Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

COPPE UFRJ

A STUDY ON DEEP CONVOLUTIONAL NEURAL NETWORKS FOR COMPUTER VISION APPLICATIONS

Roberto de Moura Estevão Filho

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: José Gabriel Rodríguez Carneiro Gomes

Rio de Janeiro
Março de 2017

A STUDY ON DEEP CONVOLUTIONAL NEURAL NETWORKS FOR COMPUTER VISION APPLICATIONS

Roberto de Moura Estevão Filho

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2017

Estevão Filho, Roberto de Moura

A Study on Deep Convolutional Neural Networks for Computer Vision Applications/Roberto de Moura Estevão Filho. – Rio de Janeiro: UFRJ/COPPE, 2017.

XII, 111 p.: il.; $29,7$cm.

Orientador: José Gabriel Rodríguez Carneiro Gomes

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2017.

Referências Bibliográficas: p. 100 – 111.

1. deep learning. 2. convolutional neural network. 3. computer vision. I. Gomes, José Gabriel Rodríguez Carneiro. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

Agradecimentos

Agradeço à minha família, aos meus professores e aos meus queridos amigos. Agradeço em especial ao meu orientador, pelo interesse em pesquisar um assunto novo; ao Igor, que foi meu parceiro nesta empreitada; à minha mãe, que perdeu o sono para me dar caronas de manhã cedo; e ao Laboratório de Sinais, Multimídia e Telecomunicações, que generosamente cedeu espaço e máquinas que eu muito utilizei durante este trabalho.

Agradeço à Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) pelo apoio financeiro.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UM ESTUDO DE REDES NEURAIS CONVOLUCIONAIS PROFUNDAS PARA APLICAÇÕES DE VISÃO COMPUTACIONAL

Roberto de Moura Estevão Filho

Março/2017

Orientador: José Gabriel Rodríguez Carneiro Gomes

Programa: Engenharia Elétrica

Este trabalho apresenta uma introdução a *deep learning* (aprendizado profundo) e suas aplicações em tarefas de visão computacional através do uso de redes convolucionais. Uma variedade de técnicas para o projeto e treino de redes neurais profundas são exploradas. Estas técnicas são demonstradas através de estudos de caso de redes neurais convolucionais aplicadas a reconhecimento de imagens. Estes estudos de caso contêm análises detalhadas de arquiteturas populares projetadas para classificação de pequenas imagens. Então, o uso de redes convolucionais para outras tarefas de visão computacional é discutido. A técnica de *class activation maps* é detalhada e implementada, e suas aplicações em localização fracamente supervisionada e interpretação de predições são apresentadas. Por fim, é explorado o uso de redes neurais profundas aplicadas à transferência de estilo automática, uma técnica que representa uma imagem ao estilo de outra. Diversas implementações de transferência de estilo para um ou múltiplos estilos foram combinadas e uma rede neural treinada em múltiplos estilos é capaz de gerar imagens estilizadas que são esteticamente agradáveis. Os programas utilizados estão disponíveis publicamente nos links `https://github.com/robertomest/convnet-study` e `https://github.com/robertomest/neural-style-keras`.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A STUDY ON DEEP CONVOLUTIONAL NEURAL NETWORKS FOR COMPUTER VISION APPLICATIONS

Roberto de Moura Estevão Filho

March/2017

Advisor: José Gabriel Rodríguez Carneiro Gomes

Department: Electrical Engineering

This work presents an introduction to deep learning and its application to computer vision tasks through the use of convolutional neural networks. Several recent techniques for designing and training deep neural networks are explored. These techniques are then demonstrated through case studies of convolutional neural networks applied to image recognition. These case studies provide detailed analysis of popular architectures designed for small image classification. Afterwards, we discuss the use of convolutional neural networks for other computer vision tasks. The *class activation maps* technique is detailed and implemented, and its applications in weakly supervised localization and prediction explanation are presented. Last, we explore how deep neural networks are applied to automatic style transfer, a technique that renders an image with the style of another. We combine different implementations of single and multi-style transfer and successfully train a multi-style neural network that generates aesthetically pleasing stylized images. The code is publicly available at `https://github.com/robertomest/convnet-study` and `https://github.com/robertomest/neural-style-keras`.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep learning [1] is a recent field of research that studies deep neural networks, their representation capabilities, and applications. This work focus on networks that were developed to process images: convolutional neural networks.

Single-layer neural networks have been used for decades and are proved to be universal approximators to smooth functions. Their application, however, was limited to small networks (with few parameters) that required handcrafted feature extraction. Only recently the use of deep (with many layers) large (with many parameters) neural networks has been successful. Previously, deep networks would fail to surpass the performance of shallow networks. There are several reasons for the recent success of such networks: novel techniques, the availability of data and computing power.

Novel techniques for training and designing deep neural networks were essential for its success. In 2006, researchers were able to train deep networks using a pre-training scheme that pre-trained individual layers one at time [2–4]. After layers were pre-trained, the whole network could be fine-tuned. Initially, pre-training was performed with undirected graphical models called restricted Boltzmann machines (RBMs) [5–7]. Later, regularized autoencoders, which are a type of neural network, were also shown to be capable of pre-training with similar results to RBMs [8].

In 2012, a deep convolutional neural network [9] was able to achieve state-of-the-art performance on large scale object recognition by learning directly from raw images [10]. This proved that deep neural networks were able to learn features that were better than carefully handcrafted features. These networks could be trained fully from scratch (without the need of pre-training) because of parameter reuse and non-saturating nonlinearities [10, 11]. They were also able to generalize well because of novel regularization techniques [10, 12].

Nowadays, deep neural networks are used for a variety of applications that include: object classification, localization, and detection [13–18]; semantic segmentation [19, 20]; super-resolution [21, 22]; automatic speech recognition [23, 24]; text-to-speech [25]; audio generation [26]; and natural language processing [27–29].

This work is an exploration of recent techniques that apply convolutional neural networks to computer vision tasks. The text is organized as follows: Chapter 2 introduces the subject of neural networks and discusses convolutional neural networks. Chapter 3 presents techniques for improving training and generalization of deep neural networks. Chapter 4 demonstrates how these techniques may be applied in the context of image classification through several case studies. Chapter 5 discusses other applications that use convolutional neural networks as feature extractors. It also includes detailed discussions on class activation maps, which is a method used for weakly supervised localization and network prediction visualization; and neural style transfer, a technique that transfers style of an image to another using features extracted from a deep convolutional network. All experiments performed were implemented in Keras [30] and Tensorflow [31], which are two popular deep learning libraries for the Python programming language. The code is publicly available at `https://github.com/robertomest/convnet-study` and `https://github.com/robertomest/neural-style-keras`. Finally, Chapter 6 presents conclusions and future study directions.

Chapter 2

Neural Networks 101

This chapter presents the basic tools for building and understanding neural networks. Its main topics are:

1. what neural networks are and what they do;
2. the fundamentals of training a neural network;
3. how to compute gradients with the backpropagation algorithm;
4. common nonlinearities used in neural networks;
5. convolutional networks and their main components.

An artificial neural network, just like in the biological case, is a group of interconnected neurons. Neurons are units capable of simple operations. The operations performed by a neuron can be arbitrarily defined, though they are most commonly a combination of linear operations (such as inner product) and nonlinear functions. These operations are performed on its inputs and return a scalar. Inputs to a neuron might be outputs from other neurons or inputs of the neural network.

2.1 Neuron Operation

The most common neuron type is one that performs a weighted sum of its inputs and applies a nonlinear function to the result. There is also a term independent of the input, that allows the neuron to be biased. Consider a neuron j with its i-th input denoted $x_{i,j}$ and the weight of the respective input $w_{i,j}$. We can compute its output o_j as

$$\begin{aligned} z_j &= \sum_{i=1}^{I} x_{i,j} w_{i,j} + b_j \\ o_j &= \sigma(z_j). \end{aligned} \tag{2.1}$$

The bias term is denoted b_j, z_j denotes the output of the linear computation of the neuron, and σ represents the nonlinear function applied to it in order to obtain the neuron output. The computation of z_j can be rewritten with the inner product

$$z_j = \mathbf{w}_j^T \mathbf{x}_j + b_j, \tag{2.2}$$

considering the vectors

$$\mathbf{x}_j = \begin{bmatrix} x_{1,j} \\ x_{2,j} \\ \vdots \\ x_{i,j} \\ \vdots \\ x_{I,j} \end{bmatrix}, \qquad \mathbf{w}_j = \begin{bmatrix} w_{1,j} \\ w_{2,j} \\ \vdots \\ w_{i,j} \\ \vdots \\ w_{I,j} \end{bmatrix}. \tag{2.3}$$

2.2 Interconnected Neurons

In most cases, neuron connectivity follows a layered pattern, where neurons are organized in layers, which receive as their input the output of the previous layer, and provide their output as input for the next layer. Neurons of the same layer are not interconnected. Considering these structural choices, all neurons in the same layer can perform their computation simultaneously. The computation for a whole layer that contains N neurons can be represented in vector notation as

$$\begin{aligned} \mathbf{z} &= \mathbf{W}^T \mathbf{x} + \mathbf{b} \\ \mathbf{o} &= \sigma(\mathbf{z}), \end{aligned} \tag{2.4}$$

with $\mathbf{x}_j = \mathbf{x}$, since all neurons share the same input, and

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1 & \mathbf{w}_2 & \cdots & \mathbf{w}_j & \cdots & \mathbf{w}_N \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_j \\ \vdots \\ b_N \end{bmatrix}. \tag{2.5}$$

Note that the nonlinearity is then applied component-wise.

A simple example of a neural network is provided in Fig. 2.1. This network is composed of three layers, with 3, 2 and 1 neurons respectively and five inputs. Since each neuron is connected to every neuron in the previous layer, this layer is

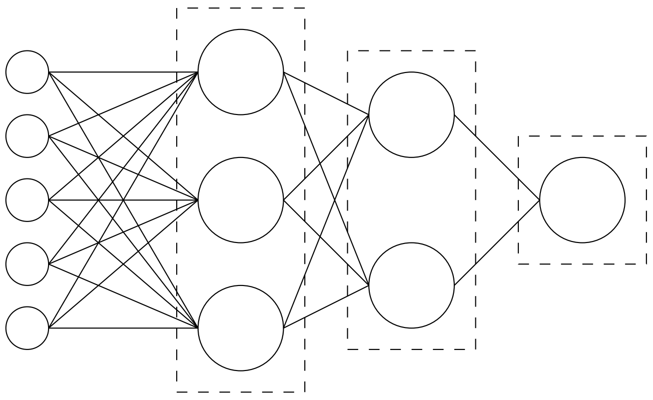


Figure 2.1: Diagram of a simple neural network. Larger circles denote neurons, smaller circles denote inputs, and dashed boxes group neurons that belong to the same layer. Solid lines represent connections.

usually called fully-connected (FC) or dense layer. The last layer is called output layer, while intermediate layers are called hidden layers. This is, therefore, a neural network that is three layers deep. Note that, while not shown in these examples, different layers may contain different neurons, that is, they may perform different operations. For example, the output layer might use a different activation function from the intermediate layers. It is also important to keep in mind that the depth of a network depends on the type of computation performed by the neuron: one could define a neuron as a unit that computes both the inner product and nonlinearity; alternatively, one could define two types of neurons, one that computes inner product, and one that applies the nonlinearity. The latter would imply a network that is twice as deep while performing the same computation. As mentioned before, the complexity of the neurons may be chosen arbitrarily and this may change the depth of the network by a constant factor [8].

2.3 Neural Networks as Function Approximators

Neural networks are parametric function approximators. Their output is a function $f(\mathbf{x}, \boldsymbol{\theta})$ of their input $\mathbf{x}$ and their parameters $\boldsymbol{\theta}$. The network weights, or parameters, can be tuned so that the output matches a desired function. This process is called learning.

An important aspect to be considered is the expressiveness of such an approximator, i.e., which functions it can approximate and how well it can approximate them. Neural networks have been proved to be able to approximate continuous functions arbitrarily well given enough neurons in a single hidden layer (also called hidden units) and under some assumptions on the activation function [32–34]. While the specifics of such proofs are outside the scope of this work, they mean that, given enough parameters, a neural network with a single hidden layer is capable of approximating any continuous function. These proofs are however not constructive: there is no indication of the number of required neurons, nor the configuration of the weights.

One could question the use of more hidden layers, considering single hidden layer networks are universal approximators themselves. However, studies show that deep networks can be exponentially more expressive than shallow[1] ones, that is, they can approximate the desired function with exponentially fewer parameters. This has been proved for logic circuits [35], but it is also conjectured to be true for other problems [8]. In practice, deep networks have been very successful at tackling several problems that shallow networks were unable to, mainly those with high-dimensional input, such as image, video, language and speech processing.

2.4 Neural Networks as Computational Graphs

Neural networks can be defined as directed acyclic graphs, where the nodes are operations that are applied to the data flowing through them [1]. In general, such operations are performed on tensors. Layers can be thought of as functions that are applied to tensors, that may be the output of another layer or the input of the graph. The total output of the graph is then a function composed of the layer functions. This idea abstracts the notion of a layer as an ensemble of neurons into a more general one: a layer can be any sort of intermediate function that transforms data.

In Fig. 2.2, we present a graph that represents a three-layer network, such as the one from Fig. 2.1. The data consists of 2-D tensors (which are conventional matrices). Note that this specific representation has two separate operations for what was previously represented as a single layer: matrix product and nonlinearity. The complexity of each operation can be chosen arbitrarily. The graph representation is very useful for determining how each part of the data flows through the network and how it influences the output (or other chosen data points). It is also useful for understanding how information flows back from the output to the parameters when the network is being trained.

2.5 Training a Neural Network

This section describes the fundamentals of training a neural network. More details are provided in Chapter 3. The process of training a neural network consists of adjusting its weights so that its output corresponds better to its expected value.

[1]The terms deep and shallow will be used to refer to networks with more than one hidden layer and a single hidden layer, respectively.

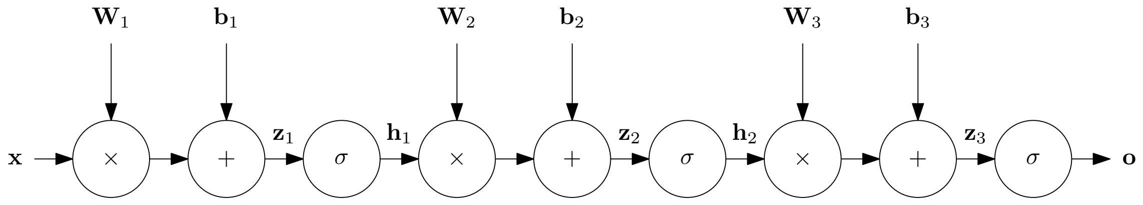


Figure 2.2: Graph representation of a three-layer neural network. Nodes with a $\times$ sign represent matrix product, while nodes with σ represent a nonlinearity.

This is also referred to as learning process or fitting the parameters. In order to train a neural network, its performance must be measured. Usually, a function that portrays how poorly the network is performing is used. It is called the loss function. The loss function $l(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})$ is a scalar that should be high if the network output $f(\mathbf{x}, \boldsymbol{\theta})$ is very different from the desired result (also called target or ground truth) $\mathbf{y}$. It should be closer to zero the closer $f(\mathbf{x}, \boldsymbol{\theta})$ and $\mathbf{y}$ are. The learning process can then be posed as an optimization problem in which the expected loss must be minimized over the parameters:

$$\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x}}[l(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})]. \tag{2.6}$$

Since the input distribution is often unknown, or the true values of such samples are not available, or both, the optimization process is instead carried out on a set of examples that is called the training set. The objective is then to minimize the loss over the training set, which should approximate the loss expected value

$$\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x}}[l(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})] \approx \min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^{N} l(f(\mathbf{x}_n, \boldsymbol{\theta}), \mathbf{y}_n). \tag{2.7}$$

2.6 Backpropagating through Graphs

The problem in Eq.(2.7) is usually solved using gradient methods, of which gradient descent and some of its variants are the most common ones. Applying gradient methods requires computing the gradient of the average loss with respect to all network parameters. It also means that all operations in the graph must be differentiable.[2] The gradients are usually computed using the backpropagation algorithm.

[2]Not all functions used in practice are strictly differentiable. Some are non-differentiable at some points. This does not seem to affect convergence, however, and networks are succesfully trained with them.

2.6.1 The Chain Rule

The chain rule states that, for functions $f, g : \mathbb{R} \to \mathbb{R}$,

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x}. \tag{2.8}$$

It means that, by knowing the derivatives of each function with respect to its input, the derivative of the composite function can be easily computed by multiplying them. The chain rule can be extended to the multivariate case: consider the functions $g : \mathbb{R}^N \to \mathbb{R}^M$ and $f : \mathbb{R}^M \to \mathbb{R}$. Denoting $\mathbf{z} = g(\mathbf{x})$, the derivative of f w.r.t[3] a single input can be computed as

$$\frac{\partial f(\mathbf{z})}{\partial x_i} = \sum_{j=1}^{M} \frac{\partial z_j}{\partial x_i} \frac{\partial f(\mathbf{z})}{\partial z_j}. \tag{2.9}$$

This relationship is usually denoted in vector notation as

$$\frac{\partial f(\mathbf{z})}{\partial \mathbf{x}} = \frac{\partial \mathbf{z}}{\partial \mathbf{x}}^T \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}, \tag{2.10}$$

where

$$\frac{\partial f(g(\mathbf{x}))}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f(g(\mathbf{x}))}{\partial x_1} \\ \frac{\partial f(g(\mathbf{x}))}{\partial x_2} \\ \vdots \\ \frac{\partial f(g(\mathbf{x}))}{\partial x_N} \end{bmatrix} \tag{2.11}$$

is the gradient vector of f w.r.t $\mathbf{x}$ and

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \frac{\partial z_1}{\partial x_2} & \cdots & \frac{\partial z_1}{\partial x_N} \\ \frac{\partial z_2}{\partial x_1} & \frac{\partial z_2}{\partial x_2} & & \\ \vdots & & \ddots & \\ \frac{\partial z_M}{\partial x_1} & & & \frac{\partial z_M}{\partial x_N} \end{bmatrix} \tag{2.12}$$

is the Jacobian matrix of $\mathbf{z}$ w.r.t $\mathbf{x}$.

2.6.2 Backprop

The backpropagation algorithm [36], or backprop, is the successive application of the chain rule in order to efficiently compute gradients along the graph. This can

[3]The term ‘with respect to’ will be from here on abbreviated as ‘w.r.t’.

be done by performing a backward pass, from the output to the input, computing gradients progressively farther from the output. Consider a function $f : \mathbb{R}^N \to \mathbb{R}$ that is a composition of L functions $f_l : \mathbb{R}^{N_l} \to \mathbb{R}^{N_{l+1}}$. The output of an intermediate function serves as input to the next one: $\mathbf{x}_{l+1} = f_l(\mathbf{x}_l)$. The output of the graph is $o = f(\mathbf{x}) = f_L(\mathbf{x}_L)$. By applying the chain rule, the gradient of f w.r.t its input $\mathbf{x}$ can be computed as

$$\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}}^T \frac{\partial o}{\partial \mathbf{x}_L}. \tag{2.13}$$

Applying the chain rule again,

$$\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial \mathbf{x}_{L-1}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{x}_L}{\mathbf{x}_{L-1}}^T \frac{\partial o}{\partial \mathbf{x}_L}. \tag{2.14}$$

Continuing with this leads to

$$\begin{aligned} \frac{\partial f}{\partial \mathbf{x}} &= \prod_{l=1}^{L-1} \frac{\partial \mathbf{x}_{l+1}}{\partial \mathbf{x}_l}^T \frac{\partial o}{\partial \mathbf{x}_L} \\ &= \prod_{l=1}^{L-1} \frac{\partial f_l(\mathbf{x}_l)}{\partial \mathbf{x}_l}^T \frac{\partial f_L(\mathbf{x}_L)}{\partial \mathbf{x}_L}. \end{aligned} \tag{2.15}$$

In short, backprop is about computing the gradient of a complex function by computing several simpler derivatives and multiplying them together. Fig. 2.3 illustrates a simple example: computing the derivative of the sigmoid function. By knowing the derivatives of the functions

$$\begin{aligned} \frac{\partial}{\partial x}\left(\frac{1}{x}\right) &= -\left(\frac{1}{x}\right)^2 \\ \frac{\partial}{\partial x}(x+y) &= 1 \\ \frac{\partial}{\partial x}(e^x) &= e^x \\ \frac{\partial}{\partial x}(xy) &= y, \end{aligned} \tag{2.16}$$

the derivative of the composed function is readily available as

$$\frac{\partial \sigma}{x} = e^{-x}\sigma(x) = \sigma(x)[1 - \sigma(x)]. \tag{2.17}$$

Despite the fact that the computation was performed assuming a scalar input, the extension for the vector case is straightforward: since the nonlinearity is applied element-wise, the gradient is also computed element-wise, and the products are element-wise scalar products.

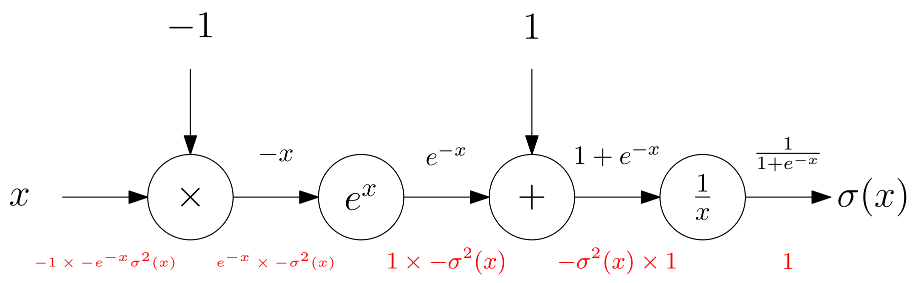


Figure 2.3: Computation of the forward and backward passes, represented in black and red fonts respectively, for a graph that computes the sigmoid function.

2.6.3 Forward and Backward Functions

While applying the chain rule directly is a valid approach to gradient computation, computing the Jacobian matrix is sometimes inefficient: consider the matrix product function $f(\mathbf{W}, \mathbf{x}) = \mathbf{W}^T\mathbf{x}$. Its inputs are $\mathbf{W} \in \mathbb{R}^{N\times M}$ and $\mathbf{x} \in \mathbb{R}^N$. The Jacobian matrix w.r.t the input vector $\mathbf{x}$ is easily obtainable

$$\frac{\partial f}{\partial \mathbf{x}} = \mathbf{W}^T. \tag{2.18}$$

The Jacobian w.r.t the weight matrix $\mathbf{W}$ can be obtained by reshaping $\mathbf{W}$ into a vector $\frac{\partial f}{\partial \mathbf{W}} \in \mathbb{R}^{MN\times M}$. However, by calculating the individual derivatives

$$\frac{\partial f_m}{\partial w_{i,j}} = \begin{cases} 0, & m \neq j \\ x_i, & m = j \end{cases}, \tag{2.19}$$

it becomes clear that the Jacobian is very sparse. If the real objective is to compute the gradient of a scalar loss function $l(f(\mathbf{x};\boldsymbol{\theta}), \mathbf{y})$ w.r.t $\mathbf{W}$, one could apply the chain rule

$$\frac{\partial l}{\partial w_{i,j}} = \sum_{m=1}^{M} \frac{\partial f_m}{\partial w_{i,j}} \frac{\partial l}{\partial f_m}. \tag{2.20}$$

Because of Eq. (2.19), Eq. (2.20) is simplified to

$$\begin{aligned} \frac{\partial l}{\partial w_{i,j}} &= \frac{\partial f_j}{\partial w_{i,j}} \frac{\partial l}{\partial f_j} \\ &= x_i \frac{\partial l}{\partial f_j} \\ \frac{\partial l}{\partial \mathbf{W}} &= \mathbf{x} \frac{\partial l}{\partial f}^T. \end{aligned} \tag{2.21}$$

Computing the gradient directly is much more efficient than the Jacobian-gradient product. This is true for many other functions and, since only loss gradients are necessary to train a neural network, it is common to define, for each operation on the graph, a function that computes the gradient of the loss w.r.t the operation inputs given the gradient of the loss w.r.t the operation output. Since the computation of the actual function performed by the graph is done by following the flow of data from the input to the output, it is called the forward function. Computing the gradient, on the other hand, follows the flow of information from the output to the input and corresponds, therefore, to computing the backward function.

2.6.4 Computing the Gradients of a Neural Network

As a more practical example of backprop, this section discusses the computation of gradients of the neural network shown in Fig. 2.2. The objective is computing gradients of the loss (not shown) w.r.t to the parameters: both the weights $\mathbf{W}_i$ and the biases $\mathbf{b}_i$. Assume the gradient of the loss, here denoted as l for simplicity, w.r.t the output $\mathbf{o}$ is given. For this particular example, the chosen nonlinearity is the sigmoid, whose gradient computation has been described in Section 2.6.2. The gradients can be computed in the following order:

$$\begin{aligned}
\frac{\partial l}{\partial \mathbf{z}_i} &= \mathbf{h}_i \odot [\mathbf{1} - \mathbf{h}_i] \odot \frac{\partial l}{\partial \mathbf{h}_i} \\
\frac{\partial l}{\partial \mathbf{b}_i} &= \frac{\partial l}{\partial \mathbf{z}_i} \\
\frac{\partial l}{\partial \mathbf{W}_i} &= \mathbf{h}_{i-1} \frac{\partial l}{\partial \mathbf{z}_i}^T \\
\frac{\partial l}{\partial \mathbf{h}_{i-1}} &= \mathbf{W}_i \frac{\partial l}{\partial \mathbf{z}_i},
\end{aligned} \tag{2.22}$$

where $\odot$ denotes element-wise product, and with $\mathbf{h}_3 = \mathbf{o}$ and $\mathbf{h}_0 = \mathbf{x}$. It is important to note that previously-computed gradients are reused several times on the computation of other gradients, and this is a major reason for the backprop efficiency.

2.7 Classifying Data with Neural Networks

Despite being able to model continuous functions, neural networks are very often used in classification tasks. Such tasks involve assigning a label to an input. This label is discrete and from a specific set. In order to perform classification, neural networks usually instead output a vector of scores with as many dimensions as there are classes. The predicted class is then chosen to be the one with the highest score.

2.7.1 Softmax and Maximum Likelihood

Output thresholding, however, is not suitable for the network training because it makes the output not differentiable. Instead, the output scores are usually transformed by a nonlinearity called softmax. The softmax function consists on interpreting the scores generated by the network as unnormalized log probabilities. The probability assigned to a class i by the network given an input is

$$p(\text{class} = i|\mathbf{x}) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}. \tag{2.23}$$

Training can be done by maximizing the likelihood of the true class c given $\mathbf{x}$:

$$\max_{\boldsymbol{\theta}} p(\text{class} = c|\mathbf{x}). \tag{2.24}$$

This is equivalent to minimizing the negative log likelihood

$$\min_{\boldsymbol{\theta}} - \log(p(\text{class} = c|\mathbf{x})). \tag{2.25}$$

Since the network output is a probability vector $\mathbf{p} = f(\mathbf{x}, \boldsymbol{\theta})$, this is equivalent to a loss function

$$l(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y}) = -\mathbf{y}^T \log(\mathbf{p}), \tag{2.26}$$

where the target vector $\mathbf{y}$ is a one-hot encoding[4] of the desired class. This is equivalent to using as target a probability distribution that assigns 100% of the probability to the correct class and minimizing the cross-entropy between the distributions. Because of that, this loss function is frequently referred to as the cross-entropy loss function.

2.8 Commonly Used Nonlinearities

The nonlinearities present in a neural network are a very important aspect of its ability to represent complex functions. Despite the fact that the universal approximation theorem does not make very specific assumptions in regard to the nonlinearities used on a network, there are a small group of functions that are very popular and have been widely used. Below is a brief discussion on some of them.

2.8.1 Sigmoid Function

The sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.27}$$

is widely used because of few interesting properties: it is bounded $0 < \sigma(x) < 1$, which means that even if weights get big during training, the outputs would remain bounded, it is differentiable everywhere, and it has a biological inspiration in the

[4]All the components of $\mathbf{y}$ are zero except the c-th component, which is equal to one.

sense that it could be interpreted as the firing rate of a biological neuron. Since the design of artificial neural networks was initially inspired by biological neural networks, it would make sense if the behaviour of an artificial neuron could be related to the behaviour of a biological one.

2.8.2 Hyperbolic Tangent Function

The hyperbolic tangent is very closely related to the sigmoid

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} = 2\frac{1}{1 + e^{-2x}} - 1, \tag{2.28}$$

and it shares some of its properties: it is also bounded and differentiable everywhere. It is usually considered as a better alternative to the sigmoid because it is centered around zero. If an input is distributed with zero mean, then the output mean is zero too. Zero mean distributions ease the optimization process [37].

2.8.3 Rectified Linear Unit

The rectifier function, usually called ReLU (for Rectified Linear Unit) [11],

$$\sigma(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{2.29}$$

is the de facto standard nowadays. It is not centered on zero and it is not differentiable at the origin. Despite that, the fact that it is not bounded from above means that it has improved gradient signal flow and networks that use this nonlinearity enjoy much better convergence properties due to smaller 'vanishing gradient' problems [10]. In fact, the use of ReLU is one of the big reasons why deeper networks are successfully trained. As seen on Section 2.6.4, the gradient of earlier layers is affected multiplicatively by the derivatives of nonlinearities of later layers. In the case of the ReLU, the gradient magnitude is not affected for inputs greater than zero. This means that these gradients are less likely to vanish. The sigmoid, on the other hand, has derivative $\frac{\partial \sigma}{\partial x} \leq 0.25$, which means that successive layers effectively kill the gradient flow. The same is true for the hyperbolic tangent with derivative $\frac{\partial \tanh}{\partial x} \leq 1$.

2.9 Convolutional Neural Network

The convolutional neural network, usually referred to as CNN or convnet, is a network that mainly utilizes convolutional layers as feature extractors. They were mainly used in image processing tasks, such as image recognition, but are now applied with great success to many other input types, which are usually high-dimensional and highly structured inputs. While there is earlier work with similar structure [38], the first CNN as we know today was used for digits recognition [9]. It featured a stack of convolutional and pooling layers and, on top of it, some fully-connected layers. In this sense, the convolutional layers were responsible for obtaining features that would be used by the vanilla neural network on top of it in order to classify the image of the digit. The key difference between a convolutional neural network and a regular neural network is that a CNN is designed to process images: it is intended to handle locally structured data. With the incorporation of this prior into the architecture, the CNN is much more efficient than a vanilla neural network: it can perform more computation using much fewer parameters. The following sections discuss the main components of a CNN: the convolutional and pooling layers.

2.10 Convolutional Layer

The convolutional layer gets its name from the operation it performs: the convolution. This layer operates on tensors, instead of the usual vectors. Consider an input $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$, which represents an image with C channels, height H and width W. The layer has a set of weights called filters, or kernels: these filters have limited width W_f and height H_f, but they always span all the input channels $\mathbf{f}_f \in \mathbb{R}^{C \times H_\mathrm{f} \times W_\mathrm{f}}$. A layer might have F of those filters, in which case the weight tensor will be $\mathbf{F} \in \mathbb{R}^{F \times C \times H_\mathrm{f} \times W_\mathrm{f}}$. Each filter will be slid across the input performing, at each position, an inner product, through which a scalar activation is obtained[5]

$$a_{f,i,j} = \mathbf{x}_{i,j} \cdot \mathbf{f}_f + b_f. \tag{2.30}$$

$a_{f,i,j}$ is the activation when the filter $\mathbf{f}_f$ is centered at the region $\mathbf{x}_{i,j} \in \mathbb{R}^{C \times H_\mathrm{f} \times W_\mathrm{f}}$ and b_f is the bias term for the filter indexed by f. If the activations are organized according to the position of the filter on the input, the result is an activation map $\mathbf{a}_f \in \mathbb{R}^{H_\mathrm{o} \times W_\mathrm{o}}$, usually referred to as a feature map. The feature maps can then be stacked to form the total output $\mathbf{a} \in \mathbb{R}^{F \times H_\mathrm{o} \times W_\mathrm{o}}$. The process is illustrated in Fig. 2.4. The dimensions along which the filter slides are usually called the spatial dimensions which, in the case of an image, are height and width. While the explanation presented considered the case of two spatial dimensions (a 2D convolution), which is the most common case, they are valid for convolutions of arbitrary dimensionality.

[5]Strictly speaking, this is correlation, instead of convolution, because the filter is not being flipped before the product. Since the filter is learned, though, there is no difference.

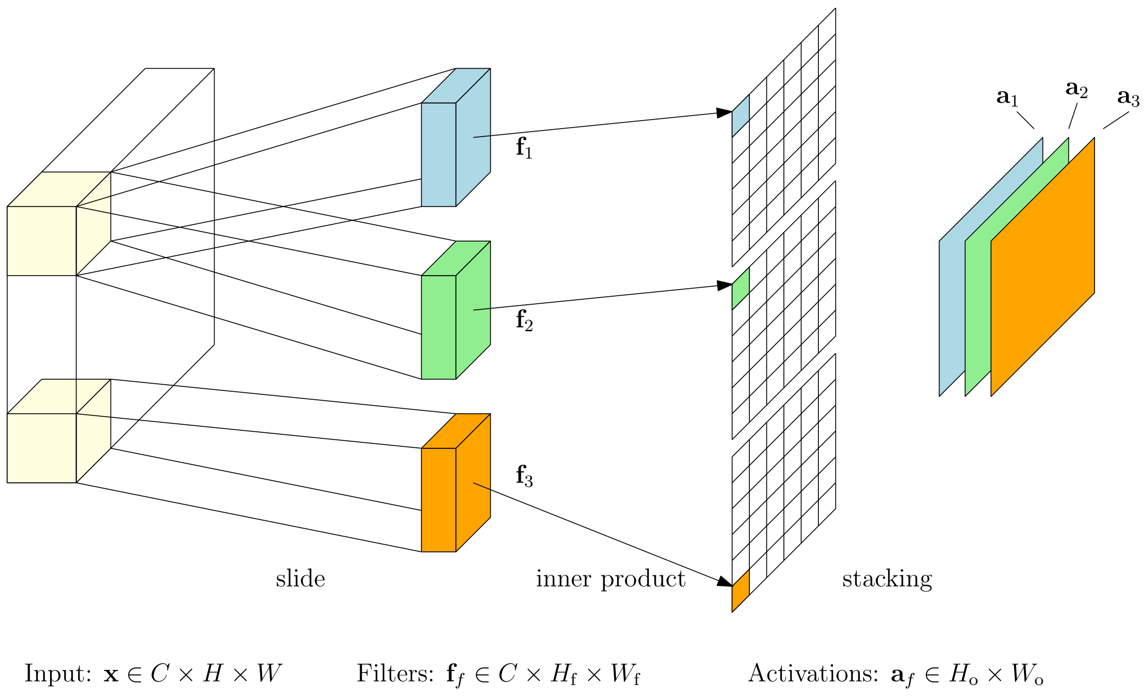

$\mathbf{a}_1$
$\mathbf{a}_2$
$\mathbf{a}_3$
$\mathbf{f}_1$
$\mathbf{f}_2$
$\mathbf{f}_3$
slide
inner product
stacking
Input: $\mathbf{x} \in C \times H \times W$
Filters: $\mathbf{f}_f \in C \times H_\mathrm{f} \times W_\mathrm{f}$
Activations: $\mathbf{a}_f \in H_\mathrm{o} \times W_\mathrm{o}$

Figure 2.4: Illustration of the computation performed on a convolutional layer.

