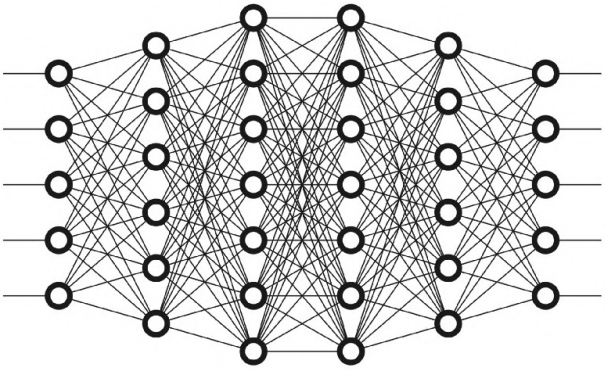
TÉCNICO LISBOA



Addressing the Texture-Bias and Domain Generalization Performance of Convolutional Neural Networks for Simulation-Based Learning

Pedro Miguel de Aguiar Coelho

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisors: Prof. Maria da Conceição Esperança Amado
Prof. Fernando José Parracho Lau

Examination Committee

Chairperson: Prof. Paulo Jorge Coelho Ramalho Oliveira
Supervisor: Prof. Fernando José Parracho Lau
Member of the Committee: Prof. Alexandre José Malheiro Bernardino

January 2021

À minha família

Acknowledgments

Antes de tudo, um enorme agradecimento ao Instituto Superior Técnico, não só por me ter dado uma formação de mais alto nível nos meus primeiros três anos de Universidade, mas também por me permitir fazer um Duplo Diploma no estrangeiro, algo que me abriu ainda mais os horizontes. Gostaria de agradecer também ao Professor Fernando Lau por coordenar um dos cursos mais reputados do país e por ainda encontrar tempo para acompanhar individualmente os seus alunos, incluindo supervisionar as teses daqueles que, como eu, partem ao estrangeiro. Um grande obrigado também à Doutora Conceição Amado por aceitar supervisionar a minha tese e por me acompanhar ao longo do semestre.

Pareillement, je remercie l'ISAE-Supaero, qui m'a enseigné à regarder le monde comme un vrai ingénieur et qui m'a ouvert plusieurs portes pour l'avenir. Je voudrais remercier aussi l'équipe DEEL de l'IRT Saint-Exupéry, dans laquelle je me suis inséré et dont ses membres m'ont présenté des approches au deep learning robuste et explicable que je ne connaissais pas encore. En particulier, je remercie mon tuteur David Vigouroux, qui m'a accompagné très prochement pendant toute la durée de mon stage. David m'a toujours motivé à aller plus loin, à explorer toutes les possibilités et à travailler de façon rigoureuse. Nos discussions ont toujours été très enrichissantes et la source de plusieurs nouvelles idées.

Agradeço profundamente à minha família, sem a qual não seria a pessoa motivada e curiosa que sou hoje. Agradeço por me suportarem toda a minha vida, por me darem as melhores condições para crescer, por me introduzirem à matemática e à música, que desde pequeno e até hoje me fascinam, e por serem o grande pilar no qual a minha vida foi construída.

I'd like to thank all my friends, without whom my life would not be as interesting and enjoyable - those whom I love spending all my time and having fun with, those whom I have incredibly interesting conversations with, those whom I play music with, and those whom I share the stage with. Lastly, a huge thank you to everyone who supported and motivated me all throughout my university years.

Resumo

As Redes Convolucionais tornaram-se a solução mais utilizada para resolver problemas de Visão por Computador. Graças a avanços em desempenho computacional e ao desenvolvimento de bibliotecas abertas de grande qualidade para aprendizagem profunda, automatizar uma tarefa de visão simples envolve apenas construir um conjunto de dados representativo, que simula as condições nas quais a rede será aplicada, e treinar uma rede adequada nesse conjunto de dados, com algumas considerações adicionais para evitar sobre-adaptar ou sub-adaptar a rede ao conjunto de dados.

Este problema torna-se mais complexo quando o conjunto de dados de treino não representa inteiramente o ambiente no qual a rede deverá funcionar. Por exemplo, uma rede que é treinada num ambiente simulado poderá não funcionar bem quando é testada num ambiente real. Esta situação requer redes que sejam capazes de generalizar de forma mais profunda. As redes deverão ser robustas a alterações no seu ambiente. No caso de uma tarefa de visão, a rede deverá reconhecer a forma dos objetos em vez de reconhecer apenas as suas texturas, para poder reconhecer objetos em diferentes ambientes. Nesta tese iremos explorar o funcionamento das Redes Convolucionais, os detalhes deste problema de generalização, diferentes estratégias para o abordar, e propomos um novo método de texturização aleatória de imagens, que reduz a dependência das redes em texturas e aumenta a sua sensibilidade à forma dos objetos.

Palavras-chave: Visão por Computador, Aprendizagem Profunda, Redes Convolucionais, Generalização de Domínio, Viés de Texturas

Abstract

Convolutional neural networks have become the de-facto standard solution for most computer vision problems. Thanks to advances in computer performance and the development of open high-end deep learning libraries, automating a simple vision task only involves gathering a reasonable dataset, that mimics the conditions in which the network will be deployed, and training a suitable network on that dataset, with some extra considerations and tuning to avoid underfitting or overfitting the dataset.

This problem becomes more difficult when the dataset on which the network is trained on does not fully represent the scenarios on which the network will be immersed. For example, a network that is trained in a simulated environment may not perform well when it is tested in a real environment, due to the differences between the simulated and real environments. This situation requires networks that are able to generalize at a deeper level. The networks must be robust to changes in their environment. Therefore, in the case of a vision task, they must rely mostly on high-level object shapes rather than low-level image textures to correctly identify objects across environments. Throughout this thesis we will explore the inner workings of convolutional neural networks, the intricacies of this generalization problem, several strategies to tackle it and propose a novel randomized image texturization method that can make make networks rely less on texture and more on the shape of objects.

Keywords: Computer Vision, Deep Learning, Convolutional Neural Networks, Domain Generalization, Texture Bias

Contents

List of Tables

List of Figures

Nomenclature

Greek symbols

α	Learning rate. / Texturization Strength.
β	Batch normalization coefficients.
$\boldsymbol{\theta}, \theta$	Network parameters.
γ	Batch normalization coefficients. / Decay factor.
$\hat{\phi}$	Normalized feature map.
λ	Loss weight factor.
μ	Mean.
Ω	Set of all image pixels. / Domain of variable.
ϕ	Feature map.
σ	Standard deviation.

Roman symbols

$\mathbf{o}$	Stylized image.
$\mathbf{s}$	Style image.
$\hat{x}$	Generated image.
$\hat{y}, \hat{\mathbf{y}}$	Network predictions. / Transformed latent representation.
$\mathbb{E}$	Expectation operator.
$\mathbb{R}$	Real numbers.
$\mathcal{B}$	N-dimensional ball.
$\mathcal{C}$	Confusion Matrix / Tensor.
$\mathcal{D}$	Dataset.
$\mathcal{L}$	Loss function.

$\mathcal{N}$	Normal distribution.
$\mathcal{S}$	Source dataset.
$\mathcal{T}$	Target dataset.
$\sim$	Distributed as...
b	Dense later bias parameters.
$Batch$	Set of input indexes of a batch.
D	Dense layer function. / Decoder function
d	Discriminator function.
f	Feature extractor function.
FN	Number of false negatives.
FP	Number of false positives.
G	Generator function. / Gram Matrix.
g	Multi-layer perceptron classifier function.
h	Hypothesis class.
H, W	Height and width.
h, w	Height and width indices.
K	Number of classes.
k	Kernel size.
L	Number of layers.
N	Number of...
P	Set of predicted pixels of a class.
p	Probability of...
$Pool$	Max-Pooling function.
$ReLU$	Rectified Linear Unit activation function.
T	Set of true pixels of a class.
T	Transformation function.
t	Pixel-wise transformation.
TP	Number of true positives.

U Uniform distribution.

v Feature vector.

Var Variance.

W Dense layer weight matrix parameters.

X Dataset input distribution.

$x, \mathbf{x}$ Function input or dataset inputs.

Y Dataset label distribution.

y Dataset labels. / Latent representation.

Z Noise distribution.

z Noise vector or matrix.

K Kernel matrix parameters.

B Kernel bias parameters. / Set of pixels in a box.

C Convolutional layer function.

Subscripts

θ, ϕ, ψ Network parameters.

c, c' Channel index.

i General vector index. / Content class.

i, j Pixel indices.

j General vector index. / Style class.

k, k' Neuron index. / Class index. / Predicted class.

m, n Pixel shift indices.

t, p True Label / Predicted Label.

Superscripts

c, c' Number of channels.

l Layer.

n Number of noise components.

s, s' Number of pixels.

x, y Variables.

Chapter 1

Introduction

The usage of machine learning models in the critical systems of vehicles and aircraft has long been desired to automate their functions. Good examples include the increased interest in autonomous driving [1] and the recent demonstration of full autonomous flights by an Airbus commercial aircraft [2]. One of the key steps to the deployment of a good computer vision solution, besides the choice of a trustworthy algorithm, is the construction of a large and representative dataset on which the algorithms can be successfully trained. This involves the manual annotation of thousands of images which can take more than one hour each, depending on the task, to achieve good results.

One possible solution to this problem would be the training of such algorithms in a simulated environment, like a driving simulator or a video game, where different objects in a scene can be automatically annotated, thus saving thousands of hours of human labor and allowing the creation of diversified training scenarios. In the case of autonomous driving, the points of failure are usually situations that are very rare and don't appear often enough in the training dataset, like overturned vehicles, pedestrians in unusual locations, vehicles transporting other vehicles and occluded signs or lines. Training on a simulator would allow the algorithm to learn in many different and rare scenarios that it would not observe often enough on real training images, thus guaranteeing more control over those situations. Furthermore, reinforcement learning algorithms typically perform better when the state vector between the perception and planning modules is learned rather than human designed, meaning that we should let the algorithm decide which visual features it considers more important to the task that is being automated. This may only be possible to achieve if the algorithms are trained in a simulator.

Although the Convolutional Neural Network (CNN) architecture offers outstanding performance in image processing tasks, it does so only when the images on which it is tested on come from the same domain as the ones that were used to train the algorithm. This is the problem of out-of-domain (OOD) performance degradation. A domain shift may include simple transformations like a colour shift, noisy images, camera artifacts, camera position, image scale, or more complex differences like the fact that an image of an object can be a photograph, a drawing, a painting or the result of a render from a simulation. Objects in different domains can also have different textures, lighting and light reflection/diffusion properties. Any of these domain shifts can catastrophically affect the performance of CNNs if it is not

accounted for. Accounting for domain shifts is not always simple and the problem of domain adaptation/generalization remains an open problem with no algorithm being capable of achieving human-level generalization to new unseen domains and with any unknown domain shift.

Despite the fact that computer graphics have tremendously evolved over the year, there will always be a domain gap between a simulated environment and the real images captured by cameras. Style transfer techniques have tried to close this gap by transforming the simulated images such that they look as similar to the real world as possible but artifacts still exist, which can interfere with the task performance. Furthermore, real world images themselves can exist in multiple domains, such as different times of the day and lighting conditions, different weather and different camera defects. Training a computer vision algorithm in a simulator would have to yield good performance for any possible domain, thus having good generalization properties. The question then becomes - How to train a CNN on a given domain in a way that it can generalize to any other domain?

1.1 Motivation

The usage of machine learning models in the critical systems of vehicles and aircraft has long been desired to automate their functions. Good examples include the increased interest in autonomous driving [1] and the recent demonstration of full autonomous flights by an Airbus commercial aircraft [2]. One of the key steps to the deployment of a good computer vision solution, besides the choice of a trustworthy algorithm, is the construction of a large and representative dataset on which the algorithms can be successfully trained. This involves the manual annotation of thousands of images which can take more than one hour each, depending on the task, to achieve good results.

One possible solution to this problem would be the training of such algorithms in a simulated environment, like a driving simulator or a video game, where different objects in a scene can be automatically annotated, thus saving thousands of hours of human labor and allowing the creation of diversified training scenarios. In the case of autonomous driving, the points of failure are usually situations that are very rare and don't appear often enough in the training dataset, like overturned vehicles, pedestrians in unusual locations, vehicles transporting other vehicles and occluded signs or lines. Training on a simulator would allow the algorithm to learn in many different and rare scenarios that it would not observe often enough on real training images, thus guaranteeing more control over those situations. Furthermore, reinforcement learning algorithms typically perform better when the state vector between the perception and planning modules is learned rather than human designed, meaning that we should let the algorithm decide which visual features it considers more important to the task that is being automated. This may only be possible to achieve if the algorithms are trained in a simulator.

Although the Convolutional Neural Network (CNN) architecture offers outstanding performance in image processing tasks, it does so only when the images on which it is tested on come from the same domain as the ones that were used to train the algorithm. This is the problem of out-of-domain (OOD) performance degradation. A domain shift may include simple transformations like a colour shift, noisy images, camera artifacts, camera position, image scale, or more complex differences like the fact that

an image of an object can be a photograph, a drawing, a painting or the result of a render from a simulation. Objects in different domains can also have different textures, lighting and light reflection/diffusion properties. Any of these domain shifts can catastrophically affect the performance of CNNs if it is not accounted for. Accounting for domain shifts is not always simple and the problem of domain adaptation/generalization remains an open problem with no algorithm being capable of achieving human-level generalization to new unseen domains and with any unknown domain shift.

Despite the fact that computer graphics have tremendously evolved over the year, there will always be a domain gap between a simulated environment and the real images captured by cameras. Style transfer techniques have tried to close this gap by transforming the simulated images such that they look as similar to the real world as possible but artifacts still exist, which can interfere with the task performance. Furthermore, real world images themselves can exist in multiple domains, such as different times of the day and lighting conditions, different weather and different camera defects. Training a computer vision algorithm in a simulator would have to yield good performance for any possible domain, thus having good generalization properties. The question then becomes - How to train a CNN on a given domain in a way that it can generalize to any other domain?

1.2 Topic Overview

The previously posed question is the gist of the Domain Generalization (DG) problem. This line of research deals with the problem of training machine learning (ML) algorithms that can work in novel, unseen domains. This problem seems trivial to human beings because we evolved to be able to do this subconsciously, filtering out irrelevant information with very complex adapting recurrent visual systems, using feed-forward, horizontal and feed-back connections in the visual cortex [3]. On the contrary, ML algorithms are usually simple feed-forward algorithms that learn or fit a particular distribution of data and their performance can be very sensible to shifts in this distribution.

The Domain Generalization problem is part of a broader set of generalization problems that also includes Domain Adaptation (DA). While DG tries to generalize to new unseen domains, DA deals with a problem of learning a task on particular distribution of data and while adapting that knowledge to a second accessible distribution, where typically the second distribution does not have supervision. DA algorithms can adapt/generalize using multiple distributions, therefore a considerable number of DG algorithms are imported directly from DA research. Therefore, to establish a strong base on the generalization problem, different DA and DG algorithms will be covered with appropriate detail in section 2 of this thesis.

The problem of domain shifts is usually tackled in the context of image processing, where distributions over the space of all possible images can be subject to a huge variety of transformations and domain shifts that occur naturally or are the results of different training and testing environments. In the context of CNNs, this failure to generalize is explained by their often strong texture-bias. When CNNs heavily rely on textural cues, instead of object shapes, to perform their tasks, they will fail to generalize to different domains where the low-level texture statistics might be different. The origins and possible solutions to

the texture-bias problem of CNNs will be covered in section 2.

To understand how CNNs capture and process these image distributions, this thesis will explore the most basic mechanisms that make up CNNs and how they are used together. Furthermore, different CNN types and architectures will be covered such that a strong theoretical basis can be established before tacking the DG, DA and texture-bias problems. In parallel, the different applications of CNNs to real-world tasks, such as image classification, object detection and image segmentation, will be exposed in a comprehensive manner. The generalization algorithms are commonly tested in these tasks, with multiple datasets from different domains, to assess how well the networks are able to generalize.

1.3 Objectives

This thesis has two main objectives:

- To offer the reader an introduction to the topic of computer vision using CNNs. The possible applications, the different architectures and the challenges that it faces in terms of generalization performance.
- To showcase different attempts to randomly alter the low-level statistics of images and finally propose a novel randomized texturization method that decreases the texture-bias of CNNs with only a small computational overhead compared to other methods.

1.4 Thesis Outline

We start by outlining the Background section, based mostly on bibliographic research:

In section 2.2 we will start by understanding exactly what a Convolutional Neural Network (CNN) is, its building blocks, what mathematical operations it performs, how it can process images and how it is trained, in the context of image classification.

Section 2.3 will cover the most common and representative CNN architectures, VGGs and ResNets, that are widely used in research because they are the basis of most subsequent architectures.

Section 2.4 will explore the many possible applications of CNNs, from Semantic Segmentation and Object Detection to Image Generation and Style Transferring techniques.

Section 2.5 will introduce the phenomenon of texture-bias, which is present across different CNN architectures and may be seen as a form of domain overfitting.

Section 2.6 will offer an extensive overview of the research line that deals with the generalization performance of CNNs. Many algorithms will be showcased, from the most commonly used to the most recent (some of them from late 2020).

The following sections include mostly original work, which is based on previous techniques but seeks to improve them:

In section 3.1 we explore a first approach to randomized image transformations based on style transferring techniques. The methods are extended to be more general and to learn without a style dataset.

In section 3.2 we propose and formalize and novel randomized texturing method based on an auto-encoder architecture. This method transforms the images in their latent space, producing spatially coherent textures. It is shown that, with certain latent transformations, this method can produce all possible spatial coherent textural variations of an image in pixel space. Further, this method is far less computationally intensive than style transferring methods.

Chapter 4 covers different experiments that were performed with the novel technique:

In section 4.1 we measure the effect of randomly texturing images during training on the texture-bias of a CNN architecture during testing.

In section 4.2 we assess the generalization performance in a semantic segmentation task when using randomly textured images during training.

In section 4.3 we evaluate the generalization performance in an object detection task when using randomly textured images during training.

Finally, chapter 5 concludes this thesis and exposes future lines of research that might better solve this problem.

1.5 Thesis Context

This thesis was the final result of my end-of-studies internship at the Institute de Recherche Technologique (IRT) Saint Exupéry in Toulouse, France. This institute is part of the Aerospace Valley and forms strong partnerships with industry leaders such as Airbus, Thales, Safran, Zodiac, Ariane Group, Collins Aerospace, Liebherr, Stelia, Latécoère, Daher, and many others. The institute is funded 50% by the state and 50% by the private sector and its main purpose is to mature new technologies in the aerospace sector into industry-ready maturity levels. In this institute, partners from academia and industry work together to develop solutions from the research phase up to the industry-ready implementation, thus accelerating the adoption of new technologies by the industry.

I did my internship with the DEEL (DEpendable and Explainable Learning) team at the IRT Saint Exupéry. This team is comprised of members of academia with a strong mathematical background in statistics, representation learning and computer science, as well as members from the industry, with expertise in machine learning and artificial intelligence. Together the team researches novel methods for explainable artificial intelligence (XAI), dependable and certifiable algorithms, including formal methods of certification, the improvement of AI robustness, the development of fair AI, the learning of disentangled representations, the applications of binary networks and k-lipschitz networks, and other more specific applications. I worked within the context of improving the robustness of AI vision algorithms, in particular the robustness when generalizing across domains.

Chapter 2

Background

The backbone of many modern computer vision and image processing algorithms are Convolutional Neural Networks. This family of architectures mimics the visual system of animals [4] and humans [5–8], where low-level features like points, edges and textures are combined to form mid-level features like shapes, curves and more complex textures. These mid-level features are then combined to form high-level features like faces, bodies, animals, objects, locations and backgrounds (see Figure 2.1). Finally the high level features are used to perform various tasks such as object classification, detection or image segmentation.

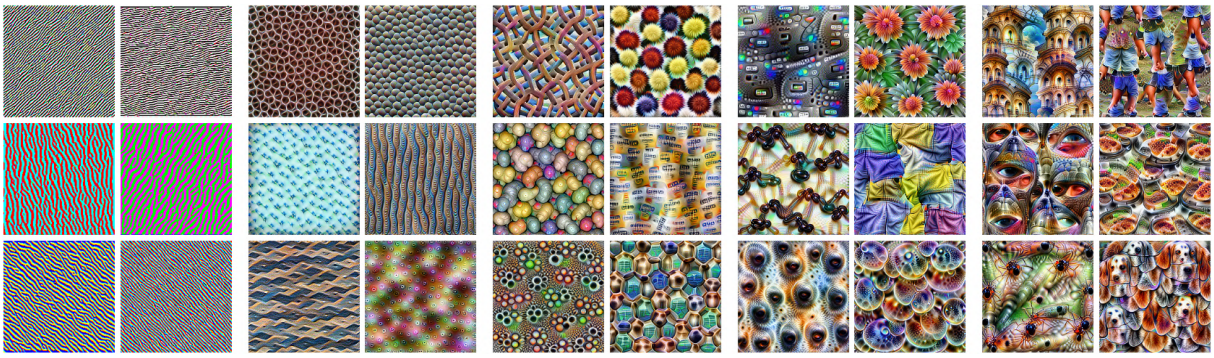


Figure 2.1: Feature Hierarchy leaned by GoogLeNet on the ImageNet dataset.

Gradient based CNNs were first applied to image processing by LeCun et al. [9] in 1989 for digit recognition and considerably outperformed any previous technique. They were later used for tasks such as document recognition in 1998 [10].

It wasn't until the increase in computational power, its parallelization using Graphical Processing Units (GPUs) and the construction of large labeled image datasets such as ImageNet [11], that a breakthrough occurred in natural image processing. In 2012 Krizhevsky et al. [12] introduced AlexNet, a deep CNN capable of classifying images into 1000 different classes with a high degree of accuracy. Alexnet was trained by leveraging the parallel processing power of Graphical Processing Units to achieve a great increase in model capacity.

From 2012 onwards, novel architectures established new state-of-the-art performances every year, such as deeper VGG (Visual Geometry Group) networks [13], parallel reduction and multi-scale convo-

lutions in inception networks [14], residual networks [15], which surpassed human-level performance, densely connected convolutional networks [16] and squeeze networks with channel-wise re-calibration [17]. The focus slowly shifted from achieving the best performance to achieving a good compromise between performance and a small model size, with less parameters, allowing CNNs to be deployed in mobile devices. With this objective, different architectures were introduced, using depth-wise separable convolutions [18], network pruning/slimming [19] and optimal scaling of network depth, width and resolution [20].

Besides image classification, convolutional architectures also found application in algorithms for object detection [21–24] and semantic segmentation [25–29], in (Variational) Auto-Encoders [30–33] and in Generative Adversarial Networks [34–38]. These algorithms can then be used in many types of downstream tasks such as self-driving [39, 40], robotics [41, 42], visual tracking [43, 44], image captioning [45, 46], image super-resolution [47], image colorization [48], image style transfer [49] and deep dreaming [50].

In this section we shall explore Convolutional Neural Networks, mainly focusing in their internal functioning and generalization properties.

2.1 Multi-Layer Perceptron

Before tackling Convolutional Neural Networks we shall first explore the Perceptron and the Multi Layer Perceptron, which are the basis of most neural network architectures, including CNNs.

A Perceptron (commonly called Neuron) is the most basic unit of a neural network. It receives several inputs and multiplies each of them by a weight. The resulting values are added together, alongside an extra bias term, and passed through an activation function, as seen in Figure 2.2. A single Perceptron can perform linear regression when a linear activation is used and can perform logistic regression when using a Sigmoid activation function.

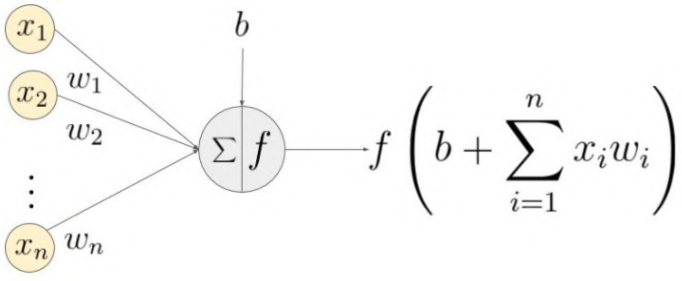


Figure 2.2: Schematic of a single Perceptron.

Multiple Perceptrons with the same inputs can be combined to form a fully connected layer, also called a dense layer. Using a Rectified Linear Unit (ReLu) as an activation function, the operation performed by a dense layer D^l, taking as input the activation vector from the previous layer v_k^{l-1} and producing a new activation vector $v_{k'}^l$, can be described as:

$$v_{k'}^{l} = D^{l}(v_{k}^{l-1}) = ReLU\left(\sum_{k=1}^{N_k} W_{k'k}^{l} \cdot v_{k}^{l-1} + b_{k'}^{l}\right) \tag{2.1}$$

where N_k is the number of elements of the input vector, k and k' are the indexes of input and output vector respectively, $W_{k'k}$ is the weight matrix that linearly transforms the input and $b_{k'}^{l}$ are the bias terms. Both $W_{k'k}$ and $b_{k'}^{l}$ are learneable parameters.

The Rectified Linear Unit (ReLU) activation function is usually preferred over the more classical Sigmoid function for reasons that will be explained later. It is defined as:

$$ReLU(x) = \begin{cases} 0, & x \leq 0 \\ x, & x > 0 \end{cases} = max(0, x) \tag{2.2}$$

A Multi Layer Perceptron (Figure 2.3) is created by composing multiple of these layers in sequence:

$$\hat{y}_{k'} = g_{\boldsymbol{\theta}}(x_k) = (D^{N_l} \circ D^{N_l - 1} \circ ... \circ D^{2} \circ D^{1})(x_k) \tag{2.3}$$

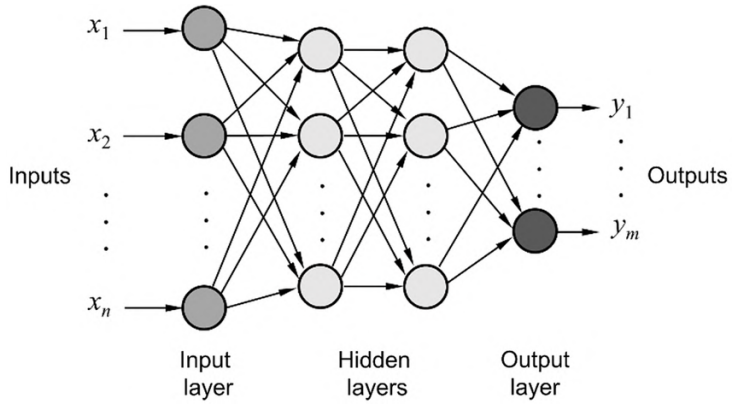


Figure 2.3: Schematic of a Multi Layer Perceptron.

where N_l is the number of dense layers, $g_{\boldsymbol{\theta}}$ is the function composition of all dense layers and x_k is an input vector. All the weights and biases of all the dense layers - $W_{k'k}^{l}$ and $b_{k'}^{l}$ - are parameters that will be adjusted using a learning algorithm such that the function $g_{\boldsymbol{\theta}}$ fits a particular dataset. We use the symbol $\boldsymbol{\theta}$ to represent all the parameters of the network that are learnable.

In a regression task, $g_{\boldsymbol{\theta}}$ will be learned such that it approaches as closely as possible a set of points from a training dataset. A training dataset is defined as a set of N_t tuples $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N_t}$, where each input $\mathbf{x}_i$ is a realization of a random variable or vector X, and each output $\mathbf{y}_i$ is a realization of a random variable or vector Y. The network is trained to produce the outputs when given the respective inputs in the dataset.

In a classification task, the dataset contains a single label y_i corresponding to a discrete class of the respective input $\mathbf{x}_i$. During training, $g_{\boldsymbol{\theta}}$ will be learned such that, given an input $\mathbf{x}_i$, it produces vector $\hat{\mathbf{y}}_i$ with the likelihood of $\mathbf{x}_i$ belonging to each of the possible classes. The objective is to predict a high likelihood for the true class y_i and, by construction, a low likelihood for the incorrect classes. Since we want to estimate the likelihoods of every class as the result of the last layer, we want the sum of

the components of $\hat{\mathbf{y}}$ to equal 1. To accomplish this, the ReLU function of the last dense layer $D^{N_{l'}}$ is replaced by the Softmax function. This function transforms a feature vector $\mathbf{v} \in \mathbb{R}^{N_k}$ into a vector of likelihoods $\hat{\mathbf{y}} \in [0,1]^{N_k}$ where $\sum_j \hat{y}_j = 1$:

$$\hat{y}_j = Softmax(\mathbf{v})_j = \frac{e^{x_j}}{\sum_k e^{v_k}} \tag{2.4}$$

Before the parameters $\boldsymbol{\theta}$ can be learned, we need to introduce a loss function which will serve as an objective function that the learning algorithm will seek to minimize.

In a regression task this loss is usually an L^2 distance loss between the output of the network and the true value in the dataset. Minimizing this loss in a single linear Perceptron produces the same results as a linear regression with least squares.

In a classification task, the loss function $\mathcal{L}$ is typically a cross-entropy loss given by:

$$\mathcal{L}(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},y\sim X,Y} - log(\hat{y}_k)\Big|_{k=y} = \mathbb{E}_{\mathbf{x},y\sim X,Y} - log(g_{\boldsymbol{\theta}}(\mathbf{x})_k)\Big|_{k=y} \tag{2.5}$$

This cross-entropy has been simplified since the true class label corresponds to a true likelihood of 1 and the other possible classes correspond to a true likelihood of 0, canceling out the predicted likelihoods of the incorrect classes and only taking into account the likelihood of the true class $k = y$ predicted by the network. Therefore, the loss function evaluates to 0 when the network assigns a likelihood of 1 to the correct class and, by construction, 0 likelihood to the wrong classes. This corresponds to the minimal loss. When the network produces likelihood predictions of less than 1 to the correct class, the loss is positive and tends to infinity as the likelihood of the correct class tends to 0.

Before the training phase, the learnable parameters $\boldsymbol{\theta}$ are initialized randomly. This means that the predictions generated by the network are completely random and far from correct. The initial loss is therefore a positive value that we will seek to minimize through training. In practice, the expectation in the loss function is estimated by an average of the negative log-likelihoods across a batch of different dataset samples, hence the stochastic nature of the gradient descent algorithm:

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i\in Batch} -log(g_{\boldsymbol{\theta}}(\mathbf{x}_i)_k)\Big|_{k=y_i} \tag{2.6}$$

The process of training a neural network is comprised of three operations that are repeated through many iterations until the network performs well. The first operation is forward propagation, where the inputs are propagated through the network and the final loss is calculated. All neuron activations are saved so that the next step, back-propagation, can occur. This second step involves using the chain rule of derivation to calculate the gradient of the loss function with respect to every single learnable parameter. The final operation consists of performing an update of the weights in the opposite direction of the gradient:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{2.7}$$

where α is the learning rate, a hyper-parameter that dictates the magnitude of the step. This parameter

plays a major role in the stability and convergence properties of gradient descent algorithm. We just described the simplest version of the Stochastic Gradient Descent (SGD) algorithm. More commonly, this algorithm is used with an additional momentum term, which stabilizes the descent using an exponentially weighted average of the previous gradient steps. Other algorithms such as ADAM (adaptive moment estimation) [51], uses a different gradient normalization technique based on an adaptive moment term. ADAM is shown to converge faster than SGD but sometimes at the cost of finding worst solutions. Nevertheless, ADAM was developed with deep learning and convolutional neural networks in mind and is widely used to train CNNs.

2.2 Convolutional Neural Networks

Convolutional Neural Networks are deep neural networks that apply at least one convolution operation. When used for image processing, two dimensional convolutions are employed to take advantage of the spatial relationship between the pixels of an image. The relationship between a pixel and its closest neighbours is far more important than its relationship with the pixels on the other side of the image. Therefore it is far more efficient to compute inter-pixel relationships using kernel convolutions. In the context of a multi-layer perceptron, this is equivalent to using shared weights between close-by pixels and setting the weights between far-away pixels to zero. A drastic reduction in the number of parameters of the network is therefore achieved. The weights in the convolution kernel matrices are learned with back-propagation to minimize a certain training loss. [52]

2.2.1 Convolution Operation

A convolution operation can be described as sliding a kernel matrix over a larger input matrix. The values of the input matrix are multiplied element-wise by the weights in the kernel matrix and finally added together to produce a value in the resulting output matrix. Each element in the resulting matrix corresponds to one position of the kernel over the input matrix, as seen in Figure 2.4. The size of the kernel f defines the dimensions of the kernel matrix. The stride s defines by how much the kernel slides in each step and is usually set to 1 or 2. The padding is the process of adding values around the input matrix to allow the kernel to slide further out than normal. Usually a reflective padding of $p = 1$ is used with a 3x3 kernel so that the input and output matrices have the same dimensions. A padding of $p = 0$ results in an output matrix smaller than the input matrix.

