitrary style images, as prior work focused on speed have not been successful at generalization.

For future work, it would be ideal to increase the quality or speed of the style transfer without sacrificing generalization, as the proposed method contains a three-way trade off between speed, quality, and generalization with no outstanding performance in either speed or quality. Speed can be achieved by replacing the patch-based distributional alignment with a faster one, while quality can be increased by using existing neural style transfer loss [24] instead of the autoencoder loss we use. This has been partly explored by Huang and Belongie [35] by replacing the style swap operation with a simpler mean and variance alignment.

The kernelized GAN is a straightforward replacement for the original GAN objective, though it is difficult to show whether discriminative power is increased by using maximum mean discrepancy as opposed to the arguably simpler Wasserstein distance for real applications. It would be helpful to generative modeling research if robust testing methods could be created specifically for generative models.

Bibliography

[1] Hussein A Aly and Eric Dubois. Image up-sampling using total-variation regularization with a new observation model. *IEEE Transactions on Image Processing*, 14(10):1647–1659, 2005.

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[3] Guillaume Berger and Roland Memisevic. Incorporating long-range consistency in cnn-based texture generation. *arXiv preprint arXiv:1606.01286*, 2016.

[4] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *arXiv preprint arXiv:1606.04838*, 2016.

[5] Alex J Champandard. Semantic style transfer and turning two-bit doodles into fine artworks. *arXiv preprint arXiv:1603.01768*, 2016.

[6] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2172–2180, 2016.

[7] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[8] Amit Daniely, Roy Frostig, and Yoram Singer. Toward deeper understanding of neural networks: The power of initialization and a dual view on expressivity. In *Advances In Neural Information Processing Systems*, pages 2253–2261, 2016.

[9] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. In *ICML*, pages 647–655, 2014.

[10] Alexey Dosovitskiy and Thomas Brox. Inverting convolutional networks with convolutional networks. *CoRR*, abs/1506.02753, 2015.

[11] Alexey Dosovitskiy, Jost Springenberg, Maxim Tatarchenko, and Thomas Brox. Learning to generate chairs, tables and cars with convolutional networks.

[12] Small Yellow Duck. Painter by numbers, wikiart.org. `https://www.kaggle.com/c/painter-by-numbers`, 2016.

[13] Vincent Dumoulin, Ishmael Belghazi, Ben Poole, Alex Lamb, Martin Arjovsky, Olivier Mastropietro, and Aaron Courville. Adversarially learned inference. *arXiv preprint arXiv:1606.00704*, 2016.

[14] Vincent Dumoulin, Jonathon Shlens, and Manjunath Kudlur. A learned representation for artistic style. *CoRR*, abs/1610.07629, 2016.

[15] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[16] Gintare Karolina Dziugaite, Daniel M Roy, and Zoubin Ghahramani. Training generative neural networks via maximum mean discrepancy optimization. *arXiv preprint arXiv:1505.03906*, 2015.

[17] 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*, pages 341–346. ACM, 2001.

[18] Alexei A Efros and Thomas K Leung. Texture synthesis by non-parametric sampling. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1033–1038. IEEE, 1999.

[19] Michael Elad and Peyman Milanfar. Style-transfer via texture-synthesis. *arXiv preprint arXiv:1609.03057*, 2016.

[20] Chelsea Finn, Ian Goodfellow, and Sergey Levine. Unsupervised learning for physical interaction through video prediction. In *Advances In Neural Information Processing Systems*, pages 64–72, 2016.

[21] Oriel Frigo, Neus Sabater, Julie Delon, and Pierre Hellier. Split and match: Example-based adaptive patch sampling for unsupervised style transfer. 2016.

[22] Leon Gatys, Alexander S Ecker, and Matthias Bethge. Texture synthesis using convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 262–270, 2015.

[23] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015.

[24] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2414–2423, 2016.

[25] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[26] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773, 2012.

[27] Arthur Gretton, Dino Sejdinovic, Heiko Strathmann, Sivaraman Balakrishnan, Massimiliano Pontil, Kenji Fukumizu, and Bharath K. Sriperumbudur. Optimal kernel choice for large-scale two-sample tests. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1205–1213. Curran Associates, Inc., 2012.

[28] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. *arXiv preprint arXiv:1704.00028*, 2017.

[29] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. Fast approximate nearest-neighbor search with k-nearest neighbor graph.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Kun He, Yan Wang, and John Hopcroft. A Powerful Generative Model Using Random Weights for the Deep Image Representation. 2016.