The filter height and width are usually the same, and they are referred to as simply the filter, or kernel, size $K = H_\mathrm{f} = W_\mathrm{f}$. Compared to the input size, the filter size is usually much smaller. For example, for an input image with each size ranging from tens to hundreds of pixels, usual filter sizes range from 3 to 9.

2.10.1 Output Size and Padding

The output size differs from the input size because of border effects. Note that for filter sizes larger than 1, there are less positions at which the filter "fits" the image than the size of the image. The output size W_o for an arbitrary input spatial dimension size W can be computed as

$$W_\mathrm{o} = W - K + 1. \tag{2.31}$$

The input size is usually much larger than the filter size, so this difference is not very pronounced for a single layer. After many layers, though, the output spatial dimensions may be quite smaller. To avoid this, it is very common to pad the image so that the input and the output have the same spatial dimensions as the input: this is referred to as same or half padding.

Padding the input involves adding elements to the input. The padding size P determines how many columns/rows are added before the first column/row and after the last column/row along that dimension. The padded input size is then

$$W_{\mathrm{p}} = W + 2P. \tag{2.32}$$

To preserve the input size, the padding size should be

$$\begin{aligned} W + 2P - K + 1 &= W \\ P &= \frac{K-1}{2}. \end{aligned} \tag{2.33}$$

For even filter sizes, which are fairly uncommon, the padding size is not an integer. This can be solved by the use of asymmetric padding, where the extra half is put only before the first row/column or after the last row/column.

Most commonly, the input is padded with zeros, which is referred to as zero padding. Other examples of padding are also useful in more specific situations, such as reflection padding, where the border elements are reflected: this can arguably reduce discontinuity and leads to a smoother continuation of the input.

2.10.2 Structured Data and Convolutions

The convolutional layer can be thought of as a collection of neurons just like the fully-connected layer. Each activation on an activation map can be thought of as the output of a neuron. These neurons have a restricted receptive field, which is determined by the filter size. This means that each neuron can only "see" a small part of the input. Besides, several neurons share weights, since all of those organized on the same feature map use the same filter in their computation. Both these properties lead to a massive reduction on the number of parameters when compared to a fully-connected layer. On the fully-connected case, the weight matrix has $C \times H \times W \times F \times H_{\mathrm{o}} \times W_{\mathrm{o}}$ parameters. In contrast, a convolutional layer with the same output size uses only $F \times C \times K \times K$ parameters. As an example, let us assume typical values $C = 3, H = W = H_{\mathrm{o}} = W_{\mathrm{o}} = 32, F = 16, K = 3$. This amounts to over 50 million parameters for a fully-connected layer and only 432 parameters for a convolutional layer. While the number of parameters scales with both the input and output spatial dimensions for the fully-connected layer, the number for the convolutional layer scales only with the number of channels and the filter size.

The large reduction in the number of parameters is not miraculous. The convolutional layer design assumes that the input contains spatially localized features and that it is interesting to find those features in different parts of the input. A convolutional layer with 32 filters effectively looks for 32 different spatially localized patterns along the input. The fully-connected layer can, in turn, look for a different pattern on each neuron, and the patterns can be arbitrarily sized. On data with

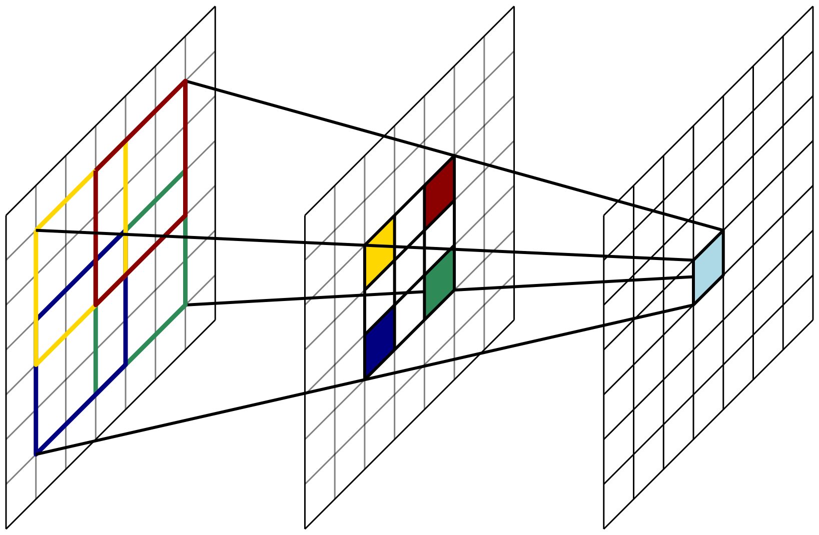


Figure 2.5: Equivalent receptive field of a neuron.

strong structure such as images, these small patterns can be very useful: they can be borders, color patterns or brightness variations. Not only are small patterns important, their location also matters, both in relation to the whole image and to themselves. By stacking layers, deeper layers can combine these patterns in order to detect more complex ones. Even if all layers have neurons with a small receptive field, neurons on deeper layers will naturally have a larger effective receptive field in relation to the input. This is illustrated in Fig. 2.5, where a neuron on the second layer has an effective 5×5 receptive field, with both layers having filter sizes equal to 3. A deep network with several convolutional layers becomes a hierarchical structure of increasingly more complex, and less localized, features as the network gets deeper.

2.11 Pooling Layers

Pooling layers are also very important components of a convolutional neural network. A pooling layer aggregates spatial information. It also works in a sliding fashion, like the convolutional layer and it also has a kernel size. It usually does not have weights: instead, it evaluates a fixed function at each window position. Since it intends to aggregate *spatial* information, it operates independently on each channel, unlike the convolution layer, where the filter took all channels into account. The most common pooling types are max and average pooling. In the case of max pooling, the output is the maximum value of all the values inside the window, whereas average pooling takes the average of those values. Pooling layers also usually use a stride larger than one: instead of sliding a window at each possible position on the input, the stride (or step) is larger, so the window skips positions. For example, operating with a

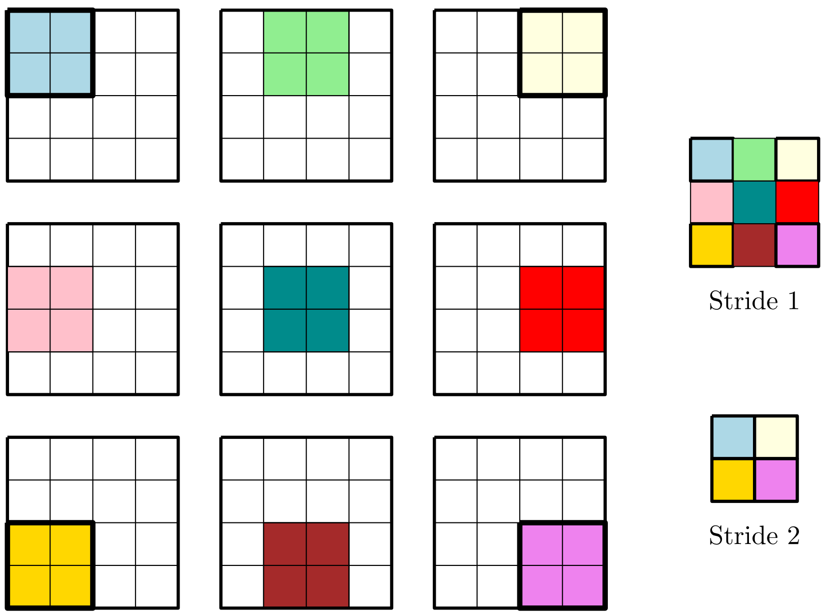


Figure 2.6: Illustration of a strided operation. Comparison between a 2×2 kernel sliding across an input with stride 1 and 2. Computations that are present on the operation with stride equal to two have a thicker border.

stride equal to two leads to function evaluation at every other position. See Fig. 2.6 for an illustration.

It is very common to perform pooling with a stride that is the same as the kernel size, that is, the windows do not overlap. The most common case is where both kernel size and stride are two, which leads to an output feature map that is half the size of the input one.

This operation is quite useful for limiting complexity: deeper layers usually contain more filters, so limiting the spatial dimensions of the feature maps is a good way to constrain complexity. It can also achieve invariance: values on different positions of the same window are summarized by the same number.

2.11.1 Max Pooling

If the pooling function is the maximum, then the pooling layer is dubbed a max pooling layer. Intuitively, taking the maximum can be useful for obtaining translation invariance: the output of a convolutional layer is said to be translation equivariant since a translated input will produce a translated version of the original output. By applying max pooling, small translations are actually aggregated into the output. This is interesting in many settings: in image classification, it does not matter where the object is in the image, all pictures of cats should belong to the class cat. By aggregating translated versions of a feature, one can detect a cat in different positions using the same computation. In general, max pooling is useful for preserving the fact that a feature has been detected in the area, even if its exact location is lost.

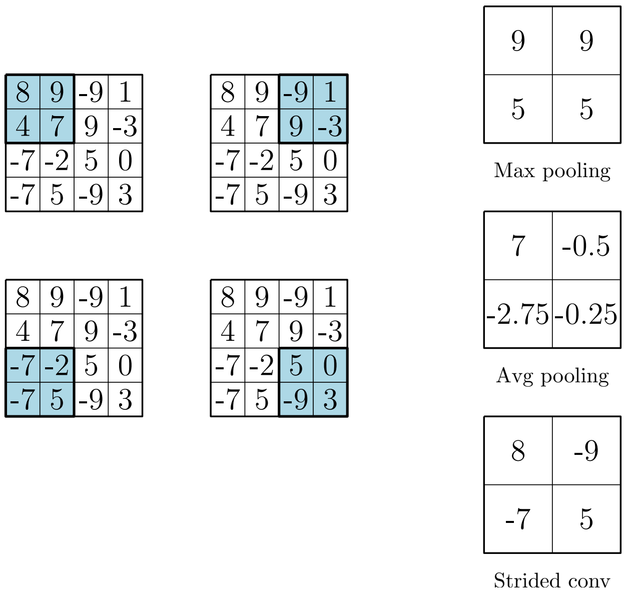


Figure 2.7: Example of different types of downsampling.

Since many scenarios have a high-dimensional input and lower dimensional output, this summarizing technique is often useful.

2.11.2 Average Pooling

An average pooling layer outputs the average of the values inside the kernel. The average is probably the most intuitive form of summarizing information. Unlike the maximum function, it does not directly "throw away" most of the information, it takes all of it into account instead. Averaging is a smooth way to downsample and serves as an indicator of how much a feature was detected in the area: if a single neuron has a high value but all others have low values, then the output would still be low; highly irregular areas with many high and low values will have medium output; and a high output indicates that the entire area has high activations.

2.11.3 Strided Convolution

Instead of applying pooling layers to subsample feature maps, one might simply use a larger stride at the convolutional layer. This is equivalent to applying a pooling function that simply picks the first value in the window, discarding all the others. See Fig. 2.7 for an example comparing different types of downsampling. Strided convolution is simple, but it has an advantage when compared to a pooling layer: instead of computing a big feature map and then reducing it, applying a strided convolution directly computes the subsampled feature map. Recent work has shown that, despite the seeming naivety of the method, good results can be obtained using only strided convolutions as the subsampling technique [39, 40].

Considerations about padding are also valid when dealing with strided convolutions. Eq. (2.33), however, is not valid for convolutions with stride larger than one. The output size of a convolutional layer with stride S, input size W, filter size K and padding P is computed as

$$W_{\mathrm{o}} = \left\lfloor \frac{W - K + 2P}{S} \right\rfloor + 1. \tag{2.34}$$

The floor operator is used since a non-integer number indicates that there is not enough room for another step at the input. It also means that different input sizes may lead to the same output size.

Chapter 3

Neural Network Training

Following the brief introduction to the neural network training process in Section 2.5, this chapter provides a more detailed discussion on several aspects of training neural networks:

1. how to evaluate the performance of a neural network;
2. how neural networks are optimized and commonly used optimizers;
3. techniques used to improve the optimization process, such as weight initialization strategies and batch normalization;
4. generalization and commonly used regularizers, such as weight decay, dropout, data augmentation, and global average pooling.

Training a neural network involves optimizing its parameters so as to minimize loss. The loss to be minimized is the average loss over the training set. The parameters are optimized with gradient-based methods. More specifically, the optimizers iteratively update parameters in order to move on parameter space towards local minima. They do so from a starting weight configuration. The weight initialization is done by assigning small random values to each weight. If, instead, all weights were initialized as zero, the symmetry would make their gradients the same, and training would fail to converge. Simple initialization strategies include sampling from Gaussian and uniform distributions with small variance. More detailed initialization strategies are discussed in Section 3.4.

Although weights are trained through an optimization algorithm, there are several other aspects of the network that are not adjusted by training and, instead, must be chosen by the designer. Some examples are: number of hidden units on an FC layer, number of filters in a convolutional layer, depth (number of layers) of a network and learning rate. These are usually called *hyperparameters*. Note that this name is not limited to numeric values that must be chosen by the designer, as *any* choice can be generically referred to as a hyperparameter. The architecture of a network, the type of pooling, or activation functions used are all hyperparameters.

3.1 Evaluating a Neural Network

While the average loss over the training set is the function that is directly optimized, the performance of the neural network should be similar for samples that are part of the input distribution even if they were not included as training examples. The ability to make reasonable predictions on samples previously unseen is called *generalization*. Usually, in order to measure generalization, a new set of samples, which the network is never trained on, is used. This set is called the test set. The test set constitutes a better representation of the operation of the network in the wild, since, after training, it would most likely operate on samples it was never trained on. The real objective then becomes obtaining a model that has good performance on the test set.

Notice that, in turn, it is more interesting to tune hyperparameters so as to maximize performance on the test set. Doing so, however, may bias the result. So, in order to tune hyperparameters, a part of the train set is usually set aside and not trained on. This set, called the validation set, is then used as a proxy to the test set. One expects that, since it is not trained on, performance on the validation set is also a good measurement of generalization. The advantage is that, even if hyperparameters are tuned in order to improve validation set performance, the test set would still be an unbiased measurement of the algorithm performance.

3.2 Optimization by Gradient Descent

Gradient descent is a simple way of optimizing the network parameters. If minimizing the average loss function over the training set is the goal, one could update the parameters with gradient descent

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta \frac{\partial \frac{1}{N}\sum_{n=1}^{N} l(f(\mathbf{x}_n, \boldsymbol{\theta}_t), \mathbf{y}_n)}{\partial \boldsymbol{\theta}_t} \\ &= \boldsymbol{\theta}_t - \frac{\eta}{N}\sum_{n=1}^{N} \frac{\partial l(f(\mathbf{x}_n, \boldsymbol{\theta}_t), \mathbf{y}_n)}{\partial \boldsymbol{\theta}_t},\end{aligned} \tag{3.1}$$

where N is the number of elements in the train set. This method will converge to a local minimum, assuming that η, called the learning rate, is greater than zero and sufficiently small. Computing the update step involves computing the gradients of the loss w.r.t every example of the training set and averaging those gradients.

This is referred to as full batch learning. In the case of deep neural networks, the computation of the loss function and its gradient can be quite expensive. For big datasets, the cost of a single update step becomes prohibitively high.

Instead of computing all the gradients and averaging them in order to update the parameters, one could, instead, update the parameters upon every gradient computation. This is equivalent to approximating the average gradient by a noisy estimate that involves a single example. This optimization procedure is called Stochastic Gradient Descent (SGD). While it is no longer certain the loss will be lower after each step, it is expected that, on average, the updates will lead the parameters to their minimum. In practice, the speedup on each step easily compensates the noisiness of the estimate, making this a much better alternative than the standard gradient descent algorithm.

A compromise between both algorithms can be achieved by averaging the gradients in a subset of the training set. This is usually referred to as mini-batch gradient descent. Instead of estimating the gradient with a single sample, the estimation is performed with a mini-batch of samples. The mini-batch size determines the compromise between the computation cost and precision of an optimization step: larger mini-batch sizes lead to more precise estimates of the true gradient, while smaller sizes lead to faster to computation. In practice, mini-batch learning is particularly interesting because of parallel computation: a mini-batch might be computed very fast by computing the gradients with optimized vector operations. This leads to more precise gradient estimation while keeping its computation quite fast. In practice, most deep learning models are trained with mini-batches. Usually, when a network is said to be optimized by SGD, it is most likely the mini-batch version, which is due to the prevalence of this method. If the gradient is computed with a single example, it is alternatively called online learning.

3.2.1 Learning Rate

The choice of an adequate learning rate is particularly important. Most notably, if the learning rate is too large, then the optimization process diverges. A very small learning rate is also not good, as it makes the already long optimization process even slower. The loss surface of a deep neural network also contains several saddle points (regions where gradients will be very small); using a small learning rate might get the weights "stuck" in such regions. The learning rate is also important with regard to the noise in the mini-batch gradient estimates: using a larger learning rate makes the noise larger, which is beneficial because it helps escaping shallow minima. Reciprocally, a smaller learning rate is interesting because the lower noise level allows the weights to settle at a point that is closer to bottom of the local minimum basin.

These complementary characteristics are usually combined through the use of a learning rate schedule: at first, a large learning rate is used for faster exploration of parameter space, thus enabling the optimizer to find good regions on the loss surface; then the learning rate is reduced, so that the weights settle closer to the location of a good minimum in this region. There are several types of learning rate schedule [40–43], the most common being schedules where the learning rate is lowered in predetermined steps (e.g., after a predetermined number of iterations).

3.3 Commonly Used Optimizers

The gradient descent algorithm performs surprisingly well when applied to neural networks. It is not, however, free from limitations that are well documented in convex optimization literature. Most notably, it is slow on ill-conditioned problems. Along high curvature directions (where gradients are very large), the algorithm tends to overshoot, causing the optimization steps to perform a zig-zagging path in parameter space. Along low curvature directions, the algorithm instead updates parameters very slowly, due to overall smaller gradients.

In convex optimization literature, there are several other methods that were developed in order to compensate for such shortcomings. In particular, these tend to estimate curvature and adapt the learning rate along each direction as to compensate for it. While second-order methods are very successful in many optimization problems, they are not commonly used when optimizing deep neural networks. Second-order methods usually require Hessian computation or line-search, which are not suitable for stochastic optimization [37]. Instead, we mostly rely on first-order methods with adaptive learning rates. There are several such methods that can be applied successfully when training neural networks. We focus on two that are particularly popular: momentum and Adam optimizers.

The momentum [44] update rule is as follows:

$$\begin{aligned} \mathbf{g}_t &= \mathrm{grad}(l(\boldsymbol{\theta}_t)) \\ \mathbf{v}_{t+1} &= \mu \mathbf{v}_t - \eta \mathbf{g}_t \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \mathbf{v}_{t+1}. \end{aligned} \tag{3.2}$$

The scalar η, again, is the learning rate and $0 < \mu < 1$ is the amount of momentum. The vector $\mathbf{g}_t$ represents the gradients and the grad operator represents their computation with a mini-batch or otherwise. The momentum update rule is characterized by updating the parameters with a temporal memory (exponentially weighted average) of the gradients, instead of the current gradient itself. If a gradient changes directions often, as it commonly happens in high curvature directions, the gradients time averages will return a small vector. In the case of very low curvature directions, the gradients will slowly add up so that the effective learning rate is increased.

A variant of momentum is Nesterov Accelerated Gradient (NAG) [44], usually called Nesterov momentum, which has the modified update rule

$$\begin{aligned}\bar{\mathbf{g}}_t &= \mathrm{grad}(l(\boldsymbol{\theta}_t + \mu \mathbf{v}_t)) \\ \mathbf{v}_{t+1} &= \mu \mathbf{v}_t - \eta \bar{\mathbf{g}}_t \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \mathbf{v}_{t+1}.\end{aligned} \tag{3.3}$$

When using Nesterov momentum, the gradient is computed with the parameters updated by the current momentum estimate, that is, the gradient is updated by 'looking ahead' and correcting the momentum by the gradient at the updated position. Nesterov momentum usually converges faster and avoids oscillations that arise when using classical momentum [44].

The Adam optimizer [45] is a popular first-order adaptive optimization method. It adapts the learning rate based on estimates of first and second moments of the gradient. The update rules are as follows

$$\begin{aligned}\mathbf{g}_t &= \mathrm{grad}(l(\boldsymbol{\theta}_t)) \\ \mathbf{m}_{t+1} &= \beta_1 \mathbf{m}_t + (1-\beta_1)\mathbf{g}_t \\ \mathbf{v}_{t+1} &= \beta_2 \mathbf{v}_t + (1-\beta_2)\mathbf{g}_t^2 \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \alpha \frac{\hat{\mathbf{m}}_{t+1}}{\sqrt{\hat{\mathbf{v}}_{t+1}} + \epsilon}.\end{aligned} \tag{3.4}$$

The vectors $\mathbf{m}$ and $\mathbf{v}$ are the first and second moment estimates, $\mathbf{g}_t^2$ is the element-wise square $\mathbf{g}_t \odot \mathbf{g}_t$, and $\sqrt{\hat{\mathbf{v}}_{t+1}}$ is the element-wise square root of $\hat{\mathbf{v}}_{t+1}$. The estimates are biased, however, since they are initialized as vectors of zeros. The unbiased estimates are then computed as

$$\begin{aligned}\hat{\mathbf{m}}_t &= \frac{\mathbf{m}_t}{1-\beta_1^t} \\ \hat{\mathbf{v}}_t &= \frac{\mathbf{v}_t}{1-\beta_2^t}.\end{aligned} \tag{3.5}$$

Empirically, Adam has good convergence properties and is faster than momentum in most cases. However, it is usually the case that, for many applications with convolutional neural networks, well-tuned momentum converges to better results. For that reason, Adam may be quite useful during hyperparameter search, leaving momentum for a narrower, and more promising, selection of models. Good standard values of β_1 and β_2 are 0.9 and 0.999, respectively [45]. Other notable optimizers include Adagrad, Adadelta and RMSProp [46–48].

3.4 Weight Initialization Strategies

The optimization process of deep neural networks is quite complex: many parameters interact simultaneously during the computation of the output, which is a very nonlinear function. Without some care, optimization may fail to converge. A very common problem is the one of *vanishing gradients*. After the discussion about backprop (see Section 2.6.2), it should be clear that gradients get transformed as they flow from the output to the input. If the Jacobian matrices have overall small eigenvalues, the gradients get smaller the farther they get from the output. This makes optimizing shallower layers difficult, and deeper layers depend on that optimization. Besides, nonlinearities also multiplicatively affect the size of gradients. For nonlinearities such as sigmoid and hyperbolic tangent, whose scalar derivatives are mostly smaller than 1, successive layers of nonlinearity also strongly contribute to vanishing gradients. This is one of the reasons why ReLU is so successful: for positive outputs, it does not interfere with gradient flow. One technique that helps alleviate vanishing gradients is careful weight initialization.

As previously discussed, weights must be randomly initialized in order to avoid symmetry and to help neurons specialize in different tasks. Simple initialization schemes include sampling from Gaussian or uniform distributions with fixed variance. The variance is then a hyperparameter that must be searched. For very deep models, using a fixed variance may lead to problems during training [49, 50]. Instead, one might initialize the weights keeping in mind data and gradient flow.

Consider a neural network composed of fully connected layers. The n-th layer transforms its input $\mathbf{x}^{(n)} \in \mathbb{R}^{N^{(n)}}$ according to equations

$$\begin{aligned} \mathbf{z}^{(n)} &= \mathbf{W}^{(n)T}\mathbf{x}^{(n)} + \mathbf{b}^{(n)} \\ \mathbf{x}^{(n+1)} &= \mathbf{h}^{(n)} = \sigma(\mathbf{z}^{(n)}). \end{aligned} \tag{3.6}$$

The network input is $\mathbf{x} = \mathbf{x}^{(1)}$ and σ represents the nonlinear function. Recall that the gradients w.r.t $\mathbf{z}^{(n)}$ can be computed as

$$\frac{\partial l}{\partial \mathbf{z}^{(n)}} = \frac{\partial f(\mathbf{z}^{(n)})}{\partial \mathbf{z}^{(n)}} \odot \left(\mathbf{W}^{(n+1)} \frac{\partial l}{\partial \mathbf{z}^{(n+1)}}\right). \tag{3.7}$$

Assume that the elements of $\mathbf{W}^{(n)}$ are sampled independently from a zero-mean distribution with variance $v_w^{(n)}$, the nonlinearity is symmetric and is working in a linear regime, the inputs and the weights are independent from each other, the input features x_i have the same variance, denoted v_x, and the bias terms are initialized at

0. The variance of the output of the layer can then be computed:

$$\begin{aligned} x_i^{(n+1)} &= \sum_{j=1}^{N^{(n)}} w_{j,i}^{(n)} x_j^{(n)} \\ \mathrm{var}[x_i^{(n+1)}] &= \sum_{j=1}^{N^{(n)}} v_w^{(n)} \mathrm{var}[x_j^{(n)}] \\ &= N^{(n)} v_w^{(n)} \mathrm{var}[x_j^{(n)}]. \end{aligned} \tag{3.8}$$

Computing the right-hand side of Eq. (3.8) recursively until the first layer is reached, one obtains

$$\mathrm{var}[x_i^{(n+1)}] = v_x \prod_{k=1}^{n} N^{(k)} v_w^{(k)}. \tag{3.9}$$

In the backward pass, the variance of gradients can be similarly computed, assuming $\frac{\partial f(z)}{\partial z} = 1$,

$$\begin{aligned} \frac{\partial l}{\partial \mathbf{z}^{(n)}} &= \mathbf{W}^{(n+1)} \frac{\partial l}{\partial \mathbf{z}^{(n+1)}} \\ \frac{\partial l}{\partial \mathbf{x}^{(n)}} &= \mathbf{W}^{(n)} \frac{\partial l}{\partial \mathbf{x}^{(n+1)}} \\ \frac{\partial l}{\partial \mathbf{x}_i^{(n)}} &= \sum_{j=1}^{N^{(n+1)}} w_{i,j}^{(n)} \frac{\partial l}{\partial \mathbf{x}_j^{(n+1)}} \\ \mathrm{var}\left[\frac{\partial l}{\partial \mathbf{x}_i^{(n)}}\right] &= N^{(n+1)} v_w^{(n)} \mathrm{var}\left[\frac{\partial l}{\partial \mathbf{x}_j^{(n+1)}}\right] \\ &= \prod_{k=i}^{L} N^{(k+1)} v_w^{(k)} \mathrm{var}\left[\frac{\partial l}{\partial \mathbf{x}_j^{(L+1)}}\right]. \end{aligned} \tag{3.10}$$

It should be clear that the variance of weights at each layer affects signal propagation multiplicatively in both forward and backward passes. In order to keep signals healthy, we would like

$$\begin{aligned} N^{(n)} v_w^{(n)} &= 1 \\ N^{(n+1)} v_w^{(n)} &= 1. \end{aligned} \tag{3.11}$$

While both constraints cannot be satisfied simultaneously (unless layers have the same number of neurons), a reasonable compromise can be achieved with

$$v_w^{(n)} = \frac{2}{N^{(n)} + N^{(n+1)}}. \tag{3.12}$$

Note that the weights are then sampled from a distribution whose variance depends on the number of inputs (fan-in) and outputs (fan-out) of the layer. This is known as *Xavier* or *Glorot initialization* [49].

The assumptions made on the derivation of the Glorot initialization are reasonable for nonlinearities like the hyperbolic tangent, which is zero-centered and has a near-linear regime around the origin, but they are not as reasonable when a nonlinearity such as ReLU is used. In this case, the variance of $z_i^{(n)}$ is

$$\mathrm{var}[z_i^{(n)}] = N^{(n)}\mathrm{var}[w_{j,i}^{(n)} x_j^{(n)}]. \tag{3.13}$$

The elements of $\mathbf{x}^{(n)}$ are not, however, zero-centered. The variance is then computed as

$$\mathrm{var}[z_i^{(n)}] = N^{(n)} v_w^{(n)} E[(x_j^{(n)})^2]. \tag{3.14}$$

Note that, since $w_{i,j}^{(n)}$ is sampled from a zero-mean symmetric distribution, $z_j^{(n)}$ also has zero mean and is symmetric. Therefore, the second moment of its rectified version $x_j^{(n+1)}$ is

$$E[(x_j^{(n+1)})^2] = \frac{1}{2}\mathrm{var}[z_j^{(n)}]. \tag{3.15}$$

This leads to a slightly different outcome

$$\mathrm{var}[z_i^{(n)}] = \prod_{k=2}^{n} \frac{N^{(n)}}{2} v_w^{(n)} \mathrm{var}[z_i^1]. \tag{3.16}$$

In order to keep the forward flow stable, we have

$$v_w^{(n)} = \frac{2}{N^{(n)}}. \tag{3.17}$$

In similar fashion, the backward propagation analysis leads to

$$v_w^{(n)} = \frac{2}{N^{(n+1)}}. \tag{3.18}$$

Instead of designing a compromise, the author of [50] argues that either strategy can be used with good results. In fact, scaling the weights in terms of the fan-in implies a constant multiplicative factor on the variance of gradients, and vice-versa. These strategies are called *MSRA* (Microsoft Research Asia) or *He initialization* [50]. The most common strategy involves scaling weights by the fan-in, but both are used in different implementations.

While these initializations are shown here for a fully-connected layer, they can be easily adapted to convolutional layers. For a convolutional layer with filter size $K^{(n)}$ and $F^{(n)}$ filters, the fan-in is $N^{(n)} = F^{(n-1)}\left(K^{(n)}\right)^2$ and the fan-out is $N^{(n+1)} = F^{(n)}\left(K^{(n)}\right)^2$. The remaining considerations are the same.

There are many other interesting initialization strategies proposed in other works:

some sample from different distributions, others compensate for different nonlinearities or even work iteratively to ensure stability of propagation [51–53]. The two initialization strategies presented here are, however, the most popular ones.

3.5 Batch Normalization

As information flows through a network, it gets transformed and its distribution changes. Therefore, each layer learns to process inputs with different characteristics. During training, however, as parameters change, the output distributions of layers also change. This phenomenon is referred to as internal covariate shift [54]. The change of the input distribution of layers slows down training, which is due to the need of the layer to adapt itself to the new distribution. One could reduce internal covariate shift by normalizing the layers inputs so that the distributions remain mostly constant. For an input $\mathbf{x} \in \mathbb{R}^D$, each dimension i can be normalized by subtracting the mean and dividing by the standard deviation according to:

$$\hat{x}_i = \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\mathrm{var}[x_i]}}. \tag{3.19}$$

The mean and variance would be computed through the train set. Note that restricting the input distribution could affect the expressiveness of the layer. In order to ensure expressiveness is kept, the final output is a scaled and shifted version of the normalized input

$$y_i = \gamma_i \hat{x}_i + \beta_i. \tag{3.20}$$

The scalar values γ_i and β_i are learnable parameters that allow the layer to undo the normalization: if $\gamma_i = \sqrt{\mathrm{var}[x_i]}$ and $\beta_i = \mathbb{E}[x_i]$, then the original input would be recovered.

Computing statistics on the entire training set, however, is impractical. Since the distribution is constantly changing, it would require frequent forward passes on the whole train set. Instead, the *Batch Normalization* algorithm [54] proposes to use mini-batch statistics, that are computed on the same forward pass that is used to compute the next optimization step. These are noisy estimates of the actual mean and variance, but they are much more efficient to compute. For a mini-batch of size

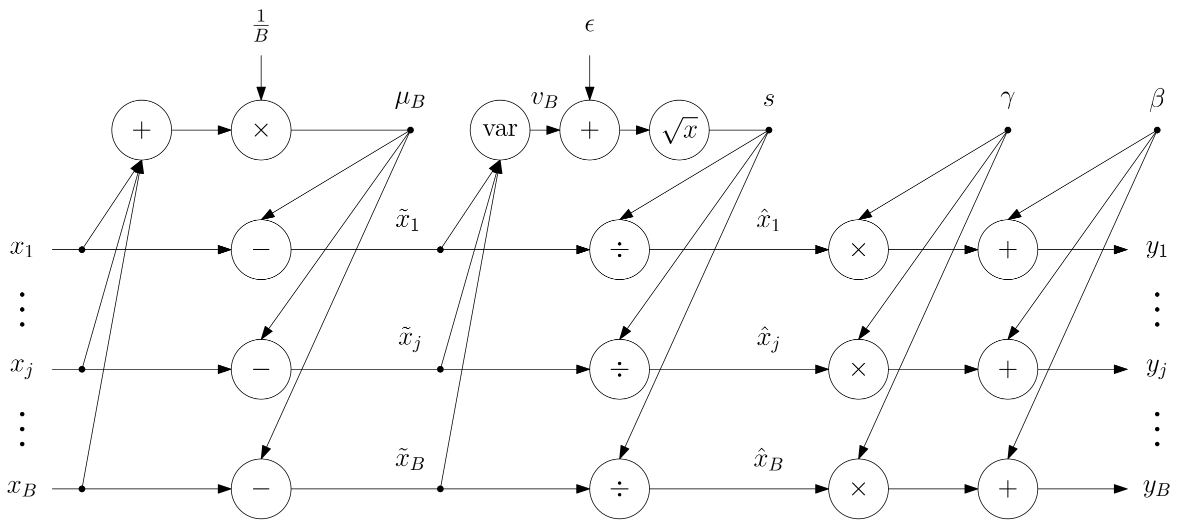


Figure 3.1: Graph representation of batch normalization. The dimension index i is omitted for improved readability.