To see how this operation might extract features like edges from an image consider Figure 2.5. A carefully chosen combination of kernel weights, when convolved with an input matrix consisting of a transition from light to dark, produces an output matrix where the transition is highlighted, thus encoding the presence of an edge.

The weights of the kernels used to be chosen manually in early computer vision algorithms. In image classification CNNs the weights in the kernels are learned using a stochastic gradient descent algorithm that minimizes a classification loss function at the end of the network. The gradient of the loss

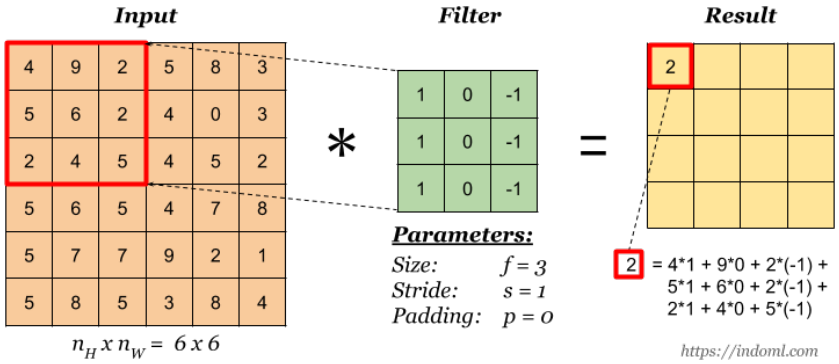


Figure 2.4: Convolution Operation.

function with respect to each learnable weight is calculated using back-propagation, also called auto-differentiation in other scientific contexts. The weights are adjusted so that the expected classification loss decreases. The network therefore learns kernel weights that can extract useful features to classify the types of objects on which the network is trained.

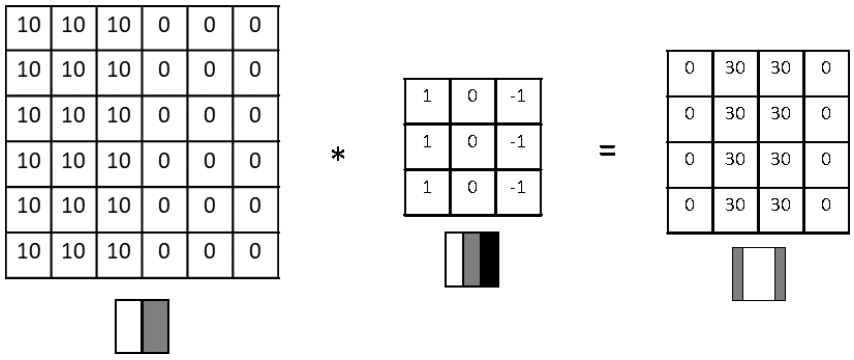


Figure 2.5: Edge detecting kernel.

A convolution layer performs multiple operations of this type. The input consists of multiple matrices, commonly called feature maps or channels. A filter is comprised a kernel matrix for each of the input channels. A convolution operation is performed for each of the kernels in the filter and the corresponding input maps. The results of these operations are added element-wise to produce a single output feature map. A convolution layer can have many of these filters, creating multiple output feature maps, as shown in Figure 2.6. Since convolutions are linear operations, a non-linear function is applied to each output channel to give the network non-linear regression capabilities.

A parallel can be established between fully connected neural networks (dense MLPs) and convolutional neural networks if we consider the following substitutions:

- A single activation value in an MLP is replaced by a feature map matrix.
- A single weight in an MLP is replaced by a kernel matrix.
- The multiplication operation is replaced by the convolution operation.
- The adding of values in an MLP becomes the element-wise addition of feature maps.

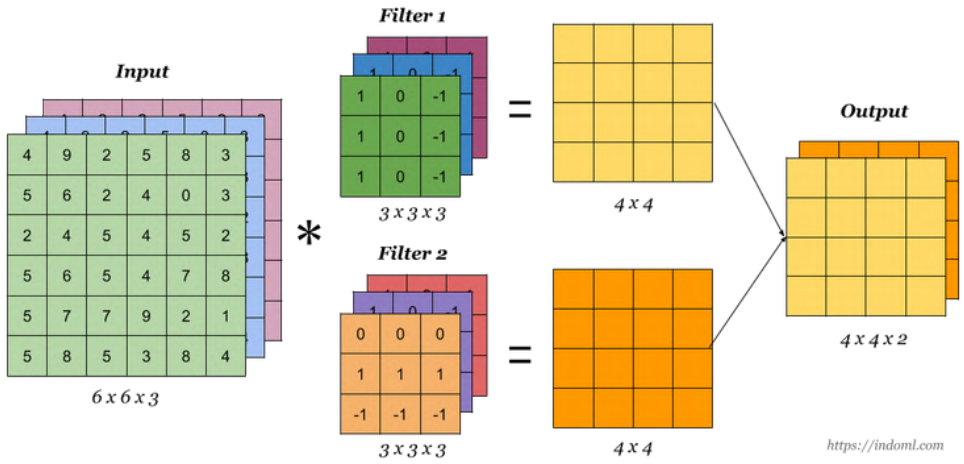


Figure 2.6: Multi-filter convolution with 3 input channels (RGB image), and 2 output channels.

Therefore, in some sense, convolutional neural networks are also fully connected in the feature map space, since the kernels map every input channel to every output channel. This is, of course, not true in the pixel space, which is the reason why CNNs are so much less computationally expensive than similarly sized fully connected neutral networks.

Mathematical Overview

A convolution layer C^l takes as input a set of feature maps $\phi_c^{l-1}(i,j)$, produced by the previous layer C^{l-1}, and performs convolutions on the input feature maps using a set of kernels $K_{c'c}^l(i,j)$ with learned weights:

$$\phi_{c'}^l = C^l(\phi_c^{l-1}) = Pool\left(ReLU\left(\sum_{c=1}^{N_c} K_{c'c}^l * \phi_c^{l-1} + B_{c'}^l\right)\right) \tag{2.8}$$

where $ReLU$ is a non-linear activation function, $Pool$ is a pooling function that sub-samples the feature maps with the objective of reducing their spacial size. N_c is the number of input feature maps (channels) and c, c' are the indexes of the input and output feature maps respectively. The bias terms $B_{c'}$ are also learnable parameters that introduce extra degrees of freedom to the neural network. The symbol $*$ represents a discrete convolution operation over the spatial indexes i and j:

$$(K * \phi)(i,j) = \sum_m \sum_n K(m,n)\phi(i+m, j+n) \tag{2.9}$$

Technically, this should be called a cross-correlation operation, which is different than the classical discrete convolution, since the filter indexes are not flipped. Herein, this operation shall be called convolution with the intent of following the literature on convolutional neural networks and the operation implied by their name.

Since the kernels are of limited size, typically 3x3, 5x5 or 7x7, the indexes m and n vary between $[-1,1]$, $[-2,2]$ or $[-3,3]$ respectively. The kernel is therefore defined for these values of x and y.

The ReLU function is preferred over the classical Sigmoid function for two main reasons (Figure 2.7).

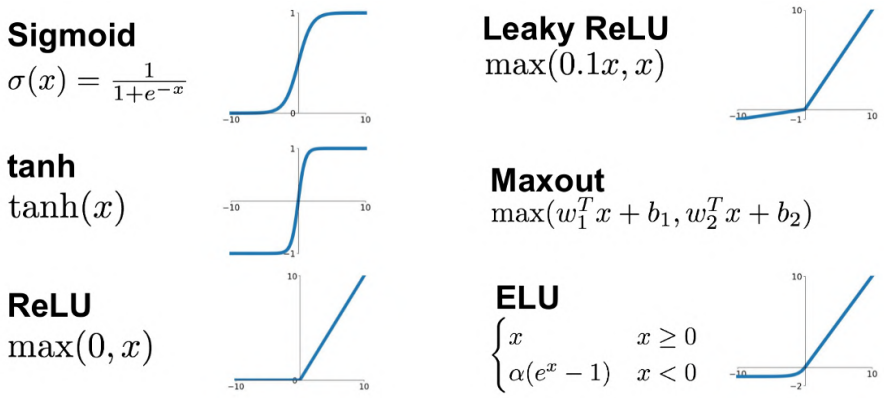


Figure 2.7: Example of different commonly used activation functions.

Firstly, it allows the activations to be large without the gradient vanishing phenomenon that occurs in deep neural networks. This phenomenon stops the network from learning effectively when the gradients of the activation function are small. This happens with the sigmoid function because its gradient tends to zero as its activation increases. The positive part of the ReLU function allows the gradient to be propagated more effectively since its gradient is always equal to 1 for $x > 0$. Secondly, it allows a single neuron to have an unbounded activation, which is argued to give the neuron a bigger expressive power. It allows the neuron to overcome the activation noise, caused by other neurons, to produce clearer representations in the following layers.

The pooling step $Pool()$ is optional and is usually not performed at every layer in very deep convolutional networks. The most common type of pooling is Max Pooling, where the maximal activations of a feature map are selected, using a pre-selected stride. A stride of 2 corresponds to sub-sampling 1 in 4 pixels of the feature maps, in squares of side 2, as seen in Figure 2.8. Another common option is the use of Average Pooling, where the average of the 4 pixels is taken instead of the maximum.

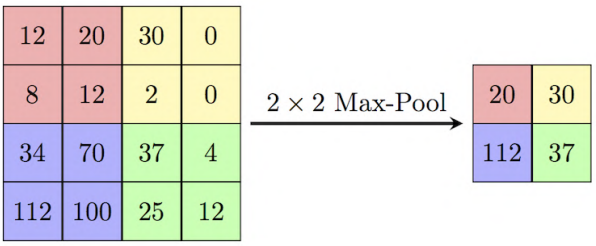


Figure 2.8: Example of Max Pooling sub-sampling operation.

2.2.2 Building a Full Convolutional Network

We just described how a single layer in a neural network transforms a set of feature maps, or an input image, into another set of feature maps. The resulting maps combine information from all input channels, encoding the spatial relationships between the input features using kernel convolutions, resulting in more complex features. A convolutional neural network consists of the composition of many of these layers to produce a complex set of features that can be easily used to produce accurate image classifications.

This process is called feature extraction and is done automatically through the learnable parameters of the network. The final feature maps are given by:

$$\phi_{c'}^{N_l} = f(\phi_c^0) = (C^{N_l} \circ C^{N_l - 1} \circ ... \circ C^2 \circ C^1)(\phi_c^0) \tag{2.10}$$

where N_l is the number of convolutional layers of the network, f is the function composition of all covolutional layers and ϕ_c^0 is the input image.

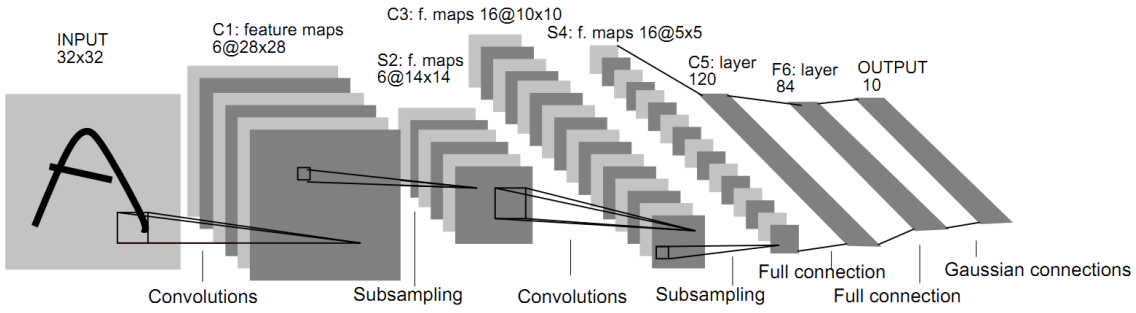


Figure 2.9: LeNet - Convolutional architecture used for character recognition in [10], where each plane is a feature map and the final 3 layers are fully connected.

Image classification CNNs are completed with a classification head. This classification head transforms the final feature maps by flattening them into a 1 dimensional vector and feeding it through a fully connected Multi Layer Perceptron, as in Figure 2.9. The output is a vector $\hat{y}_{k'}$ whose components estimate the likelihood of each object class, in this case characters, being present in the image.

The input to the Multi Layer Perceptron v_k^0 is a flattened representation of the final feature maps $\phi_{c'}^{N_l}$. $\phi_{c'}^{N_l}$ and v_k^0 have exactly the same elements but the former is a tensor with three dimensions $N_c \times H \times W$, where H and W are the height and width of the final feature maps, while the latter is a vector with a single dimension equal to $N_c \cdot H \cdot W$.

The learneable weights of network we just described are all the weights and biases in the convolutional and dense layers - $K_{c'c}^l$, $B_{c'}^l$, $W_{k'k}^l$ and $b_{k'}^l$ - henceforth conjointly designated by $\boldsymbol{\theta}$. The full classification CNN $h_{\boldsymbol{\theta}}$ can be defined as the composition of the convolution layers f from equation 2.10 and classifier g from equation 2.3, producing the class likelihood predictions $\hat{\mathbf{y}}$ from input images $\phi_c^0(i,j) = \mathbf{x}$:

$$\hat{\mathbf{y}} = h_{\boldsymbol{\theta}}(\mathbf{x}) = g(f(\mathbf{x})) \tag{2.11}$$

Training the full network is done with the Stochastic Gradient Descent algorithm described in section 2.1, where now the dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N_t}$ contains tuples of images and the respective labels.

2.3 Convolutional Neural Network Architectures

In the previous section we covered the basics of how a Convolutional Neural Network is constructed and how it learns to classify objects in an image by training with a back-propagation algorithm. This description was based on the AlexNet architecture [12], which was the first large scale CNN trained on two Graphical Processing Units (GPUs), using model parallelism, and achieved outstanding performance on the ImageNet dataset [11]. This network employed 5 convolutional layers with ReLU activation functions, 3 of them with Max Pooling layers. The number of channels of each layer, starting with 3 for the RGB input image, was the following: $N_c^l = \{3, 96, 256, 384, 384, 256\}$. The output of these convolutional layers was flattened to a 9216 component vector and fed through another 2 fully connected (dense) layers with 4096 neurons each. The classification layer had 1000 neurons, corresponding to the 1000 categories of ImageNet. A final Softmax layer produced the probability scores of each class. This network had more than 62 million learneable parameters, which was unprecedented in 2012, and achieved a 40.7% top-1 error (true class is not the highest probability prediction) and a 16.4% top-5 error (true class is not within the 5 with highest probability prediction).

The increase in computational resources, mostly in terms of GPU memory, processing speed and number of processing cores, has allowed bigger models to surface, which achieved even better classification scores on the ImageNet benchmark. Two of the most used convolutional architecture are VGGs and Residual Networks (ResNets), whose variants will be referenced and applied multiple times in this work.

2.3.1 VGG

The VGG network [13] (from the UK based Visual Geometry Group) is an extension of AlexNet that uses more convolutional layers and overall a larger number of parameters. Several variants are created from the VGG-11, with 8 convolutional layers and 3 fully connected layers totalling 133 million parameters, to the VGG-19, with 16 convolutional layers, 3 fully connected layers and 144 million parameters. Most of the parameters are in the fully connected layers, which stay the same across all network variants, but most of the computation time and memory usage, during training, is in the convolutional layers. The 19 layer variant of this architecture achieved a 23.7% top-1 error and a 6.8% top-5 error on ImageNet. Figure 2.10 shows the schematic representation of a VGG16 architecture.

Computational Performance

The number of FLOPs (floating point operations - adding, multiplying, ...) in a convolution operation with kernel size $k \times k$, feature size $h \times w$ with input channels c and output channels channels c' is equal to $2 \times k^2 \times h \times w \times c \times c'$, which for $k = 3$, $h = w = 224$ and $c = c' = 64$ equates to 3.7 Giga-FLOPs. These values correspond to the second layer of a VGG network. The number of FLOPs for a matrix operation in a $n \times n$ dense layer is equal to $n^2 + n(n-1)$, which gives only 34 Mega-FLOPs for $n = 4096$. This corresponds to the second to last dense layer of a VGG network.

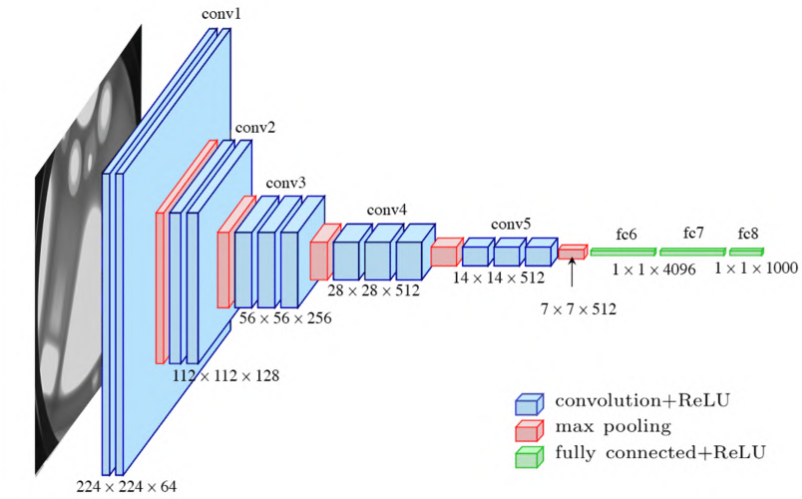


Figure 2.10: Schematic representation of a VGG16 network, with 13 convolutional layers, 5 of them with max pooling, and 3 fully connected layers, totalling 138 million parameters. The numbers $h \times w \times c$ represent the resolution $h \times w$ and the channel depth c of the feature maps at each stage.

The memory usage comes from the fact that all activations of the network must be kept in memory so that the back-propagation can occur and all the gradients can be computed. Besides this, all gradients at each neuron are also stored, which takes just as much space as the activations. The parameters and respective gradients represent only a small part of the memory usage when compared to the activations. The reason that convolutional layers require more memory is that the feature maps can have large spatial and channel dimensions like $(c, h, w) = (64, 224, 224)$ while dense layers only have a single vector of size $(n) = (4096)$ and therefore require less memory for the activations.

The importance of the model size comes into play when deploying the model, in mobile devices or other memory limited applications, because during inference the activations need not be kept in memory. In summary, convolutional layers require more FLOPs and memory during training than dense layers but less memory is required during inference, because the number of parameters is lower.

The VGG family ended up having the models with the highest computational requirements in terms of FLOPs and model size. Subsequent approaches focused on achieving better performance with the same or less computational resources. Still, they are widely used in other deep learning research topics because they represent the CNN architecture in its most basic and general form, whereas more recent approaches fundamentally alter the structure of CNNs to make them more efficient.

Batch Normalization

In most applications the VGG architecture is completed with batch normalization layers [53]. These layers normalize the activations of the feature maps. This makes the activations of the different layers more independent, which stabilizes the training of the whole network, allowing higher learning rates.

The batch normalization starts by transforming the feature maps of a layer $\phi_c^l(i, j)$ by subtracting their mean and dividing the result by the standard deviation, over a batch of images. The normalized features are then scaled and shifted by two learneable parameters γ and β:

$$\hat{\phi}_c^l(i,j) = \gamma \frac{\phi_c^l(i,j) - \mathbb{E}[\phi_c^l(i,j)]}{\sqrt{Var[\phi_c^l(i,j)]}} + \beta \tag{2.12}$$

The parameters γ and β allow the network to learn the optimal mean and standard deviation of the activations of a layer without creating training instabilities between layers. This may also have a regularizing effect because the activations for a single image are no longer deterministic, since they depend on the images from the rest of the batch, introducing perturbations that will force the network to learn more robust and general representations.

Although not discussed in the batch normalization article [53], it is believed that the normalization process itself also helps the network generalize better, because the normalization is an invariance mechanism, in this case, invariant to scalling and shifting of the input. This means that, for example, the network becomes invariant to image brightness and contrast across batches, which is a form of generalization. This effect is more pronounced with a smaller batch size and in the limit of 1 image per batch it is called instance normalization. In that case, full invariance is achieved not only across batches but across each image. The deeper batch normalization layers would therefore be invariant not only to brightness and contrast, but also to texture and more complex spurious patterns. Each batch normalization layer thus adds another level of invariance to the network and aids in generalization.

2.3.2 ResNet

One major problem with deep neural networks, in general, is the phenomenon of vanishing gradients. Gradient vanishing happens when an architecture has many layers and the gradient is not properly back-propagated to the earliest layers, which makes those layers essentially frozen during training. This can happen when using a sigmoid activation function because the gradient of this function in near 0 when the function tends to $-\infty$ and ∞. The ReLU activation function partially solves this problem but not completely.

A poor initialization of weights, with small matrix singular values, can also completely eliminate an input signal to the point that only the deeper weights affect the network output. In this case the earlier weights will receive no back-propagated gradient signal and will not move. A good initialization of the network weights may partially solve this problem.

A third solution is the introduction of skip-connections and residual connections in Residual Neural Networks (ResNets) [15]. This architecture is composed of multiple residual blocks (Figure 2.11), whose convolutional layers compute residual feature maps, which are added to the input feature maps to produce the output feature maps. In this manner, the gradient is propagated straight through the skip-connection without being altered and no gradient vanishing will occur.

This allows the creation of much deeper networks, up to hundreds of layers deep, which are much more stable and easier to train. It is argued that deeper representations are preferable for image classification since they allow more complex patterns to be captured. Many variants of ResNets were created, from the ResNet-18 with 16 convolutional layers divided into 8 residual blocks, to ResNet-152 with 150 convolution layers divided into 50 residual blocks. All variants have one extra input convolution layer and

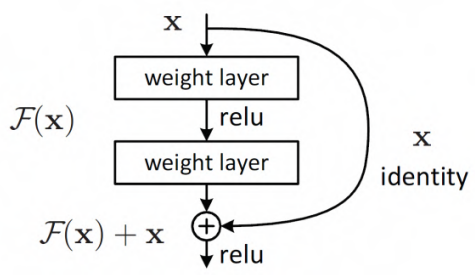


Figure 2.11: Schematic representation of a residual block with two convolutional layers and an identity branch that skips the convolutions.

one final dense classification layer. The 152 layer variant achieved a 19.4% top-1 error and 4.5% top-5 error on ImageNet. Table 2.1 summarizes all the different standard residual network architectures.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 2.1: Summary of the architectures of the ResNet family. The numbers $k \times k, c$ of each convolution represent the kernel size $k \times k$ and the number of filters c of each convolution. The brackets $[.]$ define a residual block with 2 or 3 convolutional layers. Each block can be repeated in series $\times n$ times.

ResNets are widely used in other deep learning research topics because many of the CNN architectures that were later developed are based heavily on the residual architecture. Thus, the ResNet architecture can reasonably represent many of the architectures that followed it.

2.4 Convolutional Neural Network Applications

Until now we described convolutional neural networks in the context of supervised image classification, assuming that each image only portraits a single object of interest. In typical real-world application this framework must be extended to images where one or more objects may be present, or none. The algorithms should therefore be able to identify all the objects in the images as well as their respective positions. We shall cover some of these applications since they will be the object of experiments later in this thesis.

Semantic Segmentation consists of correctly labelling each pixel in an image according to what

class of objects it belongs to. Object Detection consists of drawing a bounding box around each known object in an image and correctly predicting its class. Instance Segmentation can be thought of as the combination of both these tasks, where the pixels from two objects of the same class are identified as belonging to two different instances of the same class. This is done by drawing polygons that cover each unique object in the image and predicting its class.

Besides these supervised tasks, we shall also explore applications and some unsupervised tasks in the context of image generation and style transfer. These techniques are often the building blocks for image augmentation methods, which are useful in making networks learn more general representations. Our texturization technique is itself based some of these techniques so they will be covered with some detail.

2.4.1 Semantic Segmentation

Seminal Work

The first use of a deep convolutional neural network for semantic segmentation was in [27], where fully convolutional neural networks (FCNs) were shown to be capable of semantically segmenting images. The method consists of simply taking a section of a common architecture like a VGG16 network, adding an extra convolutional layer with the number of channels equal to the number of classes, up-sampling the resulting feature maps to the input size and finally passing it through a soft-max layer. Each feature map encodes therefore the network predictions of the probability of each pixel belonging to a class of object. These predictions are compared to the true semantic maps with a cross-entropy loss, which is minimized through a gradient descent algorithm. Unknown objects and background pixels are usually assigned their own default class. To perform inference, the class with highest probability is assigned to each pixel. This architecture is improved by adding extra prediction layers at the early convolutional layers of the encoder, upsampling the predictions to the full resolution and combining the results of all prediction layers. Figure 2.12 showcases the semantic segmentation task with a simple FCN architecture.

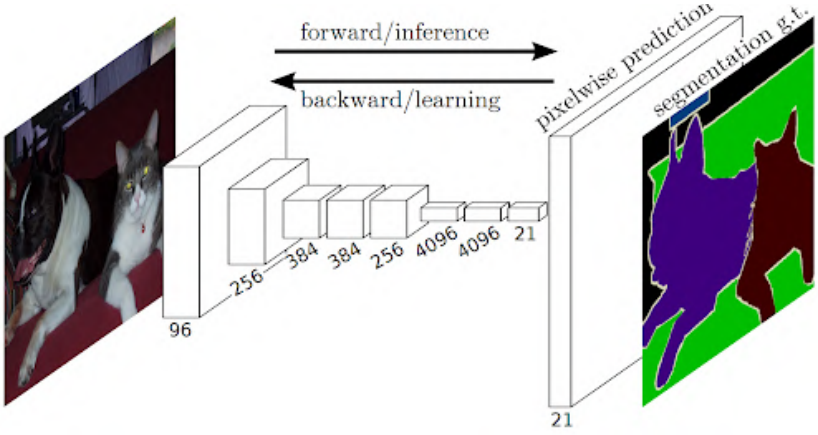


Figure 2.12: Schematic representation of a basic FCN architecture for semantic segmentation.

Performance Metrics

The predictions of the network can further be evaluated using different performance metrics. The most commonly used metric is the mean Intersection over Union (mIoU) between semantic predictions and the true semantic segmentation maps. Let Ω be the set of all pixels in the image, $P_k \in \Omega$ the set of predicted pixels for a class $k \in [1, K]$, $T_k \in \Omega$ the set of true pixels for a class k, and $|.|$ the cardinality operator. The mIoU is calculated as:

$$mIoU = \frac{1}{K} \sum_k \frac{|P_k \cap T_k|}{|P_k \cup T_k|} = \frac{1}{K} \sum_k \frac{\mathcal{C}_{kk}}{\sum_{k'} (\mathcal{C}_{kk'} + \mathcal{C}_{k'k} - \mathcal{C}_{kk})} \tag{2.13}$$

where $\mathcal{C}_{kk'}$ is the confusion matrix of the network predictions, with elements $\mathcal{C}_{kk'} = |T_k \cap P_{k'}|$, which counts the number of occurrences of all K^2 possible true/predicted class pairs.

Pixel Accuracy (PA) is calculated by dividing the True Positives by the total number of pixel class predictions:

$$PA = \sum_k \frac{|P_k \cap T_k|}{|\Omega|} = \frac{\sum_k \mathcal{C}_{kk}}{\sum_k \sum_{k'} \mathcal{C}_{kk'}} \tag{2.14}$$

Mean Pixel Accuracy (MPA) is similar except that the accuracy is calculated class-wise and averaged. The True Positives are divided by the False Negatives plus the True Positives:

$$MPA = \frac{1}{K} \sum_k \frac{|P_k \cap T_k|}{|T_k|} = \frac{1}{K} \sum_k \frac{\mathcal{C}_{kk}}{\sum_{k'} \mathcal{C}_{kk'}} \tag{2.15}$$

Finally, the Frequency Weighted Intersection over Union (FWIoU) is a version of mIoU where the IoU of each class is weighted by the frequency of the respective class:

$$FWIoU = \sum_k \frac{|T_k|}{|\Omega|} \frac{|P_k \cap T_k|}{|P_k \cup T_k|} = \sum_k \frac{\sum_{k'} \mathcal{C}_{kk'}}{\sum_{k'} \sum_{k''} \mathcal{C}_{k'k''}} \frac{\sum_k \mathcal{C}_{kk}}{\sum_{k'} (\mathcal{C}_{kk'} + \mathcal{C}_{k'k} - \mathcal{C}_{kk})} \tag{2.16}$$

Fully Convolutional Neural Networks

The main shortcoming of the FCNs in [27] was that the neurons of the final layer had a relatively small receptive field and therefore failed to capture global information to classify each pixel. To solve this, [54] proposed averaging the whole feature maps and concatenating this channel-wise vector to each neuron, giving it contextual information about the whole image.

Since the predicted segmentation maps are generated by upsampling feature maps with low spatial resolution, the resulting map typically have low spatial precision. This issue can be addressed by using fully connected Conditional Random Fields (CRFs) [26] or Markov Random Fields (MRFs) [55] to improve the localization of the predictions of the upsampled layer.

Encoder-Decoder Architectures

The performance of semantic segmentation architectures improved when encoder-decoder models with deconvolution operations were introduced in [56]. This architecture mirrors the feature extractor (en-

coder) with a decoder that learns deconvolution layers, also called transposed convolutions, to produce segmentation maps. Transposed convolutions act in an opposite manner to vanilla convolutions, by taking each pixel, multiplying it channel-wise by the kernel weights and adding the multiple pixel values to the output feature map. This allows a learned upsampling of the feature maps which incorporates and combines information from multiple channels.

The encoder-decoder design is improved in [57] by using the maxpooling indexes of the encoder to upsample the feature maps in the decoder instead of learning deconvolutions. The upsampled feature maps can then be processed using vanilla convolutions.

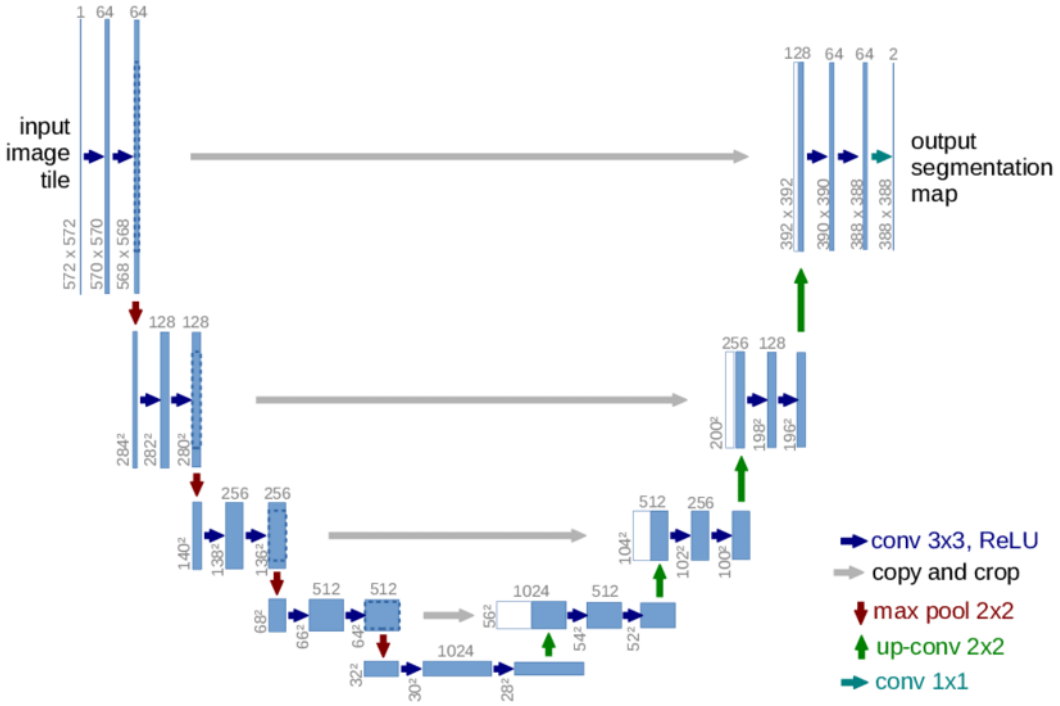


Figure 2.13: Schematic representation of the U-Net architecture for semantic segmentation.

One major semantic segmentation advancement in the category of encoder-decoder architectures was achieved in the medical sector with the design of U-Net [28], shown in Figure 2.13. This network introduces skip connections between layers of the encoder and the corresponding (mirrored) deconvolutional layers of the decoder. These connections allow the network to inject fine spatial information from the early layers of the network into the last layers of the decoder, improving the performance and localization of the semantic predictions.

Feature Pyramid Networks

Feature Pyramid Networks (FPNs) use skip-connections like U-Net but forego the need for deconvolutions. The feature maps of the encoder are passed through a 1 by 1 convolutional layer and the deeper maps are sequentially upsampled and concatenated with the maps at the next layer [58].

The method of [54] is generalized to multiple scales by [59], where the feature maps produced by an encoder are passed through a pyramid pooling scheme, which pools the image at multiple pool sizes, including global pooling. All features from pooling pyramid are passed through a dimensionality reducing

1x1 convolution, upsampled to the same resolution and concatenated. The resulting multi-scale features are then used to predict semantic maps with a final convolutional layer. Figure 2.14 exemplifies this procedure.