[32] Kun He, Yan Wang, and John E. Hopcroft. A powerful generative model using random weights for the deep image representation. *CoRR*, abs/1606.04801, 2016.

[33] Aaron Hertzmann. Paint By Relaxation. *Proceedings Computer Graphics International (CGI)*, pages 47–54, 2001.

[34] Aaron Hertzmann, Charles E Jacobs, Nuria Oliver, Brian Curless, and David H Salesin. Image analogies. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 327–340. ACM, 2001.

[35] Xun Huang and Serge Belongie. Arbitrary style transfer in real-time with adaptive instance normalization. *arXiv preprint arXiv:1703.06868*, 2017.

[36] Ferenc Huszár. How (not) to train your generative model: Scheduled sampling, likelihood, adversary? *arXiv preprint arXiv:1511.05101*, 2015.

[37] Artify Inc. Artify, 2016.

[38] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[39] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[40] Justin Johnson. neural-style. `https://github.com/jcjohnson/neural-style`, 2015.

[41] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual Losses for Real-Time Style Transfer and Super-Resolution. *Arxiv*, 2016.

[42] Taeksoo Kim, Moonsu Cha, Hyunsoo Kim, Jung Kwon Lee, and Jiwon Kim. Learning to discover cross-domain relations with generative adversarial networks. *CoRR*, abs/1703.05192, 2017.

[43] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[44] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[45] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[46] Tejas D Kulkarni, William F Whitney, Pushmeet Kohli, and Josh Tenenbaum. Deep convolutional inverse graphics network. In *Advances in Neural Information Processing Systems*, pages 2539–2547, 2015.

[47] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *ACM Transactions on Graphics (ToG)*, 24(3):795–802, 2005.

[48] Quoc Viet Le, Tamás Sarlós, and Alexander Johannes Smola. Fastfood: Approximate kernel expansions in loglinear time. *CoRR*, abs/1408.3060, 2014.

[49] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. *arXiv preprint arXiv:1609.04802*, 2016.

[50] Sylvain Lefebvre and Hugues Hoppe. Appearance-space texture synthesis. *ACM Transactions on Graphics (TOG)*, 25(3):541–548, 2006.

[51] Chuan Li and Michael Wand. Combining Markov Random Fields and Convolutional Neural Networks for Image Synthesis. *Cvpr 2016*, page 9, 2016.

[52] Chuan Li and Michael Wand. Precomputed real-time texture synthesis with markovian generative adversarial networks. In *European Conference on Computer Vision*, pages 702–716. Springer, 2016.

[53] Chun-Liang Li, Wei-Cheng Chang, Yu Cheng, Yiming Yang, and Barnabás Póczos. Mmd gan: Towards deeper understanding of moment matching network. *arXiv preprint arXiv:1705.08584*, 2017.

[54] Yanghao Li, Naiyan Wang, Jiaying Liu, and Xiaodi Hou. Demystifying neural style transfer. *CoRR*, abs/1701.01036, 2017.

[55] Yujia Li, Kevin Swersky, and Rich Zemel. Generative moment matching networks. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1718–1727, 2015.

[56] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics (ToG)*, 20(3):127–150, 2001.

[57] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[58] Peter Litwinowicz. Processing Images and Video for an Impressionist Effect. *Proc. SIGGRAPH*, pages 407–414, 1997.

[59] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[60] Jonathan L Long, Ning Zhang, and Trevor Darrell. Do convnets learn correspondence? In *Advances in Neural Information Processing Systems*, pages 1601–1609, 2014.

[61] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them. In *2015 IEEE conference on computer vision and pattern recognition (CVPR)*, pages 5188–5196. IEEE, 2015.

[62] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. In *International Conference on Artificial Neural Networks*, pages 52–59. Springer, 2011.

[63] Michael Mathieu, Camille Couprie, and Yann LeCun. Deep multi-scale video prediction beyond mean square error. *arXiv preprint arXiv:1511.05440*, 2015.

[64] Barbara J Meier. Painterly rendering for animation. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques SIGGRAPH 96*, 30(Annual Conference Series):477–484, 1996.

[65] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[66] Youssef Mroueh, Tom Sercu, and Vaibhava Goel. Mcgan: Mean and covariance feature matching gan. *arXiv preprint arXiv:1702.08398*, 2017.

[67] Krikamol Muandet, Kenji Fukumizu, Bharath Sriperumbudur, Bernhard Schölkopf, et al. Kernel mean embedding of distributions: A review and beyond. *Foundations and Trends® in Machine Learning*, 10(1-2):1–141, 2017.