B, each dimension, denoted as $\mathbf{x}^{(i)} \in \mathbb{R}^B$, can be normalized as follows:

$$\begin{aligned} \mu_B^{(i)} &= \frac{1}{B}\sum_{j=1}^{B} x_j^{(i)} \\ v_B^{(i)} &= \frac{1}{B}\sum_{j=1}^{B} (x_j^{(i)} - \mu_B^{(i)})^2 \\ \hat{x}_j^{(i)} &= \frac{x_j^{(i)} - \mu_B^{(i)}}{\sqrt{v_B^{(i)} + \epsilon}} \\ y_j^{(i)} &= \gamma_i \tilde{x}_j^{(i)} + \beta_i. \end{aligned} \tag{3.21}$$

The normalization is essentially the same as in Eq. (3.19), except for the approximation of the mean and variance by estimates computed using only batch samples, which are denoted by $\mu_B^{(i)}$ and $v_B^{(i)}$, respectively. The small positive scalar value ϵ is used to avoid numerical instability in the case of very small variance.

3.5.1 Batch Normalization Backpropagation

Normalizing the inputs and training the model without taking the normalization into account leads to instability. The optimization procedure should be performed with gradients being computed with normalization taken into account. Since batch normalization, usually referred to as batchnorm, is a differentiable transformation, it can be backpropagated through just like any layer. Fig. 3.1 illustrates the graph representation of batchnorm. Since the same operation is performed independently at each dimension of the input, the index i is omitted for convenience. The gradients can be computed by backpropagating through the graph as shown below:

$$\begin{aligned}
\frac{\partial l}{\partial \beta} &= \sum_{j=1}^{B} \frac{\partial l}{\partial y_j} \\
\frac{\partial l}{\partial \gamma} &= \sum_{j=1}^{B} \hat{x}_j \frac{\partial l}{\partial y_j} \\
\frac{\partial l}{\partial \hat{x}_j} &= \gamma \frac{\partial l}{\partial y_j} \\
\frac{\partial l}{\partial s} &= \sum_{j=1}^{B} \frac{\partial l}{\partial \hat{x}_j} \left(\frac{-\tilde{x}_j}{s^2} \right) \\
\frac{\partial l}{\partial v_B} &= \frac{1}{2s} \frac{\partial l}{\partial s} \\
\frac{\partial l}{\partial \tilde{x}_j} &= \frac{2\tilde{x}_j}{B} \frac{\partial l}{\partial v_B} + \frac{1}{s} \frac{\partial l}{\partial \hat{x}_j} \\
\frac{\partial l}{\partial \mu_B} &= -\sum_{j=1}^{B} \frac{\partial l}{\partial \tilde{x}_j} \\
\frac{\partial l}{\partial x_j} &= \frac{\partial l}{\partial \tilde{x}_j} + \frac{1}{B} \frac{\partial l}{\partial \mu_B}.
\end{aligned} \tag{3.22}$$

The gradient w.r.t the input can be rewritten into the more computationally efficient form

$$\frac{\partial l}{\partial x_j} = \frac{\gamma}{\sqrt{v_B + \epsilon}} \left[\frac{\partial l}{\partial y_j} - \frac{1}{B} \left(\frac{\partial l}{\partial \gamma} \hat{x}_i + \frac{\partial l}{\partial \beta} \right) \right]. \tag{3.23}$$

The parameters γ_i and β_i should also be trained by gradient descent and are usually initialized as 1 and 0 respectively (i.e. the output is simply the normalized input).

3.5.2 Batch Normalization at Test Time

It is possible to continue using batch statistics during test time, but it is not necessary and, in fact, it has some drawbacks: batch statistics are noisy and, because of their nature, they cannot be used if inference is performed on a single example. Since the model parameters are no longer being updated, one could compute the mean and variance for the whole train set and use the latter as estimates of the mean and variance of input distributions. Another approach that, despite being less accurate, is often preferred is to use moving averages of the batch statistics computed during training as estimates. With fixed mean $\mu^{(i)}$ and variance $v^{(i)}$, the batch normalization layer is simply an affine transformation

$$y_j^{(i)} = a^{(i)} x_j^{(i)} + b^{(i)} \tag{3.24}$$

with $a^{(i)} = \frac{\gamma_i}{\sqrt{v^{(i)}+\epsilon}}$ and $b^{(i)} = \frac{\beta_i \sqrt{v^{(i)}+\epsilon} - \mu^{(i)}}{\sqrt{v^{(i)}+\epsilon}}$.

3.5.3 Applying Batchnorm to a Neural Network

For a fully-connected layer, batchnorm is usually applied before the nonlinearity

$$\mathbf{x}^{(l+1)} = \sigma(\mathrm{BN}(\mathbf{W}^{(l)} x^{(l)})). \tag{3.25}$$

Note that the normalization is invariant to the bias term, so the bias term is usually not used when a layer is normalized. Instead, the parameters $\beta_i^{(l)}$ (there is one for each dimension) take the bias term place.

While batchnorm has been discussed for a fully-connected layer, its adaptation for a convolutional layer is quite straightforward: besides normalizing the input across the batch, the spatial dimensions are all jointly normalized. Essentially, each input channel is normalized by a single mean and variance (and with one γ and one β). For an input with batch size B, C channels, height H and width W, batch normalization is applied as if the input was a batch of size BHW with C dimensions. Just as in the fully-connected case, normalization is usually applied before the nonlinearity.

A batch normalized network usually enjoys faster training time, not only because of the batch normalization directly but also because batchnorm allows higher learning rates to be used without any drawbacks. Batchnorm makes the training process more robust to the initialization: networks are successfully trained for a wider range of variances in the distribution initial weights are sampled from. Batchnorm also improves generalization, reducing the need for other regularizers and data augmentation (see Section 3.6.3 for more information).

3.6 Generalization and Regularization

The previous sections were mostly devoted to improving the optimization of weights in a neural network, i.e, making them successfully learn from the training set. There is another very important aspect when training neural networks that relates to their ability to generalize. In this section, we discuss some aspects of generalization and techniques to improve it.

The lack of generalization is a problem that happens mostly because of lack of data. Consider the following example: one could train a convnet to classify images of cats and bats. Ideally, it should be able to discern a bat from a cat in any image. Imagine that, however, most pictures of cats available were taken during the day, while most pictures of bats were taken when it is dark. The network could learn to discern the images through their overall brightness: classifying bright images as cats and dark images as bats. Instead of capturing the true features that would allow it to discern between cats and bats, it would have captured features that were very discriminative only in the train set. This problem would have arisen due to lack of data, i.e., pictures of cats in dark environments and bats in brighter ones.

One important factor that influences generalization is the network's capacity. A network with very small capacity is not able to grasp complicated aspects of the data, and it may not be able to successfully generalize to unseen data. On the other hand, a network with very large capacity, if trained with limited data, may capture several aspects of the training data that are not generalizable, i.e, it may learn the noise that is present due to the finiteness of the training set. This may lead to degradation of generalization during training. This is usually characterized by the validation set or test set performance getting worse as the training set performance continues to improve. This behavior is usually referred to as *overfitting*.

As a simple example of overfitting, consider the following: a model is trained to approximate the output of the function $y = f(x)$, which has samples such as those shown in Fig. 3.2 in blue dots. Assume that the input-output relationship is unknown and only a few samples are available in the training set, shown in Fig. 3.2 as green '$\times$'s. A model with enough capacity might be able to perfectly fit the training data, with an output such as the one shown in solid lines. This, however, is not a good hypothesis when we consider the whole input distribution. As such, this output would, despite fitting the training set very well, perform poorly on a test set. This is a simple example that illustrates how a model with big capacity might perform worse than a simple one (shown in dashed lines) because the model with bigger capacity captures apparent characteristics that are only present due to lack of data.

There are several techniques that aim to improve generalization and avoid overfitting. These are called *regularization techniques* or *regularizers*. Regularizers are not meant to improve train set performance and may, in fact, decrease it. However, regularized networks usually enjoy higher test set performance. Next, we discuss some popular regularization techniques that are commonly applied to deep neural networks.

3.6.1 Weight Decay

A very common regularization technique is *weight decay*, also referred to as ℓ_2 regularization. It involves modifying the loss function to include a term that penalizes the norm of weights. For weight matrices, the norm used is the Frobenius norm.

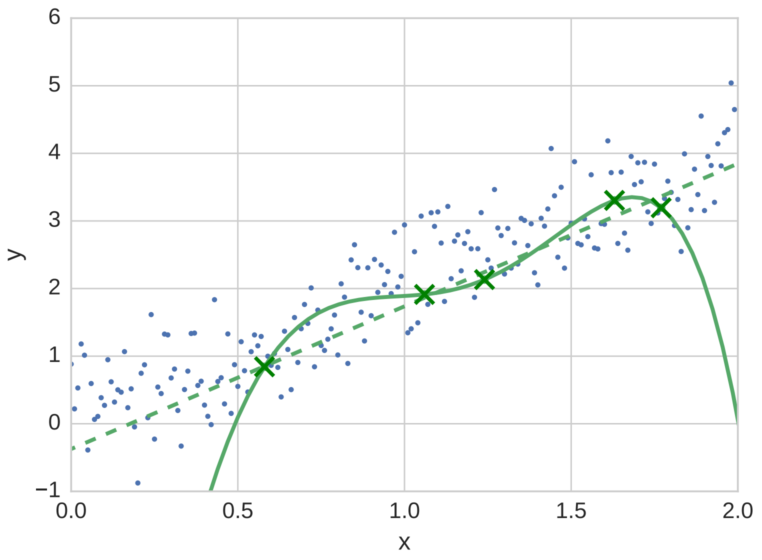


Figure 3.2: Example of overfitting. Input distribution is represented in blue dots. Training set samples are shown as green '×'s. The solid line shows the output function from a model that has overfitted. Dashed line shows a simpler hypothesis that has better generalization.

The regularized loss function is then

$$L = \frac{1}{N_B} \sum_{n=1}^{N_B} l\left(f\left(\mathbf{x}^{(n)}, \boldsymbol{\theta}\right), \mathbf{y}^{(n)}\right) + \frac{\lambda}{2} \sum_i \|\mathbf{W}_i\|_{\mathrm{F}}^2. \tag{3.26}$$

In Eq. 3.26, the scalar N_B is the size of the mini-batch, vectors $\mathbf{x}^{(n)}$ and $\mathbf{y}^{(n)}$ are the n-th input and target pair in the mini-batch, and $\mathbf{W}_i$ represents the weight matrix at layer i. The summation of the ℓ_2 norms is performed over all weight matrices of the network. The positive scalar λ controls the amount of regularization applied. The gradient w.r.t a weight matrix becomes

$$\frac{\partial L}{\partial \mathbf{W}_i} = \frac{1}{N_B} \sum_{n=1}^{N_B} \frac{\partial l\left(f\left(\mathbf{x}^{(n)}, \boldsymbol{\theta}\right), \mathbf{y}^{(n)}\right)}{\partial \mathbf{W}_i} + \lambda \mathbf{W}_i. \tag{3.27}$$

Note that, by adding the ℓ_2 norm of the weight matrices to the loss function, weight decay pushes the network into keeping its weights small. This can also be seen at the gradient expression: when moving on the opposite direction of the gradient, weight decay steers the direction of the gradient to the origin. Intuitively, small weights are not very important and, by keeping weights small, network capacity is limited. Only weights that are very important to minimize loss are allowed to grow, while others that contribute marginally would be kept small. Hopefully, weights that capture features that are not very important and contribute to overfitting are suppressed.

Weight decay is also applied in convolutional neural networks. For convolutional layers, the filter weight tensor ℓ_2 norm is computed in a similar fashion: a sum of the squares of all elements.

Penalty expressed in terms of other norms may be used for regularization as well. The ℓ_1 norm, for example, is another popular alternative that also encourages sparsity. Weight decay remains the most popular of such techniques though.

3.6.2 Dropout

Dropout [12, 55] is a regularization technique that involves randomly dropping outputs of a layer. Dropout is a layer that, for an input $\mathbf{x} \in R^N$, outputs

$$o_i = m_i x_i \tag{3.28}$$

where m_i is sampled from a Bernoulli distribution with $P(X = 0) = p$. If $m_i = 1$, the output is the same as the input. In the case of $m_i = 0$, the output is 0 and it is said that the input was dropped. p is then called the dropout probability. $\mathbf{m}$ is effectively a mask that determines which of the previous layer outputs will be dropped. The dropout probability is a hyperparameter that may be tuned, but $p = 0.5$ is a popular choice that seems to work well for most cases. The mask is sampled at every training step, so the network will have a different configuration for each mini-batch it processes during training.

Dropout helps generalization mainly because it avoids fragile co-adaptation of units, which is configured when a unit depends very strongly on other units and is not useful without them. By preventing strong co-adaptation, one could expect units to be useful on its own or to depend on a small number of units. While a long chain of units working as a set may seem useful, it is important that units are reasonably independent: a unit that serves uniquely as input to one other in the next layer most likely is not very useful. Useful units would most likely be used by a large number of units. Likewise, it is expected that a useful unit would be a combination of several different useful units of the previous layer. By randomly selecting which inputs are available, dropout enforces such robustness, which leads to better generalization.

Dropout can also be viewed as a quick way to train an exponential ensemble of networks, all of which share weights. Assuming that a network with n units is trained with dropout with probability $p = 0.5$, there are 2^n different network configurations; see Fig. 3.3 for an example of some configurations in a simple neural network. One could think of each configuration as a different (thinner) network that is randomly sampled and trained for a single step. All these networks share weights since all the units that were not dropped have their weights unaltered. Ensembles are very often used in competitions since the averaged prediction tends to be more

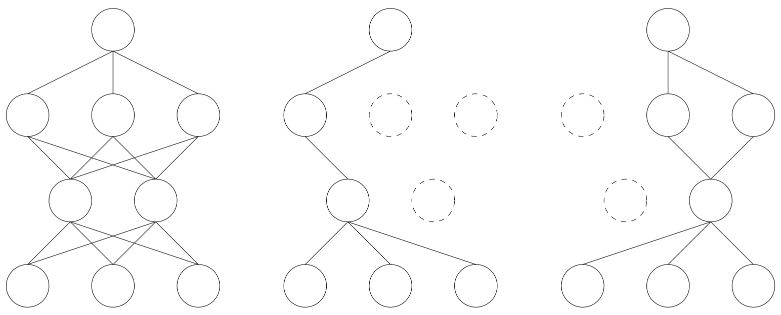


Figure 3.3: Illustration of different configurations of a network with dropout. The leftmost image represents the network with no units dropped, while the two right-side images represent configurations where some units are dropped. Dropped units are represented in dashed lines.

accurate than that of a single network [40, 42, 56].

Averaging in an ensemble is usually performed by obtaining each individual prediction and averaging their outputs. With dropout, this would be impractical, due to the enormous amount of possible configurations. Instead, the output is computed by substituting the unit's output by its expected value. For a unit trained with drop probability p, its output during test time would then be multiplied by $1 - p$. This allows the ensemble prediction to be approximated in a single forward pass. While this is only an approximation, it tends to work well in practice. In order to avoid scaling at test time, the most usual implementation of dropout scales the outputs at training time, so that the expected output is the output of the unit itself. This means that the dropout layer can be entirely ignored during test time. The behavior of a dropout layer can then be summarized as

$$o_i = \begin{cases} \frac{m_i x_i}{1-p}, & \text{if training} \\ x_i, & \text{if testing} \end{cases}. \tag{3.29}$$

A quick note about dropout during the backward pass is in order. The gradient w.r.t its input can be easily computed

$$\begin{aligned} \frac{\partial o_i}{\partial x_i} &= \frac{m_i}{p} \\ \frac{\partial l}{\partial \mathbf{x}} &= \frac{1}{p}\mathbf{m} \odot \frac{\partial l}{\partial \mathbf{o}}. \end{aligned} \tag{3.30}$$

In short, the mask is applied to the gradient w.r.t the output, just like it is applied to the input in the forward pass.

Dropout is extremely popular and is used in many networks. Most convolutional networks can be benefited by having dropout introduced. Dropout might slow down training a bit, due to its randomness, but generalization is very often improved. It also adds very little overhead during training (and none during testing), making it a very efficient regularizer.

3.6.3 Batch Normalization as a Regularizer

As discussed in Section 3.5, batch normalization is a powerful technique that eases the learning process. Another important aspect of it is that it also acts as a regularizer [54]. Its regularization properties come mostly from the noise that is introduced when normalizing with batch statistics. Since the normalization depends on the batch, the same sample may be normalized differently if it is present in different batches. Empirically, batch normalized networks generalize better and the use of other regularizers, such as dropout, can sometimes be reduced (or removed) without generalization loss.

3.6.4 Data Augmentation

We have discussed how the lack of data might introduce false structure that is not generalizable. Getting more data seems the obvious solution. Labeled data, however, is not easily obtainable and, as such, might be unavailable. One could look, instead, to artificially increase the dataset size by performing transformations on the available samples. This processed is called *data augmentation*. When dealing with classification problems, the data is augmented with class invariant transformations. In the case of image classification, this could be image cropping, mirroring, brightness perturbations, and others.

It is important to note that the augmentation used depends on the task at hand: digits cannot be flipped (horizontally or vertically) without possibly changing the class which they belong to. General objects, on the other hand, usually can be mirrored without their class being changed. Another aspect of the augmentation is that it should reflect changes that the network is expected to be invariant to. While mirroring images vertically may be a valid transformation, it may not be very useful since it is very rare for photographs to be taken upside down.

During test time, there are some alternatives to dealing with data augmentation. The most common is to simply use the sample as is; this already enjoys improved generalization from the augmentation used during training. One could, however, obtain several perturbed inputs by augmenting the original input and consider the output to be the average of the outputs obtained using these augmented inputs. While doing so generally improves results, it is more computationally intensive, so it is not commonly used.

3.6.5 Global Average Pooling

Several applications of convolutional neural networks include a topology where convolutional layers are used as feature extractors for a stack of fully-connected layers that are used for the output computation. Recall the discussion in Section 2.10.2, FC layers contain a lot of parameters for the computation they perform. This leads to a network where the bottom convolutional stack performs most of the computation, while the FC stack contains most of the parameters. This may lead to cases where the FC layers overfit. An interesting solution is to limit the number of parameters in the FC layers by performing *global average pooling* (GAP) [41].

Global average pooling is a special case of average pooling, where the kernel spans the whole feature map. For an input of size (C, H, W), the output would be simply a C-dimensional vector where each dimension is the average of all values distributed spatially on the input channel. This vector may then be fed to FC layers, where it will be processed to generate the output. Notice that, since parameter count for FC layers depends on input size, this leads to a massive reduction of parameter count.

This is particularly effective in cases where the output does not depend on spatial information directly. For example, image classification tasks simply output the class of the image: a picture of a cat should be classified as cat no matter where the cat is, or how it is oriented spatially. If the features are sufficiently descriptive, the average of how they are spatially present is usually enough. Since convolutional layers are much more efficient than fully-connected layers when dealing with spatially structured data (such as images), it is usually beneficial to put the burden on them instead of on FC layers. GAP has been adopted with great success and not only aids in generalization, but also eases computation, since it reduces the number of operations performed on FC layers.

3.6.6 Mini-Batch Regularization

Optimization via mini-batches also has regularization properties: stochastic learning usually converges to better solutions than full-batch learning. The noise present in the gradient estimates may be helpful by causing the weights to escape shallow local minima [37]. Training with small mini-batches also converges to flat solutions, while large mini-batch training tends to find solutions that are sharper [57]: the noise in the estimates obtained with small batches causes the weights to escape very sharp minima, but not flat ones. Sharp solutions correspond to higher sensitivity to the weights, which hurts generalization: the solution obtained by the optimization process is an approximation to a generalized solution; if the loss landscape around the approximate solution is very sensitive to changes in the weights, then the error in this approximation is heavily penalized. In general, the use of mini-batches is not only necessary, because of the cost of computing gradients, but also desirable, because it benefits generalization.

Chapter 4

Convolutional Network Architectures for Image Classification

Image classification is a very interesting topic with a variety of applications. While convnets had success in classifying small images (with sizes around 28×28 pixels), such as digits since 1998 [9], it was not until 2012 that a deep convolutional neural network was able to surpass other techniques that involved complicated pipelines of handcrafted features for classification of complex objects on large images (on the order of 256×256 pixels) [10]. This is due to several new techniques (many of which were discussed in Chapters 2 and 3), as well as more computing power, particularly due to the use of graphics cards (GPUs). Since then, convnets have been a hot research topic and several successful architectures have been proposed.

This chapter aims to discuss several such architectures, which were selected not only because they obtained competitive results on popular benchmark datasets, but also because they introduced interesting ideas in their design. In particular, this chapter presents:

1. discussion on classification of small images;
2. benchmark datasets used to evaluate presented architectures;
3. empirical verification of the effectiveness of techniques discussed previously such as dropout, learning rate scheduling, batch normalization, and others;
4. the Network in Network architecture and the usefulness of 1×1 convolutions;
5. a VGG-style network;
6. residual learning, pre-activation residual blocks and wide residual networks;

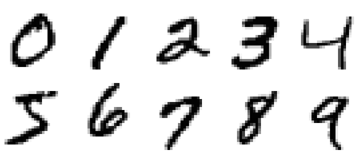


Figure 4.1: MNIST digits sampled from the dataset.

7. densely connected convolutional networks and their variants.

The architectures presented are used for classification of small images. It is important to note, though, that the ideas that made them successful are also applicable to large images[1]. The focus on small image classification is due to practical concerns: large image classification requires much more computing power, not only because the networks are much larger, but also because the datasets themselves are very big (some contain millions of images). Since computing resources available were limited, a study involving smaller images proved more manageable.

Classification performance is usually evaluated through classification accuracy, which is defined as the ratio between the number of correctly classified samples and the total number of samples in the dataset. Alternatively, we report results in the form of classification error rate, which is defined as the ratio between the number of misclassified samples and the total number of samples in the dataset or, equivalently, error rate $= 1 -$ accuracy.

4.1 Benchmark Datasets

Performance will be measured on four benchmark datasets: MNIST [9], SVHN [58], CIFAR-10 and CIFAR-100 [59]. These are described below.

The Modified National Institute of Standards and Technology (MNIST) dataset is a database of handwritten digits[2]. The digits were normalized to fit a 20×20 pixels frame, which is centered on a grayscale 28×28 pixels image. The training and test sets are composed of 60000 and 10000 samples, respectively. Fig. 4.1 presents some samples from the dataset. MNIST is a relatively "well-behaved" dataset, where digits are present in similar sizes and angles. The main source of variation is the handwriting style of different writers.

The Street View House Numbers (SVHN) dataset[3] also contains digits. The digits, however, were extracted from house numbers in Google Street View. The digits are presented in 32×32 three-channel images (channels representing R, G,

[1]Most of the architectures presented here have versions that are very successful in large image benchmark datasets. In fact, several were designed first for large-scale classification tasks and later adapted to these smaller datasets.

[2]The MNIST dataset is available for download at: `yann.lecun.com/exdb/mnist/`

[3]The SVHN dataset is available for download at: `ufldl.stanford.edu/housenumbers/`.

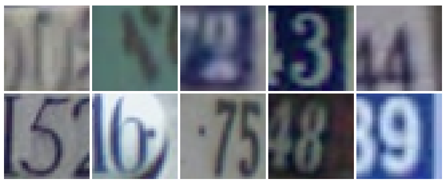


Figure 4.2: SVHN images sampled from the dataset.

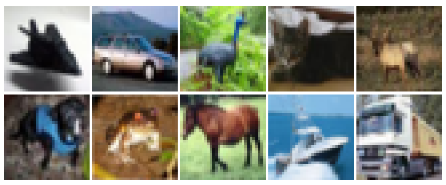


Figure 4.3: Images from the CIFAR-10 dataset.

and B colors, respectively) and, in order to preserve aspect ratio, the frames around the digits are extended if necessary. This leads to some images containing parts, or entire, digits on the sides. These distracting digits should not be classified: the task involves classifying only the center digit. The database is divided in training, test and additional sets that contain around 73000, 26000 and 500000 images, respectively. The additional set contains samples that are considered to be easier since they were automatically detected with a high detection threshold [58]. While SVHN dataset digits are mostly printed (and, therefore, do not possess the variability of handwritten digits), they are subjected to several distortions from the image capturing process. The database is also much larger than MNIST, which is attractive when training deep neural networks. Fig. 4.2 presents some samples from the database. Notice the presence of distracting digits on the sides of some images.

The Canadian Institute For Advanced Research (CIFAR) datasets[4] are both labeled subsets of the 80 Million Tiny Images dataset [60]. These datasets are composed of 32×32 natural RGB images for object recognition. The CIFAR-10 dataset is composed of 60000 images separated in 10 classes, with 6000 images of each class. These images are divided in training and test sets with 50000 and 10000 images, respectively. The classes included in the dataset are varied, including animals and man-made vehicles. The images were selected from various viewpoints and perspectives, with the restriction that there must be only one main object in the scene and that it is clearly distinguishable. The CIFAR-100 dataset is similar, but there are 60000 images divided in 100 classes, with 600 images of each class. This makes the task significantly harder: there are ten times more classes and ten times fewer examples per class. See Fig. 4.3 for an example of images taken from the CIFAR-10 dataset; images from CIFAR-100 are similar.

[4]The CIFAR datasets are available at: `www.cs.toronto.edu/cifar.html`.

Table 4.1: Baseline network: architecture and hyperparameters.

Operation	Kernel size	Stride	Feature maps	Padding	Nonlinearity
Network - $28 \times 28 \times 1$ input					
Convolution	5×5	1	32	SAME	ReLU
Average Pooling	2×2	2			
Convolution	5×5	1	64	SAME	ReLU
Average Pooling	2×2	2			
Fully-connected	200 *units*				ReLU
Dropout	*drop probability* $= 0.5$				
Fully-connected	10 units				Softmax
Preprocessing	Normalize channel mean and variance				
Optimizer	Nesterov momentum ($\eta = 0.1$, $\mu = 0.9$)				
Weight decay	10^{-5}				
Batch size	128				
Epochs	30				
Learning rate schedule	Divide by 10 after 20 epochs				
Weight initialization	He initialization (Gaussian)				

4.2 Baseline Network

As an introduction, we first discuss a simple convolutional network that will serve as a baseline. This network features common architectural principles and is composed of some convolutional and pooling layers that act as feature extractors for fully-connected layers that are used to make the final prediction. This network will be evaluated on MNIST, and, since its training is very fast, it will be used to exemplify concepts discussed in previous chapters.

The network has two convolutional layers, with 32 and 64 filters, respectively. Each of them is followed by ReLU and Average Pooling. On top of them, there are two fully-connected layers: a hidden layer with 200 units and dropout with 50% drop probabiliy and an output layer with 10 units (one for each MNIST class) and softmax nonlinearity. The network is also regularized with weight decay. The architecture and hyperparameters of the network are detailed in Table 4.1.

The network is trained with cross-entropy loss for 30 epochs. The learning rate was set according to a schedule: for the first 20 epochs, the optimizer used a learning rate of 0.1; after 20 epochs, the remaining 10 used a learning schedule of 0.01 (ten times smaller). A validation set of 10000 samples was used to tune hyperparameters. After the hyperparameters were tuned, the network was trained in all 60000 training samples. Using all samples is important due to the limited size of the dataset. The test set performance obtained after the 30 epochs is 99.4% accuracy (0.6% classification error rate).

In order to evaluate the regularization properties of dropout, the same network was also trained without dropout on its hidden FC layer. The network without dropout had worse generalization, obtaining 0.7% error rate. Their training curves

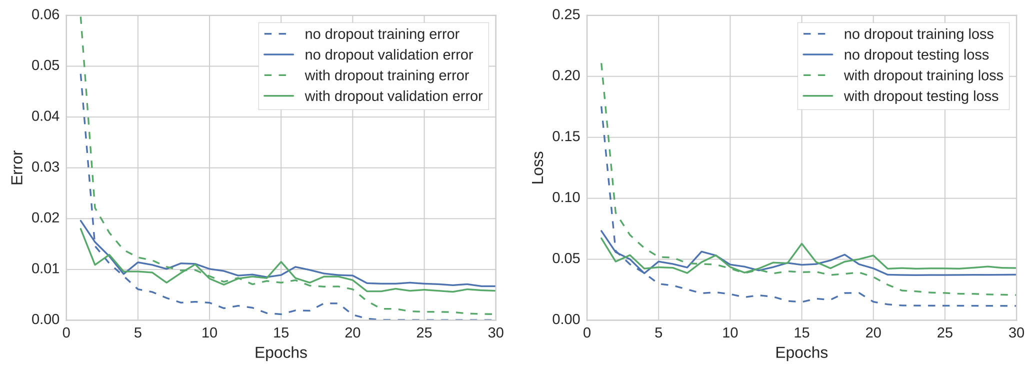


Figure 4.4: Baseline network training curves with and without dropout.

are shown in Fig. 4.4. Despite having higher training error (due to dropout noise), the network regularized with dropout has improved generalization, as expected.

It is also interesting to note the learning dynamics in relation to the learning rate. By using a schedule, the network is able to use a higher learning rate during the first stage, which makes for a faster exploration of the parameters space. On the second stage, a smaller learning rate enables it to settle closer to the minimum. This is clearly noticeable on the training curves: when the learning rate is lowered, both training and testing errors converge to a smaller value. Aside from that, a smaller learning rate also makes the performance more stable, i.e., the metrics fluctuate much less after convergence. This is particularly important when there is no validation set, as the training process is "blind": the parameters used are the ones obtained at the end of training, since picking the model based on test accuracy would bias the result.

As a practical example, the network was also trained with fixed learning rates: 0.1 and 0.01, the values of the two phases of the schedule. In both cases, the error rate is worse, with values around 0.8%. The training curves are shown in Fig. 4.5. It is also noticeable that the testing error is much more unstable when the learning rate is higher. While the effects on this small network might not appear significant, schedules are very important when training larger convnets.

4.3 Using a Neural Network as an Activation Function: Network in Network

We next discuss the Network in Network (NiN) architecture [41]. The feature extraction of convnets is usually performed by a combination of convolutional and pooling layers. The convolutional layers perform convolution, which is a linear operation, followed by a nonlinear function. The nonlinear function is essential, for it enables the model to represent nonlinear relationships. While simple nonlinear func-

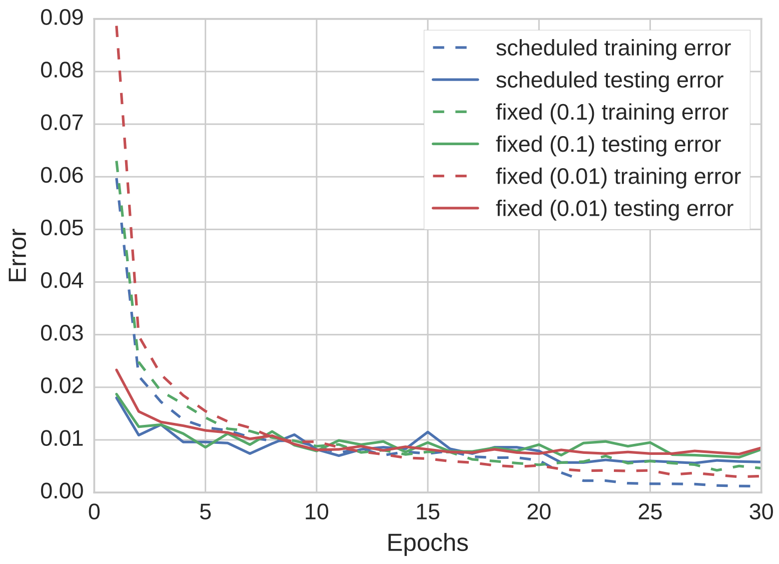


Figure 4.5: Baseline network performance with different learning rates.

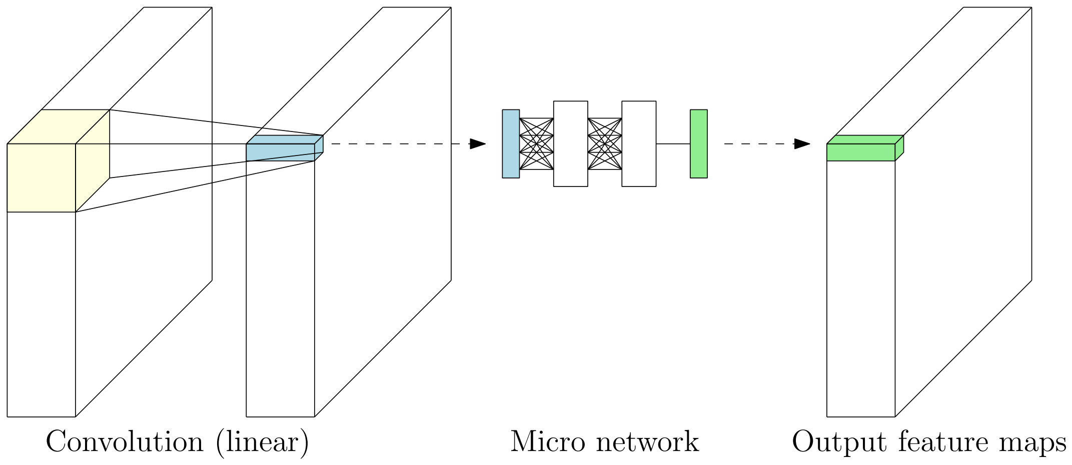


Figure 4.6: Network in network layer: convolution with a micro network as activation function.

tions such as ReLU are quite successful, one might wonder if using more complex nonlinearities might enhance the representation power of the layer. In fact, more complex nonlinear mappings have obtained competitive results [61]. Since neural networks are universal approximators, they should be able to perform more complex nonlinear mappings than the usual nonlinearities. For that reason, [41] proposes to substitute element-wise nonlinearities by “micro networks” that will be slid across the feature maps, mapping each input vector (a vector that contains the activations of the different filters in each position) to an output vector. The micro network can be backpropagated through and its weights should be learned as well. The idea is illustrated in Fig. 4.6.

This network in network module can be easily implemented using convolutional layers: the micro network with L layers is equivalent to L 1×1 convolutional layers.

In short, a network in network module is a convolutional layer with a larger kernel size, followed by 1×1 convolutional layers, with each convolutional layer being followed by ReLU nonlinearities.

Another important aspect of the architecture is the use of global average pooling. Instead of using a fully-connected layer to obtain class predictions, the last convolutional layer outputs the same number of feature maps as there are classes. These maps are averaged and these are considered to be the scores for each class. As discussed in Section 3.6.5, global average pooling improves generalization, which is verified in this model [41].

In order to showcase the properties of Batch Normalization discussed in the previous chapter, we propose a modified NiN architecture that includes batch normalization, which we call Batch Normalized Network in Network (BN-NiN). In order to compare it with the original model, the architecture remains mostly unchanged, but some hyperparameters were changed to take advantage of batch normalization.