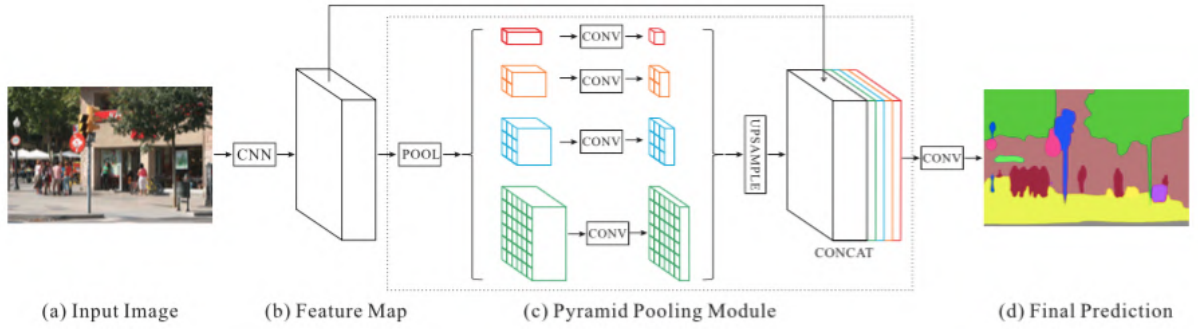


Figure 2.14: Pyramid Pooling scheme that processes the feature maps produced by a CNN at different resolutions before concatenating and combining them.

Deeplab Architecture

Deeplab [60] is one of the most used architectures for semantic segmentation. It employs dilated (also called atrous) convolutions. This type of convolution uses dilated (sparse) kernels, which increases the effective receptive field of each neuron without increasing the computational cost of the layer. With a dilation factor of d, the kernel weights will be separated by $d-1$ zero-valued weights, which do not need to be used in the convolution computation since they have no effect on the result (Figure 2.15). A $n \times n$ kernel with a dilation factor of d will have a receptive field of $d(n-1)+1 \times d(n-1)+1$. For example, a 3×3 kernel with dilation $d=2$ will have a receptive field of 5×5. This type of convolution, when applied in a deep neural network, allows the relationships between distantly spaced features to be learned in fewer layers.

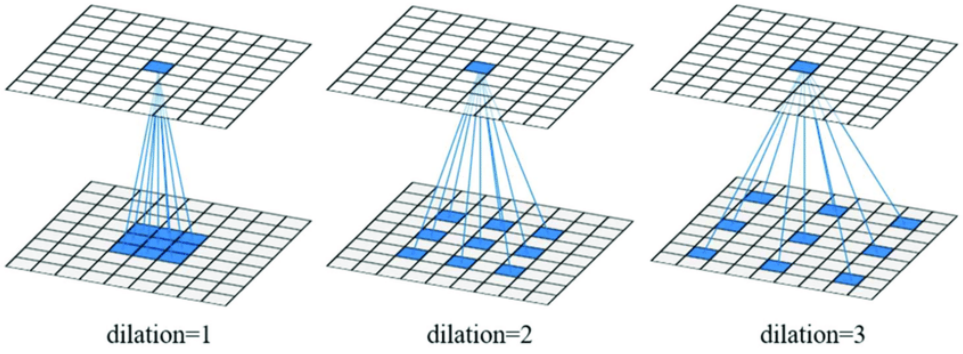


Figure 2.15: Representation of an atrous convolutional kernel with different dilation rates.

Furthermore, this architecture uses a similar pyramid pooling scheme to [59], seen in Figure 2.14, where the multi-scale pooling operations are replaced by atrous convolutions with different dilation rates to produce multi-scale feature pyramids. The predictions are produced with a final convolutional layer and a fully connected Conditional Random Field to improve the localization of the predictions, as in [26].

Figure 2.16 shows the full pipeline, where the pyramid pooling scheme is included in the DCNN module.

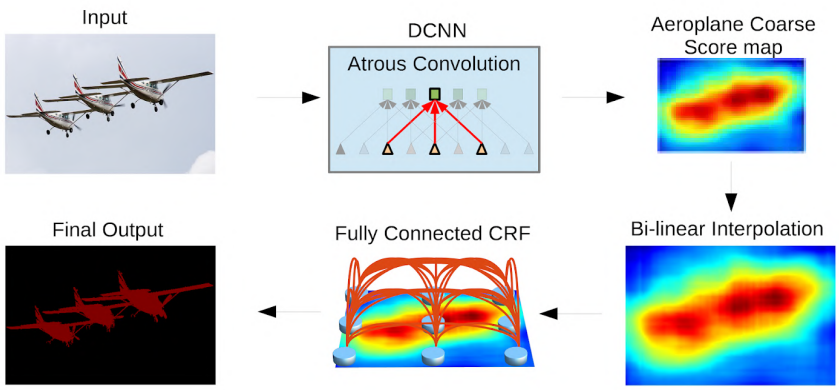


Figure 2.16: Pipeline of the Deeplab architecture.

Later in this thesis, the FCN and Deeplab semantic segmentation architectures will be subject to different experiments in the context of domain generalization.

2.4.2 Object Detection

Passing from an image classification task to an object detection task can seem trivial. To detect multiple objects in an image simply pass an image classifier in multiple patches of the input image and consider a detection when the confidence (probability prediction) for an object is above a certain detection threshold. This can be done in a rolling window fashion, at multiple scales and aspect ratios. The problem becomes apparent when we consider the number of forward passes that a classifier would have to perform to detect all the objects in a single image.

Some applications of object detection algorithms require the detection to happen in real time, so performing classification of all possible image patches becomes infeasible. Even if this process is parallelized to reduce the computation time, the shear amount of floating point operations is completely impractical.

R-CNN and Fast R-CNN

R-CNNs [21] were the first improvement to this method. They use a region proposal step, based on a selective search algorithm [61], to select the Regions of Interest (ROIs) that are more likely to contain an object. The selected image patches are further filtered by combining patches that overlap or are included in each other. The different patches are then resized to a fixed square resolution. A convolutional neural network processes each patch into a feature vector. Finally, a SVM classifier [62] predicts the probability of each class from this feature vector. The feature vector is also used to predict an offset of the ROIs to produce the final bounding boxes.

Still, this approach is not fast enough because a large number of image patches need to be processed by the CNN, often with redundant computations being performed more than once. Fast R-CNN [22] solves this problem by first processing the whole image with a CNN in a single pass, producing a set of feature maps. The selective search algorithm is then applied to the feature maps to produce a set of ROIs. Since the bulk of the computation is already performed, only some fully connected layers are needed to process each ROI into a feature vector and therefore the whole image is processed a lot faster. A final classifier and regression layer compute the label probabilities and bounding box coordinates respectively. The main speed bottleneck now becomes the selective search algorithm.

Faster R-CNN

Faster R-CNN [23] further improves this architecture by substituting the selective search algorithm by a region proposal network (RPN). The region proposal network is based on anchors, which are predefined image patches, at different scales and aspect ratios. The region proposal network calculates, for each of these anchors, a *objectedness* score indicating the probability of an object being present in that patch. This is achieved with one or more convolutional layers where the final layer has a number of channels equal to the number of anchors. The feature maps therefore represent the *objectedness* score at each pixel for each possible anchor. In the same manner as R-CNN, a final classifier layer produces the class probabilities and a final regression layer produces the bounding box coordinates from the feature vector. The full architecture is presented in Figure 2.17.

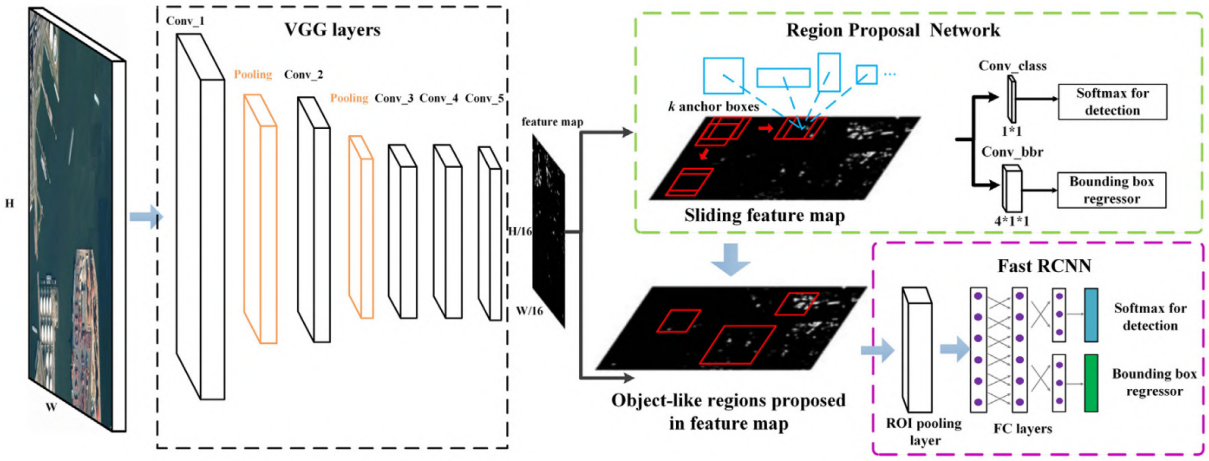


Figure 2.17: Schematic of the Faster R-CNN architecture.

The *objectedness* loss is computed by assigning a positive label (object present in the patch) to the anchors whose Intersection over Union (IoU) with a ground truth box is higher than 0.7 and assigning a negative label to the anchors whose IoU with a ground truth box is lower than 0.3. Intermediate values are ignored because they introduce noise during training. Since negatively labeled anchors are more common, the loss is calculated using a sub-sample of 256 anchors, with as many positive anchors as possible, up to 128. The box regression loss is computed as a simple regression loss to the ground truth box coordinates.

The full architecture is trained in a 4-step process. First, the RPN is trained to only predict *objectedness* scores and bounding box predictions using the two losses described in the previous paragraph. Secondly, a Fast R-CNN is trained with a different feature extracting CNN, using the predictions from the trained RPN. Third, the RPN is re-trained using the feature extractor that was learned by the Fast R-CNN, which is now frozen. Lastly, the Fast R-CNN is fine-tuned using the newly trained RPN network. The Fast R-CNN and RPN now form a unified network with a shared feature extractor.

The Faster R-CNN architecture will be tested later in this thesis in a domain generalization task.

YOLO architecture

For completeness, we present yet another successful architecture which achieves an even faster inference time with acceptable performance. You Only Look Once (YOLO) [24] dispenses with the region proposal network and poses object detection as an end-to-end regression problem. An image is subdivided into $S \times S$ patches. The network predicts the bounding boxes, a confidence score and class probabilities for each image patch. These predictions can be directly used to train the network. To perform inference, the predictions are filtered and overlapping bounding boxes are combined.

Performance Metrics

In Object Detection tasks, the networks are tasked with producing a bounding box for each object in an image with the respective class probabilities. To convert these outputs to performance metrics we need to first define what constitutes a detection. This introduces the notion of Intersection over Union (IoU) for bounding boxes. For a ground truth box B_t and a predicted box B_p, where both are sets of pixels and $|.|$ is the number of elements in the set, IoU is defined as:

$$IoU_{tp} = \frac{|B_t \cap B_p|}{|B_t \cup B_p|} \tag{2.17}$$

A pair B_t, B_p is considered a detection if IoU_{tp} is greater than a certain threshold, $IoU_{tp} > IoU$. Furthermore, a detection is considered positive if the true class is predicted with a probability higher than a certain confidence level (C) and negative otherwise. This defines the number of true positive (TP), true negative (TN), false positive (FP) and false negative (FN) detections for a given confidence level and a given IoU, for a certain class (k). Furthermore, we may now define the Precision (P_k) score and the Recall (R_k) score for a given confidence level, IoU and class:

$$P_k(IoU, C) = \frac{TP}{TP + FP}\bigg|_{IoU,C,k} \tag{2.18}$$

$$R_k(IoU, C) = \frac{TP}{TP + FN}\bigg|_{IoU,C,k} \tag{2.19}$$

To construct the performance metrics, these values are computed at multiple levels of confidence and IoU, and the results are averaged to produce the Average Precision (AP) and Average Recall (AR) for a given class:

$$AP_k = \frac{1}{N_{IoU} N_C} \sum_{IoU,C} P_k(IoU, C) \tag{2.20}$$

$$AR_k = \frac{1}{N_{IoU} N_C} \sum_{IoU,C} R_k(IoU, C) \tag{2.21}$$

The values of IoU are usually linearly spaced points between 0 and 1, excepting 1. The values of confidence level C are usually defined with linearly spaced points in an interval between 0.5 and 1, excepting 1. Occasionally, the IoU threshold may also take a single value, for example $IoU = 0.5$.

Finally, the performance metrics are constructed by averaging these values across classes to produce the mean Average Precision and the mean Average Recall:

$$mAP = \frac{1}{k} \sum_{k} AP_k \tag{2.22}$$

$$mAR = \frac{1}{k} \sum_{k} AR_k \tag{2.23}$$

Sometimes these metrics filter detections by selecting only the first n with highest confidence levels, with the purpose of eliminating noisy low-confidence detections. Furthermore, once again to avoid noisy detections, only bounding boxes above a certain area are considered. This is applicable when there is no interest in detecting small or far away objects.

2.4.3 Instance Segmentation

Once again for completion, we cover Mask R-CNN [29], which performs the tasks of semantic segmentation and object detection simultaneously. This is achieved by extending the Faster R-CNN with a binary semantic segmentation Fully Convolution Network (FCN). Each RoI produced by the Faster-RCNN is passed through the FCN, which produces the class mask.

2.4.4 Image Generation

Until now we explored different architectures for image classification, object detection and semantic segmentation of images. All of these tasks are examples of supervised learning tasks, where a label (supervision) is associated with each image during training and the purpose of the network is to learn to predict the label.

Unsupervised learning is a different class of learning algorithms where no label is associated each data sample. The algorithms of this class are typically tasked with grouping data-points, as for example the K-means algorithm, learn a lower dimensional representation of the data, like Principle Component Analysis (PCA) and other dimensionality reduction techniques, learn a distribution that can be re-sampled for new data-points, learn a mapping between datasets, and other more specific tasks.

We will now explore two architectures that can be implemented with CNNs and that are widely used for most tasks that involve generating images in an unsupervised manner. These architectures will be

reference multiple times during this work and so it is important to properly introduce them.

Auto-Encoders

Auto-Encoders (AEs) are a type of neural network architecture whose purpose is to learn a lower dimensional representation of the data by passing it through an information bottleneck and using that low dimensional representation to reconstruct the original data [52, 63]. An Auto-Encoder is composed of an encoder network which maps inputs to the lower dimensional latent space, and a decoder network which maps the latent space to the original higher dimensional input space (Figure 2.18). This procedure forces the network to learn an internal latent representation that captures as much information about the data as possible. Auto-Encoders find application in unsupervised representation learning, image denoising, image compression and image in-painting because of their ability to filter out irrelevant information. Auto-Encoders are usually trained with a reconstruction loss which can be a simple L2 distance loss, another distance metric or even a pretrained and frozen neural network based perceptual loss [64]. When applying Auto-Encoders to images the encoder and decoder networks are implemented as a vanilla CNNs with transposed convolutions or other upsampling layers for the decoder.

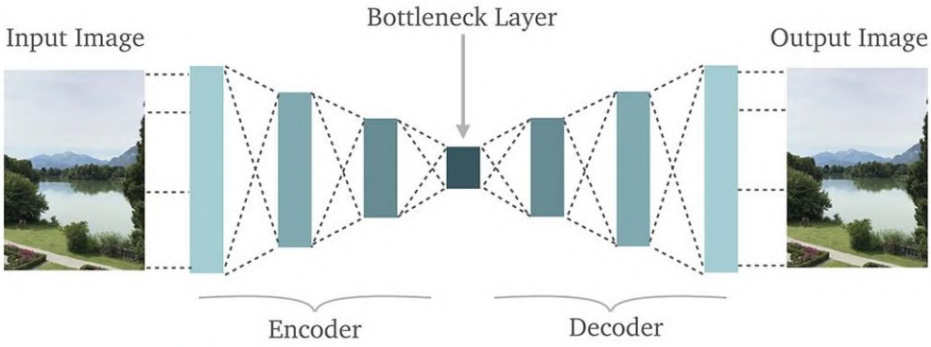


Figure 2.18: Example of a Fully Convolutional Auto-Encoder.

The main caveat of classical Auto-Encoders is that no constraint is applied on the learned latent representations. This means that the representations can have any structure and may not be suited to interpolation. Therefore, generating new data-points by sampling the latent space and decoding it will yield unpredictable results. Variational Auto-Encoders (VAEs) [31] solve this problem by introducing a reparametrization trick whereby, instead of mapping an input to its latent representation, the encoder maps an input to the parameters μ and σ of a gaussian distribution in each dimension of the latent space. The decoding process consists of sampling a point from this latent gaussian distribution and passing it through the decoder network to produce an output. Similarly to classical AEs, a distance loss is applied between the input and output. Additionally, a Kullback-Leibler (KL) divergence loss is applied between the latent gaussian distribution and a prior gaussian distribution with $\mu = 0$ and $\sigma = 1$. This KL divergence regularizes the latent space such that the distribution of representations is contiguous, and therefore two data points are interpolable in the latent space. This allows sampling and generating new data points following the same distribution as the original dataset.

A more recent evolution replaces the reparametrization trick and KL divergence with a Wasserstein

distance metric (earth-mover distance) between the latent distribution and a prior normal gaussian distribution [65]. Wasserstein Auto-Encoders (WAEs) have been shown to generate more realistic images according to the Frechet Inception Distance (FID) [66] score. Moreover, they are a generalization of adversarial Auto-Encoders (AAEs) [67], which use an adversarial loss to regularize the latent space.

Generative Adversarial Networks

Generative Adversarial Networks (GANs) [34] use a different mechanism to generate realistic images. A generator network $G(z)$ maps a prior low dimensional noise distribution $z \sim p(z)$, usually gaussian, to a high dimensional output distribution $\hat{x} \sim p(\hat{x})$, typically images, such that the output distribution is similar to a real distribution $p(x)$. A discriminator network $D(x)$ is then tasked with predicting the probability $\hat{y} = D(x)$ that an image is sampled from the dataset. This architecture is called adversarial because the generator and discriminator are playing an adversarial game in which the discriminator tries to distinguish real images form generated images and the generator tries to produce images that fool the discriminator. This is achieved with an adversarial min-max objective:

$$\min_G \max_D \quad \mathbb{E}_{x \sim p(x)} \log(D(x)) + \mathbb{E}_{z \sim p(z)} \log(1 - D(G(z))) \tag{2.24}$$

An alternated gradient step is used to update the generator and discriminator. First, the discriminator is updated with a gradient ascent step using a batch of dataset samples and batch of noise samples. The first step can be repeated multiple times. Secondly, the generator is updated with a gradient descent step using another batch of noise samples. The generator uses the gradient signal from the discriminator to learn how to fool it.

Deep Convolutional Generative Generative Networks (DCGANs) [36] apply this adversarial training scheme with a CNN based discriminator and a generator with transposed convolutions and/or upsampling layers to generate images (Figure 2.19).

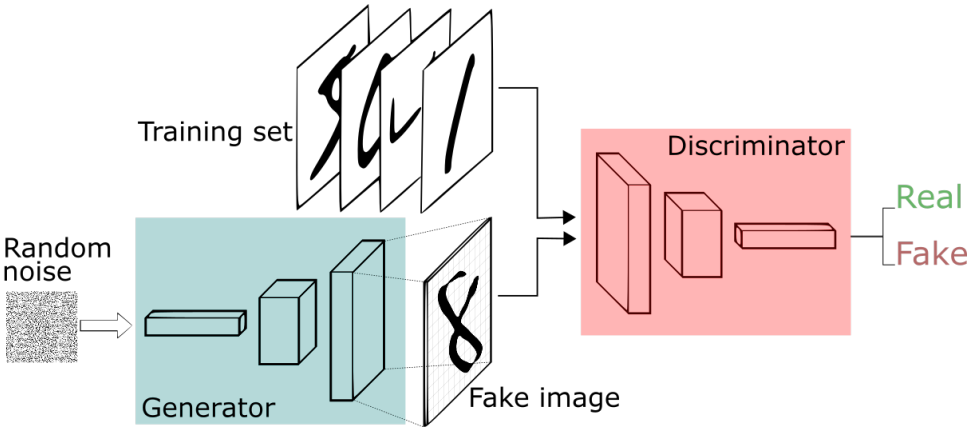


Figure 2.19: Training scheme for the DCGAN architecture.

Conditional Generative Adversarial Networks (CGANs) [35] can use label information to condition image generation. The generator will therefore learn a distribution conditioned on the image labels and generate images of the desired class. This is achieved by feeding the image label as input to the gener-

ator, sampling it randomly alongside the noise, and by feeding it as input to the discriminator, using the sampled and real image labels. InfoGAN [38] conditions the image generation in an unsupervised manner, allowing the networks to discover disentangled factors of variation in the images. This is achieved by maximising the mutual information between the latent variables and the generated images using a separate similarity measuring network and loss functions.

GANs have also found application in image to image translation. Pix2Pix [68] uses a similar architecture to the CGAN but the image is conditioned with another image as a label. For example, this network can learn to map drawings to the respective real images, map street maps to the respective satellite view, or map building façade segmentation maps to real images of building façades. This technique is however densely supervised since it requires one-to-one translation examples. CycleGAN [69] achieves unpaired image to image translation by using two translation networks, $\hat{y} = G(x)$ and $\hat{x} = F(y)$ between two image domains $x \sim X$ and $y \sim Y$, and enforcing cycle consistency between them with an L1 loss. Additionally, one discriminator is used for each network, and every network is trained with a similar adversarial scheme as before (Figure 2.20). This leads perfectly to the next section, which will explore other mechanisms for image to image translations with style transferring techniques.

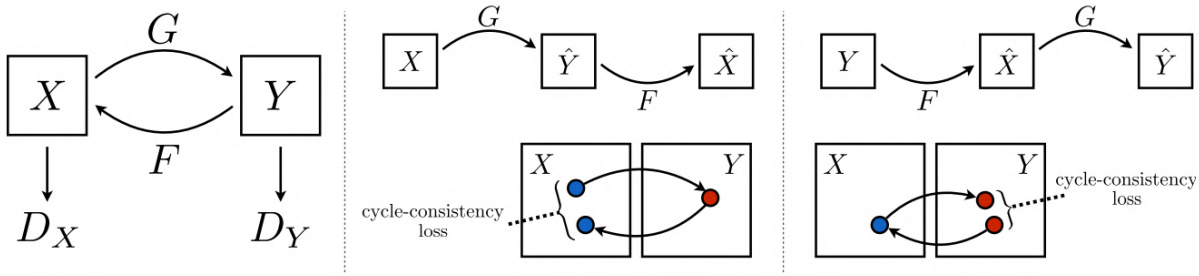


Figure 2.20: Schematic of the CycleGAN Architecture with a Cycle Consistency Loss.

2.4.5 Style Transfer

Overview

Image style transfer is the task of taking the content from an input image $\mathbf{x}$ and re-producing it in the style of a style image $\mathbf{s}$. The first instance of CNN based style transfer happens in [49]. This was achieved by optimizing the pixel values of the content image to minimize a content loss $\mathcal{L}_c$ and a style loss $\mathcal{L}_s$. Firstly, the output image is initialized as white noise. The content and style images are passed through a feature extractor pretrained on ImageNet and the activations of all layers are stored for both images. At each step, the output image is passed through the feature extractor. The content loss $\mathcal{L}_c$ is calculated by averaging the squared distance between the content features and the output features across different layers. The style loss is calculated using the Gram matrices of the features in each layer. For the activations $\phi_c^l(i,j)$ of layer l, with channel index c and spatial indexes i, j, the Gram matrix $G_{c,c'}^l$ is the spatial correlation matrix of $\phi_c^l(i,j)$ between channels:

$$G^l_{c',c} = \sum_i \sum_j \phi^l_{c'}(i,j)\phi^l_c(i,j) \tag{2.25}$$

The style loss $\mathcal{L}_s$ is then the average squared difference between the Gram matrices of the output images and the style image across different layers. Finally the gradients of the loss with respect to the output image are calculated and the output image is updated to minimize the loss.

Since this involves an optimization process, producing a single stylized image is an expensive process. Later, a convolutional auto-encoder architecture was adopted, which learns the image transformation associated with a certain style image [70]. This technique performs equivalent style transfer with a single network pass, forgoing the need of the expensive optimization steps. Still, this method is limited to a single style per network. Adaptive Instance Normalization (AdaIN) opened the door to arbitrary style transfer [71], where a single auto-encoder network, when fully trained, can perform style transfer between any two images in a single pass.

Style Transfer with Adaptive Instance Normalization

AdaIN is a moment-matching process. It is assumed that the style information of the images is fully represented in the first and second order moments of the feature maps produced by a pretrained encoder network. The encoder part of the auto-encoder $e(\mathbf{x})$ produces a set of content feature maps $e(\mathbf{x}) = \phi^{\mathbf{x}}_c(i,j)$. Similarly, the same encoder produces the style feature maps $e(\mathbf{s}) = \phi^{\mathbf{s}}_c(i,j)$. The index c represents the channel and i, j are pixel indexes. These indexes may be dropped in the mathematical formulation to ease the legibility. AdaIN normalizes the content feature maps $\phi^{\mathbf{x}}_c(i,j)$ to have the same first and centered second order moments as the style feature maps $\phi^{\mathbf{s}}_c(i,j)$. The mean and standard deviation of the feature maps are calculated by:

$$\mu_c(\phi_c) = \frac{1}{HW}\sum_i^H \sum_j^W \phi_c(i,j) \tag{2.26}$$

$$\sigma_c(\phi_c) = \sqrt{\frac{1}{HW}\sum_i^H \sum_j^W (\phi_c(i,j) - \mu_c(\phi_c))^2} \tag{2.27}$$

The normalization process of AdaIN is the following:

$$AdaIN(\phi^{\mathbf{x}}_c, \phi^{\mathbf{s}}_c) = \phi^{\mathbf{x}|\mathbf{s}}_c(i,j) = \sigma(\phi^{\mathbf{s}})\left(\frac{\phi^{\mathbf{x}} - \mu(\phi^{\mathbf{x}})}{\sigma(\phi^{\mathbf{x}})}\right) - \mu(\phi^{\mathbf{s}}) \tag{2.28}$$

The stylised image output $\mathbf{o}$ is generated by the decoder part of the auto-encoder $\mathbf{o} = d(\phi^{\mathbf{x}|\mathbf{s}}_c(i,j))$

The content and style losses evaluate the images produced by the decoder by re-passing them through the encoder and comparing the feature maps to the original ones:

$$\phi^{\mathbf{o}}_c(i,j) = e(\mathbf{o}) \tag{2.29}$$

The content loss is given by the mean squared reconstruction error across channels and pixels:

$$\mathcal{L}_c = \frac{1}{CHW} \sum_{c}^{C} \sum_{i}^{H} \sum_{j}^{W} \left(\phi_c^{\mathbf{x}|\mathbf{s}}(i,j) - \phi_c^{\mathbf{o}}(i,j) \right)^2 \tag{2.30}$$

The style loss is given by the mean squared error in the moments across all channels. This loss is averaged across the feature maps produced at multiple layers L of the encoder.

$$\mathcal{L}_s = \frac{1}{LC} \sum_{l=1}^{L} \left[\sum_{c=1}^{C} \left(\mu_{lc}(\phi_{lc}^{\mathbf{s}}) - \mu_{lc}(\phi_{lc}^{\mathbf{o}}) \right)^2 + \sum_{c=1}^{C} \left(\sigma_{lc}(\phi_{lc}^{\mathbf{s}}) - \sigma_{lc}(\phi_{lc}^{\mathbf{o}}) \right)^2 \right] \tag{2.31}$$

The final loss is the sum of the two losses parameterized by an hyper-parameter λ set to 10 [71] unless otherwise specified:

$$\mathcal{L} = \mathcal{L}_c + \lambda \mathcal{L}_s \tag{2.32}$$

The network is trained to empirically minimize the expected value of the loss $\mathbb{E}_{\mathbf{x} \sim X, \mathbf{s} \sim S} \, \mathcal{L}(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ are the learneable weights of the decoder. The complete style transfer network is shown in Figure 2.21.

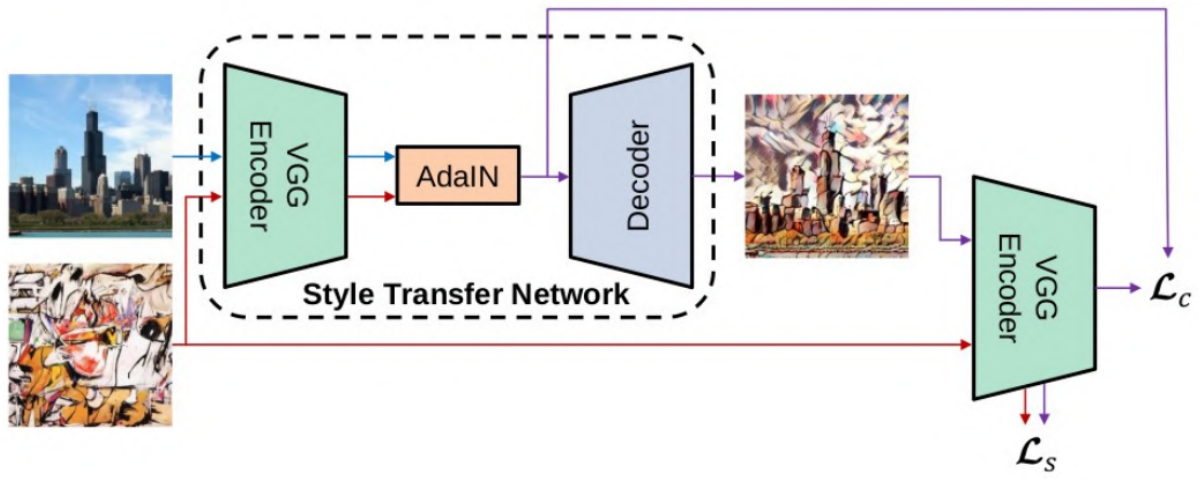


Figure 2.21: AdaIN architecture.

The encoder is a VGG network pre-trained on ImageNet with a classical image classification task. Learning the style transfer task consists of training a decoder that is able to inverse the encoder when the feature map statistics are altered by AdaIN. This stylization technique is therefore an example of transfer learning, since the representations learned by the encoder are used in a different downstream task. The full style transfer network is then trained using a content dataset and a style dataset. The content dataset is MSCOCO, Microsoft's dataset of Common Objects in Context. The WikiArt Dataset, Wikipedia's dataset of art paintings is used for the style images. Each includes around 80 000 images of differing resolutions. Since the auto-encoder is fully convolutional, this method can work in images of any resolution.

2.5 Texture Bias in Convolutional Neural Networks

It is commonly assumed that CNNs owe much of their performance to their ability of recognizing shapes. Many feature visualization techniques [50, 72, 73] show that CNNs effectively learn edge detectors, curve detectors, circle detectors, animal face detectors and human face detectors. The multi-layered and translation invariant architecture allows the network to combine multiple of these shapes to create more high-level shape detectors. It was thought that the structured combination of these features produced networks that were sensible to high level shape cues.

However, some recent studies showed that CNNs are in fact more texture biased than previously thought. In [74], networks with a very low receptive field were shown to work surprisingly well on ImageNet classification. These bag-of-local-features networks (BagNets), are restricted to only capture small scale features and are therefore texture-biased by design. The fact that they work well on ImageNet indicates that other networks could be using texture cues to classify images.

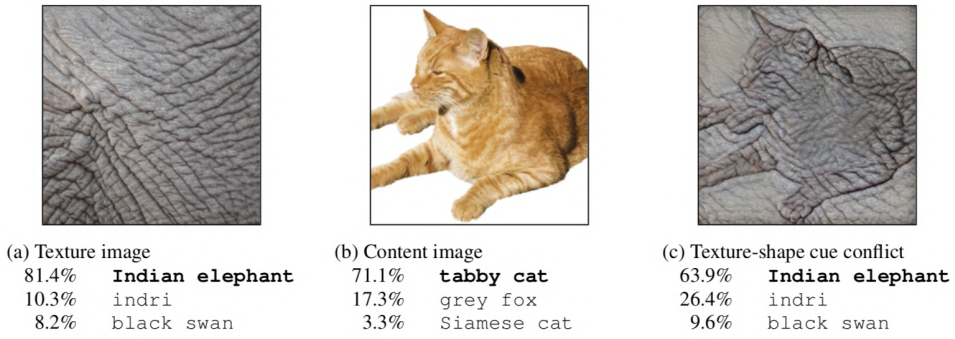


Figure 2.22: Classification of a standard ResNet-50 of (a) a texture image (elephant skin: only texture cues); (b) a normal image of a cat (with both shape and texture cues), and (c) an image with a texture-shape cue conflict, generated by style transfer between the first two images.

