# What Do We Understand About Convolutional Networks?

Isma Hadji and Richard P. Wildes

Department of Electrical Engineering and Computer Science

York University

Toronto, Ontario

Canada

# Chapter 1

# Introduction

## 1.1 Motivation

Over the past few years major computer vision research efforts have focused on convolutional neural networks, commonly referred to as ConvNets or CNNs. These efforts have resulted in new state-of-the-art performance on a wide range of classification (e.g [64, 88, 139]) and regression (e.g [36, 97, 159]) tasks. In contrast, while the history of such approaches can be traced back a number of years (e.g [49, 91]), theoretical understanding of how these systems achieve their outstanding results lags. In fact, currently many contributions in the computer vision field use ConvNets as a black box that works while having a very vague idea for why it works, which is very unsatisfactory from a scientific point of view. In particular, there are two main complementary concerns: **(1)** For learned aspects (e.g convolutional kernels), exactly what has been learned? **(2)** For architecture design aspects (e.g number of layers, number of kernels/layer, pooling strategy, choice of nonlinearity), why are some choices better than others? The answers to these questions not only will improve the scientific understanding of ConvNets, but also increase their practical applicability.

Moreover, current realizations of ConvNets require massive amounts of data for training [84, 88, 91] and design decisions made greatly impact performance [23, 77]. Deeper theoretical understanding should lessen dependence on data-driven design. While empirical studies have investigated the operation of implemented networks, to

date, their results largely have been limited to visualizations of internal processing to understand what is happening at the different layers of a ConvNet [104, 133, 154].

## 1.2 Objective

In response to the above noted state of affairs, this document will review the most prominent proposals using multilayer convolutional architectures. Importantly, the various components of a typical convolutional network will be discussed through a review of different approaches that base their design decisions on biological findings and/or sound theoretical bases. In addition, the different attempts at understanding ConvNets via visualizations and empirical studies will be reviewed. The ultimate goal is to shed light on the role of each layer of processing involved in a ConvNet architecture, distill what we currently understand about ConvNets and highlight critical open problems.

## 1.3 Outline of report

This report is structured as follows: The present chapter has motivated the need for a review of our understanding of convolutional networks. Chapter 2 will describe various multilayer networks and present the most successful architectures used in computer vision applications. Chapter 3 will more specifically focus on each one of the building blocks of typical convolutional networks and discuss the design of the different components from both biological and theoretical perspectives. Finally, chapter 4 will describe the current trends in ConvNet design and efforts towards ConvNet understanding and highlight some critical outstanding shortcomings that remain.

# Chapter 2

# Multilayer Networks

This chapter gives a succinct overview of the most prominent multilayer architectures used in computer vision, in general. Notably, while this chapter covers the most important contributions in the literature, it will not to provide a comprehensive review of such architectures, as such reviews are available elsewhere (*e.g.* [17, 56, 90]). Instead, the purpose of this chapter is to set the stage for the remainder of the document and its detailed presentation and discussion of what currently is understood about convolutional networks applied to visual information processing.

## 2.1 Multilayer architectures

Prior to the recent success of deep learning-based networks, state-of-the-art computer vision systems for recognition relied on two separate but complementary steps. First, the input data is transformed via a set of hand designed operations (*e.g.* convolutions with a basis set, local or global encoding methods) to a suitable form. The transformations that the input incurs usually entail finding a compact and/or abstract representation of the input data, while injecting several invariances depending on the task at hand. The goal of this transformation is to change the data in a way that makes it more amenable to being readily separated by a classifier. Second, the transformed data is used to train some sort of classifier (*e.g.* Support Vector Machines) to recognize the content of the input signal. The performance of any classifier used is, usually, heavily affected by the used transformations.

Multilayer architectures with learning bring about a different outlook on the problem by proposing to learn, not only the classifier, but also learn the required transformation operations directly from the data. This form of learning is commonly referred to as representation learning [7, 90], which when used in the context of deep multilayer architectures is called deep learning.

Multilayer architectures can be defined as computational models that allow for extracting useful information from the input data  multiple levels of abstraction. Generally, multilayer architectures are designed to amplify important aspects of the input at higher layers, while becoming more and more robust to less significant variations. Most multilayer architectures stack simple building block modules with alternating linear and nonlinear functions. Over the years, a plethora of various multilayer architectures were proposed and this section will cover the most prominent such architectures adopted for computer vision applications. In particular, artificial neural network architectures will be the focus due to their prominence. For the sake of succinctness, such networks will be referred to more simply as neural networks in the following.

### 2.1.1 Neural networks

A typical neural network architecture is made of an input layer, $\boldsymbol{x}$, an output layer, $\boldsymbol{y}$, and a stack of multiple hidden layers, $\boldsymbol{h}$, where each layer consists of multiple cells or units, as depicted in Figure 2.1. Usually, each hidden unit, $h_j$, receives input from all units at the previous layer and is defined as a weighted combination of the inputs followed by a nonlinearity according to

$$h_j = F(b_j + \sum_i w_{ij} x_i) \tag{2.1}$$

where, $w_{ij}$, are the weights controlling the strength of the connections between the input units and the hidden unit, $b_j$ is a small bias of the hidden unit and $F(.)$ is some saturating nonlinearity such as the sigmoid.