4.3.1 Classification on Benchmark Datasets

We first evaluate the efficacy of the NiN model on the CIFAR datasets. The architecture used is mostly unchanged from [41]: the network is composed of three NiN modules, separated by pooling and dropout layers. After the last module, a global average pooling layer is used to obtain class scores, which go through a softmax nonlinearity. Each NiN module uses two 1×1 convolutional layers. The architectures used for both CIFAR-10 and CIFAR-100 are essentially the same, the only difference being the number of filters on the last convolution, which is 10 and 100, respectively (the same as the number of classes in each dataset). We also evaluate the BN-NiN model with the same number of filters in each convolutional layer, whose only additional complexity is due to batch normalization.

As in [41], the networks were trained on all 50000 training samples and evaluated on the 10000 test samples. The regular NiN model was trained for 260 epochs [41]. It was optimized using Nesterov momentum with a scheduled learning rate with an initial value of 0.025, which is annealed to 0.01 after 200 epochs and to 0.001 after 230 epochs. The BN-NiN model was instead trained for 250 epochs. The learning rate was also scheduled differently: it has an initial value of 0.1 and is halved after every 25 epochs. In both cases, the dataset was preprocessed by normalizing each input channel (R, G, and B) by its mean and variance. We call this "per channel mean and variance normalization". This is simpler preprocessing than the one used in the original implementation, which used global contrast normalization and ZCA whitening [59] as in [61]. We found that the network performance is not impaired, so we opted for simpler preprocessing instead. The complete architecture

Table 4.2: Network in Network: architecture and hyperparameters.

Operation	Kernel size	Stride	Feature maps	Padding	Nonlinearity
Network - Input $32 \times 32 \times 3$					
NiN Module	5×5		MNIST: 128, 96, 48 CIFAR: 192, 168, 96 SVHN: 128, 96, 64		
Max Pooling	3×3	2		SAME	
Dropout	*drop probability* = 0.5				
NiN Module	5×5		MNIST: 128, 96, 48 CIFAR: 192, 192, 192 SVHN: 320, 256, 128		
Average Pooling	3×3	2		SAME	
Dropout	*drop probability* = 0.5				
NiN Module	5×5		MNIST: 128, 96, 10 CIFAR: 192, 192, 10(100) SVHN: 384, 256, 10		
Global Average Pooling					softmax
NiN Module - K, F_1, F_2, F_3					
Convolution	K	1	F_1	SAME	ReLU
Convolution	1	1	F_2		ReLU
Convolution	1	1	F_3		ReLU
Preprocessing	Per channel mean and variance normalization				
Optimizer	Nesterov momentum $\mu = 0.9$ MNIST: $\eta = 0.02$ CIFAR: $\eta = 0.025$ SVHN: $\eta = 0.025$				
Weight Decay	10^{-4}				
Batch size	MNIST: 128 CIFAR: 64 SVHN: 64				
Epochs	MNIST: 60 CIFAR: 260 SVHN: 40				
Learning rate schedule	MNIST: Divide by 2 after 40 epochs and by 5 after 50 epochs CIFAR: Divide by 10 after 200 and 230 epochs SVHN: Divide by 10 after 20 and 30 epochs				
Weight initialization	Gaussian with 0.05 standard deviation				

and hyperparameters are detailed in Tables 4.2 and 4.3.

Since there is no validation set, the models are trained and test set performance is reported with the final configuration. For the CIFAR-10 dataset, the NiN model obtained 10.34% error rate on the test set. The BN-NiN model obtains 8.85% error rate. On the CIFAR-100 dataset, they obtain 35.13% and 30.66%, respectively. The training curves are shown in Figs. 4.7 and 4.8.

The models are also evaluated with data augmentation. The data augmentation scheme applied is widely used [40, 41, 61–64]: images are padded with 4 pixels on each side and a random 32×32 crop is selected. The crop is also flipped horizontally with 50% probability. The augmented datasets are denoted CIFAR-10+ and CIFAR-100+. No changes are made to any hyperparameters when training on the augmented datasets. The NiN and BN-NiN models obtain error rates of 8.36% and 7.14% on CIFAR-10+ and of 31.29% and 28.96% on CIFAR-100+, respectively. The

Table 4.3: Batch Normalized Network in Network: architecture and hyperparameters.

Operation	Kernel size	Stride	Feature maps	Padding	Nonlinearity
Network - Input $32 \times 32 \times 3$					
NiN Module	5×5		MNIST: 128, 96, 48 CIFAR: 192, 168, 96 SVHN: 128, 96, 64		
Max Pooling	3×3	2		SAME	
Dropout	*drop probability* $= 0.5$				
NiN Module	5×5		MNIST: 128, 96, 48 CIFAR: 192, 192, 192 SVHN: 320, 256, 128		
Average Pooling	3×3	2		SAME	
Dropout	*drop probability* $= 0.5$				
NiN Module	5×5		MNIST: 128, 96, 10 CIFAR: 192, 192, 10(100) SVHN: 384, 256, 10		
Global Average Pooling					softmax
NiN Module - K, F_1, F_2, F_3					
Convolution	K	1	F_1	SAME	BN-ReLU
Convolution	1	1	F_2		BN-ReLU
Convolution	1	1	F_3		BN-ReLU
Preprocessing	Per channel mean and variance normalization				
Optimizer	Nesterov momentum ($\eta = 0.1$, $\mu = 0.9$)				
Weight Decay	10^{-4}				
Batch size	MNIST: 128 CIFAR: 64 SVHN: 64				
Epochs	MNIST: 30 CIFAR: 250 SVHN: 40				
Learning rate schedule	MNIST: Divide by 10 after 20 epochs CIFAR: Divide by 2 every 25 epochs SVHN: Divide by 10 after 20 and 30 epochs				
Weight initialization	He initialization (Gaussian)				

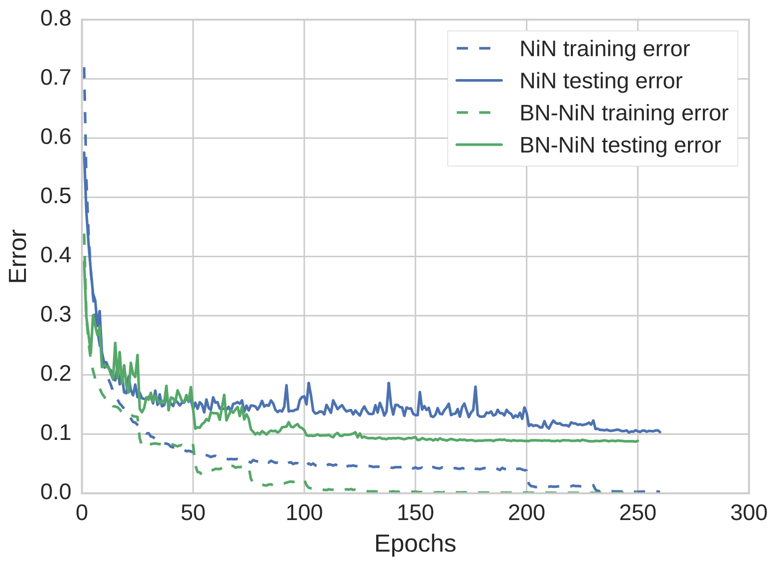


Figure 4.7: Training curves for the NiN and BN-NiN models on CIFAR-10.

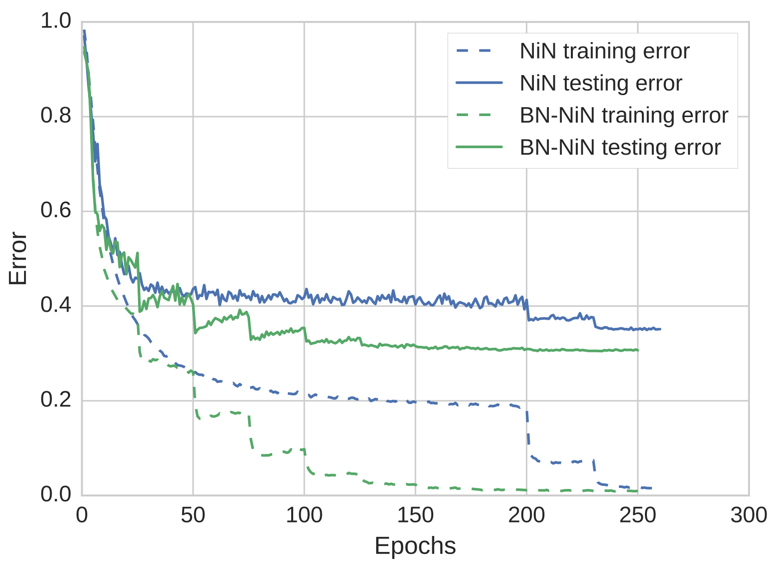


Figure 4.8: Training curves for the NiN and BN-NiN models on CIFAR-100.

training curves are shown in Figs. 4.9 and 4.10.

We next evaluate performance on the SVHN dataset. Like in [61], for each class, 400 and 200 samples are withheld from the training and extra sets to form a validation set, and all other samples are used for training. The validation set is never trained on and is used for hyperparameter tuning and model selection. The dataset is normalized by the same process described for the CIFAR datasets. Both networks (NiN and BN-NiN) are trained for 40 epochs. The learning rate is scheduled by dividing it by 10 after 20 epochs and again after 30 epochs. After training is finished, the model with the best validation accuracy is evaluated on the test set. The NiN and BN-NiN models achieve 2.39% and 1.78% error rate on the test set, respectively. The training curves are shown in Fig. 4.11.

Finally, the models are evaluated on the MNIST dataset. The NiN model requires a more careful tuning of the hyperparameters and is trained for 60 epochs. The learning rate is scheduled by dividing it by 2 after 40 epochs and by 5 after 50 epochs. The BN-NiN model presents no convergence issues and is trained for only 30 epochs. The schedule is also simpler, with the learning rate being divided by 10 after 20 epochs. At the end of training, they achieve 0.49% and 0.33% error rate on the test set, respectively. Training curves are shown in Fig. 4.12.

4.3.2 The Power of 1×1 Convolutions

Despite not being able to grasp spatial structure due to their restricted receptive field, 1×1 convolutions are very useful. Most notably, they are extremely efficient: the number of parameters as well as the amount of computation are both dependent on the receptive field size, which means that even small filter sizes such as 3×3

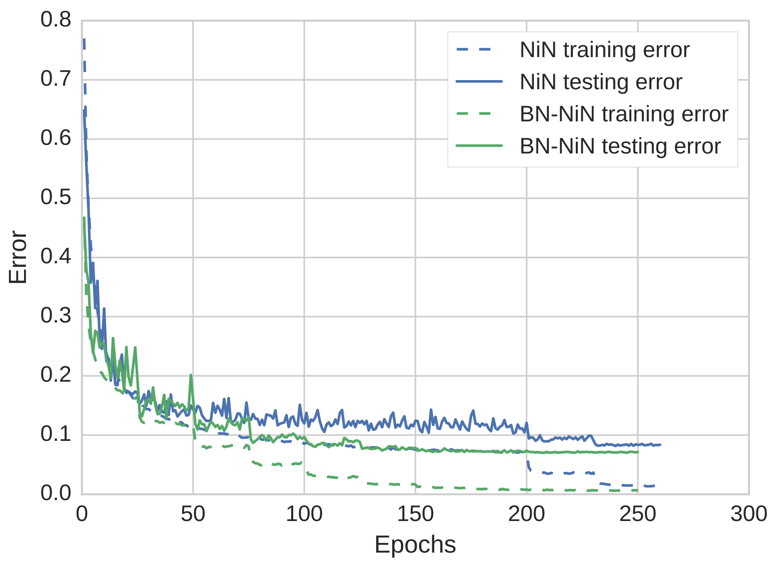


Figure 4.9: Training curves for the NiN and BN-NiN models on CIFAR-10+.

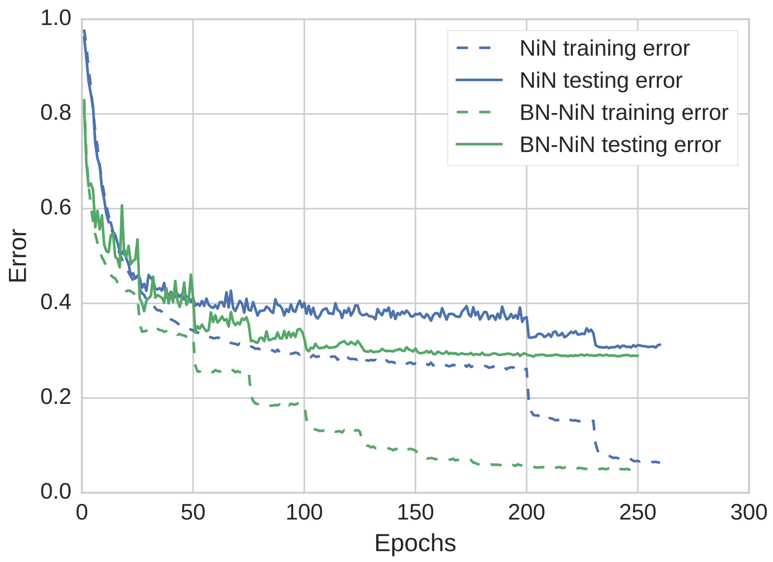


Figure 4.10: Training curves for the NiN and BN-NiN models on CIFAR-100+.

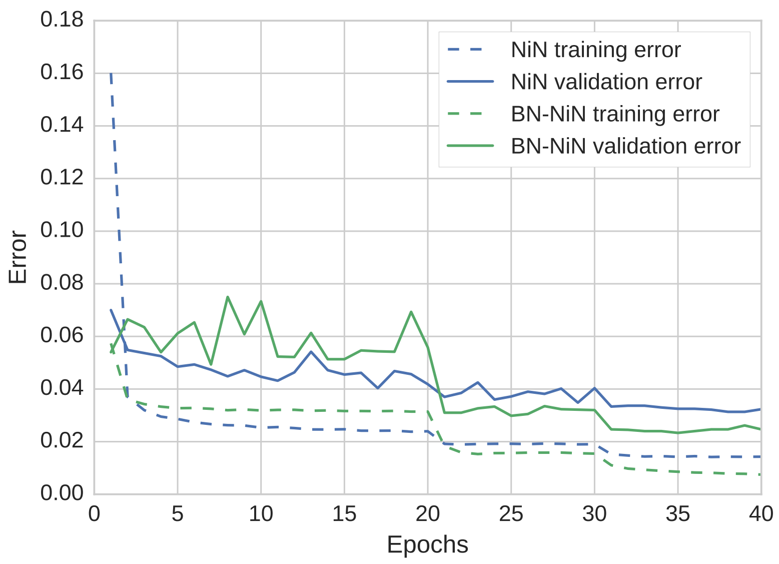


Figure 4.11: Training curves for the NiN and BN-NiN models on SVHN.

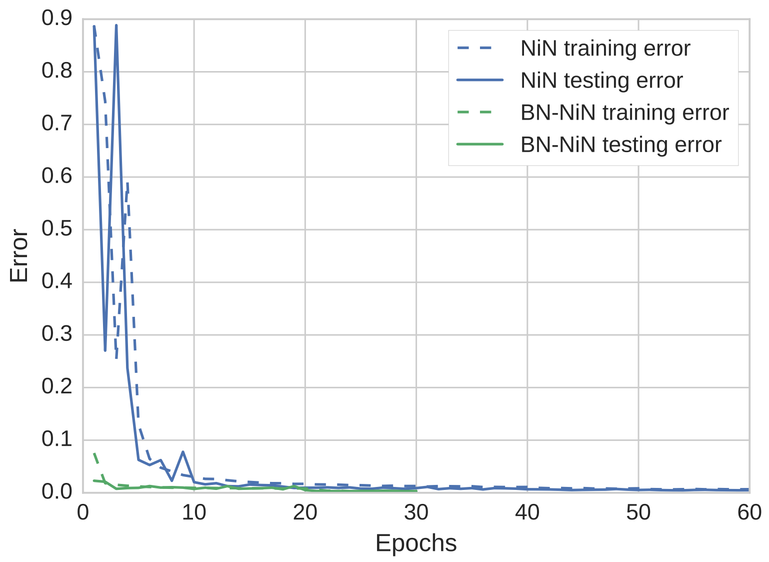


Figure 4.12: Training curves for the NiN and BN-NiN models on MNIST.

are 9 times more demanding. Besides, they are still able to perform a mapping that takes all channels in consideration. For these reasons, 1×1 convolutions can be used to reduce dimensionality, by reducing the number of feature maps, before more expensive convolutions are used. This is a sort of embedding, where high-dimensional sparse data is compressed into a low dimension representation [42].

By using 1×1 convolutions, the network can be designed to perform less computation without much loss in expressiveness (making it more efficient), and with a smaller number of parameters, which may act as a regularizer as well as ease training. These properties are exploited in the NiN module and have also been put to good use in other well-known architectures, such as GoogleNet [42, 65, 66] and ResNet [40, 67]. Both of them are winners of the ImageNet Large Scale Visual Recognition Competition (ILSVRC) [68, 69][5].

4.4 VGG-Style Network

VGG (Visual Geometry Group) networks [70] were popularized after the ILSVRC 2014. Despite not winning the classification challenge, they became popular due to their simplicity in design: its feature extractors are composed only by stacks of 3×3 convolutions, each followed by max pooling. Each stack has a fixed number of filters and after each pooling, the number of filters is doubled, keeping computation costs mostly constant. Even with this simple design, it was able to achieve very good results in large image classification.

The architecture explored next, which we call BN-VGG[6], utilizes the VGG design principles, combined with batch normalization, to small image classification. It is a network with 15 weight layers, 13 of them being convolutional layers and the last two being FC layers that generate the class scores. Like the batch normalized version of the NiN architecture, every ReLU is preceded by a batch normalization layer. The network is also regularized by dropout layers as well as weight decay.

We evaluate the network on the CIFAR datasets and use the same hyperparameters for both. The datasets are preprocessed the same way as in the NiN case and the learning rate schedule follows that of the BN-NiN. Detailed information regarding the architecture and hyperparameters is available in Table 4.4.

The BN-VGG network achieves 8.53% and 32.34% error rates for CIFAR-10 and CIFAR-100, respectively. When data augmentation is used, the network achieves 6.56% and 29.64% error rates for CIFAR-10+ and CIFAR-100+, respectively. Train-

[5]For more information about the ImageNet challenges visit http://www.image-net.org/challenges/LSVRC

[6]This network is based on the VGG-like network proposed by S. Zagoruyko (unpublished). Please see `torch.ch/blog/2015/07/30/cifar.html`. His implementation is available at: `github.com/szagoruyko/cifar.torch`.

Table 4.4: BN-VGG: architecture and hyperparameters.

Operation	Kernel size	Stride	Feature maps	Nonlinearity	Dropout
all	*convolutions use SAME padding*				
Network - Input $32 \times 32 \times 3$					
Conv + Dropout	3×3	1	64	BN-ReLU	0.3
Convolution	3×3	1	64	BN-ReLU	
Max Pooling	2×2	2			
Conv + Dropout	3×3	1	128	BN-ReLU	0.4
Convolution	3×3	1	128	BN-ReLU	
Max Pooling	2×2	2			
$2\times$ Conv + Dropout	3×3	1	256	BN-ReLU	0.4
Convolution	3×3	1	256	BN-ReLU	
Max Pooling	2×2	2			
$2\times$ Conv + Dropout	3×3	1	512	BN-ReLU	0.4
Convolution	3×3	1	512	BN-ReLU	
Max Pooling	2×2	2			
$2\times$ Conv + Dropout	3×3	1	512	BN-ReLU	0.4
Convolution	3×3	1	512	BN-ReLU	
Max Pooling	2×2	2			
Fully-Connected	512 *units*			BN-ReLU	0.5
Fully-Connected	10(100) *units*			softmax	
Conv + Dropout - F, P					
Convolution	3×3	1	F	BN-ReLU	
Dropout	*drop probability* $= P$				
Preprocessing	Per channel mean and variance normalization				
Optimizer	Nesterov momentum ($\eta = 0.1$, $\mu = 0.9$)				
Weight Decay	5×10^{-4}				
Batch size	64				
Epochs	250				
Learning rate schedule	Divide by 2 every 25 epochs				
Weight initialization	He initialization (Gaussian)				

ing curves are shown in Fig. 4.13 and Fig. 4.14. The good performance of the network indicates that small receptive field convolutions are an effective way to build a deep network.

The BN-VGG architecture has many more parameters than BN-NiN and yet does not overfit. This is because of the extensive use of regularizers. Regarding performance in CIFAR-10, BN-VGG is a bit more accurate than BN-NiN when no data augmentation is used. BN-VGG also seems to benefit more from data augmentation, probably because of the larger number of parameters. In CIFAR-100, BN-VGG performs worse than BN-NiN, which indicates that the large number of parameters is not as beneficial with a very restricted number of examples per class. This is also supported by that fact that, while still worse, performance on CIFAR-100+ is much closer to the results obtained with BN-NiN.

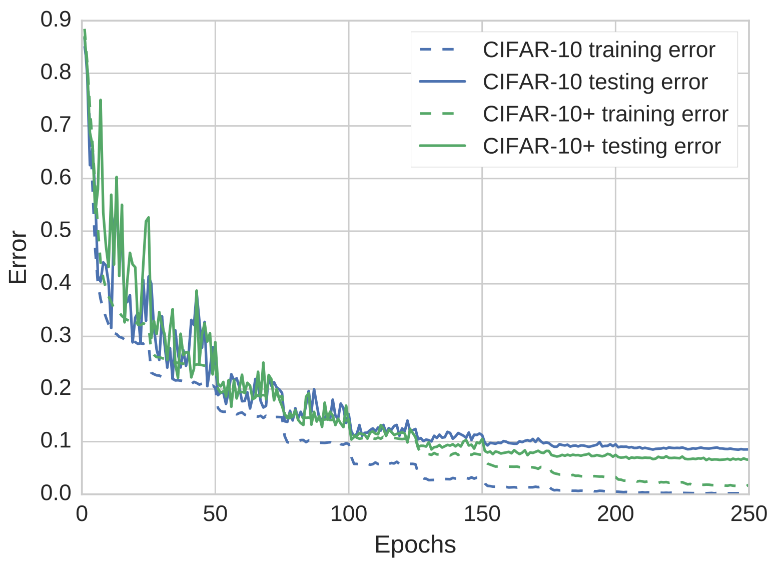


Figure 4.13: Training curves for BN-VGG on CIFAR-10 and CIFAR-10+.

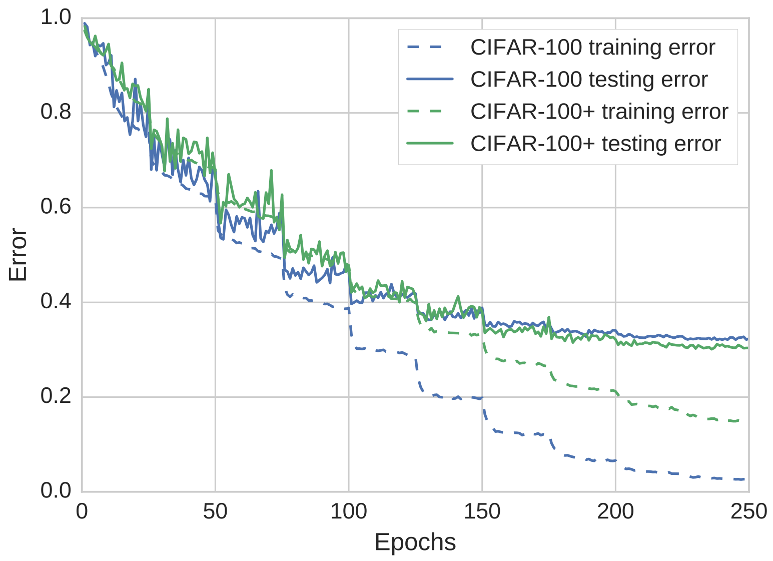


Figure 4.14: Training curves for BN-VGG on CIFAR-100 and CIFAR-100+.

4.5 Residual Learning: Extreme Depth with Identity Mappings

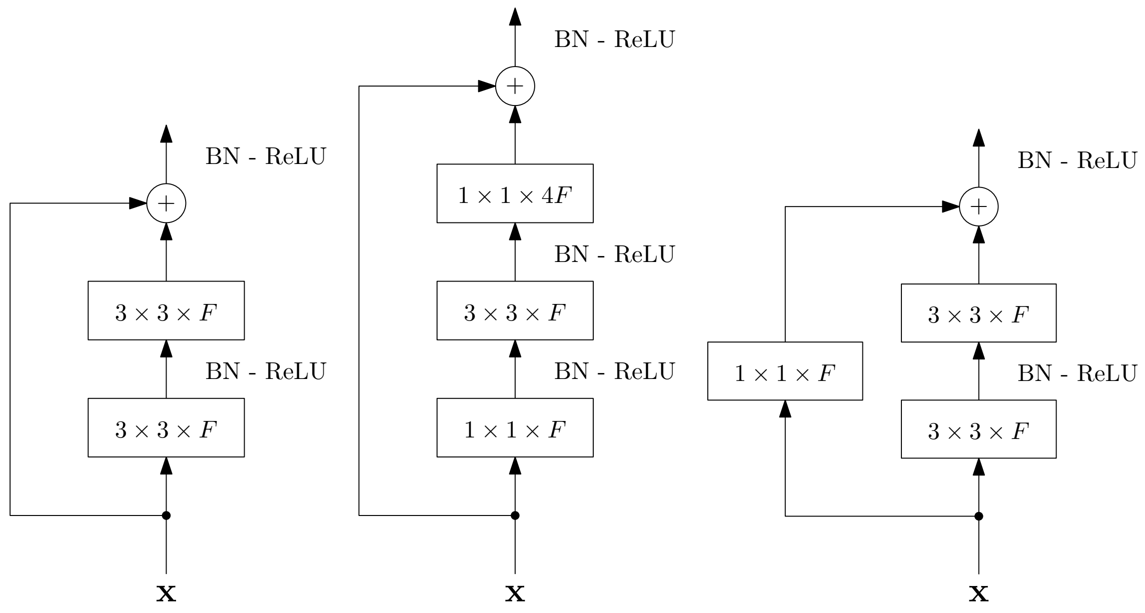
The success of increasingly deep networks seems to indicate that depth is a very important factor on the expressiveness of neural networks. Several advances in image recognition were obtained by simply training larger and deeper networks [10, 40, 42, 70]. Increasing the depth indiscriminately, however, usually leads to problems. More specifically, very deep networks start to exhibit worse performance than shallower counterparts. While deeper networks usually have more parameters, the lackluster performance does not seem to be caused by overfitting: very deep networks many times perform worse on the training set than shallower ones. Vanishing gradients also do not seem to be the whole reason. Even carefully initialized networks that also feature batch normalization suffer from this problem [40]: these networks fail to converge to good minima even when their gradients are seemingly healthy [40].

The aforementioned reasons indicate a fundamental difficulty in the optimization process that causes performance degradation. On the worst case, the extra layers could simply be set to compute the identity function and training performance would be the same as in shallower networks. The optimizers currently used seem to be unable to find such solutions. With this in mind, [40] proposes a reparametrization of the function computed by a layer in the form of a residual mapping. Instead of training a layer to compute a desired function $h(\mathbf{x})$, it is instead trained to compute the residual function $f(\mathbf{x}) = h(\mathbf{x}) - \mathbf{x}$. The original function is easily recovered as $h(\mathbf{x}) = \mathbf{x} + f(\mathbf{x})$. This can be interpreted as biasing the layers to implement functions that resemble the identity function: on the beginning of training, since weights are initialized randomly around zero, layers would start with functions close to the identity. The optimization process could then recover identity mapping by driving the layer weights toward zero or change them to implement a different function.

In the context of a graph, this formulation is realized with a shortcut connection from the input to the output, that is added to the result of the operation that implements the residual function. The block that includes both the residual function and shortcut connection is the building block of residual networks (ResNets) and is called residual unit or residual block. The residual block can be propagated through and trained just like any other layer in the network.

4.5.1 Residual Blocks on a ConvNet

In the case of a convnet, the residual function $f(\mathbf{x})$ may be realized by one or several convolutional layers (along with normalization layers and nonlinearities). The addition of the input and output feature maps is performed element-wise. We first



(a) Regular residual block. (b) Bottleneck residual block. (c) Shortcut mapping via 1×1 convolutions.

Figure 4.15: Diagrams of residual blocks. Connections with batch normalization and ReLU are denoted with the 'BN - ReLU' tag on their side.

consider a residual block with a residual function that is implemented by two 3×3 convolutional layers with F filters. The first convolution is followed by batch normalization and ReLU layers. The output of the second convolution is then added to the input and batch normalization and ReLU is applied on the sum, as shown in Fig. 4.15a. We call this the regular residual block. Note that the naive approach of applying the nonlinearity before the addition would cause the output of the residual function to be non-negative, which in turn makes the output of the blocks monotonically increasing and might affect the expressiveness of the network [67]. Because of the element-wise addition, the residual output should have the same shape as the input. In the case where output dimensions are changed, the shortcut performs a mapping with a 1×1 convolutional layer, as shown in Fig. 4.15c. Shortcuts that contain 1×1 convolutions are called projection shortcuts, because they project the input feature maps into the output feature map dimensions.

There is a second residual block considered that features three convolutional layers. Instead of two $3 \times 3 \times F$ convolutions, the block is composed of $1 \times 1 \times F$, $3 \times 3 \times F$ and $1 \times 1 \times 4F$ convolutional layers. On this case, the input is expected to have $4F$ channels. Because of how the 1×1 convolutions are used to compress the information before the most computationally intensive 3×3 convolution, this is called the bottleneck residual block, illustrated in Fig. 4.15b. The bottleneck and the regular residual blocks have similar computation requirements.

4.5.2 Pre-activation Residual Blocks

The residual blocks discussed do not strictly adhere to the residual formulation discussed because a nonlinearity is applied after the addition operation. These operations cause multiplicative modifications to the gradients as they travel backwards through the graph, which may degrade performance [67]. As discussed before, moving the nonlinearity inside the residual block is not advisable. Instead, [67] proposes an asymmetric scheme where the nonlinearity (along with the associated batch normalization) is applied only to the path towards the next residual function, leaving the shortcut of the next block unaffected. The process is illustrated in Fig. 4.16a.

The use of asymmetric activations is equivalent to pre-activating the convolutions, i.e., using the activation function *before* the convolutional layers instead of after. The dashed box in Fig. 4.16a delimits the equivalent pre-activation block. Figs. 4.16b and 4.16c illustrate the regular and bottleneck pre-activation residual blocks, respectively.

Using pre-activation blocks allows information to flow unimpeded through the shortcuts, which helps training deeper networks. Consider a residual network with N layers. The n-th layer performs a transformation $h_n(\mathbf{x}_n) = \mathbf{x}_n + \bar{f}(\mathbf{x})$, where $\bar{f}(\mathbf{x})$ represents the pre-activated block and $\mathbf{x}_n = h(\mathbf{x}_{n-1})$ is the input of layer n. The output of the n-th layer can be computed as

$$\mathbf{x}_{n+1} = \mathbf{x} + \sum_{i=1}^{n} \bar{f}(\mathbf{x}_i), \tag{4.1}$$

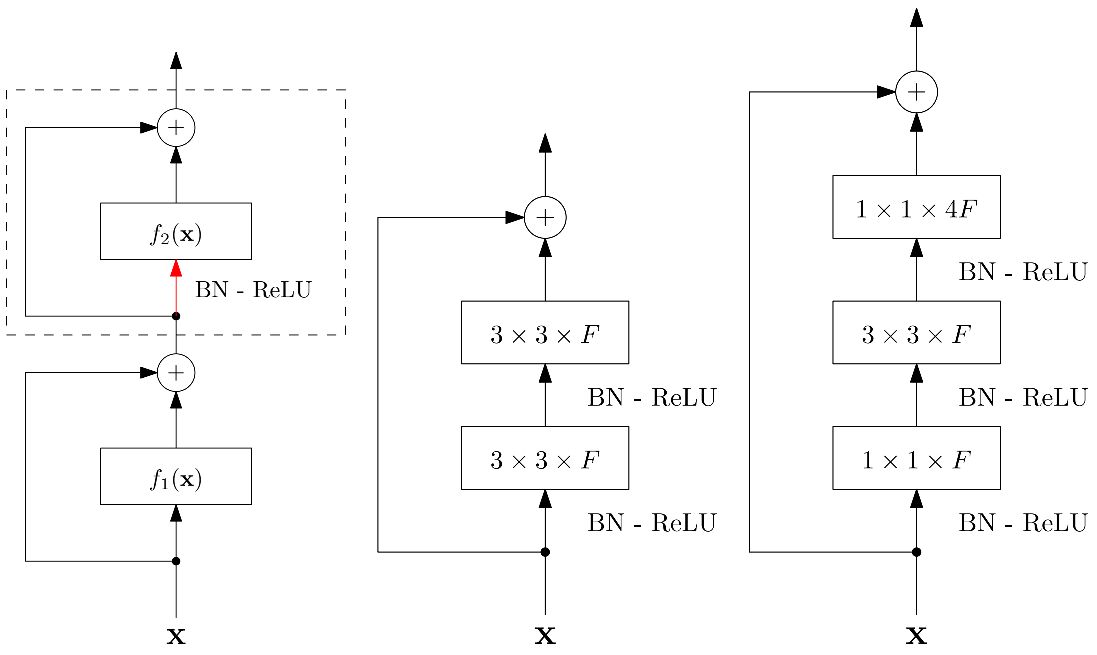
where $\mathbf{x} = \mathbf{x}_1$ is the input of the network. The output can be expressed as

$$\begin{aligned} \mathbf{o} = \mathbf{x}_{N+1} &= \mathbf{x}_1 + \sum_{i=1}^{N} \bar{f}(\mathbf{x}_i) \\ &= \mathbf{x}_n + \sum_{i=n}^{N} \bar{f}(\mathbf{x}_i). \end{aligned} \tag{4.2}$$

Conversely, the gradient of the loss l w.r.t the n-th input is

$$\begin{aligned} \frac{\partial l}{\mathbf{x}_n} &= \frac{\partial l}{\mathbf{x}_N}\frac{\partial \mathbf{x}_N}{\mathbf{x}_n} \\ &= \frac{\partial l}{\mathbf{x}_N}\left(1 + \frac{\partial l}{\partial \mathbf{x}_n}\sum_{i=n}^{N-1} \bar{f}(\mathbf{x}_i)\right). \end{aligned} \tag{4.3}$$

According to Eq.(4.3), the gradient can be decomposed into two additive terms. The first represents the gradient flow through the shortcuts and the second represents the gradient flows through the residual functions. We see that gradients flow through



(a) Asymmetric activation. (b) Regular pre-activation residual block. (c) Bottleneck pre-activation residual block.