The authors of [75] used the original iterative style transfer technique [49] to generate a Cue-Conflict dataset, based on ImageNet, with conflicting shape and textural cues. When these images are presented to CNNs and to human subjects, the CNNs are more likely to classify the image based on texture/style than their human counterparts (Figure 2.22). The human subjects, on the other hand, classified images mostly based on shape/content cues. This study then showed that the texture-bias of CNNs can be attenuated by training the network in another stylized version of ImageNet. This dataset, called Stylized-ImageNet, is generated using the AdaIN-based fast style transfer algorithm [71] with the style images coming from Kaggle's Painter by Numbers dataset [76]. This works as a data augmentation and regularization technique. By making the textural cues arbitrary with respect to the content, the network must learn to classify the content of the image purely based on the shape patterns. The data augmentation thus renders the network texture invariant. Furthermore, networks that are trained on the stylised dataset show more resistance to class-preserving image transformations like, changes in contrast, filtering and different forms of noise.

This work is continued in [77], which performs a multi-sectioned analysis of the origins of texture bias in CNNs. Some key remarks include:

- By analysing performance on Stylized-ImageNet and other mixed-cue datasets throughout training, it is concluded that shape features can be learned just as readily as textural features and often shape features can be learned with less data, less iterations and more accuracy.
- By training classifiers on top of different layers of a deep CNN it is shown that shape information tends to disappear in the deeper layers while textural information is preserved.
- By training CNNs on self-supervised tasks, it is shown that the texture bias emerges in tasks other than classification. Thus, the texture bias is not a consequence of textures being useful for classification. Despite this, ImageNet top-1 classification correlates strongly with texture bias.
- Training ventral visual system inspired CORNets [78] reveals that the texture bias is present even in architectures that are strongly based on the animal and human visual processing core.
- Training networks with local attention heads [79] instead of convolutions reveals that the texture bias does not ensue directly from the use of convolutions.
- Center cropping instead of random cropping increases shape bias but worsens classification accuracy.
- Higher learning rates favor shape bias while lower learning rates favor texture bias.

By analysing some of these observations it is possible to conclude that texture bias is a form of overfitting. Typically, overfitting happens at a data-point level, which can be controlled for with training, validation and testing dataset splits, and eliminated using regularization and data augmentation techniques. Texture-bias is an overfitting at a dataset level, where the network overfits the low level statistics of a particular dataset, such as textures and small scale patterns. This overfitting does not happen in the early stages of training neither when the learning rate is high, but it appears later in the training with low learning rates. Moreover, regularizing techniques like Random Cropping force the networks to maximise their capacity by learning the most useful features for the particular task at hand. Since low level features like textures require less network capacity to encode than high level shape features, if the low-level features are sufficient to accomplish the particular task, then the network will use its capacity to encode diverse low-level features rather than specific high level features.

In the next section we will explore a line of research that deals with distributional overfitting and see different approaches to train networks that generalize better to different datasets/distributions.

2.6 Generalization in Convolutional Neural Networks

In deep learning tasks it is expected that the training and testing data come from the same distribution. When this is not the case, we are presented with an out-of-distribution (OOD) generalization problem. This task consists of learning a source distribution and being able to perform inference in a target distribution with different characteristics. This problem can be further decomposed depending on our access to unlabeled data from the target distribution during training.

In a Domain Adaptation (DA) framework, a network is trained using the source distribution and its generalization performance is boosted by taking advantage of unlabeled samples from the target distribution. In a Domain Generalization (DG) framework, only the source distribution is accessible and the objective is to generalize to any other distribution without any prior knowledge of its nature. Furthermore, each framework can be subdivided into single-source or multi-source scenarios, according to the number of source distributions that are available during training.

Let us denote $\mathcal{S}_i$ a dataset of labeled data with a source distribution and $\mathcal{T}_i$ a dataset of unlabeled data with a target distribution. Table 2.2 illustrates the four different scenarios that arise, where the task is to correctly label samples from $\mathcal{T}_i$:

Framework	Training Data	Testing Data
Single-source DA	$\mathcal{S}_0 \cup \mathcal{T}_0$	$\mathcal{T}_0$
Multi-source DA	$\mathcal{S}_{0,1,...} \cup \mathcal{T}_0$	$\mathcal{T}_0$
Single-source DG	$\mathcal{S}_0$	$\mathcal{T}_{0,1,...}$
Multi-source DG	$\mathcal{S}_{0,1,...}$	$\mathcal{T}_{0,1,...}$

Table 2.2: Summary of Out-of-Distribution frameworks.

2.6.1 Domain Adaptation

In the literature, deep domain adaptation techniques can be divided into four categories [80, 81]: distribution matching approaches, domain adversarial approaches, generative adversarial approaches and batch normalization based approaches. Most of the covered approaches are of the single-source variant but most of them can be extended to work in multi-source scenarios.

Distribution Matching Domain Adaptation

Distribution matching approaches include, as part of their loss function, a distance metric between the source and target latent space distributions. This distance is minimized with the objective of aligning the latent spaces produced by source and target images when passed through a feature extractor.

Mean Discrepancy Distance (MDD), proposed by [82], measures the distance between the expected value of the features generated by the source and target distributions. This distance is minimized if the source and target images produce the same average features.

In [83], the covariance matrices of the features produced by source and target images are compared with a squared matrix Frobenius norm. This norm is minimized such the features have matching second order statistics. This idea is extended to higher order central moments in [84].

Furthermore, based on the theory of optimal transport, [85] minimizes a regularized Wasserstein distance loss to match source and target latent distributions. The Wasserstein distance is replaced by a learned distance metric in [86] yielding better results.

The previous techniques assume matching source and target conditional distributions with respect to the label. This hypothesis might not always be true. To address this, the MDD in [82] is further developed

in [87] by adding a class conditional metric between source and target features. The labels of the target classes are generated using a k-nearest-neighbours clustering algorithm, with the centers of each class initialized at the center of the class conditioned source features. Minimizing the learned Contrastive Domain Discrepancy (CDD) decreases intra-class discrepancy while the increasing inter-class margin.

Adversarial Domain Adaptation

Adversarial domain adaptation adds a domain discriminator to the common CNN architecture. Given a feature extractor $f(\mathbf{x}), \mathbf{x} \in \mathcal{S}_i \cup \mathcal{T}_i$, the domain discriminator head $d(f(\mathbf{x}))$ has the task of finding from which domain, $\mathcal{S}_i$ or $\mathcal{T}_i$, the original image $\mathbf{x}$ is sampled from. It achieves this by minimizing a cross-entropy loss $\mathcal{L}_d$ between the predictions of $d(f(x))$ and the true domain. This loss is back-propagated through a gradient reversal layer between f and d, which reverses the direction of the gradient step for the feature extractor f. This moves the weights of f in the sense of maximizing the domain discriminator loss, thus constraining its features to be domain-invariant. Concurrently, a classical cross-entropy classification loss $\mathcal{L}_c$ of a classifier head $g(f(x))$ is also minimized, without the gradient reversal layer. The final objective can be expressed as an adversarial min-max game:

$$\min_{\phi,\psi} \max_{\theta} \quad \mathbb{E}_{\mathbf{x} \in \mathcal{S}_i \cup \mathcal{T}_i} \left[\mathcal{L}_c(g_\psi(f_\phi(\mathbf{x}))) - \mathcal{L}_d(d_\theta(f_\phi(\mathbf{x}))) \right] \tag{2.33}$$

Where ϕ, ψ and θ are the learnable parameters of the feature extractor f, classification head g and domain discriminator d, respectively. This is the basis of domain adversarial neural networks proposed in [88].

In [89], a domain classifier is also used, but the min-max loss is replaced with a cross-entropy confusion loss applied only to the feature extractor. This confusion loss updates the feature extractor weights such that the predictions of the domain discriminator are as uniform as possible. At the same time the discriminator is trained with a classical cross entropy loss. When the features are domain invariant, the domain discriminator is maximally confused and produces uniform predictions for each domain.

The adversarial domain adaptation approach is generalized in [90] and a novel architecture is proposed. Two feature extractors with partially shared weights are used, one for the source dataset and one for the target dataset. The training is now performed in two phases. In the first phase the source feature extractor and classifier are trained with the labeled source dataset. Secondly, the target feature extracted is trained to produce domain-invariant features with an GAN-style adversarial loss, while freezing the weights of the first feature extractor. During testing, the source classifier is used on top of the target feature extractor to label images of the target dataset.

Generative Adversarial Domain Adaptation

Generative adversarial approaches make use of generative adversarial networks to achieve domain adaptation.

The approach of [91] uses two generative adversarial networks with shared weights between the high level layers of the generators and discriminators. One of the GANs sees and generates images

from the source distribution while the second sees and generates images from the target distribution. The coupled GANs learn the joint distribution of the source and target datasets by sharing the high level latent space. Since the high level representations are shared between domains, a source classifier trained on top of the discriminator's last hidden layer will perform well when changing the source GAN for the target GAN, and thus domain adaptation is achieved.

An encoder-decoder architecture is used in [92]. A classifier is added after the encoder and a cross-entropy loss is minimized on the labeled source data by adjusting the encoder and classifier weights. Simultaneously, the target data is passed through the encoder and decoder with the objective of reconstructing the target images. A pixel-level MSE loss between target and generated images is minimized by adjusting the weights of the encoder and decoder. Since the encoder is shared between the two losses, the representations will be focused on the common aspects of the two domains and therefore the classification of the target domain can be performed using these representations.

The idea of training a classifier on top of a GAN discriminator's last hidden layer, seen in [91], is reused in [93]. This time, a single conditional GAN is trained to generate pairs images according to the source and target distributions using two copies of a noise vector with each domain label appended. The high level features are regularized to be similar across the pairs of images. During inference the discriminator is used with the extra classifier on top, which was trained simultaneously.

In [94], a feature extractor produces features for source and target images. A GAN takes the source and target features as inputs and is optimized to generate images that always resemble the source images. This forces the feature extractor to produce similar features for source and target images. Simultaneously, through an alternated update, a classifier is also trained on top of the feature extractor using the labeled source data. This classifier performs well on target data since the representations are similar.

Other approaches include using GANs to directly map images from the source distribution to the target distribution and training a classifier in the mapped source images with the known labels. This image-to-image task can be achieved using a CycleGAN [69], which uses one GAN for the direct mapping, another GAN for the inverse mapping, and a cycle consistency loss between direct and inverse mappings. Using semantic segmentation labels, [95] achieves even better visual results. However, the results achieved by the best domain mapping techniques still fall very short of the state-of-the art in-domain techniques. On the GTAV to Cityscapes domain adaptation instance segmentation task, [95] achieves a mIoU of 37.43% while the state-of-the-art method training directly on Cityscapes achieves 85.1% [96].

Batch Normalization based Domain Adaptation

Finally, batch normalization based approaches use a single CNN network with batch normalization layers.

In [97], a CNN with batch normalization layers is trained on the labeled source dataset. At test time, the learned batch normalization parameters of the network are replaced by those calculated with target images, by calculating a running average over multiple batches of target images. This procedure aligns

the inner representations of the target and source distributions.

This idea is extended in [98], where a learned parameter is used to optimize the degree of alignment between source and target datasets, allowing for different levels of alignment at different depths of the network.

2.6.2 Domain Generalization

Domain generalization (DG) differs from domain adaptation because no target data is ever used during the training phase. Instead, the methods focus on finding ways to leverage multi-source labeled data to generalize to new target domains. In a concrete example, it is expected that a visual perception module for self-driving vehicles works in any city and weather condition, even if the training dataset does not include every city, weather condition and their combinations. The module should therefore learn to generalize to any other scenario.

The single-source domain generalization task is considered to be the worst-case scenario for generalization. With a single domain to train a network there is no way for an algorithm to learn what features are domain invariant and which ones are domain specific. For instance, if an algorithm is being trained in a simulated environment, it must learn to generalize to reality from that single domain. This is usually achieved by introducing stochastic augmentations to the images, which artificially shifts the domain. This will force a network to generalize across domain shifts and therefore, if the augmentations are properly chosen, it should be able to generalize to the unknown target domains.

Multi-Source Domain Generalization

A first approach to solve the multi-source DG problem is to simply train a network in all source domains and hoping that it can generalize to new domains. This is shown to work in [99].

Other approaches use similar techniques to the one used in domain adaptation. For example, [100] uses a multitask auto-encoder, similar to [92]. There is a single encoder and a decoder for every domain. The first training phase consists of minimizing all the in-domain and cross-domain L_2 reconstruction errors. A classifier is then added on top of the encoder and trains on all labeled domains. The resulting encoder-classifier can also classify other new domains since the encoder produces domain invariant features. In [101], a MDD metric is used to align the feature distribution across every pair of source domains, achieving domain invariant features.

A different approach [102], based on the meta-learning framework of [103], regularizes the feature space of a CNN by promoting domain-independent class cohesion and class-specific separation with a learned distance metric. This method uses a two step update where the first step temporarily updates the parameters using a classification loss on a meta-training batch. In the second step, a meta-testing batch is used and the features are regularized with local and global class divergence losses. The gradient is back-propagated through the two steps to effectively update the model parameters.

Single-Source Domain Generalization

In [104], single-source generalization is achieved with self-supervised learning by solving jigsaw puzzles. The jigsaw puzzles are produced by splitting the images in grid tiles and randomly permuting the tiles. The network must correctly classify the original images and correctly predict the permutation of the altered images using the same feature extractor. It is argued that the task of predicting random permutations requires a higher level of abstraction and generalization than simply classifying images because it requires the understanding of object shape while image classification can be biased towards textures [75]. This idea is further developed in [105] by randomly shifting the domain of each tile using a fast style transfer technique [71]. This procedure further impedes the network from using domain specific cues, like textures, to solve the jigsaw puzzles, and therefore the learned features will be domain invariant.

In [106], besides a standard CNN, an extra Wasserstein Auto-Encoder (WAE) is used to produce domain shifts of the source dataset. The WAE is pretrained on the source dataset. A two step update is used where the first step computes the classification loss of the CNN on a batch of images. The second step alters the input batch adversarially. The second step simultaneously increases the task loss, without changing the classifier prediction, and increases the reconstruction error of the WAE, while moving the least possible in the latent representation of the WAE. Finally the weights of the CNN and WAE are updated using both the unaltered and altered images. This generates challenging domain shifts that will train the CNN to achieve better generalization results.

Global representations are learned in [107] by penalizing local predictive power using a secondary shallow network appended to the early layers of a CNN to predict the class label. The penalization is achieved by reversing the gradient of the secondary network's loss, in a similar manner to [88].

The method in [108] performs simulation-to-real generalization using a pyramid pooling consistency loss that assures feature invariance to random image stylizations and crops. When transferring from a simulated world to a real world, domain randomization can also be performed by artificially tinkering with the objects, textures and environments in the simulated world, as is done by [109].

A very simple approach [110] performs domain randomization with a single convolution applied to the source images. The weights of the convolution are sampled randomly from a normal distribution. A consistency loss based on the Kullback-Liebler (KL) divergence ensures that random variations of the same image produce similar activations in the feature space, thus promoting domain invariant features. This shows that simple domain augmentation techniques may successfully be used to increase generalization performance.

Relation to this work

A recent massive experiment [111] performed by the Facebook AI Research (FAIR) group tested many single-source and multi-source domain generalization algorithms and model selection (validation) strategies using the same network architecture across different domain generalization benchmark datasets. It was concluded that a vanilla empirical risk minimization (ERM) algorithm, when trained with a good data augmentation strategy can generalize just as well or even better than most algorithms purposely built for

domain generalization. It was also found that the model selection strategy was far more impactful than the choice of algorithm. This indicates that the generalization performance is strongly dependent on the network's capacity, which can be maximally used with simple data augmentation techniques, suggesting that the key to achieving good domain generalization results can simply be a good data augmentation strategy.

This work continues the research effort on domain generalization by randomized domain shifting and data augmentation. Different forms of random stylization and texturization are explored, which could potentially be effective in reducing the texture bias of CNNs and improve their effectiveness in domain generalization tasks.

2.7 Applications in Aeronautics and Space

As was mentioned in section 1.1, some actual efforts are already being put into practice by commercial aircraft manufacturers to automate certain functions of their aircraft using computer vision models [2]. These models are used to identify lines in the runway and taxiway such that taxi and take-off can be performed with no pilot input. This kind of approach requires building a dataset by manually labelling the lines in the images such that a model can be trained to identify them. An alternative would be to train the model in an aircraft simulator where the images could be labeled automatically. This could speed up the process but it could also introduce a considerable domain shift between simulator and reality, which would hinder the performance of the model. Since identifying lines is a rather simple task for a computer vision model, only a relatively small number of images need to be labeled and therefore a manual, computer assisted, labeling process is preferred to avoid issues with generalization.

A second, more promising, application of simulator-based learning is in the earth-observation and defense sectors. An important task in these sectors is the automatic processing of satellite images to identify bodies of water, cloud formations, to map terrain, identify buildings, roads and runways, vehicles and even people. Currently, satellite images are processed using deep semantic segmentation models [112–116], which require large amounts of labeled data to be trained. The labeling process is usually computer assisted, but still manual. Since this is a segmentation task, every pixel of interest should be labeled correctly. An alternative would be to train these models with artificially generated satellite images, where each pixel can be labeled automatically. This would introduce a domain shift, depending on the quality of the artificially generated images. Having algorithms that are more robust to domain shifts would greatly improve the reliability of this approach. This would allow the training of satellite image segmentation models with broader capabilities and better performance, using massive automatically labeled image datasets. Furthermore, such robust algorithms would also perform better even without training in a big generated dataset. Since they are robust to domain shifts, they would perform better across different meteorological conditions, lighting, cameras and overall image condition.

A final application where simulator-based learning will most likely be a key requirement is for manned urban autonomous aerial vehicles. Such vehicles will have to include complex vision systems that can detect buildings, trees, other aerial vehicles, birds, cables, bridges and proper landing spaces. Building

an acceptable dataset to train this kind of visual system is a very difficult task, since multiple aircraft would have to exist already and be used to capture a very large amount of training data, in multiple scenarios, with different meteorological conditions. To build a certifiable system, the dataset would also have to include multiple imminent failure and complete failure scenarios, of the aircraft itself and of other aircraft, as well as all kind of other perturbations. This is the case with the datasets used to train self-driving cars. In the case of Tesla, extensive amounts of data, including multiple accident scenarios, had been collected before a first self-driving prototype was developed. This was only possible because Tesla had already sold thousands of vehicles, which silently collected data while being driven by their users. This is not an option for an aerial vehicle since they cannot be sold and used by the mass market until they are autonomous. Furthermore, aircraft accidents will be far less common, so the training data for these situations will not exist in abundance. It should now be evident that developing such a vision system for an aerial vehicle requires a simulator-based approach to collect the necessary data, in all the required scenarios, such that the vehicle can be certifiable. This would allow the testing of the vision, the planning and GNC (Guidance, Navigation and Control) systems of the aircraft within a simulated environment, to ensure the aircraft will always respond in the expected manner in any scenario. The missing link to this approach is the development of vision algorithms that are truly capable of generalizing from simulator to reality, ensuring an equal performance in both domains while only being trained in the simulated environment.

Other applications of computer vision in the aerospace sector might also benefit from an increased robustness and better generalization performance of neural networks.

One example is the quality control of aircraft and spacecraft structural components. Techniques such as Automatic Fiber Placement (AFP) [117] are being adopted as a fast and efficient method to build large structural components out of carbon fiber. These techniques often lead to defects such as wrinkles, undesired fiber tension and compression [118]. These process-induced defects often create uncertainty in the mechanical performance of the resulting components. This requires the process to be supervised so that the fiber placement can be corrected when defects appear. This process is usually performed by experts and is often time consuming. A good solution to this problem is to use machine learning and computer vision techniques to perform rapid inspection of the carbon-fiber sections while they are being built, so that they can be automatically corrected [119, 120].

UAV (Unmanned Aerial Vehicle) operations will also benefit from an increase in the robustness of computer vision algorithms that may be used to automate certain functions of the aircraft [121, 122]. As an example, UAV navigation can be performed using computer vision algorithms [123], by identifying landmarks and using them as a guidance mechanism. High-definition images are used as way-points. The UAVs identify these way-points during flight and use them to perform course corrections. Another example is the use of UAVs for urban search and rescue missions [124]. Navigating through disaster struck environments is a challenging problem since no accurate mapping of the environment exists. This requires the problem to be solved using computer visions algorithms which capture the 3D geometry of the environments so that a drone can safely navigate through them and safely land on areas full of debris. These application of computer vision to UAV operations require algorithms to be extremely robust before

they can be relied upon.

All these examples show the importance of simulator-based learning and of developing robust algorithms that can generalize across domains. These algorithms will be of extreme importance in the upcoming years to solve multiple problems in the aeronautical and space sectors.

Chapter 3

Domain Shifting Experiments

3.1 Domain Randomization via Style Transfer

The first proposed approach to achieve randomized domain shifts is by adapting the style transfer technique described in section 2.4.5. We start by analysing the baseline technique as described in [71]. We follow by analysing the different steps required to train a random stylization network without the need for a style dataset.

3.1.1 AdaIN Baseline

The baseline method achieves considerably great results, with overall good reconstruction, color matching, and pattern transfer, as seen in Figure 3.1.

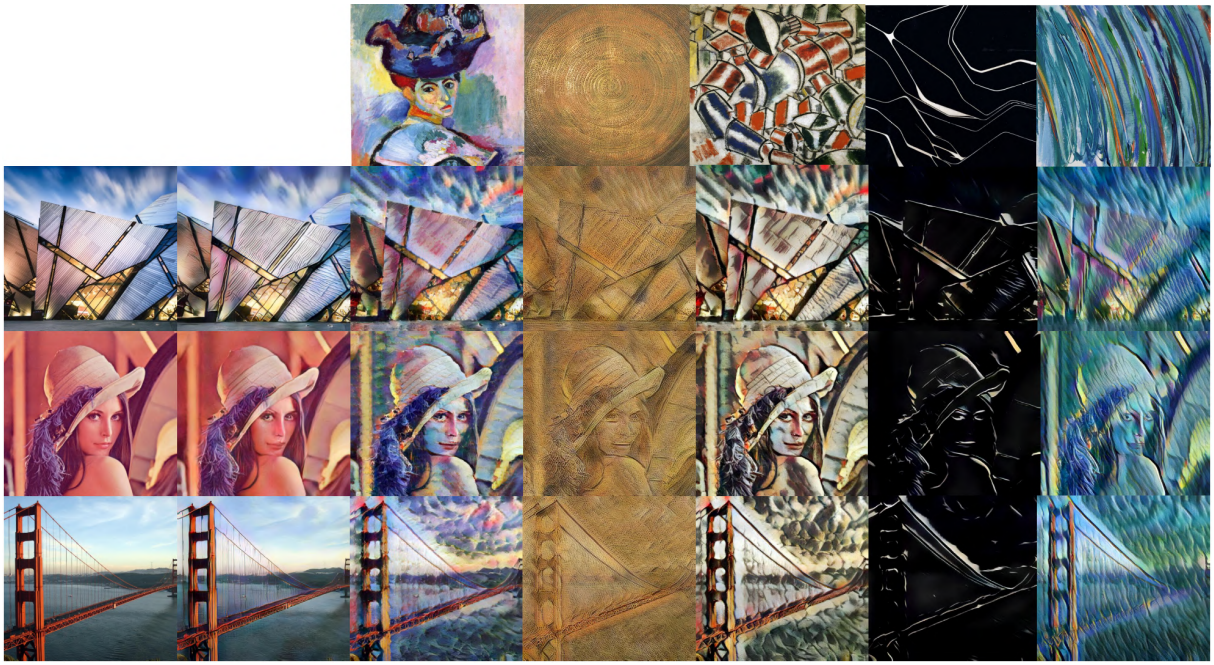


Figure 3.1: Baseline style transfer examples. Content images in the first column. Unaltered image reconstruction in second column. Style images and respective stylised images in the following columns.

The pretrained baseline method can be used as-is to produce random stylizations of content images without the need to input a style image, as was demonstrated in [125].

In the baseline method, the style images are introduced by calculating their values of μ_c and σ_c, the mean and standard deviations of each feature map in the last layer of the encoder, and applying them to the content image using AdaIN. Those values of μ_c and σ_c can be replaced by random values hence creating the novel styles in Figure 3.2. These values are sampled randomly from normal distributions $\mu_c \sim \mathcal{N}(0,1)$ and $\sigma_c \sim \mathcal{N}(1,1)$. Even though negative values for σ_c would not naturally appear, the AdaIN algorithm can make use of them just as well. Results of random stylization can be seen in Figure 3.2.

The problem of using this approach is that the stylizations are biased towards the style dataset on which the network was trained, in this case, paintings from the Wikiart. The network decoder has learned to generate images that have the style of those paintings and therefore cannot produce images with different statistics and different kinds of textures.

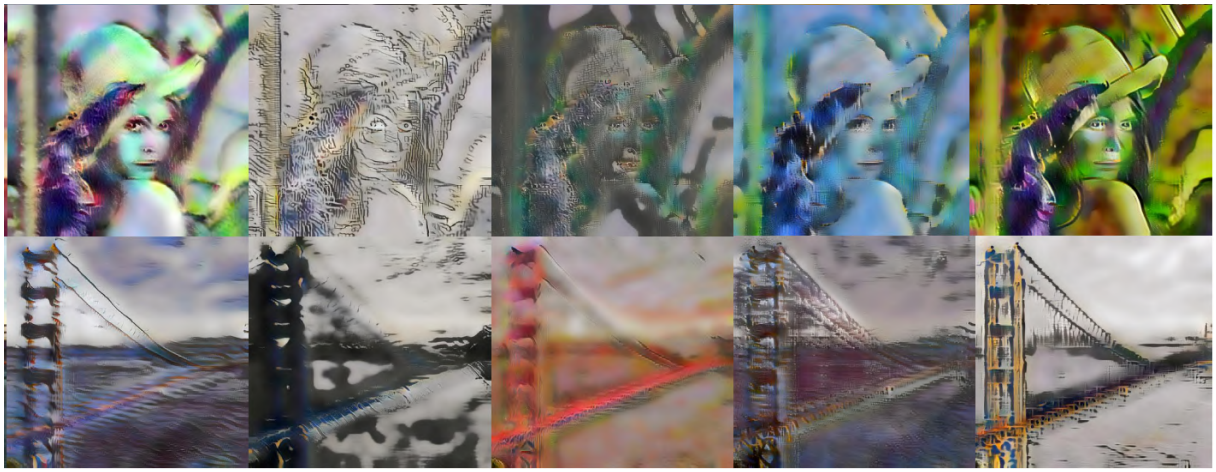


Figure 3.2: Random style transfer examples. Content images in the first column. Unaltered image reconstruction in second column. Randomly stylised images in the following columns.

3.1.2 Covariance Adapter

The introduction of random values for μ_c and σ_c assumes that the distribution of the feature maps produced by the encoder is known. The sampling distributions of μ_c and σ_c should be the same as the distributions expected when passing a style image through the decoder. To test this constraint, histograms of the feature maps are produced, permitting the analysis of the distributions generated by the encoder on a single image:

Baseline

The distributions of the feature maps are plotted for two different images, the first one with a high diversity of textures and colors, the second with a single almost homogeneous texture. See Figure 3.3.

It is noticeable that, due to the ReLU layer at the end of the encoder, all feature map activations are

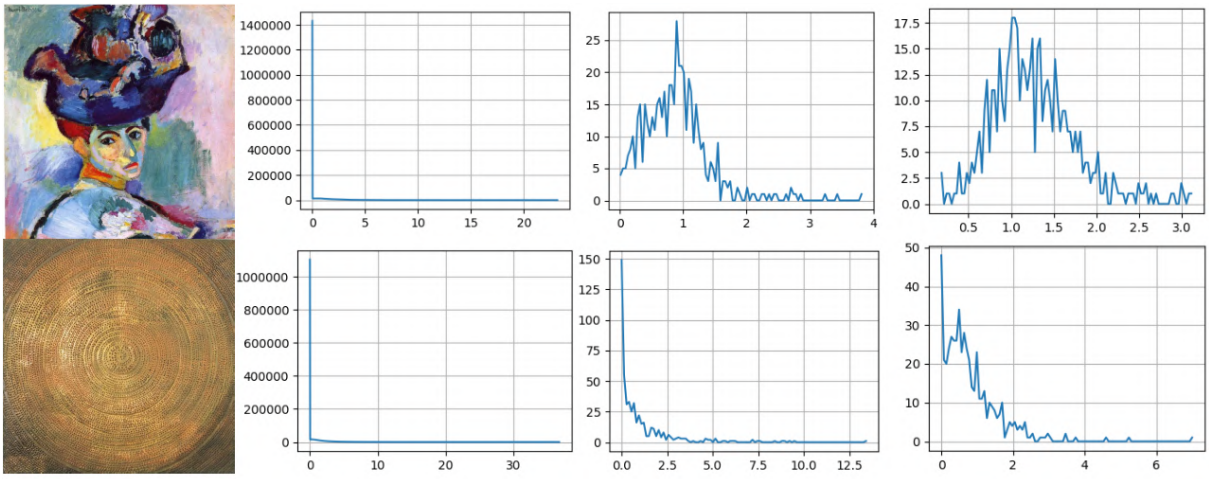


Figure 3.3: Histograms produced by the baseline encoder. a) Style Images; b) 1000 Bin Histogram of $\phi_c^{\mathbf{s}}(i,j)$; c) 100 Bin Histogram of $\mu_c(\phi_c^{\mathbf{s}})$; d) 100 Bin Histogram of $\sigma_c(\phi_c^{\mathbf{s}})$;

positive, creating a strong peak at 0. This might not pose any problems but, since we want to create a latent space of styles, parameterized by $\mu_c \sim \mathcal{N}(0,1)$ and $\sigma_c \sim \mathcal{N}(1,1)$, another experience is performed by removing the last ReLU layer of the encoder network.

Baseline without ReLU

The histograms produced by the encoder without the last ReLU layer are much more bell-like, and so, the distributions of the feature maps are closer in shape to the target distributions of $\mu_c \sim \mathcal{N}(0,1)$ and $\sigma_c \sim \mathcal{N}(1,1)$. See Figure 3.4. As expected, a more diverse painting will generate a bigger variance in the feature maps. From now on, all experiences will be made without the last ReLU layer of the encoder unless otherwise specified.

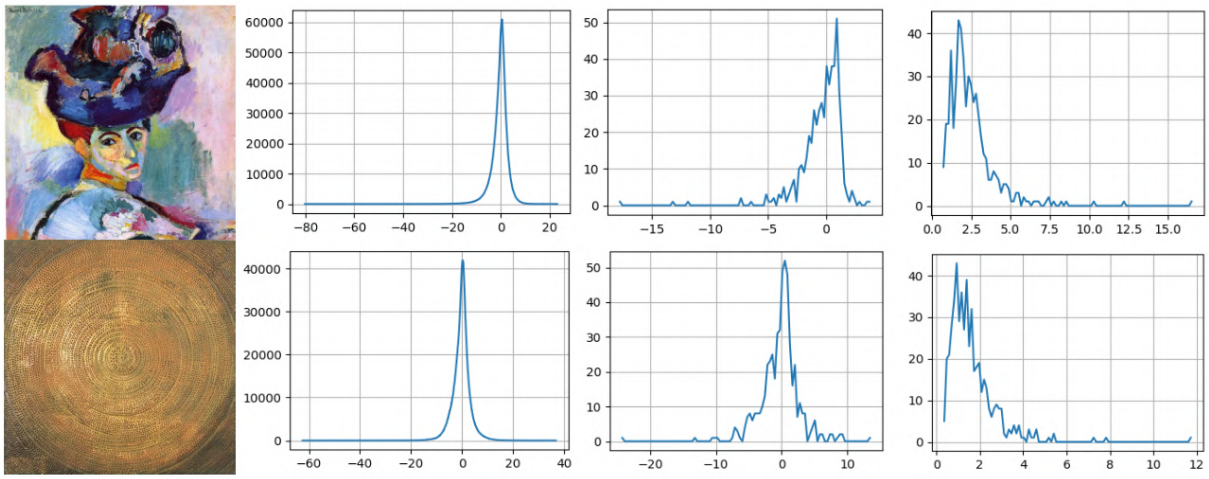


Figure 3.4: Histograms produced by the baseline encoder without the last ReLu layer. a) Style Images; b) 1000 Bin Histogram of $\phi_c^{\mathbf{s}}(i,j)$; c) 100 Bin Histogram of $\mu_c(\phi_c^{\mathbf{s}})$; d) 100 Bin Histogram of $\sigma_c(\phi_c^{\mathbf{s}})$;

Naive Adapter

In the next step we want to make sure that each feature map has a distribution with mean and variance as close as possible to 0 and 1 respectively. Normally one would achieve this by normalizing the feature maps with either instance normalization or batch normalization. The problem with instance normalization is that it renders the following Adaptive Instance Normalization completely useless since all content and style features will be normalized already and no style transfer would happen. Batch normalization would work but it introduces a lot of noise at low batch sizes since the feature map statistics would now be dependent on the other images in the batch. The proposed solution is to cap the encoder with a single convolutional layer adapter with a 1x1 kernel. This is equivalent to performing a linear transformation of the feature maps with learned weights. The outputs of the adapter are constrained to have mean and variance as close as possible to 0 and 1 with a KL divergence loss. Contrary to the other methods, this will keep the information in the statistics of the feature maps without introducing noise. Assuming that the feature maps are normally distributed, the loss of the adapter is given by:

$$\mathcal{L}_a = \frac{1}{C}\sum_{c}^{C} D_{KL}(\mathcal{N}(\mu_c, \sigma_c) \,||\, \mathcal{N}(0,1)) = \frac{1}{C}\sum_{c}^{C} \frac{1}{2}(\sigma_c^2 + \mu_c^2 - ln(\sigma_c^2) - 1) \tag{3.1}$$

This loss is minimal when the two normal distributions have the same parameters. This might seem like a good solution, however, there is no guarantee that the feature maps produced by the adapter are independent. It is entirely possible that the adapter produces multiple copies of the same feature map, with a very poorly conditioned linear transformation. The next experiment solves this issue by constraining the covariance matrix between channels of the feature maps to be close to the identity matrix.