Deep neural networks can be seen as a modern day instantiation of Rosenblatt's perceptron [122] and multilayer perceptron [123]. Although, neural network models

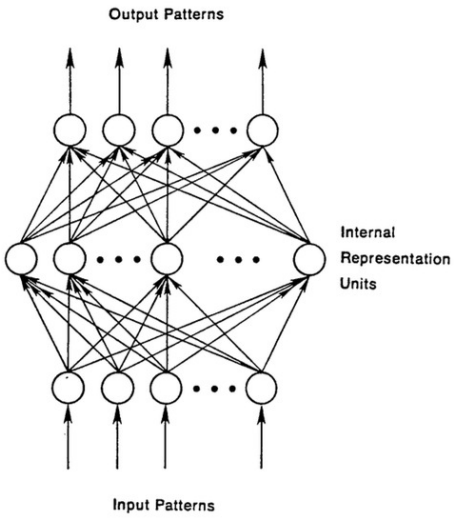


Figure 2.1: Illustration of a typical Neural Network architecture. Figure reproduced from [17].

have been around for many years (*i.e.* since the 1960's) they were not heavily used until more recently. There were a number of reasons for this delay. Initial negative results showing the inability of the perceptron to model simple operations like XOR, hindered further investigation of perceptrons for a while until their generalizations to many layers [106]. Also, lack of an appropriate training algorithm slowed progress until the popularization of the backpropagation algorithm [125]. However, the bigger roadblock that hampered the progress of multilayer neural networks is the fact that they rely on a very large number of parameters, which in turn implies the need for large amounts of training data and computational resources to support learning of the parameters.

A major contribution that allowed for a big leap of progress in the field of deep neural networks is layerwise unsupervised pretraining, using Restricted Boltzman Machine (RBM) [68]. Restricted Boltzman Machines can be seen as two layer neural networks where, in their restricted form, only feedforward connections are allowed. In the context of image recognition, the unsupervised learning method used to train RBMs can be summarized in three steps. First, for each pixel, $x_i$, and starting with a set of random weights, $w_{ij}$, and biases, $b_j$, the hidden state, $h_j$, of each unit is set to 1 with probability, $p_j$. The probability is defined as

$$p_j = \sigma(b_j + \sum_i x_i w_{ij}) \tag{2.2}$$

where, $\sigma(y) = 1/(1+exp(-y))$. Second, once all hidden states have been set stochastically based on equation 2.2, an attempt to reconstruct the image is performed by setting each pixel, $x_i$, to 1 with probability $p_i = \sigma(b_i + \sum_j h_j w_{ij})$. Third, the hidden units are corrected by updating the weights and biases based on the reconstruction error given by

$$\Delta w_{ij} = \alpha(\langle x_i h_j \rangle_{input} - \langle x_i h_j \rangle_{reconstruction}) \tag{2.3}$$

where $\alpha$ is a learning rate and $\langle x_i h_j \rangle$ is the number of times pixel $x_i$ and the hidden unit $h_j$ are on together. The entire process is repeated $N$ times or until the error drops bellow a pre-set threshold, $\tau$. After one layer is trained its outputs are used as an input to the next layer in the hierarchy, which is in turn trained following the same procedure. Usually, after all the network's layers are pretrained, they are further finetuned with labeled data via error back propagation using gradient descent [68]. Using this layerwise unsupervised pretraining allows for training deep neural networks without requiring large amounts of labeled data because unsupervised RBM pretraining provides a way for an empirically useful initialization of the various network parameters.

Neural networks relying on stacked RBMs were first successfully deployed as a method for dimensionality reduction with an application to face recognition [69], where they were used as a type of auto-encoder. Loosely speaking, auto-encoders can be defined as multilayer neural networks that are made of two main parts: First, an encoder transforms the input data to a feature vector; second, a decoder maps the generated feature vector back to the input space; see, Figure 2.2. The parameters of the auto-encoder are learned by minimizing a reconstruction error between the input and it's reconstructed version.

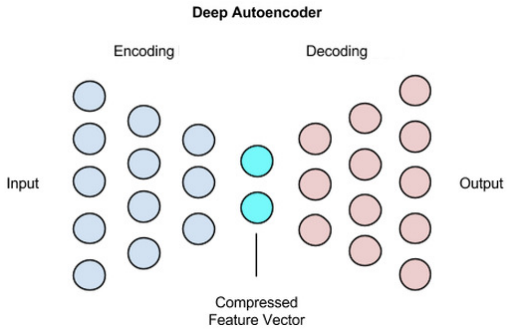


Figure 2.2: Structure of a typical Auto-Encoder Network. Figure reproduced from [17].