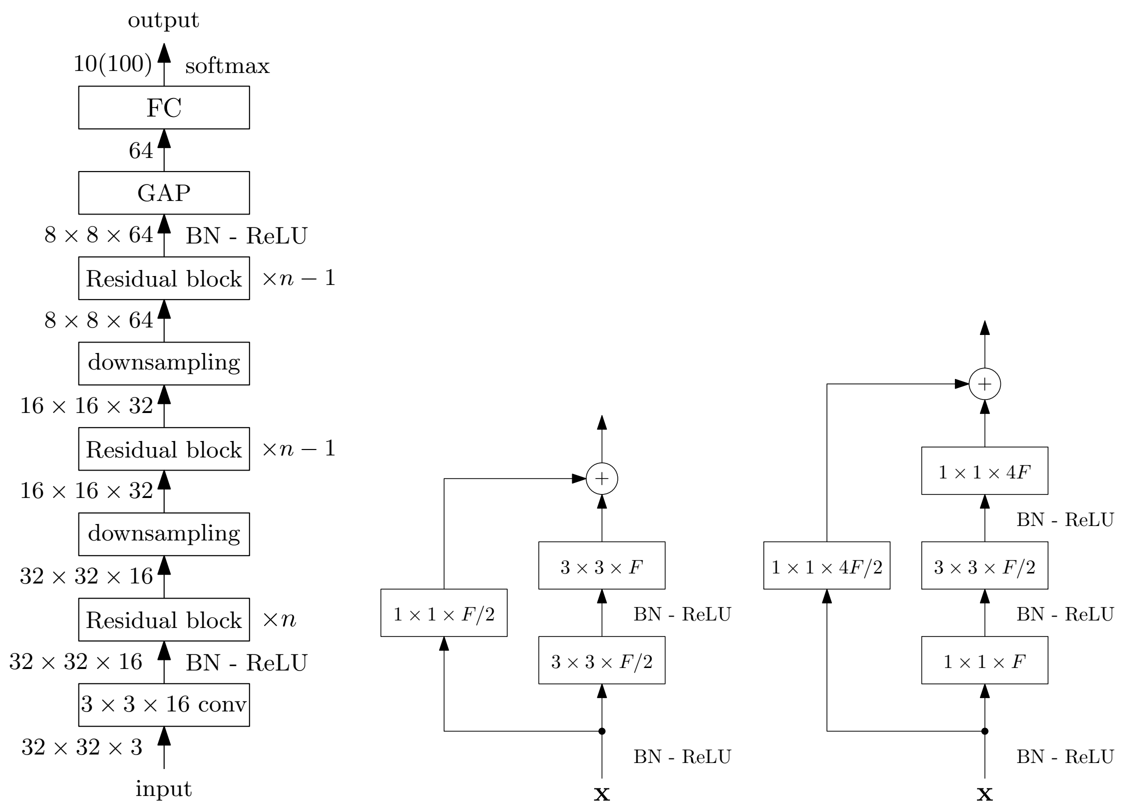
Figure 4.16: Diagrams of pre-activation residual blocks. The red path in Fig. 4.16a highlights the asymmetric path that contains the activation, and the dashed box delimits the pre-activation block.

the shortcuts without being modified and even if layer weights are very small, the gradient should not vanish. The only time gradient does not flow directly through the shortcut path is when a residual block contains a projection shorcut to map different input and output sizes. Pre-activated blocks that contain projection also feature a slight difference from other pre-activated blocks: since both paths contain convolutions, both are pre-activated, i.e, there is no asymmetry.

Architectures using post-activation residual blocks were successfully trained with depths of hundreds of layers, which was not possible with plain networks [40]. If pre-activation blocks are used, networks with more than a thousand layers can be trained [67].

4.5.3 Residual Architectures for CIFAR datasets

In this section, we evaluate the performance of deep residual networks on the CIFAR datasets. All networks evaluated are designed as in [40, 67]: the first layer is a 3×3 convolutional layer with 16 filters, then 3 stacks of n residual blocks are used, with 16, 32 and 64 filters, respectively. The first residual blocks of the second and third stacks perform subsampling by means of a 2-strided convolution. Therefore, the stacks operate on feature maps with sizes $32 \times 32, 16 \times 16$ and 8×8, respectively.



(a) General architecture for CIFAR datasets. (b) Regular downsampling block. (c) Bottleneck downsampling block.

Figure 4.17: General architecture of the networks and downsampling blocks. Values to the left of the arrows show the feature map sizes. Convolution strides are indicated after a foward slash.

This is a VGG-style design that doubles the number of filters after subsampling the feature maps. After the residual stacks, global average pooling is applied and a fully-connected layer with softmax activation is used to obtain class probabilities. The overall architecture is shown in Fig. 4.17a. Downsampling blocks are shown in Figs. 4.17b and 4.17c. All downsampling is handled using strided convolutions (except for the global average pooling), like in [39].

We consider both regular and bottleneck residual networks. All networks use pre-activation. The total depth of a regular network is $N = 6n + 2$: the first convolutional layer, 3 stacks of n residual blocks containing 2 convolutions each, and the FC output layer. For bottleneck networks, the depth is $N = 9n + 2$, since bottleneck blocks contain 3 convolutions. The networks were tested for several values of n and each network is denoted as ResNet-N, e.g., the network with $n = 3$ regular residual blocks per stack is denoted ResNet-20.

All networks were trained on CIFAR-10+ and used the same hyperparameters. They were trained with Nesterov momentum ($\mu = 0.9$) for 164 epochs, with an initial learning rate of 0.1, that was divided by 10 after 50% (82) and again after 75% (123)

Table 4.5: Error rates of residual networks trained on CIFAR-10+.

Network	**# parameters**	**Error (%)**	**Residual Block**
ResNet-20 ($n = 3$)	272K	7.68	Regular
ResNet-32 ($n = 5$)	467K	6.94	Regular
ResNet-44 ($n = 7$)	661K	6.60	Regular
ResNet-56 ($n = 9$)	856K	6.46	Regular
ResNet-65 ($n = 7$)	683K	6.29	Bottleneck
ResNet-83 ($n = 9$)	868K	5.86	Bottleneck
ResNet-110 ($n = 18$)	1.7M	5.97	Regular
ResNet-164 ($n = 18$)	1.7M	5.47	Bottleneck

of epochs were completed. All networks were regularized with weight decay of 10^{-4}. Table 4.5 presents the error rate of increasingly deep networks. Training curves are shown in Fig. 4.18. We see that even very deep networks, with over 100 layers, have no optimization difficulty and accuracy increases steadily with depth. Regarding residual block type, the use of bottleneck blocks allows for deeper networks that are more accurate while keeping parameter count low.

4.5.4 Wide Residual Networks

The residual architectures discussed so far have showcased the representation power of very deep networks: even thin networks, with few filters per convolutional layers, are capable of achieving good performance. Using few filters per layer also keeps the parameter count low despite the increased depth and allows the networks to generalize well even without regularization techniques such as dropout. Using deep thin networks has one disadvantage however: since layers depend on the output of previous ones, the computation must be done sequentially for each layer. This limits the amount of parallel computation that can be performed, which is particularly detrimental because this is the kind of computation GPUs excel at. In fact, very deep networks may be slower to train compared to shallower ones even if they contain fewer parameters.

Motivated by this, [71] proposes wide residual networks. These networks follow essentially the same previously discussed architecture with an additional hyperparameter: the widening factor. A network with widening factor k has k times more filters in each layer. Wide residual networks are interesting because they allow for better use of parallel computation. We compare these networks to the thin ($k = 1$) residual networks presented on the CIFAR-10+ dataset. A wide residual network with depth N and widening factor k is denoted as WRN-(N, k). These networks were trained for a total of 200 epochs. They were optimized with Nesterov momentum using an initial learning rate of 0.1, which is divided by 5 after 60, 120, and 160 epochs; constant amount of momentum $\mu = 0.9$; and mini-batches

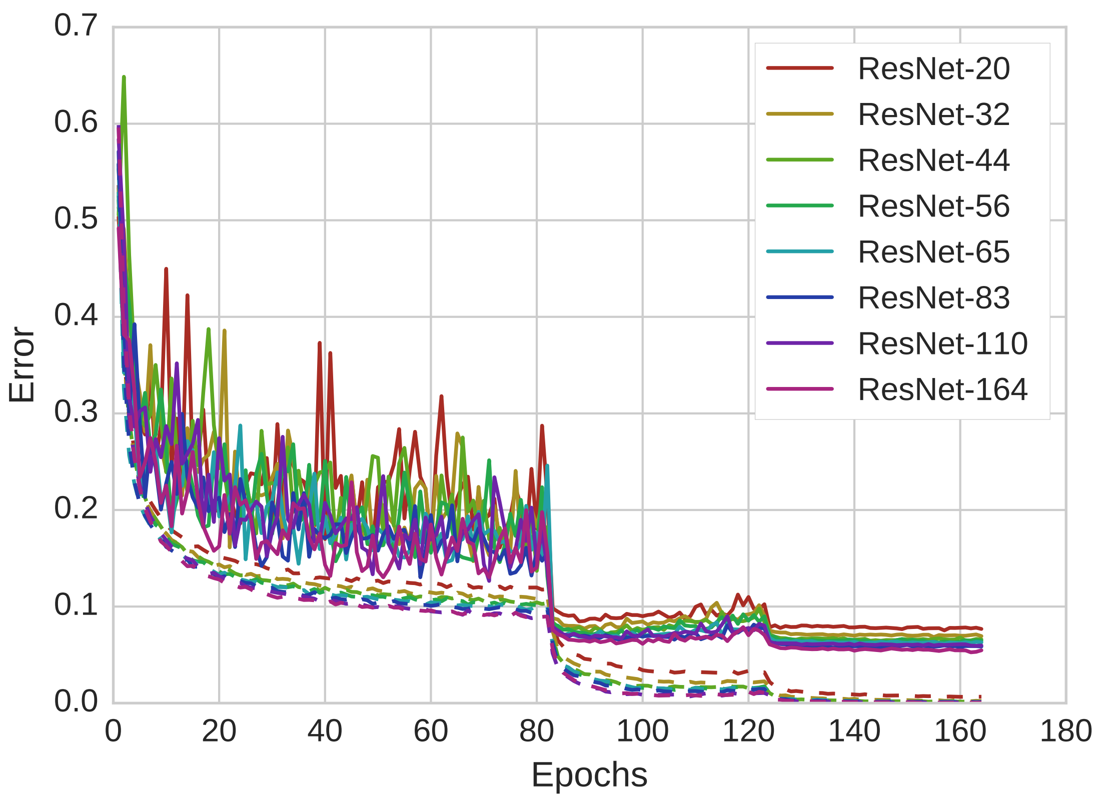


Figure 4.18: Training curves for ResNets on CIFAR-10+.

of 64 samples. The network is regularized with weight decay of 5×10^{-4}. Two wide ResNets (both using regular residual blocks) were evaluated: WRN-$(40, 4)$ and WRN-$(28, 10)$. They obtain 4.87% and 4.57% error rates, respectively. Both results are better than ResNet-164 and also train faster, despite having many more parameters. The training curves are shown in Fig. 4.19.

These results indicate that residual architectures are not only useful for building extremely deep networks: they are also successful when building wider, not as deep networks. It is also evidence that depth is not the only way to effectively improve expressiveness: these networks have good generalization and do not overfit, despite being much wider and having more parameters. These results also agree with conjectures that say that, as long as a network is deep enough, it should be able to efficiently learn from data [8]. Note that while these networks are shallower than the thin ResNets studied before, they are still quite deep: much deeper than architectures studied previously such as NiN and BN-VGG. By building deep, but not overly deep, wide nets, it is possible to achieve a good compromise between depth and width that makes the network expressive while still keeping it computationally efficient.

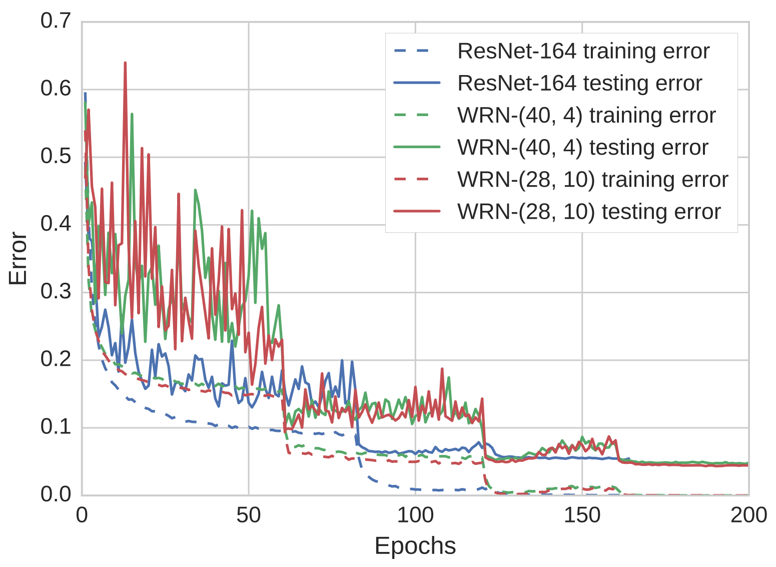


Figure 4.19: Training curves for wide ResNets on CIFAR-10+.

4.5.5 Residual Networks Family

Residual architectures became extremely popular after their impressive results in the ILSVRC2015. The principle of residual networks is also very simple: simply by using shortcut connections, the optimization of deep networks is improved. Several researchers began experimenting with ResNet variants and a whole family of them exists today. This family includes: multi-residual networks [72], which use residual units with several branches; Inception-ResNet [66], a residual version of the Inception architecture; networks with stochastic depth, which are ResNets that are trained using a method similar to dropout that drops entire layers [73]; and others [74–76]. There are also works that seek a better understanding of how residual networks represent functions [77, 78].

4.6 Effectively Reusing Features with Densely Connected Convolutional Networks

After the big success of residual networks, it became clear that shortcut connections were a good alternative when building very deep neural networks. Densely Connected Convolutional Networks (DenseNets) [64] also use shortcut connections, but with a very different objective. While residual networks use shortcuts for ease of optimization and improved gradient flow, DenseNets main design consideration is feature reuse.

A DenseNet is composed mostly of dense blocks. Inside a dense block, every layer is connected to all its predecessors. This dense connectivity pattern, in contrast with

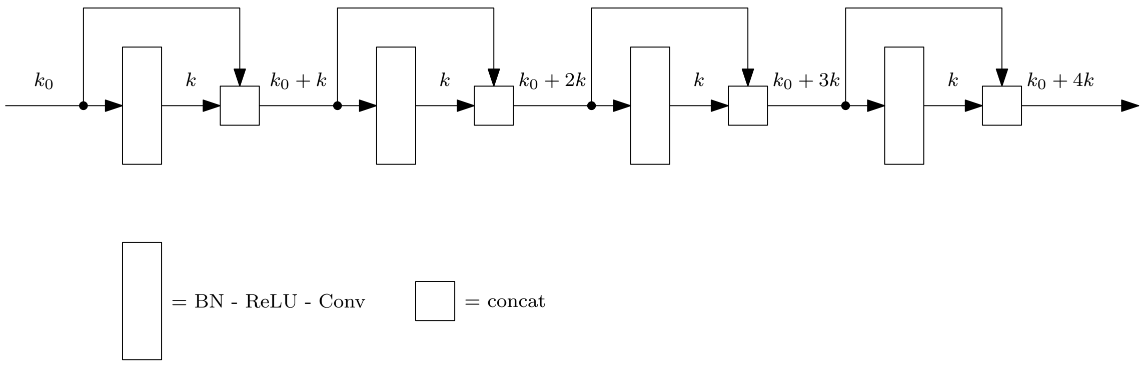


Figure 4.20: Diagram of a dense block with growth rate k and $n = 4$ layers. Labels on top of paths indicate the number of channels.

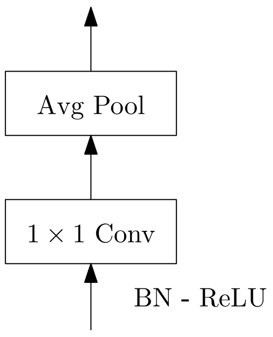


Figure 4.21: Diagram of a transition block.

the usual one where each layer is connected only to the previous one, is the reason for the name of this architecture. A dense block can be implemented through the use of skip connections that, instead of performing addition like in ResNets, concatenate input features with output features. It is important to note that, since the outputs of the layers are concatenated, the inputs of each layer get larger the deeper the layer is inside the dense block. If a dense block input has k_0 feature maps and each layer has k output feature maps, the i-th layer will have $k_0 + (i - 1)k$ feature maps as input. For this reason, the number of output feature maps in each layer is called the growth rate of the dense block. The number of layers inside a dense block is denoted n. An illustration of a dense block is provided in Fig.4.20.

Because of how a dense block concatenates feature maps, subsampling techniques cannot be applied inside of it. Instead, a DenseNet is divided in several dense blocks of different spatial resolutions. Between such blocks, a transition block performs subsampling operations. The transition block considered here is composed of a pre-activated 1×1 convolutional layer and a standard average pooling layer. The transition block is illustrated in Fig.4.21. Generally, the input and the output of a transition block have the same number of feature maps. Only spatial dimensions are changed.

The structure of DenseNets have some desirable properties, among which the most prominent is feature reuse. Because of the connectivity pattern, a layer has direct access to the output of all previous layers and the block input. Feature reuse also results in parameter efficiency: since layers do not have to recompute useful features, they can instead compute fewer feature maps. While regular architecture may have hundreds of output feature maps, typical growth rates are in the order of tens. Each layer makes a small contribution to the "collective knowledge" of the network, which is made available to all layers after it. In regards to gradient flow, the reuse of the output of a layer in several instances of the network results in gradient flowing back from many paths. These paths can be interpreted as a form of deep supervision [63, 64], akin to the effects of auxiliary classifiers present in deeply-supervised networks [63] and Inception architectures [65]. In regards to the transition block, it is responsible for capturing information and summarizing it spatially. By using a 1×1 convolutional layer before the pooling operation, it is able to transform the feature maps into representations that are more appropriate in lower spatial resolution.

4.6.1 DenseNet Classification

We test the performance of DenseNet architectures on the CIFAR, and SVHN datasets. The architecture used for all three datasets is described next. The input is first transformed by a 3×3 convolutional layer with 16 filters. The output of this layer is fed to three dense blocks of decreasing spatial resolution (32×32, 16×16 and 8×8). Between each dense block, there is a transition layer that halves the height and width of the feature maps and keeps the number of them the same. The number of layers n and the growth k are the same for all three dense blocks. After the last dense block, global average pooling is applied and the output is mapped to class probabilities by an FC layer with softmax activation. Therefore, the total depth of the network is $N = 3n + 4$: 1 for the first convolution, $3n$ for the dense blocks, 2 for the transition blocks and 1 for the FC layer. A network with total depth N and growth size k is denoted as DenseNet-(N, k).

All networks are trained with Nesterov momentum ($\mu = 0.9$) with initial learning rate 0.1. The learning rate is divided by 10 after 50% and again after 75% of the epochs have been completed. Networks were trained for 300 epochs in the CIFAR datasets and 40 epochs in the SVHN dataset. All networks are regularized with weight decay of 10^{-4}. For datasets without augmentation (CIFAR-10, CIFAR-100, SVHN), all convolutions are regularized with dropout with probability of 20%. For data augmented datasets (CIFAR-10+, CIFAR-100+) there is no dropout.

We evaluate DenseNet-$(40, 12)$, a relatively small network with only about 1M parameters. This network achieves 6.53%, 5.53%, 27.25% and 25.83% error rates on the CIFAR-10, CIFAR-10+, CIFAR-100, and CIFAR-100+ datasets, respectively. When trained on the SVHN dataset, it achieves 1.86% error rate. Training curves are shown in Figs. 4.22, 4.23, and 4.24.

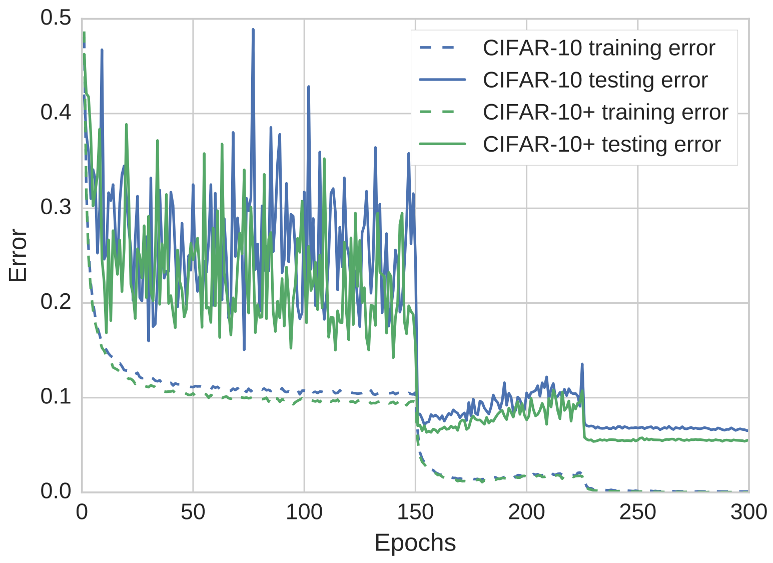


Figure 4.22: Training curves for DenseNet-(40, 12) on CIFAR-10 and CIFAR-10+.

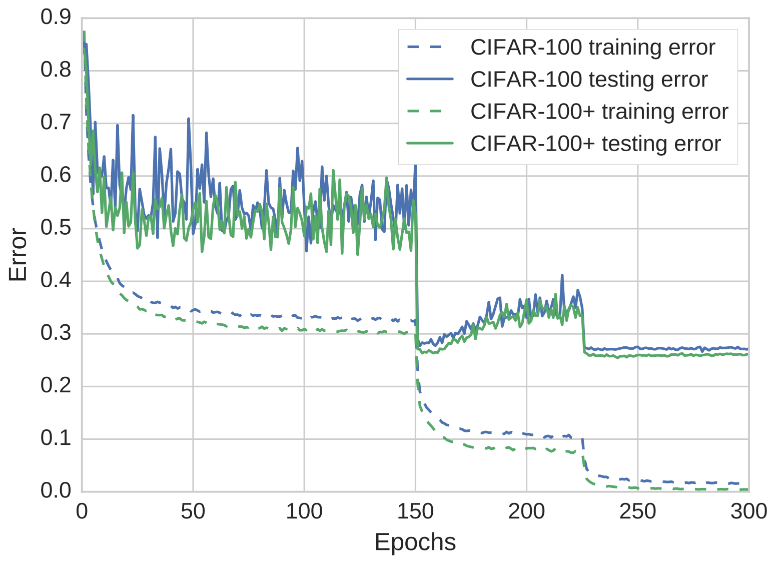


Figure 4.23: Training curves for DenseNet-(40, 12) on CIFAR-100 and CIFAR-100+.

DenseNets are more accurate than ResNets with similar number of parameters [64], and large DenseNets achieve state-of-the-art results on small image benchmark datasets. DenseNets have also been applied to large-scale classification and yield performance comparable to ResNets with significantly fewer parameters [64]. A version of DenseNet was also used for semantic segmentation, obtaining state-of-the-art results on benchmark datasets [79].

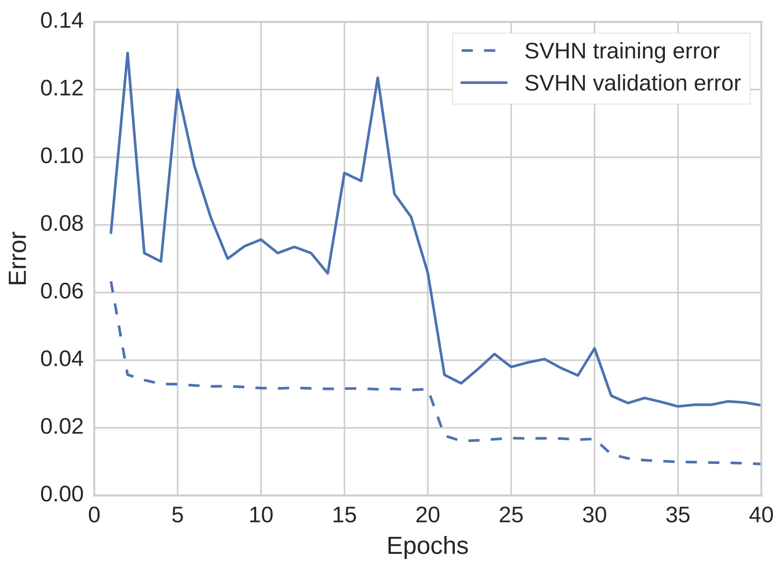


Figure 4.24: Training curves for DenseNet-$(40, 12)$ on SVHN.

4.6.2 DenseNet Variants

While DenseNets are very efficient, there are some modifications that improve them even further. This section aims to analyze some characteristics of DenseNets and how they motivate the design of variants that exploit them. Because of its design, a dense block allows for layers to have access to all feature maps from previous layers. In order to evaluate how the layers utilize this connectivity, we compute the average squared weight of each input to a layer. Consider the j-th layer of the network: it has $C_{\mathrm{i}}^{(j)}$ feature maps as inputs and outputs $C_{\mathrm{o}}^{(j)}$ feature maps. For filters with sizes $H^{(j)}$ and $W^{(j)}$, the set of filters is a tensor $\mathbf{F}^{(j)} \in \mathbb{R}^{C_{\mathrm{o}}^{(j)} \times C_{\mathrm{i}}^{(j)} \times H^{(j)} \times W^{(j)}}$. In order to quantify the importance of each input feature map to the layer, we compute

$$m_{i,j} = \begin{cases} \frac{1}{C_{\mathrm{o}}^{(j)} H^{(j)} W^{(j)}} \sum_{o,h,w} (f_{o,i,h,w}^{(j)})^2, & i < C_{\mathrm{i}}^{(j)} \\ 0, & i \geq C_{\mathrm{i}}^{(j)} \end{cases} . \tag{4.4}$$

The matrix $\mathbf{M}$ should then indicate how important feature map i is to layer j: high values indicate that weights that connect outputs feature maps to input map i are high, which means the feature map is important; while low values indicate that these weights are low and, therefore, the input map is relatively unimportant. This matrix is plotted in the form of a heat map, shown in Fig. 4.25.

The heat map in Fig. 4.25 indicates that layers have fairly distributed weights, i.e., they tend to use the features obtained from several different layers, including early ones. In fact, experiments show that if the input feature maps are restricted to only those of recent layers, then performance is impaired [64][7]. This indicates

[7]This experiment, along with some similar analysis with respect to input feature map impor-

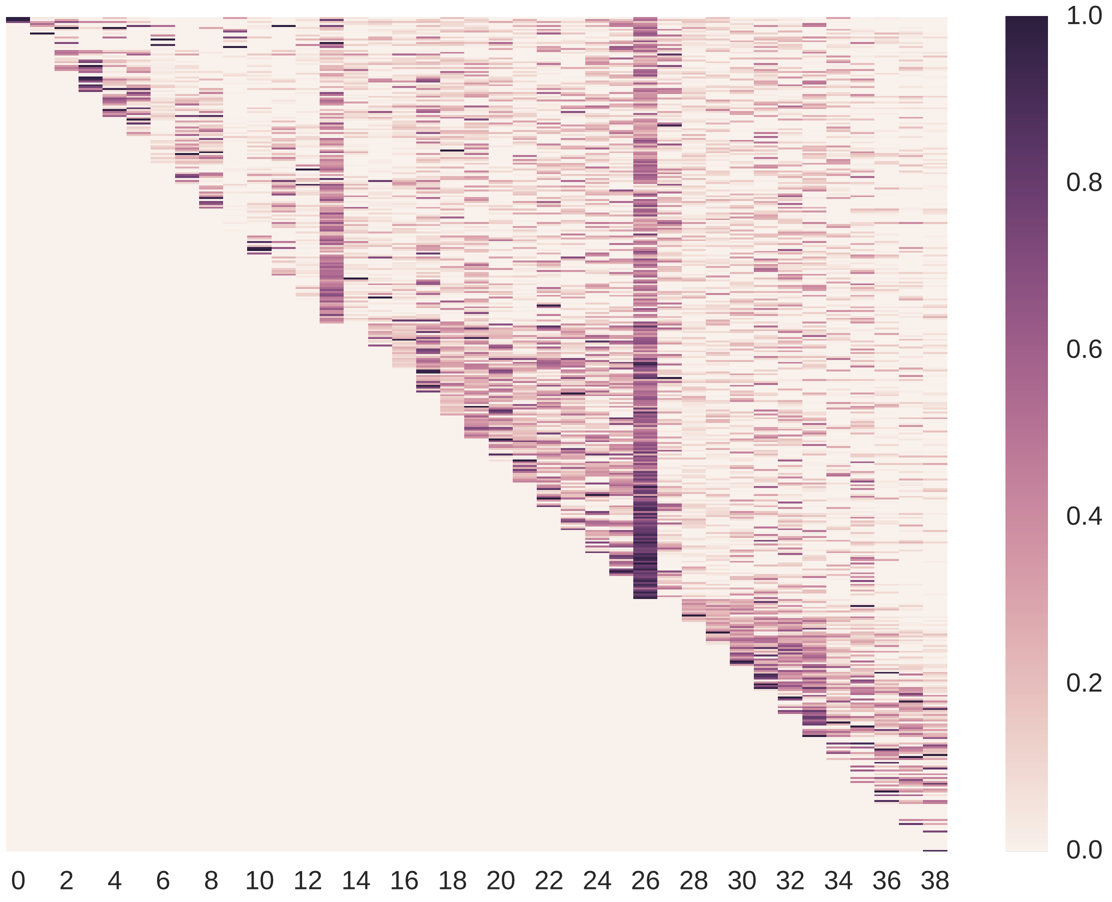


Figure 4.25: Importance of input feature maps for DenseNet-$(40, 12)$ trained on CIFAR-10+. Each column is normalized so that its maximum value is 1 for improved visualization.

that the dense connectivity pattern in DenseNets is, in fact, very important.

A more careful analysis, however, leads to the observation that, while each layer has distributed weights, this distribution is relatively sparse: a specific layer only takes a limited number of layers into account when computing its output. The previous attempt to limit the number of input feature maps failed because it was based on the assumption that recent feature maps are the most important, which is not true. Instead, it is would be interesting to select the most important feature maps for each individual layer. This can be efficiently implemented with the use of *bottleneck layers*: a 1×1 convolutional layer is used to map the input feature maps into a fixed number of feature maps that will serve as input to the original 3×3 convolutional layer. This allows each layer to learn how to select and/or combine relevant input feature maps before computing its output. We follow [64] and use $4k$ (k denoting the growth rate) bottleneck feature maps. A DenseNet that uses

tance, is available in the first version of the paper. These discussions were later removed from the paper due to space constraints. The interested reader should read previous versions of the paper (also available on ArXiv) for more information.

bottleneck layers is denoted DenseNet-B.

Another aspect of the connectivity is that layers on the second and third dense blocks do not seem to utilize layers from the transition blocks as much as the features from layers inside the dense block. The fact that most layers use only a few of the transition layer's feature maps could mean that features computed by the transition layers are redundant. This is motivation for the second modification: *compression*. In the original design, transition layers were used only for spatial downsampling and kept the number of feature maps the same. A DenseNet that uses compression, denoted DenseNet-C, has transition layers that, for an input with C_i feature maps, output $C_\text{o} = \theta C_\text{i}$ feature maps. The scalar $\theta < 1$, called the compression factor, controls how much the feature maps are compressed. We follow [64] and use $\theta = 0.5$.

In order to evaluate these variations on the DenseNet architecture, a DenseNet-BC (that uses both bottleneck and compression) was trained on the CIFAR datasets, both with and without data augmentation. The total depth of the network is now $N = 6n + 4$, since bottleneck layers contain two convolutional layers. The model tested is a DenseNet-BC-$(100, 12)$, i.e., a network with $n = 16$ bottleneck layers per dense block and growth rate $k = 12$. Despite being deeper than the vanilla DenseNet, it has fewer parameters (0.8M, compared to the original 1M). This network was trained with the same hyperparameters as the original DenseNet, and obtained $5.91\%, 4.58\%, 24.96\%$, and 22.29% error rates on the CIFAR-10, CIFAR-10+, CIFAR-100, and CIFAR-100+ datasets, respectively. The training curves are shown in Figs. 4.26 and 4.27. All these results are improved when compared to the vanilla DenseNet. Larger DenseNets also follow the same trend, with BC variants performing consistently better than vanilla counterparts. All in all, these modifications are capable of further improving an already very efficient architecture.

4.7 Architectures Recap

In this chapter, we discussed several architectures for image classification and evaluated them on benchmark datasets. The results are summarized in Table 4.6. Due to hyperparameter choice, framework differences, and others, some results differ from those claimed in the original implementations; these results are, however, consistent enough for the discussion presented in this work.

The first architecture presented was a fairly simple convnet containing two convolutional layers that were used as feature extractors for the classifier, implemented using FC layers. It served as an introduction to the general topology of convnets for classification, as well as a good platform for testing some concepts discussed in previous chapters, such as dropout and learning rate scheduling. It was, however, fairly simple and relied too much on FC layers, which increased the parameter count:

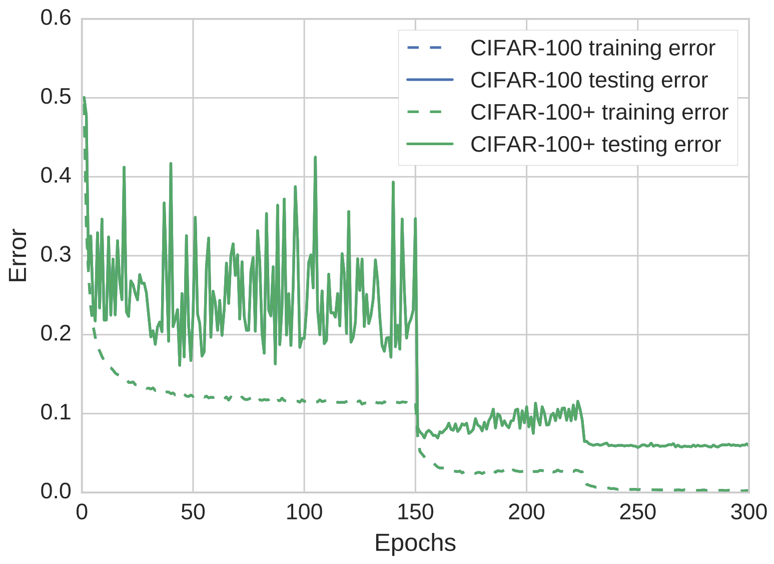


Figure 4.26: Training curves for DenseNet-BC-(100, 12) on CIFAR-10 and CIFAR-10+.