Covariance Adapter

This covariance matrix is commonly called the Gram matrix G of the feature maps [49] and has dimensions $c \times c$:

$$G_{c'c} = \sum_{i}^{H}\sum_{j}^{W} \phi_{c'}(i,j)\phi_c(i,j) \tag{3.2}$$

A mean squared error term between the Gram matrix and the identity is added to the previous adapter loss.

$$\mathcal{L}_a = \frac{1}{2C}\sum_{c}^{C}(\sigma_c^2 + \mu_c^2 - ln(\sigma_c^2) - 1) + \frac{1}{C^2}\sum_{c'}^{C}\sum_{c}^{C}(G_{c'c} - I_{c'c})^2 \tag{3.3}$$

The results are show in Figure 3.5.

Although not noticeable in the results, this produces feature maps that are sure to be as orthogonal as possible from each other. We are therefore able to introduce the parameters μ_c and σ_C from an i.i.d. normal distribution. The next section will show the results of this procedure.

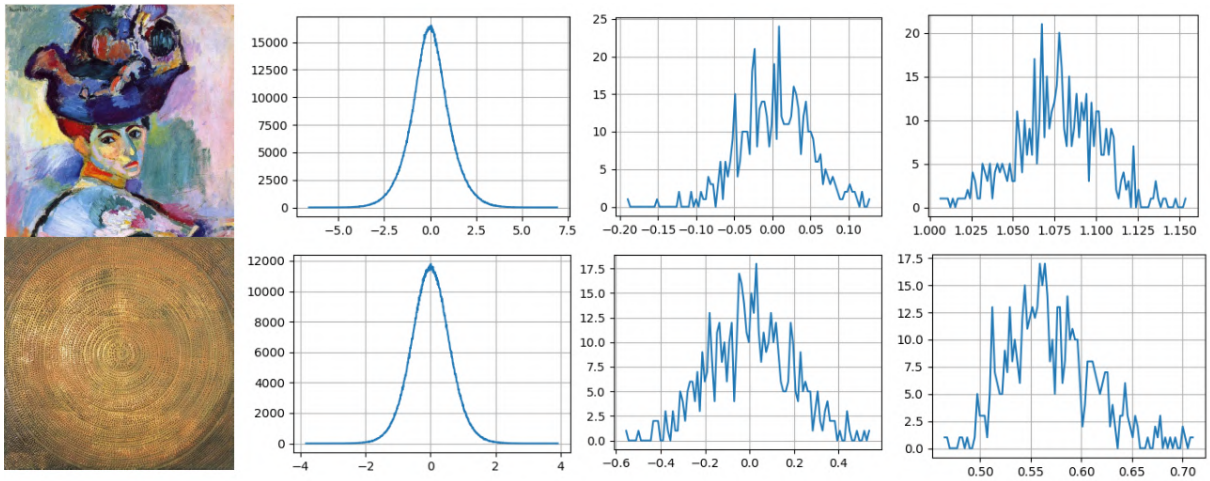


Figure 3.5: Histograms produced by the encoder followed by a KL divergence + Gram matrix adapter. a) Style Images; b) 1000 Bin Histogram of $\phi_c^{\mathbf{s}}(i,j)$; c) 100 Bin Histogram of $\mu_c(\phi_c^{\mathbf{s}})$; d) 100 Bin Histogram of $\sigma_c(\phi_c^{\mathbf{s}})$;

3.1.3 AdaIN + Adapter trained with style dataset

When training the full style transfer model we have the option of training the adapter first and then the decoder, or to train them both simultaneously. The first two experiments will test these two options with the full multi-layered style loss. The third experiment will have a style loss that is only calculated in the last layer. This is because, in the following experiments, where the style dataset is replaced by the random values of μ_c and σ_c, there is no style image to produce the middle encoder activations and the loss is calculated only with the parameters μ_c and σ_c directly.

Adapter and Decoder trained separately

Training the adapter and decoder separately allows the adapter to produce feature maps as uncorrelated as possible without interference from the decoder. The results are quite similar to the original method. Reconstruction artifacts are visible, which shows that this type of adapter isn't ideal. Figure 3.6 shows examples of style transfer and Figure 3.7 shows examples of random stylization using this technique. The adapter was trained for 15k iterations and the decoder for 65k iterations.

Adapter and Decoder trained together

Training the adapter and decoder together makes both networks compete to achieve both of their objectives. The independence of the feature maps is therefore worst. The big advantage with this approach is that it allows the training to happen in a single session. Figure 3.8 and Figure 3.9 show examples of style transfer and random stylization after 85k iterations.

The reconstruction is still lacking in detail. The patterns produced are slightly different than those produced when training separately. It is considered that training the adapter and decoder together poses no significant detriment to the results when compared with the separate training approach.

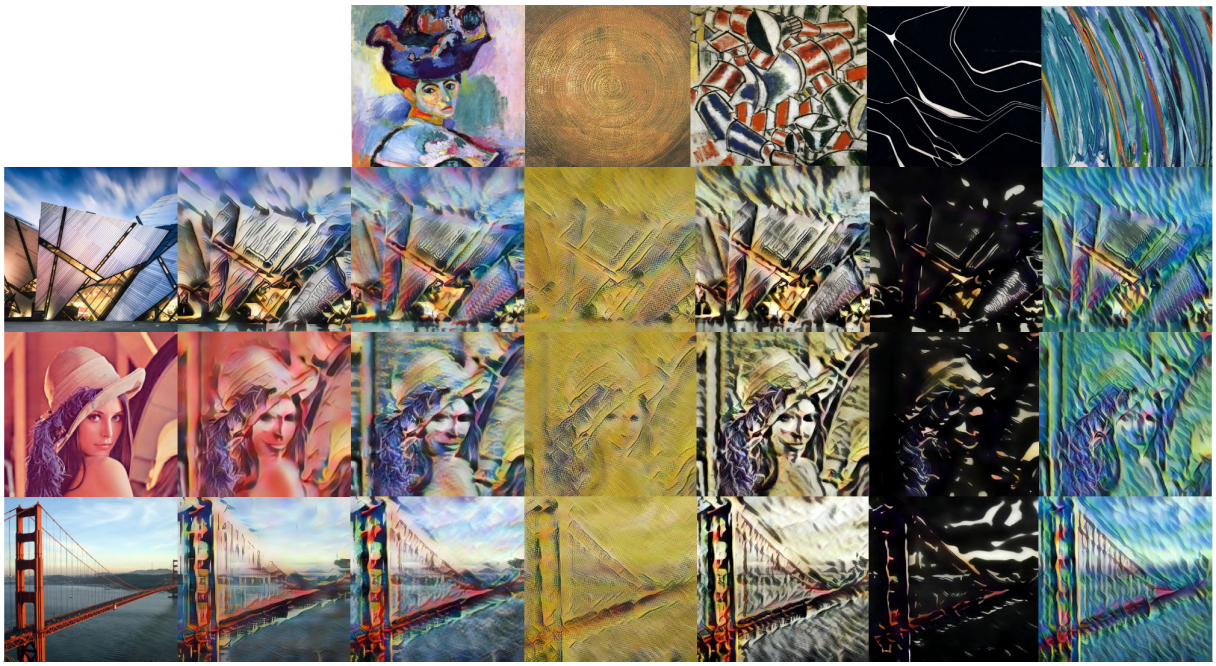


Figure 3.6: Style Transfer Examples with an adapter and decoder trained separately.

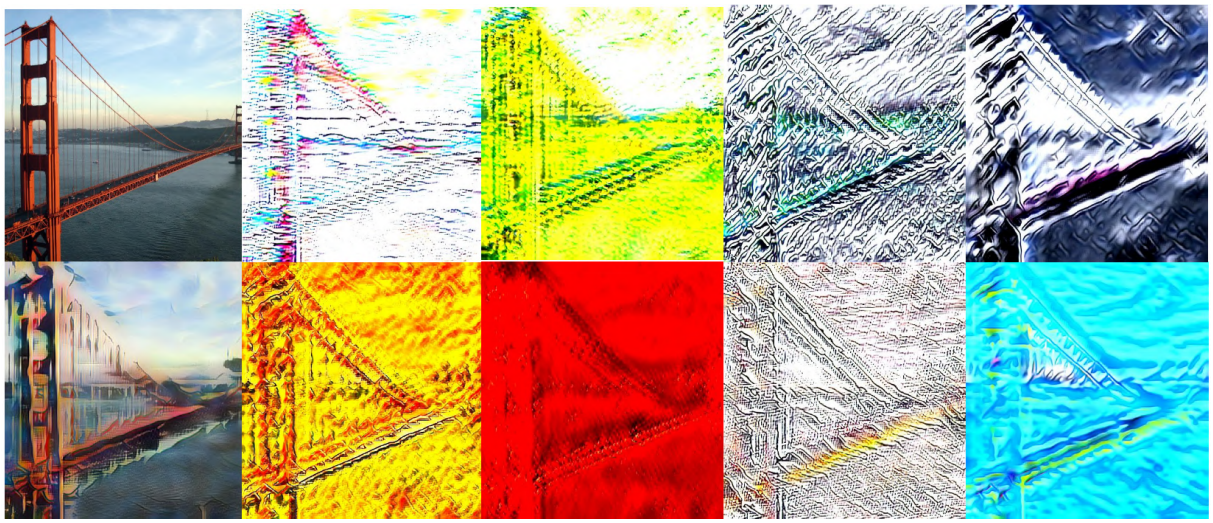


Figure 3.7: Random Style Examples with an adapter and decoder trained separately. The first column contains the input image (top) and reconstructed image (bottom).

Last Layer Loss Only

To simulate the use of random parameters μ_c and σ_c, the following experiment is performed while calculating the style loss on the last layer only. Examples of style transfer are shown in Figure 3.10

The generated images are of similar quality to the original method in Figure 3.1. There is a slight colour shift and the transferred patterns are slightly smaller in scale. It is concluded that using only the last layer to calculate the style loss should not pose any major quality problems when training without a style dataset.

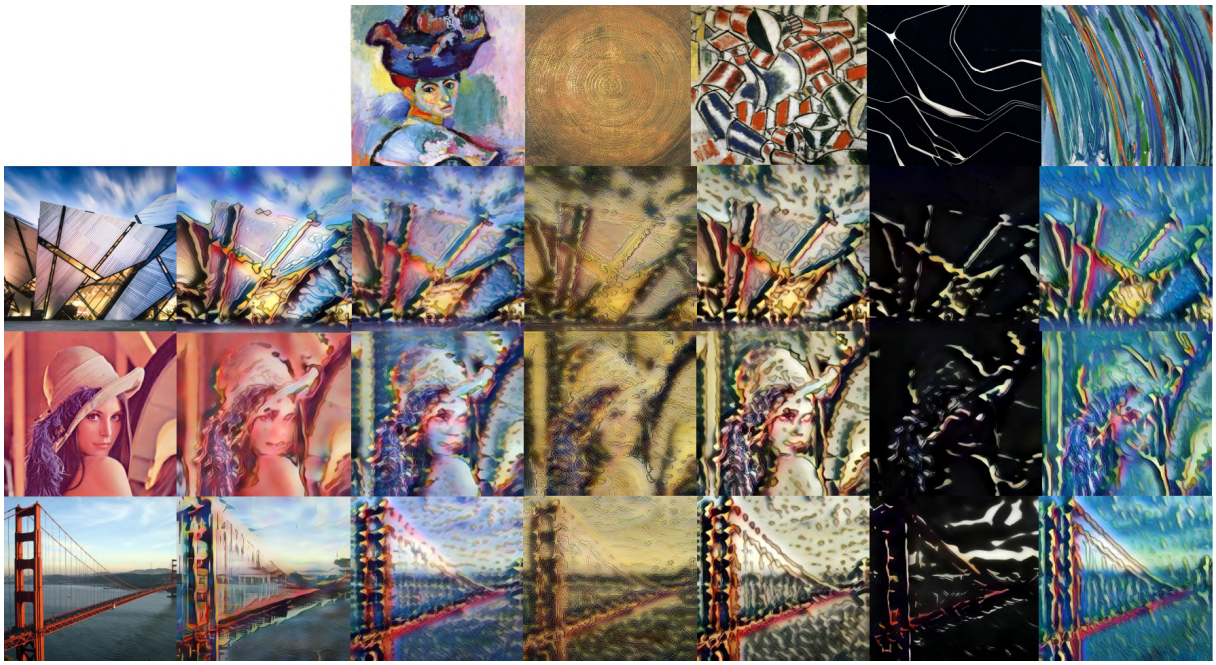


Figure 3.8: Style Transfer Examples with an adapter and decoder trained together.

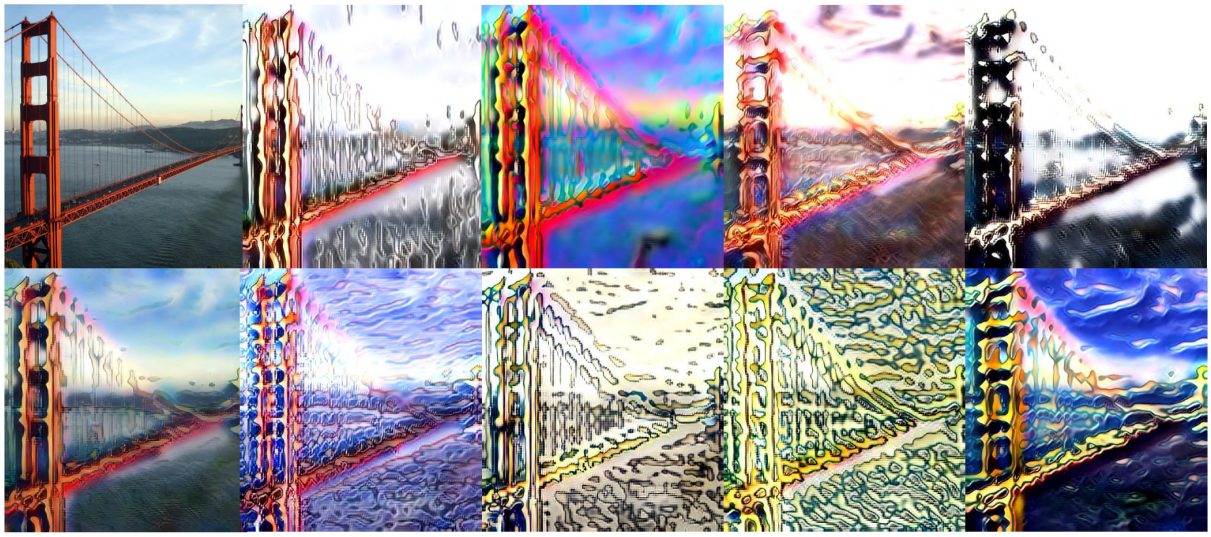


Figure 3.9: Random Style Examples with an adapter and decoder trained together. The first column contains the input image (top) and reconstructed image (bottom).

3.1.4 AdaIN + Adapter trained without style dataset

We are now prepared to train the full stylization network, with an adapter after the encoder, random μ_c and σ_c parameters and a last layer style loss. The results are shown in Figure 3.11 and Figure 3.12.

The reconstruction is almost perfect besides the color shift. The major problem is the pattern transfer and generation. Both style transfer and random style images are subject to a colour shift, with only very small scale patterns being generated.

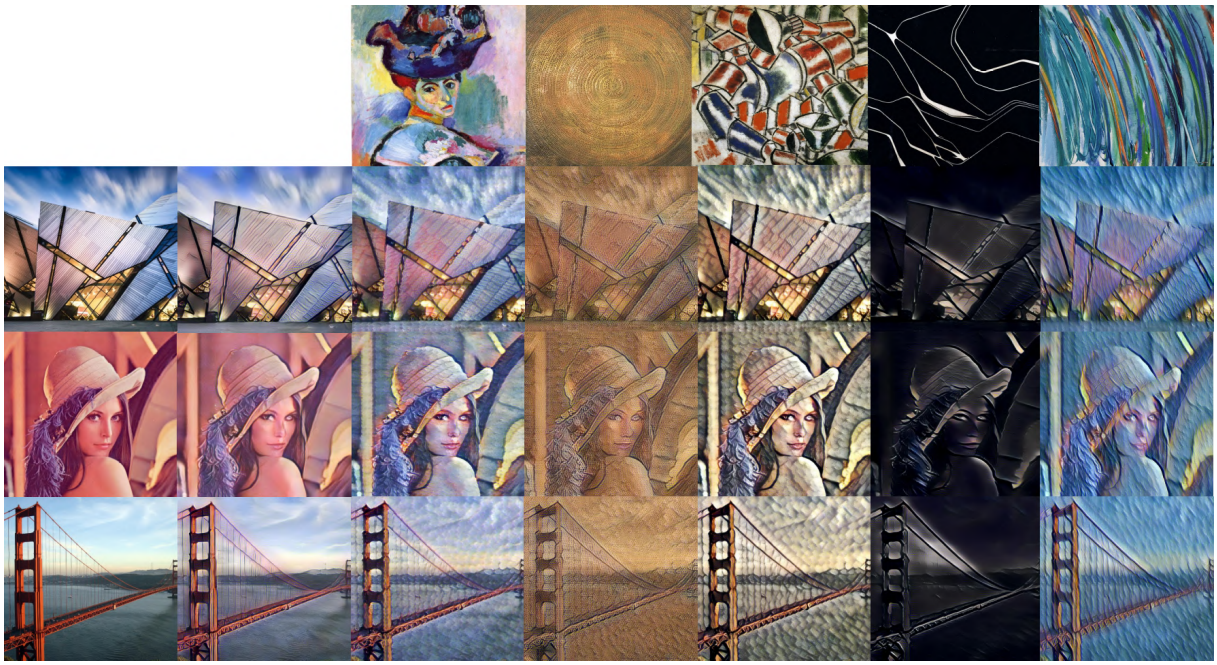


Figure 3.10: Style Transfer Examples with last layer style loss only. Trained for 50k iterations.

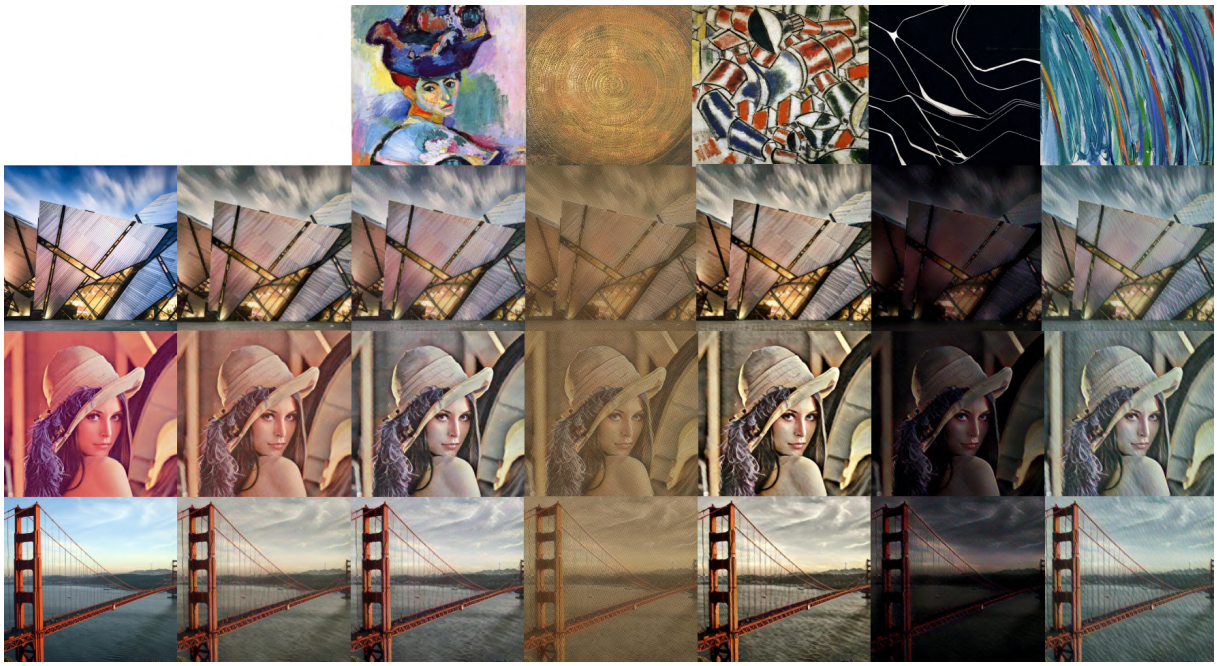


Figure 3.11: Style Transfer Examples. Trained without style dataset for 105k iterations.

3.1.5 Adaptive Instance Normalization Conclusions

The previous experiments have shown that a great diversity of styles can be achieved when training the style transfer network with a style dataset. This diversity is preserved when a feature adapter is introduced and the style loss is only calculated in the last layer.

This is not the case when the network is trained without a style dataset, only minor small-scale patterns are transferred and the random stylization only affects the color balance of the images, without great diversity and barely any created patterns. We are forced to conclude that the emergence of

Figure 3.12: Random Style Examples. Trained without style dataset for 105k iterations.

complex patterns is a result of using a style dataset. The high level feature correlations cannot be captured with independently varying normal distributions of parameters.

An auxiliary network that generates the parameters μ_c and σ_c from a lower dimensional noise was also tested. This network induces more correlations between these parameters but it still isn't capable of generating the specific patterns seen in common style transfer methods that use a style dataset.

When training with a style dataset, the decoder network learns to generate patterns like brushstrokes, lines and curves, because those are very prominent in the style dataset. These complex patterns can be considered "high-level" patterns, which can only occur with specific feature map correlations. Therefore, when training without a style dataset, these correlations are very unlikely and the complex patterns never form. Only "low-level" homogeneous patterns like colour and small scale features will ever form. Like electrons filling the orbitals of an atom, the low energy states are filled first. It is possible that increasing the model capacity may open the door to more complex patterns, although, it is suspected that the width of the model varies exponentially with pattern complexity. Thus, the creation of complex patterns would require a prohibitively big network. This network would have to generate all the less complex patterns before being able to produce more complex patterns. This approach is therefore deemed inappropriate for random high level stylization. It could nonetheless still be useful as a data augmentation tool since it is able to generate multiple differently textured versions of the same image.

3.2 Encoder-Transformer-Decoder Networks

Ideally, a random stylization method should be able to reconstruct the input image with a high degree of fidelity when no noise is introduced in the auto-encoder network. The AdaIN stylization method achieves this when the same image is used for content and style, as seen in Figure 3.1, second column. The problem with the AdaIN stylization method is that it requires a big network, usually a VGG16, and the respective mirrored decoder. Furthermore, the styles that it produces are heavily conditioned on the style dataset that it is trained on, and are therefore not general. A data augmentation method should

be lightweight enough to not cause a considerable computational overhead on the main method. It should also be as general as possible, in a sense that, it should produce all possible variations of the transformation that it applies.

With these considerations in mind, a different approach is therefore devised where a simple convolutional auto-encoder is used to learn a latent representation of the textures in an image. The auto-encoder has no information bottleneck. Instead an encoder converts spatial information into channel-wise information using down-sampling with strided convolutions. This means that each pixel in the latent feature maps encodes the texture of a patch in the input image. A decoder based on transposed convolutions is then tasked with reconstructing the input image. A Mean Squared Error (MSE) reconstruction loss is used between the input and output images. An extra KL Divergence loss is used in the latent space to softly constrain the representations to have $\mu = 0$ and $\sigma = 1$. The encoder and decoder can have 2 to 4 layers each, depending on how wide we want the texture patches to be and how non-linear we want the transformation to be. The auto-encoder is trained on a set of content images using these two losses. Since there is no information bottleneck, the auto-encoder should be able to perfectly reproduce the input images and have an MSE loss that tends to 0.

The purpose of the encoder-decoder architecture is not to compress the information of the images. It is rather to encode the images into a texture-space, where they are transformed and then decoded back. Therefore the total volume $(C \times W \times H)$ of the feature maps should remain constant at every layer of the encoder. When a down-sampling is performed, the number of channels should increase by the same ratio i.e. dividing the resolution by 3 in height and width, should be accompanied by a 9 times increase in channel number. The information is therefore encoded channel wise and not pixel wise, creating the aforementioned texture-space.

To perform randomized texturing of images the auto-encoder is frozen and a transformation is introduced in the latent space. The full architecture, presented in Figure 3.13, consists of an encoder E, a transformation T that introduces the noise in the feature maps created by the encoder, and a decoder D that generates the transformed images.

The key element of this architecture is the choice of transformation T, which is responsible for introducing the noise vector z in a meaningful manner. The influence of z on the output feature maps should be similar to that of the input feature maps. This roughly means that $\left|\left|\frac{\partial T}{\partial z}\right|\right|_\infty$ and $\left|\left|\frac{\partial T}{\partial \phi}\right|\right|_\infty$ should have the same order of magnitude. To induce a great variety of styles, the components of $\frac{\partial T}{\partial z}$ should also be as orthogonal as possible, meaning that, each component of z affects the output in a different manner.

3.2.1 Formalization

Let $\hat{x} = F(x, z)$ be a function $\Omega_x^{s \times c} \times \mathbb{R}^n \rightarrow \Omega_x^{s \times c}$, where s represents a spatial dimension, c represents a channel dimension, n represents a noise dimension and $\Omega_x \subset \mathbb{R}$ is the support of the values of x and $\hat{x}$. The function $F(x, z)$ is the composition of an encoder $y = E(x)$, a transformation $\hat{y} = T(y, z)$ and a decoder $\hat{x} = D(\hat{y})$ with $y \in \Omega_y^{s' \times c'}$ and $\hat{y} \in \Omega_y^{s' \times c'}$, where $\Omega_y \subset \mathbb{R}$ is the support of the values of y and $\hat{y}$. The encoder E encodes the information contained in the spatial dimension into the channel dimension

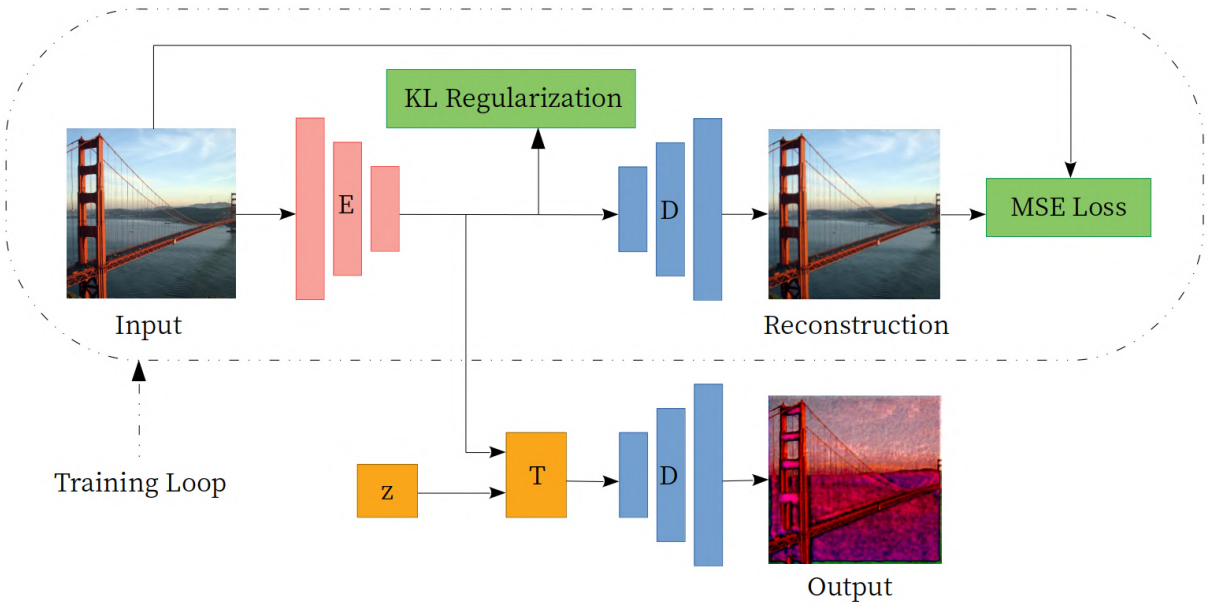


Figure 3.13: Encoder-Transformer-Decoder architecture. During training, the Encoder (E) and Decoder (D) are trained together, using the MSE Loss and a KL Divergence regularizer. At inference time, a Transformer (T) transforms the latent space, according to a noise vector z, before decoding the image.

and the decoder D decodes the information from the channel dimension back into the spatial dimension. We have therefore $s' < s$ and $c' > c$. To ensure that no information bottleneck exists we set s' and c' such that $s' \times c' = s \times c$.

Let the encoder and decoder be the inversion of each other when the transformation is bypassed, which is the case when the auto-encoder is fully trained:

$$D(E(x)) = x, \forall x \in \Omega_x^{s \times c} \tag{3.4}$$

Since both the encoder and decoder are invertible, they must be bijective. Furthermore, Ω_x and Ω_y are isomorphic to each other. When the transformation $T(y, z)$ is introduced, if the transformation is bijective for any given z, then the function $F(x, z)$ must also be bijective for any given z. This means that all the spatial information in x must be encoded in y, $\hat{y}$ and $\hat{x}$, for a known z.

The input x can be thought of as any type of single or multi-channel data with temporal or spatial components (pixels) with a certain distribution $x \sim X$. Typically in natural data, the intrinsic dimensionality of this data is far lower than the total possible dimensionality of X. This means that the probability density of X is zero almost everywhere. A domain expansion consists of producing a distribution $\hat{X}$ with a bigger intrinsic dimension than the original distribution X.

Consider that $T(y, z)$ is a per-pixel transformation $T_i(y, z) = t(y_i, z)$. We want the transformation t to map a given pixel vector of y, notated $y_i \in \Omega_y^{c'}$, to any other given pixel vector $\hat{y}_i \in \Omega_y^{c'}$, for a certain value of z. More concretely:

$$\forall y_i \forall \hat{y}_i \exists z : t(y_i, z) = \hat{y}_i \tag{3.5}$$

This operation effectively maximises the intrinsic channel-wise dimensionality of $\hat{y} \sim \hat{Y}$ and therefore, since D is bijective, the dimensionality of $\hat{X}$ is also increased. This condition may be relaxed to decrease the degree of variation of the outputs while still ensuring a domain expansion:

$$\forall y_i \forall \hat{y}_i \in \mathcal{B}_\epsilon(y_i) \exists z : t(y_i, z) = \hat{y}_i \tag{3.6}$$

where $\mathcal{B}_\epsilon(y_i)$ denotes a ball of radius ϵ around y_i and ϵ controls the magnitude of variation of the outputs.

The distribution Z of z and the choice of transformation t will define the diversity observed in the outputs of $F(x, z)$. A first choice of t might be a simple translation:

$$t(y_i, z) = y_i + z \tag{3.7}$$

If $z \sim Z$ comes from an uniform distribution $Z = U_{\mathcal{B}_\epsilon(0)}$ then this transformation satisfies the relaxed condition for the maximisation of the dimensionality of $\hat{y}$. If z is sampled from a normal distribution, $Z = \mathcal{N}(\mu, \sigma)$, then the transformation satisfies the first unrelaxed condition for the maximisation of the channel-wise dimensionality.

Some of the spatial information of the input is encoded in the channels of y. Transforming y into $\hat{y}$ with an increase in dimensionality, results in increasing the dimensionality of this spatial information. When the generating the output $\hat{x}$, this will create not only a diversity in each pixel but also a diversity in the inter-pixel relationships. In the case of images, the generated images will suffer not only a diversification of color, but also of texture, according to z. The maximisation of the channel-wise dimensionality ensures that an image patch can be transformed to every possible texture with non-zero probability.