Beyond RBM based auto-encoders, several types of auto-encoders were later proposed. Each auto-encoder introduced a different regularization method that prevents the network from learning trivial solutions even while enforcing different invariances. Examples include Sparse Auto-Encoders (SAE) [8], Denoising Auto-Encoders (DAE) [141, 142] and Contractive Auto-Encoders (CAE) [118]. Sparse Auto-Encoders [8] allow the intermediate representation's size (*i.e.* as generated by the encoder part) to be larger than the input's size while enforcing sparsity by penalizing negative outputs. In contrast, Denoising Auto-Encoders [141, 142] alter the objective of the reconstruction itself by trying to reconstruct a clean input from an artificially corrupted version, with the goal being to learn a robust representation. Similarly, Contractive Auto-Encoders [118] build on denoising auto-encoders by further penalizing the units that are most sensitive to the injected noise. More detailed reviews of various types of auto-encoders can be found elsewhere [7].

### 2.1.2 Recurrent neural networks

When considering tasks that rely on sequential inputs, one of the most successful multilayer architectures is the Recurrent Neural Network (RNN) [9]. RNNs, illustrated in Figure 2.3, can be seen as a special type of neural network where each hidden unit takes input from the the data it observes at the current time step as well as its state at a previous time step. The output of an RNN is defined as

$$h_t = \sigma(w_i x_t + u_i h_{t-1}) \tag{2.4}$$

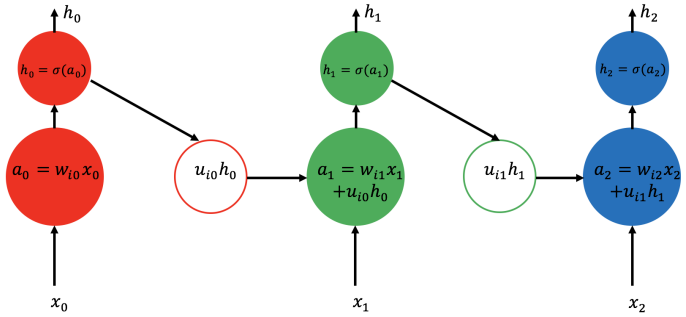


Figure 2.3: Illustration of the operations of a standard Recurrent Neural Network. Each RNN unit takes new input at the current time frame, $x_t$, and from a previous time step, $h_{t-1}$ and the new output of the unit is calculated according to (2.4) and can be fed to another layer of processing in a multilayer RNN.

where $\sigma$ is some nonlinear squashing function and $w_i$ and $u_i$ are the network parameters that control the relative importance of the present and past information.

Although RNNs are seemingly powerful architectures, one of their major problems is their limited ability to model long term dependencies. This limitation is attributed to training difficulties due to exploding or vanishing gradient that can occur when propagating the error back through multiple time steps [9]. In particular, during training the back propagated gradient is multiplied with the network's weights from the current time step all the way back to the initial time step. Therefore, because of this multiplicative accumulation, the weights can have a non-trivial effect on the propagated gradient. If weights are small the gradient vanishes, whereas larger weights lead to a gradient that explodes. To correct for this difficulty, Long Short Term Memories (LSTM) were introduced [70].

LSTMs are recurrent networks that are further equipped with a storage or memory component, illustrated in Figure 2.4, that accumulates information over time. An LSTM's memory cell is gated such that it allows information to be read from it or written to it. Notably, LSTMs also contain a forget gate that allows the network to erase information when it is not needed anymore. LSTMs are controlled by three different gates (the input gate, $i_t$, the forget gate, $f_t$, and the output gate, $o_t$), as well as the memory cell state, $c_t$. The input gate is controlled by the current input, $x_t$, and the previous state, $h_{t-1}$, and it is defined as

$$i_t = \sigma(w_i x_t + u_i h_{t-1} + b_i), \tag{2.5}$$

where, $w_i$, $u_i$, $b_i$ represent the weights and bias controlling the connections to the input gate and $\sigma$ is usually a sigmoid function. The forget gate is similarly defined as

$$f_t = \sigma(w_f x_t + u_f h_{t-1} + b_f), \tag{2.6}$$

and it is controlled by its corresponding weights and bias, $w_f$, $u_f$, $b_f$. Arguably, the most important aspect of an LSTM is that it copes with the challenge of vanishing and exploding gradients. This ability is achieved through additive combination of the forget and input gate states in determining the memory cell's state, which, in turn, controls whether information is passed on to another cell via the output gate. Specifically, the cell state is computed in two steps. First, a candidate cell state is estimated according to

$$g_t = \phi(w_c x_t + u_c h_{t-1} + b_c), \tag{2.7}$$

where $\phi$ is usually a hyperbolic tangent. Second, the final cell state is finally controlled by the current estimated cell state, $g_t$, and the previous cell state, $c_{t-1}$, modulated by the input and forget gate according to

$$c_t = i_t g_t + f_t c_{t-1}. \tag{2.8}$$

Finally, using the cell's state and the current and previous inputs, the value of the output gate and the output of the LSTM cell are estimated according to

$$o_t = \sigma(w_o x_t + u_o h_{t-1} + b_o), \tag{2.9}$$

where

$$h_t = \phi(c_t)\, o_t. \tag{2.10}$$