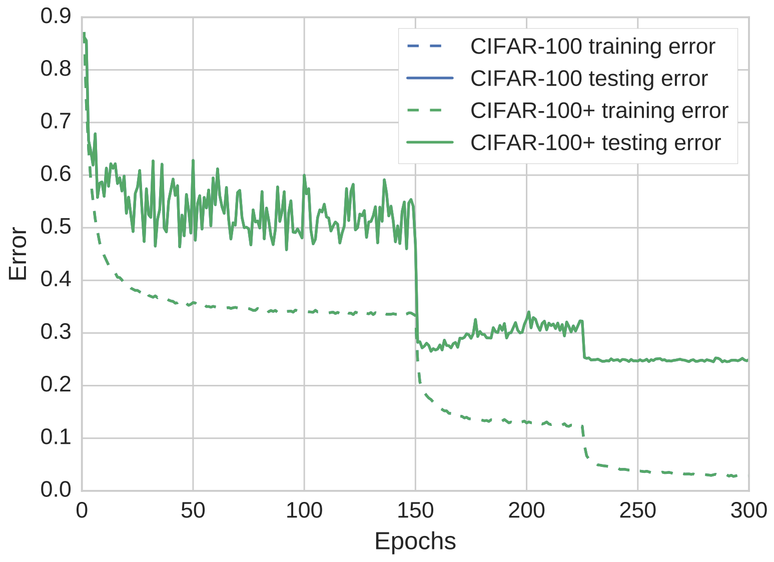


Figure 4.27: Training curves for DenseNet-BC-(100, 12) on CIFAR-100 and CIFAR-100+.

this network had more than 680 thousand parameters despite being quite shallow. The network in network architecture was described next. It is a very interesting architecture because it highlights the power and efficiency of 1×1 convolutions, which are used extensively to great success in several other architectures. It also introduces global average pooling, a technique that is very useful as a regularizer and reduces the number of parameters in the output FC layer; GAP is also widely used in other architectures. The batch normalized version, BN-NiN, also served as a nice example of the usefulness of batch normalization as a technique to improve the optimization process as well as generalization. The BN-VGG architecture was then discussed. It showcased how a simple design process can be used to good effect to build deep networks. It is also an example of how a large network (with 15 million parameters) can generalize well with the regularization techniques discussed: the use of weight decay, batch normalization, and a lot of dropout allows the model to generalize well even in datasets with limited data.

We then explored architectures that use shortcut connections. Residual networks attack the optimization problem that arises when training very deep networks. Residual learning is interesting not only because it enables training networks that are hundreds of layers deep, but also because it speeds up the optimization process. These very deep residual networks are also very efficient: ResNet-56 has 17 times fewer parameters than BN-VGG and obtains better performance. Even when compared to NiN architectures, which are more efficient than BN-VGG, ResNet-32 outperforms BN-NiN with 3.4 times fewer parameters. An alternative approach is then discussed in the form of wide residual networks. These networks reduce depth in favor of increased width. While these networks have many more parameters, and, as such, are not as efficient, they can outperform very deep thin ResNets. Wide ResNets are also attractive in the sense that they allow for more parallel computation. Finally, we investigate DenseNets. These networks use shortcut connections in order to improve feature reuse. They are even more efficient than ResNets, and the reduced parameter count also helps with generalization. The DenseNet variants, with bottleneck layers and compression, are even more efficient, outperforming vanilla DenseNets and, consequently, all the other architectures discussed previously.

Table 4.6: Performance on benchmark datasets. Values reported are percentage error rates.

Network	**Depth**	**# parameters**	**MNIST**	**C10**	**C10+**	**C100**	**C100+**	**SVHN**
Baseline	4	682K	0.6					
NiN	9	358K	0.49					
		1.6M		10.34	8.36	35.13	31.29	
		2M						2.39
BN-NiN	9	358K	0.33					
		1.6M		8.85	7.14	30.66	28.96	
		2M						1.78
BN-VGG	15	15M		8.53	6.56	32.34	29.64	
ResNet	20	272K			7.68			
	32	467K			6.94			
	44	661K			6.6			
	56	856K			6.46			
	65	683K			6.29			
	83	868K			5.86			
	110	1.7M			5.97			
	164	1.7M			5.47			
Wide ResNet								
k = 4	38	8.9M			4.87			
k = 10	26	36.5M			4.57			
DenseNet	40	1M		6.53	5.53	27.25	25.83	1.86
DenseNet-BC	100	768K		5.91	4.58	24.96	22.29	

Chapter 5

Beyond Classification: ConvNets as Feature Extractors

The most important aspect of a convnet is its feature extractor, that is characterized by convolutional and pooling layers. If the network is used for classification, a simple classifier on top of such features is used to obtain class predictions. The unparalleled results in such tasks are evidence that these features can be learned to be very discriminant [10, 40, 42, 70]. Other applications may require features with different characteristics. This chapter discusses:

1. the transferability of features learned by convolutional networks;
2. convnets as off-the-shelf feature extractors;
3. alternative computer vision tasks at which convnets excel;
4. class activation maps: a technique that uses features learned by a convnet for weakly supervised localization and visualization of network predictions;
5. neural style transfer: the use of convolutional neural networks for rendering images in the style of another. Iterative method, single-style networks, and multi-style networks.

Much work has been done to evaluate what features these networks learn and how generic they are. There are several techniques that seek to understand how networks interpret images by visualizing their features [80–82]. Related work reconstructs images based on their feature description [83].

There are studies that evaluate the transferability of features by using features learned in one dataset on another, which is a technique usually called *transfer learning*. Some of these works show, through several experiments, that off-the-shelf convnet features are very powerful and can obtain state-of-the-art performance even when compared to task-specific handcrafted features [84, 85]. The authors of [86] demonstrate how features get progressively more specialized the deeper the layer they are extracted from: shallow features are very generic pattern detectors, while deep features are more descriptive and specialized to the particular task the network was trained on. This indicates that the depth at which features are extracted depends on the similarity between the task the network was trained on and the task the features will be transferred to. The work of [87] demonstrates how one could take advantage of transfer learning on data-restricted settings: a convnet can be trained on a large labeled dataset (such as ImageNet [68]) and later fine-tuned to a smaller dataset with state-of-the-art results. This fine-tuning technique was also applied in object detection [13].

CNNs can also be trained to perform multiple tasks, such as classification and localization. These tasks are performed with shared features [13, 88]. This makes CNNs particularly useful for object detection [13–18, 88], with some techniques being capable of performing detection in real time [18].

CNN feature extraction has also been applied in image transformation tasks, where an input image is transformed into an output image. This is usually performed by encoding the input image into more abstract features that can then be mapped to the corresponding output image. Image transformation can be done with fully-convolutional neural networks. These networks have the advantage of being able to process images of any size. Examples of image transformation tasks include: semantic segmentation [19, 79], super-resolution [21, 22], colorization [89–91], and style transfer [22, 92].

5.1 Understanding Predictions with Class Activation Maps

In the last chapter, several architectures for image classification were presented. This section presents a technique that uses the features computed by the networks to visualize how the scores for each class are distributed spatially. There is a variety of work that seeks to understand the predictions of neural networks [81, 93–96]: [81] uses saliency maps computed with backprop to determine the importance of input pixels for the predicted class; [93] discusses how convnets trained on scene classification learn object detectors, despite not being trained to detect specific objects; [94] studies how convnets are able to localize objects despite being trained only with class annotations. These studies are useful for several reasons: they help elucidate how convnets make predictions, which may be used to guide network design as well as to explain mistakes; and they show the potential of convnets for weakly supervised localization. This type of learning is termed weakly supervised because there is no specific annotation for localization bounding boxes. The network learns to localize objects when trained only with class annotations.

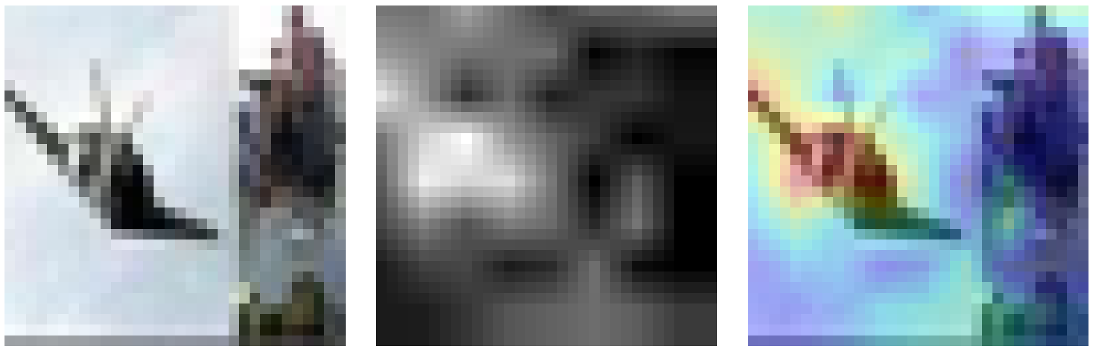
We focus our discussion on *class activation maps* (CAM) [95]. This technique is able to generate class-specific maps that indicate discriminative regions in the input image. The original work focuses on generating maps for networks that are trained on large images and how these maps can be used for understanding predictions and localizing class objects; we apply these maps to the networks discussed in Chapter 4. Experiments show how CAMs may be used for weakly supervised localization as well as to better understand the "thought process" of these networks.

Recall the NiN architecture presented in Section 4.3: it was composed entirely of convolutional and pooling layers, i.e., there were no FC layers used to compute class predictions. Instead, the output features maps $\mathbf{f}_i \in \mathbb{R}^{H \times W}$ were averaged with global average pooling and these values were used as class scores c_i:

$$c_i = \frac{1}{HW} \sum_{h,w} f_{i,h,w}. \tag{5.1}$$

This essentially means that the output features in each feature map correspond to how strongly each input region contributes to the respective class. Because of the pooling operations along the network, these feature maps are smaller than the input image. Class activation maps can be easily obtained by upsamping the feature maps back to the input size. These CAMs should roughly correspond to heat maps that indicate input regions that activate their respective classes. One could interpret these heat maps as to *where* in the image the network pays attention to when it makes its prediction. An example of a CAM obtained through this method is available in Fig. 5.1: it demonstrates that the network pays attention to the airplane on the left and ignores the object on the right when making its prediction. In this and all the following examples the feature maps were upsampled using bilinear interpolation.

Obtaining CAMs is fairly simple in architectures that already have feature maps that encode class-level features. This was, in fact, one of the advantages claimed by the authors of the NiN architecture [41]. Several other architectures use GAP in a slightly different way: they perform pooling of convolutional features and then map the resulting vector into class scores using a fully-connected layer. Denoting $\mathbf{v}$ the vector of features obtained after GAP, $\mathbf{W}$ the weight matrix of the FC layer, and $\mathbf{b}$



(a) Input image. (b) Generated activation map for predicted class 'airplane'. (c) Heat map on top of the reference image.

Figure 5.1: Example of a class activation map obtained with the NiN architecture. Images were scaled for easier visualization.

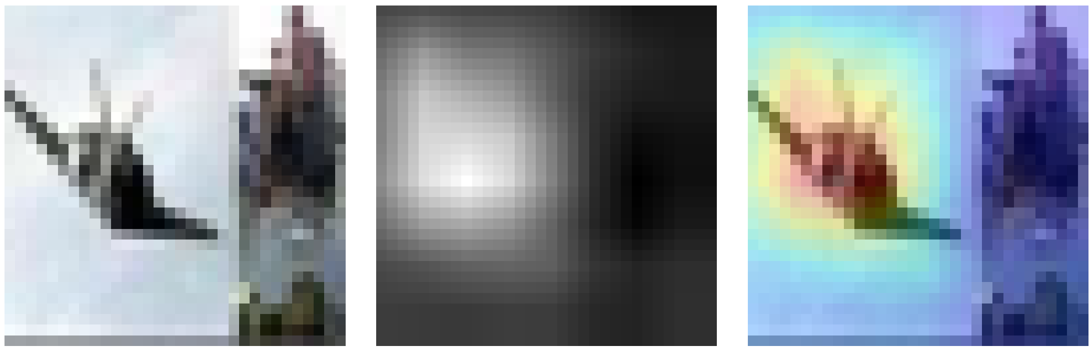
the corresponding bias vector, this operation is described as below:

$$\begin{aligned} v_i &= \text{GAP}(f_{i,h,w}) \\ v_i &= \frac{1}{HW}\sum_{h,w} f_{i,h,w} \\ c_j &= \text{FC}(\mathbf{v}) \\ &= \mathbf{w}_j^T\mathbf{v} + \mathbf{b} \\ &= \sum_i w_{j,i}v_i + b_j \\ &= \frac{1}{HW}\sum_{h,w,i} w_{j,i}f_{i,h,w} + b_j. \end{aligned} \tag{5.2}$$

In short, each weight matrix column $\mathbf{w}_j$ encodes how important each feature map is for its respective class. Note that Eq. 5.2 can be rewritten as

$$\begin{aligned} c_j &= \frac{1}{HW}\sum_{h,w}\left(\sum_i w_{j,i}f_{i,h,w} + b_j\right) \\ &= \text{GAP}\left(\sum_i w_{j,i}f_{i,h,w} + b_j\right) \\ &= \text{GAP}(m_{j,h,w}). \end{aligned} \tag{5.3}$$

Eq. 5.3 demonstrates that using an FC layer after GAP is equivalent to performing GAP on a transformation of the feature maps $\mathbf{f}_j$, denoted as $\mathbf{m}_j$. In fact, this transformation is easily implemented as a 1×1 convolutional layer (without any nonlinearity). Note that this equivalent representation has the same form of the NiN architecture: a 1×1 convolution that generates feature maps that encode class-level



(a) Input image. (b) Generated activation map for predicted class 'airplane'. (c) Heat map on top of the reference image.

Figure 5.2: Example of a class activation map obtained with the DenseNet architecture. Images were scaled for easier visualization.

features before the GAP layer[1]. CAMs may then be extracted by transforming the output feature maps using a 1×1 convolutional layer that shares the weights of the FC layer, $\mathbf{W}$ and $\mathbf{b}$.

This transformation can be applied to networks such as the ResNets and DenseNets presented in Chapter 4. An example is shown in Fig. 5.2: the CAM for the same input image presented in Fig. 5.1 is obtained using the DenseNet-$(40, 12)$ trained on the CIFAR-10+ dataset. Note that not only the DenseNet-$(40, 12)$ model achieves better classification performance, but its CAM is also more well localized than the NiN model. This agrees with the intuition that networks classify images by learning to detect objects in them [93].

5.1.1 Weakly Supervised Localization

It is expected that regions that contribute heavily in the classification process are probably regions that contain the object of interest itself. This means that the activation map may be used as a guideline for localizing the object of interest in the image. In order to obtain a bounding box, the activation map is thresholded and the smallest box that contains all activations above such threshold is selected[2]. For our experiments, we use the DenseNet-$(40, 12)$ architecture trained on CIFAR-10+. Some examples are shown in Fig. 5.3. It is interesting that, although the network has never been trained on any localization task, through this simple method we are able to extract reasonable bounding boxes. In several examples, the bounding

[1]The only difference is that the last convolutional layer in NiN includes the nonlinearity, which is not the case for the equivalent representation obtained.

[2]This process is simpler than the one used in [95], where an algorithm selects the bounding box that covers the largest connected component in the thresholded map. The images studied in this work are smaller (and simpler), so a simpler method still performs adequately for the purposes of this discussion.

boxes only contain part of the object in the image: this is because the network focuses mostly on the most discriminant parts of each class, such as animal faces. Ultimately, the database is very simple, and the small size of the images limits the variations of position and perspective the objects are available on. Despite such limitations, these simple cases are still able to showcase how networks naturally learn to localize objects. More complex examples with larger images are available in works such as [93–95].

5.1.2 Investigating Classification Mistakes

CAMs may also be used to understand mistakes made by the network: by seeing where the network was looking at when it made its prediction, it may be possible to explain why the image was misclassified. Several examples of mistakes are shown in Fig. 5.4. The first example is the image of a dog that is misclassified as a cat. By looking at the CAM for the predicted class, we see that the network most likely classified the image as a cat because of the apparent striped pattern of the fur. The network seems to ignore the dog's muzzle, probably the most dog-discriminant part of the image, when making its predictions. The second example also features a dog. This image is misclassified as a horse, with the second most probable class being deer. This probably happens because most horse and deer images in the dataset have this sort of perspective, with the animal standing with the whole body exposed. Pictures of dogs are usually taken from a closer perspective, with the face appearing prominently. Also note that the biggest evidence that this is in fact a dog is the person close to it: humans are able to easily perceive the relative sizes of the animal and the person standing close to it. We can see through the CAM that the network does not pay attention to the person standing there: this sort of relationship is not captured by the network. The third example is the image of a frog that is incorrectly classified as a dog. Interestingly, by analyzing the CAM for the ground truth label, we see that the network does identify the correct region as being very "frog-like"; it just weighs other regions as more strongly discriminant towards other classes. The fourth example is a picture of a white dog with gray ear looking to the right. This image is classified as bird by the network, with the correct class receiving about 35% probability. The CAM for the dog class indicates that the network is in fact able to localize the dog's face. The CAM for bird, however, is focused on a slight different part of the image: it seems that the network interprets the dog's ear as a gray beak. Because of the low resolution, the dog's muzzle is not very clear, and the image is misinterpreted as a white bird with gray beak looking to the left. One feature that points to the animal being a dog is the collar: it is much more common for dogs to wear collars than birds. The network, however, is not able to make such indirect

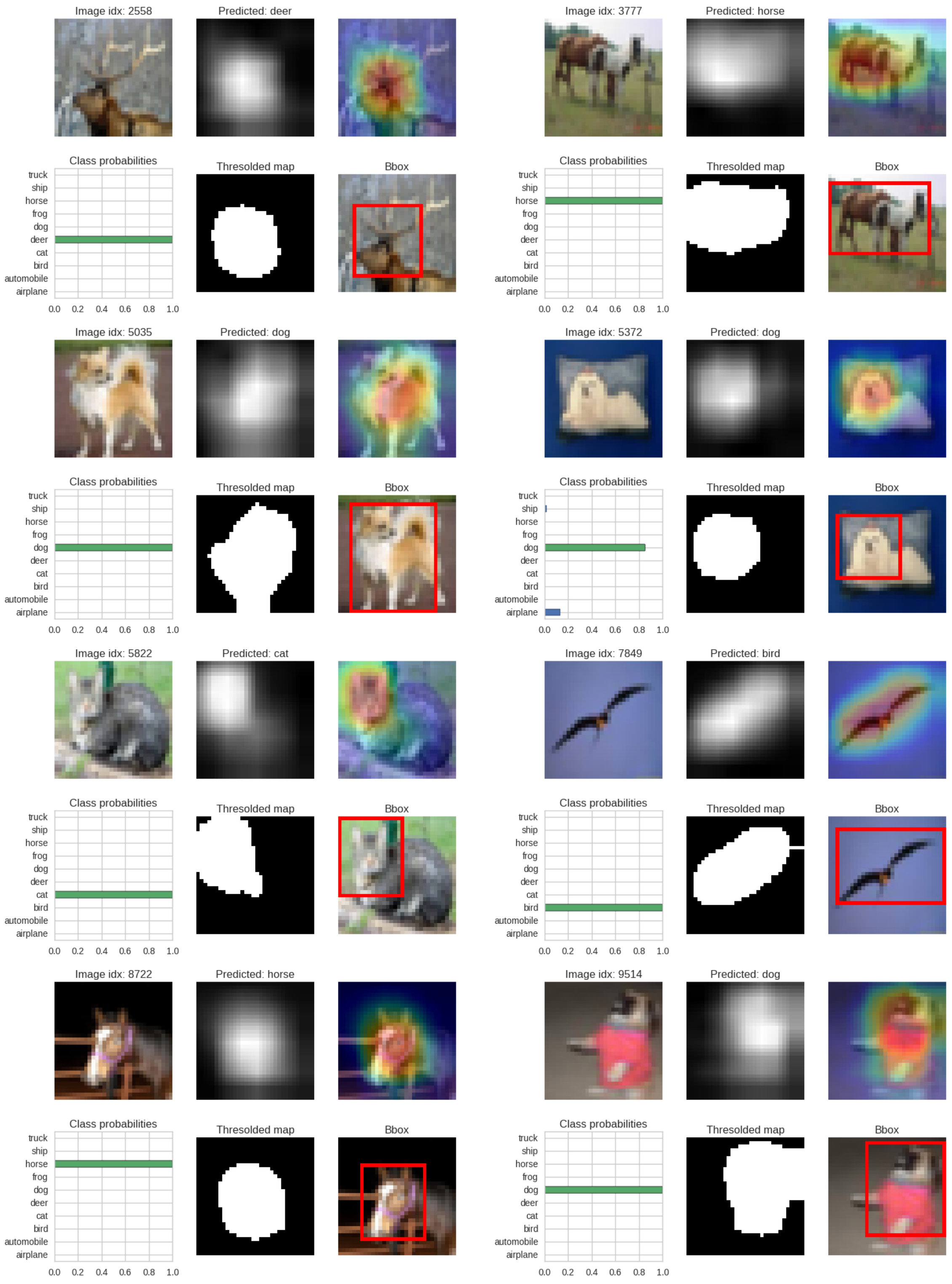


Figure 5.3: Examples of weakly supervised localization using thresholded CAMs.

associations.

5.2 Neural Style Transfer

Consider the task of reproducing an image in the style of another, that is, obtaining an image $\mathbf{p}$ that is a combination of the *content* of a content image $\mathbf{c}$ and the *style* of a style image $\mathbf{s}$. Artists have practiced this art form that is known as pastiche. Our interest is in obtaining an algorithm that is capable of automatically transferring the style from $\mathbf{s}$ while maintaining the perceived content of $\mathbf{c}$.

In other words, the desired image $\mathbf{p}$ is the one that minimizes the loss function

$$l(\mathbf{p}, \mathbf{c}, \mathbf{s}) = \lambda_{\mathrm{c}} l_{\mathrm{c}}(\mathbf{p}, \mathbf{c}) + \lambda_{\mathrm{s}} l_{\mathrm{s}}(\mathbf{p}, \mathbf{s}), \tag{5.4}$$

where l_{s} and l_{s} are a loss functions that represent how much the content of $\mathbf{p}$ differs from the content of $\mathbf{c}$ and the style of $\mathbf{p}$ differs from the style of $\mathbf{s}$, respectively. The scalars λ_{c} and λ_{s} establish a compromise between the fidelity of the content and how stylized the image is. The quality of the pastiche image depends on how well these two loss functions capture aspects that represent content and style. Because of how the loss functions capture how the images are perceptually different in terms of content and style, or both, they are referred to as *perceptual losses.* l_{c} and l_{s} are, therefore, the content and style perceptual losses, respectively. l will be called total perceptual loss or simply perceptual loss.

5.2.1 Content Representation

The content loss should be low for images that share the same perceptual content. Pixel-domain losses such as MSE (mean square error) are not adequate because images rendered in different styles will most likely have very different pixel values even if they have similar perceptual content. It is necessary to compare images on more abstract levels.

Convnets trained in large object recognition datasets extract features that are capable of capturing discriminant aspects of a wide variety of objects. Because of the variability of poses, lighting, and position of the objects, these features must also be able to capture information in a form that is resilient to a variety of transformations. These features can be thought of as alternative representations of the image. Related work has shown that images can be recovered with varying levels of abstraction by finding inputs that share similar feature representations to the original image [82, 83]. The level of abstraction of the representations depend on the depth of the layer from which the features are obtained. For deeper layers, images may be very different in

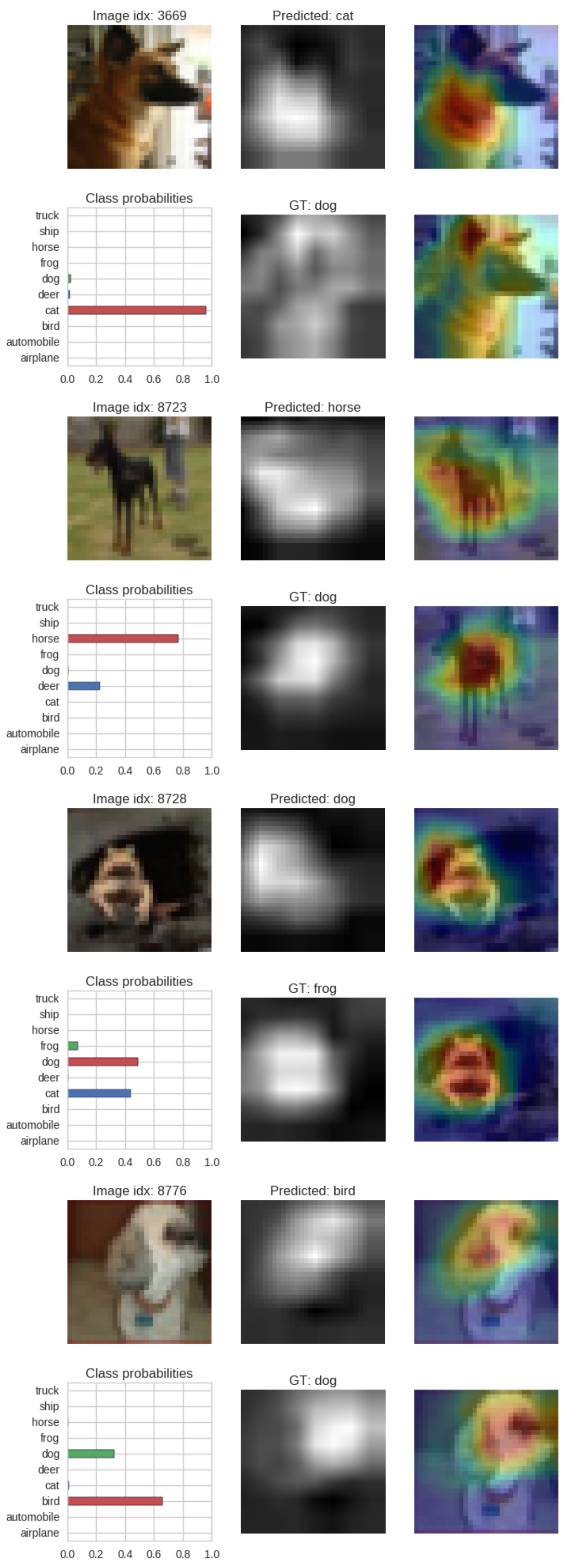


Figure 5.4: Examples of classification mistakes. First row contains the input image, the CAM for the predicted class and the respective heat map, respectively. Second row contains a bar graph of the predicted class probabilities, the CAM for the ground truth class and the respective heat map.

the pixel domain and still have similar feature representations if they share common elements (borders, objects, etc.).

The content loss function l_c is then characterized by the weighted sum of the MSE between features computed on different layers of a covnet [97]. Consider a convnet with input $\mathbf{x}$. The output of its n-th layer is a tensor that is a function of the input $f^{(n)}(\mathbf{x}) \in \mathbb{R}^{H^{(n)} \times W^{(n)} \times C^{(n)}}$. For a chosen set of content layers $\mathcal{C}$, the content loss function is

$$l_{\mathrm{c}}(\mathbf{p}, \mathbf{c}) = \sum_{n \in \mathcal{C}} \frac{w_{\mathrm{c}}^{(n)}}{H^{(n)} W^{(n)} C^{(n)}} \sum_{h,w,c} \left(f_{h,w,c}^{(n)}(\mathbf{p}) - f_{h,w,c}^{(n)}(\mathbf{c}) \right)^2, \tag{5.5}$$

where $w_{\mathrm{c}}^{(n)}$ is the weight of layer n in the computation of the content loss. $f^{(n)}(\mathbf{p})$ and $f^{(n)}(\mathbf{c})$ are expected to be similar if $\mathbf{p}$ and $\mathbf{c}$ are perceptually alike, even if the pixel values are very different. Because of how the network is used to define the loss function, it is called the loss network.

The depth of the layers affect the representations that are being compared: shallow features are more spatially localized, while deeper features grasp more complex relationships [39, 80–83]. Shallow output features might be similar if both images contain similar edges and other important object boundaries. Deeper output features might be similar even if spatial arrangements are different so long as both images represent similar objects.

5.2.2 Style Representation

Much like content, style is a relatively loose term. It is used to describe *how* objects are depicted and not *what* they are. In this sense, content and style are disjunct. For this reason, the style loss function should not directly capture how the feature maps differ. Inspired by previous work on texture synthesis [98, 99], the authors of [97] propose a style representation that essentially captures texture information. Textures are descriptors that preserve local spatial relationships but discard global arrangement. In the case of paintings, textures might represent elements of style such as brush strokes, color arrangements, and common visual motifs.

The textures of an image are represented by correlations between activations of feature maps. Consider again the output of the n-th layer of the loss network $f^{(n)}(\mathbf{x})$. The texture information is given by the Gram matrix $\mathbf{G}^{(n)} \in \mathbb{R}^{C^{(n)} \times C^{(n)}}$, whose elements are computed as

$$g_{i,j}^{(n)} = \frac{1}{H^{(n)} W^{(n)}} \sum_{h,w} f_{h,w,i}^{(n)}(\mathbf{x}) f_{h,w,j}^{(n)}(\mathbf{x}). \tag{5.6}$$

Each element $g_{i,j}^{(n)}$ represents the correlation between activations of feature maps i and j, with the expectation taken spatially. If we denote as $F^{(n)} \in \mathbb{R}^{H^{(n)}W^{(n)} \times C^{(n)}}$ the matrix of vectorized feature maps, the Gram matrix can be efficiently computed as

$$\mathbf{G}^{(n)} = \frac{1}{H^{(n)}W^{(n)}} \left(F^{(n)}\right)^T F^{(n)}. \tag{5.7}$$

By computing correlations between feature maps for different layers, textures can be described in multiple scales. The style loss function is then the weighted sum of the MSE between Gram matrices computed with the feature maps obtained with both $\mathbf{p}$ and $\mathbf{s}$ as inputs. $\mathbf{G}^{(n)}$ and $\bar{\mathbf{G}}^{(n)}$ denote the Gram matrices computed with features extracted from $f^{(n)}(\mathbf{s})$ and $f^{(n)}(\mathbf{p})$, respectively. For a set of style layers $\mathcal{S}$, the style loss is:

$$l_{\mathrm{s}}(\mathbf{p}, \mathbf{s}) = \sum_{n \in \mathcal{S}} \frac{w_{\mathrm{s}}^{(n)}}{\left(C^{(n)}\right)^2} \sum_{i,j} \left(g_{i,j}^{(n)} - \bar{g}_{i,j}^{(n)}\right)^2, \tag{5.8}$$

where $w_{\mathrm{s}}^{(n)}$ is the weight of layer n in the computation of the style loss. Since the Gram matrices sizes only depend on the number of feature maps of the output layer, the style loss is well-defined even if the input and style images $\mathbf{x}$ and $\mathbf{s}$ have different spatial dimensions.

5.2.3 Total Variation Regularizer

A third term may be added to the total loss function. It is the total variation (TV) regularization [83] term

$$l_{\mathrm{tv}}(\mathbf{p}) = \sum_{h,w,c} \left((p_{h+1,w,c} - p_{h,w,c})^2 + (p_{h,w+1,c} - p_{h,w,c})^2 \right)^{\frac{\beta}{2}}. \tag{5.9}$$

The scalar β is a hyperparameter that is originally set to one. However, in the presence of pooling layers, $\beta > 1$ is recomended [83], so we use $\beta = 2$. The TV regularizer penalizes images that contain high (finite-diference approximations of) gradients. This leads to smoother images with improved local image coherence.

The total perceptual loss that must be minimized is then

$$l(\mathbf{p}, \mathbf{c}, \mathbf{s}) = \lambda_{\mathrm{c}} l_{\mathrm{c}}(\mathbf{p}, \mathbf{c}) + \lambda_{\mathrm{s}} l_{\mathrm{s}}(\mathbf{p}, \mathbf{s}) + \lambda_{\mathrm{tv}} l_{\mathrm{tv}}(\mathbf{p}), \tag{5.10}$$

where λ_{tv} is the weight of the TV regularizer.

5.3 Iterative Style Transfer

The iterative style transfer algorithm consists of optimizing the image $\mathbf{p}$ so as to minimize loss l. The losses defined in Sections 5.2.1 and 5.2.2 are both differentiable, so the total loss can be optimized with gradient-based methods. The optimization process is carried out by initializing $\mathbf{p}$ with noise and iteratively modifying it so as to minimize the loss. The optimization process is carried out as follows: obtain layer outputs $f^{(n)}(\mathbf{p})$ by computing a forward pass through the network; compute the gram matrices $\bar{\mathbf{G}}^{(n)}$; compute the loss functions l_{c} and l_{s}. After the total loss is computed, one can compute the gradient w.r.t $\mathbf{p}$ by backpropagating gradients up to the input. The gradient w.r.t $\mathbf{p}$ may then be used by an optimizer to make a step that changes the image to minimize the loss. A diagram of the process is available in Fig. 5.5a. It is expected that the image converges to an image that simultaneously contains the content of $\mathbf{c}$ and the style of $\mathbf{s}$. Since $\mathbf{c}$ and $\mathbf{s}$ are fixed during the optimization process, their features can be computed once and reused. Note that the network weights *are not* being optimized: the network is only a part of the loss function used to update $\mathbf{p}$.

5.3.1 Experiments

Several experiments were performed in order to evaluate the quality of the generated images. Images $\mathbf{p}$ were either initialized with Gaussian noise with a small variance or with the content image itself. Both initialization techniques yield images that combine content and style, but initializing with the content image usually leads to images that more closely resemble the content image. For images such as faces, which humans are very particular about, such content initialized images may appear more appealing because they retain more strongly the content features. Fig. 5.6 presents some stylized images.

5.3.2 Hyperparameter Discussion

In our experiments, as in [97], we use VGG-19, which was trained on the Imagenet dataset [68, 69], as the loss network [70]. This network (with its pretrained weights) as well as many other popular networks trained on Imagenet are available freely on the internet[3]. The content layer set $\mathcal{C}$ is composed of only one layer: 'conv4_2'. The style layer set $\mathcal{S}$ includes layers 'conv1_1', 'conv2_1', 'conv3_1', 'conv4_1', and 'conv5_1'. In regard to the optimizer, [97] proposes the use of L-BFGS [100].

[3]VGG-19 is available in Keras out of the box and its weights are automatically downloaded if necessary.

For practical reasons[4], our experiments are performed with Adam. Since we use a first-order method, the learning rate is also a hyperparameter (the other Adam parameters are kept at standard values). For the sake of simplicity, all style weights $w_s^{(n)}$ are the same: this choice removes several hyperparameters and still yields pleasing results [97].