A second option for the transformation t could be a rotation in the texture space. This transformation is parameterized by a orthogonal matrix $z \in \mathbb{R}^{c' \times c'}$:

$$t(y_i, z) = z y_i \tag{3.8}$$

This transformation does not satisfy the first non-relaxed condition to have a domain expansion because during a rotation, the points are kept at a fixed distance from the origin and therefore not every point can be mapped to every other point. The combination of a rotation with a translation however does satisfy the condition. The advantage of the rotation transformation is that it can transform any texture into any other texture, at the same distance from the origin in texture space, with equal probability.

To construct the rotation matrix, we start by sampling the values of z from a standard normal distribution. To ensure that t is bijective for any z, z needs to be a non-singular matrix. This happens on a set with measure 0 so in theory there is no need to introduce a constraint. In practice, we would like to also preserve the orthogonality of features. This means that features that are dissimilar (orthogonal) should be kept dissimilar, so as to not collapse the feature space. The matrix z should thus be constrained to an orthogonal matrix. This can be achieved through an orthogonalization algorithm like the Grand-Schmidt process (QR decomposition), Householder transformation, a Singular Value Decomposition, or a spectral algorithm like Björck's iterative singular value normalization. We chose the QR decomposition as it is

fast to compute and does not involve iterative processes. The matrix z is therefore the result of a random sampling from a normal distribution followed by a QR decomposition which renders it orthogonal.

A third option for the transformation t involves an adaptive normalization of the features, with $z = [z_1, z_2]$ and $z_1, z_2 \in \mathbb{R}^{c'}$. This transformation is applied pixel-wise but depends on the statistics of the other pixels of y:

$$t(y_i, y, z) = \frac{y_i - \mu(y)}{\sigma(y)} \cdot z_1 + z_2 \tag{3.9}$$

where μ and σ are the per-channel means and variances of y. Once again, the non-relaxed condition is satisfied for any y with z normally distributed, therefore this approach maximises the dimensionality of $\hat{Y}$ and increases that of $\hat{X}$. This approach is more closely related to the common method of style transfer seen in the literature.

Any of these channel-wise transformations performs image texturizations that are uniform in space. If we wish to add more textural variety space-wise, an additional noise component can be sampled from a normal distribution and added to each pixel and channel individually.

3.2.2 Encoder-Transformer-Decoder Experiments

The following experiments showcase the type of randomized texturizations that is possible to obtain with the previously mentioned techniques. The encoder has 2 convolutional layers (C) with a Hyperbolic Tangent (Tanh) activation in between. The decoder has 2 transposed convolutional layers (TC) with the same activation in between. The architecture is summarized in Table 3.1.

Layer	Channels In	Channels Out	Kernel Size	Stride	Padding	Activation
Encoder C0	3	27	3×3	3	1	Tanh
Encoder C1	27	243	3×3	3	1	None
Decoder TC0	243	27	3×3	3	1	Tanh
Decoder TC1	27	3	3×3	3	1	None

Table 3.1: Architecture of the Auto-Encoder for random texturization.

This Auto-Encoder is trained on the MS-COCO dataset for 25000 iterations with a batch size of 16 and a learning rate of 1×10^{-4} with Pytorch's default ADAM optimizer. All images are resized to 505×505 pixels during training. The latent feature maps have 243 channels and 57×57 resolution ($((505+2)/3+2)/3$). The two convolutions with a stride of 3 generate latent representations that encode the texture in a 9×9 patch.

Once the Auto-Encoder is fully trained, it is frozen and the transformation module is inserted in between the encoder and decoder networks. At test time, the latent feature maps are altered by the transformation module. The final feature maps are a linear interpolation between the unaltered and altered feature maps, parameterized by a coefficient α. When $\alpha = 0$ the image is unaltered, when $\alpha = 1$ the image is fully altered, and for values between 0 and 1 the image is the result of the interpolation.

Translation

The translation is implemented by sampling a 243-dimensional noise vector z from a normal distribution $\mathcal{N}(\mu = 0, \sigma = 0.5)$ and adding it to each of the pixels of the feature maps. The results for random samples of z are presented in Figure 3.14.

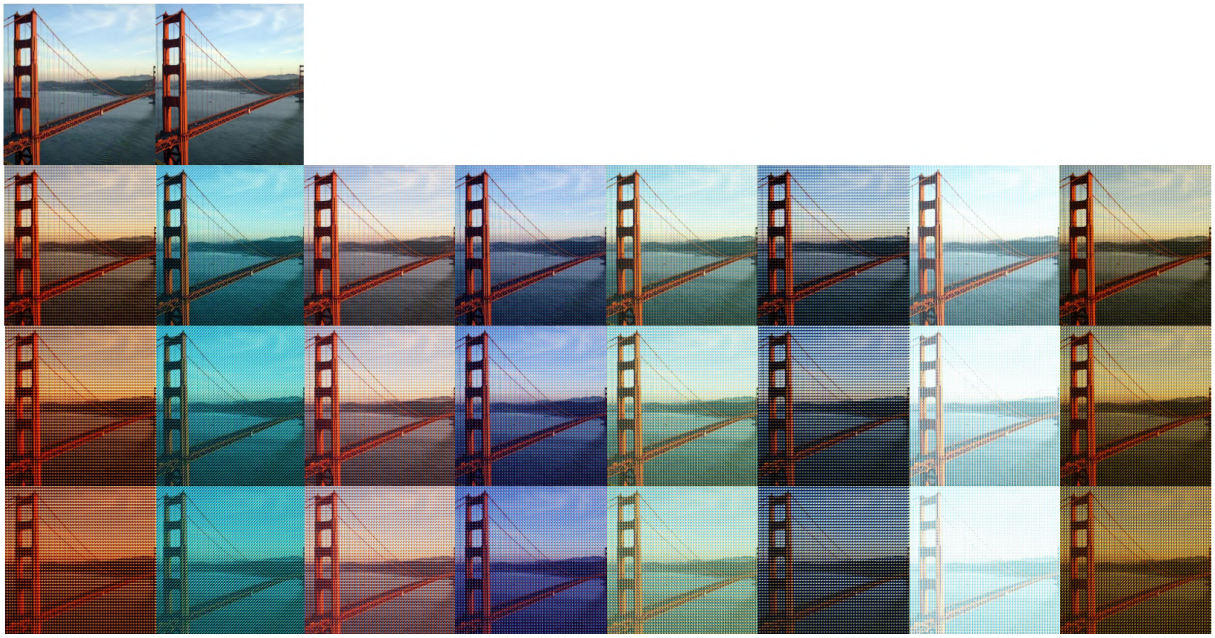


Figure 3.14: Images randomly textured by the Encoder-Transformer-Decoder architecture with translation. First row: reconstructed image and input image. Second row with $\alpha = 0.3$, third row with $\alpha = 0.6$ and final row with $\alpha = 1.0$.

To add more diversity across space, a noise tensor of the same dimensions as the latent feature maps is sampled from a normal distribution with a smaller variance $\mathcal{N}(\mu = 0, \sigma = 0.1)$ and added to feature maps component-wise. This introduces more spatial variation, as seen in Figure 3.15.

Rotation

The rotation module is implemented by first generating a 243×243 matrix with components sampled from a normal distribution $\mathcal{N}(\mu = 0, \sigma = 1)$. This matrix is then subject to a Grand-Schmidt process (QR decomposition) which factorizes the matrix into an orthonormal matrix Q and an upper triangular matrix R. All the 243-dimensional pixel vectors of the latent feature maps are transformed using the orthogonal matrix Q, essentially rotating the feature maps in this 243-dimensional feature-space. Since the channel dimension represents different textures, this procedure can be interpreted as a rotation in the space of all possible textures per image patch. The results of this procedure are displayed in Figure 3.16.

Additionally, we can increase the spatial diversity by introducing component-wise noise in the feature maps, as before. The results are presented in Figure 3.17.

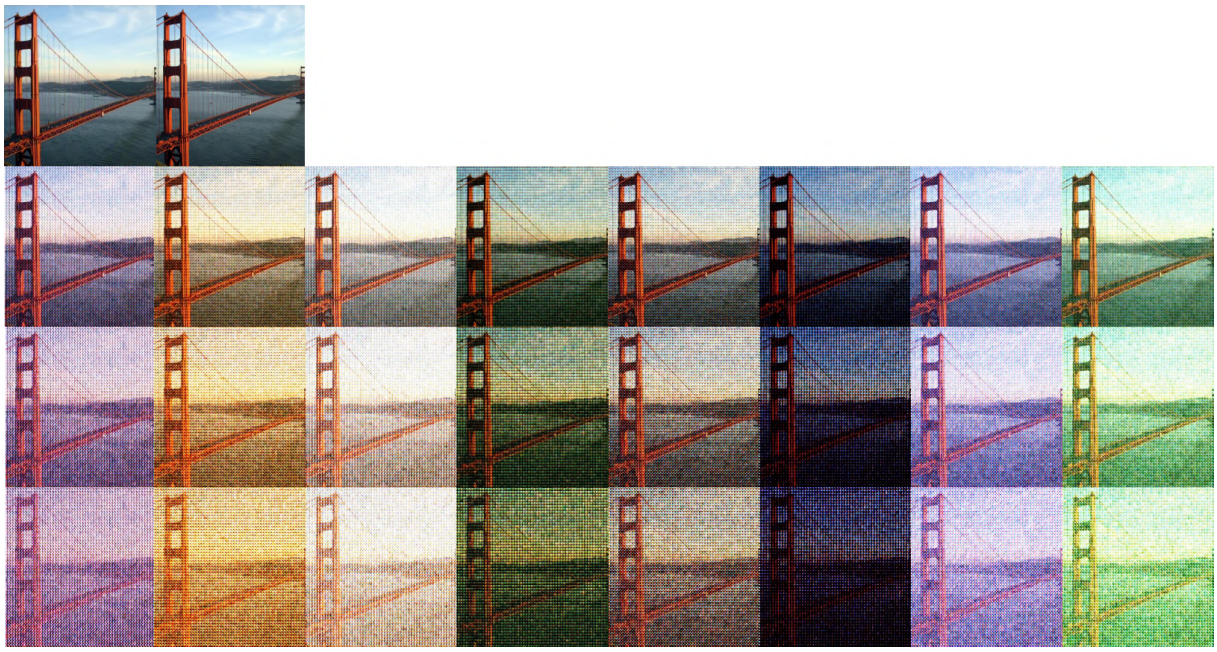


Figure 3.15: Images randomly textured by the Encoder-Transformer-Decoder architecture with translation and feature noise. First row: reconstructed image and input image. Second row with $\alpha = 0.3$, third row with $\alpha = 0.6$ and final row with $\alpha = 1.0$.

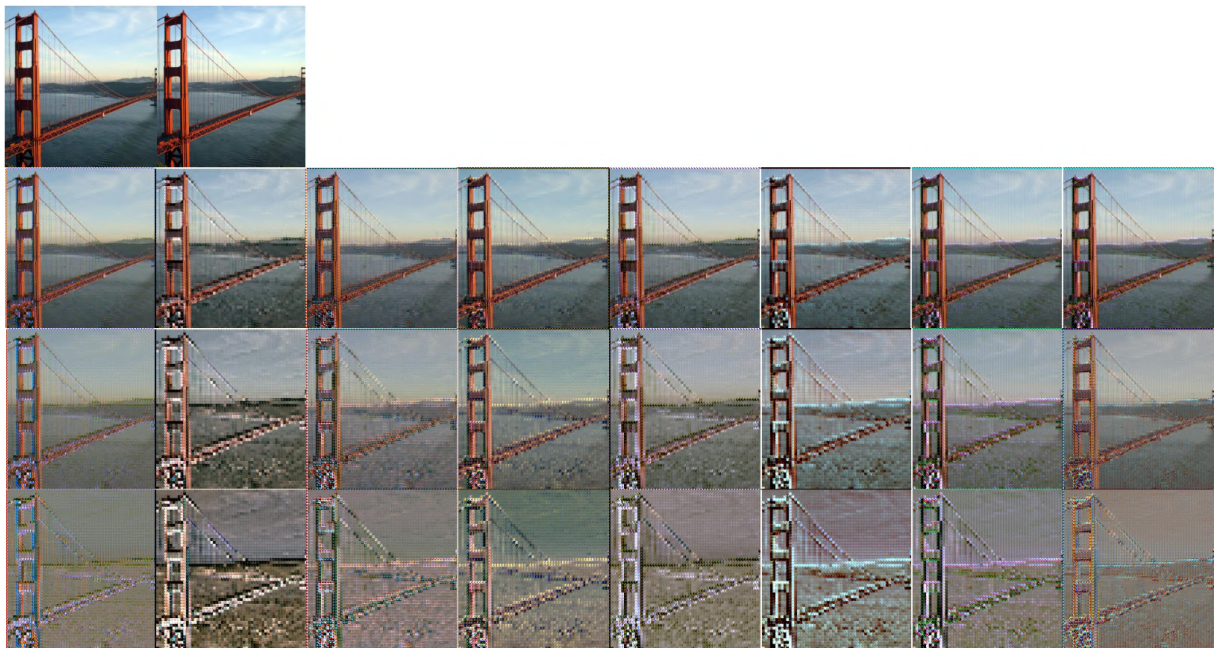


Figure 3.16: Images randomly textured by the Encoder-Transformer-Decoder architecture with rotation. First row: reconstructed image and input image. Second row with $\alpha = 0.3$, third row with $\alpha = 0.6$ and final row with $\alpha = 1.0$.

AdaIN Transformation

This implementation uses the same adaptive instance normalization technique as the AdaIN stylization method. [71]. This is essentially the same architecture as the random instance normalization procedure that was covered before, but this time the encoder is trained from scratch. The μ and σ for each channel are sampled from gaussian distributions $\mu_c \sim \mathcal{N}(\mu = 0, \sigma = 0.3)$ and $\sigma_c \sim \mathcal{N}(\mu = 1, \sigma = 0.1)$. The

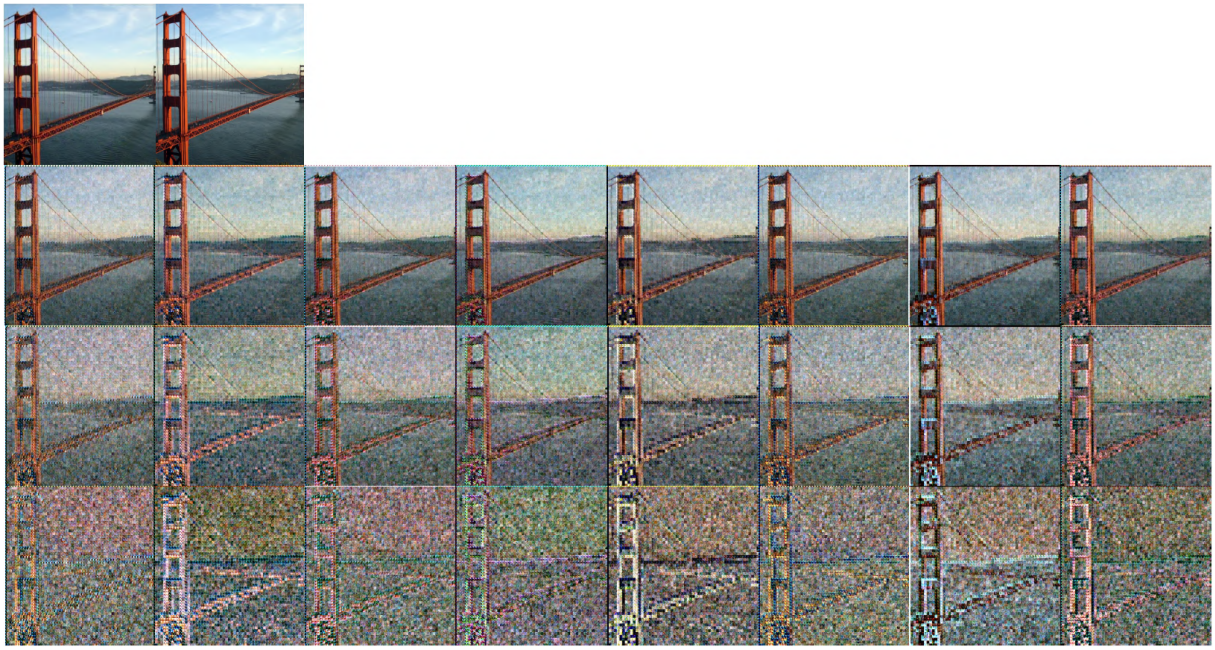


Figure 3.17: Images randomly textured by the Encoder-Transformer-Decoder architecture with rotation and feature noise. First row: reconstructed image and input image. Second row with $\alpha = 0.3$, third row with $\alpha = 0.6$ and final row with $\alpha = 1.0$.

results are presented in Figure 3.18.

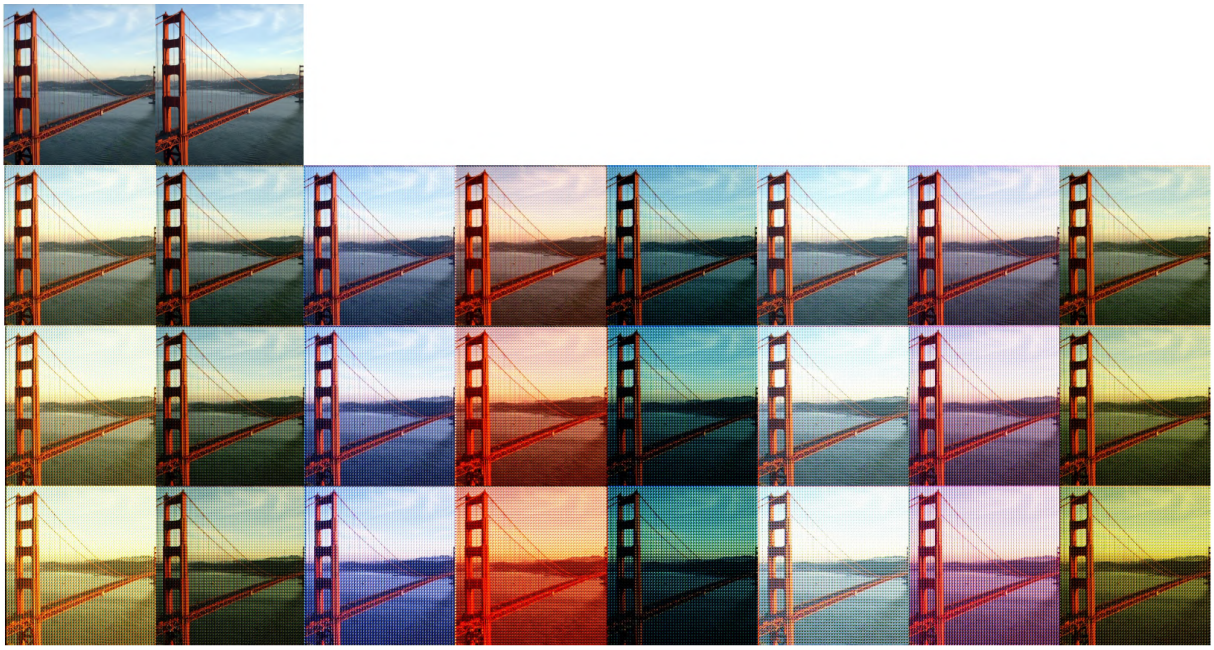


Figure 3.18: Images randomly textured by the Encoder-Transformer-Decoder architecture with AdaIN. First row: reconstructed image and input image. Second row with $\alpha = 0.3$, third row with $\alpha = 0.6$ and final row with $\alpha = 1.0$.

Yet again, adding a component-wise noise to the feature maps increases the spatial diversity of the images. This solves a problem with the AdaIN method, where textureless regions of the input image produces textureless regions in the output image. This happens because a convolution operation applied to a flat image will allays produce another flat image. In $\mathbb{R}$, a convolution of a kernel $k(t)$

applied to a constant function $f(t) = a$ will produce the integral of the kernel multiplied by a, which is constant. All consequent feature maps are therefore flat. The introduction of pixel-wise noise, even in small magnitude compared to μ and σ, is enough to generate more prominent patterns in the output image. This is noticeable in Figure 3.19.

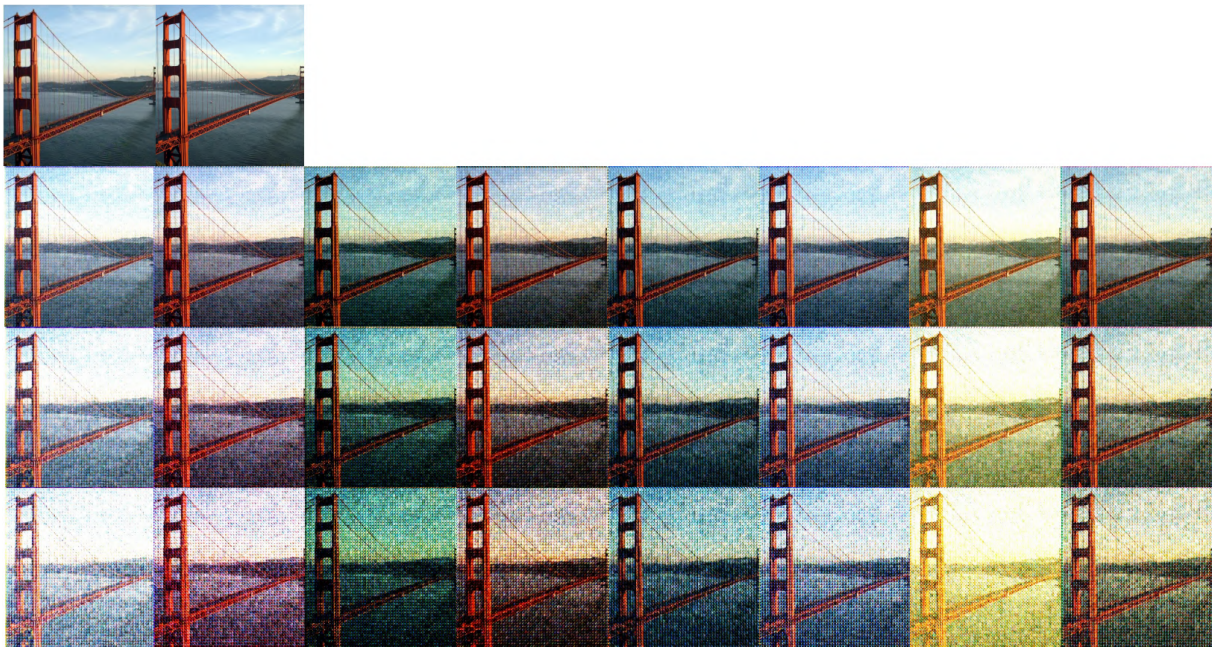


Figure 3.19: Images randomly textured by the Encoder-Transformer-Decoder architecture with AdaIN and feature noise. First row: reconstructed image and input image. Second row with $\alpha = 0.3$, third row with $\alpha = 0.6$ and final row with $\alpha = 1.0$.

Combinations

Finally, it is possible to combine multiple of these transformations to generate even more kinds of texturizations. While every transformation can theoretically generate all possible textures, they do so with different probability distributions. A translation will transform the textures by slightly altering them in a neighbourhood of the original texture, with more probability the closest it is to the original. A rotation will map a texture to any other texture, at the same distance from the origin, with equal probability and therefore generates more extreme and seemingly random texturizations. The AdaIN transformation will map one texture to another with a probability that is dependant on the relation of the input texture to the mean and variance of the latent feature maps. Therefore combining different transformations will generate texturizations with different probability distributions.

Combining a translation with AdaIN results in an equivalent transformation to AdaIN so it will not be explored. Figure 3.20 shows the results of performing a rotation followed by a translation and adding a component-wise noise. In Figure 3.21 a rotation is followed by an AdaIN transformation.

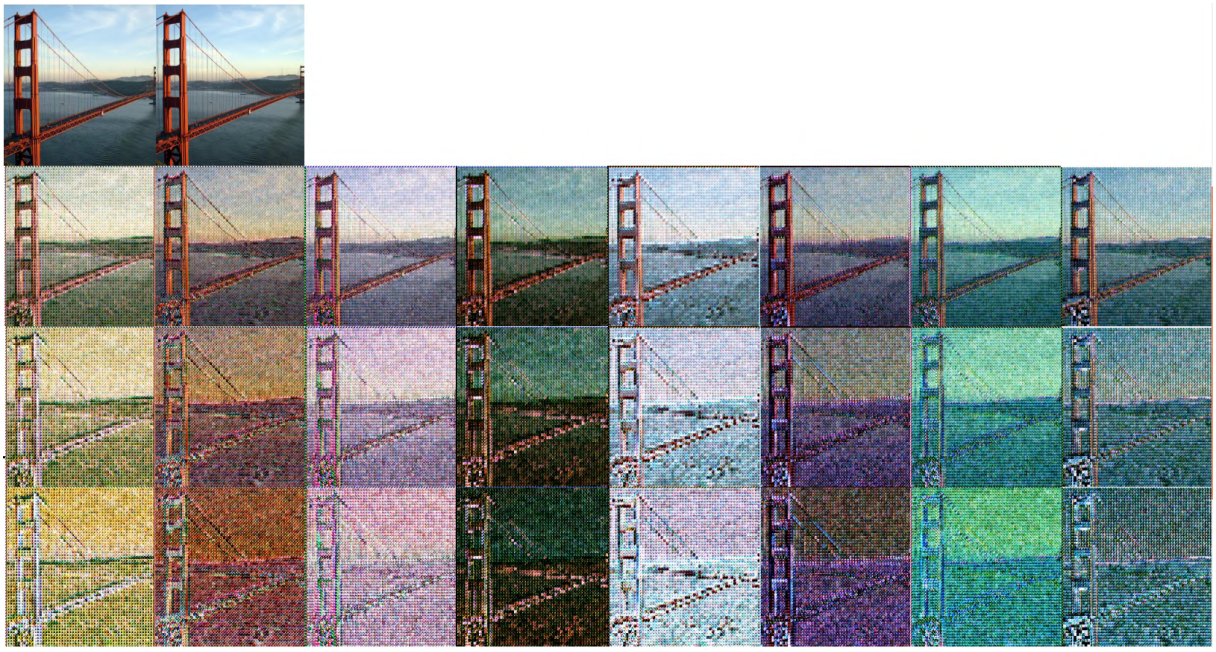


Figure 3.20: Images randomly textured by the Encoder-Transformer-Decoder architecture with rotation, translation and feature noise. First row: reconstructed image and input image. Second row with $\alpha = 0.3$, third row with $\alpha = 0.6$ and final row with $\alpha = 1.0$.

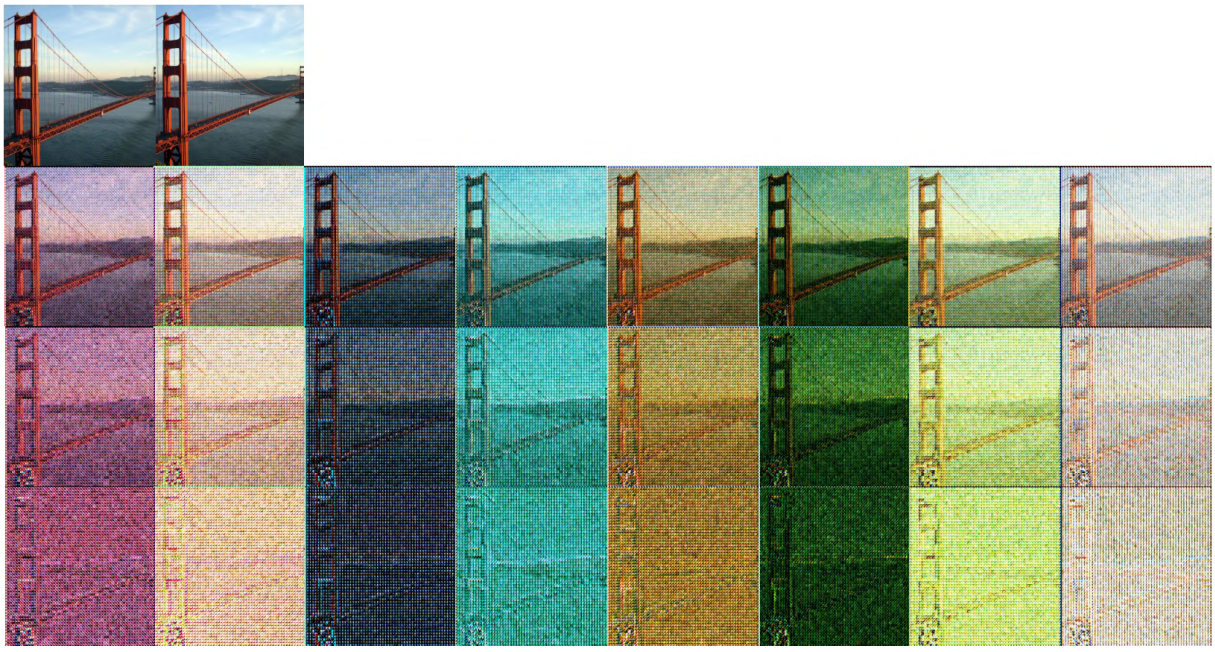


Figure 3.21: Images randomly textured by the Encoder-Transformer-Decoder architecture with AdaIN, translation and feature noise. First row: reconstructed image and input image. Second row with $\alpha = 0.3$, third row with $\alpha = 0.6$ and final row with $\alpha = 1.0$.

Encoder-Transformer-Decoder Conclusions

Since this method is based on a simple auto-encoder with no bottleneck, the reconstruction of the images with no noise introduced is very close to perfect, as can be noticed in the top left image of each Figure. This means that the transformations can be controlled with the coefficient α and produce only very slight changes in the image. This makes α a good hyper-parameter to tune when performing other

experiments with the texturized images.

The three transformation types and their combinations all produce convincing and usable randomized texturizations. The translation in channel-space is useful when we need a domain expansion that is centered around the original domain. The rotation in channel-space is useful when we need to cover all possible texturizations with equal probability without loosing the orthogonality of the features and the information in the input image. The AdaIN transformation is useful when we need a domain expansion that is not necessarily centered in the original domain and covers a wide spectrum of textures while keeping the orientation of the representations the same and therefore keeping more of the visible content from the input image.

The main caveat with this architecture is the fact that it transforms the images by patch. The space of possible textures is defined as the space of all possible square patches of a chosen size. A texture is not necessarily encoded in square tiles and it is not necessarily periodic in space. Therefore a good texturization method should not be based on square patches. This generates textures that have a characteristic periodicity to the human eye.

However, this technique is meant as a data augmentation tool for further CNN training. Since most CNNs process the images in the same square-like periodic manner, it is considered that this method is suited to augment images that will later be processed by a CNN.

3.3 Randomized Domain Shifting Conclusions

In purely visual and stylistic terms, the randomized styles produced by the AdaIN style transfer based method, when trained on a style dataset, are by far the most pleasing. Nevertheless, this technique suffers from the lack of generality that comes with training the decoder on a style dataset. The styles that are produced are thus constrained by the style dataset and do not cover all possible textures. Adapting this technique to forgo the need of a style dataset produces far less impressive results. The results do not justify the use of a big VGG16 encoder and corresponding decoder.

When compared to the randomized AdaIN style transfer technique, the Encoder-Transformer-Decoder method is much simpler and far less computationally intensive. Further, it is not limited by a style dataset to generate novel textures and is therefore more general. It lacks in visual neatness to the human eye because of the patch based nature of the architecture but this might not be a problem for CNNs. This approach will therefore be tested as a data augmentation tool in various scenarios in the following sections.

Chapter 4

Domain Generalization Experiments

In this chapter we shall test the generalization performance of networks that are trained using the Encoder-Transformer-Decoder architecture for image domain augmentation. The generalization performance will be assessed in the standard setting for Domain Generalization (DG) experiments, described in section 2.6.2. The networks will be trained on a dataset of a specific domain and then tested in a second dataset of a different domain, that was not seen during training. The generalization performance is compared between networks by measuring the performance on the novel testing set.

In section 4.1, we measure how the ETD architecture affects the texture bias of a CNN, by testing it in a mixed-cue dataset. In section 4.2, we compare, in a semantic segmentation task, the baseline network training against our augmentation technique with the same main network architecture, on the same datasets and with the same training conditions. In section 4.3 we perform the same comparison in an object detection task.

4.1 Texture Bias Experiments

In this chapter we shall test the texture-bias of networks that are trained using the Encoder-Transformer-Decoder domain-shifting technique, using a similar approach to [75]. The objective of this experiment is to show that training a network with randomly textured images reduces the texture-bias of the network, when compared to vanilla training. The texture-bias is assessed using a mixed-cue dataset, generated using the STL10 dataset [126, 127] and the AdaIN-based fast style transfer algorithm [71]. We perform style transfer with the content image of one category and the style image of another. We test networks with standard training against networks trained with randomized domain shifted images and test them with this mixed-cue dataset. Within the network responses that correspond to either the correct style or correct content category, the texture bias is the percentage of predictions that correspond to the correct style category.