[68] Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

[69] Anh Nguyen, Jason Yosinski, Yoshua Bengio, Alexey Dosovitskiy, and Jeff Clune. Plug & play generative networks: Conditional iterative generation of images in latent space. *arXiv preprint arXiv:1612.00005*, 2016.

[70] Roman Novak and Yaroslav Nikulin. Improving the neural algorithm of artistic style. *arXiv preprint arXiv:1605.04603*, 2016.

[71] Sebastian Nowozin, Botond Cseke, and Ryota Tomioka. f-gan: Training generative neural samplers using variational divergence minimization. In *Advances in Neural Information Processing Systems*, pages 271–279, 2016.

[72] Javier Portilla and Eero P Simoncelli. A parametric texture model based on joint statistics of complex wavelet coefficients. *International journal of computer vision*, 40(1):49–70, 2000.

[73] Inc. Prisma Labs. Prisma. `http://prisma-ai.com/`, 2016.

[74] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[75] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1177–1184. Curran Associates, Inc., 2008.

[76] Manuel Ruder, Alexey Dosovitskiy, and Thomas Brox. Artistic style transfer for videos. pages 1–14, 2016.

[77] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[78] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2226–2234, 2016.

[79] Thomas Schlegl, Philipp Seeböck, Sebastian M. Waldstein, Ursula Schmidt-Erfurth, and Georg Langs. Unsupervised anomaly detection with generative adversarial networks to guide marker discovery. *CoRR*, abs/1703.05921, 2017.

[80] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[81] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162(1-2):83–112, 2017.

[82] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[83] Casper Kaae Sønderby, Jose Caballero, Lucas Theis, Wenzhe Shi, and Ferenc Huszár. Amortised map inference for image super-resolution. *arXiv preprint arXiv:1610.04490*, 2016.

[84] Bharath K Sriperumbudur, Arthur Gretton, Kenji Fukumizu, Bernhard Schölkopf, and Gert RG Lanckriet. Hilbert space embeddings and metrics on probability measures. *Journal of Machine Learning Research*, 11(Apr):1517–1561, 2010.

[85] PicsArt Photo Studio. Picsart. `https://picsart.com/`, 2016.

[86] Dougal J Sutherland, Hsiao-Yu Tung, Heiko Strathmann, Soumyajit De, Aaditya Ramdas, Alex Smola, and Arthur Gretton. Generative models and model criticism via optimized maximum mean discrepancy. *arXiv preprint arXiv:1611.04488*, 2016.

[87] Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor Lempitsky. Texture Networks: Feed-forward Synthesis of Textures and Stylized Images. *CoRR*, 2016.

[88] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. Instance normalization: The missing ingredient for fast stylization. *CoRR*, abs/1607.08022, 2016.

[89] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 479–488. ACM Press/Addison-Wesley Publishing Co., 2000.

[90] Christopher K. I. Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 682–688. MIT Press, 2001.

[91] Jiajun Wu, Chengkai Zhang, Tianfan Xue, Bill Freeman, and Josh Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *Advances in Neural Information Processing Systems*, pages 82–90, 2016.

[92] Pinar Yanardag, Manuel Cebrian, Nick Obradovich, and Iyad Rahwan. Nightmare machine, 2016.

[93] Raymond Yeh, Chen Chen, Teck-Yian Lim, Mark Hasegawa-Johnson, and Minh N. Do. Semantic image inpainting with perceptual and contextual losses. *CoRR*, abs/1607.07539, 2016.

[94] Wojciech Zaremba, Arthur Gretton, and Matthew B. Blaschko. B-test: A non-parametric, low variance kernel two-sample test. *CoRR*, abs/1307.1954, 2013.

[95] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. Deconvolutional networks. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2528–2535. IEEE, 2010.

[96] Fang Zhao, Jiashi Feng, Jian Zhao, Wenhan Yang, and Shuicheng Yan. Robust lstm-autoencoders for face de-occlusion in the wild. *CoRR*, abs/1612.08534, 2016.

[97] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networkss. *arXiv preprint arXiv:1703.10593*, 2017.

Appendices

Appendix A

Linear time and space complexity kernel sums

It is often believed that the computation of MMD is prohibitive as the kernel matrix requires quadratic time and space complexity with respect to the number of samples. However, this isn't always the case when the explicit kernel matrix isn't required. In maximum mean discrepancy, only the sum of all elements of the kernel matrix is required. We describe a few simple tricks to allow computation in linear time and space complexity. With the use of parallel computation on graphics processing units (GPUs), linear time may not affect the computation time much, but the linear space complexity allows much higher minibatch sizes due to the restricted memory sizes on current GPUs.