The hyperparameters used can heavily influence the aspect of the pastiche image. The style image size and loss term weights are particularly important. The loss term weights influence the trade-off between content fidelity, stylization, and smoothness. An image generated with very high content weight will be very similar to the original, but carry little style. If the image is generated with high style weights, the result will not resemble the content image. Instead, it will be a textured version of the style image. The tv regularization term can also influence the image: adding some of it may help smooth the image and avoid pixelated results. If too much regularization is added, images get blurred and may have artifacts. Fig. 5.7 shows images obtained from the same content and style images with varying loss term weights.

It is also very important to choose an appropriate image size for the style image when computing its Gram matrices: images optimized with target Gram matrices computed at larger sizes feature larger smoother patterns. If the style image is scaled to small sizes, the resulting image will have its style represented by small textures. In order to illustrate this effect, Fig. 5.8 contains images obtained by different scalings of the style image. The style image was scaled preserving its aspect ratio so that its maximum size (height or width) was a set value. This maximum size was varied so that the algorithm could be evaluated with different image scales.

5.4 Fast Style Transfer

The iterative algorithm has some drawbacks. Namely, its optimization procedure requires several forward and backward passes through a big neural network. This makes the algorithm quite slow: images take several minutes to be generated. The time also scales with the input image size: the larger the image, the more time it takes. This makes the algorithm impractical for large images or in restricted computation environments.

In place of iteratively refining an initial image, one could train a neural network, which we call pastiche network, to perform stylization. After the network is trained, style transfer could be achieved in a single forward pass of the pastiche network. Two distinct network architectures were proposed [22, 92]. We follow the work of [22].

The pastiche network works as follows: it receives as input a content image $\mathbf{c}$

[4]Our code is implemented in Keras and Tensorflow, which lack an L-BFGS optimizer implementation.

and outputs a pastiche image $\mathbf{p} = \mathcal{P}(\mathbf{c}; \boldsymbol{\theta})$, where $\mathcal{P}$ represents the pastiche function realized by the network with weights $\boldsymbol{\theta}$. The pastiche image $\mathbf{p}$ has the content of $\mathbf{c}$ and the style of a style image $\mathbf{s}$.

5.4.1 Training procedure

The learning procedure is performed by minimizing expected perceptual loss over *the pastiche network weights*:

$$\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{c}} \left[l(\mathcal{P}(\mathbf{c}; \boldsymbol{\theta}), \mathbf{c}, \mathbf{s}) \right]. \tag{5.11}$$

Note that, after training, the network does not require the style image to generate outputs: *the style is encoded in the network weights.* Conversely, the network learns to stylize according to the single style image that is used during training.

The perceptual loss used is the same: it is computed by comparing features of a loss network. The main difference is that this loss is used to minimized the weights, instead of an input image. The gradients are computed by performing forward and backward passes on both networks: $\mathbf{p}$ is computed with a forward pass on the pastiche network; $\mathbf{c}$, $\mathbf{s}$, and $\mathbf{p}$ are used to compute the loss, which includes a forward pass on the loss network; finally, the gradients are backpropagated from the loss through the loss and pastiche networks so as to compute the gradients w.r.t $\boldsymbol{\theta}$.

5.4.2 Pastiche Network Architecture

The pastiche network is a fully convolutional network [19]. It features a bottleneck structure where the image is initially downsampled, processed by several convolutional layers, and upsampled back to its original size. This bottleneck structure is interesting because it allows for the use of more filters in the bottleneck layers while keeping computation cost low. It also allows for the output pixels to have larger effective input receptive fields, which is interesting because transferring style might involve coherently changing large patches of the input image [22].

The image is first transformed a regular convolutional layer. Then, it is downsampled by two stride-2 convolutions. Afterwards, the downsampled feature maps are processed by five residual blocks. The image is then upsampled back to its original spatial dimensions with two upsampling blocks. A last convolutional layer is used to map the upsampled feature maps into an RGB image.

The upsampling blocks in [22] were composed of *transposed convolutional* layers[5] [19]. These layers can cause checkerboard pattern artifacts on the output im-

[5]This layer is also commonly called *deconvolutional* layer. We avoid this name because the operation performed by this layer is not a deconvolution, i.e., the operation that inverts a convolution.

age [101]. Instead, we follow [102] and use upsampling blocks that are composed of nearest-neighbor upsampling and a regular stride-1 convolutional layer.

All convolutions are followed by normalization and ReLU nonlinearities, except for the last convolution (that generates the output). The last convolution of the network features tanh nonlinearity scaled to the range $(-150, 150)$[6]. The architecture from [22] uses batch normalization layers. Later work has shown that instance normalization layers can be used for improved perfomance [103]. All our networks are trained with instance normalization layers.

Instance normalization is similar to batch normalization in the sense that samples are normalized by mean and variance. The difference is that, instead of computing batch statistics, mean and variance are computed separately per sample. For the b-th sample $\mathbf{x}^{(b)} \in \mathbb{R}^{H\times W\times C}$ in a mini-batch, the instance normalized output $\hat{\mathbf{x}}^{(b)}$ is computed according to

$$\begin{aligned} \mu_c^{(b)} &= \frac{1}{HW}\sum_{h,w} x_{h,w,c}^{(b)} \\ v_c^{(b)} &= \frac{1}{HW}\sum_{h,w}\left(x_{h,w,c}^{(b)} - \mu_c^{(b)}\right)^2 \\ \hat{x}_{h,w,c} &= \gamma_c \frac{x_{h,w,c} - \mu_c^{(b)}}{\sqrt{v^{(b)} + \epsilon}} + \beta_c. \end{aligned} \tag{5.12}$$

The scalar ϵ is a small number that avoids division by zero. Like in batchnorm, γ_c and β_c are learnable parameters. The per-instance normalization used is equivalent to contrast normalization.

The number of filters in the convolutional layers is doubled every time a downsampling occurs and halved every time an upsampling occurs, keeping computation costs similar throughout the network. For a concise representation of the network architecture, see Table 5.1.

5.4.3 Fast Style Transfer Experiments

The experiments were performed as in [22]. The networks were trained on the COCO (Common Objects in COntext) dataset [104]. The COCO dataset contains around 80000 images on the training set of varied sizes. Images are resized so that their smallest size is 256 and center cropped so that all images are 256×256. The images preprocessing is the same as applied to loss network inputs, which is per-channel mean subtraction. Since the loss network was trained on the ImageNet dataset, the

The forward function it implements is actually the backward function of a convolutional layer (and vice-versa). This layer is also called *backwards* convolution, and *fractional-stride* convolution.

[6]This range is chosen so that the network outputs have similar scale to the inputs the loss network was trained on.

Table 5.1: Architecture and hyperparameters of the pastiche network.

Operation	Kernel size	Stride	Feature maps	Nonlinearity	Output size
Network - $256 \times 256 \times 1$ input					
Convolution	9×9	1	16	IN-ReLU	$256 \times 256 \times 3$
Downsampling			32		$128 \times 128 \times 32$
Downsampling			64		$64 \times 64 \times 64$
Residual Block			64		$64 \times 64 \times 64$
Residual Block			64		$64 \times 64 \times 64$
Residual Block			64		$64 \times 64 \times 64$
Residual Block			64		$64 \times 64 \times 64$
Residual Block			64		$64 \times 64 \times 64$
Upsampling			32		$128 \times 128 \times 32$
Upsampling			16		$256 \times 256 \times 16$
Convolution	9×9	1	3	150(IN-tanh)	$256 \times 256 \times 3$
Downsampling - K					
Convolution	3×3	2	K	IN-ReLU	
Residual Block - K					
Convolution	3×3	1	K	IN-ReLU	
Convolution	3×3	1	K	*Linear*	
Shortcut	*add input to the output*				
Upsampling - K					
Nearest neighbor interpolation	*upsampling factor of 2*				
Convolution	3×3	2	K	IN-ReLU	
Preprocessing	VGG preprocessing				
Optimizer	Adam ($\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$)				
Batch size	4				
Epochs	2				
Weight initialization	Gaussian with 0.05 standard deviation				

means are as computed on that dataset[7]. The networks were trained for a total of 40000 iterations with batch size 4, which corresponds to 2 epochs. The Adam optimizer is used with learning rate of 0.001. Loss term weights w_s, w_c, and w_s were determined on a per-style-basis. The loss network in this case is VGG-16 (a smaller version of VGG-19) [70]. The content layer set is composed of layer conv_2_2. The style layer set is composed of layers conv_1_2, conv_1_2, conv_3_3, and conv_4_3.

Six networks were trained, each using a different style image. Figs. 5.9 and 5.10 presents the stylization obtained by these networks.

5.5 Multi-style Networks

The fast style transfer algorithm presented in Section 5.4 has several advantages compared to the original iterative algorithm. After the pastiche network is trained, it is capable of applying the learned style very fast (even in real time[8] [22]). Since the network is fully convolutional, it is also able to stylize images of any size, even if it was trained only on 256×256 images. Overall, it is a very fast approximation

[7]The R,G, and B means are $123.68, 116.779, 103.939$, respectively.

[8]The authors of [22] provide code that is able to stylize images obtained from a webcam. It is available at: `https://github.com/jcjohnson/fast-neural-style`.

to the optimization method that still yields high quality style transfer. Its major drawback is that a network is only able to learn a single style: in order to be able to transfer different styles, several networks must be trained (a process that takes several hours). This is not an issue in the iterative method. This weakness can be solved with a simple approach: conditional instance normalization [102].

Conditional instance normalization layers apply the same contrast normalization technique used in the regular instance normalization layer. Its main feature is the use of conditioned scaling and bias parameters. Instead of a pair of parameters per channel of the input, a set of parameters are trained. This set contains one pair of parameters *per class*. Consider the $b-$th sample $\mathbf{x}^{(b)} \in \mathbb{R}^{H\times W\times C}$, the conditional instance normalization layer computes the normalized input $\hat{\mathbf{x}}^{(b,s)} = \text{CIN}(\mathbf{x}|s)$, which is conditioned on the style s, as follows:

$$\begin{aligned}
\mu_c^{(b)} &= \frac{1}{HW}\sum_{h,w} x_{h,w,c}^{(b)} \\
v_c^{(b)} &= \frac{1}{HW}\sum_{h,w}\left(x_{h,w,c}^{(b)} - \mu_c^{(b)}\right)^2 \\
\hat{x}_{h,w,c}^{(b,s)} &= \gamma_c^{(s)}\frac{x_{h,w,c} - \mu_c^{(b)}}{\sqrt{v^{(b)} + \epsilon}} + \beta_c^{(s)}.
\end{aligned} \tag{5.13}$$

In summary, the network learns different styles simply by applying different scaling factors at each normalization layer. This approach requires a very small number of extra parameters and does not increase computation requirements.

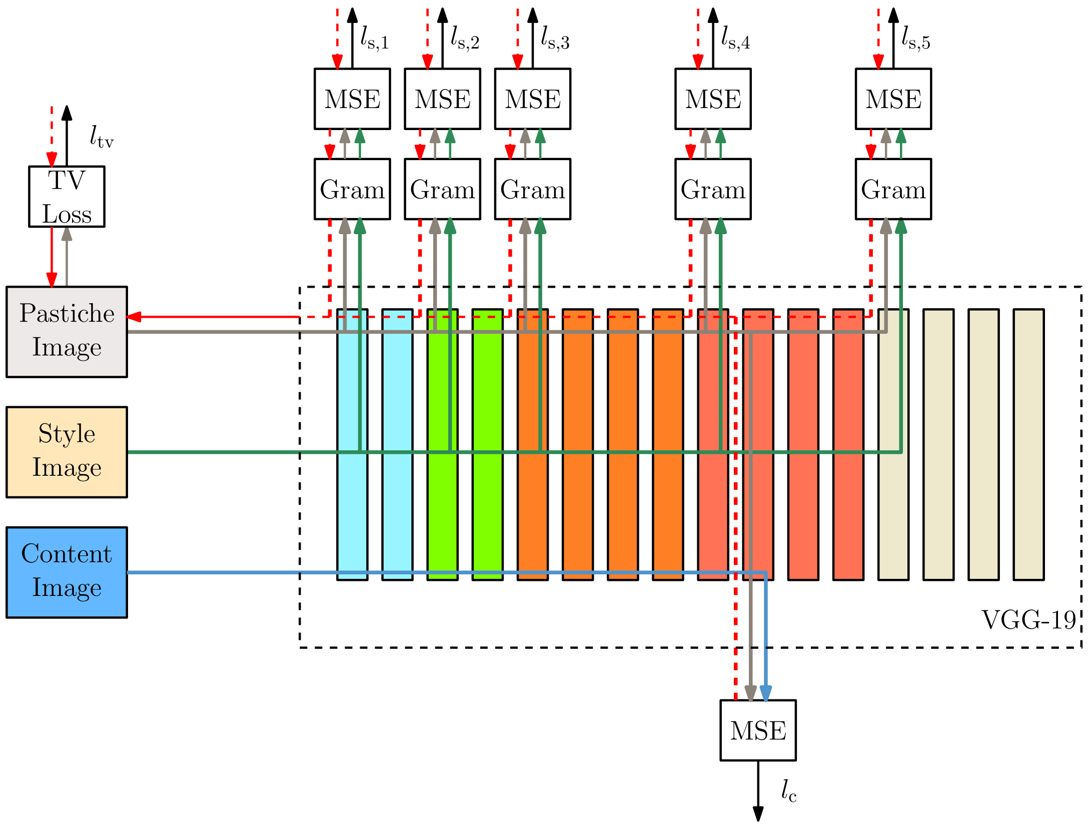
5.5.1 Multi-style Experiments

Training multi-style pastiche networks with conditional instance normalization is very similar to training single style networks. In fact, multi-style networks have convergence properties very similar to those of single-style networks. We trained a 6-style neural network with the same styles as the six single-style networks discussed in Section 5.4.3. The 6-style network has almost the same architecture, the only difference being the substitution of instance normalization by its conditioned version. During training, each sample of a batch is assigned a random style label; the network then outputs the image stylized with that specific set of parameters; the loss function is then computed with the corresponding style images. The 6-style network is trained for the same 40000 iterations and is able to reproduce all styles with comparable quality to the single-style networks. Fig. 5.11 presents a side-by-side comparison of images stylized by the 6-style network and each of the single-style networks.

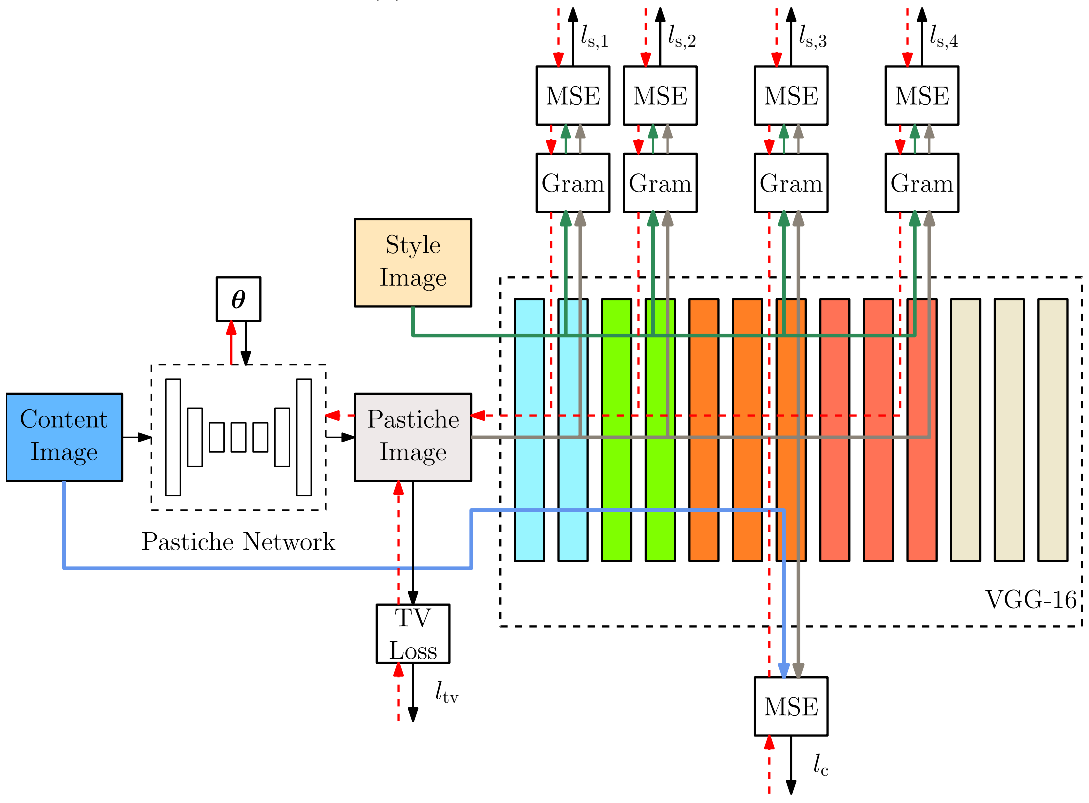
Using a single network for several styles has another advantage: it is possible to very efficiently stylize the input image with a combination of the learned styles using *weighted instance normalization* [102]. Because of how the different styles are encoded in the scale and bias parameters of the normalization layers, using a combination of parameters of different styles during inference yields an image that also combines the stylistic features of the respective styles. Consider a network trained with S styles. An arbitrary combination of such styles is obtained by performing a convex combination of the normalization parameters. Denoting the weight of each style as ρ_s, and the scale and shift parameters for the l-th layer as $\gamma^{(s,l)}$ and $\beta^{(s,l)}$, the combined parameters are:

$$\begin{aligned} \bar{\gamma}_c^{(l)} &= \frac{\sum_s^S \rho_s \gamma^{(s,l)}}{\sum_s^S \rho_s} \\ \bar{\beta}_c^{(l)} &= \frac{\sum_s^S \rho_s \beta^{(s,l)}}{\sum_s^S \rho_s}. \end{aligned} \tag{5.14}$$

In short, interpolating between the normalization parameters results in an interpolation between the respective styles. Note that the network is never trained on any combination of styles: it is naturally able to combine them at test time without any such consideration. This suggests that these networks obtain a representation for style that consists of relatively generic elements of style (encoded in the shared features) that are combined differently (through the conditional parameters) for each specific style. Examples of combined styles are shown in Fig. 5.12.

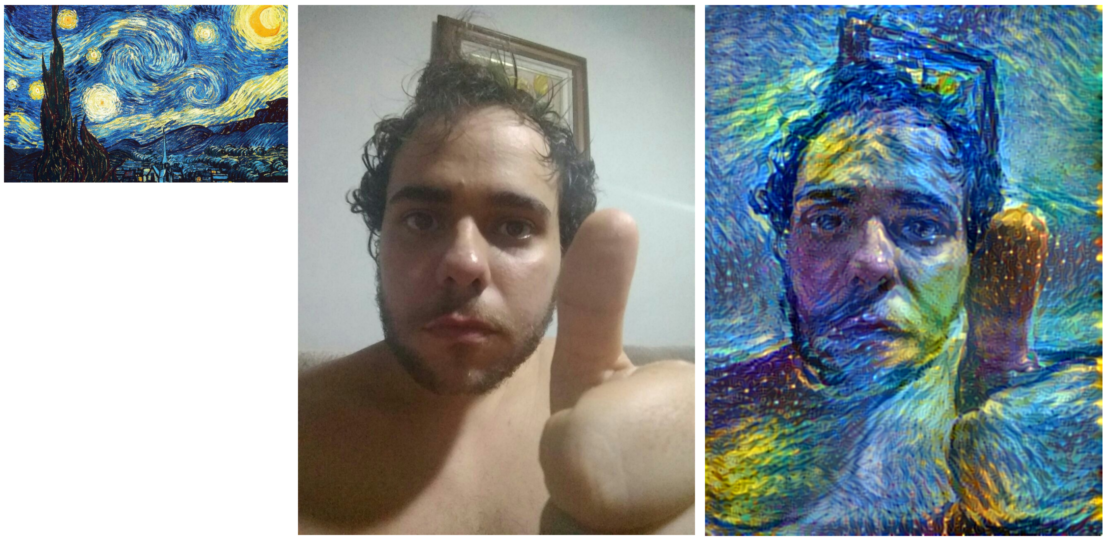


(a) Iterative style transfer.

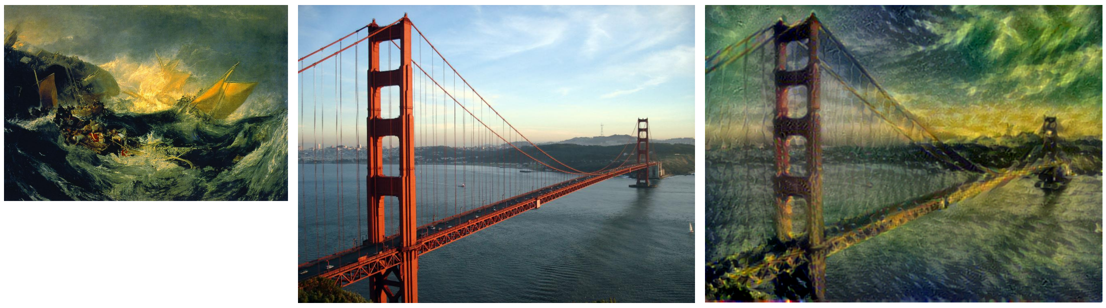


(b) Fast style transfer.

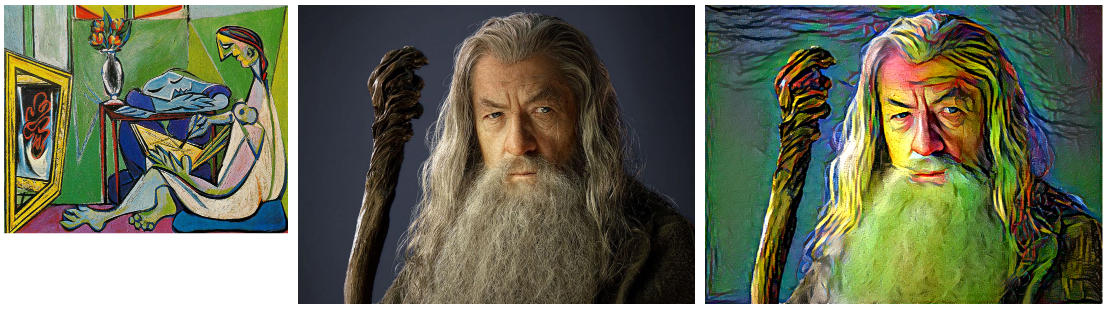
Figure 5.5: Diagrams of style transfer algorithms. Blue, green and gray lines represent the forward propagation for the content, style and pastiche images, respectively. Red lines represent gradient flow. Solid red lines indicate that the gradients are used in the update steps.



(a) Random initialization.



(b) Random initialization.

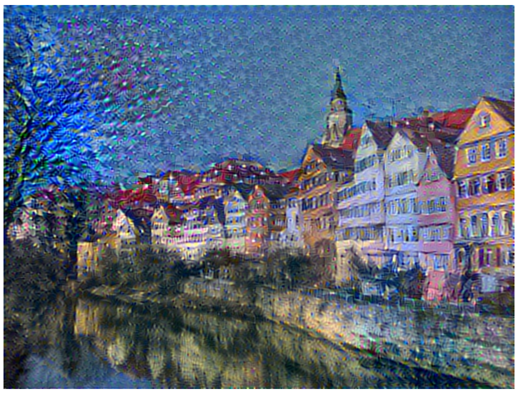


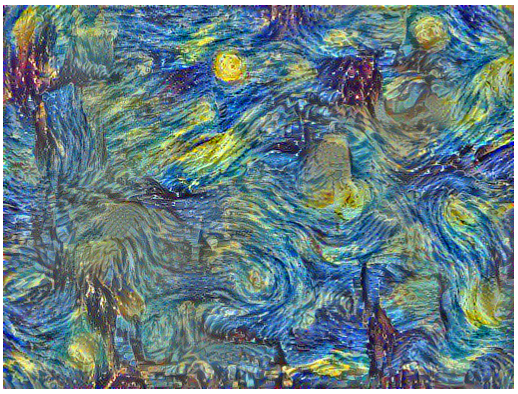
(c) Content initialization.

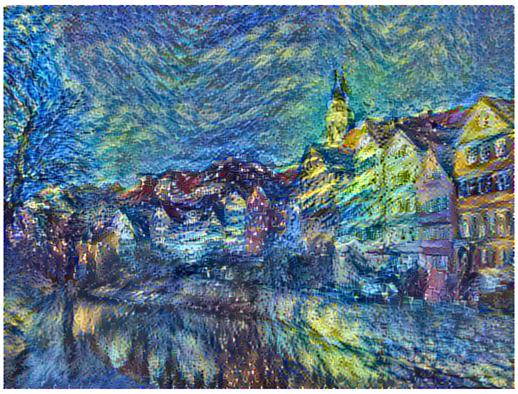
Figure 5.6: Style transfer examples. Columns contain style, content, and resulting images, respectively.

(a) Content image.

(b) Style image.


(c) Small style loss weight.


(d) Large style loss weight.


(e) Small regularization weight.

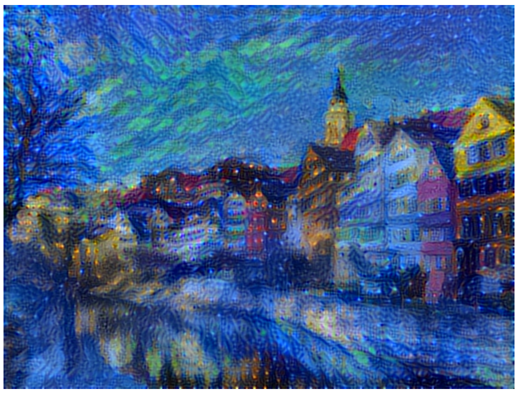
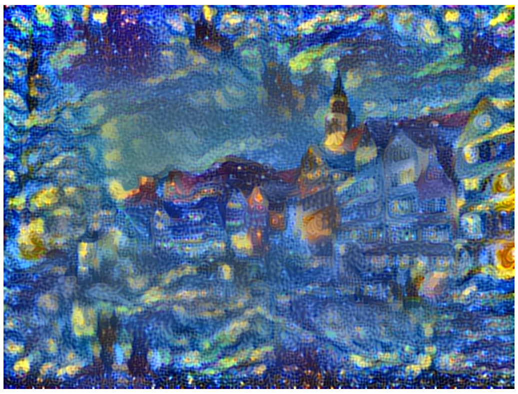
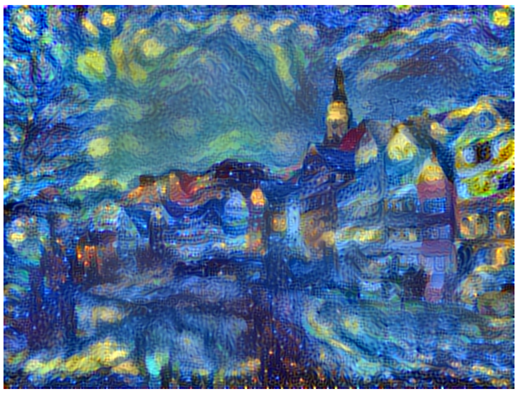

(f) Large regularization weight.

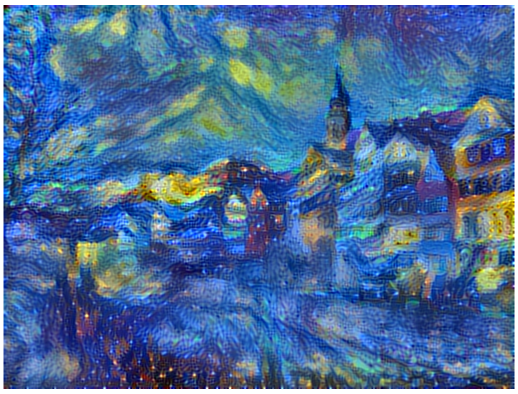
Figure 5.7: Style representation changes with different loss term weights.

(a) Content image.

(b) Style image.


(c) Largest size of 180 pixels.


(d) Largest size of 256 pixels.


(e) Largest size 384 pixels.

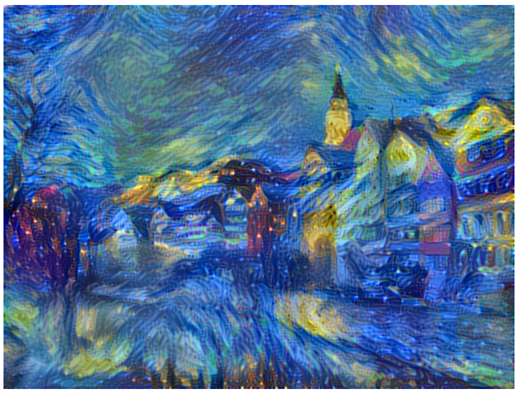

(f) Largest size 512 pixels.

Figure 5.8: Style representation changes with different style image sizes.

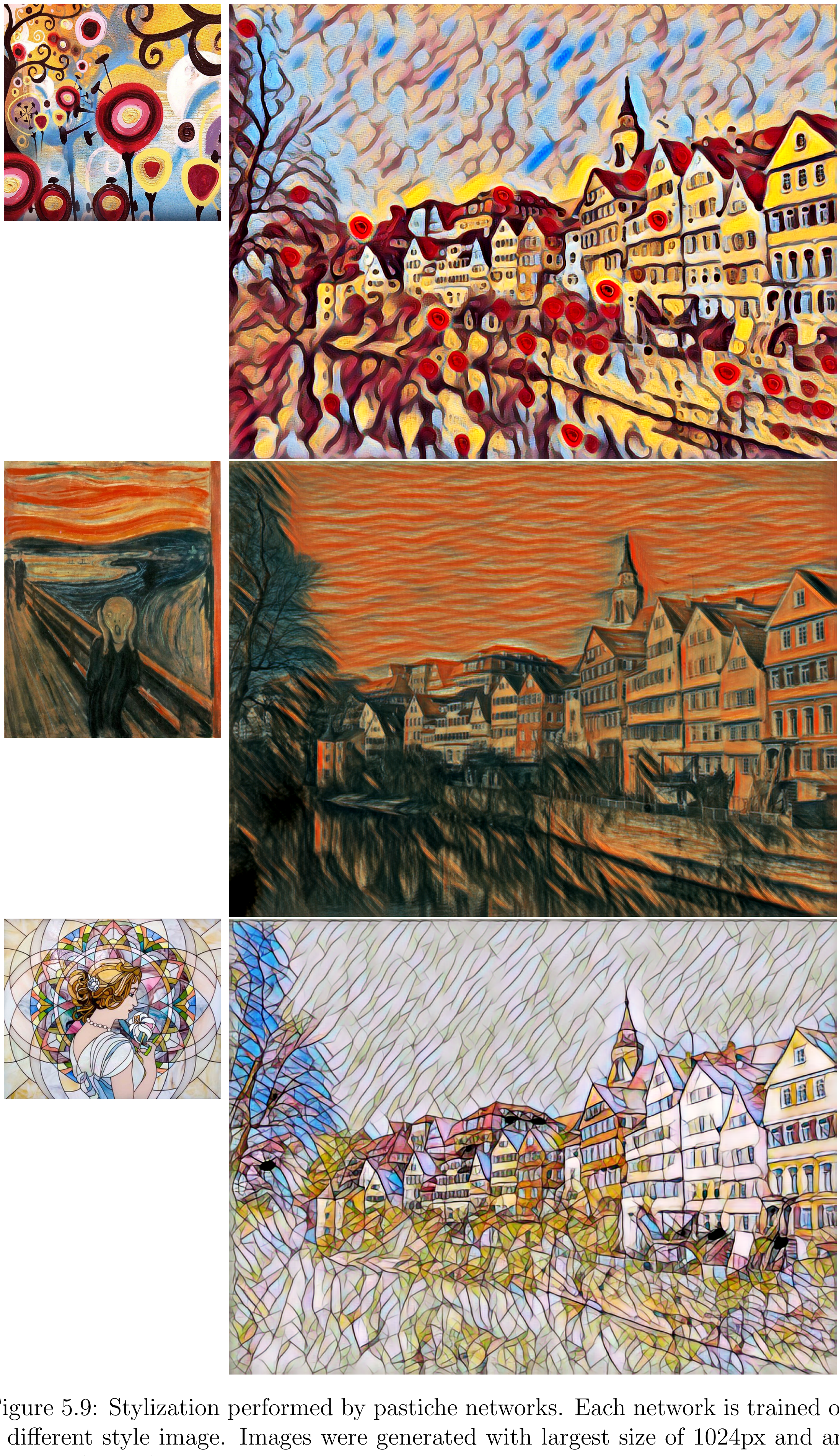


Figure 5.9: Stylization performed by pastiche networks. Each network is trained on a different style image. Images were generated with largest size of 1024px and are rescaled (part 1).

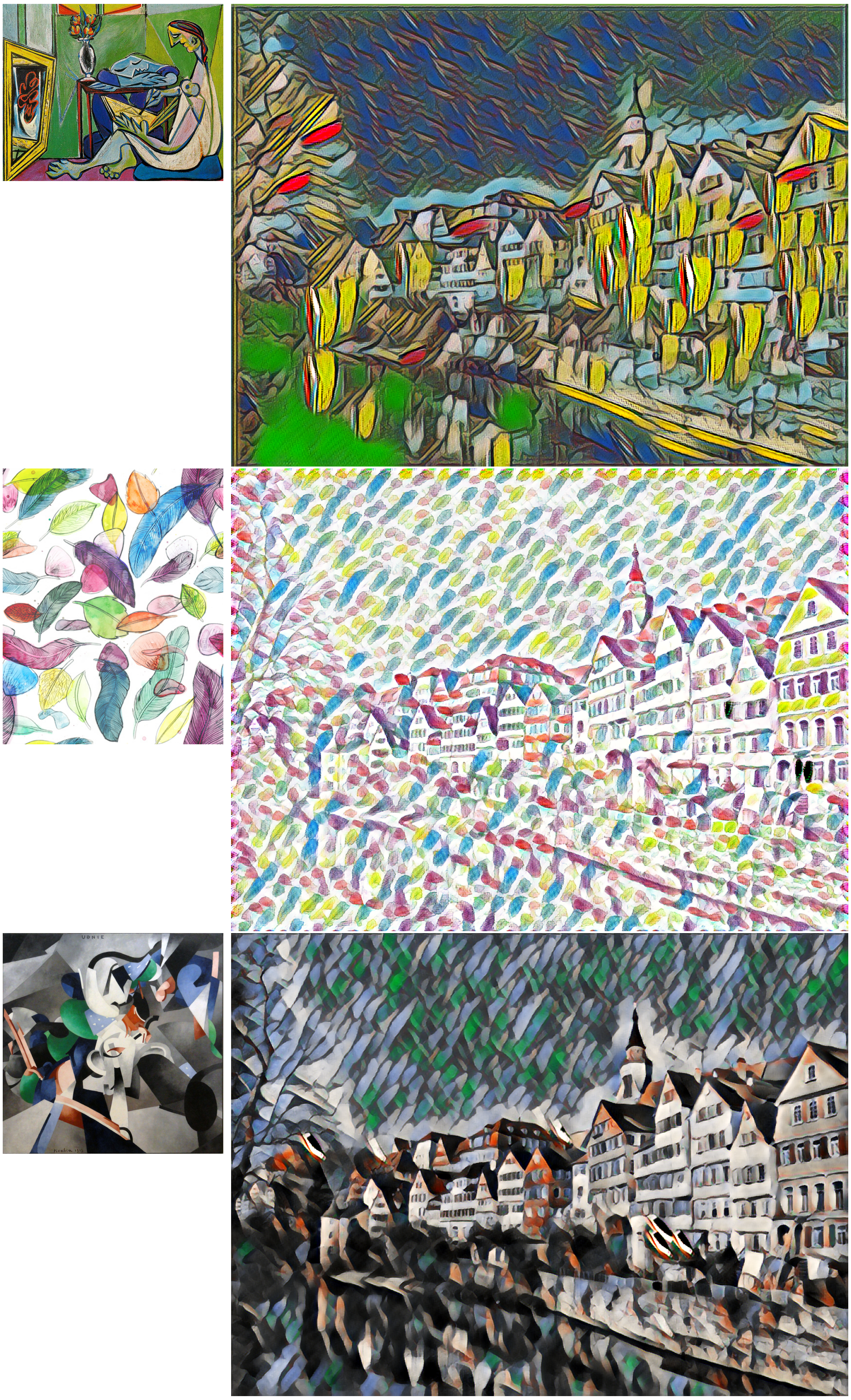


Figure 5.10: Stylization performed by pastiche networks. Each network is trained on a different style image. Images were generated with largest size of 1024px and are rescaled (part 2).