4.1.1 Experiment Description

Dataset

This experiment is performed on the STL10 dataset. This dataset contains 10 classes of objects with 500 training images and 800 testing images per class, at a resolution of 96×96. The image categories are, in the following order: airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck.

The original dataset has two splits, one for training with 5000 images and one for testing with 8000 images. We further randomly split the training set into a training subset with 4000 images and validation subset with 1000 images. We also split the testing set into 4000 content images and 4000 style images to generate 4000 mixed-cue images on which the networks are tested.

Domain Randomizer Architecture

We use the same encoder and decoder that was described in section 3.2.2, in table 3.1. The Auto-Encoder is trained on the MS-COCO dataset for 25000 iterations with a batch size of 16 and a learning rate of 1×10^{-4} with Pytorch's default ADAM optimizer.

A translation transformation is applied in between the encoder and decoder. The standard deviation of the 243 dimensional vector is set to a variable parameter σ and the standard deviation of the noise tensor is set to $\sigma/2$. The parameter σ controls the intensity of the transformation and is varied to test different texturization strengths. The following values of σ are tested: $\sigma \in \{0.05, 0.1, 0.2, 0.3, 0.5\}$.

We show examples of the texturization applied to the STL10 dataset in Figure 4.1.

Figure 4.1: Examples from the STL10 dataset with a random texturization applied. Each column represents one class in the following order: bird, car, cat, deer, dog, plane, ship, truck. These images were used for training the networks.

Network Architecture

We perform the tests using Pytorch's implementation of a ResNet-34, which has an initial 7×7 strided convolutional layer, followed by 32 convolutional layers divided into 16 residual blocks and capped with a

fully connected classifier layer. The classifier layer is altered to have 10 output neurons, corresponding to the 10 classes of STL10. The network is initialized with the weights from pre-training on ImageNet.

Training Details

The baseline training session uses no image augmentation technique. The network is trained for 100 epochs with a batch size of 16. The optimizer is Pytorch's standard SGD with learning rate 1×10^{-4} and 0.9 of momentum. After each epoch, the network is evaluated on the validation subset. The best performing network in the validation subset over the 100 epochs is kept for the testing phase.

In the following training sessions, the network is reinitialized and trained using the images generated by random texturization, with a different values of σ each time. The same training parameters and procedure are used. Each configuration is ran 3 times to produce the mean and standard deviation of the results.

Style Transfer Architecture

The style transfer architecture is the same used in [71]. The encoder consists of the 9 first convolutional layer of a VGG16. The decoder is a mirrored version of the decoder where the 2×2 max-pooling layers are replaced by 2 times up-sampling layers. We use the pre-trained style transfer network weights available online [128]. We show examples of the mixed-cue STL10 images generated by this architecture in Figure 4.2.

Figure 4.2: Examples of style transfer between different classes of STL10. Top row: content images, Middle row: style images, Bottom row: resulting images. These images were used to evaluate the networks.

Metrics

We evaluate the performance and texture bias of the final trained network by constructing a $10 \times 10 \times 10$ confusion tensor $\mathcal{C} = [\mathcal{C}_{ijk}]$, where each entrance corresponds to the number of times the network predicted a class k when shown a stylized image with content category i and style category j.

The overall Texture Bias is calculated as the correct texture decisions over the correct texture decisions plus the correct content decisions:

$$TB = \frac{\sum_{i,j,k|j=k} \mathcal{C}_{ijk}}{\sum_{i,j,k|j=k} \mathcal{C}_{ijk} + \sum_{i,j,k|i=k} \mathcal{C}_{ijk}} \tag{4.1}$$

where the notation $\sum_{i,j,k|j=k}$ means the sum over indexes i, j, k such that $j = k$.

Content Accuracy is measured as the fraction of correct content decision ($i = k$):

$$CA = \frac{\sum_{i,j,k|i=k} \mathcal{C}_{ijk}}{\sum_{i,j,k} \mathcal{C}_{ijk}} \tag{4.2}$$

Texture Accuracy is calculated as the fraction of correct style decision ($j = k$):

$$TA = \frac{\sum_{i,j,k|j=k} \mathcal{C}_{ijk}}{\sum_{i,j,k} \mathcal{C}_{ijk}} \tag{4.3}$$

Finally, Class-specific Texture Bias is the Texture Bias constrained to a content class (i):

$$CTB_i = \frac{\sum_{j,k|j=k} \mathcal{C}_{ijk}}{\sum_{j,k|j=k} \mathcal{C}_{ijk} + \sum_{j,k|k=i} \mathcal{C}_{ijk}} \tag{4.4}$$

4.1.2 Results

Figure 4.3 shows the mean values of the Texture Bias (TB), Content Accuracy (CA) and Texture Accuracy (TA) while varying the values of the texturization strength σ. The value $\sigma = 0$ corresponds to the baseline network that was trained without the domain randomizer. The shaded region represents the area within one standard deviation of the mean.

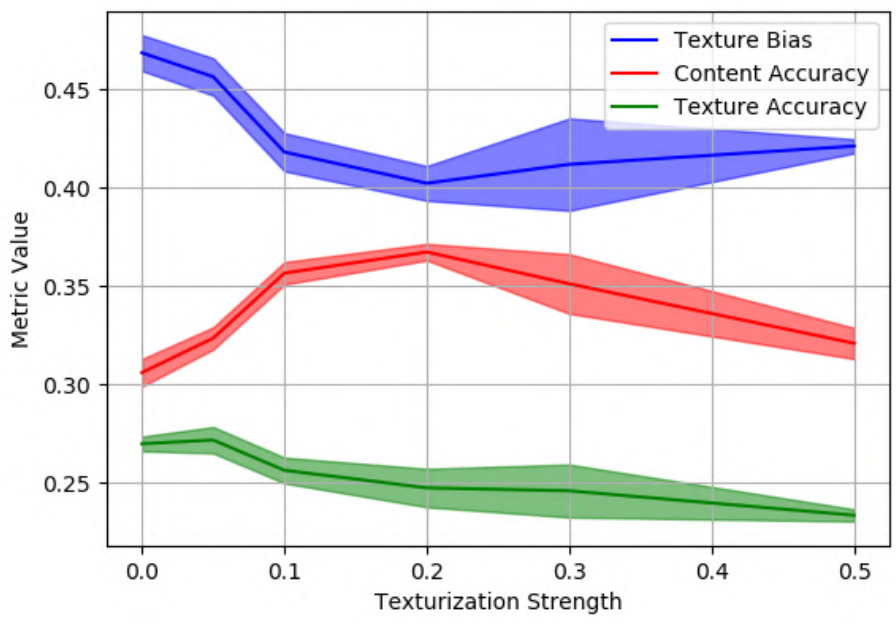


Figure 4.3: Texture Bias (TB), Content Accuracy (CA) and Texture Accuracy (TA) while varying the values of the texturization strength σ, averaged over three runs. Shaded region represents one standard deviation from the mean.

A clear optimal value for the texturization strength $\sigma = 0.2$ simultaneously minimizes the texture bias and maximises the content accuracy in the mixed-cue dataset. This optimal value is specific to this task and network architecture, meaning that, further implementations of this texturization method, in different contexts, should be adapted to the scenario in question.

It is also noticeable that the texturization strength has a big impact in the network performance. If the value of σ is too large the network will start underfitting both the content and style images. And no advantage is gained. The process of optimizing the texturization strength to the task at hand is therefore crucial. In case of doubt or limited resources, if the network is not already underfitting, a lower value of σ is sure to improve performance, acting as data augmentation, while a high value of σ may negatively affect performance.

In general, it is observed that increasing σ always renders the network more less texture sensitive, since the texture accuracy is monotonously decreasing. Even though we search to train texture invariant networks, the textural information may still be necessary to achieve good results in a particular task and therefore the content accuracy drops when the texture invariance becomes too strong (indicated by a low texture accuracy).

Figure 4.4 shows the comparison of Class-specific Texture Bias (CTB) between standard training and training with textured images. All classes in the STL10 dataset are learned with a smaller texture bias when textured images are used during training. The results for the classes ”horse” and ”cat” are unclear since they fall within one standard deviation around the mean. In general, the results indicate that training with textured images successfully renders the network more texture invariant, across classes, since the network is more capable of correctly classifying an image with texture cue conflicts. The network is therefore more capable of ignoring the low level statistics of the images.

4.1.3 Conclusions

Randomly changing the low level statistics of images during training of a CNN is not a new technique. It has been first proposed by [75], which used a style transfer technique to randomly transfer painting styles to ImageNet images, as described in section 2.5. This previous technique is successful in reducing the texture bias of the fully trained CNN but it requires the use of a VGG16 network, a mirrored image decoder and a style dataset to perform the style transfer. From this ensues a big computational overhead during training, which might be undesirable.

Our technique, while it is highly inspired by the encoder-decoder architecture of the style transfer technique, it uses much smaller networks and no style dataset. The style transfer technique needs a big network because it needs to accurately capture style, which can contain ”medium-level” information. Meanwhile, our technique is focused on generating the lowest level spatial-coherent information that is possible. Furthermore, we search to generate all possible variations of this low-level information without learning the styles from an inherently biased painting dataset. This is, in fact, possible with a very simplistic architecture.

From this section, we conclude that our technique offers a significant reduction of the texture-bias of

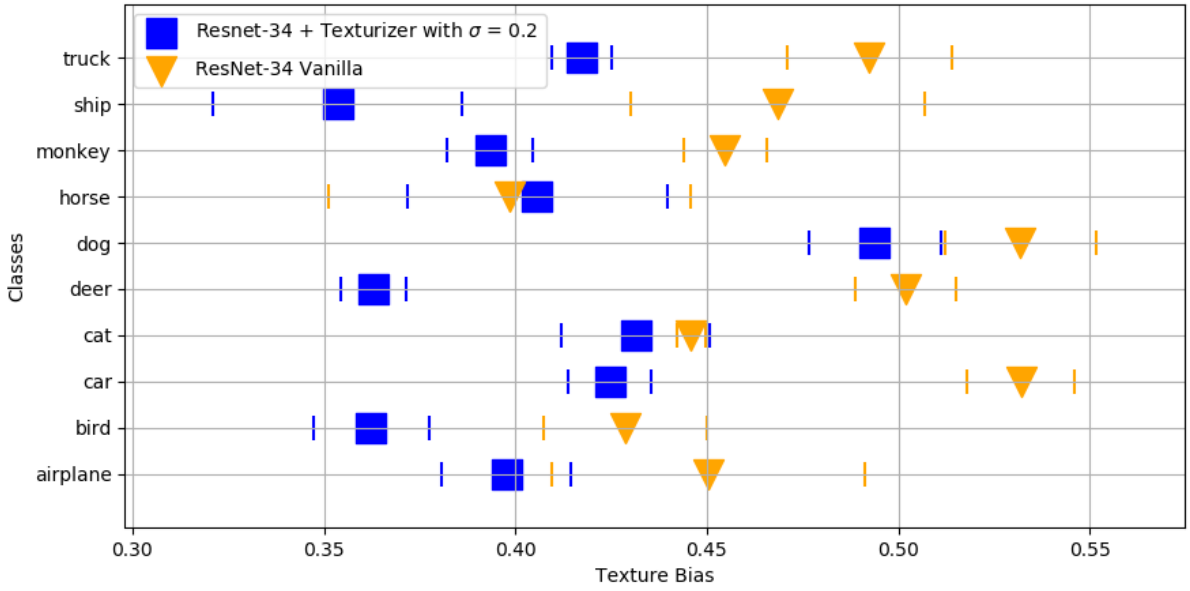


Figure 4.4: Comparison of Class-specific Texture Bias (CTB) between standard training and training with textured images, averaged over three runs. Vertical bars represent one standard deviation from the mean (triangles and squares).

convolutional neural networks in a classification task, with low computational overhead.

4.2 Domain Generalization in Semantic Segmentation

We test the Domain Generalization performance in the task of Semantic Segmentation of road scenes. In this scenario, the network is trained on a single dataset with images from a simulated environment. The network is then tested with images of a real environment. The vanilla training scheme usually produces networks that perform well in the simulated environment but whose performance drops when tested in the unseen real domain. The techniques with better generalization performance should be able to perform better in the real environment.

As described in section 2.4.1, the task of Semantic Segmentation consists of attributing each pixel in an image to the correct class label. In the context of road scene segmentation, this consists of labeling the pixels that correspond to the road, cars, trucks, people, sidewalk, buildings, poles, traffic lights, traffic signs, sky, and others.

4.2.1 Experiment Description

Datasets

We chose two training datasets and one testing dataset for this experiment.

The first training set is the GTA5 dataset [129], which includes 24966 densely annotated images of in-game road scenes, with 19 different classes and a resolution 1914×1052. During training, the training and

validation subsets, consisting of 18786 images, are used for training, and the testing subset, consisting of the remaining 6180 images, is used for validation. The images are down-sampled to 1280×720 resolution due to GPU memory constraints. Textured examples of this dataset are displayed in Figure 4.5.

Figure 4.5: Examples from the GTA5 dataset with a random texturization applied. These images were used for training the networks.

The second training dataset is the Synthia dataset [130], whose training subset includes 9000 images, generated from a simulated city environment, with resolution 1280×760. The testing subset, containing 400 images, is used for validation during training. The images are densely annotated with 13 classes, compatible with those of the GTA5 dataset. Textured examples of this dataset are displayed in Figure 4.6.

Figure 4.6: Examples from the Synthia dataset with a random texturization applied. These images were used for training the networks.

We chose the Cityscapes dataset [131] of real road scenes as a testing dataset due to its class compatibility with with the other two training sets. Cityscapes includes 5000 frames of real road scene images with pixel-level annotations and 20000 frames with coarse annotations. Only the finely annotated images will be used for testing. The images have a resolution of 2048×1024 which is again down-sampled to 1280×720 to keep the image scale of GTA5 and Synthia. The images are annotated with 30 different classes. The testing results are calculated only with respect to the classes that are common between the training and testing datasets. An example of this dataset with labels is presented in Figure 4.9.

Domain Randomizer Architecture

The architecture of the image texturization network is summarized in table 4.1.

Layer	Channels In	Channels Out	Kernel Size	Stride	Padding	Activation
Encoder C0	3	48	4×4	4	2	Tanh
Encoder C1	48	192	4×4	2	2	Tanh
Decoder TC0	192	48	4×4	2	2	Tanh
Decoder TC1	48	3	4×4	4	2	None

Table 4.1: Architecture of the Auto-Encoder for random texturization.

Instead of the KL divergence loss, an hyperbolic tangent (Tanh) activation is used to keep the feature maps in the range of -1 to 1, guaranteeing the translation of the feature maps is comparable to their variance.

A translation transformation is applied in between the encoder and decoder. The standard deviation of the 192 dimensional translation vector is set to a variable parameter σ and the standard deviation of the noise tensor is also set to σ. The parameter σ controls the intensity of the transformation and is varied to test different texturization strengths. The following values of σ are tested: $\sigma \in \{0, 0.01, 0.03, 0.06, 0.1, 0.15, 0.22, 0.3\}$.

Furthermore, a decay parameter γ is used to reduce the strength of the texturization as the training progresses, as a form of curriculum learning. This approach has been shown to produce good results when using gaussian blur [132]. The decay parameter modulates the texturization strength throughout training by updating it every 1000 iterations in the following manner:

$$\sigma \leftarrow \sigma(1 - \gamma) \tag{4.5}$$

Modulating the texturization strength with a decay parameter induces a shape bias in the early training stages. Slowly suppressing the domain shifts in the later phases then allows the network to fully learn the statistics of original dataset, hopefully without losing the initial shape bias.

To simulate a real application scenario, the texturization network is trained in the same training dataset on which the main network will be trained. The Pytorch's default ADAM optimizer is used with a learning rate of 1×10^{-4}. The Auto-Encoder is trained for 5000 iterations and is frozen before the start of the main training stage.

Network Architectures

Our technique is tested with two semantic segmentation architectures.

The first is a fully convolutional network (FCN-8s) architecture for semantic segmentation, described in section 2.4.1. This architecture is implemented with a VGG16 backbone, upsampling 32 times the features from the layer pool5, 16 times the features from layer pool4 and 8 times the features from layer pool3. The full architecture is available at [133]. The backbone is initialized with the weights from pre-training in ImageNet.

The second architecture that was tested is the DeeplabV2 network with a pretrained ResNet-101 backbone, described in section 2.4.1. The full network implementation and weights are available at [134].

Training Details

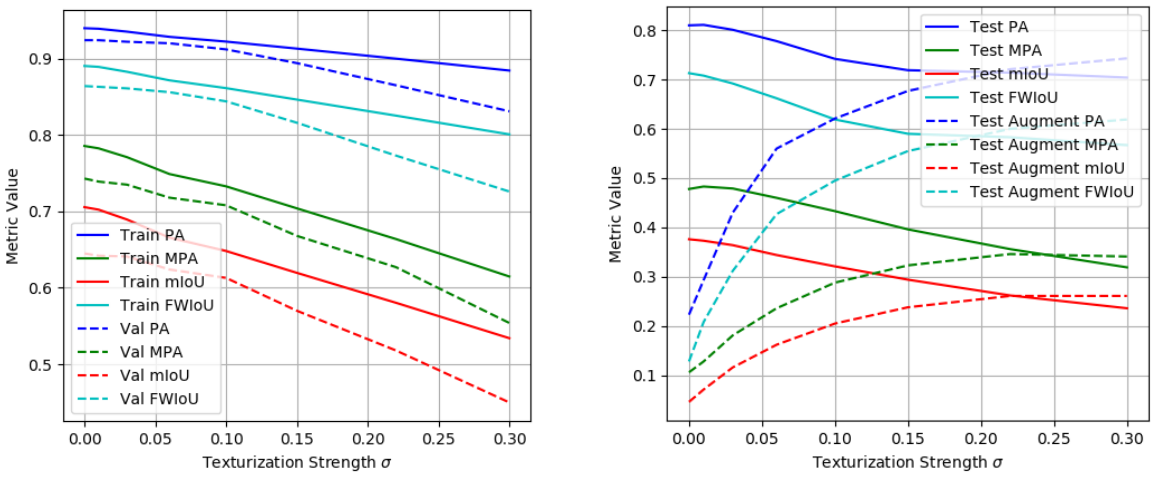
The main network is trained for 150000 iterations using a SGD optimizer with a learning rate of 2.5×10^{-4}, momentum of 0.9, and a weight decay of 5×10^{-4}. The learning rate is updated during training with a polynomial learning rate scheduler [135], which decays the learning rate to 0 at the end of training using a power law. The batch size is set to 1 due to GPU memory limitations. The 150000 training iterations equate to 7.98 epochs in GTA5 and 16.67 epochs in the Synthia dataset. The networks are validated at the end of each epoch and the network with the best validation score is kept for testing. The training datasets are augmented with colour jitter and random horizontal flipping, both standard data augmentation techniques. The training implementation, with exception of the texturization network, is based on [136].

Metrics

The networks are evaluated with the metrics described in section 2.4.1.

4.2.2 Results

We begin by showing the effect of changing the texturization strength on the different performance metrics in Figure 4.7, during training, validation and testing.



(a) Training and Validation Phase

(b) Testing Phase

Figure 4.7: Evolution of performance metrics with changing texturization strength. Training is done on the GTA5 dataset and testing in the Cityscapes dataset. The semantic segmentation task is performed with a FPN architecture and a VGG16 backbone. In (a), we show the training and validation metrics, evaluated on the simulated GTA5 dataset. In (b) we show the testing scores on the real Cityscapes dataset, where the full lines correspond to testing on the unchanged images and the dotted lines correspond to testing on textured images, with a fixed texturization strength of $\sigma = 0.2$.

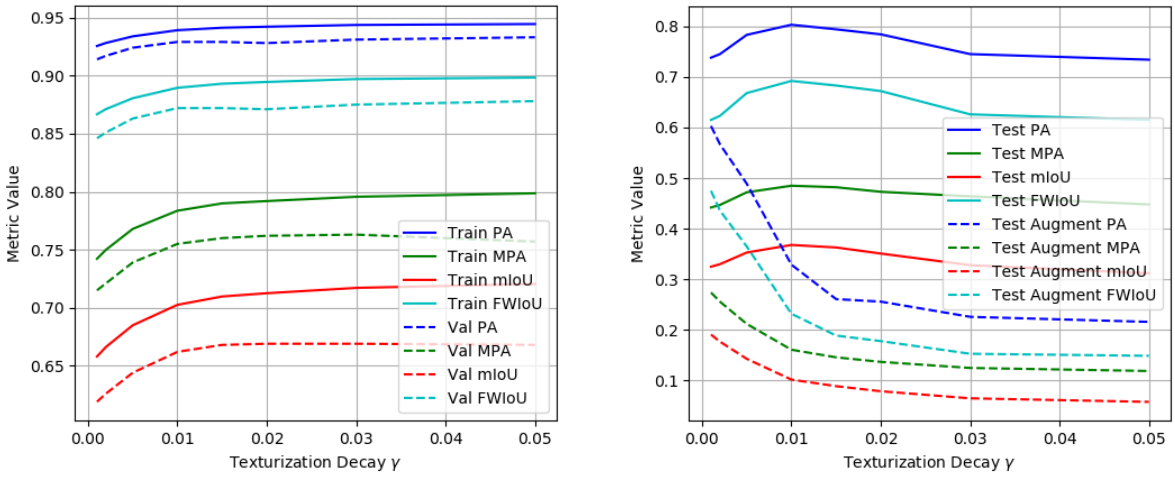
Unlike the results of section 4.1.2, we notice in Figure 4.7 (b) that introducing progressively textured images always decreases the generalization test scores of the network. This indicates that there is no value in using this technique in this situation. This happens not only for the testing scores but for the training and validation scores as well. Since the training and validation scores are relatively close together, it is evident that no significant overfitting to the training set is occurring. On the contrary, the fact that the training performance decreases with the texturization strength indicates that a substantial amount of underfitting may be occurring. This is explained by the fact that the GTA5 dataset is already rather diverse. This fact, coupled to the use of further data augmentation techniques like random horizontal flipping and color jitter, means that the full capacity of the network is already being utilised successfully. Any further data augmentation technique will therefore saturate the learning capacity of the network and it will start to underfit the texture information as well as the shape information. Moreover, the testing performance on augmented images peaks at the same value as the testing in untouched images. From this observation we deduce that the network successfully became invariant to the low level image statistics that are introduced by the texturization process. A simple extrapolation indicates that, at a texturization strength of around $\sigma = 0.5$, the network would become so invariant to textures that the simulated and real images would yield the same performance. This would happen when the validation and testing scores are the same. Unfortunately, this level of texture invariance comes at the cost of a big performance drop in either dataset. We thus conclude that texture invariance is very expensive in terms of network learning capacity if it is obtained through data augmentation techniques. Other means to render networks texture invariant might therefore be needed.

In the same experiment we also tested the effect of changing the value of the texturization decay parameter γ at a fixed initial value for the texturization strength $\sigma = 0.1$. A low decay value leads to a final texturization strength that is only a small percentage below the initial value. A high decay setting leads to a final texturization strength that it is close to 0. Results are shown in Figure 4.8.

It is noticeable in Figure 4.8 (a) that, for larger decay settings the underfitting effect seems to disappear. This is because, in the later training stages, the network has access to the original dataset statistics and is able to fit it as best as possible. In fact, a marginal increase in performance is observed in the validation scores when comparing Figure 4.7 (a) at $\sigma = 0$ and Figure 4.8 (a) at $\gamma > 0.02$. This means that the early shape bias induced by the textured images improves the generalization performance in the validation dataset.

For a fixed initial texturization strength $\sigma = 0.1$, we observe an optimal value for the decay at $\gamma = 0.01$. This equates to a texturization strength of $\sigma = 0.0022$ at the end of the 150000 iterations (decay every 1000 iterations). This indicates that, when using this texturization technique, additionally using decay is also beneficial. However, the baseline performance is still better than this optimal value, because the underfitting problem is still present. For higher values of the decay, the texture bias phenomenon becomes prominent and the generalization performance drops again. Lastly, and as expected, the generalization performance in the augmented test images becomes worse when the texturization strength is reduced throughout training because the network is less exposed to the augmented images during training.

We now compare the generalization performance between a network with standard training and a



(a) Training and Validation Phase

(b) Testing Phase

Figure 4.8: Evolution of performance metrics while changing the texturization decay parameter for a fixed texturization strength of $\sigma = 0.1$. Training is done on the GTA5 dataset and testing in the Cityscapes dataset. The semantic segmentation task is performed with a FPN architecture and a VGG16 backbone. In (a), we show the training and validation metrics, evaluated on the simulated GTA5 dataset. In (b) we show the testing scores on the real Cityscapes dataset, where the full lines correspond to testing on the unchanged images and the dotted lines correspond to testing on textured images, with a fixed texturization strength of $\sigma = 0.2$.

network trained with randomly textured images for a fixed texturization strength $\sigma = 0.1$. Table 4.2 shows this comparison across classes for the two chosen architectures and the two chosen training datasets.

It is observed, as expected regarding the previous experiment, that the performance is deteriorated when using our proposed technique. In particular, the Deeplab architecture with a ResNet101, which is generally considered to perform better and be more capable than the VGG16-FCN, seems to be far more affected by the use of textured images during training. This is consistent with the findings of [75], which showed that more recent network architectures, with skip-connections and blocks with multiple processing paths, tend to be more texture-biased than simple sequential architectures like VGG16.

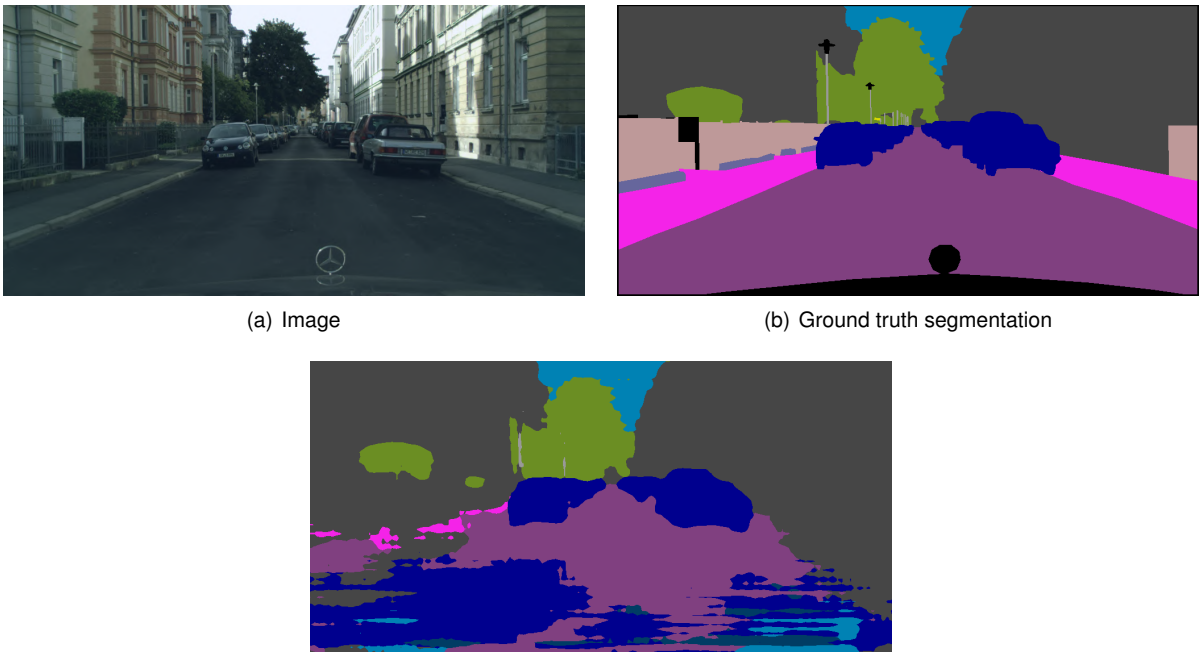
It is also visible that training on the GTA5 dataset produces better results. This is due to the fact that GTA5 is a more complete simulated environment than Synthia, with more realistic vehicles, people, scenarios, textures, lighting, and overall detail. This shows that working on better computer graphics will be fundamental to achieve a good adaptation from simulation to real.

An example of the segmentation maps produced on the Cityscapes dataset by the ResNet-101 architecture, trained on GTA5 with texturization, is displayed is displayed in Figure 4.9. The method clearly produces unacceptable semantic maps that would be unusable in a self-driving scenario.

We notice that some important classes in an autonomous driving scenario, like road, vehicle and person, have a particularly high degradation in performance. This means that the proposed technique might not be suitable to this purpose.

Training Dataset	GTA5				Synthia			
Architecture	ResNet101		VGG16		ResNet101		VGG16	
Data Augmentation	V	V+T	V	V+T	V	V+T	V	V+T
road	**71.45**	9.76	**72.07**	60.14	**42.51**	6.34	**53.61**	23.2
sidewalk	**15.03**	13.01	26.8	**28.88**	**19.31**	12.17	**27.02**	22.24
building	**79.61**	58.62	**81.85**	77.7	**77.59**	59.38	**73.49**	70.31
wall	**15.06**	0.08	**24.56**	21.13	**1.1**	0.1	**5.12**	0.24
fence	**19.36**	0.01	**20.46**	16.42	**0.03**	0.01	0.15	**0.3**
pole	**29.72**	12.76	**19.43**	14.25	**26.35**	11.63	**23.62**	17.34
traffic light	**35.03**	2.05	**28.07**	22.36	**2.47**	1.56	**10.76**	5.65
traffic sign	**23.41**	2.51	**15.32**	8.45	**10.79**	3.03	**13.48**	11.66
vegetation	**80.09**	43.79	**80.92**	78.09	**75.29**	44.54	**78.94**	74.09
terrain	**70.1**	0.0	**35.71**	33.32	0.0	0.0	0.0	0.0
sky	**91.72**	79.29	**82.75**	82.17	**83.29**	80.08	**82.16**	80.24
person	**74.04**	40.32	**52.62**	44.14	**59.51**	43.14	**56.11**	51.57
rider	**37.65**	4.37	16.94	**17.46**	**23.22**	3.56	**21.73**	19.67
car	**89.84**	34.62	**80.52**	41.11	**45.61**	35.72	**55.76**	20.61
truck	**66.59**	0.0	**20.4**	19.58	0.0	0.0	0.0	0.0
bus	**51.35**	0.67	**17.57**	15.72	**17.74**	0.57	**17.34**	14.69
train	**1.78**	0.0	2.78	**3.64**	0.0	0.0	0.0	0.0
motorbike	**55.91**	1.03	**21.87**	20.65	**13.23**	1.52	**21.49**	17.8
mIoU	**38.4**	16.7	**37.1**	32.1	**27.9**	16.5	**29.7**	23.6

Table 4.2: mIoU scores (%) on the Cityscapes testing dataset for different training datasets (GTA5 and Synthia), different network architectures (DeeplabV2 with Resnet101 backbone and FCN with VGG16 backbone) and different training data augmentations (Vanilla (V) and Textured Images (T)).



(a) Image

(b) Ground truth segmentation

(c) Predicted segmentation

Figure 4.9: Example of semantic segmentation on the Cityscapes dataset, produced by a DeeplabV2 architecture with a ResNet-101 backbone, trained on the GTA5 dataset with texture augmentation.

4.2.3 Conclusions

After the success in reducing the texture bias in section 4.1, in the task of image classification, it was thought that the domain generalization performance would increase also in other tasks. This section shows that, once the task becomes more demanding, the networks are not capable of training in data distributions that are highly augmented and still learn effectively. Thus, the generalization performance on a previously unseen training dataset dramatically drops when the degree of data augmentation increases.

Still, we showed that using this texturing technique in the early stages of training and then slowly decaying it off brings a marginal increase in the validation scores (in-distribution performance). This application to in-distribution learning may be object of future research but it is off the scope of this thesis, whose main concern is out-of-distribution performance.

We then showed that more recent and efficient architectures like the ResNet101, tend to suffer more in out-of-domain testing than previous parameter-heavy networks like the VGG16. This indicates that efficient architectures might be more inclined to fit low-level textures information instead of high-level shape information. Generally, architectures with a higher number of parameters, seem to be more capable of learning domain invariance. The problem arises when the task becomes too difficult and no standard network architecture is capable of learning this domain invariance. It is suspected that this technique could work in networks that are largely oversized for the task, as was the case in the classification task of the previous section. This becomes impracticable in a road scene segmentation task. Future research should therefore seek novel ways to obtain domain invariance, through invariance mechanisms that are introduced in the networks.

4.3 Domain Generalization in Object Detection

In this section, we test the Domain Generalization performance in the task of Object Detection in road scenes. The networks are trained to detect objects in images from a simulated environment and are then tasked with identifying the same class of objects in images from real road scenes.

As described in section 2.4.2, the task of Object Detection consists of drawing a bounding box around every instance of an object in a given picture and correctly identifying the class of the object. In the context of road scene object detection, this can include the identification of any number of objects. In this experiment we will reduce the task to simply finding the vehicles (cars, buses, trucks) in the image, all under one single ”vehicle” class.