Let X and Y denote two $n \times d$ datasets in $\mathbb{R}^d$. For simplicity, we've assumed the two datasets have the same number of samples. However, this is only required for the polynomial kernel we derive below.

Firstly, the linear kernel sum can be easily computed in $\mathcal{O}(nd)$ in both space and time complexity. This is done by replacing summation with multiplication by a vector of ones. Let e denote a vector of ones of length n. Then

$$\sum_{i,j}^{n} x_i^T y_j = e^T X Y^T e = \underbrace{(e^T X)}_{1\times d}\underbrace{(e^T Y)}_{d\times 1} \tag{A.1}$$

Note $e^T X$ can also be computed as $\sum_i x_i$ and can be performed in linear time.

The polynomial kernel of degree 2 with bias b:

$$\begin{aligned}\sum_{i,j}^{n}(x_i^T y_j + b)^2 &= ||XY^T + bI||_F^2 \\ &= \text{trace}\left[(XY^T + bI)^T(XY^T + bI)\right] \\ &= \text{trace}\left[(XY^T)^T(XY^T) + 2bXY^T + b^2 I)\right] \\ &= \text{trace}\left[(XY^T)^T(XY^T)\right] + \text{trace}\left[2bXY^T\right] + \text{trace}\left[b^2 I\right] \\ &= \text{trace}\left[(XY^T)(XY^T)^T\right] + \text{trace}\left[2bX^T Y\right] + \text{trace}\left[b^2 I\right] \\ &= ||X^T Y||_F^2 + 2b\sum_{j}^{d}\sum_{i}^{n} x_{ij}y_{ij} + nb^2\end{aligned} \tag{A.2}$$

This changes the $\mathcal{O}(N^2 D)$ operation to a $\mathcal{O}(ND^2)$ operation in both time and space complexity. For the purposes of our method, D can be made very small due to the use of a parametric neural net. (As opposed to computing this directly on images, where the dimension of the image is $3HW$. For even medium sized images, the D^2 cost can be intractible.)

For shift-invariant kernels such as the Gaussian RBF, Random Kitchen Sinks (RKS) or the Fastfood method can be used to construct approximate feature maps $\hat{\phi}(x_i)^T\hat{\phi}(y_j) \approx k(x_i, y_i)$. For other kernels, the Nystroem method can be used, though it's unknown how well it'll perform in an adversarial setting. Since these approximate using a linear kernel, the space and time complexity are the same as linear kernels.

Appendix B

Inverse network architecture

The architecture of the truncated VGG-19 network used in the experiments is shown in Table B.1, and the inverse network architecture is shown in Table B.2. It is possible that better architectures achieve better results, as we did not try many different types of convolutional neural network architectures.

- Convolutional layers use filter sizes of 3×3, padding of 1, and stride of 1.
- The rectified linear unit (ReLU) layer is an elementwise function $\text{ReLU}(x) = \max\{x, 0\}$.
- The instance norm (IN) layer standardizes each feature channel independently to have 0 mean and a standard deviation of 1. This layer has shown impressive performance in image generation networks [88].
- Maxpooling layers downsample by a factor of 2 by using filter sizes of 2×2 and stride of 2.
- Nearest neighbor (NN) upsampling layers upsample by a factor of 2 by using filter sizes of 2×2 and stride of 2.

Layer Type	Activation Dimensions
Input	$H \times W \times 3$
Conv-ReLU	$H \times W \times 64$
Conv-ReLU	$H \times W \times 64$
MaxPooling	$1/2H \times 1/2W \times 64$
Conv-ReLU	$1/2H \times 1/2W \times 128$
Conv-ReLU	$1/2H \times 1/2W \times 128$
MaxPooling	$1/4H \times 1/4W \times 128$
Conv-ReLU	$1/4H \times 1/4W \times 256$

Table B.1: Truncated VGG-19 network from the input layer to "relu3_1" (last layer in the table).

Layer Type	Activation Dimensions
Input	$1/4H \times 1/4W \times 256$
Conv-IN-ReLU	$1/4H \times 1/4W \times 128$
NN-Upsampling	$1/2H \times 1/2W \times 128$
Conv-IN-ReLU	$1/2H \times 1/2W \times 128$
Conv-IN-ReLU	$1/2H \times 1/2W \times 64$
NN-Upsampling	$H \times W \times 64$
Conv-IN-ReLU	$H \times W \times 64$
Conv	$H \times W \times 3$

Table B.2: Inverse network architecture used for inverting activations from the truncated VGG-19 network.