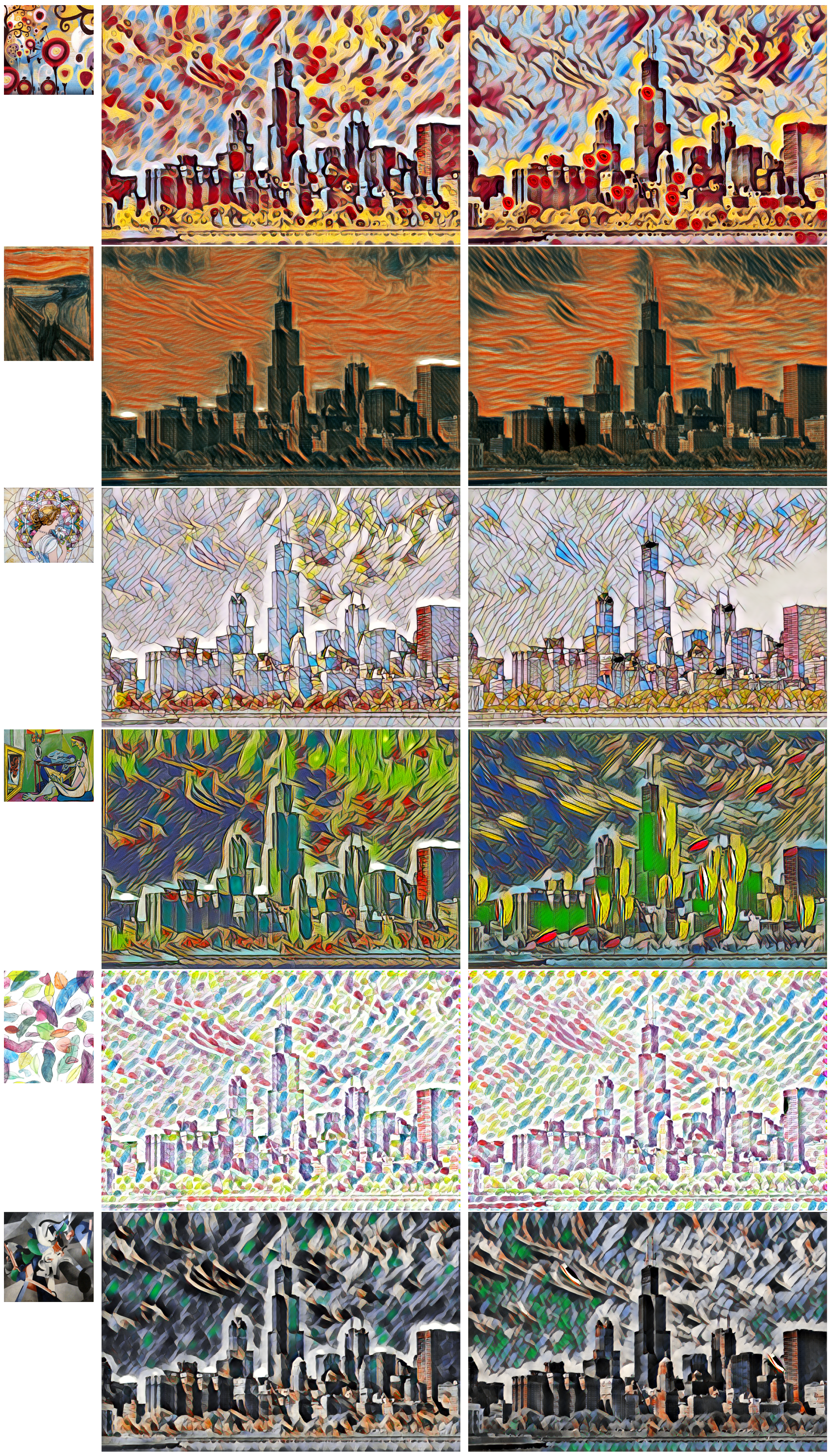


Figure 5.11: Visual comparison of style transfer quality between multi (left) and single (right) style networks.

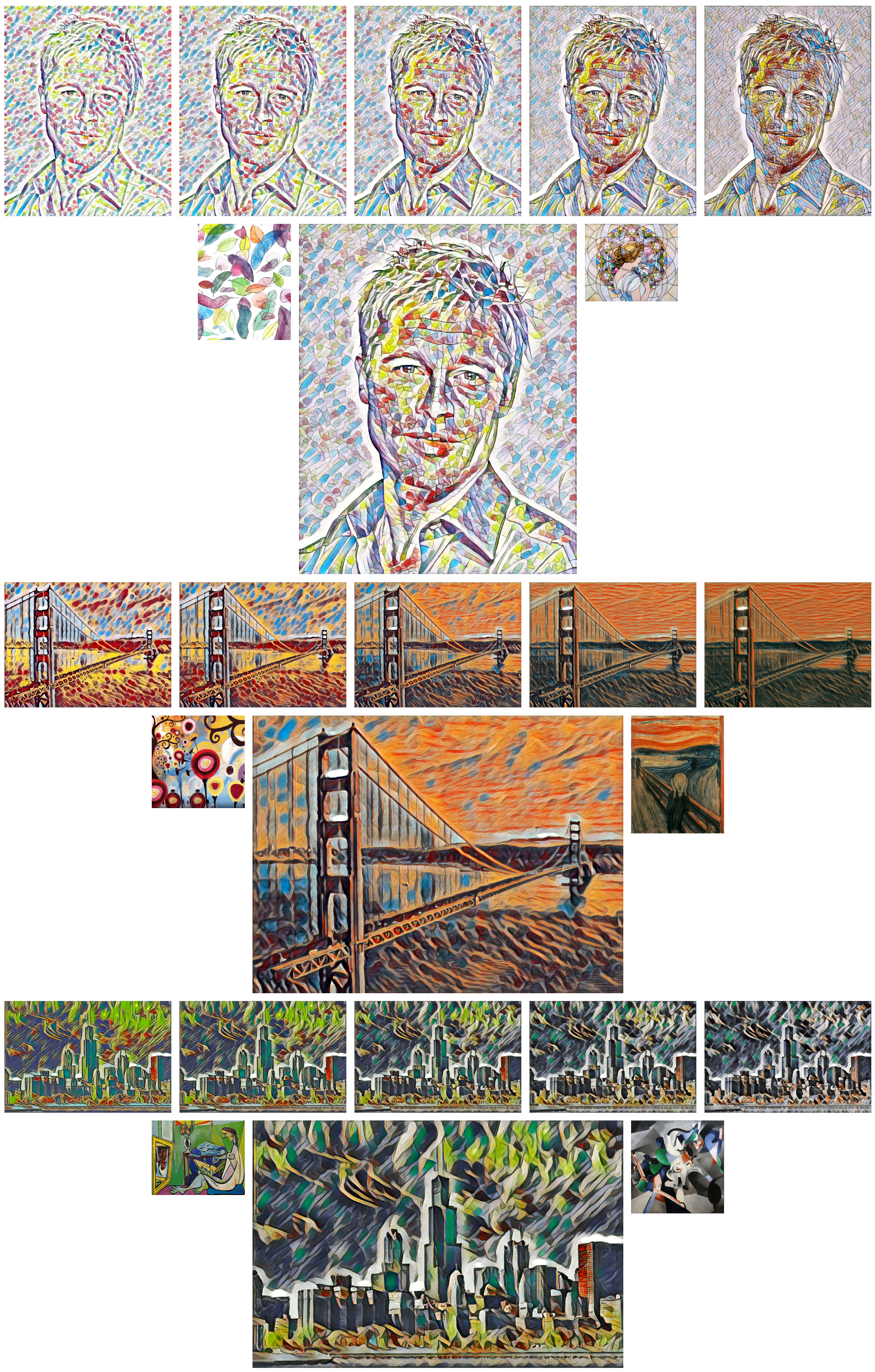


Figure 5.12: Examples of combined styles via weighted instance normalization. Odd rows present images rendered with interpolations of two styles. Even rows present the style images on their respective sides and the image with same weight for both styles in more detail.

Chapter 6

Conclusions

We have presented a study of deep learning techniques applied to computer vision. The reader was introduced to neural networks and their main components. Then, a basic framework of how to fit the network parameters in order to obtain an approximation of the desired function was established. Convolutional neural networks, the motivation behind their design, and their core components were also discussed. Several aspects of how to train and evaluate neural networks were explored, along with techniques that enhance the optimization process and performance by improving generalization.

All these ideas were put to the test through case studies of successful convnet architectures for image classification. Several aspects of such architectures were discussed: the main concepts behind their design, their performance on benchmark datasets, and their training process. All architectures featured detailed explanations about their structure as well as their hyperparameters. This, together with the companion code that is available online should enable readers to easily replicate results and work with these deep learning techniques themselves.

Finally, the use of convnets for other applications is discussed. Most notably, convolutional neural networks are very powerful feature extractors that can be trained to achieve state-of-the-art results in several different domains. Besides, the filters that are learned also generalize well to datasets and tasks other than the ones they were trained on. Through the use of class activation maps, the filters of convnets are applied to weakly supervised localization. CAMs are also useful for explaining predictions by revealing the most discriminant parts of the image. We also study the task of style transfer. This task is interesting not only for its visual appeal, but because of how neural networks are creatively incorporated as a means of capturing perceptual differences in content and style of images. It is also an example of transfer learning, where a neural network trained for classification is able to extract features that can be used to describe images in disentangled representations of content and style. Last but not least, it is also an example of image transformation tasks, for which we discuss a fully convolutional architecture that is able to transform input images in their stylized version through a single forward pass.

6.1 Future Directions

A natural expansion of the study performed in image classification is large scale object recognition: large convolutional networks are able to detect, classify and localize objects in images with state-of-the-art performance. Several ideas studied here for small image classification can be (and have been) applied to large networks. There are, however, other aspects of designing and training large scale networks that are not covered in this text and could be the subject of future study.

A particularly interesting topic that has not been covered in this work is that of generative models. Several algorithms are capable of learning input distributions and then generating samples from it. Variational autoencoders are one of such models that are capable of learning complicated distributions [105–107]. Generative Adversarial Networks (GANs) [108–111] are another approach to generative modeling that has become extremely popular: it incorporates a novel training method called adversarial training that is based on game theory and proposes a game where two networks work against each other. As they compete, one to generate samples and the other ones to discriminate between real data and generated data, the generator learns to output data that follows the desired distribution. GANs can generate samples from *very* complex distributions with high fidelity.

Deep neural networks are also useful in reinforcement learning [112]. Recent work has trained algorithms that are able to play games and beat the very best human players [113]. It has also been used to teach machines how to play video games with only pixel input information [114]. Reinforcement learning has also been applied in robotics [115].

Deep learning is a technique that has recently been applied with great success in many different domains, and the exploration of its applications and state-of-the-art methods is sure to be of much use.

Bibliography

[1] GOODFELLOW, I., BENGIO, Y., COURVILLE, A. *Deep Learning.* Adaptive Computation and Machine Learning series. Cambridge, MIT Press, 2016.

[2] HINTON, G. E., OSINDERO, S., TEH, Y.-W. "A fast learning algorithm for deep belief nets", *Neural Computation*, v. 18, n. 7, pp. 1527–1554, July 2006.

[3] BENGIO, Y., LAMBLIN, P., POPOVICI, D., et al. "Greedy Layer-Wise Training of Deep Networks". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 153–160, Vancouver, British Columbia, Canada, December 2006.

[4] HINTON, G. E. "To recognize shapes, first learn to generate images". In: Paul Cisek, T. D., Kalaska, J. F. (Eds.), *Computational Neuroscience: Theoretical Insights into Brain Function*, v. 165, *Progress in Brain Research*, Elsevier, pp. 535–547, 2007.

[5] HINTON, G. E., SEJNOWSKI, T. J. "Learning and relearning in Boltzmann machines". In: Rumelhart, D., McClelland, J. L. (Eds.), *Parallel distributed processing: explorations in the microstructure of cognition*, Adaptive Computation and Machine Learning series, MIT Press, cap. 7, pp. 283–317, Cambridge, 1986.

[6] HINTON, G. E. "Products of Experts". In: *The 9th International Conference on Artificial Neural Networks (ICANN)*, pp. 1–6, Edinburgh, United Kingdom, September 1999.

[7] HINTON, G. E. "Training Products of Experts by Minimizing Contrastive Divergence", *Neural Computation*, v. 14, n. 8, pp. 1771–1800, 2002.

[8] BENGIO, Y. *Learning Deep Architectures for AI.* Foundations and Trends(r) in Machine Learning. Now Publishers, 2009.

[9] LECUN, Y., BOTTOU, L., BENGIO, Y., et al. "Gradient-based Learning Applied to Document Recognition", *Proceedings of the IEEE*, v. 86, n. 11, pp. 2278–2324, 1998.

[10] KRIZHEVSKY, A., SUTSKEVER, I., HINTON, G. E. "Imagenet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*, pp. 1097–1105, Lake Tahoe, Nevada, USA, December 2012.

[11] NAIR, V., HINTON, G. E. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pp. 807–814, Haifa,Israel, June 2010.

[12] SRIVASTAVA, N., HINTON, G. E., KRIZHEVSKY, A., et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", *Journal of Machine Learning Research (JMLR)*, v. 15, pp. 1929–1958, 2014.

[13] GIRSHICK, R., DONAHUE, J., DARRELL, T., et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 580–587, Columbus, Ohio, USA, Jun 2014.

[14] GIRSHICK, R. "Fast R-CNN". In: *The IEEE International Conference on Computer Vision (ICCV)*, pp. 1440–1448, Santiago, Chile, December 2015.

[15] REN, S., HE, K., GIRSHICK, R., et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Advances in Neural Information Processing Systems 28 (NIPS)*, pp. 91–99, Montreal, Canada, May 2015.

[16] DAI, J., LI, Y., HE, K., et al. "R-FCN: Object Detection via Region-based Fully Convolutional Networks". In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 379–387, Barcelona, Spain, December 2016.

[17] HE, K., ZHANG, X., REN, S., et al. "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 37, n. 9, pp. 1904–1916, 2015.

[18] REDMON, J., DIVVALA, S., GIRSHICK, R., et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, Las Vegas, Nevada, USA, June 2016.

[19] LONG, J., SHELHAMER, E., DARRELL, T. "Fully convolutional networks for semantic segmentation". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3431–3440, Boston, Massachusetts, USA, June 2015.

[20] YU, F., KOLTUN, V. "Multi-Scale Context Aggregation by Dilated Convolutions". In: *International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, May 2016.

[21] DONG, C., LOY, C. C., HE, K., et al. "Learning a Deep Convolutional Network for Image Super-Resolution". In: *European Conference on Computer Vision (ECCV)*, pp. 184–199, Zurich, Germany, September 2014. Springer.

[22] JOHNSON, J., ALAHI, A., FEI-FEI, L. "Perceptual Losses for Real-Time Style Transfer and Super-Resolution". In: *European Conference on Computer Vision (ECCV)*, pp. 694–711, Amsterdam, Netherlands, October 2016. Springer.

[23] HINTON, GEOFFREY, DENG, LI, YU, DONG, et al. "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups", *IEEE Signal Processing Magazine*, v. 29, n. 6, pp. 82–97, November 2012.

[24] XIONG, W., DROPPO, J., HUANG, X., et al. "Achieving Human Parity in Conversational Speech Recognition", *CoRR*, v. abs/1610.05256, 2016.

[25] VAN DEN OORD, A., DIELEMAN, S., ZEN, H., et al. "WaveNet: A Generative Model for Raw Audio", *CoRR*, v. abs/1609.03499, 2016.

[26] HADJERES, G., PACHET, F. "DeepBach: a Steerable Model for Bach chorales generation", *CoRR*, v. abs/1612.01010, 2016.

[27] WU, Y., SCHUSTER, M., CHEN, Z., et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation", *CoRR*, v. abs/1609.08144, 2016.

[28] BAHDANAU, D., CHO, K., BENGIO, Y. "Neural Machine Translation by Jointly Learning to Align and Translate". In: *International Conference on Learning Representations (ICLR)*, San Diego, California, USA, May 2015.

[29] SOCHER, R., PERELYGIN, A., WU, J. Y., et al. "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank". In: *Confer-*

ence on Empirical Methods in Natural Language Processing (EMNLP), pp. 1631–1642, Seatle, Washington, USA, October 2013.

[30] CHOLLET, F. “Keras”. `https://github.com/fchollet/keras`, 2015.

[31] ABADI, M., AGARWAL, A., BARHAM, P., et al. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems”. 2015. Disponível em: <`http://tensorflow.org/`>. Software available from tensorflow.org.

[32] CYBENKO, G. “Approximation by Superpositions of a Sigmoidal Function”, *Mathematics of Control, Signals, and Systems (MCSS)*, v. 2, n. 4, pp. 303–314, 1989.

[33] HORNIK, K. “Approximation Capabilities of Multilayer Feedforward Networks”, *Neural Networks*, v. 4, n. 2, pp. 251–257, 1991.

[34] LESHNO, M., LIN, V. Y., PINKUS, A., et al. “Multilayer Feedforward Networks with a Nonpolynomial Activation Function can Approximate any Function”, *Neural Networks*, v. 6, n. 6, pp. 861–867, 1993.

[35] HASTAD, J. “Almost Optimal Lower Bounds for Small Depth Circuits”. In: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pp. 6–20, Berkley, California, USA, May 1986. ACM.

[36] RUMELHART, D. E., HINTON, G. E., WILLIAMS, R. J. “Learning Representations by Back-propagating Errors”, *Nature*, v. 323, pp. 533–536, 1986.

[37] LECUN, Y. A., BOTTOU, L., ORR, G. B., et al. “Efficient Backprop”. In: *Neural Networks: Tricks of the Trade*, v. 1524, *Lecture Notes in Computer Science*, Springer, cap. 1, pp. 9–50, Berlin, Germany, 1998.

[38] FUKUSHIMA, K., MIYAKE, S. “Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Visual Pattern Recognition”. In: *Competition and Cooperation in Neural Nets*, v. 45, *Lecture Notes in Biomathematics (LNBM)*, Springer, cap. 18, pp. 267–285, Berlin, Germany, 1982.

[39] SPRINGENBERG, J. T., DOSOVITSKIY, A., BROX, T., et al. “Striving for Simplicity: The all Convolutional Net”, *arXiv preprint arXiv:1412.6806*, 2014.

[40] HE, K., ZHANG, X., REN, S., et al. “Deep Residual Learning for Image Recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, Las Vegas, Nevada, USA, June 2016.

[41] LIN, M., CHEN, Q., YAN, S. "Network in Network", *arXiv preprint arXiv:1312.4400*, 2013.

[42] SZEGEDY, C., LIU, W., JIA, Y., et al. "Going Deeper with Convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, Boston, Massachusetts, USA, June 2015.

[43] SMITH, L. N. "Cyclical Learning Rates for Training Neural Networks", *CoRR*, v. abs/1506.01186, 2015.

[44] SUTSKEVER, I., MARTENS, J., DAHL, G. E., et al. "On the Importance of Initialization and Momentum in Deep Learning". In: *The 30th International Conference on Machine-learning (ICML)*, pp. 1–9, Atlanta, Georgia, USA, June 2013.

[45] KINGMA, D., BA, J. "Adam: A Method for Stochastic Optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[46] DUCHI, J., HAZAN, E., SINGER, Y. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research (JMLR)*, v. 12, pp. 2121–2159, 2011.

[47] ZEILER, M. D. "ADADELTA: An Adaptive Learning Rate Method", *arXiv preprint arXiv:1212.5701*, 2012.

[48] TIELEMAN, T., HINTON, G. "Lecture 6.5–RMSProp: Divide the Gradient by a Running Average of its Recent Magnitude". COURSERA: Neural Networks for Machine Learning, 2012.

[49] GLOROT, X., BENGIO, Y. "Understanding the Difficulty of Training Deep Feedforward Neural Networks". In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 249–256, Sardinia, Italy, May 2010.

[50] HE, K., ZHANG, X., REN, S., et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification". In: *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, Santiago, Chile, December 2015.

[51] SAXE, A. M., MCCLELLAND, J. L., GANGULI, S. "Exact Solutions to the Nonlinear Dynamics of Learning in Deep Linear Neural Networks", *arXiv preprint arXiv:1312.6120*, 2013.

[52] HENDRYCKS, D., GIMPEL, K. "Generalizing and Improving Weight Initialization", *arXiv preprint arXiv:1607.02488*, 2016.

[53] MISHKIN, D., MATAS, J. "All you Need is a Good Init", *arXiv preprint arXiv:1511.06422*, 2015.

[54] IOFFE, S., SZEGEDY, C. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", *arXiv preprint arXiv:1502.03167*, 2015.

[55] HINTON, G. E., SRIVASTAVA, N., KRIZHEVSKY, A., et al. "Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors", *arXiv preprint arXiv:1207.0580*, 2012.

[56] CIREŞAN, D., MEIER, U., MASCI, J., et al. "Multi-Column Deep Neural Network for Traffic Sign Classification", *Neural Networks*, v. 32, pp. 333–338, 2012.

[57] KESKAR, N. S., MUDIGERE, D., NOCEDAL, J., et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *International Conference on Learning Representations (ICLR)*, Toulon, France, April 2017.

[58] NETZER, Y., WANG, T., COATES, A., et al. "Reading Digits in Natural Images with Unsupervised Feature Learning". In: *Deep Learning and Unsupervised Feature Learning Workshop*, pp. 1–9, Granada, Spain, December 2011.

[59] KRIZHEVSKY, A. "Learning Multiple Layers of Features from Tiny Images". 2009.

[60] TORRALBA, A., FERGUS, R., FREEMAN, W. T. "80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 30, n. 11, pp. 1958–1970, 2008.

[61] GOODFELLOW, I. J., WARDE-FARLEY, D., MIRZA, M., et al. "Maxout Networks". In: *The 30th International Conference on Machine-learning (ICML)*, pp. 1–9, Atlanta, Georgia, USA, June 2013.

[62] CHANG, J.-R., CHEN, Y.-S. "Batch-Normalized Maxout Network in Network", *arXiv preprint arXiv:1511.02583*, 2015.

[63] LEE, C.-Y., XIE, S., GALLAGHER, P. W., et al. "Deeply-Supervised Nets". In: *The 18th International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 1–5, San Diego, California, USA, May 2015.

[64] HUANG, G., LIU, Z., WEINBERGER, K. Q., et al. "Densely Connected Convolutional Networks", *arXiv preprint arXiv:1608.06993*, 2016.

[65] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., et al. "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, Las Vegas, Nevada, USA, June 2016.

[66] SZEGEDY, C., IOFFE, S., VANHOUCKE, V., et al. "Inception-v4, Inception-Resnet and the Impact of Residual Connections on Learning", *arXiv preprint arXiv:1602.07261*, 2016.

[67] HE, K., ZHANG, X., REN, S., et al. "Identity Mappings in Deep Residual Networks". In: *14th European Conference on Computer Vision (ECCV)*, pp. 630–645, Amsterdam, Netherlands, October 2016. Springer.

[68] DENG, J., DONG, W., SOCHER, R., et al. "Imagenet: A Large-Scale Hierarchical Image Database". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 248–255, Miami, Florida, USA, June 2009. IEEE.

[69] RUSSAKOVSKY, O., DENG, J., SU, H., et al. "ImageNet Large Scale Visual Recognition Challenge", *International Journal of Computer Vision (IJCV)*, v. 115, n. 3, pp. 211–252, 2015.

[70] SIMONYAN, K., ZISSERMAN, A. "Very Deep Convolutional Networks for Large-Scale Image Recognition", *arXiv preprint arXiv:1409.1556*, 2014.

[71] ZAGORUYKO, S., KOMODAKIS, N. "Wide Residual Networks". In: *Proceedings of the British Machine Vision Conference (BMVC)*, pp. 1–12, York, United Kingdom, September 2016.

[72] ABDI, M., NAHAVANDI, S. "Multi-Residual Networks", *arXiv preprint arXiv:1609.05672*, 2016.

[73] HUANG, G., SUN, Y., LIU, Z., et al. "Deep Networks with Stochastic Depth". In: *14th European Conference on Computer Vision (ECCV)*, pp. 646–661, Amsterdam, Netherlands, October 2016. Springer.

[74] SHEN, F., GAN, R., ZENG, G. "Weighted Residuals for Very Deep Networks". In: *3rd International Conference on Systems and Informatics (ICSAI)*, pp. 936–941, Shanghai, China, November 2016. IEEE.

[75] ZHANG, K., SUN, M., HAN, T. X., et al. “Residual Networks of Residual Networks: Multilevel Residual Networks”, *arXiv preprint arXiv:1608.02908*, 2016.

[76] TARG, S., ALMEIDA, D., LYMAN, K. “Resnet in Resnet: Generalizing Residual Architectures”, *arXiv preprint arXiv:1603.08029*, 2016.

[77] VEIT, A., WILBER, M. J., BELONGIE, S. “Residual Networks Behave Like Ensembles of Relatively Shallow Networks”. In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 550–558, Barcelona, Spain, December 2016.

[78] GREFF, K., SRIVASTAVA, R. K., SCHMIDHUBER, J. “Highway and Residual Networks Learn Unrolled Iterative Estimation”, *arXiv preprint arXiv:1612.07771*, 2016.

[79] JÉGOU, S., DROZDZAL, M., VAZQUEZ, D., et al. “The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation”, *CoRR*, v. abs/1611.09326, 2016.

[80] ZEILER, M. D., FERGUS, R. “Visualizing and Understanding Convolutional Networks”. In: *European Conference on Computer Vision (ECCV)*, pp. 818–833, Zurich, Germany, September 2014. Springer.

[81] SIMONYAN, K., VEDALDI, A., ZISSERMAN, A. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”. In: *International Conference on Learning Representations (ICLR)*, pp. 1–8, San Diego, California, USA, April 2014. CBLS.

[82] YOSINSKI, J., CLUNE, J., NGUYEN, A., et al. “Understanding Neural Networks through Deep Visualization”, *arXiv preprint arXiv:1506.06579*, 2015.

[83] MAHENDRAN, A., VEDALDI, A. “Understanding Deep Image Representations by Inverting them”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5188–5196, Boston, Massachusetts, USA, June 2015.

[84] RAZAVIAN, A. S., AZIZPOUR, H., SULLIVAN, J., et al. “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pp. 806–813, Columbus, Ohio, USA, June 2014.

[85] DONAHUE, J., JIA, Y., VINYALS, O., et al. “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”. In: *The 31st International Conference on Machine Learning (ICML)*, pp. 647–655, Beijing, China, June 2014.

[86] YOSINSKI, J., CLUNE, J., BENGIO, Y., et al. “How transferable are features in deep neural networks?” In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 3320–3328, Montreal, Canada, December 2014.

[87] OQUAB, M., BOTTOU, L., LAPTEV, I., et al. “Learning and Transferring Mid-level Image Representations Using Convolutional Neural Networks”. In: *The IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1717–1724, Columbus, Ohio, USA, June 2014.

[88] SERMANET, P., EIGEN, D., ZHANG, X., et al. “OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks”. In: *International Conference on Learning Representations (ICLR)*, pp. 1–16, San Diego, California, USA, April 2014. CBLS.

[89] ZHANG, R., ISOLA, P., EFROS, A. A. “Colorful Image Colorization”. In: *14th European Conference on Computer Vision (ECCV)*, pp. 649–666, Amsterdam, Netherlands, October 2016. Springer.

[90] LARSSON, G., MAIRE, M., SHAKHNAROVICH, G. “Learning Representations for Automatic Colorization”. In: *14th European Conference on Computer Vision (ECCV)*, pp. 577–593, Amsterdam, Netherlands, October 2016. Springer.

[91] IIZUKA, S., SIMO-SERRA, E., ISHIKAWA, H. “Let there be Color!: Joint End-to-end Learning of Global and Local Image Priors for Automatic Image Colorization with Simultaneous Classification”, *ACM Transactions on Graphics (Proc. of SIGGRAPH 2016)*, v. 35, n. 4, pp. 1–11, 2016.

[92] ULYANOV, D., LEBEDEV, V., VEDALDI, A., et al. “Texture networks: Feed-forward synthesis of textures and stylized images”. In: *International Conference on Machine Learning (ICML)*, pp. 1–16, New York, New York, USA, June 2016.

[93] ZHOU, B., KHOSLA, A., LAPEDRIZA, A., et al. “Object Detectors Emerge in Deep Scene CNNs”. In: *International Conference on Learning Representations (ICLR)*, San Diego, California, USA, May 2015.

[94] OQUAB, M., BOTTOU, L., LAPTEV, I., et al. "Is Object Localization for Free? - Weakly-Supervised Learning With Convolutional Neural Networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 685–694, Boston, Massachusetts, USA, June 2015.

[95] ZHOU, B., KHOSLA, A., LAPEDRIZA, A., et al. "Learning Deep Features for Discriminative Localization". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2921–2929, Las Vegas, Nevada, USA, June 2016.

[96] SELVARAJU, R. R., DAS, A., VEDANTAM, R., et al. "Grad-CAM: Why did you say that? Visual Explanations from Deep Networks via Gradient-based Localization", *CoRR*, v. abs/1610.02391, 2016.

[97] GATYS, L. A., ECKER, A. S., BETHGE, M. "Image Style Transfer Using Convolutional Neural Networks". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2414–2423, Las Vegas, Nevada, USA, Jun 2016.

[98] GATYS, L. A., ECKER, A. S., BETHGE, M. "Texture Synthesis Using Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 28 (NIPS)*, pp. 1–9, Montreal, Canada, May 2015.

[99] PORTILLA, J., SIMONCELLI, E. P. "A Parametric Texture Model Based on Joint Statistics of Complex Wavelet Coefficients", *International Journal of Computer Vision*, v. 40, n. 1, pp. 49–70, 2000.

[100] ZHU, C., BYRD, R. H., LU, P., et al. "Algorithm 778: L-BFGS-B: Fortran Subroutines for Large-Scale Bound-Constrained Optimization", *ACM Transactions on Mathematical Software (TOMS)*, v. 23, n. 4, pp. 550–560, December 1997.

[101] ODENA, A., DUMOULIN, V., OLAH, C. "Deconvolution and Checkerboard Artifacts", *Distill*, 2016. http://distill.pub/2016/deconv-checkerboard.

[102] DUMOULIN, V., SHLENS, J., KUDLUR, M. "A Learned Representation For Artistic Style", *CoRR*, v. abs/1610.07629, 2016.

[103] ULYANOV, D., VEDALDI, A., LEMPITSKY, V. "Instance Normalization: The Missing Ingredient for Fast Stylization", *CoRR*, v. abs/1607.08022, 2016.

[104] LIN, T.-Y., MAIRE, M., BELONGIE, S., et al. “Microsoft COCO: Common Objects in Context”. In: *European Conference on Computer Vision (ECCV)*, pp. 740–755, Zurich, Germany, September 2014. Springer.

[105] KINGMA, D. P., WELLING, M. “Auto-Encoding Variational Bayes”. In: *International Conference on Learning Representations (ICLR)*, pp. 1–14, San Diego, California, USA, April 2014. CBLS.

[106] REZENDE, D. J., MOHAMED, S., WIERSTRA, D. “Stochastic Backpropagation and Approximate Inference in Deep Generative Models”. In: *The 31st International Conference on Machine Learning (ICML)*, pp. 1278–1286, Beijing, China, June 2014. IMLS.

[107] DOERSCH, C. “Tutorial on Variational Autoencoders”, *CoRR*, v. abs/1606.05908, 2016.

[108] GOODFELLOW, I., POUGET-ABADIE, J., MIRZA, M., et al. “Generative Adversarial Nets”. In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 2672–2680, Montreal, Canada, December 2014.

[109] RADFORD, A., METZ, L., CHINTALA, S. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”. In: *International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, May 2016.

[110] SALIMANS, T., GOODFELLOW, I., ZAREMBA, W., et al. “Improved Techniques for Training GANs”. In: *Advances in Neural Information Processing Systems (NIPS)*, pp. 2234–2242, Barcelona, Spain, December 2016.

[111] REED, S., AKATA, Z., YAN, X., et al. “Generative Adversarial Text to Image Synthesis”. In: *The 33rd International Conference on Machine Learning (ICML)*, pp. 1060–1069, New York, New York, USA, June 2014.

[112] SUTTON, R. S., BARTO, A. G. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[113] SILVER, D., HUANG, A., MADDISON, C. J., et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search”, *Nature*, v. 529, n. 7587, pp. 484–489, 2016.

[114] MNIH, V., KAVUKCUOGLU, K., SILVER, D., et al. “Human-Level Control through Deep Reinforcement Learning”, *Nature*, v. 518, n. 7540, pp. 529–533, 2015.

[115] LILLICRAP, T. P., HUNT, J. J., PRITZEL, A., et al. "Continuous Control with Deep Reinforcement Learning". In: *International Conference on Learning Representations (ICLR)*, San Juan, Puerto Rico, May 2016.