4.3.1 Experiment Description

Datasets

As a simulated training environment we chose the VirtualKITTI2 dataset [137]. This dataset consists of 5 driving sequences in a simulated city environment, each with 10 different camera angles and weather conditions, totalling 42520 images with a resolution of 1242×375. The dataset was split with *scene02* (4661 images) used for validation and the 4 remaining scenes used for training.

We chose the KITTI real road scene dataset as a testing set due to its compatibility with VirtualKITTI2. This dataset contains 7481 images, with the same resolution as VIrtualKITTI2, in different real driving scenarios.

Domain Randomizer Architecture

We use the same encoder and decoder that was described in section 3.2.2, in table 3.1. The Auto-Encoder is trained on the MS-COCO dataset for 25000 iterations with a batch size of 16 and a learning rate of 1×10^{-4} with Pytorch's default ADAM optimizer.

A translation transformation is applied in between the encoder and decoder. The standard deviation of the 243 dimensional random translation vector is set to $\sigma = 0.2$ and the standard deviation of the noise tensor is set to $\sigma = 0.1$.

Network Architecture

We test the DG performance using the Faster R-CNN object detection architecture described in section 2.4.2. We test this method with three different backbone architectures of differing capacities.

The first and second are a VGG11 and a VGG19 with Batch Normalization layers, as described in section 2.3.1, using the default Pytorch implementation.

The third is a ResNet50 architecture, described in table 2.1, completed with a Feature Pyramid Network (FPN) scheme, described in section 2.4.1. Once again, this is an instance of the default Pytorch implementation.

Training Details

The networks are trained in the simulated environment for 10 epochs, using Pytorch's default ADAM optimizer, with a batch size of 8 and an initial learning rate of 10^{-4}. The learning rate is updated during training with a step learning rate scheduler, which decays the learning rate exponentially, multiplying it by 0.9 every epoch. Further, at the beginning of each epoch, a warm up scheme is used to ramp up the learning rate from 0 to its desired value over the first 1000 iterations. After each epoch, the networks are evaluated on the validation subset. The network with the best validation score (mAP@ 0.5 IoU) is kept for testing. In the baseline scenario there is no data augmentation procedure. The implementation follows [138].

Metrics

The networks are evaluated with the metrics described in section 2.4.2. They are implemented using the COCO evaluator [139]. This evaluator calculates the mean Average Precision (mAP) and the mean Average Recall (mAR) over all classes, over 101 linearly spaced values of confidence between 0 and 1, and over different ranges of IoU. The range of IoU $0.50 : 0.95$ indicates that the scores are averaged over IoU values starting from 0.50, in increments of 0.05 until 0.95.

The metrics are filtered at different object sizes, small area (between 40 and 1024 (32×32) pixels), medium area (between 1024 and 9216 (96×96) pixels) and large area (larger than 9216 pixels). They are also filtered at different maximum number of detections per image (MaxDets).

4.3.2 Results

We show the results of the COCO evaluation procedure in table 4.3, for the two chosen architectures, trained with and without the random texturization module.

Similarly to the results of the Semantic Segmentation experiments, introducing random textures to the images in an Object Detection task does not improve the overall performance of the networks. Both architectures seem to suffer more significantly for objects with a Small and Medium area (less than 96 pixels of height and/or width). This indicates that our texturization method destroys important information at small image scales. A marginal increase in performance is observed in objects of Large area. This method could therefore be useful in in a scenario where we only need to detect large objects or objects that are close-by.

Analysing the mF1 scores, we notice once again that the decrease in performance is larger for the ResNet architecture than it is for the VGG architectures, even though the ResNet performs better over all. Once more, this suggests that older simpler architectures may be more capable of learning domain invariance due to the higher number of parameters.

Comparing the VGG11 and VGG19 shows that simply increasing the size of the network is enough to achieve better performance in general but training in textured images affects them both equally.

Metric	IoU	Area	MaxDets	VGG11		VGG19		ResNet50-FPN	
				N	T	N	T	N	T
mAP	$0.50:0.95$	All	100	**0.341**	0.326	**0.410**	0.386	**0.434**	0.380
	0.50	All	100	**0.654**	0.625	**0.704**	0.682	**0.725**	0.669
	0.75	All	100	**0.328**	0.306	**0.433**	0.395	**0.465**	0.385
	$0.50:0.95$	Small	100	**0.115**	0.070	**0.170**	0.140	**0.245**	0.170
	$0.50:0.95$	Medium	100	**0.333**	0.307	**0.413**	0.368	**0.440**	0.358
	$0.50:0.95$	Large	100	0.479	0.**498**	0.525	**0.546**	**0.541**	0.538
mAR	$0.50:0.95$	All	1	**0.120**	0.116	**0.133**	0.128	**0.137**	0.127
	$0.50:0.95$	All	10	**0.383**	0.369	**0.449**	0.429	**0.476**	0.428
	$0.50:0.95$	All	100	0.391	0.391	**0.465**	0.443	**0.499**	0.450
	$0.50:0.95$	Small	100	**0.160**	0.129	**0.266**	0.199	**0.344**	0.260
	$0.50:0.95$	Medium	100	**0.386**	0.377	**0.469**	0.432	**0.500**	0.431
	$0.50:0.95$	Large	100	0.529	**0.561**	0.568	**0.598**	0.584	**0.590**
mF1	$0.50:0.95$	All	100	**0,364**	0,356	**0.436**	0.413	**0,464**	0,412

Table 4.3: Mean Average Precision (mAP) and mean Average Recall (mAR) in the VirtualKITTI2 to KITTI Object Detection task, for two different backbone architectures and two different data augmentation strategies (None (N) and Textured Images (T)). Mean F1 score (mF1) calculated directly from the mAP and mAR according to $mF1 = 2(mAP \cdot mAR)/(mAP + mAR)$.

4.3.3 Conclusions

The results of this experiment show that our technique alters images in a way that prevents the networks from learning useful representations of objects of small sizes. The marginal increase in performance for bigger objects suggests that, in fact, changing the textural information of images aids in generalization, but only when the scale of the objects is at least one order of magnitude larger than the characteristic length of the texture (periodicity). The network is not able to learn useful representations of small objects because the texturing of the image with our process obfuscates this small-scale information.

The results of the Domain Generalization experiments in both the Semantic Segmentation and Object Detection tasks reveal that these tasks do not benefit from an extreme data augmentation strategy. Expanding the domain of the training images leads to a severe amount of underfitting, which is characterized by a significant reduction in performance on the training set as well as on the validation and testing sets. This indicates that learning texture-invariant representations requires a significant investment of learning capacity by the network. Thus, a big increase in network capacity would be necessary to effectively learn using this data augmentation technique. This is obviously not the ideal solution to this problem. A better solution should seek to integrate invariance mechanisms within the network architecture itself. Such mechanisms would allow the network to ignore textural information without investing a significant amount of learning capacity. This way, our random domain shifting technique could be applied without the need to increase the network size.

Chapter 5

Conclusions

We arrive at the end of this thesis. This chapter will summarize our results and delineate possible future research avenues in the topics that were discussed throughout this work.

5.1 Achievements

In the introductory chapter of this thesis we set out to offer the reader a comprehensive look into the fundamentals of CNNs, their functioning, their applications and the generalization problems that they suffer from. In the background chapter, all these topics were covered in appropriate detail and a strong theoretical basis was established.

In chapter 3 we explored different domain shifting methods. Until now, image domain augmentation used to be performed either with simple transformations like image translations, rotations, scalling, additive noise, other kinds of noise, or with more computationally expensive techniques like style transfer. We first tried generalizing the AdaIN based style transfer method to work without a style dataset, this method generated images that had different textures but the diversity of those textures did not justify the computational resources required just to run the data augmentation process. This thesis proposes a novel technique for randomly altering the texture of an image. This technique is simpler and performs data augmentation in a manner that is very suited CNNs, bringing only a small computational overhead.

In chapter 4 we showed that the proposed technique successfully renders CNNs less texture-biased in a classification task with mixed textural and shape cues. We then tested our method in more demanding scenarios. The semantic segmentation experiments in domain generalization from a simulated world to a real world revealed that introducing our data augmentation technique during training leads to severe underfitting. This shows that CNN architectures do not have good and efficient mechanisms to filter out textural information. In an object detection scenario the same phenomenon was observed. Interestingly, the technique was shown to work for large object instances but the performance in smaller detections remained severely affected.

The final conclusion is that the proposed data augmentation technique, although it renders CNNs less texture-biased, it does not help their domain generalization performance due to an underfitting

phenomenon.

5.2 Future Work and Discussion

Two future major research lines were identified throughout this thesis. The first is related to the proposed image texturization technique itself. The second is related to the architectural design of the networks that perform the actual tasks.

The proposed technique tackles the problem of maximally expanding the domain of possible textures in images with a tile-based approach. This tile-based approach generates textures that are therefore periodic in nature. Introducing a noise component across space reduces this periodicity but it does not eliminate it. All generated textures still have a strong grid-like appearance with a periodicity defined by the convolutional auto-encoder architecture. Further work on this technique could seek to generalize it to non-periodic textures. A possible future technique would be a middle-ground between the ETD architecture and the AdaIN-based style transfer architecture. Key requisites to maintain would be the nonexistence of an information bottleneck in the auto-encoder, so that images can be reconstructed with fidelity, and the possibility to generate every possible texture within a certain ”texture complexity” limit. This texture ”complexity” could be defined as a certain length over which a certain set of textural features are repeated. A more formal approach to define the space of all possible textures would be to explore other regular or semi-regular tilling schemes. This would generate textures that have different translational invariances and other symmetries.

Other possibility to enhance this technique would be to study how different auto-encoder architectures, with different depths, periodicity and kernel sizes would affect the texture generation and the generalization performance of the main networks. It is entirely possible that adapting the architecture of the auto-encoder to the architecture of the main network could prove to be more efficient than choosing any auto-encoder architecture with no further considerations. This approach was not tested in this work because it was considered that the architecture of the main network played a bigger role. After all, if a network is truly texture-invariant, it should be invariant to any texturization scheme.

This avenue might be interesting from a pure research point of view but it would serve no purpose to solve the actual problem at hand until we are able to construct networks that are able to filter out textural information without an increase in size. This constitutes the second possible avenue for future research.

One area of research that is still very underdeveloped is the fundamentals of how CNN’s learn. There is no formal theory and very few useful theoretical results about the properties of CNNs during training and testing. Most computer vision research and state-of-the-art algorithms are based on empirical results, rules-of-thumb, intuition and the combination of previous techniques into a new one. This approach has produced enormous advances in performance but it has left the research community with no tools to analyse the behaviour of such architectures. Some explainability tools exist for CNNs [72, 140, 141], mostly based on the back-propagation of the gradient of the predictions with respect to the image pixels. this creates interpretable heat maps that allow humans to understand what parts of the image are used to produce the prediction. Still these techniques are very limited in their explanations and they do not

show the reasoning behind a decision. Furthermore, some of these techniques fail some basic sanity checks [142], such as producing convincing explanations from models with randomly altered weights. Some results in network robustness exist for specific architectures, such as k-Lipschitz networks, which formally guarantee robustness to local perturbations and some generalization bounds [143], but do not have any guarantee in out-of-domain robustness. The problems of Domain Generalization and controlling the Texture-Bias of CNNs are therefore devoid of any mathematical analysis tool. A first step to totally solve these problems thus requires the development of the mathematical tools that would allow us to make informed decisions with regards to network design, data augmentation and training schemes.

Until these tools are developed, the research community should keep producing better and better techniques, incrementally, based on previous work, and with a pinch of intuition and creativity. This process is time consuming, with many failed techniques but it has worked until now.

Possible techniques that may increase the generalization performance of networks and render them more shape-biased include the use of batch normalization or instance normalization to remove global textural information from the internal representations of the CNNs. These techniques have already been shown to substantially increase the generalization performance of CNNs [53] and the fact that instance normalization is used for fast style transfer indicates that this technique truly targets textural information but not shape information. These techniques eliminate global information at each layer of the networks. An improvement to Instance Normalization could perform a local normalization that eliminates textural information differently on different segments of the feature maps. This could eliminate textural information in a per-object basis at the deepest layers of the CNN. The information that is kept at the last layers would therefore be fully based on the shapes present in the images. The nature of this local normalization scheme is still undefined. One would intuitively think that the scale of the map segments where the features are normalized should start at a small value, in the low-level layers of the network, and progressively increase so that higher-level textural information is sequentially filtered out. Another option to filter out textural information would be to use a high-pass filter in the feature maps, using gaussian kernel convolutions. Instance Normalization can be though of as a high-pass filter that only filters out the first frequency component of the feature maps, which is the average activation, and normalizes the energy of all others. A high-pass filter is therefore a generalization of an instance normalization layer. Still, textural information can be useful to perform a certain task. One could imagine a network architecture where textural information and content information are processed through different pathways. The possibility of disentangling textural and content information with a local instance normalization type approach could open many doors to built better generalizing networks and even more explainable networks.

Another option to create more shape-biased networks would be a change in the nature of the convolutional layers. A convolution kernel is multiplied pixel-per-pixel with the feature maps at each position. The learned weights control how much of each pixel in the grid passes to the next layer. This grid-like processing of images may be picking up unnecessary pixel-level noise. An alternative would be to use sparse kernels, where the positions at which each value is sampled are also learneable parameters of the network. This would require a form of differentiable indexing, which can be achieved with an interpolation of the feature maps. This would introduce a considerable computational overhead during

training but once the final kernel positions are chosen, the number of floating point operations during inference would remain the same. This technique results in convolutional operations that can capture relationships between faraway regions of the image, even at the early layers of the network. In fact, the first layers of a CNN are the most computationally intensive due to the large resolution of the feature maps, so this technique could drastically decrease the amount of computation to achieve similar results. Since the positions of the weights on the feature maps are learned parameters, the network should be able to more efficiently learn complex shape representations and be able to capture texture information just as well, when needed.

A final idea to construct networks that rely more on the content of images rather than texture is to do away with convolutions all together. Convolutions severely constrain the invariances that a network can learn. The grid-based translation of the kernels is a great approach to achieve translational invariance but it fails, for instance, to achieve rotational invariance. The previous approach of varying the positions where the image is sampled with differentiable indexing can be re-utilized in a different manner. Instead of learning the kernel weights and sampling positions, and performing a convolution with the learned sparse kernel, this new technique would only learn the kernel weights, and adapt the sampling positions at inference time. This would give the networks a much bigger set of invariances, to translation, rotation, scalling, other affine transformations, and even other non-linear geometrical distortions. This would make the learning of high-level features much more efficient. During inference, the sampling positions are adapted using an optimization procedure that tries to maximize a certain objective. In classification, this objective could be the maximization of the confidence in the predictions, until one of them is selected and maximized. This procedure is expensive during inference and might not be robust. It might also be very difficult to train this kind of algorithm. However, if these challenges are resolved, an adaptive algorithm in this fashion would be very close to the actual way in which humans and animals process visual information.

Bibliography

[1] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda. A survey of autonomous driving: Common practices and emerging technologies. *arXiv:1906.05113*, 2019.

[2] Airbus. Airbus concludes attol with fully autonomous flight tests. *Airbus Press Release*, Jun 2020.

[3] V. A. Lamme, H. Super, and H. Spekreijse. Feedforward, horizontal, and feedback processing in the visual cortex. *Current Opinion in Neurobiology*, 8(4):529–535, 1998.

[4] D. H. Hubel and T. N. Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of Physiology*, 148(3):574–591, 1959.

[5] D. Marr. Vision: A computational investigation into the human representation and processing of visual information. *MIT Press*, 1982.

[6] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, 1980.

[7] M. Riesenhuber and T. Poggio. Computational models of object recognition in cortex: A review. *Technical Report C.B.C.L. Paper No. 1695, MIT*, 2000.

[8] G. Lindsay. Convolutional neural networks as a model of the visual system: past, present, and future. *Journal of Cognitive Neuroscience*, pages 1–15, 2020.

[9] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 248–255. IEEE, 2009.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[13] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

[14] S. Christian, L. Wei, J. Yangqing, S. Pierre, R. Scott, A. Dragomir, E. Dumitru, V. Vincent, R. Andrew, et al. Going deeper with convolutions. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[15] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[16] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017.

[17] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018.

[18] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

[19] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2736–2744, 2017.

[20] M. Tan and Q. V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv:1905.11946*, 2019.

[21] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[22] R. Girshick. Fast r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1440–1448, 2015.

[23] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*, pages 91–99, 2015.

[24] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[25] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Learning hierarchical features for scene labeling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1915–1929, 2012.

[26] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. Semantic image segmentation with deep convolutional nets and fully connected crfs. *arXiv:1412.7062*, 2014.

[27] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[28] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 234–241, 2015.

[29] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2961–2969, 2017.

[30] Q. V. Le. Building high-level features using large scale unsupervised learning. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8595–8598. IEEE, 2013.

[31] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv:1312.6114*, 2013.

[32] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra. Draw: A recurrent neural network for image generation. *arXiv:1502.04623*, 2015.

[33] Y. Pu, Z. Gan, R. Henao, X. Yuan, C. Li, A. Stevens, and L. Carin. Variational autoencoder for deep learning of images, labels and captions. In *Advances in Neural Information Processing Systems*, pages 2352–2360, 2016.

[34] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

[35] M. Mirza and S. Osindero. Conditional generative adversarial nets. *arXiv:1411.1784*, 2014.

[36] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv:1511.06434*, 2015.

[37] J. Donahue, P. Krähenbühl, and T. Darrell. Adversarial feature learning. *arXiv:1605.09782*, 2016.

[38] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2172–2180, 2016.

[39] E. Santana and G. Hotz. Learning a driving simulator. *arXiv:1608.01230*, 2016.

[40] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv:1604.07316*, 2016.

[41] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[42] S. Levine, P. Pastor, A. Krizhevsky, J. Ibarz, and D. Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018.

[43] N. Wang and D.-Y. Yeung. Learning a deep compact image representation for visual tracking. In *Advances in Neural Information Processing Systems*, pages 809–817, 2013.

[44] L. Bertinetto, J. Valmadre, J. F. Henriques, A. Vedaldi, and P. H. Torr. Fully-convolutional siamese networks for object tracking. In *European Conference on Computer Vision*, pages 850–865, 2016.

[45] A. Farhadi, M. Hejrati, M. A. Sadeghi, P. Young, C. Rashtchian, J. Hockenmaier, and D. Forsyth. Every picture tells a story: Generating sentences from images. In *European Conference on Computer Vision*, pages 15–29, 2010.

[46] A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[47] C. Dong, C. C. Loy, K. He, and X. Tang. Image super-resolution using deep convolutional networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(2):295–307, 2015.

[48] R. Zhang, P. Isola, and A. A. Efros. Colorful image colorization. In *European Conference on Computer Vision*, pages 649–666, 2016.

[49] L. A. Gatys, A. S. Ecker, and M. Bethge. A neural algorithm of artistic style. *arXiv:1508.06576*, 2015.

[50] A. Mordvintsev, C. Olah, and M. Tyka. Inceptionism: Going deeper into neural networks.

[51] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

[52] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[53] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv:1502.03167*, 2015.

[54] W. Liu, A. Rabinovich, and A. C. Berg. Parsenet: Looking wider to see better. *arXiv:1506.04579*, 2015.

[55] Z. Liu, X. Li, P. Luo, C.-C. Loy, and X. Tang. Semantic image segmentation via deep parsing network. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1377–1385, 2015.

[56] H. Noh, S. Hong, and B. Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1520–1528, 2015.

[57] V. Badrinarayanan, A. Kendall, and R. Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017.

[58] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2117–2125, 2017.

[59] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia. Pyramid scene parsing network. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2881–2890, 2017.

[60] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):834–848, 2017.

[61] J. R. Uijlings, K. E. Van De Sande, T. Gevers, and A. W. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104(2):154–171, 2013.

[62] J. A. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural Processing Letters*, 9(3):293–300, 1999.

[63] N. Tishby, F. C. Pereira, and W. Bialek. The information bottleneck method. *arXiv preprint physics/0004057*, 2000.

[64] J. Johnson, A. Alahi, and L. Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European Conference on Computer Vision*, pages 694–711, 2016.

[65] I. Tolstikhin, O. Bousquet, S. Gelly, and B. Schoelkopf. Wasserstein auto-encoders. *arXiv:1711.01558*, 2017.

[66] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Advances in Neural Information Processing Systems*, pages 6626–6637, 2017.

[67] A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey. Adversarial autoencoders. *arXiv:1511.05644*, 2015.

[68] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1125–1134, 2017.

[69] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2223–2232, 2017.

[70] D. Ulyanov, V. Lebedev, A. Vedaldi, and V. S. Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images. In *ICML*, volume 1, page 4, 2016.

[71] X. Huang and S. Belongie. Arbitrary style transfer in real-time with adaptive instance normalization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1501–1510, 2017.

[72] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*, pages 818–833, 2014.

[73] C. Olah, A. Mordvintsev, and L. Schubert. Feature visualization. *Distill*, 2017.

[74] W. Brendel and M. Bethge. Approximating cnns with bag-of-local-features models works surprisingly well on imagenet. *arXiv:1904.00760*, 2019.

[75] R. Geirhos, P. Rubisch, C. Michaelis, M. Bethge, F. A. Wichmann, and W. Brendel. Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness. *arXiv:1811.12231*, 2018.

[76] Painter by numbers. URL `https://www.kaggle.com/c/painter-by-numbers`. Accessed on 29.08.2020.

[77] K. L. Hermann and S. Kornblith. Exploring the origins and prevalence of texture bias in convolutional neural networks. *arXiv:1911.09071*, 2019.

[78] J. Kubilius, M. Schrimpf, A. Nayebi, D. Bear, D. L. Yamins, and J. J. DiCarlo. Cornet: Modeling the neural mechanisms of core object recognition. *BioRxiv*, page 408385, 2018.

[79] N. Parmar, P. Ramachandran, A. Vaswani, I. Bello, A. Levskaya, and J. Shlens. Stand-alone self-attention in vision models. In *Advances in Neural Information Processing Systems*, pages 68–80, 2019.

[80] G. Wilson and D. J. Cook. A survey of unsupervised deep domain adaptation. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 11(5):1–46, 2020.

[81] W. M. Kouw and M. Loog. A review of domain adaptation without target labels. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[82] M. Long, Y. Cao, J. Wang, and M. Jordan. Learning transferable features with deep adaptation networks. In *International Conference on Machine Learning*, pages 97–105, 2015.

[83] B. Sun and K. Saenko. Deep coral: Correlation alignment for deep domain adaptation. In *European Conference on Computer Vision*, pages 443–450, 2016.

[84] W. Zellinger, T. Grubinger, E. Lughofer, T. Natschläger, and S. Saminger-Platz. Central moment discrepancy (cmd) for domain-invariant representation learning. *arXiv:1702.08811*, 2017.

[85] N. Courty, R. Flamary, D. Tuia, and A. Rakotomamonjy. Optimal transport for domain adaptation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(9):1853–1865, 2017.

[86] J. Shen, Y. Qu, W. Zhang, and Y. Yu. Wasserstein distance guided representation learning for domain adaptation. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[87] G. Kang, L. Jiang, Y. Yang, and A. G. Hauptmann. Contrastive adaptation network for unsupervised domain adaptation. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 4893–4902, 2019.

[88] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky. Domain-adversarial training of neural networks. *The Journal of Machine Learning Research*, 17(1):2096–2030, 2016.

[89] E. Tzeng, J. Hoffman, T. Darrell, and K. Saenko. Simultaneous deep transfer across domains and tasks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4068–4076, 2015.

[90] E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell. Adversarial discriminative domain adaptation. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 7167–7176, 2017.

[91] M.-Y. Liu and O. Tuzel. Coupled generative adversarial networks. In *Advances in Neural Information Processing Systems*, pages 469–477, 2016.

[92] M. Ghifary, W. B. Kleijn, M. Zhang, D. Balduzzi, and W. Li. Deep reconstruction-classification networks for unsupervised domain adaptation. In *European Conference on Computer Vision*, pages 597–613, 2016.

[93] X. Mao and Q. Li. Unpaired multi-domain image generation via regularized conditional gans. *arXiv:1805.02456*, 2018.

[94] S. Sankaranarayanan, Y. Balaji, C. D. Castillo, and R. Chellappa. Generate to adapt: Aligning domains using generative adversarial networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 8503–8512, 2018.

[95] P. Li, X. Liang, D. Jia, and E. P. Xing. Semantic-aware grad-gan for virtual-to-real urban scene adaption. *arXiv:1801.01726*, 2018.

[96] A. Tao, K. Sapra, and B. Catanzaro. Hierarchical multi-scale attention for semantic segmentation. *arXiv:2005.10821*, 2020.

[97] Y. Li, N. Wang, J. Shi, X. Hou, and J. Liu. Adaptive batch normalization for practical domain adaptation. *Pattern Recognition*, 80:109–117, 2018.

[98] F. M. Cariucci, L. Porzi, B. Caputo, E. Ricci, and S. R. Bulo. Autodial: Automatic domain alignment layers. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 5077–5085. IEEE, 2017.

[99] G. Blanchard, G. Lee, and C. Scott. Generalizing from several related classification tasks to a new unlabeled sample. In *Advances in Neural Information Processing Systems*, pages 2178–2186, 2011.

[100] M. Ghifary, W. Bastiaan Kleijn, M. Zhang, and D. Balduzzi. Domain generalization for object recognition with multi-task autoencoders. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2551–2559, 2015.

[101] S. Motiian, M. Piccirilli, D. A. Adjeroh, and G. Doretto. Unified deep supervised domain adaptation and generalization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5715–5725, 2017.

[102] Q. Dou, D. C. de Castro, K. Kamnitsas, and B. Glocker. Domain generalization via model-agnostic learning of semantic features. In *Advances in Neural Information Processing Systems*, pages 6450–6461, 2019.

[103] Y. Balaji, S. Sankaranarayanan, and R. Chellappa. Metareg: Towards domain generalization using meta-regularization. In *Advances in Neural Information Processing Systems*, pages 998–1008, 2018.

[104] F. M. Carlucci, A. D'Innocente, S. Bucci, B. Caputo, and T. Tommasi. Domain generalization by solving jigsaw puzzles. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 2229–2238, 2019.

[105] N. Asadi, M. Hosseinzadeh, and M. Eftekhari. Towards shape biased unsupervised representation learning for domain generalization. *arXiv:1909.08245*, 2019.

[106] F. Qiao, L. Zhao, and X. Peng. Learning to learn single domain generalization. In *Proceedings of the IEEE/CVF conference on Computer Vision and Pattern Recognition*, pages 12556–12565, 2020.

[107] H. Wang, S. Ge, Z. Lipton, and E. P. Xing. Learning robust global representations by penalizing local predictive power. In *Advances in Neural Information Processing Systems*, pages 10506–10518, 2019.

[108] X. Yue, Y. Zhang, S. Zhao, A. Sangiovanni-Vincentelli, K. Keutzer, and B. Gong. Domain randomization and pyramid consistency: Simulation-to-real generalization without accessing target domain data. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2100–2110, 2019.

[109] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30. IEEE, 2017.

[110] Z. Xu, D. Liu, J. Yang, and M. Niethammer. Robust and generalizable visual representation learning via random convolutions. *arXiv:2007.13003*, 2020.

[111] I. Gulrajani and D. Lopez-Paz. In search of lost domain generalization. *arXiv:2007.01434*, 2020.

[112] A. Buslaev, S. S. Seferbekov, V. Iglovikov, and A. Shvets. Fully convolutional network for automatic road extraction from satellite imagery. In *CVPR Workshops*, volume 207, page 210, 2018.

[113] A. Ghosh, M. Ehrlich, S. Shah, L. S. Davis, and R. Chellappa. Stacked u-nets for ground material segmentation in remote sensing imagery. In *CVPR Workshops*, pages 257–261, 2018.

[114] A. Rakhlin, A. Davydow, and S. I. Nikolenko. Land cover classification from satellite imagery with u-net and lovasz-softmax loss. In *CVPR Workshops*, pages 262–266, 2018.

[115] J. H. Kim, H. Lee, S. J. Hong, S. Kim, J. Park, J. Y. Hwang, and J. P. Choi. Objects segmentation from high-resolution aerial images using u-net with pyramid pooling layers. *IEEE Geoscience and Remote Sensing Letters*, 16(1):115–119, 2018.

[116] P. Ulmas and I. Liiv. Segmentation of satellite imagery using u-net models for land cover classification. *arXiv:2003.02899*, 2020.

[117] K. A. Jeffries. Enhanced robotic automated fiber placement with accurate robot technology and modular fiber placement head. *SAE International Journal of Aerospace*, 6(2013-01-2290):774–779, 2013.

[118] K. Croft, L. Lessard, D. Pasini, M. Hojjati, J. Chen, and A. Yousefpour. Experimental study of the effect of automated fiber placement induced defects on performance of composite laminates. *Composites Part A: Applied Science and Manufacturing*, 42(5):484–491, 2011.

[119] B. Zieliński, M. Iwanowski, B. Salski, and S. Reszewicz. Detection of defects in carbon-fiber composites using computer-vision-based processing of microwave maps. In *Image Processing & Communications Challenges 6*, pages 245–252. 2015.

[120] C. Sacco. Machine learning methods for rapid inspection of automated fiber placement manufactured composite structures, (master thesis). *University of South Carolina Scholar Commons*, 2019.

[121] C. Zhang, J. Chen, C. Song, and J. Xu. An uav navigation aided with computer vision. In *The 26th Chinese Control and Decision Conference (2014 CCDC)*, pages 5297–5301. IEEE, 2014.

[122] Y. Lu, Z. Xue, G.-S. Xia, and L. Zhang. A survey on vision-based uav navigation. *Geo-Spatial Information Science*, 21(1):21–32, 2018.

[123] A. Cesetti, E. Frontoni, A. Mancini, P. Zingaretti, and S. Longhi. A vision-based guidance system for uav navigation and safe landing using natural landmarks. *Journal of Intelligent and Robotic Systems*, 57(1-4):233, 2010.

[124] M. Mittal, R. Mohan, W. Burgard, and A. Valada. Vision-based autonomous uav navigation and landing for urban search and rescue. *arXiv:1906.01304*, 2019.

[125] P. T. Jackson, A. A. Abarghouei, S. Bonner, T. P. Breckon, and B. Obara. Style augmentation: data augmentation via style randomization. In *CVPR Workshops*, pages 83–92, 2019.

[126] A. Coates, A. Ng, and H. Lee. An analysis of single-layer networks in unsupervised feature learning. In *Proceedings of the fourteenth International Conference on Artificial Intelligence and Statistics*, pages 215–223, 2011.

[127] Stl-10 dataset. *Stanford University*. URL `http://cs.stanford.edu/~acoates/stl10`. Accessed on 05.09.2020.

[128] naoto0804. Pytorch-adain. URL `https://github.com/naoto0804/pytorch-AdaIN`. Accessed on 26.08.2020.

[129] S. R. Richter, V. Vineet, S. Roth, and V. Koltun. Playing for data: Ground truth from computer games. In B. Leibe, J. Matas, N. Sebe, and M. Welling, editors, *European Conference on Computer Vision (ECCV)*, volume 9906 of *LNCS*, pages 102–118, 2016.

[130] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, and A. M. Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 3234–3243, 2016.

[131] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 3213–3223, 2016.

[132] S. Sinha, A. Garg, and H. Larochelle. Curriculum by texture. *arXiv:2003.01367*, 2020.

[133] Wkentaro. pytorch-fcn. URL `https://github.com/wkentaro/pytorch-fcn/`. Accessed on 25.08.2020.

[134] ZJULearning. Maxsquareloss. URL `https://github.com/ZJULearning/MaxSquareLoss`. Accessed on 12.09.2020.

[135] P. Mishra and K. Sarawadekar. Polynomial learning rate policy with warm restart for deep neural network. In *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*, pages 2087–2092. IEEE, 2019.

[136] M. Chen, H. Xue, and D. Cai. Domain adaptation for semantic segmentation with maximum squares loss. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2090–2099, 2019.

[137] Y. Cabon, N. Murray, and M. Humenberger. Virtual kitti 2, 2020.

[138] Torchvision object detection finetuning tutorial. URL `https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html`. Accessed on 14.08.2020.

[139] Cocodataset. cocodataset/cocoapi. URL `https://github.com/cocodataset/cocoapi/tree/master/PythonAPI/pycocotools`. Accessed on 16.08.2020.

[140] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv:1312.6034*, 2013.

[141] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 618–626, 2017.

[142] J. Adebayo, J. Gilmer, M. Muelly, I. Goodfellow, M. Hardt, and B. Kim. Sanity checks for saliency maps. In *Advances in Neural Information Processing Systems*, pages 9505–9515, 2018.

[143] M. Cisse, P. Bojanowski, E. Grave, Y. Dauphin, and N. Usunier. Parseval networks: Improving robustness to adversarial examples. *arXiv:1704.08847*, 2017.